EFFECTIVE GROUPING FOR ENERGY AND PERFORMANCE: CONSTRUCTION OF ADAPTIVE, SUSTAINABLE, AND MAINTAINABLE DATA STORAGE

by

David S. Essary

BS, Computer Science, University of Pittsburgh, 2003

BS, Mathematics, University of Pittsburgh, 2003

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

David S. Essary

It was defended on

November 22, 2010

and approved by

Ahmed Amer, Computer Engineering, Santa Clara University

Daniel Mossé, Computer Science, University of Pittsburgh

Panos Chrysanthis, Computer Science, University of Pittsburgh

Kirk Pruhs, Computer Science, University of Pittsburgh

Dissertation Advisors: Ahmed Amer, Computer Engineering, Santa Clara University,

Daniel Mossé, Computer Science, University of Pittsburgh

ABSTRACT

EFFECTIVE GROUPING FOR ENERGY AND PERFORMANCE: CONSTRUCTION OF ADAPTIVE, SUSTAINABLE, AND MAINTAINABLE DATA STORAGE

David S. Essary, PhD

University of Pittsburgh, 2011

The performance gap between processors and storage systems has been increasingly critical over the years. Yet the performance disparity remains, and further, storage energy consumption is rapidly becoming a new critical problem. While smarter caching and predictive techniques do much to alleviate this disparity, the problem persists, and data storage remains a growing contributor to latency and energy consumption.

Attempts have been made at data layout maintenance, or intelligent physical placement of data, yet in practice, basic heuristics remain predominant. Problems that early studies sought to solve via layout strategies were proven to be NP-Hard, and data layout maintenance today remains more art than science. With unknown potential and a domain inherently full of uncertainty, layout maintenance persists as an area largely untapped by modern systems. But uncertainty in workloads does not imply randomness; access patterns have exhibited repeatable, stable behavior. Predictive information can be gathered, analyzed, and exploited to improve data layouts. Our goal is a dynamic, robust, sustainable predictive engine, aimed at improving existing layouts by replicating data at the storage device level.

We present a comprehensive discussion of the design and construction of such a predictive engine, including workload evaluation, where we present and evaluate classical workloads as well as our own highly detailed traces collected over an extended period. We demonstrate significant gains through an initial static grouping mechanism, and compare against an optimal grouping method of our own construction, and further show significant improvement over competing techniques. We

also explore and illustrate the challenges faced when moving from static to dynamic (*i.e.* online) grouping, and provide motivation and solutions for addressing these challenges. These challenges include metadata storage, appropriate predictive collocation, online performance, and physical placement. We reduced the metadata needed by several orders of magnitude, reducing the required volume from more than 14% of total storage down to less than $\frac{1}{2}$%. We also demonstrate how our collocation strategies outperform competing techniques. Finally, we present our complete model and evaluate a prototype implementation against real hardware. This model was demonstrated to be capable of reducing device-level accesses by up to 65%.

Keywords: computer systems, collocation, data management, file systems, grouping, metadata, modeling and prediction, operating systems, performance, power, secondary storage.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF EQUATIONS

PREFACE

I will forever be in debt, and eternally grateful, to my committee, friends, and colleagues. Words do not give proper praise, but here, in harsh brevity, I will have my tilt at the windmill nonetheless.

To he that is most deserving, let first praises fly. Dr. Ahmed Amer has, in every capacity, exceeded all expectation and calling; for his counsel, I am a better man, in science and in life. He is my advisor, my guide, my teacher, my colleague, and my very good friend. All students should be so lucky. Patience, wisdom, guidance, support, and respect are his trademarks, along with an impeccable sense of humor. He taught me that work, science, and joy are not mutually exclusive terms. For this lesson, I am happy in my toil.

To Dr. Daniel Mossé, I am also particularly indebted. First as a teacher, then as an advisor, and finally as department chair, his sage advice has been invaluable. With simple, direct words, and a few reality checks, he has helped, with gentle nudges, keep a willful student on a disciplined path. To Dr. Darrell Long, I would express my sincere gratitude for valuable lessons in elegance and systems expertise. I am very fortunate to have known such an authority of the storage system community. I would also thank Dr. Kirk Pruhs, one of my first professors. Those early lessons went further than I could have anticipated; they are among my first steps towards higher understanding. I would also thank Dr. Panos Chrysanthis, the very first person to welcome me to graduate study. I've yet to meet a finer person, teacher, and colleague. For his conversations, his guidance, and his wit, I shall always be grateful. To Dr. Alex Jones, with whom I worked but briefly, I am genuinely appreciative for early electrical engineering help and discussion.

I also wish to thank Dr. Milos Hauskrecht, as a fine teacher and excellent scientist, for helping to set in motion my admission into graduate study. And to the fine staff and excellent faculty of the Department of Computer Science at the University of Pittsburgh, and to my friends and fellow students, both past and present, thank you for the years of support. I will cherish our time together,

now and always.

In triumph, and in need, we look most often to those closest, our families. Without them, our proudest moments could not be; for them, our dreams are realized. Proper thanks is, inevitably, impossible. All the same, to Dad, Mom, Bill, Kevin, and Tom, for your unremitting support, thank you.

Finally, and most eagerly, I wish to thank my wife, Sarah. Every day, in ways great and small, she remains my greatest companion, and truest friend. She is my champion, my shelter in the storm, my light in the dark; she is my now, my always, and my everything. In this, I remain blessed beyond worth.

This dissertation is dedicated to my father; my first teacher, my greatest role model, and my hero. For pushing me when I lacked the strength and will to push myself, for putting trust in your wayward son, and for moments of praise that mean more to me than could be put to words...

Thanks, Dad.

1.0 INTRODUCTION

The disparity between the throughput of processors and underlying storage devices has presented a challenge to systems designers since the inception of the modern CPU. Incremental achievements over the past sixty years have greatly alleviated the strain of these devices. Yet the latency observed by end-users due to the limitations of storage system hardware remains significant. Additionally, data storage represents an ever-growing portion of energy consumption, and energy is an increasingly critical problem. As energy becomes a more valuable resource, and storage demands grow along with their corresponding energy footprint, every Joule becomes more precious. We address the performance disparity and increasing energy costs of storage systems through the predictive grouping of data based on the dynamic analysis of access workloads.

Recent trends indicate that latency and energy concerns of storage devices are not only justified, but will continue becoming more and more critical. In 2002, approximately 5 exabytes of data was generated and stored on magnetic media [85]. In 2007, that amount of data was generated every 6 and a half days [42]; today, it can take as little as two days [120]. In fact, the year 2007 marked the first time that the amount of data produced and/or replicated outpaced available storage [42]. By 2011, it is estimated that nearly half of all data generated will have no permanent storage location. Additionally, the energy cost of moving this information flood is becoming astronomical. In 2006, a study by the EPA estimated that U.S. data centers and servers consumed 61 billion kilowatt-hours, at an estimated $4.5 billion [3]. A simple carbon footprint estimate, using the CDIAC's general estimate of 2.3 pounds of CO_2 per kilowatt-hour [38], yields an estimated 140.3 billion pounds of CO_2, more than a third the carbon footprint of all U.S. aircraft in 2003 [2].

Thankfully, the driving workloads that storage systems must satisfy are very often far from random. They represent the needs of applications and systems users; their behavior may not be deterministic, but it is rarely arbitrary. The resulting predictable nature of these workloads has

culminated some of the most important technological breakthroughs of the personal computer era. Caching and memory management are among the earliest achievements. These strategies look to the recent past as an approximation of the near future. More recently, prefetching strategies have been explored in great detail, looking at prior events, both recent as well as venerable, in an effort to glean established patterns with which to predict the future. And yet, the challenge of keeping pace with processors persists.

To understand these challenges, it is essential to understand the nature of the latency delays and the energy costs of data retrieval. Latency delays are of two general varieties; those that are inevitable, and those that are avoidable, or at least maskable. The first flavor is best exemplified by a streaming live video. Data is transferred, decoded, and consumed, never to be used again. Even with optimized hardware and the best compression methods, there will be unavoidable delays. However, with storage systems, this is far from the norm. Most often, data items are accessed repeatedly within a relatively small window of time. Without this trend, caching would have little effect; it is perhaps the most powerful trend that we can exploit. Yet caching alone is not the silver bullet. Data must first be *accessed* before it can be *re-accessed*; thus, we will pay an initial cost to bring an item into the cache. Prefetch caching seeks to alleviate this initial cost by masking the latency; a predictive request, generated *before* the data is actually needed, can bring data into the cache early, so that it is immediately available upon its actual request. But the request must have sufficient lead-time, and must contend with other queued requests, in order to have significant impact. This inexorably ties prefetching to the data path, and dictates a narrow window of opportunity for action. Further, predictions must be very accurate, as mistakes can be very costly. Many predicting methods require keeping extra metadata in the form of records of accesses to improve their accuracy, but storing these required records can be quite costly.

On yet another front, predictions used for traditional prefetching of data must not only be accurate, but must be timely. As such, predictive and prefetching caches have tended to go to great lengths to increase the lead-time of the predictions they offer, desperate to make predictions further into the future. This is a problem for any approach that attempts to act on a prediction by immediate prefetching. Such techniques are caught in a catch-22, where they expend resources to make longer-reaching accurate predictions. And yet the further ahead a prediction is attempted, the more likely for it to be inaccurate. This problem is addressed by using prediction as the basis for

a grouping strategy, or a means by which the layout can be improved by dynamically collocating data chunks on the fly at the device level. This allows prefetching to become implicit and removed from the demand path of user data requests. When combined with a data layout strategy, such groupings can potentially reduce latencies and energy consumption simultaneously.

Data layout strategies seek to make predictions more inherent by reducing the cost of accessing items, assuming one must go all the way down to the device level to retrieve them. In this way, one can cleverly arrange data so that items that will be needed or prefetched are easier to access. Early strategies tried to use frequency-based positioning, but ignored the interdependence of data access. Prediction models from prefetch caching can be applied, but once again a common problem is the required space for access metadata.

This common challenge, tracking predictive metadata, continues to be a daunting task, and is exacerbated by increasing storage demands. On mobile devices, the problem is compounded by a need to make the best use of available resources. Additionally, larger metadata translates to further strain on the storage system for updates and retrieval, both of which must be streamlined operations in order to have little or maskable impact on observed latencies. These problems exist for strategies operating at the file or object granularity, but can be crippling for block-level strategies. This reason alone can be enough for developers to operate at the abstract but more human-oriented file level, rather than at the block level, which is more native to storage subsystems. However, it is desirable to operate at the block level, as that results in a solution applicable for any storage system, not just object stores. Of course, one could restrict the amount of metadata, but arbitrary limits will often only benefit “hot” data, or those blocks within the current working set; arguably, this set is less in need of pattern discovery due to the effectiveness of even basic caching schemes. This inevitably precludes the opportunity to discover longer-term patterns across less intensely active regions. Attempts to manage storage devices to reduce energy have typically been at the cost of performance, however, successfully collocating data has the potential of realizing the ideal of simultaneously reducing energy and latency.

1.1 CONTRIBUTIONS

This dissertation presents solutions to the problems faced by data layout strategies, and develops a practical data collocation solution aimed at reducing energy and latency. We tackle the spatial requirements of metadata that could be used *both* for predictive caching as well as layout management. In particular, we address the problem of tracking information at the block level, where the state-space explosion of metadata has the highest burden. Further, we present applications of predictive models to group related data chunks (for eventual collocation at the device level). We explore the difficulties and challenges of moving from static grouping, such as a one-time defragmentation, to dynamic grouping, where the system automatically maintains a layout strategy on the fly. We also present preliminary bounding efforts on the possible benefits from these strategies. The effectiveness of our methods is examined using workload trace sets, including established traces used in prior and related work, as well as our own newer workload trace sets. These traces represent a variety of systems and configurations, allowing for workloads indicative of newer software and hardware while providing the ability to compare our work with prior and related efforts. Our evaluations are done via simulation. This affords us the opportunity to test more parameters and more workloads, supplying generalized results that are not tied to any particular system configuration. However, in order to validate these results, we include the design and analysis of our prototype hardware test bed system using accurate power measurements on a prototype system using a DAQ (Data AcQuisition) system to measure voltage for the mechanical components of the disk at 20,000 samples per second.

2.0 RELATED WORK

Placing related items within close proximity to one another has been a traditional standard in storage systems as a means of reducing latency. This data grouping goal can be addressed in a number of ways, and as such, several related areas of research influence our work. Of particular impact are studies on file access prediction, including access and workload modeling, prefetching, and metrics used to evaluate and compare prediction and prefetching policies. Caching is perhaps the oldest and certainly among the most successful and popular data placement strategies. We explore both traditional and recent caching research. Additionally, we include background discussion on I/O workload manipulation, as these represent some of the earliest device-level strategies for reducing power and enhancing performance.

The most closely affiliated research areas to our own involve data layout maintenance. A number of file systems attempt some amount of replication and migration of data, and are discussed along with these layout maintenance strategies. Recently, power consumption goals have become more imperative to designers. Therefore, we will conclude the chapter by considering recent and classic strategies for reducing the overall power consumption of storage systems. As many of our initial results are based on simulated systems, we detail a number of strategies used to model power consumption and disk simulation.

2.1 PREDICTION, PREFETCHING, AND CACHING

A study on graph-based access predictors was first presented by Griffioen and Appleton [49]. These predictors were used to provide sufficient lead-time to render the prediction useful for prefetching as well as managing access patterns spanning multiple applications. The use of the last succes-

sor model for file prediction, and more elaborate techniques based on pattern matching, were first presented by Lei and Duchamp [77]. Similar work has been done researching a last successor predictor, finite multi-order context modeling (*FMOC*) models from branch prediction methods, and a partitioned context model (*PCM*) [71]. While a last successor strategy predicted with surprising accuracy, there tends to be enough noise in an access stream to confuse it [6]. A more stable predictor, *Noah*, is presented that removes this noise by predicting only if a stability condition is satisfied.

Previous work has also shown that comparing two different predictors is non-trivial. To aid in this dilemma, three measures of prediction accuracy were developed; general accuracy and specific accuracy [5,8] and effective-miss-ratio [124]. General and specific accuracy were used to compare *Noah* with last successor and first successor [8]. It is noted that *Noah* suffers from non-decreasing general accuracy for high stability parameters. A new predictor, *Recent Popularity*, is shown to solve this problem. It is also noted that *Recent Popularity* adapts quicker with changing workloads than *Noah* [8]. To benefit from this robustness and adaptability, our techniques use variants on *Recent Popularity* for gathering data for prediction.

Further advancements in predictive caching has taken various forms and addressed various problems. Advances in caching strategies include using multiple experts in cache management [10], power aware storage cache management [129], and self-tuning cache replacement policies [87,88]. Work has also been done on augmenting caches with prefetching capabilities [70] and the effects caches have when placed back to back [9]. Prefetching and predictive caching have also been used as a means of overcoming latency in web proxies [31, 73, 93] as well as in object prefetching for internet applications [82, 98]. Similar work on the aggregating cache [7] differs from related work on predictive prefetching systems, but uses analogous structures to Griffioen and Appleton's graph-based scheme [49]. The work on the aggregating cache allows the gathering of more access information at the server, while decoupling client from the any critical timing issues related to prefetching. This is accomplished via cooperative client and server-side modules, as with AFS or Coda [65]. Kroeger and Long [71] compared the predictive performance of the last successor model, Griffioen and Appleton's graph-based strategy, and new techniques based on context modeling and data compression [72]. The earliest proposed use of data compression strategies to predict disk accesses was presented by Vitter and Krishnan [23, 67, 121]. The strategies studied

included *LZ* compression [130], prediction by partial match (*PPM*), and first-order Markov prediction (*FOM*). Shriver *et al.* [111] has provided analytical reasoning for the benefits of read-ahead buffering and prefetching. Other recent work on *ASP* [12] presents a study of a strip prefetching scheme for striped disk arrays. The authors provide separate management of prefetched and regular cache lines with a culling scheme using differential feedback similar to the adaptive marginal utility used in *SARC* [44]. Such prefetching of data is not without costs, many of which are addressed in *ASP*. Any prefetching strategy must have a reasonable lead-time in order to retrieve data before it is actually requested. Additionally, any benefit from this prefetching, like spin-down techniques, lie directly on the data path. Our strategy enables the decoupling of the strategy from the data path, allowing us to disable device-level rearrangement while still benefitting from previous efforts to properly cluster data.

Recent work has shown advances toward utilizing device-level knowledge of physical data layout. Prediction for both caching purposes and prefetching purposes have begun emphasizing spatial locality as having a higher utility than a random access; that is, of two blocks with identical expected likelihood of occurrence, the block nearer the current location of the read head has higher utility. *DULO* [60] presents a buffer cache management scheme that exploits both temporal and spatial locality, while *DiskSeen* [27] presents work utilizing similar table structures for use of predictive prefetching. *DiskSeen* fetches at the device level, and is designed to be synergistic with file-level prefetching strategies. More recent work on *TaP* [83] describes using a separate data structure to store previous addresses in order to identify sequential data streams without having to use precious cache space to do so. Our work seeks to decrease the expected distance between consecutively requested blocks, and would be highly beneficial to such location- and stream-aware strategies.

2.2 I/O WORKLOAD SHAPING

Traditional research to improve performance of hard disks by modifying I/O workloads include scheduling strategies such as *SSTF*, *SCAN* [26], *C-SCAN* [106], and *LOOK* [90]. More recently, approaches for decreasing the growing impact of rotational delay have been presented [57–59,

99, 108]. These efforts are considered orthogonal to the work on prediction and data regrouping presented in this dissertation.

The use of prediction as a means of workload shaping to reduce power consumption has been proposed by Flinn and Satyanaryanan [37] and also Lorch and Smith [84]. These suggestions focused on the ability of prefetching data to allow for increased idle-time periods, which in turn would hopefully allow greater opportunities for disk spin-downs. Similarly, recent work by Weissel *et al.* [123], and Papathanasiou and Scott [94, 95], attempts to actively modify the workload and increase workload burstiness to increase opportunities for disk spin-down. Predictive methods such as these are expected to benefit from metadata strategies we have developed, and are considered orthogonal to our predictive work.

2.3 DATA LAYOUT MAINTENANCE AND FILE SYSTEMS

The desire to place related data together on disk is traditionally accepted as a wise storage-system goal, and recent work indicates that its uses continue to present themselves [27, 60, 62]. For example, work by Kandemir *et al.* [62] focuses on utilizing disk layout knowledge at compiler time for data intensive applications, notably scientific applications.

Access patterns can be used to rearrange tracks on the disk [101], a problem known to be NP-Hard [20], to improve on the organ-piping method [52], detailed and discussed in depth by Wong [126]. Such patterns can also use be used to identify which files to move to tertiary storage [43]. Other forms of disk management include storing data that does not cross track boundaries [104] as well as how to extract that information and use it as stripe unit boundaries [105], storing inodes by embedding them in their directory, and grouping together small files on disk to be read as one [39]. It has been demonstrated that it is possible to separate inodes from data over a distributed system [19].

Early data placement and predictive grouping studies attempted to use frequency of access as an estimated likelihood in order to optimally place high-demand data. The optimum arrangement of files on disk was originally a manual task, placing popular files near the center of the disk cylinder. The necessary automation of this process has been addressed by Staelin and Garcia-

Molina [113–115], whose work dealt with models that provided optimal placement of files where accesses were independent. However, data accesses often involve dynamic relationships, where access dependencies change over time. Berkeley's *FFS* [86, 112] includes attempts to cluster related data and metadata into cylinder tracks on a disk. More recently, Li and Wang combined *FFS* (or *UFS*) and *GFS* modules into a single file system, *EEFS* [80, 81]. However, these approaches typically require disjoint sets as groups. Our approach makes no such constraints, allowing replication between groups formed, although not within them. Such static optimizations are common among modern file systems [24, 86, 119], while our work is toward dynamic solutions that have possible static application. Similar replication was performed by Akyürek and Salem in 1995, where popular "hot" blocks were copied to a common disk area to improve disk performance [4]. However, this study was based only on the global popularity, or percentage of access, rather than inter-file relationships. Dynamic groups [116] attempt to exploit inter-file relationships, but required explicit application hints to determine group membership. Examples of efforts in automated grouping include *C-FFS* [39] (collocating *FFS*), which bases grouping on a directory-membership heuristic, and Hummingbird [110] which utilizes the underlying structure of web files. In contrast, our model does not require any knowledge of underlying data structure, as our grouping mechanism can establish relationships based on observed access behavior, as opposed to inference from file location or content.

Recent work most closely related to our own would include phased-based on-disk caching [15] and use of on-disk free space for file replication [56]. Efforts exploiting free space for reorganization achieve impressive results only after repeating the same access patterns multiple times [56]. While reasonable, we believe the use of repeated runs to be confounding, as such repetition would eliminate single-event occurrences, or the requests of blocks that will never be requested again, as well as strengthening access noise. Efforts for dynamic persistent grouping must be adaptive but resistant to this noise, yet these phenomena introduced by workload repetition would actually reward strategies that refrain from doing so. On-disk caching also shows promise, but requires multiple phases of extraction, analysis, planning, and execution [15], and has several drawbacks. First, they incur high computational costs at various phases in the cycle. While these costs can be alleviated to a point by using low priority operations, they must be completed in a timely manner. Second, the on-disk cache has a single location. Leaving this location may result in a large seek in

order to return to the designated cache area. Finally, with strategies that operate in distinct phases, opportunistic updating becomes difficult or impossible. Indeed, it is entirely possible that all previous caching efforts need to be updated, and this updating occurs at once, rather than gradually. Our work is directed toward dynamic, adaptive, gradual updating that is robust to swift changes in workload behavior.

2.4 POWER

Greenawalt presented one of the earliest studies on modeling power, latency, and life expectancy of hard disks using multiple power states in 1994 [48]. A more detailed approach was presented by Zedlewski *et al.* in 2003 [128] using an extension of the *DiskSim* simulator [17, 18, 40, 41] called *Dempsey*. Zedlewski used simulated disk traces as well as a portion of the 1992 *cello* trace [102] as validation for *Dempsey*, while Greenawalt used a Poisson distribution to model hard disk access behavior.

The earliest suggested use of predictive techniques to dynamically adjust the spin-downs for hard disks for power conservation was presented by Wilkes in 1992 [125]. In 1994, Douglis, Krishnan, and Marsh demonstrated that perfect, non-invasive spin-downs were capable of decreasing disk power consumption 60%, while online algorithms achieved a 53% reduction over the manufacturer's recommended five minute time-out [30]. Later work by Krishnan *et al.* analytically modeled spin-down decisions as a rent-to-buy problem in 1995 [68, 69]. Studies on how to capitalize on these spin-downs by predicting when they should occur were presented by Golding *et al.* [45, 46] and Douglis *et al.* [29, 30]. The greatest power savings achievements to date that use these techniques on an unaltered workload employed an adaptive machine learning algorithm [53, 54] that used a variant on the weighted majority voting algorithms [122] called the "share" algorithm [55]. Similar work focuses on device-level management, similar to spin-down techniques, using various dynamic power management decision engines on a large data set of traces [97]. Recent work on thermal modeling of disk drives suggests that temperature, as well as power, is increasing in importance for drives [50, 51, 64]. Other recent work on data centers uses fast transitions between "active" and "idle" states to save energy on server idle periods [89]. Our work differs from these

efforts in that we seek to change the physical location of information on the hard disk rather than adjust any spin-down timeouts or moving devices between levels of power consumption. Such a strategy has the benefit of being taken off the data path completely in the event of high activity, while previous efforts of restructuring are expected to continue to have a positive effect on system performance. Furthermore, our techniques demonstrate an ability to reduce a workload's footprint or working set by reducing the percentage of raw storage volume retrieved or traversed unnecessarily. This makes these techniques useful to multi-machine systems at the system level, while spin-down efforts only benefit such systems at the machine level.

Recent work by Narayanan *et al.* seeks to accomplish further spin-down savings in enterprise storage systems by temporarily off-loading pending write requests to available persistent storage locations elsewhere in the storage system [92]. Other studies by Crk and Gniady [22] seeks to predict upcoming transitions from a low-power state to a high-power state of storage devices, thus reducing the observed latencies incurred from spinning up the disk. Joukov and Sipek [61] show that constantly spinning the disk up and down decreases the life expectancy of the device. They present *GreenFS*, a file system that utilizes flash technology for providing hierarchical run-time data protection that keeps disks spun down and limits the amount of spin-ups necessary.

The common thread in these works is the concentration on spin-downs or similar *on-off* switching as the mechanism for reducing power consumption. The primary costs associated with these strategies are the observed latency while waiting for the disk to re-enter the active state, or the so-called "spin-up" time, as well as the corresponding power costs. In addition, all power conservation attempts lie directly on the data path, and none are attempted while the disk is active. Recent efforts on data compression show promise, but remain ungeneralizable, and tend to have limited application [66]. Our approach seeks to employ reduced disk activity utilizing predictive grouping while the disk remains active, and incurs no such latency or power penalties. Additionally, our experiments utilize more detailed power measurements than prior published research.

3.0 PROBLEM DEFINITION

As we have discussed, the general goal of placing related data items near one another on the disk can be tackled in a number of ways; therefore, in this chapter we will detail the precise problem that our research seeks to solve. The first step is to understand that latency, in our terms, means not only the amount of time observed for satisfying a single request for a particular item, but the total observed time of the entire system. In particular, we note the existence of a number of unavoidable costs for *any* single request; for instance, a device on a distributed system might have bandwidth constraints and other communication overheads. A single device might have to wait for a channel to become available. These costs are largely uncontrollable, and occur at the start of any request. If we can reduce the number of total requests, we might thereby reduce these unavoidable costs. As a reasonable real-world example, consider a track buffer, part of a standard modern computer. Any request to the hard disk will read an entire track into the track buffer, which acts as a one-item cache whose size is equivalent to the size of the disk track. Any future requests that occur within the same track are read from the track buffer, thereby avoiding costly disk reads. Therefore, in our example, the collocating group becomes the track, and latency reduction is achieved as a result of reduced disk reads. Thus our first goal becomes reducing latency by reducing total requests; in particular, we seek to accomplish this goal through collocating data chunks on disk.

A second goal of our research is also held in our analogy, that of reducing power consumption. Hard disks have been shown to consume up to 30% of total system power, and remain a major concern for reducing the lifetime cost of the system. By reducing the number of disk reads in our analogy, we reduce the total disk seeks, which are among the most costly operations performed by the disk. Since the disk head is mechanical, its operations cost much more than accessing an electrical track buffer. If we collate highly correlated data, we might reduce the workload footprint, and the track buffer will presumably remain the source of requests granted for a long

period of time. This increases the workload's "bursty" nature [14, 16, 91, 94, 95, 102], and aids other strategies orthogonal to our own, such as spin-down time manipulation.

However, the ability to reduce a workload's footprint has additional applications. In distributed systems, a major cost of the system is not just the total number of devices, but energy costs of the total number of *active* devices in the system. Dynamically reducing a workload's footprint could potentially reduce the number of active devices, thereby greatly reducing the cost of the entire system.

Thankfully, data accesses have been shown to exhibit high predictability, which we will exploit in our collocation. The next question we must address is how to gather the appropriate metadata to ensure accurate prediction. We would like our strategy to be applicable dynamically, as workloads can and do shift over time. Thus, the ability to rapidly adapt is also desirable. However, adjusting to the workload prematurely before a trend has been established can be detrimental. Thus, a certain robustness to noise in the signal is also highly desirable. Finally, we must ensure minimal requirements for this volume of metadata. It does no good to greatly reduce workload accesses if metadata accesses increase accordingly, nor does it behoove the system to completely fill system memory with it. We therefore wish to limit the total amount of system metadata used in our predictions.

We therefore seek to accomplish the following.

1. Gather predictive metadata without taxing the underlying system.
2. Use this metadata to collocate related data at the device level.
3. Employ these collocated regions to reduce total device-level accesses, and thereby reduce system latency and energy consumption.

Throughout our work, references to *accesses* and *data accesses*, as well as *access patterns* and *workload traces* are used to refer to *block-level* accesses, rather than file-level, unless otherwise noted. We do this for a number of reasons. First of all, assuming all data chunks are unit sized reduces the calculation costs significantly. Additionally, our work strives to remain as generalizable as possible, and refrain from imposing high-level abstractions upon workloads. Rather than using file-level information to *guess* at how a workload *should* behave, we allow the access pattern to emerge from observed events. Finally, since the vast majority of storage systems operate at the

block level, there is no additional translation necessary. As a result, unless noted, we consider *block*, *chunk*, or *file* to be equivalent. We often refer to a file ID, or a file's estimated probability; these translate to a block ID and a block's probability, for all intents and purposes.

4.0 EXPERIMENTAL METHODOLOGY

The merits of trace-driven simulation of system performance have long been understood [109]. The use of access traces is highly desirable for its realism, particularly so when compared to synthetic functions and independent distributions. This is especially true when evaluating predictive techniques, which must be judged on their ability to identify and exploit predictability in real-world workloads, and not on their ability to coincidentally match a synthetic or statistical generator. Applications for these traces are wide-ranging, including caching, prefetching, memory management, data layout, hybrid (NVRAM) system evaluation, low-level system behavior analysis, system design and performance tuning.

Recently, it has been suggested that longer traces can be approximated by repeatedly using the same smaller trace (or traces) [56]. While reasonable, this strategy introduces new pitfalls that traditional trace usage avoids. For instance, repeating a single trace will boost the access counts, including access "noise". While the overall percentage of noise would remain unchanged, the *same* noise would be recorded, making it more difficult for strategies to eliminate or ignore this system static. Moreover, single-time events completely disappear, giving the illusion that the system need not be concerned with an event that it might never witness again. Further, the access signal becomes somewhat stagnant, with no new event ever arriving. The advantage of repeating a single trace is that a learning or adaptive strategy is given ample time to gather necessary information. However, we would argue that adaptive strategies should be able to cope with sparse information to remain generalizable.

For these reasons, our work is inevitably linked with that of trace gathering and analysis. We have strived to use traces that represent multiple workload characterizations, including established sets that are well documented for ease of comparison and newer traces that we have collected ourselves. These newly collected traces represent varied workloads and conditions.

4.1 CLASSIC TRACE SETS

The trace sets presented in this dissertation represent established workloads used in previous work in related fields, such as caching [8] and file prediction [11, 96, 127], system benchmarking [118], and workload characterization [100].

4.1.1 mozart

The *mozart* set consists of a workstation trace gathered using the *DFSTrace* system [91], providing information at the system-call level. This set represents the original access stream, prior to any caching. These traces were converted into equivalent block-level traces with block sizes of 512, 4096 (4K), and 8192 (8K) bytes. There were four different original trace sizes; day length, week length, month length, and year length. This set has the appeal of allowing the analysis of our strategies over different definitive time periods as well as allowing us to convert easily to different block sizes.

Except where noted, these traces had block IDs numbered according to order of initial access. This numbering strategy implicitly includes a level of optimization in terms of accesses and space, thereby providing a more ambitious baseline against which to compare our grouping strategies.

4.1.2 hplajw

The second set, *hplajw*, is a block-level workstation trace from a HP-UX system [102]. This trace had a single user, John Wilkes, and was used primarily for email and paper editing. This trace set represents disk-level accesses; the authors note that little activity was seen at this level, due to the effectiveness of the UNIX buffer-cache.

This set has the advantage of natively being a block-level trace, and therefore does not require conversion. However, there is only a single trace length, and lacks any information of original file-system level access information, and therefore cannot be accurately converted to traces of differing block sizes.

Much like our *mozart* trace set, unless otherwise noted, this workload had block IDs numbered in order of appearance, allowing for higher quality baselines for our predictive grouping.

4.2 NEWLY COLLECTED TRACE SETS

As storage systems, operating systems, and file systems change, are revised, and evolve, so evolve the demands placed upon them. Workload habits adapt and shift due to increased storage capacities and higher bandwidths. New applications cause new behavior; new behavior yields new demands; new demands dictate new design. As a result, more modern traces are constantly needed in order to avoid outdated assumptions for updated systems. To this end, we have collected our own trace sets, for several workload classifications, in order to verify assumptions made from established trace study.

4.2.1 ranin

The custom trace set *ranin* was collected on a Mac PowerBook G4 1.25 GHz processor with 512 MB of memory on a 5400 RPM Seagate Momentus hard drive with 160 GB capacity. The workstation was running OS X 10.4 with vanilla Darwin and XNU kernel and used the standard `fs_usage` command found on OS X. These traces were gathered in 2007 from November to December. The workload represents a typical graduate student workstation, namely the author's, used for day to day activities, including internet browsing, file editing, code compiling, and running and testing experimental simulations, most of which were custom C++ programs. While there were a few trace interruptions due to rebooting, including one major software update, inaccuracies introduced we considered negligible due to their infrequency. Additionally, the ensuing shifts in workload behavior represent realistic changes due to real-world activity. Cache activity was gathered, but for the majority of our work, they were ignored; only device-level requests were used. These requests were in the form of read and write data and metadata as well as page ins and outs. Table 1 details all the system calls collected through the entire trace, while Table 2 details the system call counts and byte information most pertinent to our research.

4.2.2 playlist

The custom trace set *playlist* was gathered on two different Mac mini G4 workstations, each with 512 MB of memory and running Mac OS X 10.3.9 with vanilla Darwin and XNU kernel. We ran

Table 1: Table of all system call counts from *ranin* trace.

System Call	Count	%	System Call	Count	%	System Call	Count	%
CACHE_HIT	**60,000,000**	**30.4**	fchdir	300,000	0.151	delete	5,270	0.00265
lstat	39,000,000	19.8	**WrData**	**208,000**	**0.105**	buffer	3,976	0.00200
read	31,000,000	15.8	chown	183,000	0.0922	map_fd	3,235	0.00163
write	17,000,000	8.42	rename	175,000	0.0878	access_extended	2,989	0.00150
stat	11,000,000	5.57	fsync	154,000	0.0775	**RdData**	**2,933**	**0.00148**
open	7,500,000	3.78	unlink	146,000	0.0736	exchangedata	1,088	0.000548
pread	6,000,000	3.01	mmap	133,000	0.0668	PAGE_OUT_V	814	0.000410
getattrlist	4,500,000	2.27	mkdir	131,000	0.0659	fchmod_extended	640	0.000322
close	4,300,000	2.18	statfs	124,000	0.0624	flistxattr	618	0.000311
lseek	2,500,000	1.25	getdirentriesat	120,000	0.0603	**RdMeta[async]**	**507**	**0.000255**
fstat	2,200,000	1.13	utimes	115,000	0.0578	symlink	396	0.000199
getdirentries	1,700,000	0.874	lchown	100,000	0.0503	fstat_extended	348	0.000175
PAGE_IN	1,500,000	0.733	execve	90,000	0.0454	getxattr	297	0.000149
WrMeta[async]	**1,100,000**	**0.541**	fsctl	68,000	0.0344	searchfs	211	0.000106
RdData[async]	**1,100,000**	**0.531**	listxattr	56,000	0.0284	writev	195	9.81e-05
fstatfs	775,000	0.390	fchmod	54,000	0.0273	**WrMeta**	**189**	**9.51e-05**
WrData[async]	**691,000**	**0.348**	sync	33,000	0.0166	pathconf	155	7.80e-05
RdMeta	**667,000**	**0.335**	setattrlist	26,000	0.0129	setxattr	146	7.35e-05
pwrite	526,000	0.265	stat_extended	26,000	0.0128	fgetxattr	70	3.52e-05
PgIn[async]	**484,000**	**0.244**	fchown	22,000	0.0112	link	66	3.32e-05
chmod	421,000	0.212	rmdir	9,794	0.00493	fsetxattr	33	1.66e-05
access	409,000	0.206	readlink	9,145	0.00460	revoke	11	5.54e-06
PAGE_OUT_D	398,000	0.200	ftruncate	8,817	0.00444	removexattr	10	5.03e-06
PgOut[async]	**394,000**	**0.198**	**PgOut**	**5,379**	**0.00271**	chflags	4	2.01e-06
chdir	302,000	0.152	**PgIn**	**5,337**	**0.00269**	fremovexattr	1	5.03e-07

Table 2: Table of select system call counts and byte counts from *ranin* trace. Percentages reported include only those calls present in this table.

System Call	Total Bytes	Byte %	Accesses	Access %	Bytes / Access
CACHE_HIT	**231 GB**	**52.5**	60,000,000	**92.9**	**4096 B**
RdData[async]	**102 GB**	**23.2**	1,060,000	**1.62**	**101 KB**
WrData[async]	**59 GB**	**13.51**	691,000	**1.06**	**90 KB**
RdMeta	**14 GB**	**3.13**	666,000	**1.02**	**22 KB**
WrMeta[async]	**11 GB**	**2.47**	1,070,000	**1.65**	**11 KB**
PgIn[async]	**8.1 GB**	**1.84**	484,000	**0.744**	**18 KB**
PgOut[async]	**7.6 GB**	**1.73**	394,000	**0.605**	**20 KB**
WrData	**7.3 GB**	**1.65**	208,000	**0.320**	**37 KB**
PgOut	**55 MB**	**0.0123**	5,379	**0.00827**	**11 KB**
PgIn	**54 MB**	**0.0120**	5,337	**0.00820**	**10 KB**
RdData	**11 MB**	**0.00253**	2,933	**0.00450**	**4075 B**
WrMeta	**2.9 MB**	**0.000638**	189	**0.000291**	**16 KB**
RdMeta[async]	**1.2 MB**	**0.000262**	507	**0.000779**	**2442 B**

Table 3: Table of all system call counts from *playlist, shuffle* trace.

System Call	Count	%	System Call	Count	%	System Call	Count	%
pread	5,080,000	40.5	setattrlist	1,416	0.0113	PAGE_IN	25	0.000199
RdData[async]	**4,740,000**	**37.8**	exchangedata	1,416	0.0113	lstat	23	0.000183
CACHE_HIT	**1,480,000**	**11.8**	delete	1,416	0.0113	stat	21	0.000167
getattrlist	784,000	6.25	read	813	0.00648	rename	20	0.000159
open	151,000	1.20	fstat	777	0.00619	chmod	20	0.000159
close	151,000	1.20	**RdMeta**	**321**	**0.00256**	**PgIn[async]**	**9**	**7.17e-05**
fsync	149,000	1.18	**WrData**	**185**	**0.00147**	getdirentries	8	6.38e-05
WrData[async]	**2,144**	**0.0171**	**WrMeta[async]**	**56**	**0.000446**			
pwrite	2,124	0.0169	write	34	0.000271			

Table 4: Table of select system call counts and byte counts from *playlist, shuffled* trace. Percentages reported include only those calls present in this table.

System Call	Total Bytes	Byte %	Accesses	Access %	Bytes / Access
RdData[async]	**290 GB**	**98.0**	4,740,000	**76.1**	**64 KB**
CACHE_HIT	**5.7 GB**	**1.91**	1,480,000	**23.8**	**4096 B**
WrData[async]	**391 MB**	**0.129**	2,144	**0.0345**	**187 KB**
RdMeta	**2.4 MB**	**0.000792**	321	**0.00516**	**7838 B**
WrData	**543 KB**	**0.000175**	185	**0.00297**	**3006 B**
WrMeta[async]	**437 KB**	**0.000141**	56	**0.000900**	**7982 B**
PgIn[async]	**43 KB**	**1.37e-05**	9	**0.000145**	**4836 B**

Table 5: Table of all system call counts from *playlist* trace (no shuffle).

System Call	Count	%	System Call	Count	%	System Call	Count	%
pread	5,120,000	37.6	setattrlist	2,544	0.0187	**PgIn[async]**	**15**	**0.000110**
RdData[async]	**4,780,000**	**35.1**	exchangedata	2,544	0.0187	write	14	0.000103
CACHE_HIT	**2,380,000**	**17.5**	delete	2,544	0.0187	stat	14	0.000103
getattrlist	842,000	6.19	read	1,347	0.00990	rename	14	0.000103
open	156,000	1.15	fstat	1,330	0.00978	chmod	14	0.000103
close	156,000	1.15	**RdMeta**	**386**	**0.00284**	getdirentries	6	4.41e-05
fsync	152,000	1.12	**WrData**	**179**	**0.00132**	statfs	1	7.35e-06
WrData[async]	**3,830**	**0.0282**	PAGE_IN	48	0.000353	pathconf	1	7.35e-06
pwrite	3,816	0.0281	lstat	22	0.000162			

Table 6: Table of select system call counts and byte counts from *playlist* trace (no shuffle). Percentages reported include only those calls present in this table.

System Call	Total Bytes	Byte %	Accesses	Access %	Bytes / Access
RdData[async]	**292 GB**	**96.76**	4,780,000	**66.7**	**64 KB**
CACHE_HIT	**9.1 GB**	**3.01**	2,380,000	**33.2**	**4096 B**
WrData[async]	**703 MB**	**0.227**	3,830	**0.0535**	**188 KB**
RdMeta	**2.7 MB**	**0.000876**	386	**0.00539**	**7357 B**
WrData	**90 KB**	**2.83e-05**	179	**0.00250**	**512 B**
PgIn[async]	**72 KB**	**2.28e-05**	15	**0.000209**	**4915 B**

a playlist of 148 songs (mp3 files), with a runtime of approximately 14.8 hours, on each machine. Traces were gathered from August 31, 2008 to March 23, 2009, resulting in play counts over 300. All disk activity due to the music software was isolated and recorded using built-in tracing facilities of Mac OS X. One trace gathered information on a sequential playlist, while the other playlist was shuffled. These traces, denoted as *playlist* and *playlist, shuffled*, represent one extreme of predictability, an estimated upper bound on how predictable a realistic workload could be. Table 3 and 4 detail system calls collected through the *playlist, shuffled* trace, while Table 5 and 6 detail the *playlist* trace.

5.0 OPTIMAL GROUPING

In order to construct a solution to both latency and power agendas detailed in Chapter 3, we begin by first exploring how an optimal solution might be constructed. To do so, we must formally define precisely the problem of data grouping and what it means for a strategy to be optimal. We will show that relaxing our initial definition of this problem results in a problem easily solvable with a greedy approach. Further, we provide proofs of our greedy algorithm's optimality in terms of group transitions, disk distance, and power consumption due to mechanical movement of the disk arm.

5.1 MOTIVATION

Our strategy for optimally grouping data chunks utilizes a future-aware algorithm. Using such an oracle-based strategy in practice is impossible; our view of future requests is imperfect. However, this strategy serves as an illuminating bound on the impact of predictive grouping strategies and serves to illustrate what trends we might expect or strive toward.

5.2 OPTIMAL GROUPING PROBLEM DEFINITION

The problem definition for optimal data grouping is as follows.

Input: A sequence of requests for stored items $FT = \{(f_0, s_0), (f_1, s_1), ...\}$, where ordered pairs (f_i, s_i) represent file ID (f_i) and file size (s_i), C_{max}, a maximum size of a group in bytes, and D, a maximum size of the disk in bytes.

Output: A list of groups, allowing for replication between them, such that the total number of groups traversed (*i.e.* the total number of switches or transitions between groups) is minimized, satisfying the following constraints.

1. Every file must be in at least one group (no loss of data)
2. No group uses more disk space than C_{max} (all groups fit into a track)
3. No more than $\lfloor D/C_{max} \rfloor$ groups are used (we do not use more space than is available on the disk)

If no solution exists, we output 0 (or "no solution").

This problem definition provides the general formulation that we address throughout the remainder of this dissertation. But optimally reducing track transitions is difficult. However, should we relax the problem, the solution becomes quite simple. The relaxed problem we chose to solve is identical to the general problem with the exception that we remove the disk size constraint, D. In effect, we allow D to be arbitrarily large. Supposing we know the future exactly, and have a disk of *arbitrarily large capacity*, a simple greedy algorithm can produce a static grouping scheme that is optimal in the number of transitions. We call this optimal algorithm *DrNO* (Data replication: Naïve Optimal) [32, 33]. The pseudocode is given in Algorithm 1.

5.3 OPTIMAL BEHAVIOR OF DRNO

The general strategy behind our optimal algorithm is try to make the current group as big as possible and throw it away as soon as we are done with it. Since we do not care about the total disk space used *in solving this optimal grouping problem*, *i.e.* how many groups are used in the process, we are able to extract the greatest benefit from each group.

Theorem 5.3.1 *DrNO provides an optimal solution for minimizing the number of transitions.*

Proof The proof of optimality that we construct is an indirect proof. We assume that algorithm *DrNO* is not optimal and reach a contradiction.

First, we note that both the second and the third constraints hold for our algorithm. The third is guaranteed because as soon as a file does not fit within the current group, we form a completely

ALGORITHM 1 DRNO(FT, max) - an optimal, oracle-based greedy algorithm for solving the relaxed version of the optimal grouping problem.

Input: a sequence of requests for stored items, FT; a maximum size of a group in bytes, max, equivalent to C_{max} from the general grouping problem in Section 5.2

Output: a list L of n groups $G_1 ... Gn$

```
for all f in FT do
    if SIZEOF(f) > max then
        PRINT "No solution"
        return NIL
    end if
    if SIZEOF(G)+ SIZEOF(f) ≤ max then
        ADDTOGROUP(G, f)
    else
        ADDTOGROUPLIST(L, G)
        G ← NIL
    end if
end for
return L
```

new group. The second constraint is guaranteed since every file in the trace is placed into a group that can contain it (unless a file's size exceeds C_{max}, in which case we immediately exit out of the program). Therefore, our algorithm finds a solution if one exists.

To prove optimality, we compare our algorithm's behavior to that of an optimal solution's behavior. Note that an optimal solution exists. Among all optimal solutions, we consider the one whose behavior most closely resembles *DrNO*. Call this solution OPT.

We define "most closely resembles" to mean the following. Consider the positions in FT that an algorithm inserts a group switch. (In *DrNO*, this is just before a new group is formed.) We say that a solution behaves most like *DrNO* if it has the most number of consecutive group switches in the same positions as *DrNO*, beginning at the first file in FT.

Define X as the first group switch in OPT that differs from *DrNO*. In other words, at position X, OPT has had m group switches and *DrNO* has had n group switches, where $n \neq m$.

Note that it must be the case that $n \leq m$. Since *DrNO* greedily fills its groups until no more files can fit, it can not be the case that *DrNO* has a group switch before OPT. We now define Y as the next position in FT that *DrNO* has a group switch after X. Call the last place in FT where *DrNO* and OPT had a group switch at the same place Z.

Call k the number of groups that OPT has. We construct a solution O' from OPT that more closely resembles *DrNO* and remains optimal, thus reaching our contradiction. Construct O' in the following way. Add the $k+1^{st}$ group, which contains all the files in FT from position Z to position $Y-1$, to OPT. This does not violate the second condition, nor the third condition, since these files can fit into a group (since *DrNO* put them all in a group), and no files are removed from any groups. Use the $k+1^{st}$ group to put a group switch at location Y and remove all other group switches from Z to Y. Thus, O' has s group switches, where $s \leq m$, and O' more closely resembles *DrNO* than OPT. But OPT was the optimal solution that *most* closely resembles *DrNO*. Thus, we reach a contradiction, and we are done. ∎

This optimal grouping scheme does not just minimize transitions. If the groups created by our optimal grouper are laid out linearly on the disk, we obtain an optimal solution for minimizing *distance*, or the number of groups that we must traverse throughout the entire workload. This distance is defined as

$$dist = \sum_{i=1}^{n} dist(G(i), G(i+1)) \tag{1}$$

where n is the total number of transitions and $dist(G(i), G(i+1))$ denotes the distance between the current group at the time of the i^{th} transition and the target group $G(i+1)$ that will be switched to. This is of particular interest because of its applicability to hard disks. Our constructed groups can easily be interpreted as tracks on the disk. While we denote $G(i)$ as the group used until the i^{th} transition, we will denote G_j as the j^{th} group or track on the disk. In other words, $G(i+1)$ will be *needed after* $G(i)$ in the workload, while track G_{j+1} is *located after* track G_j on the disk.

In order to translate from a requested group $G(i)$ to a location on disk, G_j, we use a transformation function T such that

$$j = T(G(i)) \tag{2}$$

Alternatively, this relationship can be described as

$$G_j = G_{T(G(i))} \tag{3}$$

Using this definition, we can redefine our distance metric, replacing $dist(G(i), G(i+1))$ by using our transformation function T.

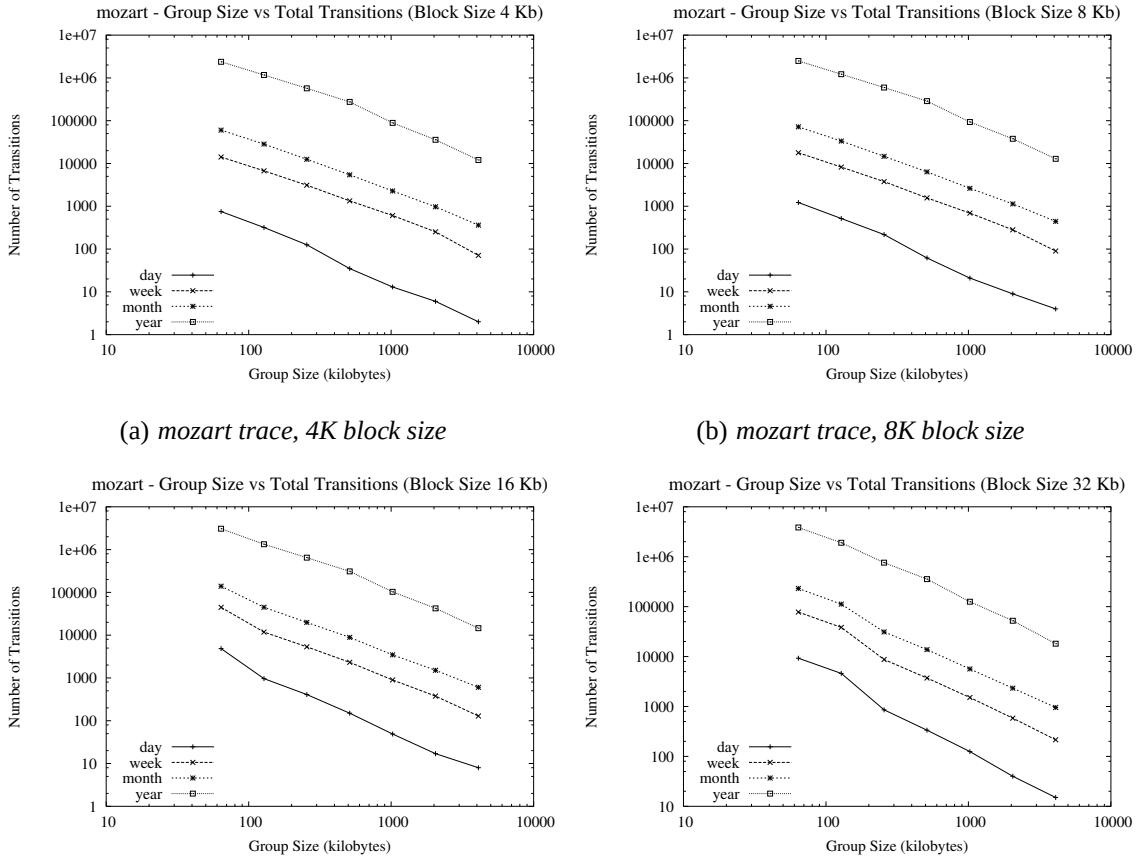
$$dist = \sum_{i=1}^{n} |T(G(i)) - T(G(i+1))| \tag{4}$$

While it is easy to see that our algorithm, combined with a linear layout strategy of groups in order of creation, produces unit size distances, this definition will become useful when we discuss other grouping methods.

Since the number of transitions is minimized by *DrNO*, and all transitions result in a distance of 1, it follows directly that the distance is also minimized by *DrNO*.

Corollary 5.3.2 *Assuming a linear layout of groups in order of creation, algorithm DrNO provides an optimal solution for minimizing distance.*

Once again, this result has particular applicability for hard disks. Our distance metric from Equation (4) translates directly into track distance. This is especially appealing since power and latency penalties due to mechanical components of the hard disk depend upon the number of times we seek a new track and the track distance of each seek. Minimizing the distance via a minimized number of seeks, each of which is unit sized, results in minimized power and latency costs due to these mechanical components. Thus, our algorithm *DrNO* with a linear layout of groups in the form of disk tracks is also optimal for minimizing power and latency due to the mechanical components of the disk arm.



(a) *mozart trace, 4K block size*

(b) *mozart trace, 8K block size*

(c) *mozart trace, 16K block size*

(d) *mozart trace, 32K block size*

Figure 1: Comparison of various block sizes for month length *mozart* trace.

5.4 DISCUSSION

Perhaps the most interesting trend exhibited by *DrNO* is the number of group transitions against the size of each group. For every case tested, we see roughly linear, generally decreasing relationship on a log-scaled graph, making the actual relationship (approximately) inversely proportional between group size and transitions. This suggests inherent diminishing returns for our grouping problem.

In more practical predictive grouping, this behavior of diminishing returns is expected to per-

petuate, and become more critical. Without a perfect oracle upon which to draw, predictions must be made based upon heuristics. Thus, very valuable blocks will, presumably, be chosen first. As the group size increases, blocks' "values" will decrease, granting less and less benefit per block. In this way, the choice of a grouping scheme becomes somewhat less crucial for small blocks; one reasonable grouping scheme should closely approximate another. However, when group sizes become very large, and grouping schemes form drastically groups, the choice of scheme becomes much more critical. While larger group sizes will allow for better performance, we should anticipate a need to reduce the number of predictive groups necessary. Figure 1 shows a brief comparison of *mozart* traces for varying block and group sizes.

Interestingly, our algorithm exhibits a counter-intuitive result in the amount of necessary space for predictive groups. When forming predictive groups using replication, one would assume increasing the group size would result in a larger overall footprint for predictive groups. This is not the case for *DrNO*; the total number of groups needed decreases faster than the group size increases. This result is also shown in Figure 1, since the number of *groups* necessary is identical to the number of *transitions* necessary. (Actually, the number of groups is equal to the number of transitions plus one.) While intriguing, this behavior is not to be expected in more practical methods, as we will see in Chapter 6, where we will compare practical methods to *DrNO* to see how close those methods can come to optimal behavior. Resisting diminishing returns will also become increasing crucial for designing a dynamic grouping engine, in Chapter 9.

6.0 STATIC GROUPING

As we have stated, predictive grouping is the identification of relationships between data, based on predictions of future access patterns, with the aim of grouping together the most related data items. If a workload exhibits repetitive or otherwise predictable access patterns, then predictive

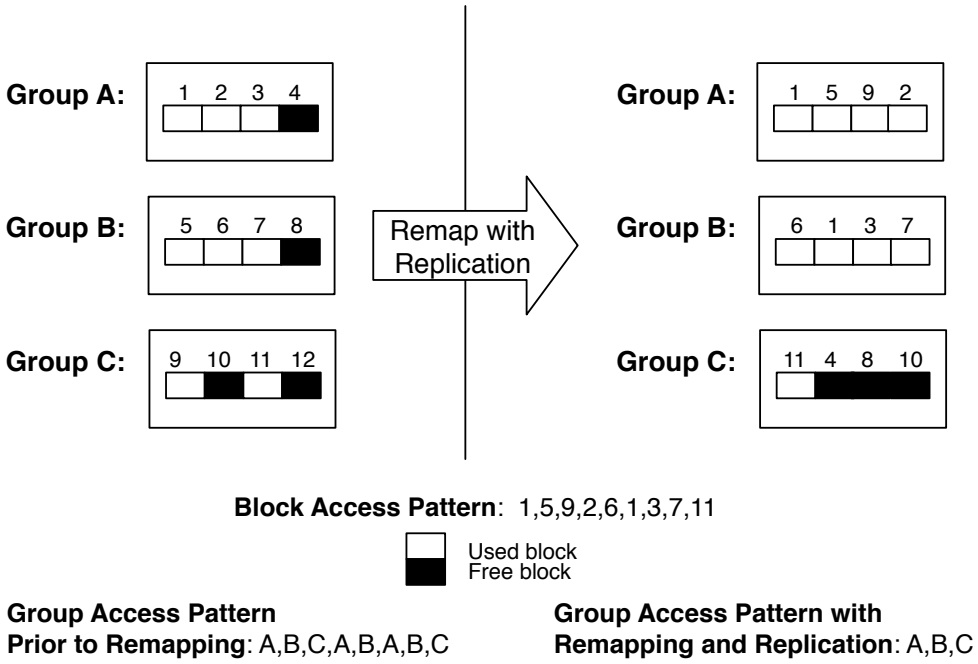


Figure 2: Predictive grouping example. The grouping on the left is a sequential layout, including some free blocks. The grouping on the right is one possible remapping that allows for replication between groups.

grouping can be used to reduce the number of power-state transitions, reduce the number of active nodes in a multi-device system, and even reduce the mechanical activity within a single device. Predictive grouping can thereby improve performance and energy efficiency in storage systems, while simultaneously reducing access latencies. See Figure 2 for examples of group remapping with replication.

Successful strategies result in fewer transitions among groups; our working analogy of disk tracks demonstrates this eloquently. Intuitively, fewer transitions among groups translates to fewer disk seeks and smaller overall latency and power. A secondary result is an increase in access burstiness, allowing for orthogonal strategies such as disk spin-down additional opportunity of application. To this end, we must examine the effectiveness of several prediction strategies, each allowing for replication *across* groups, but not within them.

6.1 MOTIVATION

Our optimal grouping strategy in Chapter 5 utilizes an oracle-based strategy, allowing for perfect future prediction. Obviously, such a strategy is impractical in practice; no perfect predictor exists. However, the use of past events to predict the future has been established as a solid strategy capable of adaptability, high accuracy, and resilience to signal noise [6, 8, 23, 77, 111, 121]. Scores of applicable policies abound, from graph-based modeling [49] to multi-order context prediction [71]. Even simple strategies such as last successor have been shown to have surprising effectiveness for predictive purposes [77]. Ergo, we tackle our first question. What strategy are we to choose?

A number of factors influence this critical decision. Prediction accuracy is, of course, a high priority. Speed, or asymptotic behavior, is certainly a significant concern. Robustness in the face of signal noise, seemingly random behavior of the workload, is highly desirable, but we must remain adaptable. Small storage requirements are also critical; it does little good to require great amounts of metadata stored for small workload footprints. With these factors in mind, our first decision was to explore the use of first-order successor information (*i.e.* based on a context depth of size one). This metadata strategy has far reduced storage requirements than multi-order strategies and has been shown to have applications for predictive caching, exhibiting adaptability, resilience to

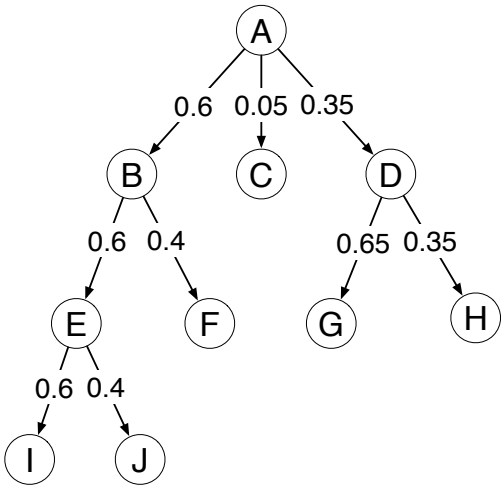


Figure 3: Simple access tree, where nodes denote blocks and directed edges are weighted with estimated likelihood of occurrence.

signal noise, and high prediction accuracy [8].

6.2 GROUPING STRATEGIES

6.2.1 Baseline Strategies

We present two baseline static grouping schemes used in this project. The first method, *NoRep*, or *no replication*, lays the data out linearly on the disk according to ID, maximizing density. To use this grouping scheme, on a transition we seek the only group that contains the offending file (the file that caused the transition). This method proves useful for reducing the distance of a transition compared to other replicating strategies, but is expected to have many transitions.

The second method, *maximal replication*, or *MaxRep*, groups blocks linearly, with each group G_i beginning at block B_i and ending at block $B_{i+groupSize}$. This scheme maximizes replication of blocks. To use this scheme, on a transition we seek the group that begins with the offending block.

Such a scheme might prove useful in reducing transitions over *NoRep*, but may suffer from greater track distances.

ALGORITHM 2 OE_ME$(T, root, max)$ - a balanced approach for forming a predictive group.

Input: set of first-order successor trees, T; a root ID, $root$; a maximum group size, max

Output: a set of IDs, G, representing the predictive group

```
ENQUEUE(max_pq, root, 1)
while ISNOTEMPTY(max_pq) and SIZEOF(G) < max do
    p ← TOPPRIORITY(max_pq)
    f ← DEQUEUE(max_pq)
    if SIZEOF(G)+ SIZEOF(f) ≤ max then
        ADDTOGROUP(G, f)
        max_pq ← OE_ME_EXPAND(T, max_pq, f, p)
    end if
end while
return G
```

6.2.2 Predictive Grouping

For the predictive grouping methods, we need to maintain successor information for each ID. However, tracking successor paths of arbitrary length has high metadata overheads. Instead, we use first order successor information, tracking immediate successors, drastically reducing the spatial requirements to a practical amount. We then use this simple information to build larger groups of related files based on access trees. A simple access tree is given in Figure 3.

Using this first order successor information, there are two strategies we can adopt. The first strategy is a breadth-first expansion (*BFS*), cautiously capturing all of a block's successors before moving on to further descendants. Adopting a *BFS* strategy for the tree in Figure 3, we would add (in order) A, B, D, C, E, F, G, H, etc. The second strategy is an aggressive depth-first expansion (*DFS*), seeking to obtain as many successors along a single "most likely" path, hoping to maximize the use of that successor path at high risk of missing other paths. Adopting a *DFS* strategy for the tree in Figure 3, we would add A, B, E, I, etc.

ALGORITHM 3 OE_ME_EXPAND(T, max_pq, f, p) - expands the maximum priority queue used in Algorithm 2. Note that P(T, f, s) denotes a function used to calculate the estimated probability of child s of f's access tree within T.

Input: set of first-order successor trees, T; a maximum priority queue, max_pq; a file or block ID, f; an estimated probability, p

Output: max_pq

for all s such that s is a child of f in T **do**

$p \leftarrow p\times$ P(T, f, s)

ENQUEUE(max_pq, s, p)

end for

return max_pq

A third strategy, which we call *OE ME*, or *Optimal Expansion, Maximized Expectation*, uses both of these simpler strategies by performing an automatically balanced expansion. This strategy is similar to the balanced approach used in recent access predictors [7, 74] and has previously shown pattern modeling qualities [23]. It is also similar to A* searching and Huffman encoding. See Russell and Norvig [103] for comprehensive discussion on A* search and Sedgewick [107] for details on Huffman encoding. The tree in Figure 3, under our new strategy, would add A (prob $= 1.0$), B (prob $= 0.6$), E (prob $= 0.6 \times 0.6 = 0.36$), D (prob $= 0.35$), F (prob $= 0.6 \times 0.4 = 0.24$), etc.

The general algorithm for grouping using these methods is given in Algorithm 2. The crucial point is within the Expand subroutine, in Algorithm 3, where we use the global estimated likelihood of the file f multiplied by the local estimated likelihood of child s of f.

6.3 EXPERIMENTAL SETUP AND DESIGN

We evaluated our performance through simulation on several different trace sets. The first set, *mozart*, consists of a typical workstation file system trace gathered using the *DFSTrace* sys-

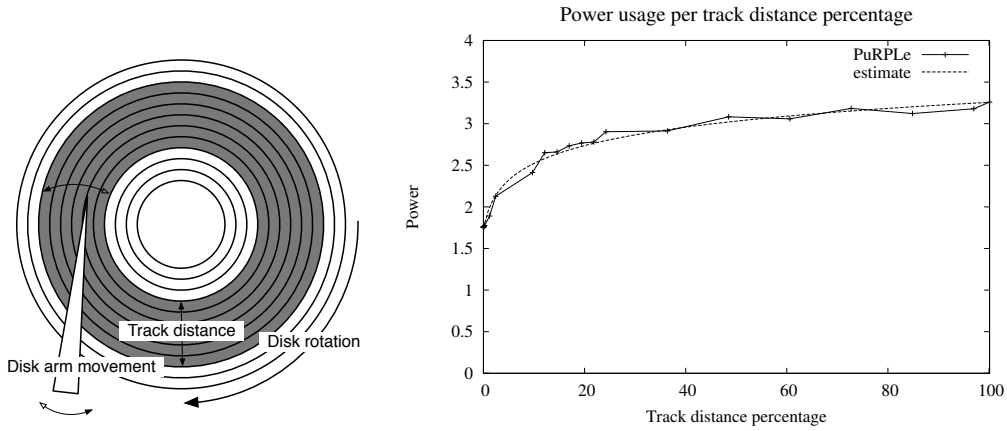
tem [91]. The file system workloads were converted to their equivalent 4KB block-level read workloads. The second set, *hplajw*, is a block-level workstation trace [102]. These workloads were chosen because they are not drawn from synthetic functions or independent distributions, but rather represent real-world traces that exhibit the realistic predictability and patterns of a data access workload resulting from user, program and operating system behavior. In addition, they are lengthy traces obtained over extended periods of time, allowing us to evaluate the effectiveness of grouping as a layout mechanism. We find the *mozart* trace particularly useful since we are able to evaluate the effectiveness of our strategies at varying time spans, up to a year-long period. We have found our balanced approach to be more robust than other methods in terms of effectiveness over extended periods of time. The *hplajw* workload, a block-level workstation trace, shows results very similar to those from the *mozart* traces, in spite of their different origins, both traces were evaluated in terms of block-level layout.

In order to obtain accurate energy estimates, we used average power measurements from a variety of hard-drives [21]. We selected these measurements for our estimates because they represent detailed, isolated power measurements of a disk arm rather than aggregate measurements of total energy over time. Detailed energy usage was gathered using IDE hard-drives ranging from 2 GB to 80 GB. The energy consumed was evaluated using benchmarks and simultaneous measurement of energy consumed over the separate 12 and 5 Volt power lines. Separate power lines provided the advantage of isolating the energy usage of drive mechanics (12 V line) from drive electronics (5 V line). The voltage drop was measured across 0.01 Ω resistors in series with each of the two lines.

We used a DAQ system collecting 20,000 samples per second for each benchmark experiment. These samples were used to calculate the average power usage of the drive based on the percentage of the disk that was traversed during each seek. See Figure 4(a) for an illustration. The ability to isolate drive motor power sources, combined with high frequency sampling, allowed us to isolate the contribution of the disk arm movement to the disk's overall energy usage.

We used these average power measurements to estimate power using log functions. We used a generic log function of the form

$$power = a \times log(perc + b) + c \tag{5}$$



(a) *Disk Mechanical Movement* (b) *Disk Power Consumption*

Figure 4: Reducing disk power consumption by reducing arm movements.

Note that $perc$ here represents the percentage of the disk that was traversed, a bounded quantity ranging from 0 to 100. A repeated simulated annealing process was used to adjust the weights a, b, and c such that the average squared distance was minimized. This strategy was adopted to obtain functions that were much closer to data than available through simple function estimators. The power estimation function, shown in Figure 4(b), has an average squared distance of 0.002604. The calculated parameters for this drive are provided in Table 7.

The energy used by each of the trace workloads, and the latency incurred, was evaluated through simulation based on the above drive performance parameters. Our simulations were conducted as follows: First, we read through the appropriate trace, gathering first successor information. We then read the trace again, calculating track distances, estimating latencies and energy consumption. During this second run through the workload, we evaluated energy consumption and latencies for the different grouping algorithms and their resulting layouts. We also record the number of groups formed by the different algorithms, the total transitions between these groups, and the number of requests satisfied before each group transition. The following equation was used for calculating the time of a disk arm movement due to transitions.

Table 7: Power parameters for caviar2gb through repeated simulated annealing.

Drive	a	b	c
caviar2gb	0.331219	1.036054	1.729115

$$time = avgSeekTime \times \sqrt{\frac{trackDist}{avgTrackDist}} \tag{6}$$

We used a minimum seek time of 0.001 seconds and an average seek time of 0.008 seconds for the results presented below.

Total energy consumed was estimated by multiplying the power consumption by the latency. Note that since we are using an average power figure, we can use simple multiplication and need not integrate.

$$energy = power \times time \tag{7}$$

Tracks were laid out in the order they were requested, imposing no structure to the tracks themselves. This is in accordance with the expected behavior of the different algorithms being dynamically applied to a workload. The exceptions are the two baseline algorithms for which tracks were laid out in linear order for consistency, as they do not offer a clear sequence of group creation.

6.4 RESULTS

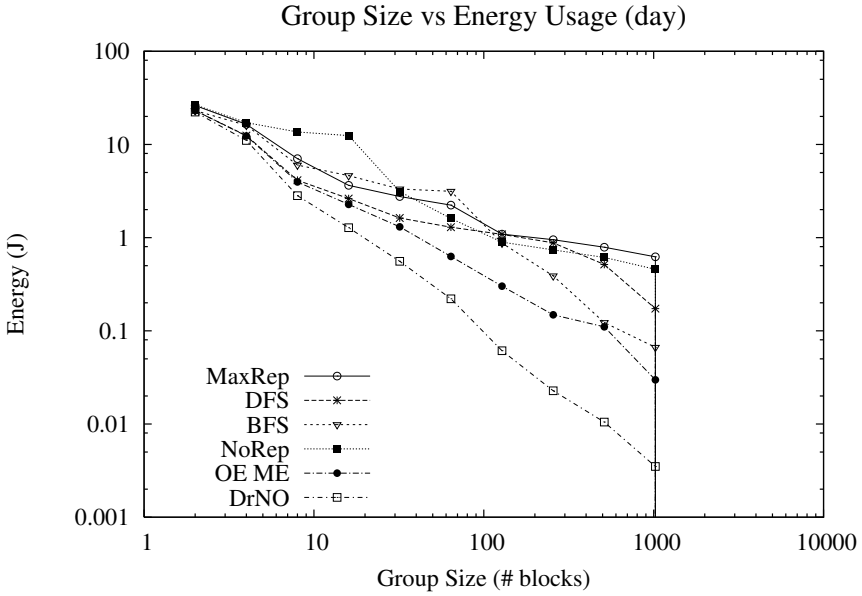
Table 8 shows a comparison of strategies based on the energy and latency penalties of the disk arm movement. These numbers are for the *mozart* year trace, a group size of 2048 blocks, and the performance and measured energy characteristics of a Western Digital caviar2gb disk. As we can see from the table, *DrNO* is by far the most effective strategy, requiring less than 5 Joules of energy and less than 8 seconds to process the entire trace. This is an impossible result to achieve

Table 8: Comparison of strategies based on the energy and latency costs associated with the disk arm. These numbers are for the *mozart* year trace, group size 2048, based on a Western Digital caviar2gb disk.

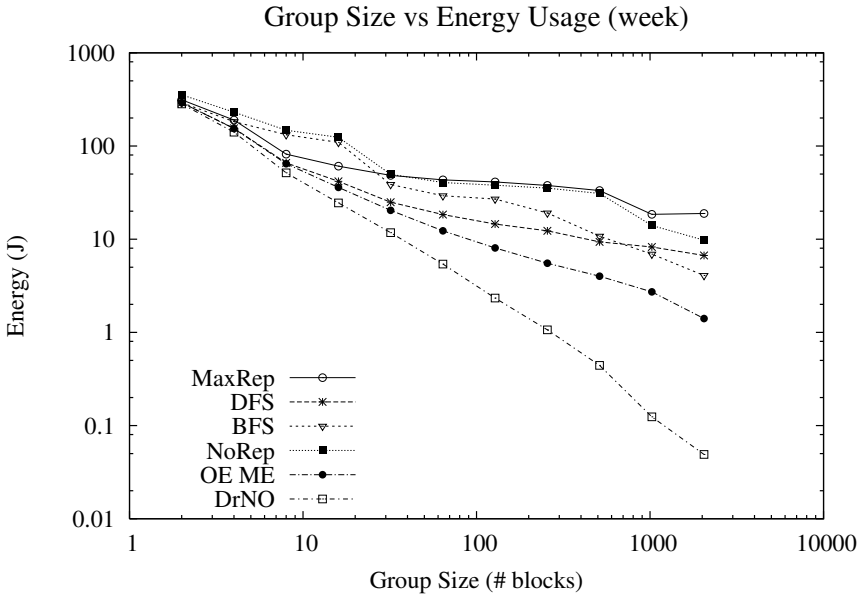
Strategy	**Energy (J)**	**Time (sec)**
DrNO	**4.31**	**7.54**
OE ME	**194.63**	**389.53**
DFS	555.31	1153.90
BFS	790.05	1608.94
MaxRep	4533.75	13072.22
NoRep	**968.56**	**1694.98**

in practice, due to *DrNO*'s ability to perfectly know the future and to use an unlimited degree of replication and space for its groups, but it does illustrate the dramatic potential of grouping reductions for data reads. To further clarify it is important to point out that these results are for all block read requests generated, and do not include write requests. This means that our approach is being applied to a subset of the workload, read requests, but this is the very subset that cannot be addressed by dynamic relocation of data to the current position of the disk-head (as is done in logging or copy-on-write techniques). While writes allow us to physically write the data to a new location and update metadata to indicate this new location, reads must be satisfied from wherever the data is available, and predictive grouping attempts to avoid having physically remote requests. While *DrNO* is an unattainable ideal, it does demonstrate how effective predictive grouping with replication may be. Our algorithm, *OE ME*, is far from reaching this infeasible ideal but is nonetheless more than a four-fold improvement over the non-replicating baseline (*NoRep*) for both energy and latency. The improvement over the aggressively replicating strategy (*MaxRep*) is even more impressive, at almost twenty times, illustrating the dangers of unrestrained replication.

In Table 8 we show the performance of our predictive grouping approach compared for a single group size and workload. A better view of these results and their meaning can be seen when

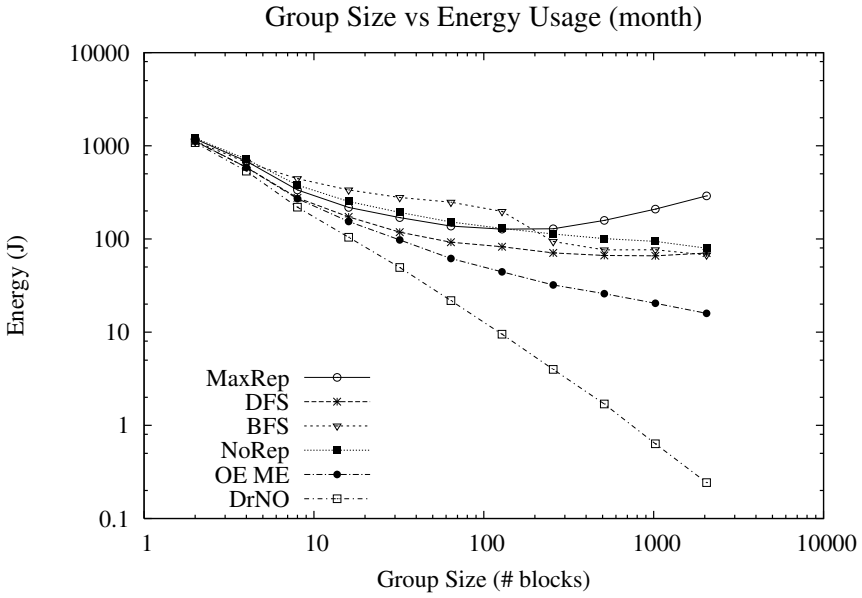


(a) *Day-long trace*

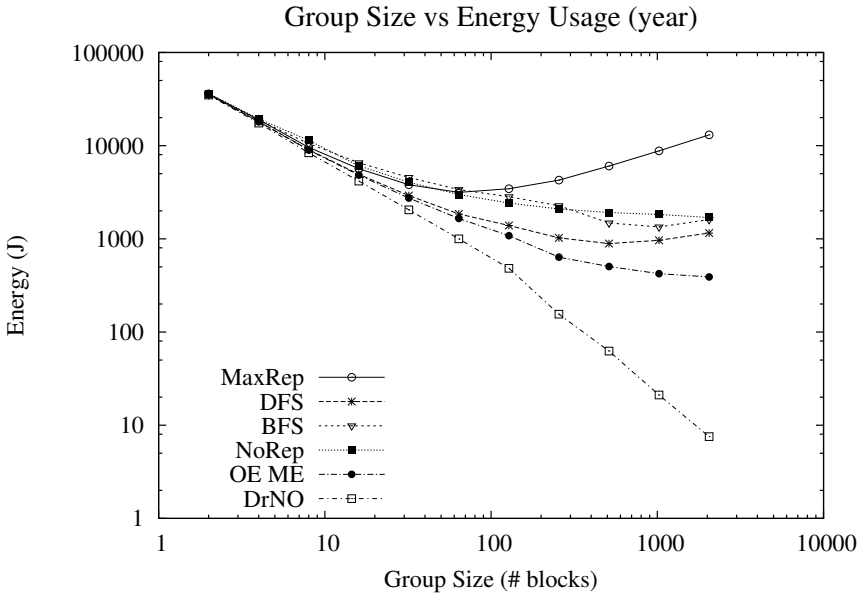


(b) *Week-long trace*

Figure 5: Energy usage due to disk arm movement for the *mozart* day and week length traces. Note that the sudden drop off for the last group size in the day length trace indicates the point at which all the unique files fit within a single group.



(a) *Month-long trace*



(b) *Year-long trace*

Figure 6: Energy usage due to disk arm movement for the *mozart* month and year length traces.

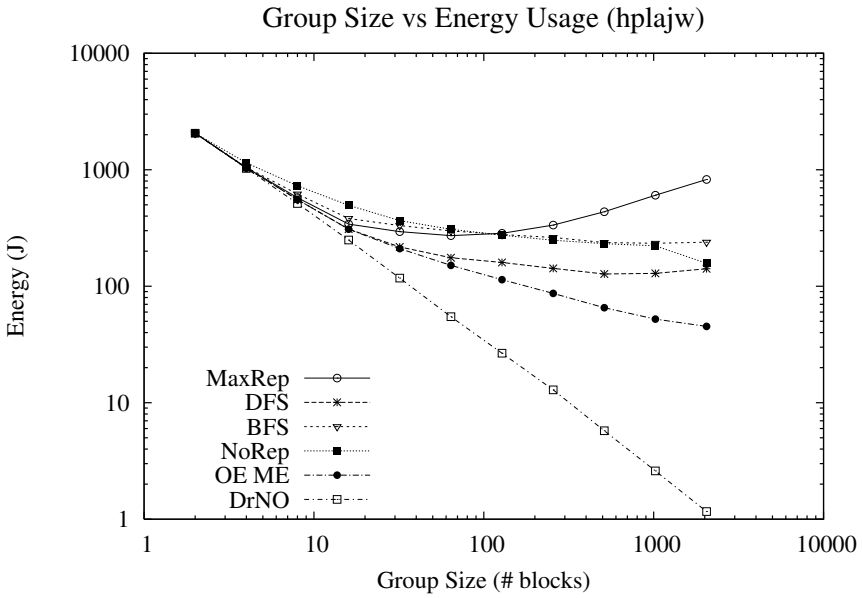


Figure 7: Energy usage due to disk arm movement for the *hplajw* trace.

considering different workloads, durations, and group sizes. Figures 5 and 6 show the energy consumption from the *mozart* trace set, evaluated for four different durations and for different group sizes. This is the energy cost that would be incurred by the different grouping algorithms, based on the power consumption function estimated from the caviar2gb drive. Improved performance is indicated by a reduction of this cost, as we aim to reduce overall energy consumption.

These results demonstrate that there are massive savings potentials, over 99%, achievable by our optimal grouper, *DrNO*. Our balanced expansion algorithm, *OE ME*, demonstrates up to 70% reduction over the closest competing strategy for non-trivial group sizes. This balanced expansion also demonstrates robustness to both group size as well as trace length not exhibited by the competing strategies. As the group size increases for *DrNO*, we see a continuing decrease in energy consumed. This is not surprising given the optimal nature of the grouping performed, adding more space to individual groups results in the maximum possible reduction in inter-group travel (and the equivalent mechanical activity). For small group sizes the distinction between the strategies is not

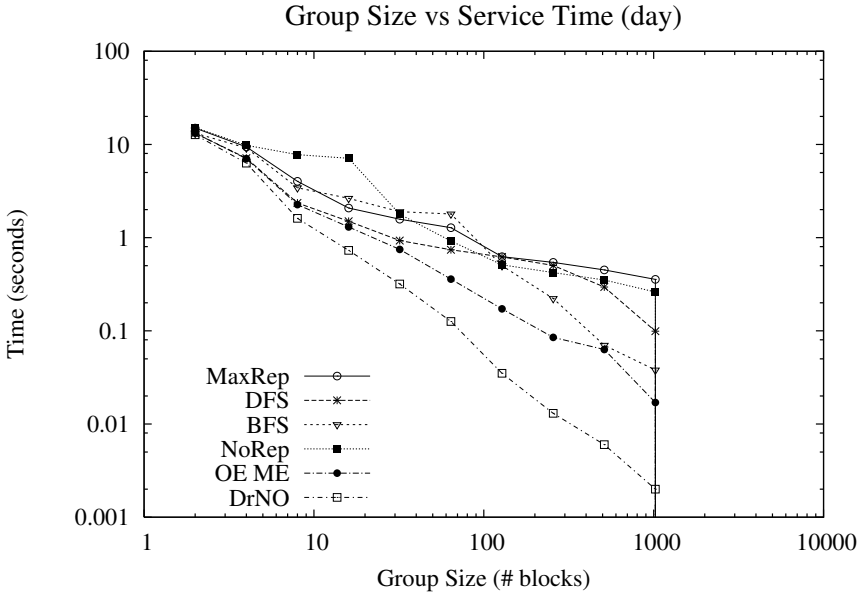
pronounced, but as group sizes grow the difference in the content of these groups becomes more pronounced, and the performance impact of better predictive grouping becomes more pronounced. *OE ME* does not follow the continuous improvement of *DrNO*, showing a leveling off of energy gains as group sizes increase. Nonetheless, these decreasing returns are much better than those of the competing strategies, suggesting that *OE ME*, while offering impressive energy gains, can be further improved upon. Such improvement would require better knowledge of the future, or a different method of group construction. *OE ME* is optimal in terms of group formation based on successor predictions, and can directly use any improved predictors that are developed. Improving the grouping mechanism would require a more complex algorithm that uses more than successor predictions, with the added complexity and metadata overheads that this implies.

The trends in Figures 5 and 6 were more pronounced as trace durations grew and were similar to the trends shown for the *hplajw* block trace shown in Figure 7. As the duration of a trace grows, predictive grouping has a greater chance to impact future performance. While *DrNO* has full knowledge of the future, there is no warm-up or training period for our *OE ME* algorithm, and so extended durations offer more time to learn and adapt to the workload's access patterns. It is interesting to note that aggressive replication can be detrimental to performance (as we see for the *MaxRep* results). In spite of using the same successor predictions as *OE ME*, the enthusiastic construction of groups for every context results in increased energy consumption for larger group sizes.

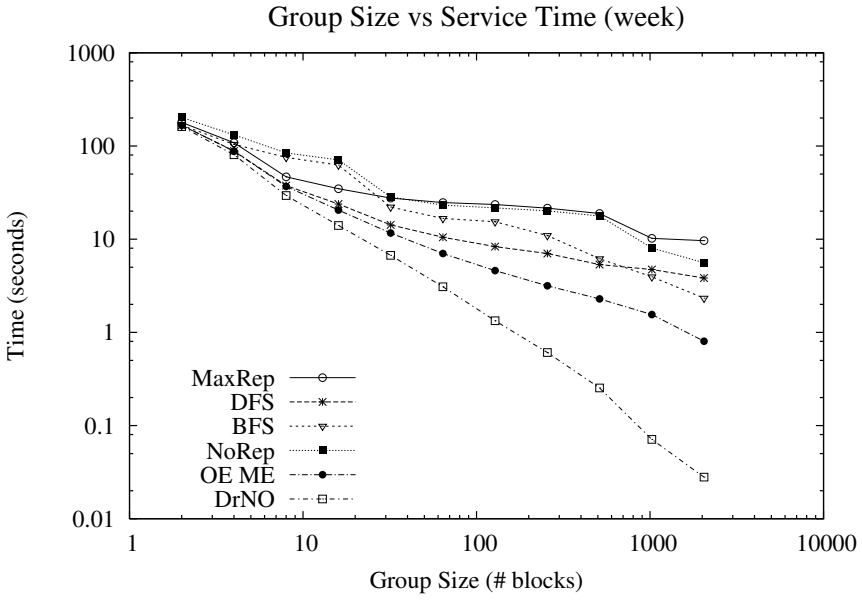
Figures 8, 9, and 10 show the latencies experienced for the *mozart* traces and the *hplajw* trace. In these figures we see a mirroring of the results for energy, once again we have over 99% reduction for the optimal grouping, while *OE ME* shows 70% reduction for larger group sizes. The correspondence of energy and latency results is expected, since the energy results are specifically for the mechanical movement in a disk drive, which are the primary component of access latency.

6.4.1 Group Formation, Access Behavior and Transitions

Fewer unique groups formed means a reduced usage of total storage space, as well as decreased likelihood of physical movement between these fixed-size groups covering large distances. The number of transitions is the number of times a workload resulted in a request for a group other

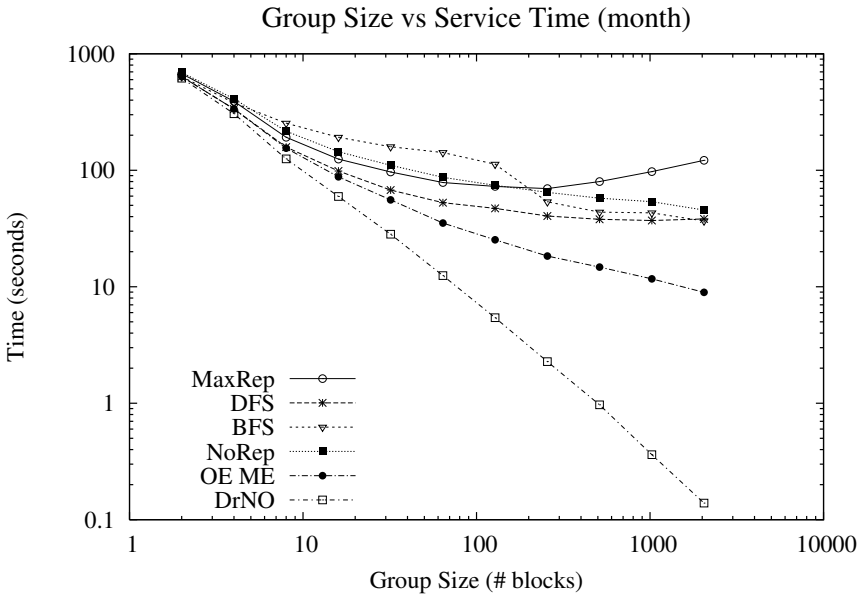


(a) *Day-long trace*

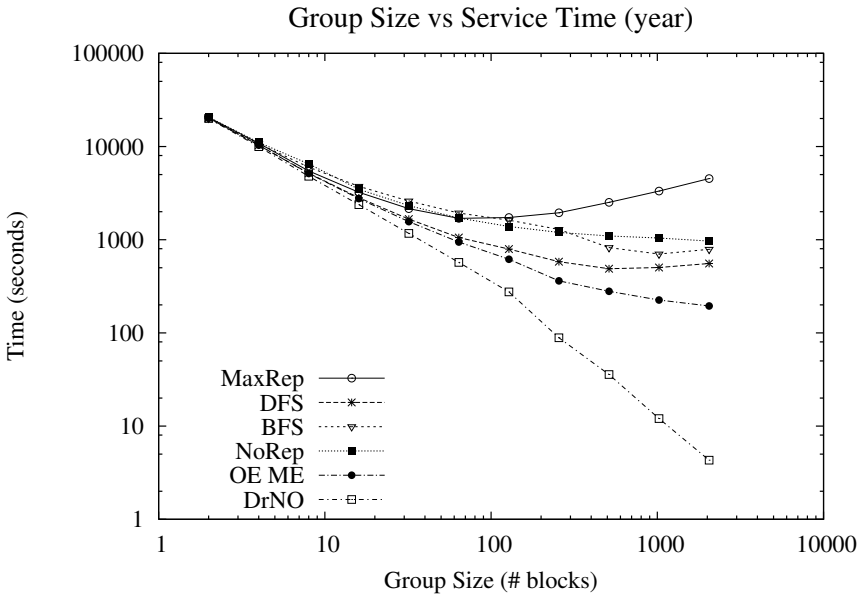


(b) *Week-long trace*

Figure 8: Total time delay (access latency) due to disk arm movement for the *mozart* day and week length traces. Note that the sudden drop off for the last group size in the shortest trace indicates the point at which all the unique files fit within a single group.



(a) *Month-long trace*



(b) *Year-long trace*

Figure 9: Total time delay (access latency) due to disk arm movement for the *mozart* month and year length traces.

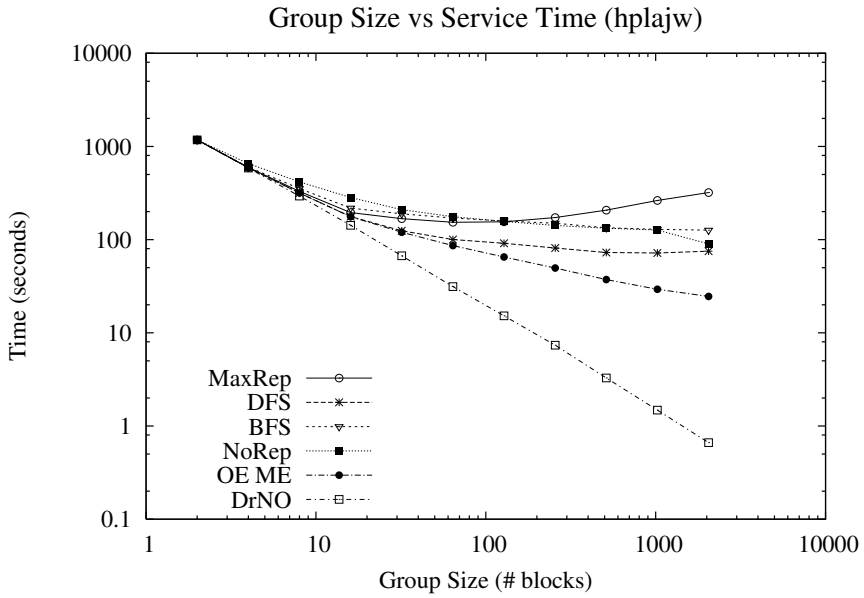


Figure 10: Total time delay (access latency) due to disk arm movement for the *hplajw* trace.

than that of the last request, reducing this number means a direct reduction in device activity. If groups are used as the unit of data retrieval, as in the case of the aggregating cache [7], then transitions correspond to the total number of read requests that a server will need to satisfy. In the case of data layout on a disk, if a group size corresponds to a track buffer size then the number of transitions equates to the number of disk requests that require physical activity. Applied directly to a disk's data layout, reduced transitions result in reduced mechanical activity.

Table 9 shows a comparison of strategies based on the number of groups formed and the number of transitions. As with Table 8 these numbers are for the *mozart* year trace, and a group size of 2048 blocks. It is interesting to note the behavior of the baseline algorithms. The maximum replication strategy forms a very large number of groups, and results in a large number of transitions (more than eight times the number for our *OE ME* algorithm, six times the competing strategies, and fifty times the optimal limit). This behavior is consistent with the poor energy and latency behavior of this strategy for large group sizes and long trace durations.

Table 9: Number of groups formed and total transitions. These numbers are for the *mozart* year trace, group size 2048.

Strategy	**Groups Formed**	**Transitions**
DrNO	**4311**	**4310**
OE ME	21916	143505
DFS	31318	368472
BFS	34147	554660
NoRep	**136**	20685059
MaxRep	277710	20546301

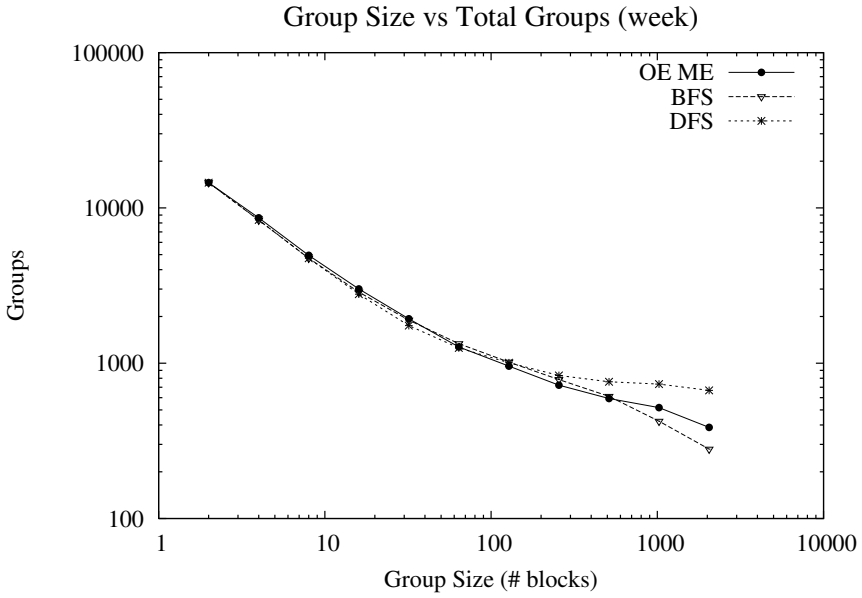
What is also interesting, and even more illuminating, is the behavior of the first baseline algorithm. The *no replication* (*NoRep*) strategy forms a very small number of groups, 136 groups compared to the optimal algorithm's 4311 groups. This is an excellent result in terms of space usage, and is to be expected since there is no replication of data. But without replication we see a number of group transitions that is even worse than the aggressive replication strategy. The question this table raises is how this baseline strategy can offer better energy and latency results than *MaxRep* in spite of this slight increase in total group transitions. The answer lies in the small number of groups formed, and a subtle optimization in data layout. While tempting to describe *NoRep* as a baseline equivalent to the static layout of data on a disk without optimization, a more accurate description would be that it is a static, yet optimized, layout of data. The block layout for *NoRep* was based on the initial request order for the data. This means that for each test workload, blocks were placed based on the order in which they appeared in the workload. This avoided penalizing the baseline algorithm for any artificially poor layout choices, such as the dislocation of metadata and its associated metadata. This inherent optimization accounts for the better than expected energy and latency performance of the *NoRep* baseline strategy, and the small storage footprint accounts for its tolerance of slightly higher transition rates than the maximum replication strategy.

In Figure 11 we see the number of groups formed for the competing probabilistic approaches. Our predictive grouping algorithm can be seen to produce the fewest groups, a trend that increases with lengthier workloads. This suggests that our algorithm avoids constructing superfluous groups, but we need to consider the number of transitions between groups.

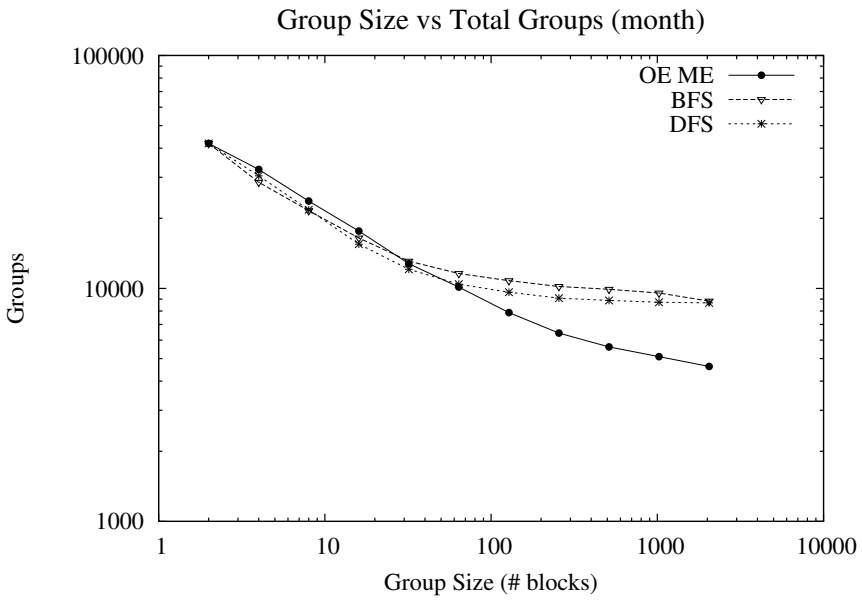
Our results confirm that of all probabilistic approaches, our balanced expansion (*OE ME*) has fewer groups formed and fewer transitions for non-trivial group sizes (shown in Figure 12). These results are consistent across all workloads. As the group size increases, it becomes increasingly important to strike a balance between replicating a block and simply moving it to another group. Too much replication will result in groups that contain too little variety and a large number of groups, thereby increasing the amount of movement between groups. Not replicating data sufficiently between groups will also result in unnecessary movement between groups due to blocks that are accessed with high overall frequency.

For lengthier workloads (*mozart* month and year, as well as *hplajw*), the maximal replication baseline strategy (*MaxRep*) shows diminishing returns more rapidly than other strategies. This supports our assertion that, while aggressive replication can be beneficial, it must be done intelligently. We see little or no improvement in number of transitions over the *no replication* strategy (*NoRep*). This confirms our suspicion that maximal replication is a poor grouping strategy, effective only for the collocation of small numbers of blocks. This maximum replication strategy is effectively building a group for every predicted sequence of length equal to the group size, and as the group size increases the number of such groups becomes excessive. This raises the question of the relative worth of different groups. A group that is useful will be accessed frequently, and most of its contents will be used. This brings us to the metric of *accesses-per-transition*.

In Figures 13 and 14 we see the average number of requests satisfied before a group transition is required. As group sizes increase, these requests are expected to decrease, with the rate of decrease being indicative of how effective the grouping strategy has been at building useful groups. With fewer groups, and a tendency to build more effective groups, the energy and latency performance of our predictive grouping algorithm is further explained. Fewer groups implies a reduction in overall distance traveled by the arm mechanism, while fewer transitions indicated a reduction in the number of inter-group "trips" that had to be made. With Figures 13 and 14 we see how these results correspond to more effective work (requests satisfied) before requiring a move to a new

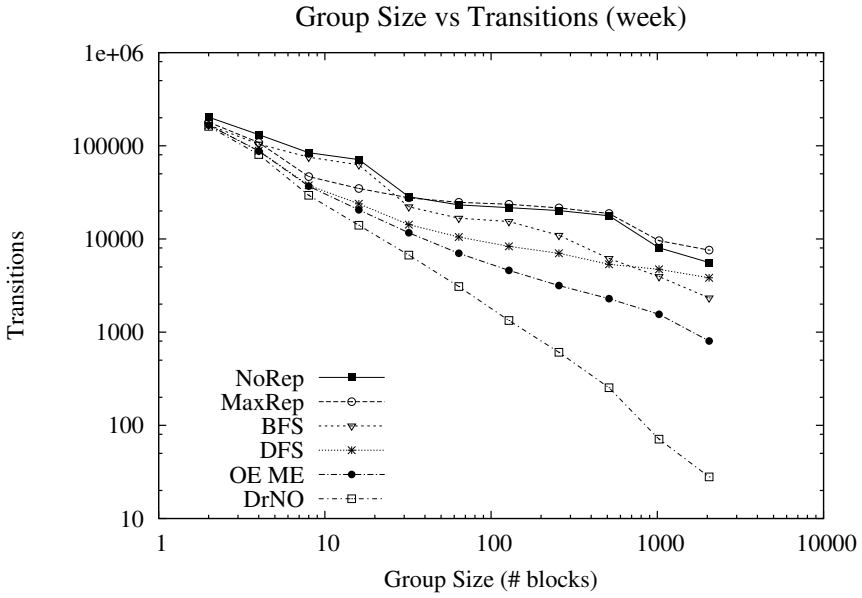


(a) *Week-long trace*

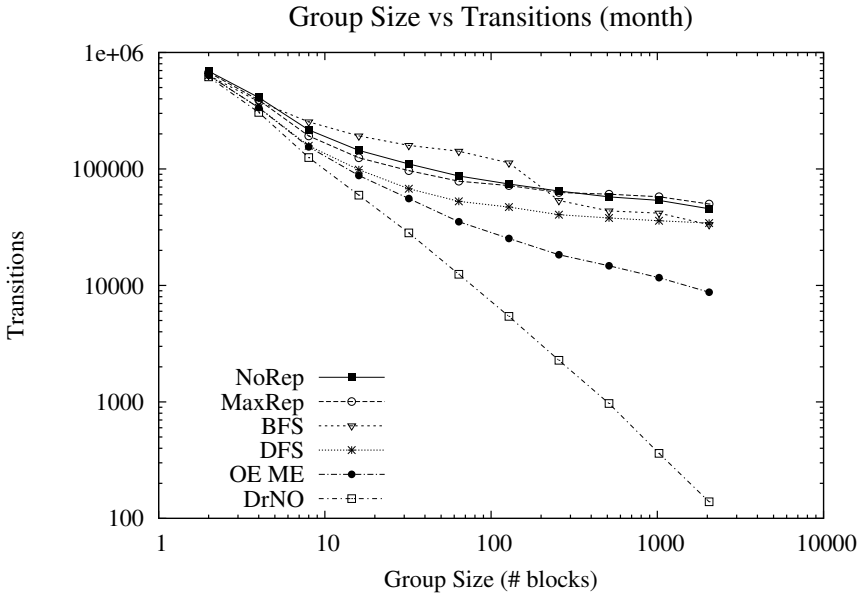


(b) *Month-long trace*

Figure 11: Total number of groups formed for the *mozart* trace.



(a) *Week-long trace*



(b) *Month-long trace*

Figure 12: Total transitions for the *mozart* trace.

group.

6.5 DISCUSSION

Throughout the majority of our work, our *Optimal Expansion, Maximized Expectation* algorithm forms the core of our prediction and grouping strategy. As we have discussed, this strategy employs a balanced approach to tree expansion using only the estimated likelihood of successor occurrence. No other information is used; data chunks, assumed to be blocks, are treated as unit sized pieces, and no consideration is given to how far two blocks are from one another. For the sake of discussion as well as completeness, we briefly present here two strategies, one that utilizes distance from the root, and another that utilizes variable file sizes.

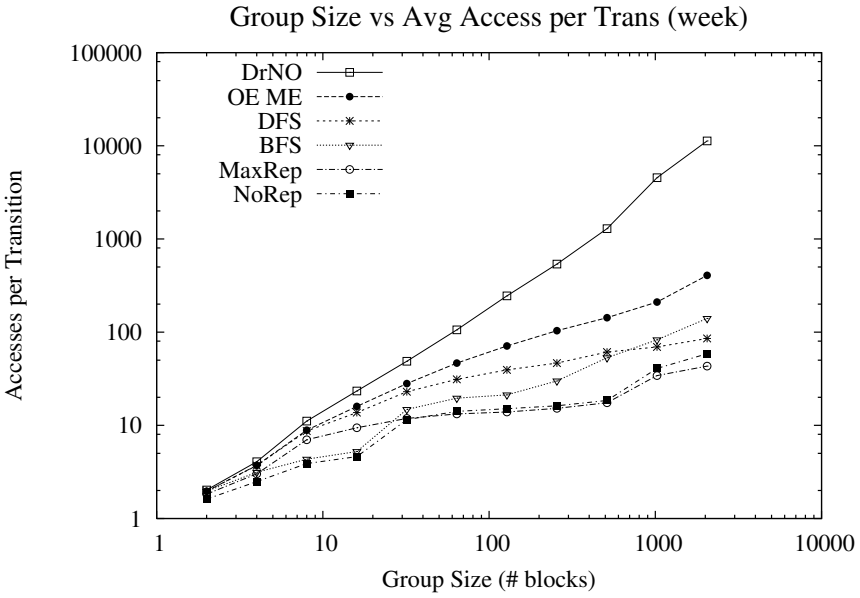
6.5.1 Optimal Expansion, Estimated Distance

In future chapters, we will discuss a number of ways to reduce "track distance", as well as transitions. It is worth noting that such distance concerns are not addressed in our *OE ME* algorithm. An early test we performed was to compare *OE ME* with a variant that included a distance metric. This algorithm, *Optimal Expansion, Estimated Distance*, or *OE ED*, is given in Algorithm 4, with its queue expansion function given in Algorithm 5.

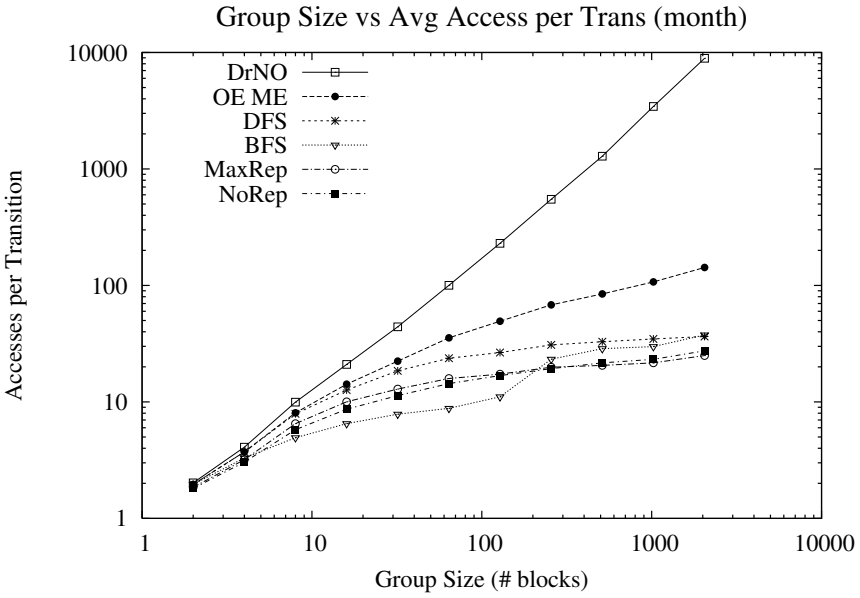
The key decision in this algorithm was the use of a sigmoid function in the priority queue. We use the following equation as this sigmoid function.

$$\frac{1}{1 + e^{-\frac{d-2E}{E}}} \tag{8}$$

Since distance has no known maximum without prior knowledge of a system, we need some way to bound an unbounded quantity. Further, once we establish the need for a "very long" seek, it is somewhat pedantic to discern between such very large distances; one "very long" seek is near the equivalent to another. Similarly, items that are "very near" our current location should not be harshly penalized for small differences in distance.

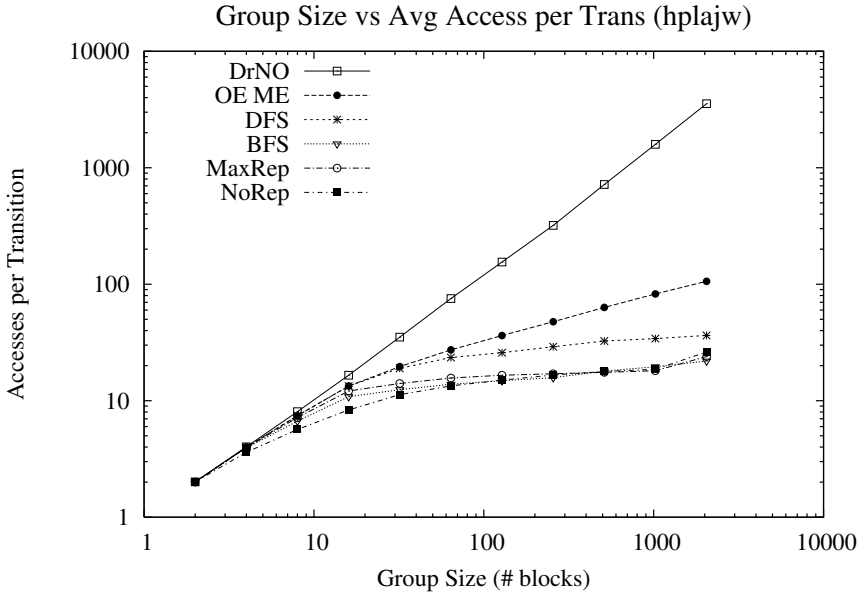


(a) *Week-long mozart trace*



(b) *Month-long mozart trace*

Figure 13: Average accesses per transition for *mozart* week and month length traces.



(a) *hplajw trace*

Figure 14: Average of accesses per transition for *hplajw* trace.

Our preliminary tests showed a noticeable sensitivity to the sigmoid function used. In particular, the algorithm seems to be sensitive to the constant E. With proper selection of this constant, *OE ED* had similar performance to *OE ME*, suffering only minimally in the number of transitions.

Incorporated in a dynamic setting, *OE ED* is expected to match or outperform *OE ME* in terms of *distance*, but not in *transitions* between groups, although the difference between them is expected to be negligible, even with a proper selection of sigmoid function. As a result, *OE ME* was chosen over *OE ED* for several reasons. First, there is no sensitivity to a constant, making *OE ME* more generalizable. Second, *OE ED* has a more complicated floating point calculation within the expansion function, which exists within the loop. Since our goal is a dynamic variant of prediction, we need to be concerned with reducing the computational overhead. Without visible improvements to both transitions as well as distance, the faster, more elegant computation *OE ME* was favored.

ALGORITHM 4 OE_ED$(T, root, max)$ - a balanced approach for forming a predictive group using a distance metric. The sigmoid function is given in Equation (8).

Input: set of first-order successor trees, T; a root ID, $root$; a maximum group size, max

Output: a set of IDs, G, representing the predictive group

```
ENQUEUE(max_pq, root, 1)
while ISNOTEMPTY(max_pq) and SIZEOF(G) < max do
    p ← TOPPRIORITY(max_pq)
    f ← DEQUEUE(max_pq)
    d ← ABSVAL(root − f)
    c ← p/ SIGMOID(d, E)
    if SIZEOF(G)+ SIZEOF(f) ≤ max then
        ADDTOGROUP(G, f)
        max_pq ← OE_ED_EXPAND(T, max_pq, f, c, root)
    end if
end while
return G
```

6.5.2 Optimal Expansion, Estimated Storage Space

We mentioned in Chapter 3 that we restrict the bulk of our work for block-level prediction. However, it may be of interest to discuss the possibility of extending our work to the file level. The algorithm *Optimal Expansion, Estimated Storage Space*, or *OE ESS*, was developed for this reason. At first glance, this algorithm, given in Algorithm 6, looks almost identical to the algorithm for *Optimal Expansion, Maximized Expectation* from Algorithm 2. However, there are several key differences. First, the priority queue for *OE ESS* is a *min* queue, not a *max* queue as in *OE ME* and *OE ED*. Second, the priority queue must be able to store three pieces of data, not just two. Each node in the queue stores the file ID and the priority, just as in *OE ME*. But in *OE ESS*, the priority is *not* equal to the estimated expectation. Rather, the priority is the product of the *complement* of the file occurring and the file's size.

ALGORITHM 5 OE_ED_EXPAND$(T, max_pq, f, c, root)$ - expands the maximum priority queue used in Algorithm 4. Note that P(T, f, s) denotes a function used to calculate the estimated probability of child s of f's access tree within T. The sigmoid function is given in Equation (8).

Input: set of first-order successor trees, T; a maximum priority queue, max_pq; a file or block ID, f; an estimated probability, c; a root ID, $root$

Output: max_pq

for all s such that s is a child of f in T **do**

$c \leftarrow c\times$ P(T, f, s)

$d \leftarrow$ ABSVAL$(root - f)$

$p \leftarrow c\times$ SIGMOID(d, E)

ENQUEUE(max_pq, s, p)

end for

return max_pq

If all file sizes are equivalent, this algorithm is functionally equivalent to *OE ME*. However, there are significant calculation costs involved, as well as increased memory usage. As we mentioned, one main goal of our work is to remain as generalizable as possible, hence this strategy was not explored beyond the algorithm development, as we focus on block prediction rather than file prediction.

ALGORITHM 6 OE_ESS$(T, root, max)$ - a balanced approach for forming a predictive group using variable size files. Note that the priority queue function ENQUEUE(min_pq, f, c, p) stores ID f and confidence, or probability, c, while using the priority p for ordering within the queue. The function TOPPROBABILITY(min_pq) returns the confidence c of the top node in the queue, not the priority. This is a key distinction; *OE ME* makes no distinction between confidence and priority. The function DEQUEUE(min_pq) simply returns the file ID f from the top node.

Input: set of first-order successor trees, T; a root ID, $root$; a maximum group size, max

Output: a set of IDs, G, representing the predictive group

ENQUEUE$(min_pq, root, 1, 0)$
while ISNOTEMPTY(min_pq) **and** SIZEOF$(G) < max$ **do**
    $c \leftarrow$ TOPPROBABILITY(min_pq)
    $f \leftarrow$ DEQUEUE(min_pq)
    if SIZEOF$(G)+$ SIZEOF$(f) \leq max$ **then**
        ADDTOGROUP(G, f)
        $min\_pq \leftarrow$ OE_ESS_EXPAND(T, min_pq, f, c)
    end if
end while
return G

ALGORITHM 7 OE_ESS_EXPAND(T, min_pq, f, c) - expands the maximum priority queue used in Algorithm 6. Note that P(T, f, s) denotes a function used to calculate the estimated probability of child s of f's access tree within T. Also note that the priority queue function ENQUEUE(min_pq, f, c, p) stores ID f and confidence, or probability, c, while using the priority p for ordering within the queue.

Input: set of first-order successor trees, T; a minimum priority queue, min_pq; a file or block ID, f; an estimated probability, or confidence, c

Output: min_pq

for all s such that s is a child of f in T **do**

$c \leftarrow c\times$ P(T, f, s)

$p \leftarrow (1 - c)\times$ SIZEOF(f)

ENQUEUE(min_pq, s, c, p)

end for

return min_pq

7.0 DYNAMIC GROUPING AND METADATA

Having studied the effects of static grouping strategies, the next step is to develop a dynamic grouper. This transition from static to dynamic boasts a number of benefits, but presents new challenges. We have already established the need for reducing the size of predictive metadata required by the storage system grouper; this was one of our driving motivations for choosing a first-order successor strategy. But the problem is compounded in dynamic grouping; indeed, managing metadata in general is becoming increasingly challenging [78, 79]. With static grouping, we apply the grouping algorithms on a system in a fixed state, and so can be done offline, or in applications where we do not wish to update our grouping decisions in response to workload changes. With dynamic grouping we aim to perform grouping decisions based on an ongoing workload, in an online manner, updating grouping decisions where necessary, yet continuously collecting and updating metadata. With such goals, it is essential for dynamic grouping to be highly optimized in terms of memory usage, disk space, and CPU cycles required. Every piece of a dynamic grouper must be compact, fast, and, in the case of metadata, easily retrievable. This necessitates the revisiting of our metadata problem.

In this chapter, we will detail several new data structures used for tracking our first-order successor metadata. These structures allow us to reduce the necessary volume of data by several orders of magnitude. Our goal is to have an efficient method of tracking this information regardless of block size. We will continue to refine our grouping strategies in following chapters. Static grouping strategies discussed in Chapter 6 organized groups as they were created, and chose *roots*, or block IDs upon which to begin predictive grouping, based on what block or chunk was requested next. Dynamic grouping requires knowledge of what block is likely to be requested. These challenges and our solution strategies are discussed in Chapter 8. In Chapter 9, we will tie these solutions together by describing our dynamic regrouper, *SPORe*, and detail how each previous solution is

applied.

7.1 MOTIVATION

Optimizing storage system performance in the face of varying workloads requires the accurate tracking and exploitation of patterns in data access behavior. Such information is useful for a broad range of applications, including caching, placement, workload shaping, data collocation and migration. Unfortunately, tracking access behavior and predicting future access behavior can result in large metadata demands. This is true when dealing with data at the granularity of files and objects, but quickly becomes unmanageable when attempting to monitor block-level access behavior in large storage systems. An explosion in metadata volume is doubly problematic when we consider that retrieving and updating such metadata can suddenly become an additional burden upon the storage subsystem. On the other hand, arbitrarily limiting the volume of metadata being maintained will only allow for optimizations to data within a current hotspot, the currently active working set, which is arguably less in need of pattern discovery and placement optimization (due to the effectiveness of even basic caching schemes on such subsets). This inevitably precludes the opportunity to discover longer-term patterns across less intensely active regions.

To improve the accuracy of placement and collocation decisions, and improve the overall performance of predictive analysis of data access patterns, we wish to maintain as much metadata as possible, but only if it is useful. Our previous work on predictive data grouping [34] (see Chapter 6) demonstrates one such strategy that stores a number of direct block successors for each data access. Our strategy shows promise in the area of data grouping, and is similar to previously explored strategies in prefetching and prefetch-caching strategies adopted by Kroeger *et al.* at the file level [71]. We present a study of how it is feasible to reduce the metadata requirements of our strategy in the face of block-level I/O workloads. The structures used in our work are reminiscent of the limited-length queue of access successors in the *Recent Popularity* strategy [8], also used in *EEFS* [80, 81]. Such single-successor strategies are better chosen for their efficiency benefits over multicontext modeling, yet still require huge amounts of storage. Minimally, we would need to track the root block's ID, which could simply be a translated location within an array, and the

queue of accesses, each of which is a block ID. Thus, the total storage space would be the number of successors stored, s, times the total number of blocks, t. For modern systems, this metadata volume is too large. For a 4 TB disk array, assuming a block size of 4 KB, this would mean storing information for 1 billion blocks. Assuming a 64-bit address, this system would require 8 GB of space *just for storing a single successor*. We address the issue of metadata volume requirements in *SESH* [35, 36] by observing that most blocks share two properties.

1. They only have a single successor.

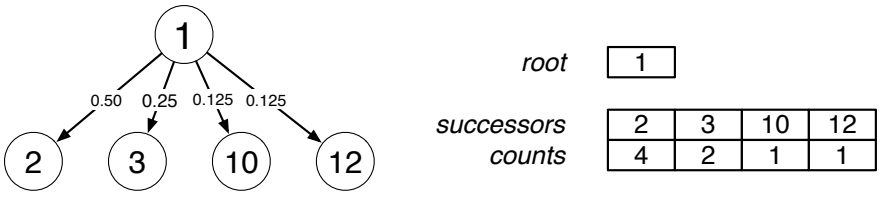
2. The only successor they have is the next sequential block.

Using this information, we are able to drastically reduce the total size needed for our predictive information while incurring little overhead. Further, our strategy scales better in the number of successors tracked.

7.2 EXPERIMENTAL SETUP AND DESIGN

Our goal is to develop space-efficient structures for tracking metadata, specifically for predictive information. Ideally, these structures would incur little to no overhead while maintaining undiminished usefulness. Further, we seek to define, in a general case, what the expected benefits of these structures would be. Finally, we endeavor to verify our expectations by testing working implementations against realistic workloads in order to determine how effective our data pattern exploitation techniques would be at reducing metadata volumes in real systems.

We have developed a novel mechanism for reducing the metadata storage requirements for predictive block-based metadata. Our approach was found to reduce such capacity requirements by more than 98%. We discuss this strategy by describing the structures we have developed as well as an estimated reduction formula. We then describe the different trace sets used to evaluate our current implementations. Finally, we detail our metadata volume calculations.



(a) *Visualization of OpExTree structure*

(b) *OpExTree implementation*

Figure 15: Optimal Expansion Tree (*OpExTree*) example.

7.3 DATA STRUCTURES

Several new data structures were designed for as components of our *SESH* structure. The *Optimal Expansion Tree*, or *OpExTree*, is the base structure used in our previous efforts (see Chapter 6) for tracking metadata for predictive data grouping. The *Dynamic Bitmap* is a functional equivalent to a normal bitmap, but with the advantage of being dynamically allocated and able to spontaneously grow or shrink. The *Dynamic Region* is used to map a fixed number of bits to some ID. Finally, the *SESH* structure is the combination of the above structures used to decrease the size of the necessary metadata. Following is a brief discussion of each structure.

7.3.1 Optimal Expansion Tree

Our standard metadata storage structure consists of a root ID, or the element's block number, and an array of immediate successors, or children. The structure is based on the *Recent Popularity* strategy from earlier work on predictive caching and prefetching [8], and was chosen for its robustness to signal noise and speed of adaptation to changing workloads.

Children are in the form of block numbers that occurred directly after the root ID. While our structure allows this array to be unbounded, we limit the number of children. Additionally, we track how often each child occurred.

Upon seeing a new event's successor, we add it to the tree by

1. Updating the appropriate count, or
2. Adding a new child to the successor array and setting the appropriate count to 1.

In the case of a bounded structure, once we reach the maximum number of children to track, we update the structure by choosing the lowest occurring successor and removing it from the structure. The new successor is then placed into the array and its count is set to 1. See Figures 15(a) and 15(b) for an example.

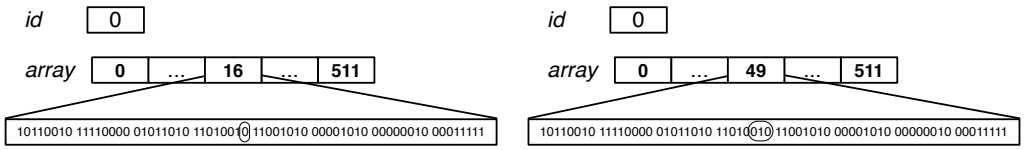
An alternate structure design replaces the successor array with a queue of children, in order of occurrence, representing an access history. Upon reaching the maximum capacity, a dequeue is performed before adding the new event. In this case, the counts are calculated by iterating through the queue. While this method will typically adjust to workload shifts easier, in practice we find the event counting to be a severe bottleneck.

A third alternate structure contains a queue as well as an array and counts. The queue is used in the same way as above, but dequeued items have their counts deducted, and are removed once their count reaches zero. In practice, we have found that our standard use of only an array very closely approximates this method, and the queue was removed from the standard version.

7.3.2 Dynamic Bitmap

The *Dynamic Bitmap* (see Figure 16(a)) structure consists of a count of total number of entries and a hash table of nodes. Each node consists of a simple integer array that represents a region of the functional bitmap. Three operations are possible on any location: *Set*, *Unset*, and *Check*. Each Set, Unset, or Check of any particular location is hashed and the appropriate node, if existent, is fetched. On a Set, the appropriate integer within the node's array is adjusted to update the map. If the node does not exist, it is created. Similarly, on an Unset, the appropriate integer is adjusted. If the Unset results in an empty node, equivalent to an array of all zeroes, the node is destroyed. On a Check, if the node does not exist, zero is returned. Otherwise, the appropriate bit within the existent array is returned.

As an example, assume we have a node consisting of 512 8-byte long long integers, and we are attempting a Check operation. The total number of entries in each node is equal to the number of bits; in this case, 32768, and each entry in the array, as an 8-byte integer, contains 64 bits.



(a) *Dynamic Bitmap, key 1055* (b) *Dynamic Region, key 1055*

Figure 16: *Dynamic Bitmap* and *Dynamic Region* examples.

Given below are calculations of the node ID (Equation (9)), the array position within the node (Equation (10)), and the bit location within the 8-byte integer (Equation (11)). Note that all are integer division operations.

$$id = key/total\_node\_size \tag{9}$$

$$ary\_loc = key/single\_location\_size \tag{10}$$

$$bit\_loc = key\%single\_location\_size \tag{11}$$

In our example, $id = 1055/32768 = 0$, $ary\_loc = 1055/64 = 16$, and $bit\_loc = 1055\%64 = 31$. In this case, we calculate our ID of 0, hash on that ID to retrieve the node, if it exists. Assuming existence, we calculate the array location of 16, retrieve the 8-byte integer, calculate the bit location of 31, and perform a bit shift and bit mask to retrieve the value. Thus, the overhead of a single Check operation is a three integer division operations, a hash table retrieval, an array retrieval, a bit shift, and a bit mask, all of which are very efficient.

7.3.3 Dynamic Region

The *Dynamic Region* structure is very similar to a bitmap. Instead of each bit being used to represent some property of some event, a number of bits are used. This is achieved by utilizing a *Dynamic Bitmap*, and for each event ID, we increment some region on the map. For our purposes,

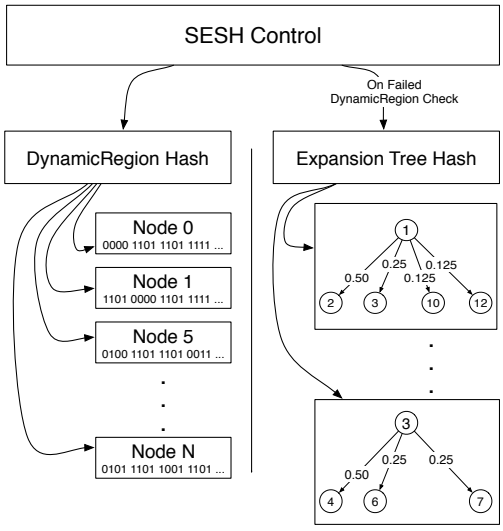


Figure 17: *SESH* figure.

we required only that each region denote a count, or integer. All analogous operations follow easily from the *Dynamic Bitmap* structure. The only change needed is that we must multiply the key by the number of bits stored for each region. Using our example from earlier, assuming 3 bits per region, $id = 1055 \times 3/32768 = 0$, $ary\_loc = 1055 \times 3/64 = 49$, and $bit\_loc = 1055 \times 3\%64 = 29$. See Figure 16(b) for clarification and comparison to the *Dynamic Bitmap* structure's analogous operation.

The computational overhead for the *Dynamic Region* is expected to be almost identical to the *Dynamic Bitmap*. Assuming that the region size is smaller than the number of bits in an array location, there are only two cases where significant differences occur. First, a single array operation may access two array locations, requiring an additional array position calculation and retrieval and additional bit location calculation. The other case involves node overlap, requiring and additional hash retrieval and array retrieval. Thus, in the worst case, we require two node ID calculations and hash retrievals, two array location calculations and retrievals, and n bit shifts and masks, where n is the number of bits in each region. Note that all of these operations are expected to be very efficient, and that the number n is expected to be quite small, usually 3 to 5.

7.3.4 SESH, or Space-Efficient Storage of Heredity

During our work on prediction and data regrouping (Chapter 6), we noted that many blocks have only a single successor. Most commonly, this successor happens to be the next block. The *SESH* data structure utilizes this observation by removing such *OpExTrees* from the successor table, typically a hash table, and utilizing a *Dynamic Region* to represent the tree. Some region being non-zero within the *Dynamic Region* structure represents a tree having only a single successor, which happens to be the block directly after the root block in question. We call the successors stored within the region *heir apparents*. These heir apparents occur the vast majority of the time (see Figure 18, and each reduces the amount of metadata required from (minimally) several bytes to only a few bits (on average). Thus, we have most of our metadata, that of all heir apparents, contained within a *Dynamic Region*, with the small remainder held within a successor table storing Optimal Expansion trees. See Figure 17 for clarification. As a realistic example, tracking eight successors (64-bit addresses, or 8 bytes) on a 256 GB hard drive with a block size of 512 bytes

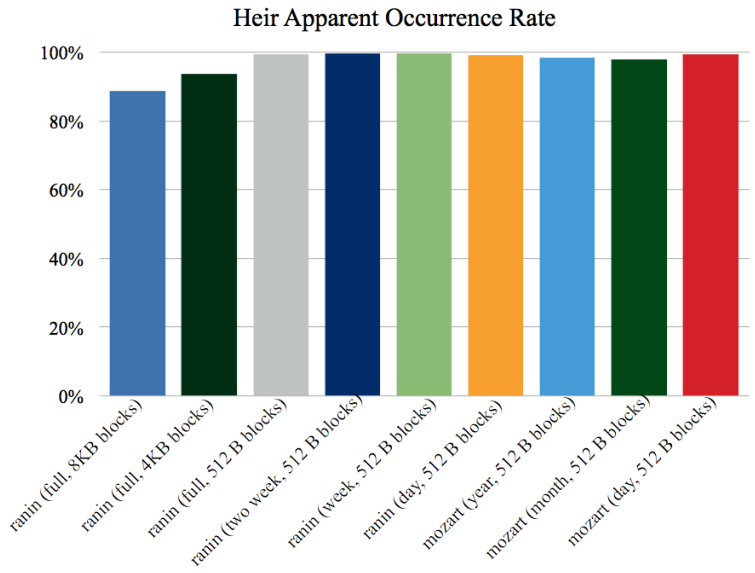


Figure 18: Rate of occurrence of heir apparents for various traces.

would require 32 GB of metadata.

$$8 \times 8 \times (256\,GB/512) = 32\,GB$$

However, each heir apparent would only require, on average, 3 bits. Given below is a estimated calculation for the reduced size, in bits, r, based on the number of blocks, b, the percentage of blocks that only contain heir apparents, p, and the number of successors tracked for each block, n.

$$r = b \times (log(n) \times p + (1 - p) \times (64 + (64 \times n)))$$

Note that this assumes 64-bit block numbers and ignores internal fragmentation within our *Dynamic Bitmap* structure. One note of interest presented by this formula is that when p is very high, the resulting size r becomes very scalable with respect to the number of successors, n. Since most blocks will be stored in the *Dynamic Region*, increasing n results in a $log(n)$ increase in the space necessary to store it. The larger structures increase linear to n. Even though these structures are

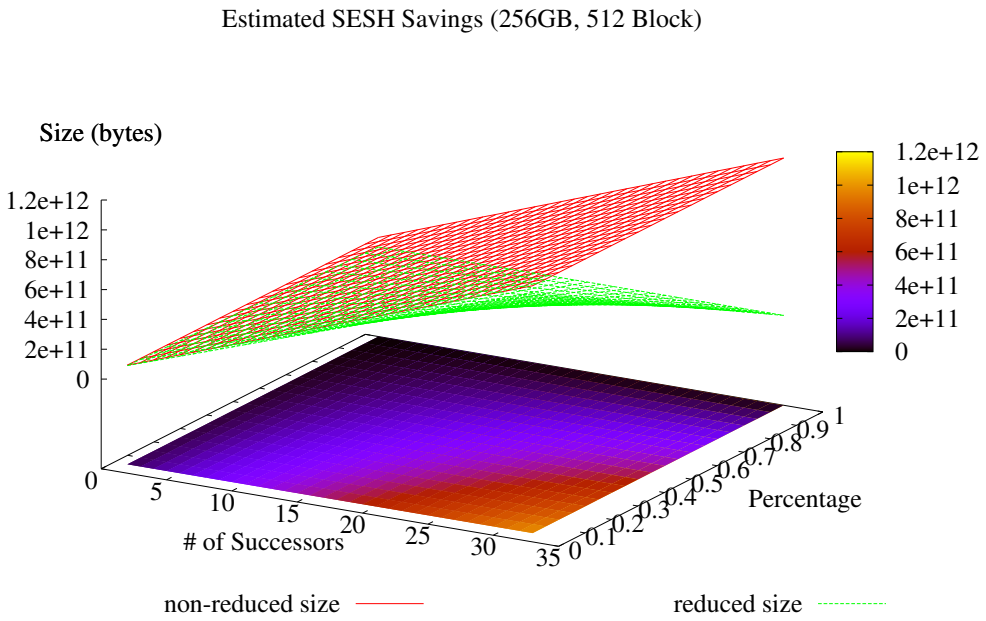


Figure 19: Estimated calculation of metadata storage space savings on a 256 GB hard drive with a block size of 512 bytes.

expected to represent only a small percentage of all items tracked, they *are* expected to dominate the space used fairly quickly. Figure 19 show a 3D plot of a 256 GB hard drive and the metadata required for storing information for all blocks, both before and after reduction, against the number of children tracked and the percentage of blocks that contain heir apparents.

The computational overhead of our *SESH* object's operations is expected to be quite small. The worst-case overhead is the sum of the worst-case overhead of a failed *Dynamic Region* operation and a hash retrieval of an *OpExTree* structure. However, most *SESH* operations will be a single *Dynamic Region* operation, as most blocks are expected to be heir apparents.

7.4 TRACES

In order to test the reductions of *SESH*, we used four different workload sets. The *mozart* set consists of a workstation trace gathered using the *DFSTrace* system [91]. These traces were converted into equivalent block-level traces with block sizes of 512, 4096 (4K), and 8192 (8K) bytes. There were four different original trace sizes; day length, week length, month length, and year length. This set has the appeal of allowing the analysis of our strategies over different definitive time periods as well as allowing us to convert easily to different block sizes.

The second set, *hplajw*, is a block-level workstation trace [102]. This set has the advantage of natively begin a block-level trace, and therefore does not require conversion. However, there is only a single trace length, and lacks any information of original file-system level access information, and therefore cannot be converted to traces of differing block sizes.

The third set, *ranin*, is a trace set we gathered using the standard `fs_usage` command found on Mac OS X. The traces were gathered in 2007 from November to December on a Mac PowerBook G4 running Mac OS X 10.4. The workload represents a typical graduate student workstation, and was used for internet browsing, file editing, code compiling, and running and testing experiments (predominantly C++ programs). While there were a few trace interruptions due to rebooting, including one major software update, the inaccuracies introduced would be negligible. Additionally, the software update had no impact on the `fs_usage` command itself, and any system-level workload shifts due to this update would represent realistic workload shifts experienced by users updating their operating system. Cache activity was gathered, but for these traces they were ignored; only device-level requests were used. These requests were in the form of read and write data and metadata as well as page ins and outs.

The final set, *playlist*, is a trace set gathered using the same `fs_usage` command. This set was gathered on two different Mac mini G4 workstations, each with 512 MB of memory and running Mac OS X 10.3.9. A playlist of 148 songs, with a runtime of approximately 14.8 hours, was run on each machine. Traces were gathered from August 31, 2008 to March 23, 2009, resulting in play counts over 300. All disk activity due to the mp3 software was isolated and recorded. One trace gathered information on a sequential playlist, while the other playlist was shuffled. These traces represent one extreme of predictability, an estimated upper bound on how predictable a realistic

workload could be.

Similar to the *mozart* traces, our *ranin* and *playlist* workloads include information about how large an access was requested, and therefore could easily be converted to equivalent block-level workloads. Perhaps the most interesting block size is 512 bytes, which is the natively preferred block size of the hard drives, both for the PowerBook and the Mac minis. However, we included runs on 4K and 8K block sizes for consistency.

Since the `fs_usage` command collects information on *all* devices, these traces do require a bit of attention to what raw device is being accessed. Some devices, such as `/dev/NOTFOUND`, were pruned. All devices that seem viable were included in the test run and mapped to a single device. This mapping was done by giving a 200 GB range to each device. Table 10 summarizes the devices found in the *ranin* traces and how often each occurred, as well as noting which of these were ignored.

Table 10: List of all devices found in the *ranin* trace set.

Device	Occurrence Count	Included?
`/dev/disk0s3`	3700000	Yes
`/dev/NOTFOUND`	570000	No
`/dev/disk2s1`	190000	Yes
`/dev/disk2`	73000	Yes
`/dev/disk2s0`	11000	Yes
`/dev/disk1s1`	950	Yes

All of these traces consist of data gathered from actual systems, and as such contain real-world predictability due to user, program, and system behavior, rather than being drawn from a distribution or synthetic function.

Table 11: Comparison of total space of all *ranin* traces.

Trace	Block Size (bytes)	# Blocks	Projected	Reduced
day	512	33,000,000	3.2 GB	37 MB
week	512	74,000,000	7.2 GB	77 MB
two week	512	99,000,000	9.6 GB	108 MB
full	512	120,000,000	11.6 GB	140 MB
day	4096	4,260,000	417 MB	41 MB
week	4096	12,000,000	1.1 GB	92 MB
two week	4096	16,400,000	1.6 GB	132 MB
full	4096	22,900,000	2.2 GB	180 MB
day	8192	2,150,000	207 MB	41 MB
week	8192	6,270,000	604 MB	91 MB
two week	8192	8,590,000	823 MB	131 MB
full	8192	12,200,000	1.1 GB	176 MB

Table 12: Comparison of reduction by percentage and savings of all *ranin* traces.

Trace	Block Size (bytes)	Savings	Reduction (%)
day	512	3.2 GB	98.9
week	512	7.1 GB	99.0
two week	512	9.4 GB	98.9
full	512	11.5 GB	98.8
day	4096	376 MB	90.2
week	4096	1.1 GB	92.2
two week	4096	1.4 GB	91.7
full	4096	2.0 GB	91.9
day	8192	167 MB	80.4
week	8192	513 MB	84.9
two week	8192	692 MB	84.1
full	8192	990 MB	84.9

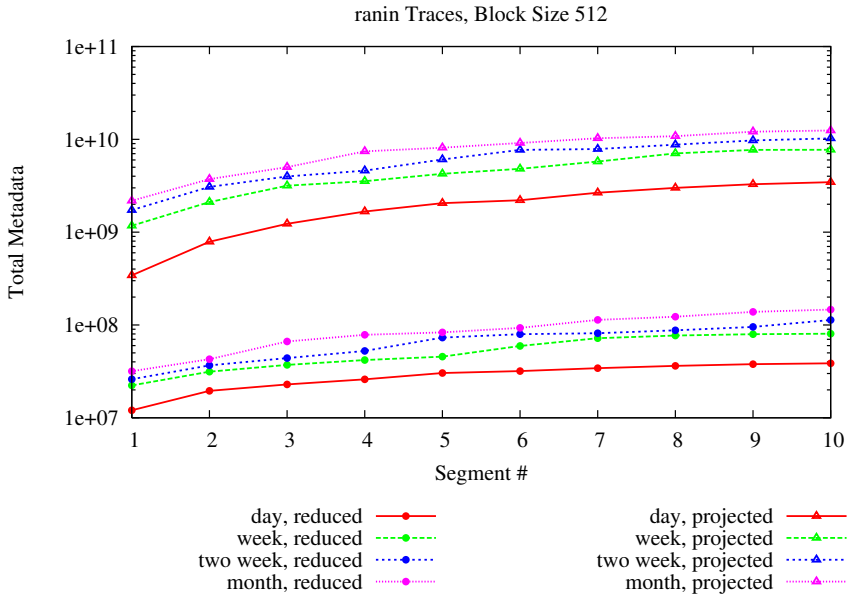
7.5 CALCULATING METADATA REQUIREMENTS

Each workload was split into ten sequential segments of approximately equal access counts. The trace was then run through our simulator. Each run consisted of the first segment, followed by running the first and second segments together, and so on until the entire trace was run. At the end of each segment run, the total metadata space used was recorded.

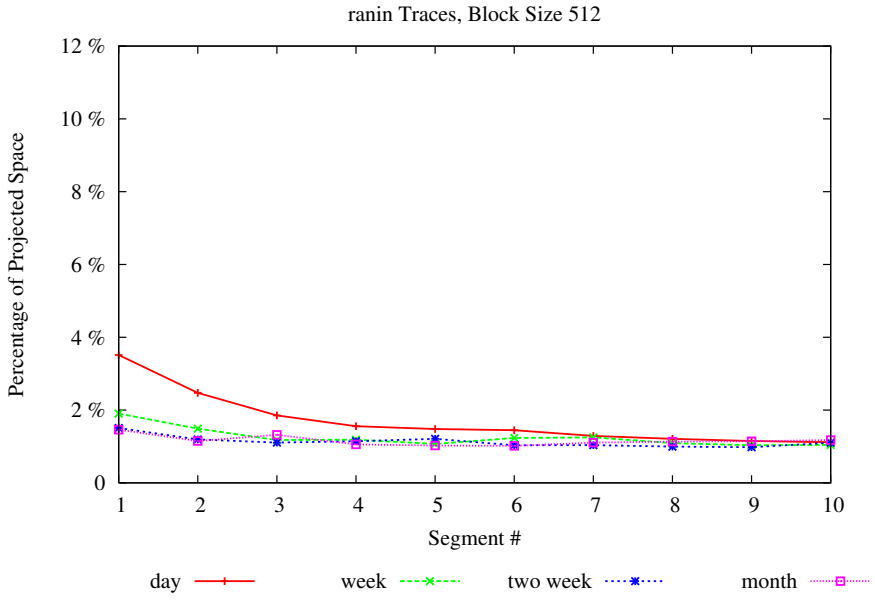
Each segment's metadata requirement consisted of the calculation of total space used by our *SESH* structure. This includes any and all extra metadata we used for sake of statistics gathering, though these extra object fields are negligible. In calculating these metadata requirements, we count all nodes of all *Dynamic Bitmaps* used in our *Dynamic Regions*, rather than estimating a number of bits per heir apparent as in Figure 19. In order to calculate the projected size of metadata using a hash table of *OpExTrees*, we multiply the number of heir apparents by the total size of the same number of single-child *OpExTrees* and add the appropriate hash table metadata needed to track the extra trees.

7.6 RESULTS

Our results show that almost all traces of non-trivial size show a drastic decrease in necessary metadata. For most workloads, we can reduce this storage space to only a small percentage of the original space, typically between 1 and 3 percent for smaller block sizes. Table 11 summarizes the sizes recorded at the very end of the *ranin* workloads, while Table 12 summarizes the reductions and savings. Figure 20(a) illustrates the difference between the projected metadata requirements and the reduced space on the *ranin* traces with 512 byte blocks, while Figure 20(b) shows the reduced size in terms of projected volume's percentage. Figures 21(a) and 21(b) show the respective results for the *mozart* traces, again with 512 byte blocks. The *hplajw* trace showed results similar to these 512 byte block traces, with reductions falling between 91% and 97%. The interesting difference is that the *hplajw* trace does better early on, then quickly falls to 91% reduction before flattening out. The *playlist* traces showed reductions similar to the *ranin* workloads, exceeding 98% reductions for small (512 byte) blocks.

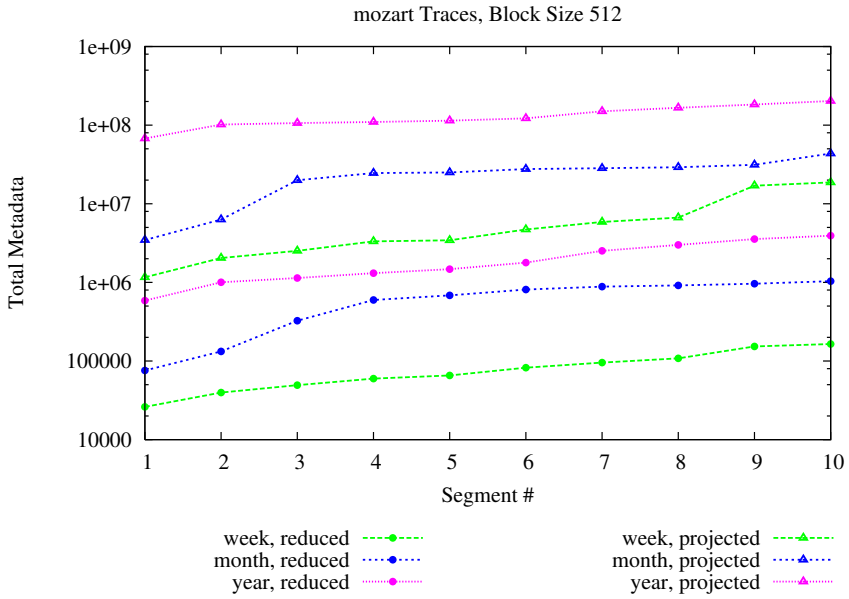


(a) *total space*

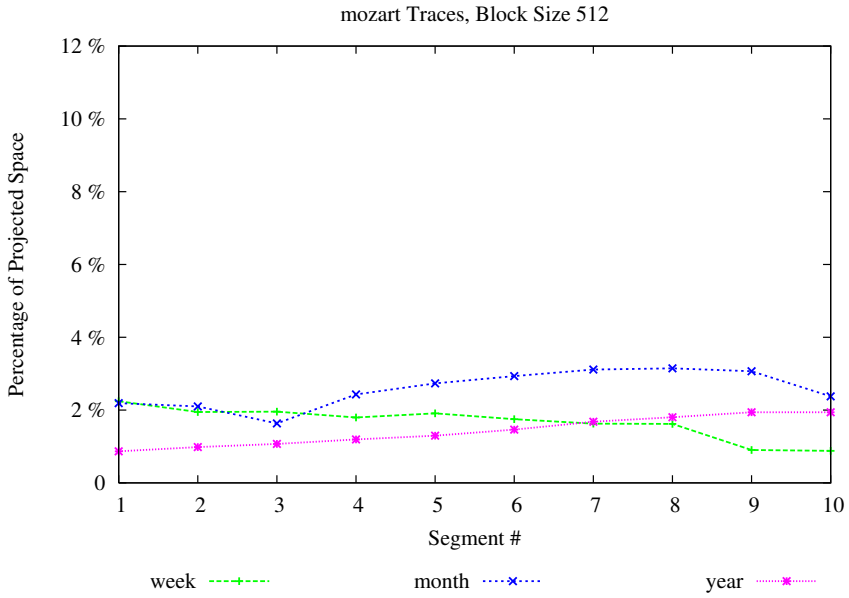


(b) *space reduction*

Figure 20: Comparison of total projected metadata storage versus reduced storage for all *ranin* traces with 512 byte blocks.



(a) *total space*



(b) *space reduction*

Figure 21: Comparison of total projected metadata storage versus reduced storage for various *mozart* traces with 512 byte blocks.

Figure 22 shows the amount of space that *SESH* requires for a representative selection of our traces, as a percentage of the total storage volume. Note that the total amount of space across all traces and block sizes is less than half a percent. Also notice that the actual space required by *SESH* is higher than our estimate. However, this is not unexpected, as our implementations of *OpExTrees* keeps additional information than what is accounted for in our estimate.

As expected, larger traces have more consistent requirements for metadata storage. Smaller data sets would not adequately capture the larger picture, and would have new blocks introduced quite frequently, while larger sets would add only the occasional new block.

An interesting result is that total required storage space, after reductions, is reasonably consistent across block sizes, varying only by about 20%, while the total number of blocks increases by a factor of 10 to 15, depending on block size. For instance, the full *ranin* trace, at roughly a month in length, requires about 150 to 189 MB, depending on block size, while the total number of blocks increases from about 12 million (for 8 KB blocks) to 119 million (for 512 byte blocks). It is also interesting that for *reduced* sizes, it is the middle block size (4096 bytes) that requires the most space. As expected, the smallest block size has a much higher reduction rate, as it would exhibit a far greater amount of predictability, while the largest block size has far fewer blocks to track.

7.7 DISCUSSION AND POSSIBLE ENHANCEMENTS

The application of storage prediction greatly depends upon the efficient management of supporting metadata. We have described a novel method for greatly reducing such a volume of first-order successor information. Our introduced structure, *SESH*, in addition to requiring minimal, fast operations for its implementation, dramatically reduces the memory demands of the metadata, two key complementary features allowing highly optimized and efficient metadata tracking.

An interesting augmentation to our *SESH* structure follows directly from the observation that the *Dynamic Region* structure, used to track successor trees with only the next sequential block ID as a successor, is actually more potent than presented. We are able to use these successor trees to look up candidates for the next block ID, but to look up candidates for *previous* block IDs, we have to perform an exhaustive search, looking at each block ID's children for a match. At

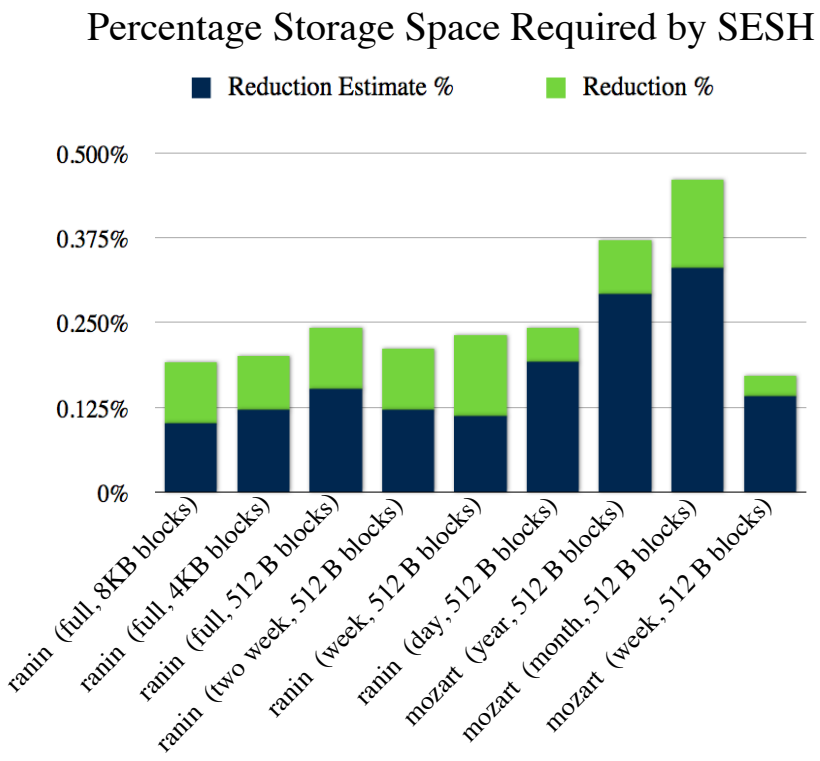


Figure 22: Percentage of total storage volume that *SESH* requires, compared to the estimate.

first glance, a request for the previous block ID seems superfluous; that request has already been satisfied. However, several interesting applications arise from extending that request. Knowing all previous block IDs for the last n accesses is the very definition of a working set, a set that caching techniques attempt to replicate. Further, knowing not only what the working set is, but the exact *sequence* created it has appealing application as well. Any static attempt to regroup data on the storage device, easily exemplified by defragmentation, would negate dynamic attempts to replicate predicted blocks. However, knowing the sequence of requests leading up to the time of defragmentation, if large enough, would allow for replication opportunity with the new data

grouping. The system could "replay" the sequence, make its decisions on where and what to replicate, and carry out those replications, all as part of the defragmenting process.

These applications rely on the ability to recreate large sets and sequences. Our *SESH* structure can easily be modified to allow such recreations. Since most blocks are only ever succeeded by the next sequential block ID, it follows that most blocks are only ever *preceded* by the *previous* sequential block ID. Thus, our heir apparents may be modified to contain only these block pairs. We call these objects *apparent pairs*. Tracking these pairs follows directly from our original *SESH* design. However, the obvious question that follows is what to do with the block pairs that do not fit this new criterion. We resolve this by keeping our old successor table, storing all block successor information not found in the apparent pairs, and adding an ancestor table, storing Optimal Expansion trees that store a block's predecessors. Thus, to get the successor candidate list for block b we check the apparent pairs for b's entry being non-zero. Upon failure, we retrieve the Optimal Expansion tree from the successor list. To get the ancestor candidate list for the same block, we check apparent pairs for $b-1$, and upon a failure, retrieve the expansion tree from the ancestor list.

The addition of our ancestor table, coupled with the new restriction on our apparent pairs structure, effectively doubles the amount of necessary metadata, but allows for straightforward recreations of our working set and sequence. We begin by checking the predecessor candidates for the last requested block ID, p. If $p-1$ is in apparent pairs, then $p-1$ has to have preceded p the last time p occurred. However, the value returned from the request to apparent pairs, n is the total number of known times that $p-1$ preceded p. Thus, we need to track how many times in the reconstructed sequence we have encountered each block. If p is not in apparent pairs, we check the Optimal Expansion tree from the ancestor table. Recall from Section 7.3 that an alternate version of our Optimal Expansion tree included a successor queue, rather than an array. This queue contains the last n successors of the block. Thus, the last item, at the *back* of the queue, was most recent successor for the block, and is added to the reconstructed sequence. Note that we need to track how many times each block occurs in our sequence, just like for those blocks that exist in our apparent pairs. Working this way, we can "unravel" the sequence, up to the point where we encounter a block b with n total predecessors (not necessarily unique), where we have encountered block b in the reconstructed sequence n times previously. This gives us the maximum size working sequence, and reconstructing the working set from this sequence is trivial.

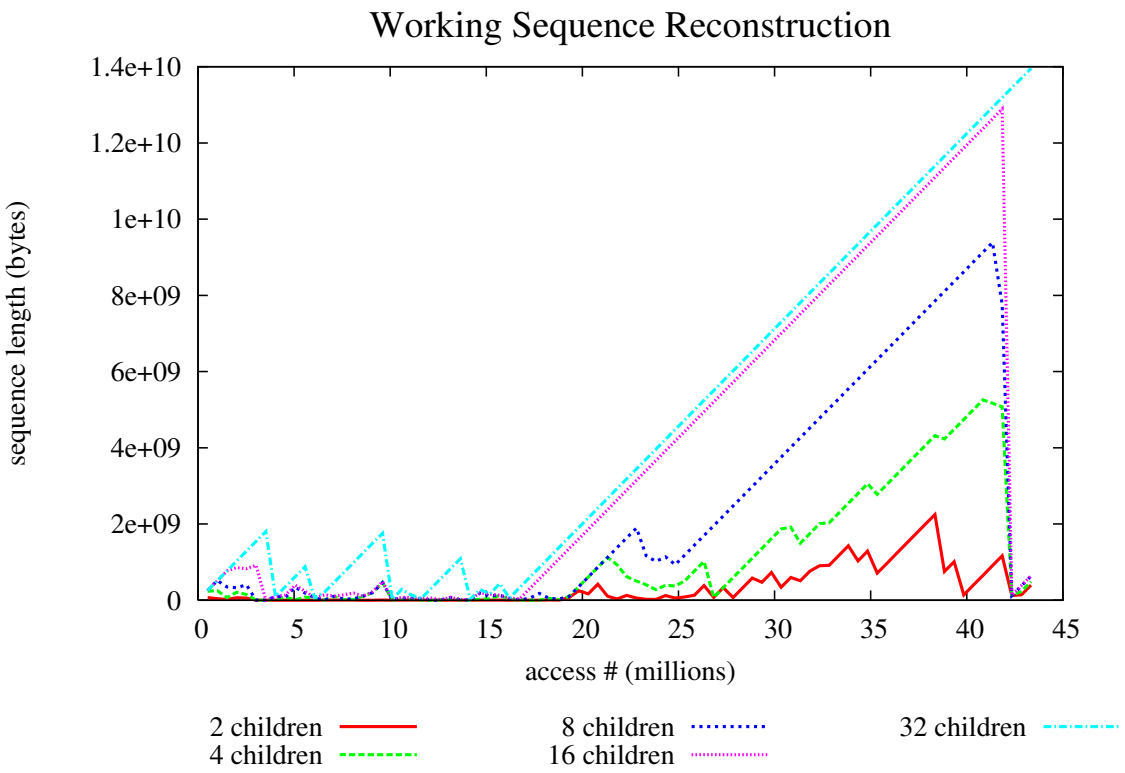


Figure 23: Total size, in bytes, of the reconstructed sequence for the *ranin* day trace by access # (block size 512 bytes).

Our initial trials on this strategy indicated that there were many instances in the workload where the maximum size of the reconstructed set and sequence dropped off very suddenly, but we could often reconstruct large chunks of both set and sequence perfectly. Figure 23 shows the total number of bytes we were able to reconstruct for the sequence of the *ranin* day trace (512 byte blocks), based on the number of children tracked, while Figure 24 shows the percentage of the total encountered volume. Figures 25 and 26 show the byte and percentage figures for the reconstructed set.

These initial results show a great deal of promise for reconstructed set and sequence application, but due to time constraints, we chose to continue our focus on dynamic regrouping. However,

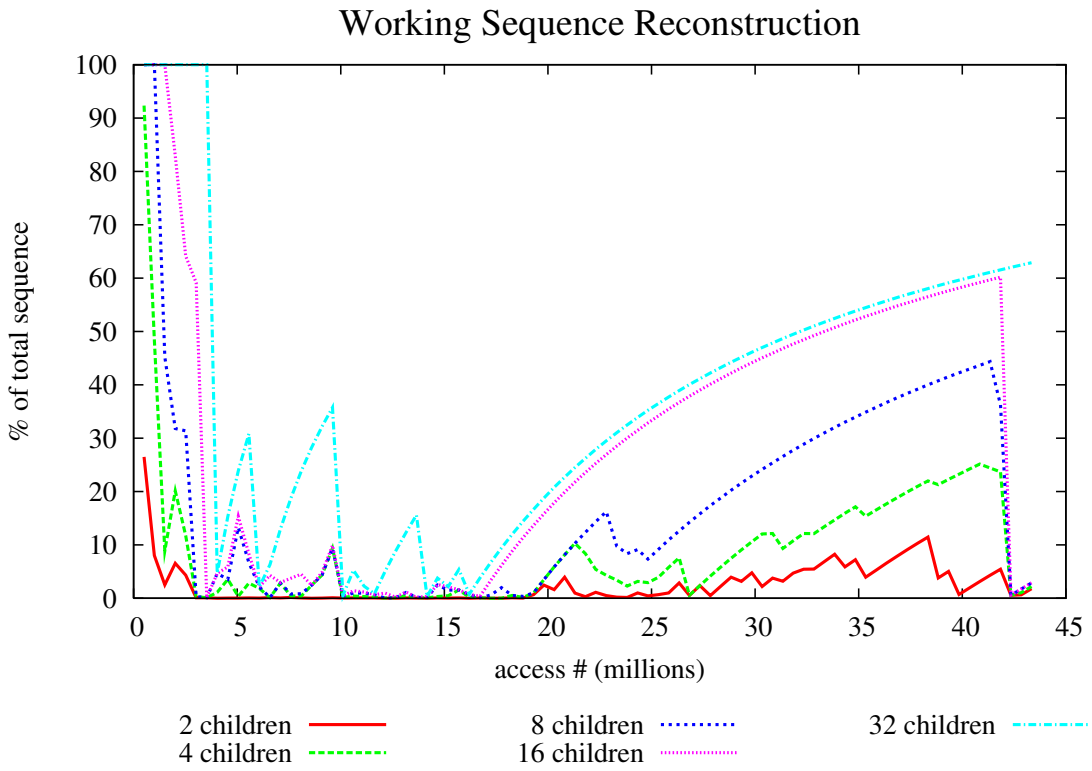


Figure 24: Total size, as a percentage of the total volume, of the reconstructed sequence for the *ranin* day trace by access # (block size 512 bytes).

it should be noted that, for the remainder of our research, we actually utilized this augmented structure, including the queue version of our Optimal Expansion tree, in the interest of facilitating future integration of these reconstructing strategies.

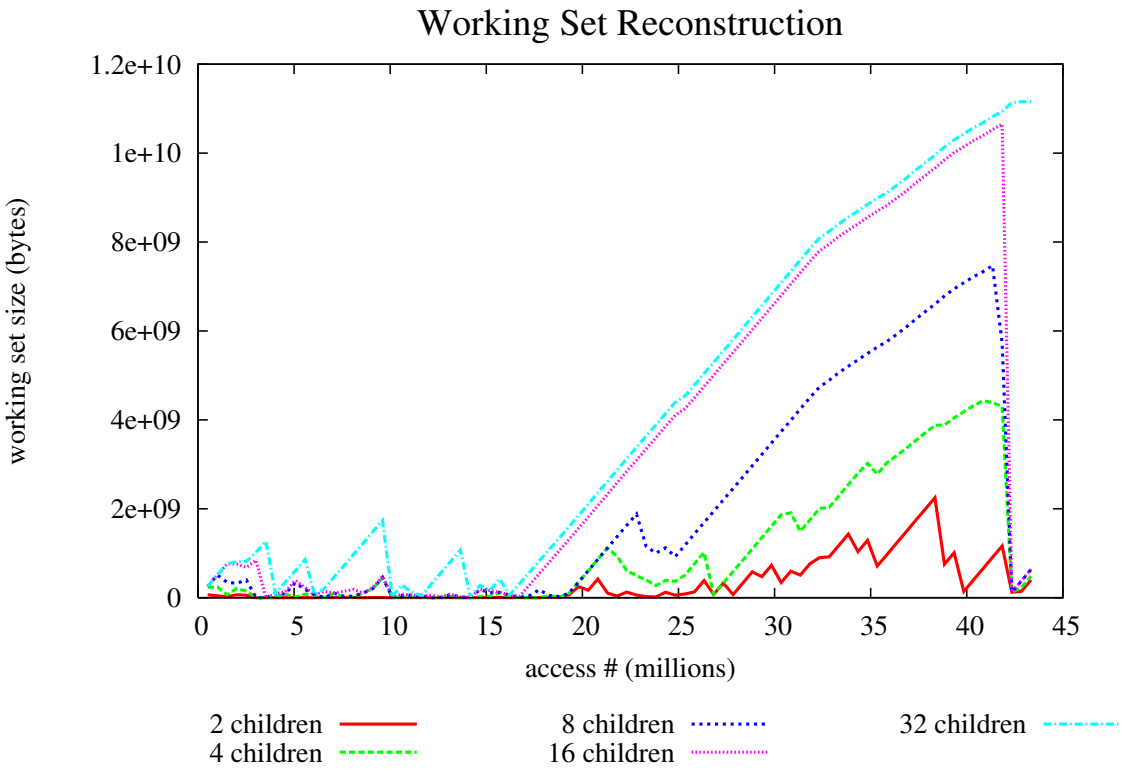


Figure 25: Total size, in bytes, of the reconstructed working set for the *ranin* day trace by access # (block size 512 bytes).

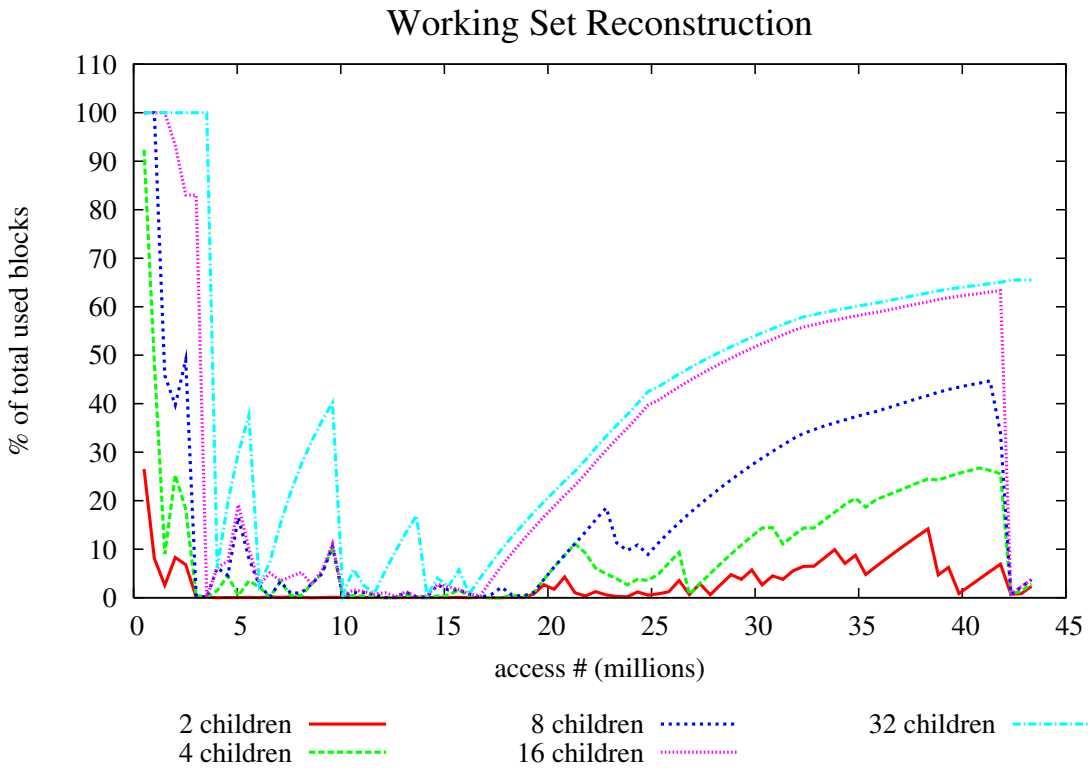


Figure 26: Total size, as a percentage of the total volume, of the reconstructed working set for the *ranin* day trace by access # (block size 512 bytes).

8.0 ROOT SELECTION

In this chapter we describe our strategy for choosing which blocks constitute good selections as predictive group roots, or starting points for prediction. We have presented grouping strategies for identifying what data needs to be collocated, and have enhanced these strategies to be effective in an online manner, as well as reducing their metadata overheads to allow their efficient application at the block level. While these techniques identify good choices for collocation, a practical decision needs to be made about which items to collocate first, a ranking of starting points for groups, what we call the "root selection" for the groups.

As we have previously mentioned, disk layout maintenance allows for the decoupling of strategy from the data path. As a result, we would be able to shut down regrouping efforts, presumably during heavy workload periods, while still benefiting from earlier efforts. However, such decoupling introduces new challenges. In caching, as well as other strategies lying directly on the data path, we are typically given a block address, and decide what actions may be taken. Notice that we are *given* a starting point for our scenario because we are operating on the data path. Such is not the case in disk layout maintenance; we must *find* such a starting point to act upon.

Finding these starting points, or roots, is non-trivial. Many strategies that immediately present themselves may easily fail. For instance, we may choose a block that occurs very often. However, if that block is very well placed, a predictive group might yield little benefit, if any at all. But if we choose a block that is very poorly placed, but does not occur very often, we may never have opportunity to use the predictive track.

8.1 MOTIVATION

There are several distance metrics that can be used to estimate the "cost" of a disk access. The most straightforward method is *block distance*. This metric is simply the distance between the previous block address requested and the current block address. A similar metric is *track distance*, defined in Equation (4). A third metric is a simplification of track distance, which counts only the number of transitions caused by an offending block.

Additionally, there are several properties that would be highly desirable among structures used to track any set of potential roots. The most obvious is some way to sort potential roots so that events that have higher potential benefit might be chosen first. Secondly, it must be adaptive. Extrapolating from caching and memory management strategies, we can guess that global information is important, but should be skewed towards more recent trends. Thus, our distance metrics should be tempered with an aging strategy.

Most likely, we will need to set a maximum size for this information. The easiest way to do so is to treat the structure like a cache; once it reaches a certain maximum capacity, items are evicted to make way for newer items. Thus, it is desirable to have a high number of "hits" in our structure. Just like a cache hit, this occurs when we encounter an object in our workload that is contained within the structure. However, in our case, we have an additional concern. Usually in caching strategies, the most crucial statistic to maximize is the number of hits. In layout management, we not only want to maximize the number of hits, but we want to minimize the number of requisite updates. It does no good to have a predictive group used if we the cost of updating outweighs the benefit of its use. Thus, it is desirable to have a strategy that, while not static, exhibits stability. This stability would allow us to make choices about roots with a higher confidence that they will remain good choices.

8.2 DATA STRUCTURES

We have observed that finding roots, upon which predictive groups can be formed, is crucial to our predictive disk layout strategy. But how does one design a structure for capturing what constitutes

a good root? How are we to capture distance, as well as recency? How can we obtain adaptable, but stable, predictive roots?

The first mechanism one might consider is a strict *LRU* structure. This presents an easily sortable structure that captures recency well. However, it does not capture frequency. A Least Recently Frequently Used mechanism [75], or *LRFU*, that contains simple aging, is a reasonable alternative. But such aging techniques very quickly become prohibitively expensive with large data sets due to large priority queues. In order to obtain the benefits of a strict *LRFU* structure without paying high computational costs, we used a hot list structure, similar to the frequency tracker used by Deng [25], first detailed as the Segmented *LRU* strategy (*SLRU*) by Karedla *et al.* [63]. These strategies use a dual *LRU* structure, one as a recency list, and one as a hot list that items are promoted to out of the recency list. Our structure utilizes a fixed-size *LRU* recency list, or "filter", and an *LRFU* hot list of the same size. Our attempt to balance between recency and frequency is also similar to efforts made in ARC [87], where two distinct lists are merged, biasing towards the list that would, in pure form, provide better performance. We further expand on the base *LRFU* scheme by using a distance metric as the score addition to the *LRFU* structure. We call this altered strategy Least Recently Distantly Used (*LRDU*).

In order to test our *LRU*, *LRFU*, and *LRDU* structures, we tested "pure" strategies against each other and against their hot list counterparts, as well as against two "best case" strategies. Following is a brief discussion of each structure and strategy tested.

8.2.1 Highest Count

The *highest count* structure, or *highest count* array, is simply an array of ordered pairs, (b, c), containing each event's block ID, b, and a count, c, of how many times in the workload the event occurred. For ranking purposes, this array was kept sorted by count. While computationally unrealistic to implement in a real system, this structure provides a reasonable target for our structures.

8.2.2 Highest Distance

While tracking how often an event occurs might be a reasonable strategy, tracking the block distance or track distance caused by each event would prove more beneficial. For our purposes, if

block A is followed by block B, then the distance $|A - B|$ is said to be caused by B, since the storage device must move from A to B. Another way to clarify this is to observe that A has been satisfied, and thus has already "caused" its associated cost, while B is an outstanding request. Thus, we can define distance of each event, similar to Equations (1) and (4), where B is preceded by A.

$$dist(B) = |A - B| \tag{12}$$

A similar design using track distance for block A in track α to block B in track β would have block B's distance as follows.

$$dist(B) = |\alpha - \beta| \tag{13}$$

This equation has the additional benefit of reduced risk of overflow errors, since track distances are much smaller than block distances. Additionally, there is little to no overhead incurred, since we need to calculate the respective tracks for determining whether a transition has occurred.

Using this definition of distance, we kept an array of ordered pairs, (b, d), containing each event's block ID, b, and the distance caused by that event, d. Similar to our *highest count* array, our *highest distance* array was sorted by distance for ranking purposes.

8.2.3 LRU and LRU Hot List

We utilized a structure nearly identical to Deng's frequency tracker [25], containing an *LRU* "filter" or "recent list", as well as an *LRU* "hot list" of the same size. Upon an event's request, we check the filter as well as the hot list. If the item is absent from both, it is added to the filter. If the item exists in the filter, it is promoted to the first rank within the hot list. If the item is in the hot list, it is moved to the first rank. Once the hot list is full, upon an event's promotion, the lowest item is popped off and demoted to the filter's first rank.

An important clarification for the hot list structures is that the entire structure's *size* is determined by the sum of the size of its parts, with each part, both filter and hot list, having equal size. A *hit* is defined as a requested item being in the *hot list*; an item's existence within the filter constitutes a miss. A miss is also generated if an object is not present in either the hot list or the filter.

8.2.4 LRFU and LRFU Hot List

Our *LRFU* structures utilize an aging scheme that closely resembles the *NFU* strategy detailed and discussed by Tanenbaum [117]. Upon an event's request, we check the structure to see if the event exists. If it does not, an ordered pair (e, s) is added to the structure, where e is the event's unique ID and s is the default "new addition score". If the item already exists within the structure, the score is updated with a default "event addition" score that is slightly smaller than the new addition score. If the structure exceeds the maximum number of ordered pairs, the item with the lowest score is removed.

Each request for an item generates a structure "clock tick". Upon reaching a pre-determined maximum number of ticks, all scores are halved by bit shifting.

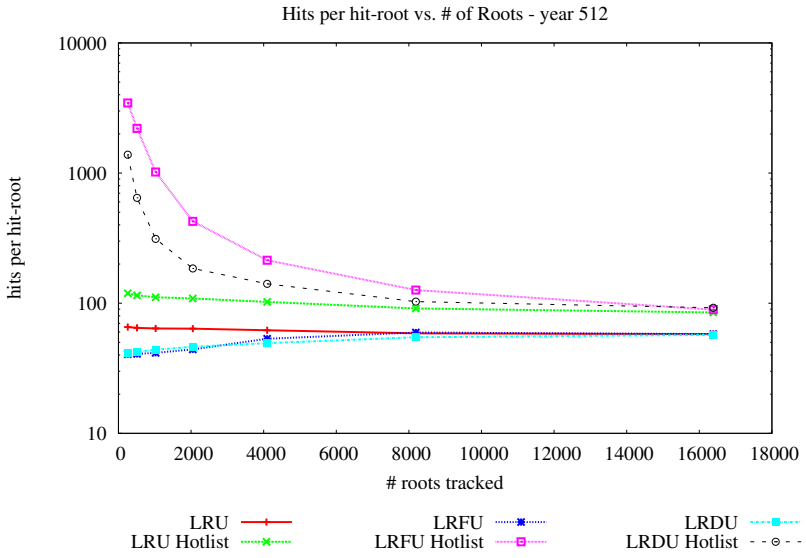
This base *LRFU* structure is then used for an *LRFU* Hot List, similar to the *LRU* Hot list detailed in Section 8.2.3. The strategy for this hot list structure follows directly from the *LRU* based example, with an *LRU* filter and an *LRFU* hot list. This strategy is expected to be significantly faster than a straight *LRFU* structure due to the expensive *LRFU* portion being half the size.

8.2.5 LRDU and LRDU Hot List

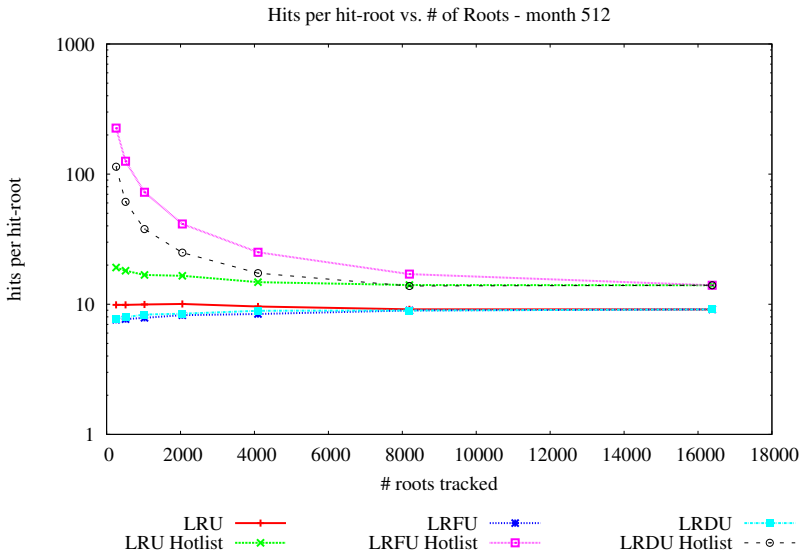
Our *LRDU* strategy is exactly our *LRFU* structure with the default "new addition" and "event addition" scores replaced with the event's most recent track distance, as calculated by Equation (13). The hot list version follows directly from the *LRFU* version in Section 8.2.4, with an *LRU* filter and a *LRDU* hot list.

8.3 EXPERIMENTAL SETUP AND DESIGN

For this project, we used a selection of traces detailed in Chapter 4 to test our prototype structures. In particular, the *mozart*, *hplajw*, *ranin*, and *playlist* sets were used. In order to determine sensitivity to structure size, we ran each trace with a variety of sizes, ranging from 256 to 16384, doubling the size at each step. Preliminary tests of the *mozart* trace, where we also varied the block sizes of each workload. The *mozart* tests verified insensitivity to block size, detailed in the next section.

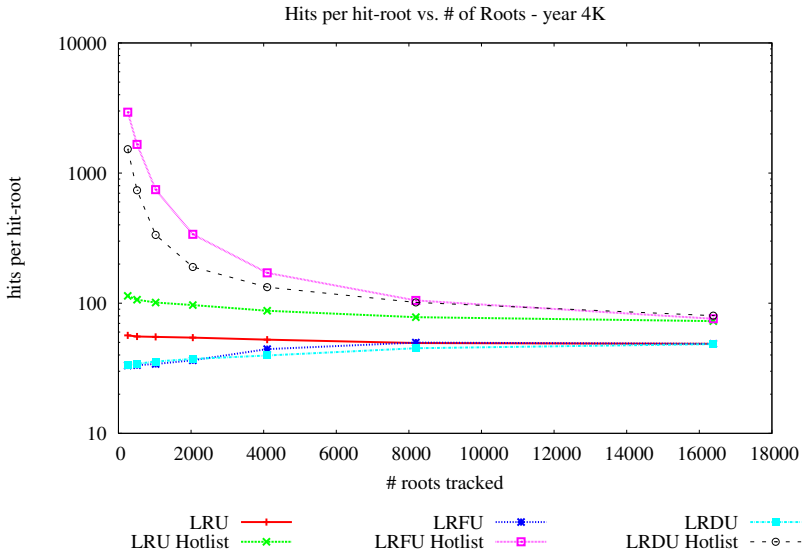


(a) *year*

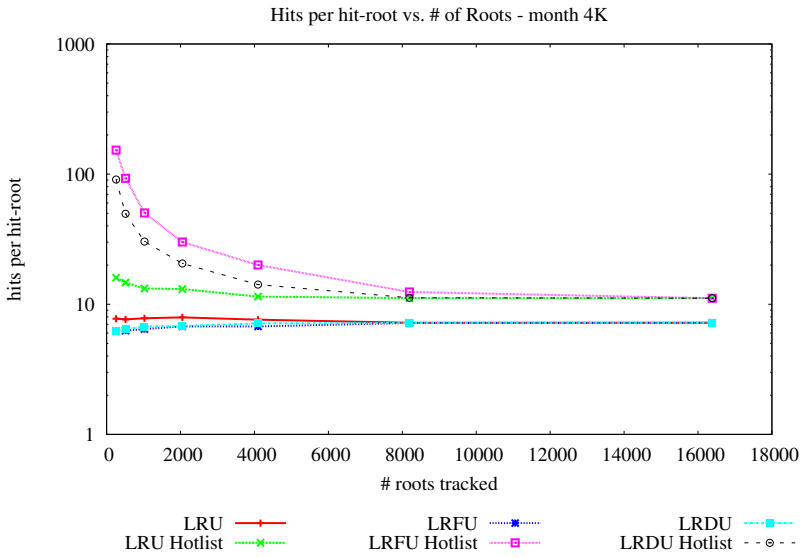


(b) *month*

Figure 27: *LRDU* stability of *mozart*, month and year traces, with 512 byte blocks.

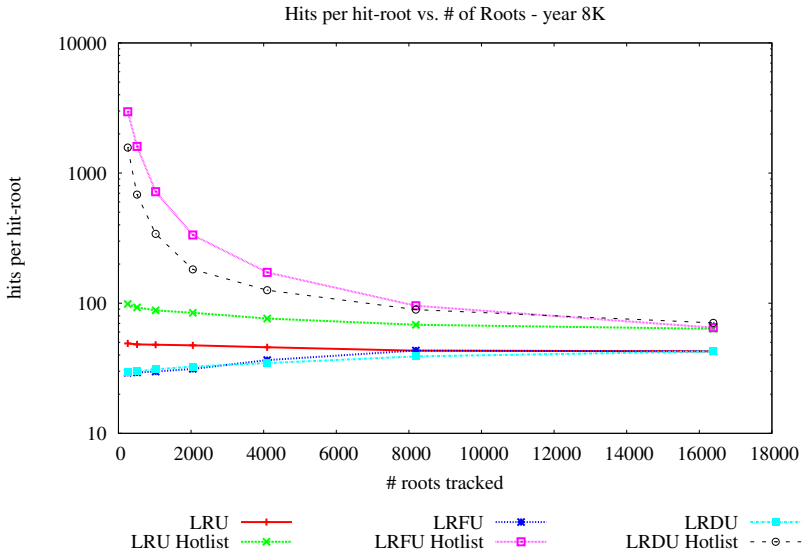


(a) *year*

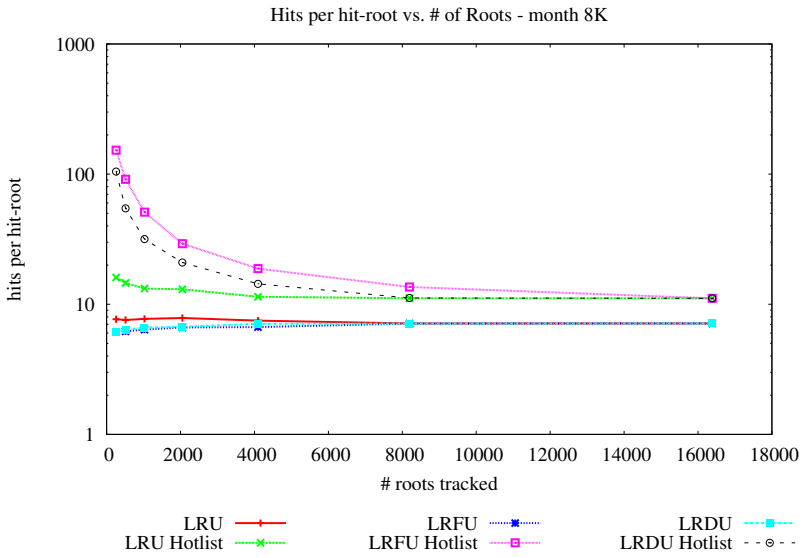


(b) *month*

Figure 28: *LRDU* stability of *mozart*, month and year traces, with 4 KB blocks.



(a) *year*



(b) *month*

Figure 29: *LRDU* stability of *mozart*, month and year traces, with 8 KB blocks.

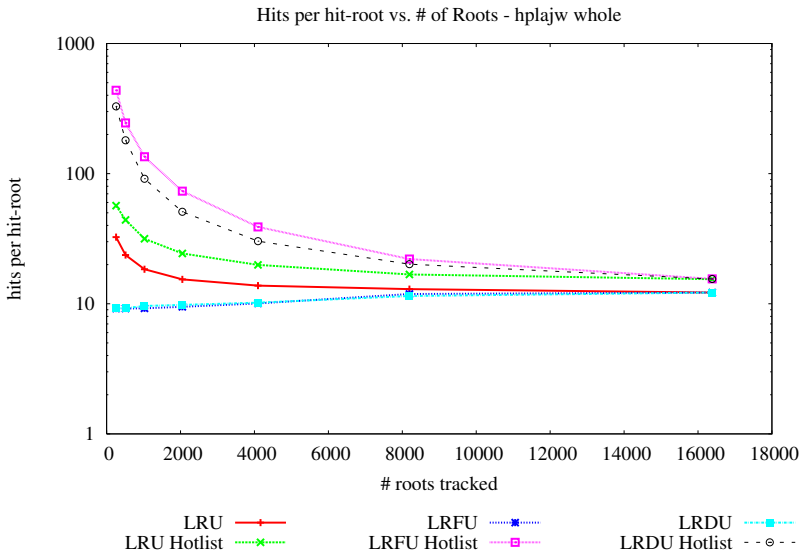


Figure 30: *LRDU* stability of *hplajw* trace.

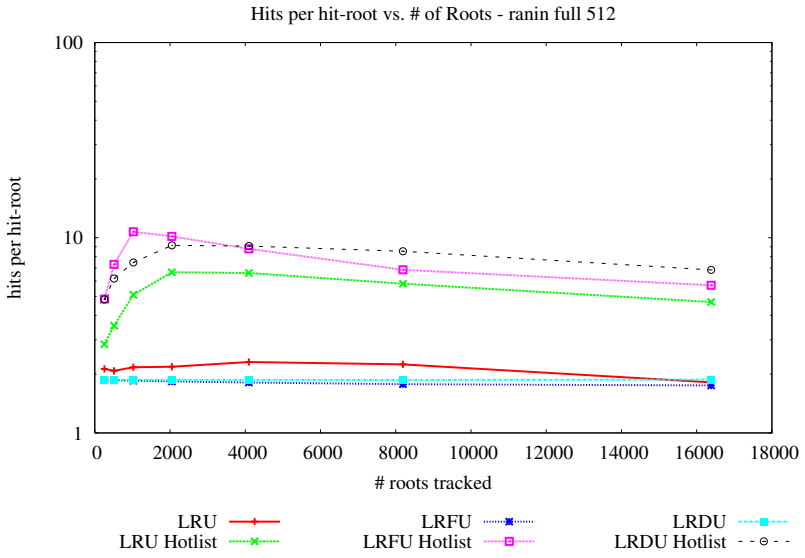


Figure 31: *LRDU* stability of full *ranin* trace, 512 byte blocks.

As a result, all other workloads were kept in their "natural" block sizes; 512 bytes for the *ranin* and *playlist* traces, and 8 KB for the *hplajw* trace.

In order to eliminate the bulk of blocks throughout the workload, we employ the use of *track distance*. Any block that causes a track transition is considered for addition to each structure.

The most tempting statistic of interest to gather for these experiments is the number of hits within each structure. However, as we mentioned in Section 8.1, this is not our only concern. We must choose roots that exhibit high stability in order to reduce the number of necessary updates at the storage system level. Thus, we need to track how many hits occur *and* how many unique root IDs caused a hit. Ideally, this hit-per-hit-root statistic (hit per unique potential root that has caused a hit) should be high, showing high stability.

We also mentioned that an ideal structure would exhibit another form of stability, easy sortability, with higher priority items possessing a higher estimated savings potential. In order to estimate this "sorting stability", we track the total track distance caused by each block, using Equation (13) throughout the entire trace. At the very end of the trace, we iterate through each structure and check the global track distance caused by each ID at each rank. For simplicity, we restrict our graphs to the top 250 items in each structure. These items represent the highest priority offered by each structure. Structure with sorting stability should have a cumulative distribution graph that closely matches that of our Highest Distance metric, perhaps even above the Highest Count metric, while strategies with low sorting stability will have a cumulative distribution graph well below these metrics.

8.4 RESULTS

Our results show that for both hits per hit-root as well as sorting stability, our *LRDU* hot list closely matches or outperforms other feasible strategies. In particular, we see that this *LRDU* hot list strategy is a close second to the *LRFU* hot list in terms of hits per hit-root, even outperforming it for larger structure sizes on the *ranin* traces in Figure 31. In almost all cases, in fact, similar strategies tend to perform more and more alike as we increase the structure sizes. This behavior is expected; the better strategies choose better roots when resources are scarce. As available resources

become abundant, eviction decisions begin to matter less and less. Thus, as the structure sizes increase, all three hot list strategies perform more and more alike, while all three "pure" strategies remain dominated by the hot lists, but perform more and more like each other. We also note, as we mentioned previously, that these trends hold across block sizes, as well as across trace lengths and workloads.

For block comparisons, consider Figures 27(a), 28(a), and 29(a) for the *mozart* year traces and Figures 27(b), 28(b), and 29(b) for *mozart* month traces. Note that Figure 27 displays results for 512 byte blocks, Figure 28 for 4 KB blocks, and Figure 29 for 8 KB blocks. For further comparison across workloads, see Figure 30 for the *hplajw* trace and Figure 31 for the full *ranin* trace with its native block size of 512 bytes.

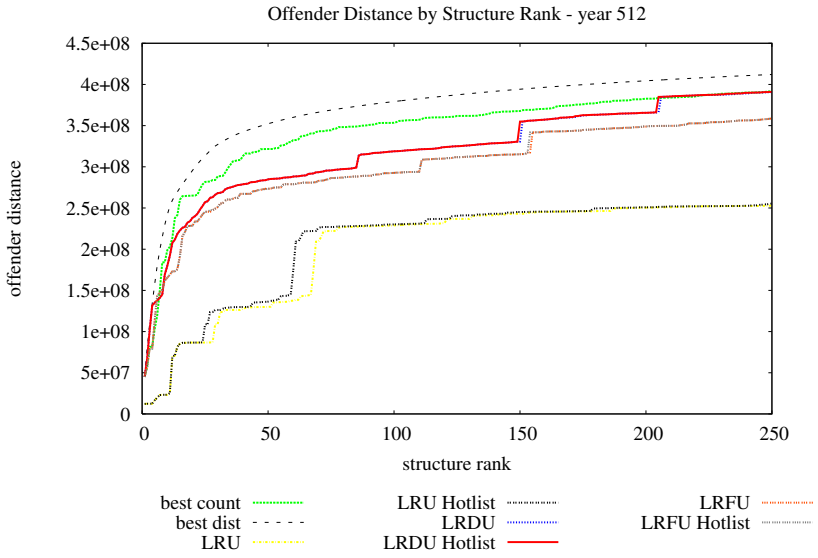
Our sorting stability results in Figures 32, 33, 34, 35, and 36 show that our *LRDU* and *LRDU* hot list structure consistently outperforms other feasible strategies. Indeed, in some cases, we even outperform the *highest count* strategy, a computationally infeasible policy, as can be seen in Figures 35(a) and 36(a). In fact, Figure 35(a) shows our *LRDU* strategies very closely approximate the *highest distance*. We also note that, for all cases, each hot list version either closely matches or outperforms its "pure" counterpart. Again, these trends are consistent across block sizes, trace lengths, and workloads.

We also note that, for sorting stability, as the structure size increases, the hot list strategies and the "pure" strategies tend to perform more and more alike. For comparison, Figures 32(a), 33(a), 34(a), 35(a), and 36(a) all have a structure size of 16384, while Figures 32(b), 33(b), 34(b), 35(b), and 36(b) all have a structure size of 512. Table 13 summarizes these results for all six of the tested strategies.

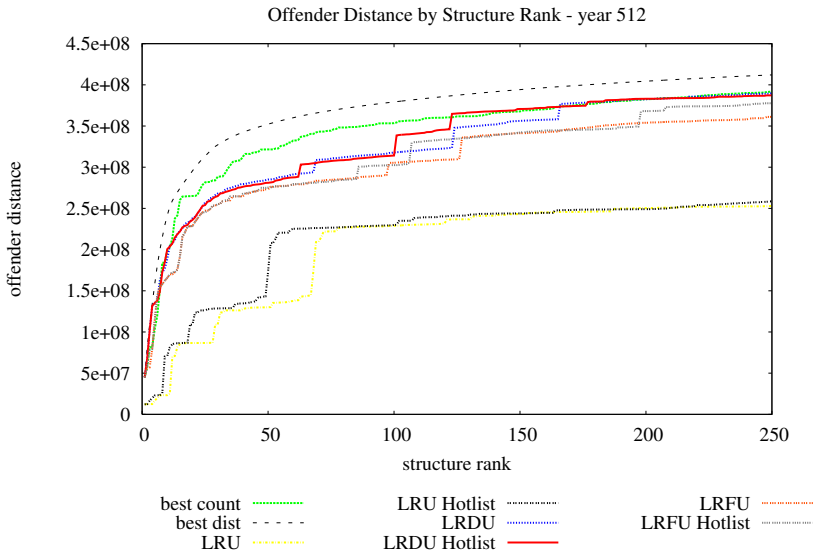
8.5 DISCUSSION

Now that we have established that both our *LRDU* hot list and *LRFU* hot list represent good candidates for root selection, we are faced with a choice. Which of these strategies are we to adopt? In order to address this, we must consider two factors:

1. Do we prefer hits per hit-root to sorting stability?

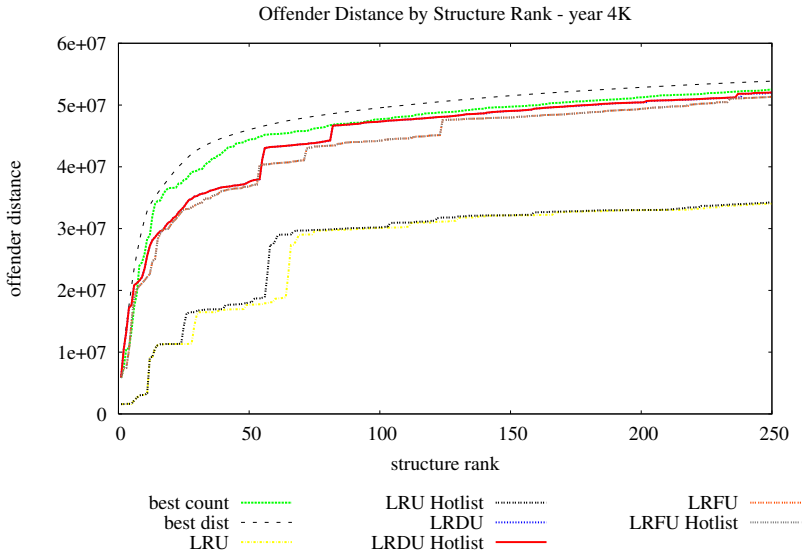


(a) *Structure size 16384*

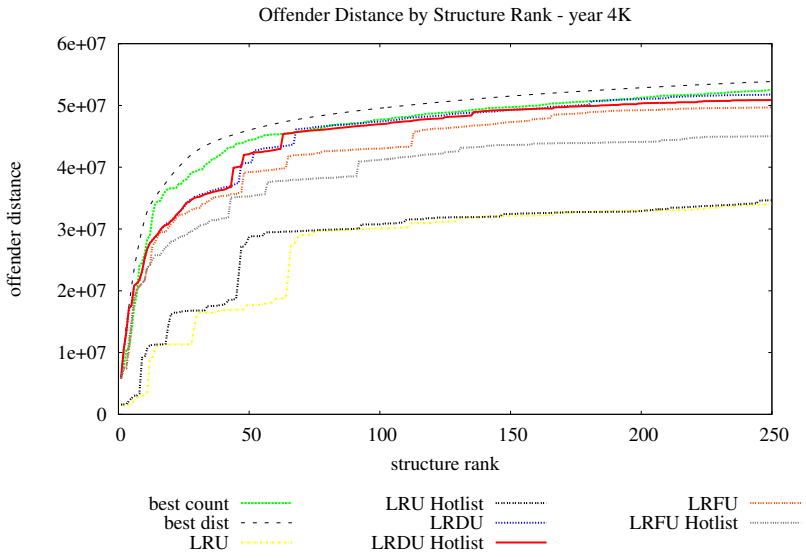


(b) *Structure size 512*

Figure 32: *LRDU* sorting stability of *mozart*, year trace, with 512 byte blocks.

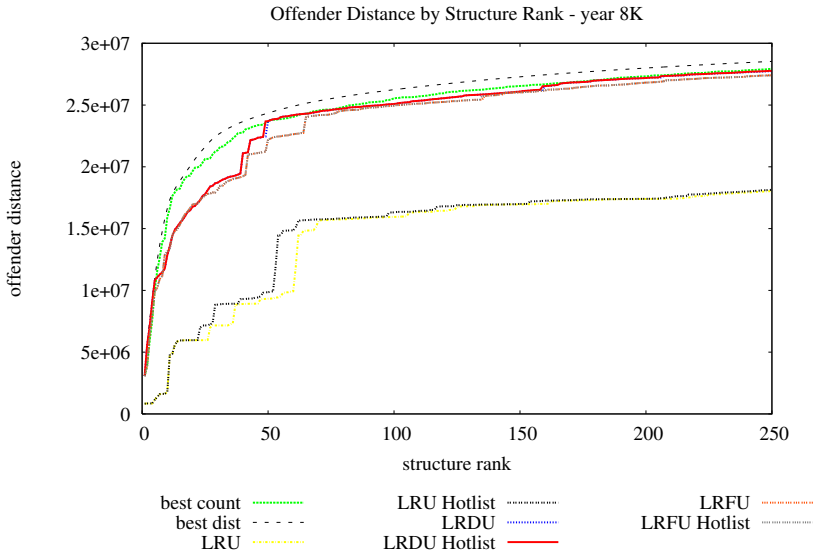


(a) *Structure size 16384*

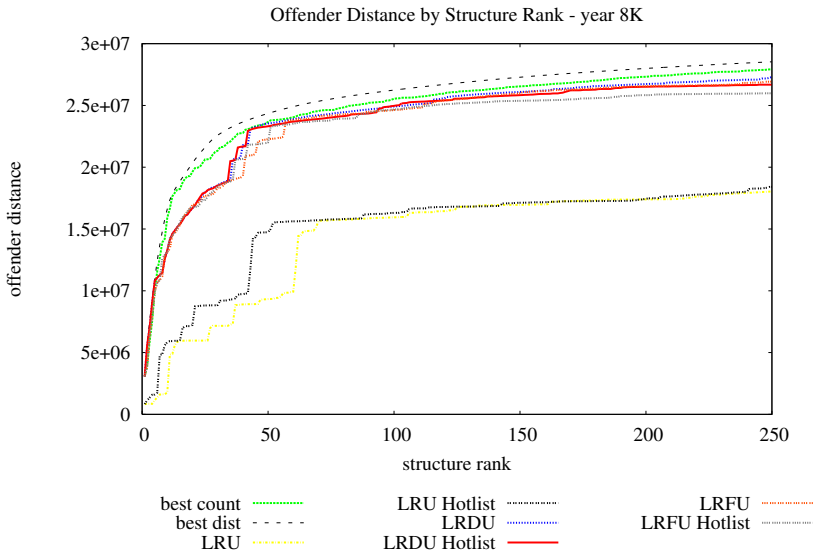


(b) *Structure size 512*

Figure 33: *LRDU* sorting stability of *mozart*, year trace, with 4 KB blocks.

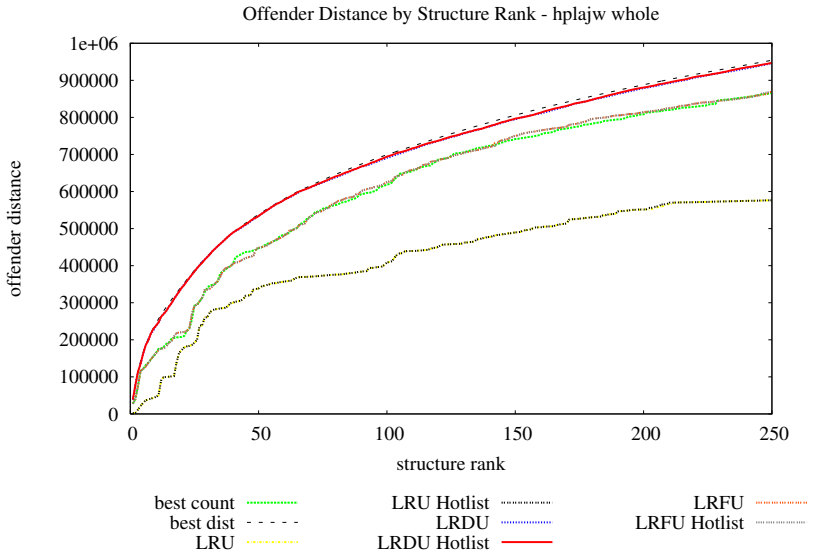


(a) *Structure size 16384*

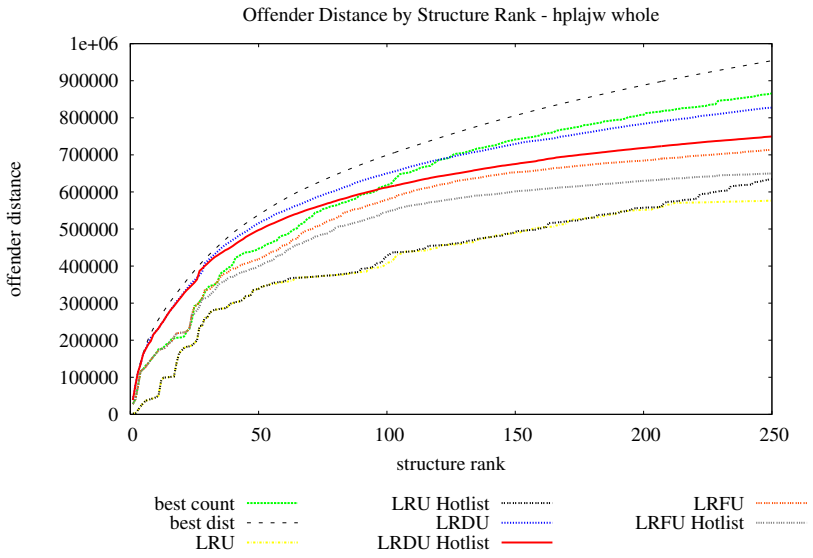


(b) *Structure size 512*

Figure 34: *LRDU* sorting stability of *mozart*, year trace, with 8 KB blocks.

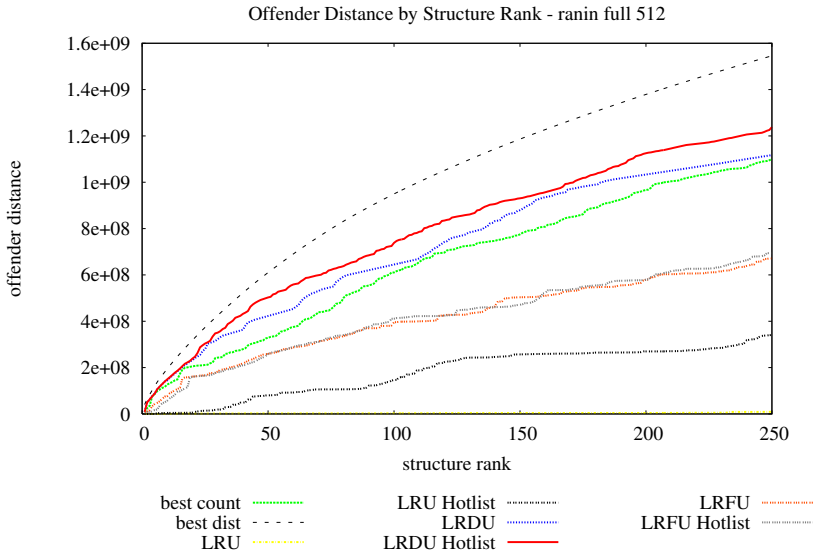


(a) *Structure size 16384*

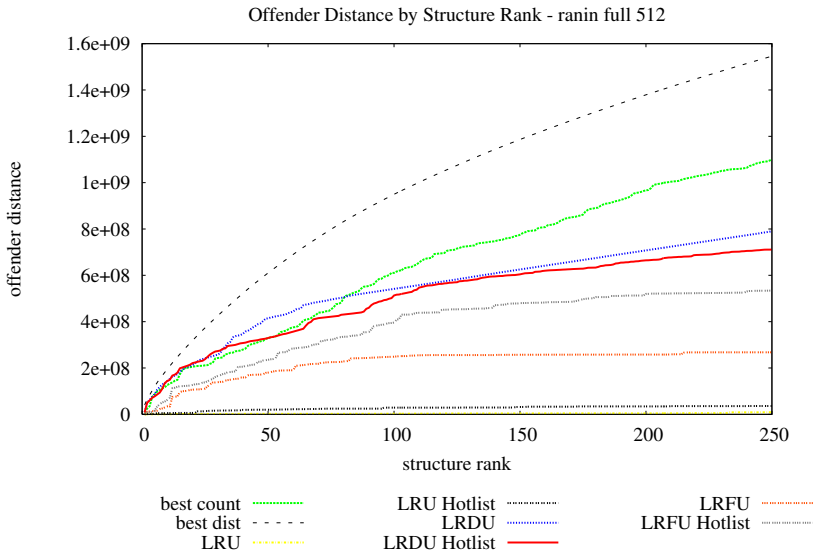


(b) *Structure size 512*

Figure 35: *LRDU* sorting stability of *hplajw* trace.



(a) *Structure size 16384*



(b) *Structure size 512*

Figure 36: *LRDU* sorting stability of full *ranin* trace, with 512 byte blocks.

Table 13: Summary of advantages of all root selection strategies.

Strategy	**Remains efficient with larger structure size**	**High Hit to Hit-root ratio**	**High Sorting Stability**
LRU	✓		
LRFU			✓
LRDU			✓
LRU hot list	✓	✓	
LRFU hot list	✓	✓	✓
LRDU hot list	✓	✓	✓

2. How big do we anticipate our structures to be?

The answer to the first question is straightforward; we prefer hits per hit-root. If the structures are anticipated to be small, we should choose our *LRFU* hot list. However, if we anticipate our structures being quite large, then both strategies will tend to have the same or very similar ratio of hits to hit-roots, while the sorting stability for the *LRDU* hot list would improve faster than that of the *LRFU* hot list. Thus, we must answer only this question; do we anticipate the structure sizes to be quite large? In general, the larger the structure, the more hits we can anticipate. Figures 37 and 38 show that, for all hot list structures, this appears to hold true. It is interesting to note that both pure *LRFU* and pure *LRDU* seem to suffer from Belady's anomaly, while the other strategies do not. Thus, if we are to increase the total potential of a strategy, we may wish to increase the structure size, and we would prefer the *LRDU* hot list to the *LRFU* hot list.

Alternatively, we can consider the choice between an *LRDU* and an *LRFU* hot list by considering our original intent from Section 8.1. We wish to have a stable set of potential predictive roots that have a high probability of occurring within the workload *and* have a large potential benefit. Stability, we have mentioned, corresponds to our hit per hit-root ratio. Potential benefit can be considered in one of several ways. First, we can consider the distance potentially saved by each hit. Thus, larger distances correspond to larger potential savings. This implies that our *LRDU* strategy

should be chosen because of its consideration towards distance. But there is another consideration as well. Assuming we have a predictive group already made, when will we switch to it? While we address this question in detail in Chapter 9, it is worthwhile to briefly answer here. In general, if we are *forced* to switch to a different group, we prefer a group that is closer to one that is far away, all other things being equal. Thus, if we give preference to roots that cause large seeks, we are more likely to use those roots when we are faced with this choice, and again, we should choose our *LRDU* hot list over our *LRFU* hot list, regardless of structure size.

8.5.1 Conclusions

We have examined a number of candidate solutions, including several new and novel strategies, for tracking group roots, a necessary task for an online grouping engine utilizing first-order successor information. The best candidates, a *LRFU* hot list and *LRDU* hot list, both exhibit desirable qualities. Both are easily sortable, efficient even for large structure sizes, and both exhibit high hit to hit-root ratios. In fact, in some cases, these strategies even outperform competing but computationally infeasible methods. Of these two, our *LRDU* hot list is preferred due to higher expected value of *distant* roots over *frequent* roots, leading to higher potential use of groups within a distance-aware engine.

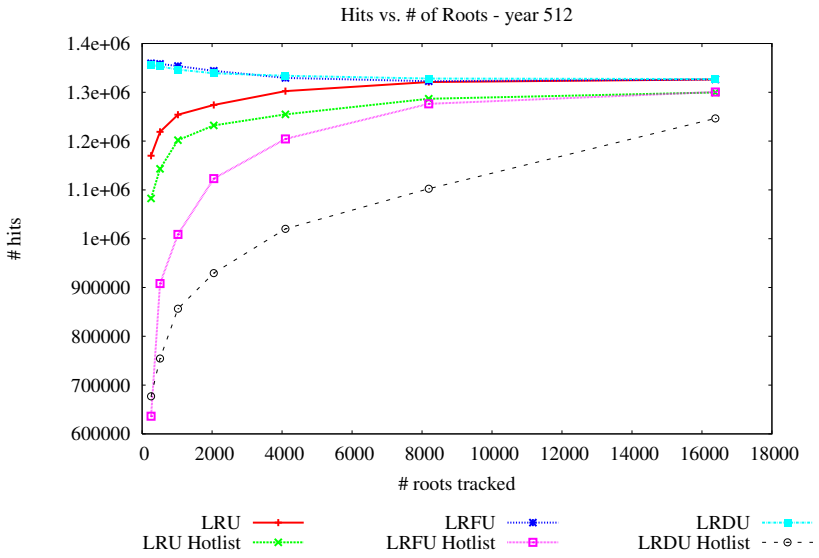


Figure 37: Hits vs. structure size for *mozart* year trace, with 512 byte blocks

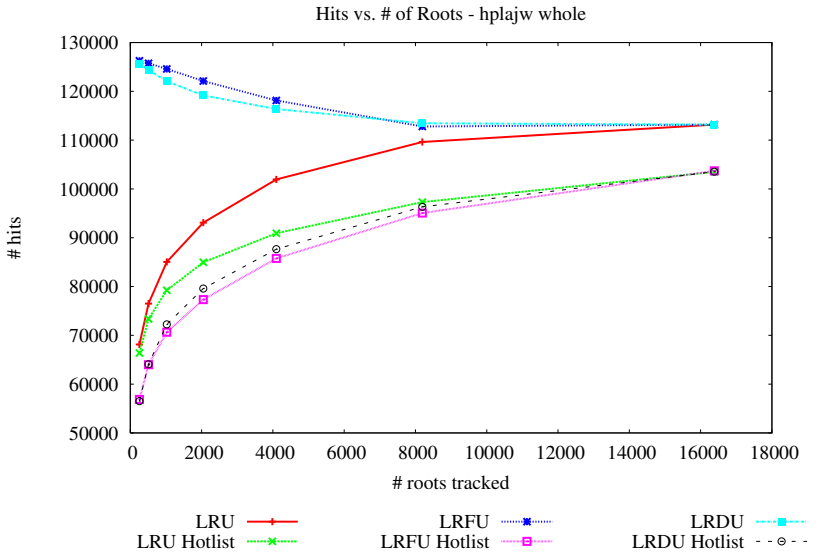


Figure 38: Hits vs. structure size for *hplajw* trace

9.0 SPORE - SPACE-EFFICIENT ONLINE REORGANIZER

Having studied various prospective collocation methods, and having addressed metadata storage solutions as well as root tracking strategies, we now present the construction of our dynamic grouping engine. We have previously described strategies for grouping and collocating data, as well as further enhancements to provide effective online application. We have further detailed mechanisms of metadata reduction to greatly reduce predictive information overhead, and have provided methods of identifying practical starting points for group formation, called roots.

Our comprehensive predictive grouping engine, *SPORe*, or *SPace-efficient Online Reorganizer*, uses our optimal expansion strategy, *OE ME*, from Chapter 6. We also incorporate our *SESH* structure for tracking metadata; in practice, we use the augmented structure that tracks predecessor information as well as successor information, in hopes of future augmentations using ancestor history. As a result, the Optimal Expansion tree structure we chose to use is the queued variant detailed in Section 7.3. Our *LRDU* hot list from Section 8.2.5 is used to monitor potential roots for predictions.

New challenges arise as we attempt to unify these solutions into a complete grouping policy. The rest of this chapter focuses on these new challenges and new optimizations. While Chapters 7 and 8 discuss isolated solutions to individual problems related to dynamic grouping, new challenges are introduced and addressed when combining them as parts of a single online predictive grouping engine. Consequently, we seek to offer the reader a greater understanding of the challenges within the context of a dynamic grouping engine, rather than generalizing outside such context as in previous chapters.

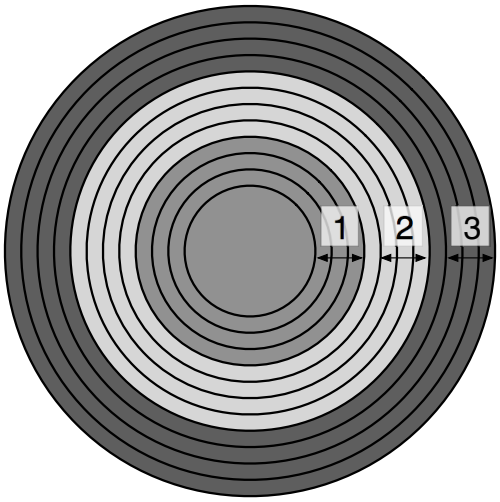


Figure 39: Hard disk drive, separated into three ranges, each consisting of four tracks.

9.1 MOTIVATION

As we have already established, a dynamic grouper must operate effectively in the face of many constraints. In general, we need to be able to do the following:

1. *Form a predictive group from some root or roots.*
2. *Gather the necessary metadata for predicting.*
3. *Choose roots for our predictive groups.*
4. *Decide where these predictive groups will reside on the physical storage device.*
5. *Choose when to write out and when to update these predictive groups.*
6. *When necessary, decide which group, predictive or not, to switch to.*

Our constraints while achieving these goals are:

1. *Avoid excessive strain on the CPU.*
2. *Eliminate unnecessary updates to reduce the burden on the storage system.*
3. *Avoid excessive metadata volume size.*

Most of these goals and constraints we have already addressed. We have demonstrated that accurate predictive groups can be formed with our *Optimal Expansion, Maximized Expectation* (Section 6.2) strategy, given some root (goal 1). Further, this strategy shows higher accuracy and better resilience to diminishing returns than competitive strategies. Satisfying this goal allows us to adapt to previously observed behavior in order to predict future events. With such a prediction strategy, it becomes necessary to track this previously observed behavior (goal 2) without taxing the storage subsystem with metadata retrievals and updates (constraint 3). We have addressed this issue with *SESH*, and it has allowed us to compact this potent first-order successor information from 14% down to less than 1% the total volume of the storage system.

Predicting from some root allows us adaptability, but necessitates a starting position, the root itself, in order to proceed (goal 3). Our proposed solution, an *LRDU* hot list structure (Section 8.2), easily lends itself to the task by providing roots that will be easier to transition to, given a group transition is necessary, as well as providing roots with a high savings potential.

Having formed a predictive group, we must decide where it will reside on the storage system; it must exists somewhere in order for that group to be available for future use (goal 4). One generally accepted rule of thumb in systems research is that disk accesses are precious, a costly commodity, while disk space is, by comparison, expendable [47]. This observation, along with early work by Akyürek and Salem on adaptive block placement [4], suggests a preference to copying, rather than shuffling or migrating, a suggestion followed by recent work for data layout [15, 56]. Thus, we seek to utilize empty disk space in order to store predictive groups. Further, we would like to both manipulate and utilize these predictive groups opportunistically (goals 5 and 6).

From a high level, the objective is easy to describe. We wish to have a predictive group close by when needed; when we write out the group, it should be close by, and contain blocks that are normally located very far away but have high estimated likelihood of occurrence from the current position of the disk head. We accomplish this by having not only a single *LRDU* hot list structure, but multiple structures, each for a portion, or *range*, of the disk, as shown in Figure 39. This strategy has several benefits. Decisions become local; we view roots as “good roots for this section”. This allows us to reduce the size of the *LRDU* hot list structure, which allows for faster updates and retrievals (constraints 1 and 3) while enabling more adaptable behavior. In addition, the task of locating a suitable position for the predictive group is simplified; we need to find a

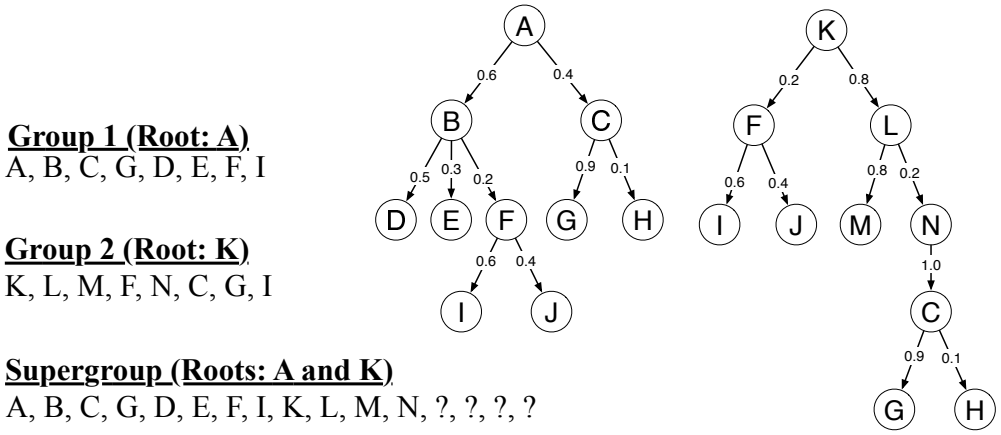


Figure 40: Example of a *supergroup*, with provided expansion trees, rooted at A and K. Group 1 and Group 2 have their members in order of addition to the group. Notice that, after merging the two smaller groups, there are four additional objects that can be added to the *supergroup*, due to the overlap between Group 1 and Group 2 of C, F, G and I.

nearby free physical location, or a group that has not yet been used by the system. This is easily accomplished by a free-list bitmap to track unused *groups*, rather than blocks.

Once we have a predictive group written out, we would like to keep it as up-to-date as possible. Stale blocks are of limited use. Recall, however, that disk operations are precious; we do not want to overburden the storage system with an inordinate number of updates (constraint 2). We can accomplish this via several strategies. First, recall from our work on static grouping (Chapter 6) that larger group sizes yield better savings potential, but experience diminishing returns. The expected benefit is roughly inversely proportional to the group size. For these larger groups, we may be able to combat this by allowing several smaller groups to overlap, creating one *supergroup* from several smaller ones. Consider a simple motivating example; two groups with 8 objects may have an overlap. When merged into one group, we are able to fit more than 8 objects per group without exceeding a total of 16 objects. Figure 40 clarifies this example.

Using *supergroups* in this way yields several benefits. Fewer total groups are necessary, since we have more than one root used per group; fewer total groups would yield fewer necessary up-

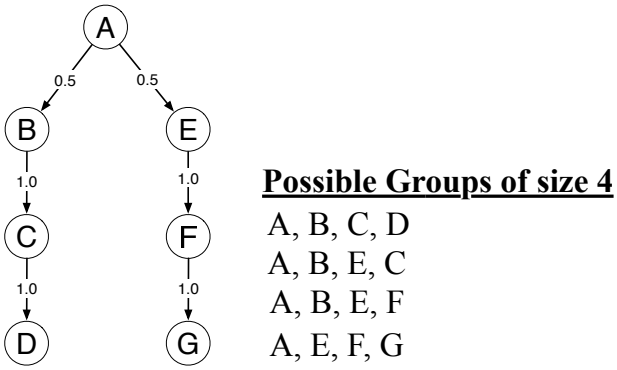


Figure 41: Example of four possible *OE ME* groupings of size 4 from the given expansion tree.

dates. In addition, we may be able to have a higher group size per root due to overlap, as in our example. In fact, we can even try to consciously exploit this. In our example, the formed *supergroup* was the merge of Group 1 and Group 2. Suppose we know that Group 1 has a higher potential. Presumably, both A and K were obtained from our *LRDU* hot list structure. If A was reported to have higher potential, we may wish to bias the *supergroup* towards A. This can be done by forming the group for K first, up to half the total *supergroup* size, then fill the remainder with A's group. Any overlap between the groups would allow additional objects that follow A to be added, biasing the group towards A *without penalizing* K*'s group*.

Our example in Figure 40 also illustrates another possible improvement. In our example, notice that object N has only a single successor, C. Indeed, our prior work on *SESH* exploits such cases; they are expected to occur the vast majority of the time. In these cases, performing priority queue operations is somewhat superfluous; if there is room for C after adding N, we may simply add it and expand the priority queue with C's successors. Indeed, it may even be beneficial to avoid using the priority queue in this case. The first issue to consider is computational cost. These operations we have observed as the most demanding for the *OE ME* algorithm for large groups; reducing them could greatly improve time complexity (constraint 1). Second, it is expected for block accesses to occur sequentially more often than not. Continually going to the priority queue can, in the case of equal priorities, have unexpected results, depending on particular implementation. Consider

another example, Figure 41. Notice that it is not clear which of the four groupings will actually occur, though we would clearly prefer either the first or the last, since these groups contain intact sequences.

9.2 EXPERIMENTAL SETUP AND DESIGN

Our dynamic grouper, *SPORe*, is composed of four objects; a controller, a scribe, a root monitor, and a cartographer, shown in Figure 42. The data request stream, which normally would have gone directly to the storage system, is redirected through the *SPORe* control. This is a high-level decision engine, used to generate requests to the other three primary objects. The scribe is a *SESH* object implemented with a maximum of 8 children. This object, detailed in Chapter 7, is responsible for monitoring first-order successor information for all unique block IDs encountered as well as providing this information upon request.

The root monitor is a collection of *LRDU* hot list objects from Chapter 8. Each hot list is responsible for tracking potential roots for a single range on the disk. These ranges are further detailed in Section 9.2.1. Since we reduce the number of roots that need tracked by each structure, we are able to reduce the structure size required of each hot list in order to maximize stability. In addition, small structures allowed for very fast updates and retrievals. The default structure size we used was the group size divided by 256. This number was chosen based on preliminary exploration; however, further research is needed to confirm insensitivity to this parameter that we observed.

The cartographer object was responsible for several tasks. Most importantly, it is responsible for making the final requests to the underlying storage system. These requests could be untouched requests from the data stream or redirected requests to predictive groups. The cartographer is also responsible for generating write operations for initializing and updating these predictive groups.

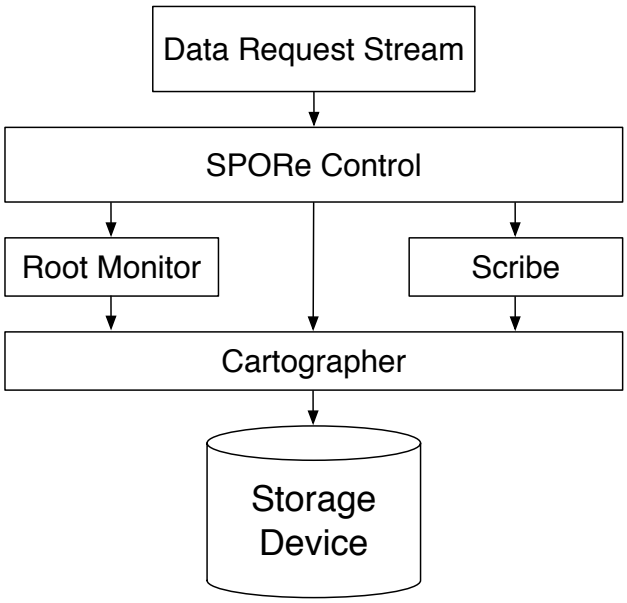


Figure 42: *SPORe* figure.

9.2.1 Root Placement

One function our cartographer object serves is to keep track of physical device layout. We do this by first marking each group that is to be accessed without translation. We call these accesses *raw accesses*, and refer to these groups as *raw groups*. Groups are divided into *ranges* on the device, with the translation from any particular group ID to its corresponding range ID calculated by a simple integer division, provided in Equation (14).

$$range\_id = \frac{group\_id}{range\_size} \tag{14}$$

Each range is allotted an *LRDU* hot list for tracking prospective roots for predictive grouping. These hot lists are contained within the root monitor object, and provide localized views of what roots have the highest cost to access from some particular range. In order to place these groups, we first scan the range that we are predicting for. If there is an unused group within the range (a group not yet marked as raw), we perform our prediction there. If not, we begin scanning the device

outward from the range in question for an unused group. If we reach some maximum distance from the range, r_max, and have not yet found an empty group, we cease the scan and mark the range as unpredictable. In practice, we have r_max set to the average raw track distance, or the average track distance that would be traveled without any predictive grouping. This allows us to avoid unnecessary scans for the range; in the future, we can simply check to see if the range is unpredictable *before* scanning.

9.2.2 Reducing Update Overhead

ALGORITHM 8 SUPERGROUP_OE_ME(T, R, max) - a variant of *OE ME*, given in Algorithm 2, that forms a group from several roots. Note that we call OE_ME_Prime (Algorithm 9).

Input: set of first-order successor trees, T; an array of root IDs, R, arranged in increasing order of estimated potential; a maximum group size, max

Output: a set of IDs, G, representing the predictive *supergroup*

$p\_max \leftarrow max/\text{SIZEOF}(R)$
$s \leftarrow p\_max$
for $i = 0$ to $\text{SIZEOF}(R) - 1$ **do**
    $G \leftarrow OE\_ME\_Prime(G, T, R[i], s)$
    $s \leftarrow p\_max$
end for
$G \leftarrow OE\_ME\_Prime(G, T, R[\text{SIZEOF}(R)], max)$
return G

We are able to reduce the overhead of updates performed by *SPORe* in three ways. First, in order to cut down on the number of necessary predictive groups, and thereby reducing the number of necessary updates, we use predictive *supergroups*. Rather than form groups using a single root, as we did in our work on static grouping, we use multiple roots. Our modified algorithm, given in Algorithm 8, goes through an ordered array of roots, each with increasing potential, predicting on each successive root in turn while allowing overlap between the groups.

We are also able to reduce the total number of updates by aborting updates that have very limited use. While we hope to keep predictive groups fresh, there is little gain to be expected from

updating an entire group if only a few blocks are to be replaced. In such cases, where the majority of blocks remain unchanged, we should consider the group "fresh enough", and avoid updating it at this time. We are able to accomplish this task by performing a group difference between a previously written predictive group and its updated version, as yet unwritten. Computationally, this group difference is equivalent to performing a set difference. If the overlap between the groups exceeds some threshold, t, we perform the update. If not, we consider the old version sufficiently fresh, and abort the update. In practice, we used an overlap threshold of 0.75. This number was chosen based on preliminary exploration; however, further research is needed to confirm insensitivity to this parameter that we observed.

The last way to reduce the overhead of updates is by opportunistically using blocks that exist in memory. As we are forming a predictive group, we check each block for existence in memory. The block is added to the group only if it is readily available, without causing an additional device access for retrieval, as shown in Algorithm 9. For our simulations, this requires a memory model. In practice, we initially modeled a strict *LRU* memory object, varying the size from 512 MB to 2 GB. In all cases, we found only a small percentage of rejections due to a block not in memory; usually, this rejection rate was between one and three percent. We consider this rate to be small enough to warrant removing the memory object model from our final version of *SPORe*.

This decision to remove the memory object was made for two reasons. In practice, memory management is much more sophisticated than simple *LRU*. Accurately capturing the behavior of modern systems' memory management we consider beyond the scope of our research. Second, it is our goal to have generalized results. Modeling memory would require multiple settings, including size, speed, and policy choices. With a rejection rate that is much smaller than our overlap threshold, especially from a simple strategy, such as *LRU*, indicates that memory management would have minimal impact.

9.2.3 Reducing Priority Queue Size

During our work on static grouping, we noticed that one of the largest demands on resources, both memory and CPU cycles, was the priority queue operations. In order to substantially reduce the cost of these operations, as well as to promote intact sequential sequences within our predictive

groups, we employ a priority queue "short circuit". Whenever we decide that a block should exist within a predictive group, *regardless* whether the block is in memory, we immediately check the block's number of successors. If only a single successor exists, we attempt to add this successor, and repeat the process until more than one successor is found. This optimization is shown in Algorithm 9.

9.2.4 Group Scanning

Once we have written a predictive group out to the disk, we would ultimately like to maximize the use of the group for several reasons. First, and perhaps most obviously, the aim of predictive grouping is to form areas on the disk that require fewer transitions and shorter disk head movement than the sequentially organized areas of the disk. In addition, frequent visits to predictive groups grants us low-cost opportunities for attempting updates. Until now, the only discussed method of entering a predictive group has been by a request for a root within the group. We are able to dramatically increase the number of uses of predictive groups by relaxing this constraint.

To this end, we use two simple methods. First, upon a request for a block not contained within the current group, we immediately check the current range's predictive group. If the block is contained within this target group, we transition there. If it is not, we scan each predictive track between the current head location and the target raw track. The nearest predictive group containing the block, assuming such a group is found, becomes the target group, and the disk head is redirected to this closer group.

This strategy intuitively results in higher usage of predictive groups, but weakens each use. Arguably, these groups' greatest strength is the amount of transition reduction. Entering the group on a block that is not a root, or at least near a root, is expected to reduce the number of accesses within the group. We claim that our grouping strategy is powerful enough to withstand this weakening, and is expected to continue to outperform raw accesses in this regard. The benefits of increased predictive group usage are therefore expected to greatly outweigh the risks of entering on a block far down in the priority queue.

ALGORITHM 9 OE_ME_PRIME$(G, T, root, max)$ - a balanced approach for forming a predictive group, using a group that may or may not already have members. Note that we use the same expansion function, OE_ME_Expand (Algorithm 3), as *OE ME* (Algorithm 2).

Input: an existing group G; a set of first-order successor trees, T; a root ID, $root$; a maximum group size, max

Output: a set of IDs, G, representing the predictive group

```
ENQUEUE(max_pq, root, 1)
while ISNOTEMPTY(max_pq) and SIZEOF(G) < max do
    p ← TOPPRIORITY(max_pq)
    f ← DEQUEUE(max_pq)
    if SIZEOF(G)+ SIZEOF(f) ≤ max then
        if f is in memory then
            ADDTOGROUP(G, f)
        end if
        while SIZEOF(G) < max and f has only one successor do
            f ← SUCCESSOR(f)
            if SIZEOF(G)+ SIZEOF(f) ≤ max and f is in memory then
                ADDTOGROUP(G, f)
            end if
        end while
        max_pq ← OE_ME_EXPAND(T, max_pq, f, p)
    end if
end while
return G
```

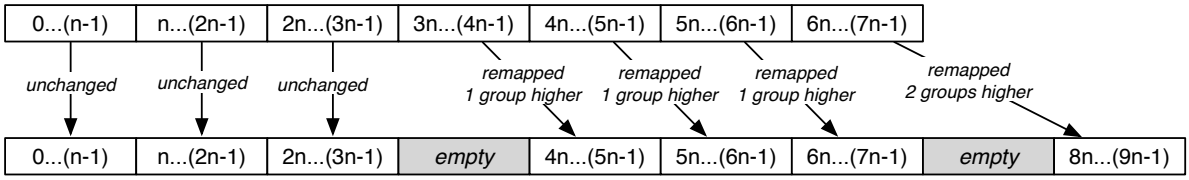


Figure 43: Inserting empty groups; in this case, we insert one empty group per four groups, simulating a device that is 25% empty.

9.2.5 Traces

We tested our prototype predictive engine *SPORe* on three workload sets representing three different environments. The first set used was the *mozart* workload traces from Section 4.1.1, gathered using the *DFSTrace* [91] system, and represent a pre-cache access stream. This set consists of four trace lengths, namely day, week, month, and year length. To test sensitivity to varying block size, these traces were converted to 512 byte, 4 KB, and 8 KB block traces.

In order to provide empty space for predictive grouping, we altered this workload by placing empty groups within the trace. We provide results for both a 25% empty disk as well as a 75% empty disk. Distance results provided are based on the original, compact workloads without empty groups. Figure 43 demonstrates this insertion strategy.

The second set, *hplajw* from Section 4.1.2, represents a native block-level trace gathered from an HP-UX system [102] at the disk level (*i.e.* filtered through the UNIX buffer-cache). This set was not converted into other block sizes, due to lack of file-system level information in the trace. Similar to the *mozart* trace set, we inserted empty tracks to simulate a 25% empty and 75% empty disk, with reported distances based on the original, compact trace without any empty groups.

Finally, we used our own trace set, *ranin*, detailed in Section 4.2.1. This trace set represents more modern disk access patterns, taken from a graduate student laptop workstation, namely the author's. For our work, cache-level information was, unless otherwise noted, ignored; only device level information was used. Much like the *mozart* trace set, this set was converted into different

block sizes to test for block size sensitivity.

These workloads were used without any warm up period or *a priori* information stored; all predictive information was captured on the fly. As a result, we can expect shorter traces to show little improvement, as the system requires time to gather and act upon observed workload patterns.

9.2.6 System Configuration

We conducted all of our experiments on a MacBook running OS X 10.5.8 with a 2.4 GHz Intel Core 2 Duo processor. This workstation contained 2 GB of 667 MHz DDR2 SDRAM, 3 MB of L2 cache, and an 800 MHz system bus. The test hard drive was an internal 5400 RPM Hitachi Travelstar 2.5 inch SATA drive with 160 GB capacity and an 8 MB cache. A vanilla version of Darwin 9.8.0 with a standard XNU kernel was used.

All programs were implemented in C or C++ compiled with the default versions of the GNU project `gcc` and `g++` compilers. To ensure correct program behavior, no optimization flags were used during project compiling.

9.2.7 Competing Model - Hot Block Clustering

In order to provide a strong competitive comparison for *SPORe*, we implemented our own version of an on-disk caching scheme based on hot blocks [4], varying over how often blocks were moved, the size of a group (or track) in blocks, and the number of tracks reserved for on-disk caching. Additionally, to further strengthen this hot block clustering scheme, rather than use the numerical "middle" of the disk, we calculated the average block location throughout each trace, and centered the disk caching at this average location. This provides an impractical but very beneficial optimization.

This scheme was run over all block sizes (512 byte, 4 KB, and 8 KB) on the *mozart* week, month, and year length traces as well as the *ranin* day, week, two week, and full length traces. The caching interval was varied between every 8192 accesses, 65536 accesses, and 524288 accesses. Group sizes used were 1024, 2048, 4096, and 8192 blocks. Reserved caching spaces (*region sizes*) used were 1, 2, 3, 4, 8, 16, 32, 64, and 128 groups.

We also used two organization techniques for how blocks were laid out within the hot block

clustering region. The first technique used organ-piping [52] as originally suggested for this on-disk caching strategy [4]. The second technique was to arrange the blocks by ID. Due to the highly sequential nature of the traces used, organ-piping performed very poorly for any region size greater than 1 group. As a result, in all cases tested, the sequential layout strategy outperformed organ-piping. For a region size consisting of a single group, the results were always identical to the sequential layout. For these reasons, only the sequential layout technique was compared against *SPORe*, providing a stronger competitive comparison.

These caching schemes were implemented in C++ compiled with the default versions of the GNU project `gcc` and `g++` compilers. To ensure correct program behavior, no optimization flags were used during project compiling.

9.3 RESULTS

We divide our results into five general categories: transition reductions, distance reductions, average number of accesses per use, write reductions, and estimated throughput of *SPORe*. The most crucial of these we expect to be transition reductions, or the number of times that the disk must seek to a new location. Of course, given that a transition is necessary, we would prefer a smaller seek distance in order to satisfy the request. The average accesses per use shows the strength of our predictive groups against that of the "raw", sequential groups already present on the disk. We also present the effect of our write reduction strategy, presenting both the number of writes attempted and the number of writes performed. Finally, we present the estimated throughput of *SPORe*.

In order to minimize effects of strategies not directly related to data grouping, we assume, in all experiments, a never-idle disk. This avoids power and latency effects associated with other, largely orthogonal power-saving techniques, such as spin-down techniques.

9.3.1 Reducing Transitions

Our results show transition reductions for almost all *mozart* workloads longer than the day trace for all parameters tested. Figures 44 and 45 show transition reductions for the week, month, and

Table 14: Comparison of estimated predictability of various workloads.

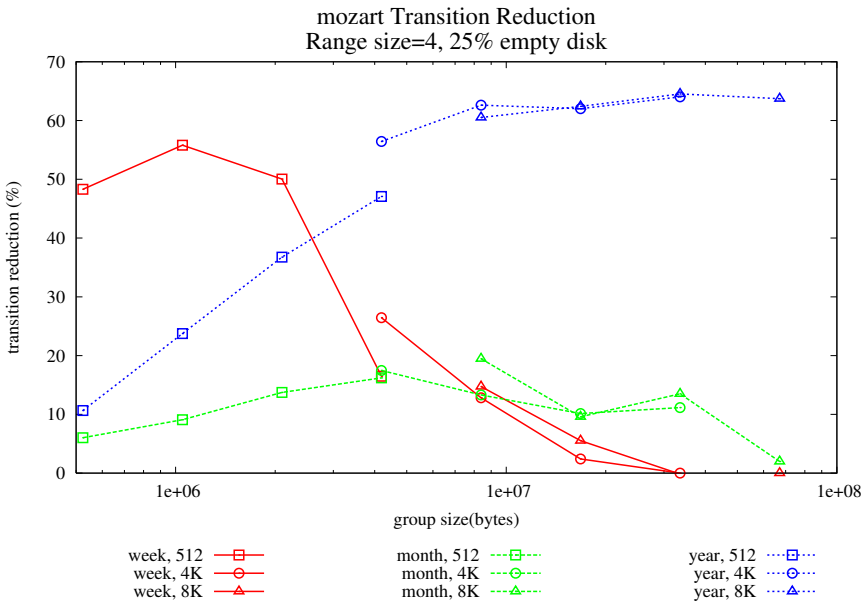
Trace	Block Size (bytes)	Total Accesses	% Sequential	Unique Blocks	% Unique
mozart, day	512	91,000	94.3	11,000	11.9
mozart, week	512	1,760,000	96.3	191,000	10.8
mozart, month	512	7,730,000	98.0	444,000	5.74
mozart, year	512	299,000,000	99.5	2,070,000	0.691
hplajw	8192	2,360,000	92.9	198,000	8.39
ranin, day	512	43,700,000	99.6	33,300,000	76.1
ranin, week	512	125,000,000	99.6	74,200,000	59.2
ranin, two week	512	181,000,000	99.6	98,700,000	54.5
ranin, full	512	260,000,000	99.5	120,000,000	46.2

year length traces for the 25% and 75% empty devices, respectively. Similar results for the *hplajw* workload are provided in Figure 46. These results include both group switches as well as updates for predictive groups.

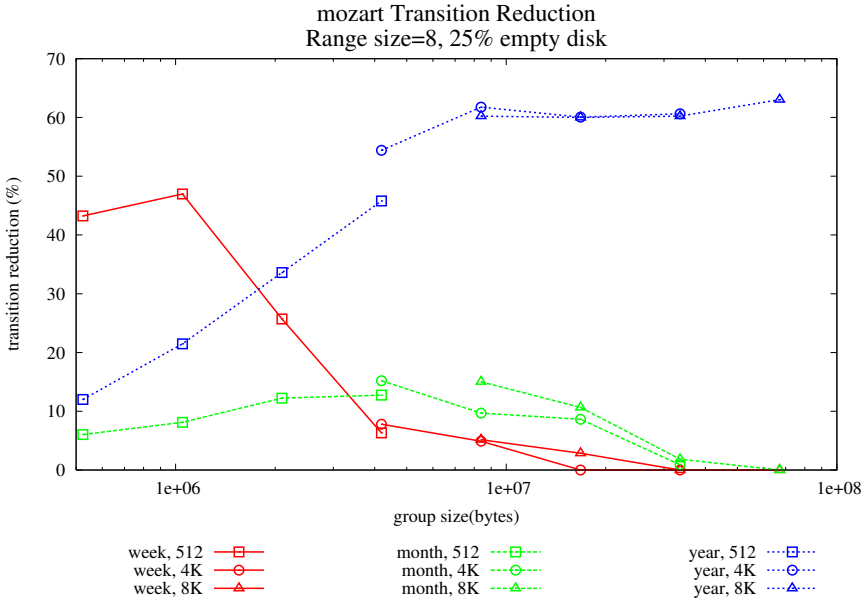
Of these 72 tested parameter permutations, only 8 showed no improvement, and only one of which showed any degradation. All of these cases had larger block sizes (4 KB or 8 KB) and larger group sizes (4096 objects or 8192 objects). Only the week length trace with 8 KB blocks and 4096 objects per group, with 25% empty disk, and 4 groups per range showed a decline, registering a 1 additional transition and 1 update from 1126 raw transitions, an increase of 0.1776%.

These results indicate several trends. First of all, we note that changing block size, while holding the number of blocks per group constant, increases group size in bytes. This parameter, group size *in bytes*, is the most crucial parameter observed; beyond its influence in this way, block size is observed to have little effect, especially for larger workloads.

Trace length trends show that increasing group size in bytes tends to have smaller effect, while larger workloads show an increase in performance. One possible explanation is that the deep pre-

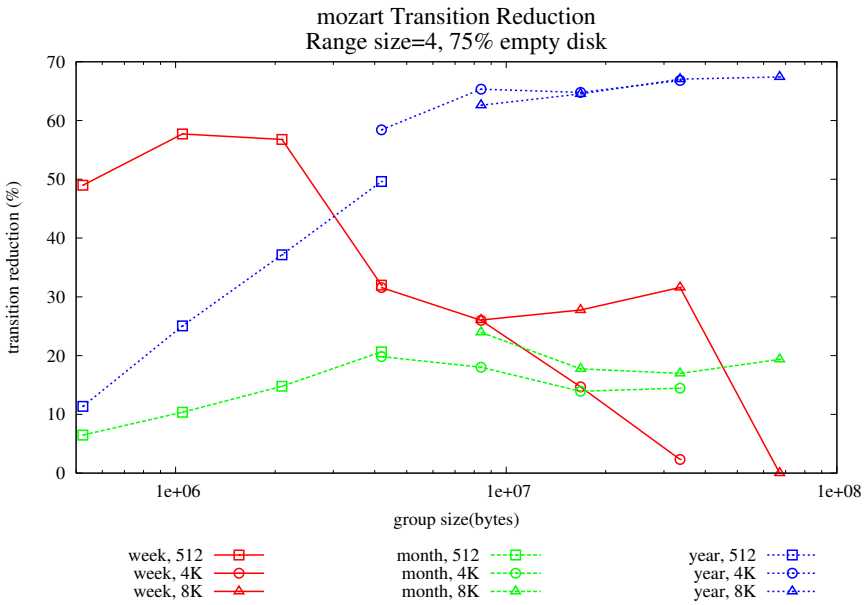


(a) *Four groups per range*

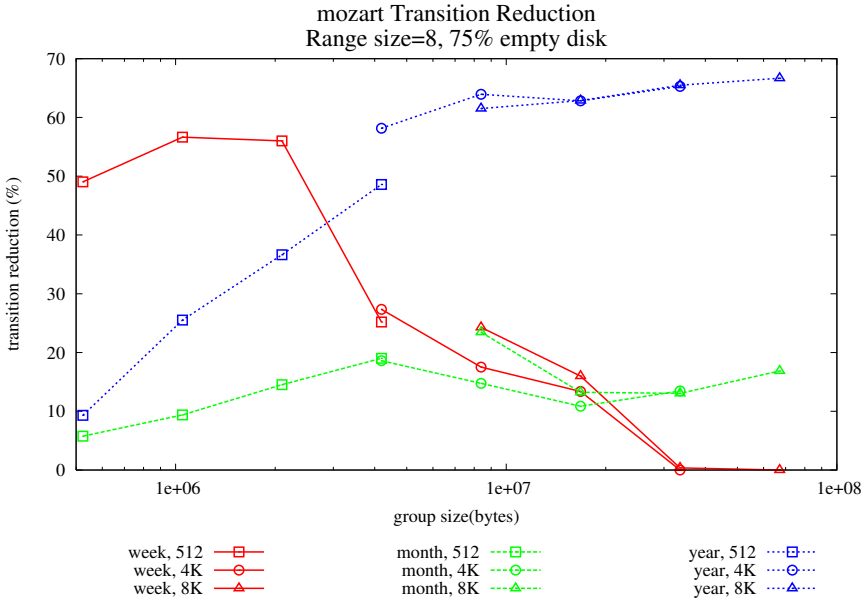


(b) *Eight groups per range*

Figure 44: *SPORe* transition reductions for 25% empty disk, *mozart* traces. Included are the week, month, and year length traces, with block sizes of 512 bytes, 4 KB, and 8 KB.



(a) *Four groups per range*



(b) *Eight groups per range*

Figure 45: *SPORe* transition reductions for 75% empty disk, *mozart* traces Included are the week, month, and year length traces, with block sizes of 512 bytes, 4 KB, and 8 KB.

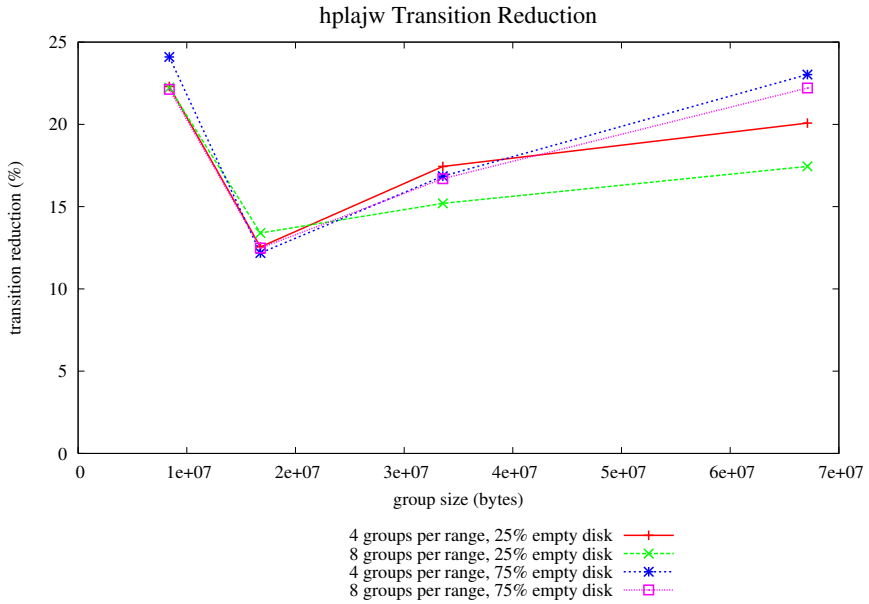
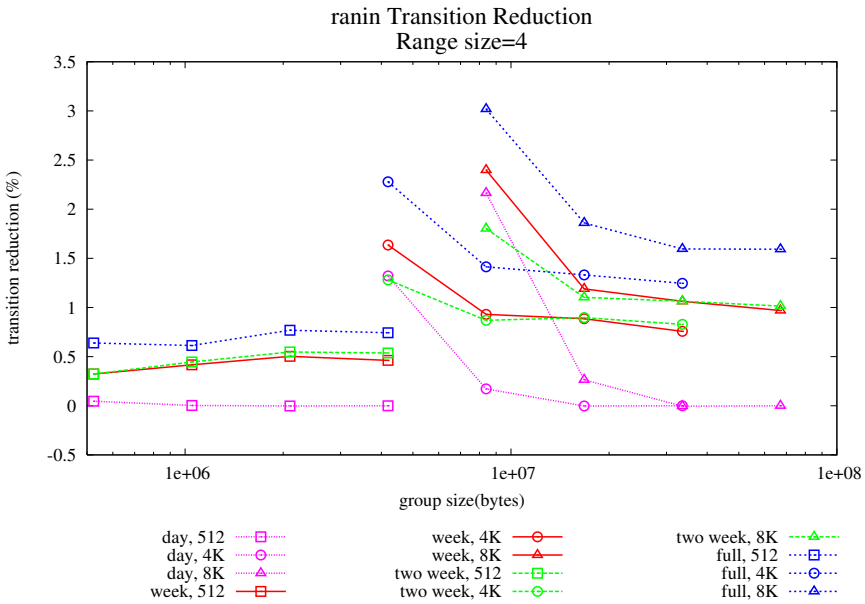


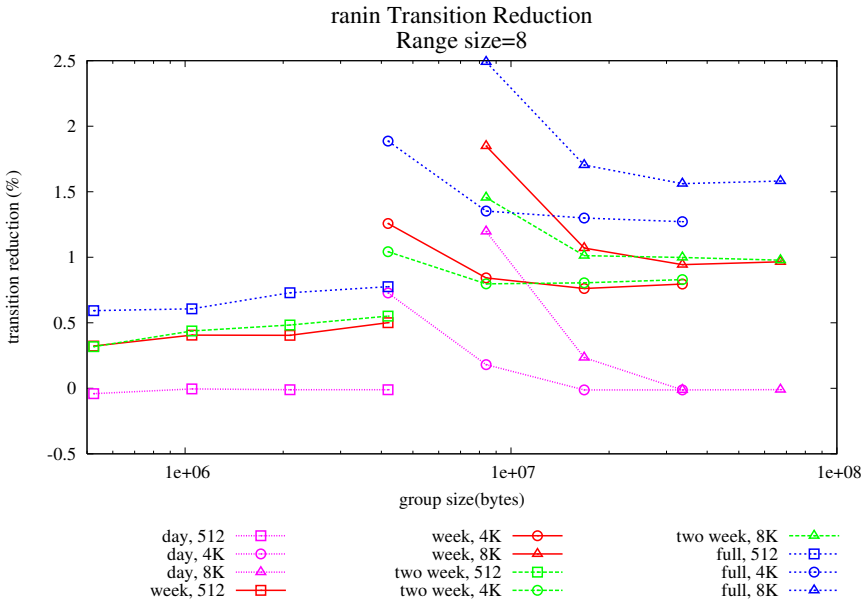
Figure 46: *SPORe* transition reductions for *hplajw* trace.

dictions necessary for larger group sizes require longer periods of time to develop. Additionally, longer workloads will have predictive groups available for most of the trace, while shorter workloads need to actively write predictive groups, yet have little time in which to use them. Finally, as group sizes increase, the total volume of shorter traces approaches that of a single group.

Our own *ranin* trace set represents a test of robustness for our predictive engine. While more sequential than either the *mozart*, the *ranin* trace set exhibits less predictability due to a large number of blocks that occur only once in the entire trace. Table 14 shows a comparison of the *mozart* traces with 512 byte blocks, along with *hplajw*, a natural 8 KB block trace, against our *ranin* traces. As expected, over time, the percentage of accesses that occur on an item not previously observed drops. Yet the shortest *mozart* workload has a ratio of unique blocks to accesses almost four times smaller than the largest *ranin* trace. Our full *ranin* trace, over a month long, has almost half the accesses as unique blocks. Such an environment is a true test of a predictive engine's sustainability. Even under these difficult conditions, *SPORe* not only does not suffer, but shows



(a) *Four groups per range*



(b) *Eight groups per range*

Figure 47: *SPORe* transition reductions for *ranin* traces. Included are the week, two week, and full-length traces, with block sizes of 512 bytes, 4 KB, and 8 KB.

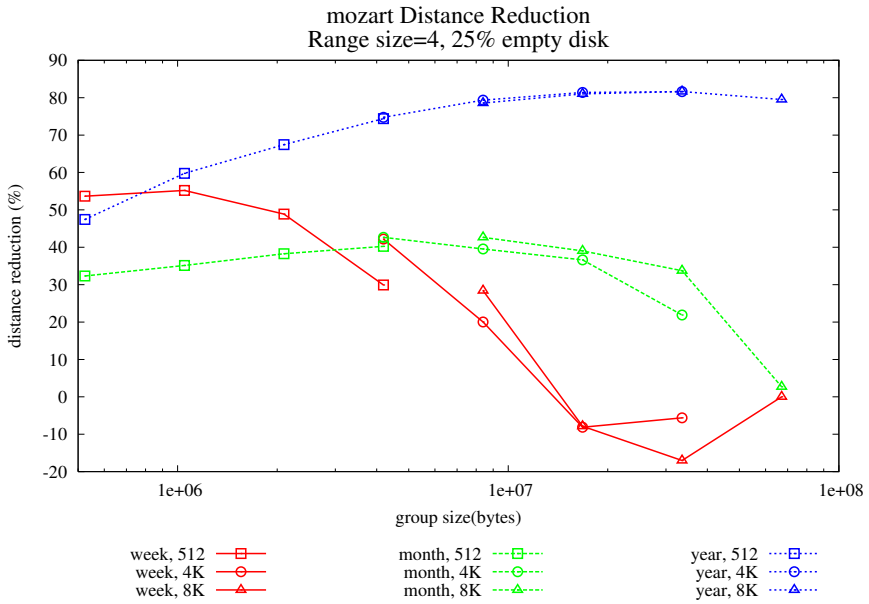
transition improvements, usually between 0.5% and 1.5%, as shown in Figure 47.

9.3.2 Reducing Seek Distance

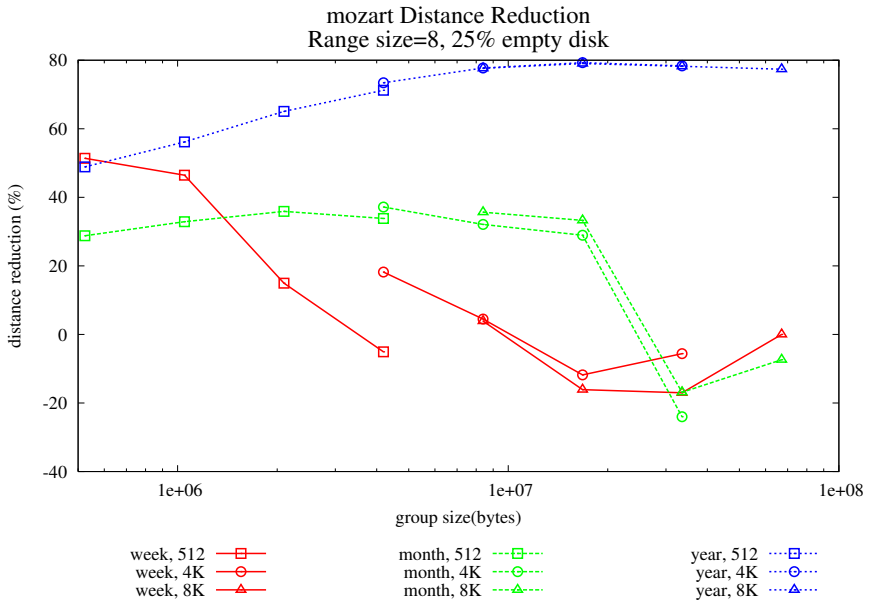
Distance reduction is among the earliest data layout strategies studied. While we expect transitions to play a larger role in modern disk drives, a generally applicable strategy should reduce distance as well. *SPORe* results indicate that distance reductions are more difficult to generalize. For sufficiently large traces, using limited disk space (25%), distance reductions actually outpace transition reductions. However, it takes time to learn and act upon observed patterns. For shorter *mozart* traces, we typically show an increase in distance. This is largely due to the empty tracks inserted in order to allow replication. For instance, inserting one empty track for every 3 raw tracks corresponds to a 25% empty disk, but results in an expected 33% increase in track distance. As a result, *SPORe* needs to exhibit approximately 33% reduction in distance in order to break even with the original, unexpanded trace. Figure 48 shows the reductions for the *mozart* week, month, and year length traces on a 25% empty disk, while Figure 50(a) shows the reductions for the *hplajw* trace for a 25% empty disk. In order to better visualize the impact of our strategy on a single trace, we summed track distance within windows of 2 million accesses on the *mozart* year trace. Figure 52 shows that *SPORe* has a dramatic impact after 50 million accesses, or less than 20% of the total length of the trace.

The problem of additional distance due to empty track insertion is compounded when using a 75% empty disk, resulting in an expected 300% increase in track distance. In this case, even cutting the distance in half using the expanded trace would result in doubling the distance of the original, unexpanded trace. Only the year length *mozart* trace supplies sufficient time for *SPORe* to exhibit a benefit under these conditions, as shown in Figure 49.

Once again, even in the face of high uncertainty, our own *ranin* traces show reductions. The only trace not to do so was the day trace, for 8192 blocks per group (all block sizes). Figure 51 shows our distance reductions for these traces.

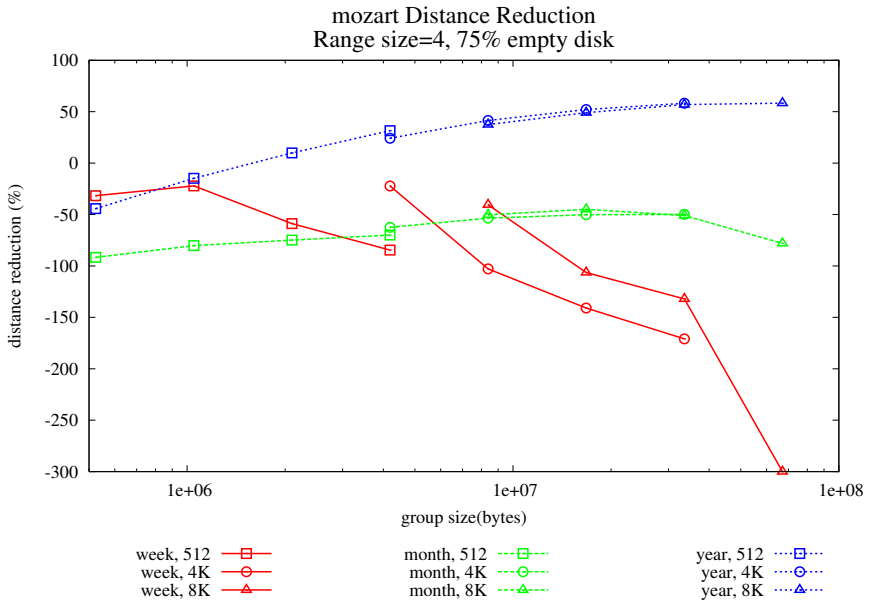


(a) *Four groups per range*

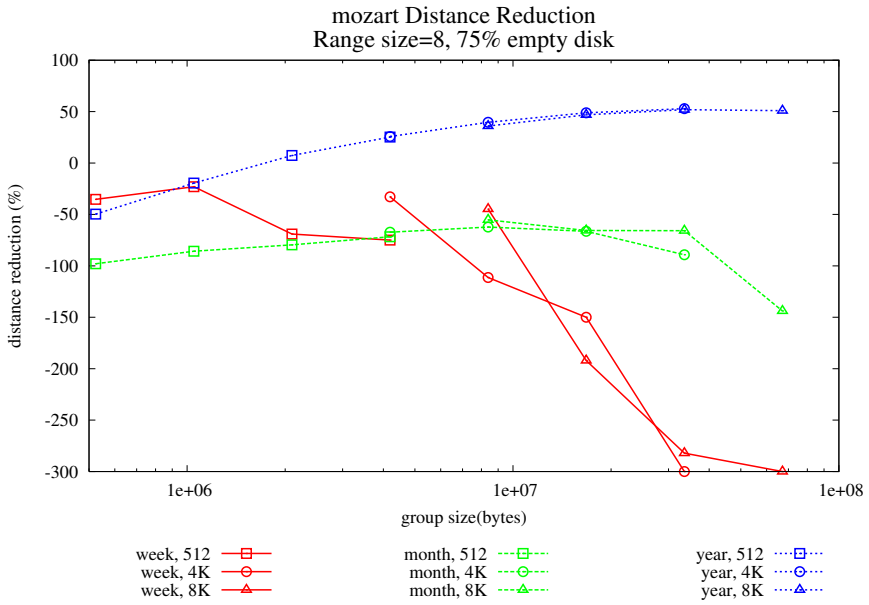


(b) *Eight groups per range*

Figure 48: *SPORe* distance reductions for 25% empty disk, *mozart* traces. Included are the week, month, and year length traces, with block sizes of 512 bytes, 4 KB, and 8 KB. Distance reductions are based on ungrouped, compact trace with no empty groups.

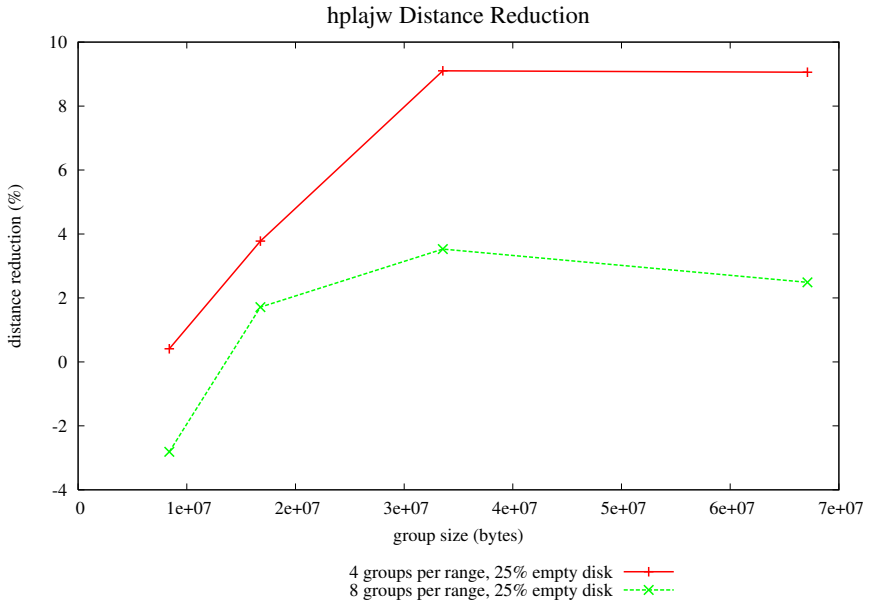


(a) *Four groups per range*

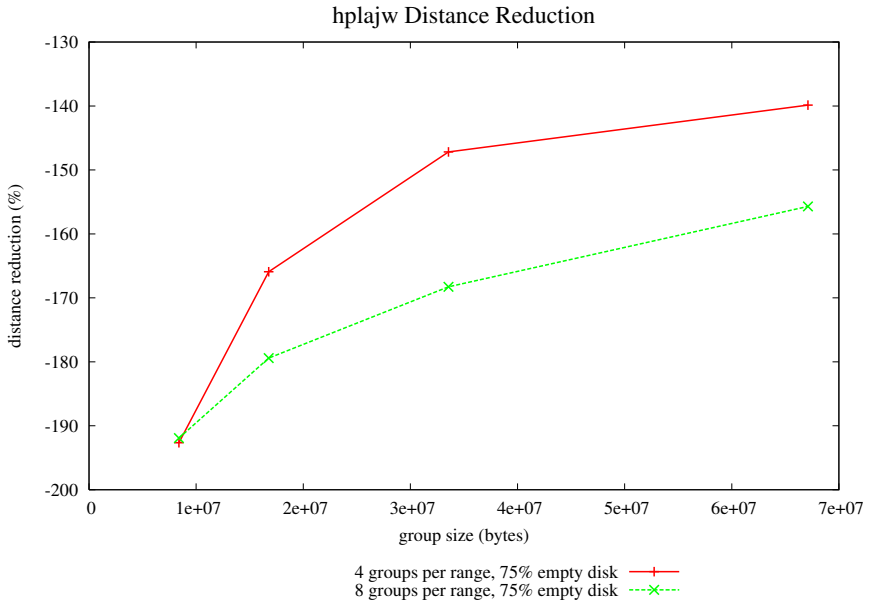


(b) *Eight groups per range*

Figure 49: *SPORe* distance reductions for 75% empty disk, *mozart* traces Included are the week, month, and year length traces, with block sizes of 512 bytes, 4 KB, and 8 KB. Distance reductions are based on ungrouped, compact trace with no empty groups.

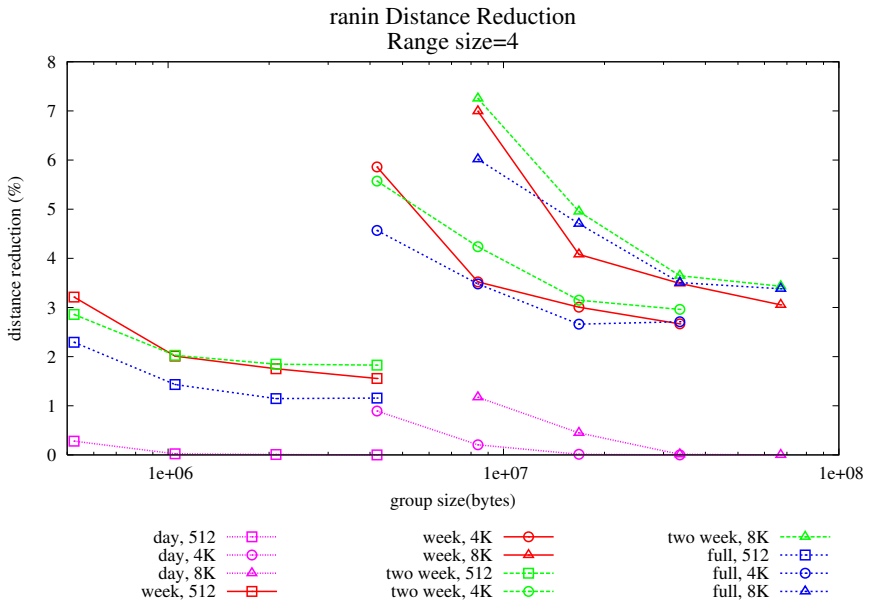


(a) *25% empty disk*

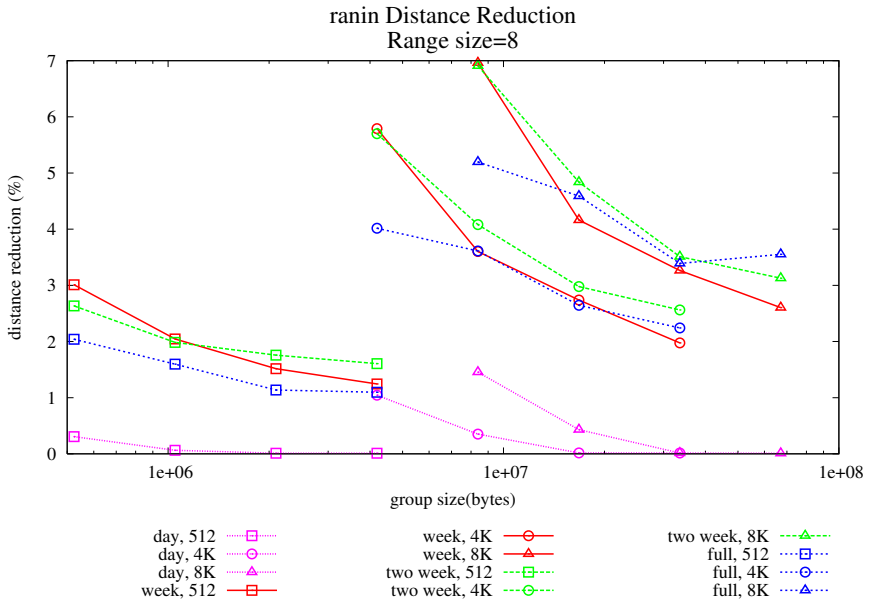


(b) *75% empty disk*

Figure 50: *SPORe* distance reductions for *hplajw* traces, with 25% and 75% empty disk. Distance reductions are based on ungrouped, compact trace with no empty groups. Note that the 75% empty disk has increased distance for all group sizes.



(a) *Four groups per range*



(b) *Eight groups per range*

Figure 51: *SPORe* distance reductions for *ranin* traces. Included are the week, two week, and full-length traces, with block sizes of 512 bytes, 4 KB, and 8 KB.

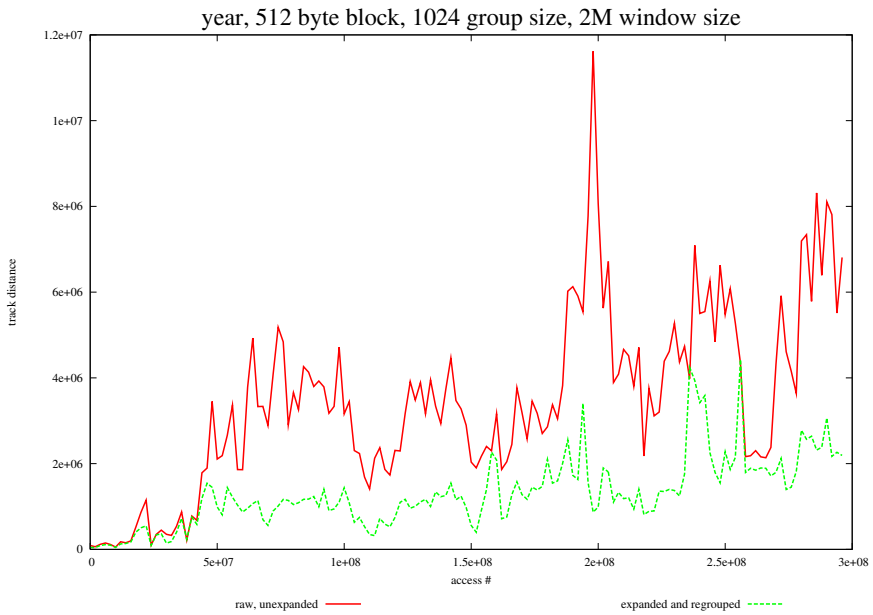


Figure 52: Windowed track distance of *mozart* year trace, 512 byte blocks, 1024 objects per group. We used a window size of 2 million accesses. Notice that we see a dramatic decrease in distance after less than 20% of the trace.

9.3.3 Accesses and Group Usage

In order to show the effect of our predictive groups, we tracked usage counts throughout our regrouping simulation, both for "raw" groups as well as predictive groups. These counts were used to calculate the average number of accesses per use of each group type. This is similar to the average accesses per transition (Figures 13 and 14) from Section 6.4.1. The slight difference is that a *transition* occurs when the disk head moves, including writes. A *use* occurs only when a track is read. This provides us with a metric with less ambiguity than transitions.

Figure 53 shows indicative comparisons of the *mozart* month and year traces with varying group sizes, while Figure 54 shows the *hplajw* trace. These results compare the predictive groups formed in *SPORe* to both the untouched, original groups, both within the original trace as well as within *SPORe*. In almost all cases, our predictive groups show much higher average accesses

per use than sequential tracks. Interestingly, for smaller group sizes, we also see an increase in sequential group performance as well. We project that this is due to a very simple expectation. Groups that are poorly formed are likely to have very low accesses per use; hence, these groups are likely to cause a transition. These transitions are likely absorbed by our predictive groups, allowing well-formed sequential groups to be predominantly used.

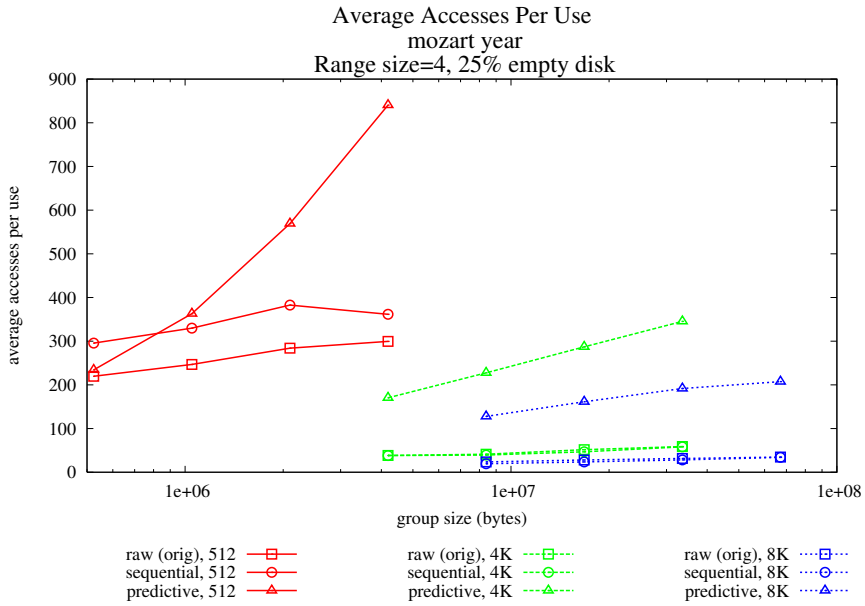
Additionally, we note that our predictive groups tend to be entered predominantly by scanning, as we discussed in Section 9.2.4. This causes a predictive group to commonly be entered on an object that the group has not been optimized for (*i.e.* a non-root). Even in the face of this limitation, our predictive grouping scheme is powerful enough to maintain high accuracy and usage.

Figure 55 shows the indicative impact of *SPORe* on our own *ranin* trace set. In the face of high uncertainty in the workload, coupled with common sub-optimized entrance via scanning, our predictive groups tend to have much lower accesses per transition, yet commonly manage to allow for better sequential group usage.

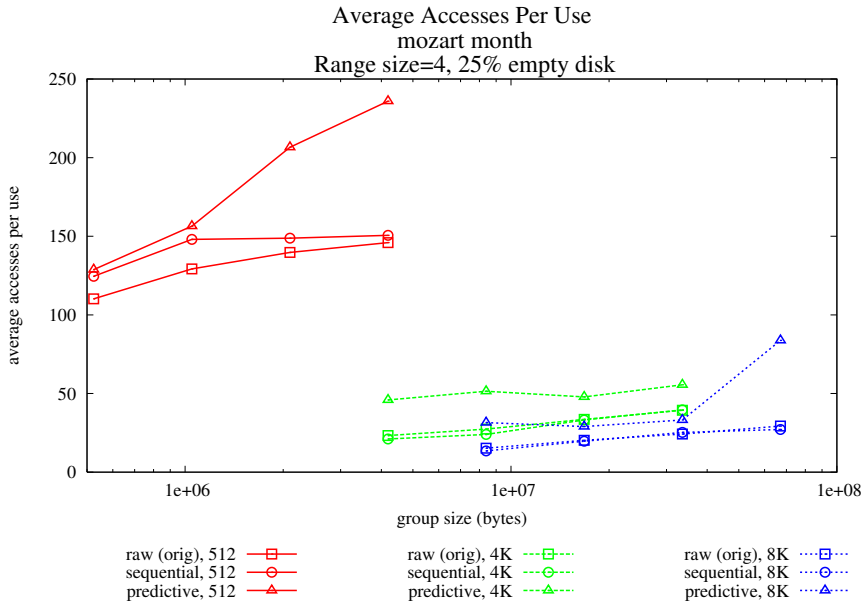
9.3.4 Updating and Storage System Overhead

Our results also show a dramatic decrease in necessary writes by using our group difference strategy, with typical reductions of about 80%. We show this result by tracking how many updates were attempted as well as how many were committed to the disk. Ultimately, we would like to show the impact of this reduction on the total number of transitions. We estimate this by adding the updates that were *not* committed. This is a reasonable estimate because most differences between group versions are expected to be deep predictions with little overall chance of occurrence; hence, they are not expected to have great impact on the system performance.

Figure 56 shows indicative results for update reductions for the *mozart* year and month traces, while Figure 57 exhibits indicative results for the *hplajw* trace and Figure 58 for our own full *ranin* trace. We note that, in most cases, a reduction in updates is necessary to show a net gain in the number of transitions.



(a) *year trace*



(b) *month trace*

Figure 53: Average accesses per group use for *SPORe*, *mozart* traces.

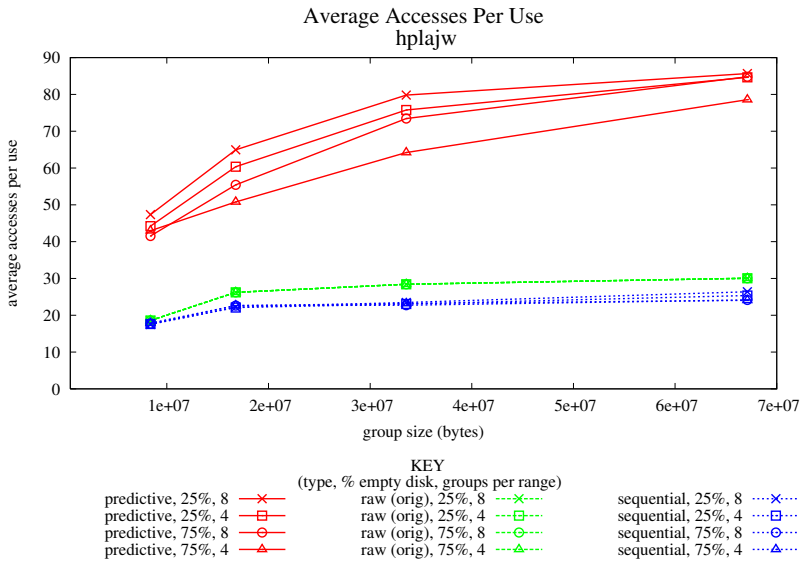


Figure 54: Average accesses per group use for *SPORe*, full *hplajw* trace.

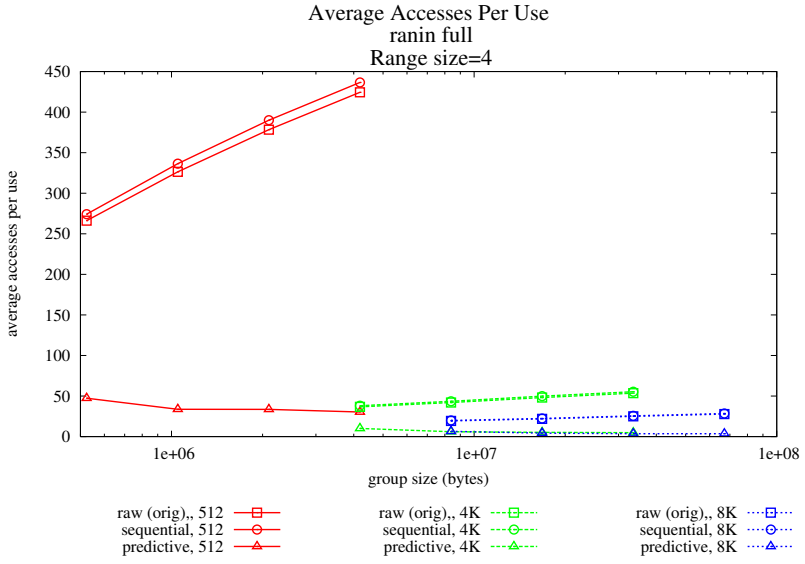
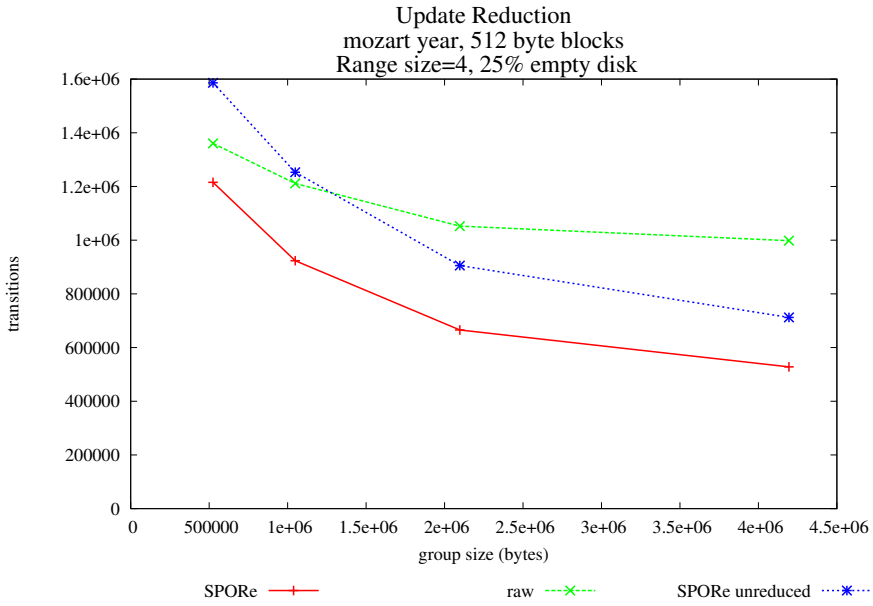
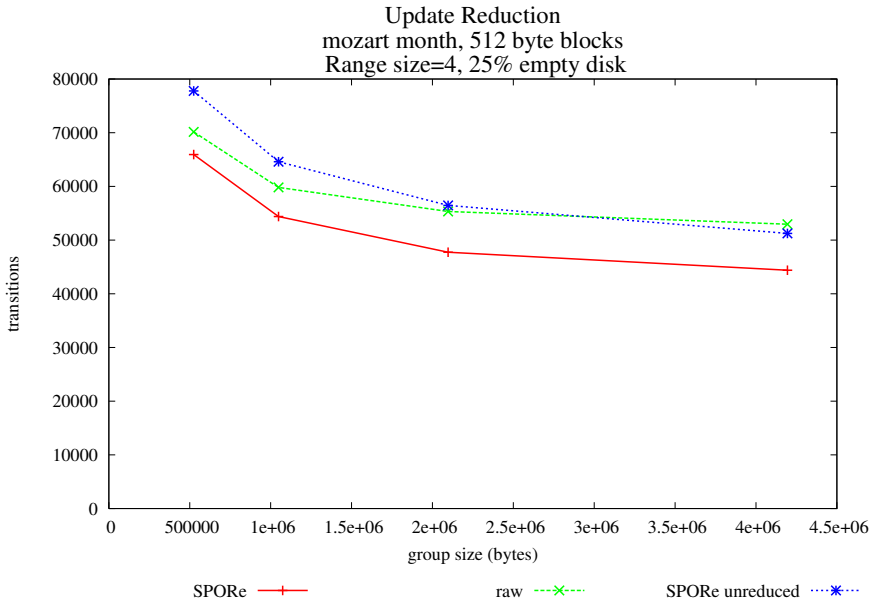


Figure 55: Average accesses per group use for *SPORe*, full *ranin* trace.



(a) *year trace*



(b) *month trace*

Figure 56: Estimated impact of update reduction for *SPORe*, *mozart* traces.

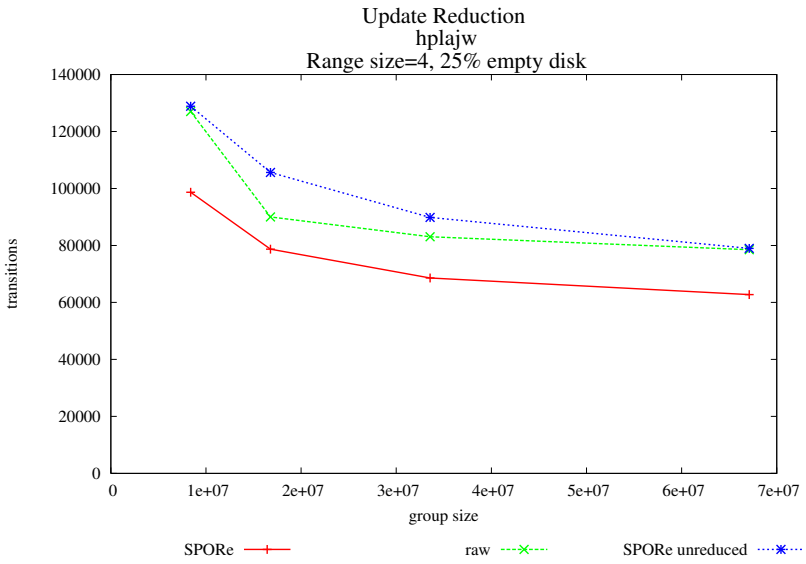


Figure 57: Estimated impact of update reduction for *SPORe*, *hplajw* trace.

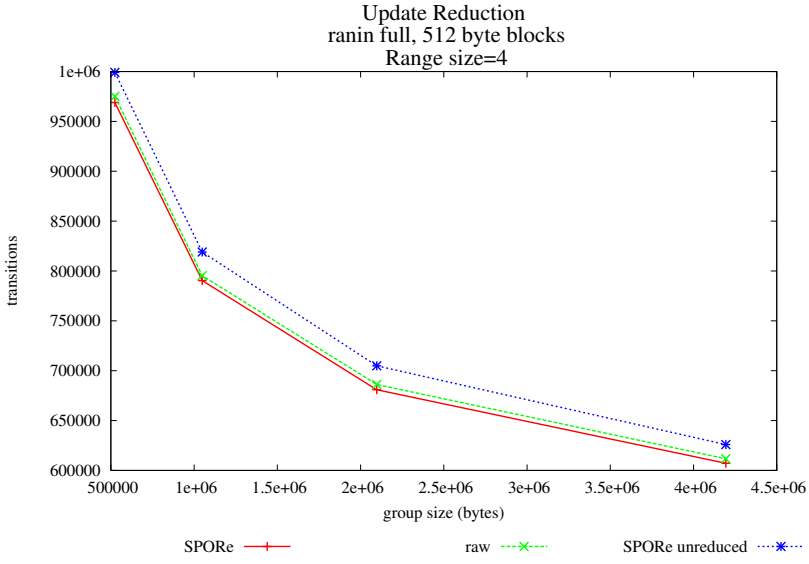


Figure 58: Estimated impact of update reduction for *SPORe*, full *ranin* trace.

Table 15: Subset of trace parameters for throughput of *SPORe*. Due to the observed limited impact of range size and disk free space, all traces used the same parameters. In particular, we used a range size of 4 and, for *mozart* and *hplajw*, 25% empty disks.

Trace	Block Size (bytes)	Group Size (blocks)	Group Size	Throughput (blocks / sec)
mozart, year	512	1024	512 KB	133,000
mozart, year	512	2048	1 MB	113,000
mozart, year	512	4096	2 MB	86,000
mozart, year	512	8192	4 MB	62,000
mozart, year	4096	1024	4 MB	82,000
mozart, year	4096	2048	8 MB	65,000
mozart, year	4096	4096	16 MB	36,000
mozart, year	4096	8192	32 MB	19,000
mozart, year	8192	1024	8 MB	56,000
mozart, year	8192	2048	16 MB	41,000
mozart, year	8192	4096	32 MB	24,000
mozart, year	8192	8192	64 MB	12,000
ranin, full	512	1024	512 KB	145,000
ranin, full	512	2048	1 MB	151,000
ranin, full	512	4096	2 MB	149,000
ranin, full	512	8192	4 MB	132,000
ranin, full	4096	1024	4 MB	83,000
ranin, full	4096	2048	8 MB	66,000
ranin, full	4096	4096	16 MB	57,000
ranin, full	4096	8192	32 MB	44,000
ranin, full	8192	1024	8 MB	52,000
ranin, full	8192	2048	16 MB	46,000
ranin, full	8192	4096	32 MB	34,000
ranin, full	8192	8192	64 MB	25,000

9.3.5 Throughput

Estimating the computational overhead of a predictive engine is non-trivial and greatly depends on a number of factors, including hardware, specific operating system and file system version number, and characteristics of the workload. Additionally, the specific workload contributes to the exact behavior of the system. Idle periods within the workload play a significant role, especially when estimating CPU overhead of a system. Personal computers can sit unused for hours at a

time, and recent work indicates that servers and data centers typically experience about 20-30% utilization [13, 16, 89]. Our own 5400 RPM Hitachi Travelstar 2.5 inch SATA drive has a reported peak transfer rate of 665 Mb/s [1], or about 79.27 MB/s, yet only 123.79 GB was read during our entire *ranin* workload, averaging 3.25 GB of data per day. This translates to an aggregate demanded transfer rate less than 0.005% of the drive's maximum.

Table 16: Subset of multi-run trace parameters for throughput of *SPORe*. Due to the observed limited impact of range size and disk free space, all traces used the same parameters. In particular, we used a range size of 4 and, for *mozart* and *hplajw*, 25% empty disks. The reported mean throughput is in blocks per second, with 99% confidence interval expressed as a percentage of the mean.

Trace	Block Size (bytes)	Group Size (blocks)	Mean Throughput (blocks / sec)	Confidence Interval (%)
mozart, month	512	1024	103,000	95.5–104.5
mozart, month	4096	1024	34,000	96.9–103.1
mozart, month	8192	1024	20,000	96.5–103.5
mozart, month	512	8192	38,000	99.8–100.2
mozart, month	4096	8192	10,000	99.8–100.2
mozart, month	8192	8192	21,000	99.8–100.2
hplajw	8192	1024	23,000	99.6–100.4
hplajw	8192	8192	4,000	99.5–100.5
ranin, day	512	1024	137,000	99.9–100.1
ranin, day	4096	1024	128,000	99.8–100.2
ranin, day	8192	1024	87,000	99.8–100.2
ranin, day	512	8192	173,000	99.8–100.2
ranin, day	4096	8192	161,000	99.9–100.1
ranin, day	8192	8192	153,000	99.9–100.1

In order to exhibit generalizable results, we present the throughput demonstrated by *SPORe*, in terms of blocks per second, on our test bed system. Each test reported throughout this chapter was timed using the standard `time` command found on Mac OS X. These times included all programs used in our script test bed, including decompressing the trace files; however, the overhead of these additional scripted commands we consider minimal. A subset of these results for longer traces (*mozart* year trace and *ranin* full trace) are provided in Table 15. In addition, we tested a subset of traces and parameters on multiple runs in order to establish confidence intervals. The set of

workloads and parameters timed in this way is provided in Table 16, along with mean throughput and corresponding 99% confidence intervals. We chose only traces that lasted longer than 25 seconds in order to help ensure consistency. Each trace in this multi-run subset was timed 30 times using the standard `time` command found on Mac OS X.

In order to calculate throughput, we divided the total number of block accesses in the trace by the total CPU time, which is calculated as the sum of the reported user and system time, shown in Equation (15).

$$throughput = \frac{total\ block\ accesses}{user\ time\ +\ system\ time} \tag{15}$$

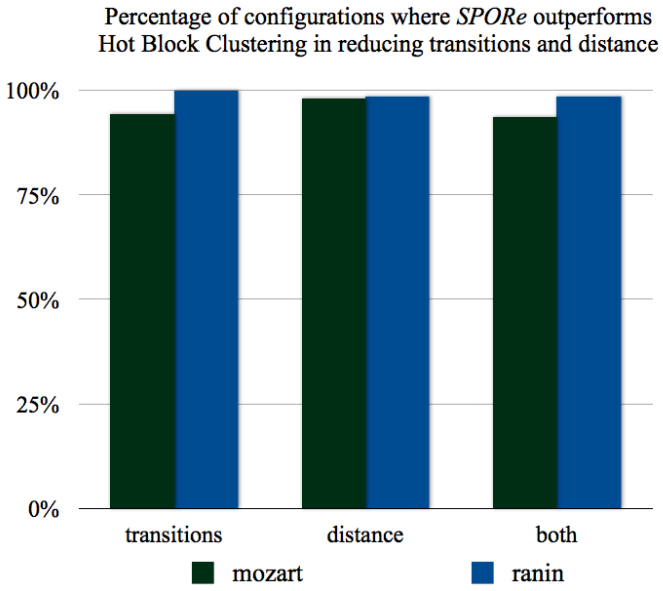
The 99% confidence interval was then calculated using the student's t-distribution.

9.3.6 Comparison against Hot Block Clustering

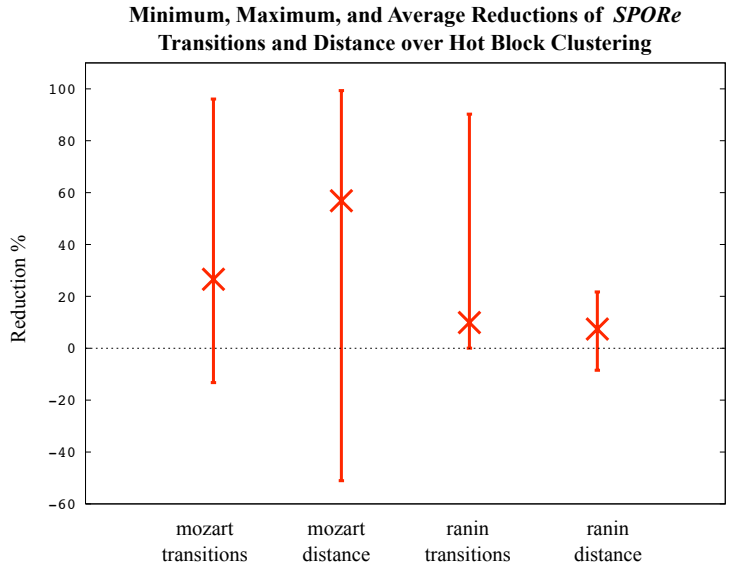
We compared our implementation of on-disk caching to *SPORe* in terms of transitions as well as distance (see Figure 59). In particular, we compared against *SPORe* with a range size of 4 and, for the *mozart* traces, a 25% empty disk. We found that *SPORe* outperformed on-disk caching in terms of both transitions and distance for 93.7% of *mozart* configurations and 98.7% of *ranin* configurations. On average, *SPORe* showed a 26.6% reduction in transitions and a 56.9% reduction in distance over on-disk caching for *mozart traces*, and a 9.9% reduction in transitions and 7.4% reduction in distance for the *ranin* traces. Table 17 shows these results separated by trace and block size.

9.4 DISCUSSION

Recall our original goals and motivations for a dynamic data grouper. We sought an engine, dynamic and adaptive, that was robust, sustainable, and resilient, aimed at predictions that are not only *accurate*, but also *persistent*, unlikely to change quickly, and avoid "knee-jerk" reactions that become stale before they can be used. In an effort to improve upon the basic goal of prefetching (*i.e.* satisfying requests before they are made) we have intended to decouple the satisfying strategy from the data path by employing predictions that are expected to endure. These predictions, by



(a) *Percentage of configurations where SPORe outperforms on-disk caching*



(b) *Average, max, and min improvements of SPORe over on-disk caching*

Figure 59: *SPORe* compared with on-disk caching (hot block clustering).

Table 17: Percentages of configurations where *SPORe* outperforms Hot Block Clustering, broken down by trace and block size.

Trace	Block Size (Bytes)	Percentage (for Transitions)	Average Transition Reduction	Percentage (for Distance)	Average Distance Reduction
mozart, week	512	95.4	28.7	100	60.1
mozart, week	4096	92.6	23.4	98.1	64.5
mozart, week	8192	84.3	20.9	100	72.3
mozart, month	512	100	26.7	100	44.8
mozart, month	4096	100	21.6	100	60.7
mozart, month	8192	100	18.0	100	66.4
mozart, year	512	77.8	3.6	86.1	15.8
mozart, year	4096	100	27.4	100	54.4
mozart, year	8192	100	26.1	100	60.3
ranin, day	512	100	18.5	100	8.5
ranin, day	4096	100	6.3	100	3.4
ranin, day	8192	100	4.5	87.0	-11.9
ranin, week	512	100	18.7	100	10.5
ranin, week	4096	100	7.0	100	8.4
ranin, week	8192	100	5.3	100	8.0
ranin, two week	512	100	17.5	100	9.3
ranin, two week	4096	100	6.5	100	8.6
ranin, two week	8192	100	4.8	100	9.1
ranin, full	512	100	17.1	100	6.4
ranin, full	4096	100	7.1	100	6.8
ranin, full	8192	100	5.7	97.2	4.5

the nature of their expected persistence, lend themselves to *persistent replication* on the storage system, rather than the transient replication of caching and memory management.

Such persistent replication maintains the essence of prefetching implicitly without requiring immediate action. This provides several improvements over prefetch caching. First, replication done in the past has value. Caching, by its nature, is transient and short-lived. Work that is done at the caching level is expected to be short lived, and predictions made long before the current moment of execution is unlikely to have survived until now. Such is not the case with persistent predictions, which allow for future uses of the same set of collocated, replicated data. As a result, the system builds upon itself, improving over time. Additionally, it allows for serendipitous use. Where prefetch caching must be useful *now*, persistent replication allows for extended utility *later*, even when we may not have expected it. Arguably, this engineered serendipity is the greatest strength of data layout management.

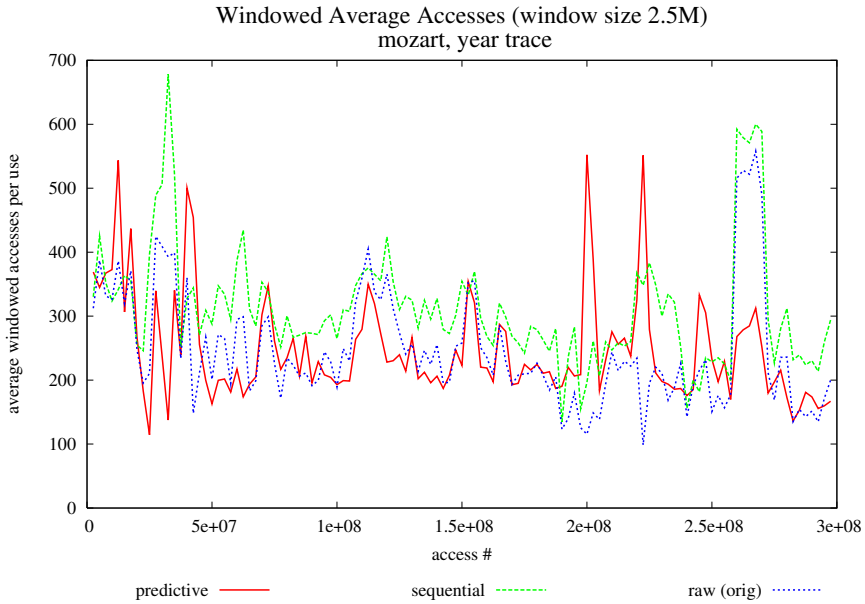
9.4.1 Persistence of Predictions

In order to verify that our predictions continue to be useful, we altered our *SPORe* engine to discontinue prediction after a predetermined number of accesses. In practice, the decision to disable replication should be made dynamically, and potentially for a variety of reasons. For instance, in the case of high activity, low predictability, or observing the underlying hardware as idle. For verifying the intuition that persistent predictions remain valuable, however, requires manual control.

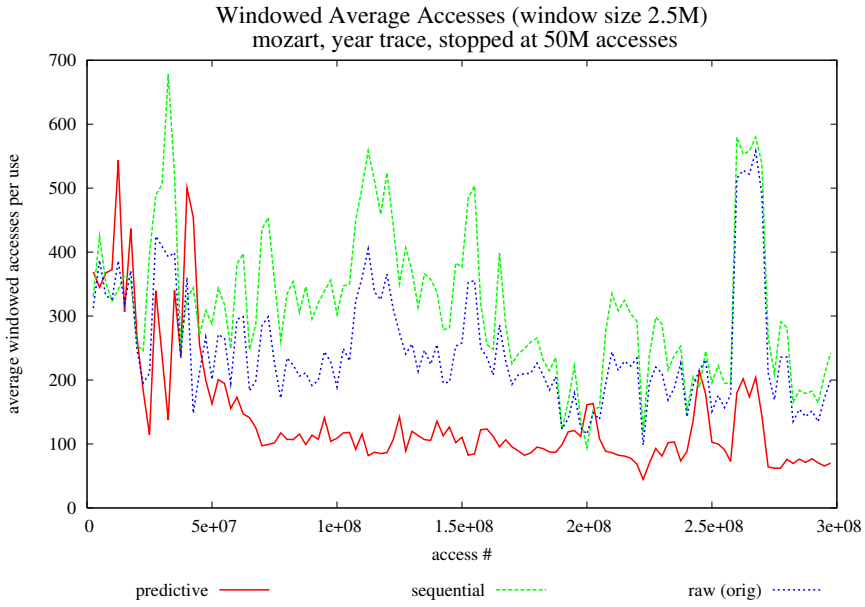
In order to verify predictive groups' persistent value, we ran the *mozart* year trace with a block size of 512 bytes, group size of 1024, and a range size of 4 on the simulated 25% empty disk trace. This workload and configuration showed approximately 50% reduction in distance and about 11% reduction in transitions. Figure 60(a) shows the windowed average accesses per use for this run *without* halting predictions. Figure 60(b) shows the same run with the same parameter set, halting predictions after 50 million accesses. At the end of this workload, after acting upon only one sixth of the entire trace, we observed a 25% reduction in distance and a 3.3% reduction in transitions. Of particular note is the behavior of the sequential groups in Figure 60(b). Notice that almost immediately after ceasing prediction, our predictive group behavior stabilizes well below that of both the raw groups as well as the sequential groups within *SPORe*. However, the behavior of sequential groups significantly improves. This supports our earlier observation that our predictive groups tend to absorb the "difficult" locations within the workload, leaving well-formed sequential groups as they are, and continues to do so long after we cease to actively predict and update groups.

9.4.2 Robustness to Track Size

The motivating hardware example used throughout our work has been equating *groups* with disk *tracks*. We contend that knowing this track size, while certainly useful and feasible [104], is not strictly necessary. In order to test this, we ran the *mozart* month trace with 512 byte blocks and simulated 25% empty disk with multiple parameter settings, changing the actual track size and the presumed track size, or the group size used by *SPORe*. Note that, since *SPORe* uses only groups that are completely empty for predicting, when the actual size of tracks is smaller than the presumed, no prediction takes place, since no empty tracks are found. We calculated the transition reduction as well as the distance reduction, both in terms of actual track size, shown in Tables 19



(a) *windowed average accesses per use*



(b) *windowed average accesses per use, stopped at 50M accesses*

Figure 60: Windowed comparison of vanilla and stopped *SPORe* for *mozart*, year trace. Block size is 512 bytes, group size is 1024, range size is 4, disk is simulated 25% empty. Window size is 2.5 million accesses. Trace stopped prediction at 50 million accesses.

Table 18: Comparison of reduced distance percentages for presumed vs. actual track sizes for *SPORe* on *mozart* month trace, 512 byte blocks.

	Presumed			
Actual	1024	2048	4096	8192
1024	67.7	-	-	-
2048	66.8	64.9	-	-
4096	67.2	63.7	61.7	-
8192	69.6	65.1	63.2	59.7

Table 19: Comparison of reduced transition percentages for presumed vs. actual track sizes for *SPORe* on *mozart* month trace, 512 byte blocks.

	Presumed			
Actual	1024	2048	4096	8192
1024	94.0	-	-	-
2048	96.5	90.9	-	-
4096	95.2	91.3	86.3	-
8192	95.7	91.2	86.2	83.8

and 18, respectively. These results indicate that, as long as *SPORe*'s group size is at least as big as the actual track size, we can expect similar benefit.

9.4.3 Confidence Thresholds

Workloads change, shift, and expand; it is inevitable. This behavior is the very essence of what makes automated adaptability so powerful. Just as we expect storage system demand to be "bursty" in nature [14, 16, 91, 94, 95, 102], so do we expect changes to occur in bursts. One undesirable

scenario that we seek to avoid is predicting (and committing to the device) groups that use new, immature information. Such "knee-jerk" predictions may be of limited use; it would be better if we were able to wait until our predicting engine was operating with a higher confidence to perform predictions.

To accomplish this, we attempted the use of a confidence threshold. At each block request, we predict the most likely expected block, given the previous request. This request actually generates two return values, the predicted block, and the confidence of that prediction. This confidence is exactly the same $P(T, f, s)$ used in our expansion functions. We keep a running prediction confidence, r, using the following equation.

$$r = \alpha \times r + (1 - \alpha) \times P(T, f, s) \tag{16}$$

Notice that this equation is *not* a global average confidence. Rather, it is an approximation of such an average, but with a strong bias towards recent predictions, with lower values of α corresponding to stronger bias. In practice, we used the value 0.99 for α. Using this running confidence, we are able to avoid attempting to predict, and therefore avoid poor device-level predictive updates, when we become skeptical of our predictive engine. In practice, this strategy was actually outweighed by our other update reduction strategies. Our initial tests indicated that reducing this threshold to zero actually improved performance; thus, our final results do not include use of a running confidence.

10.0 HARDWARE-BASED VALIDATION

Much of our work revolves around *generalizable application*. Adaptable, dynamic systems are more generalizable by their very nature; their ability to change and adapt allows them to tailor themselves to differing workloads and patterns. Operating at the block level within a system, without knowledge of requesting processes or file names, affords more generalizable strategies to be applied. Even robustness can be viewed as a particular kind on generalizability to withstand adversarial workloads.

Keeping in this spirit of extensive application, our results have largely been gathered via simulation. In particular, the two most common results we present are *transitions* and *distance*. These are generally accepted as the two metrics most closely associated with latency and energy costs of a storage system's underlying mechanical hardware. By presenting simultaneous reductions on these two metrics, we provide a means by which many hardware system configurations can be evaluated by approximation. The only knowledge required to do so is an estimating function of system performance based on transition count and distance.

In order to validate our results, we present in this chapter the design and analysis of a prototype hardware system configuration. We have continued to use the hard drive as a motivating hardware example, and have stressed the number of transitions as the expected primary indicator of hardware performance. We test these examples and verify our conjectures in our prototype system by timing a selected subset of the workloads and parameter settings from our *SPORe* experiments while simultaneously sampling the isolated power consumption of test drive mechanical components.

10.1 EXPERIMENTAL SETUP AND DESIGN

We obtained our results from Chapter 9 via simulation in order to present generalizable conclusions while minimizing ties to any particular hardware configuration. These results are based on assumptions about storage system behavior. In particular, we assume the most likely indicator of system energy cost and latency to be the number of group transitions; the second most likely indicator we assume to be track distance. While these metrics are generally accepted as accurate indicators, they are only *estimates*, rather than empirical measurements of real systems.

In order to accurately validate these metrics, a high sampling rate is required. Some disk arm actuators are capable of moving the physical arm from on size of the disk to the other within milliseconds; we will need to sample at a rate fast enough to capture these very rapid changes.

Additionally, we need to minimize interference of existing energy and latency reduction techniques, including caching and disk spin-down strategies. Keeping the disk busy should ensure that no spin-down occurs, but avoiding the cache can be more difficult. Complicating this issue is our recurring motivating hardware example of disk tracks. The original motivation was that a track buffer would be accessed more frequently for well-organized groups, leading to fewer device-level accesses. But this buffer is a cache; steps taken to avoid caches altogether would avoid the buffer as well. Thus, we need to provide some mechanism for modeling track buffer behavior.

10.1.1 System Configuration

All of our experiments were conducted using two test machines, an external hard drive enclosure, and a DAQ. Two test machines were used in order to prevent the recording of voltage drops from interfering with the data request stream going to the test drive. The workload replay system used the same MacBook test machine from our *SPORe* experiments, running OS X 10.5.8 with the same Darwin and XNU versions, 2.4 GHz Intel Core 2 Duo processor, 2 GB of 667 MHz DDR2 SDRAM, 3 MB of L2 cache, and an 800 MHz system bus. The internal hard drive containing the trace set was the same 5400 RPM Hitachi Travelstar 2.5 inch SATA drive with 160 GB capacity and an 8 MB cache. The voltage measurement workstation was a PowerBook running Mac OS X 10.3.9 with vanilla Darwin and XNU kernel, 867 MHz G4 processor, 640 MB of 333 MHz DDR

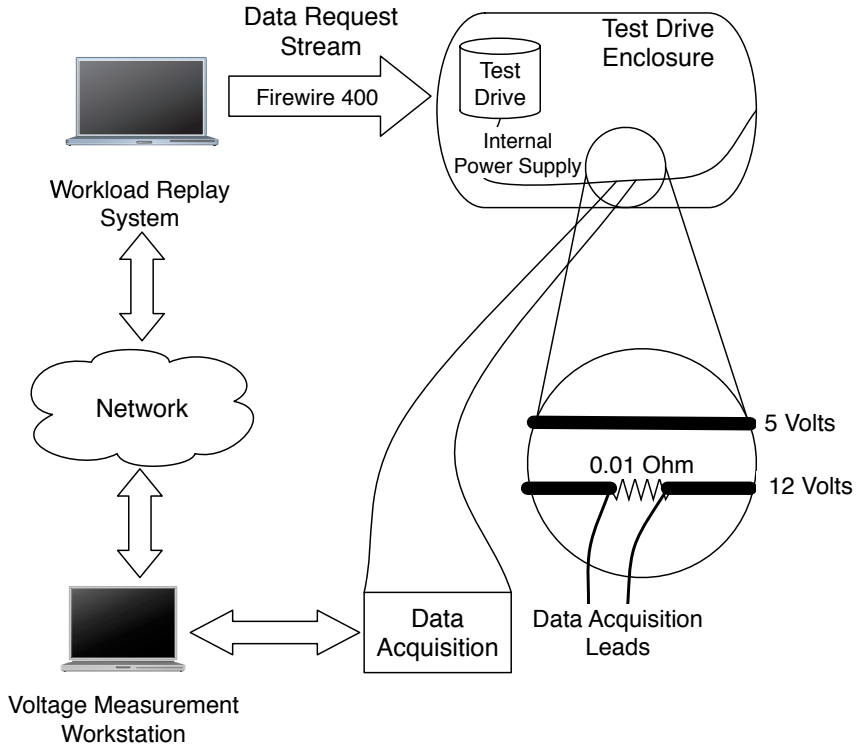


Figure 61: Prototype hardware *SPORe* evaluation system.

SDRAM, 256 KB of L2 cache, and a 133 MHZ system bus. The internal hard drive used to store the voltage histories was a 4200 RPM Fujitsu Mobile MHS 2.5 inch ATA-100 drive with a capacity of 60 GB and a 2 MB cache.

There were three IDE (PATA) drives used to measure latency and energy costs. We used a 320 GB Hitachi Deskstar 7200 RPM with 8 MB cache, a 250 GB Samsung SpinPoint 5400 RPM with a 2 MB cache, and a Western Digital 320 GB Caviar Blue 7200 RPM with an 8 MB cache. The test drives were placed into an external drive enclosure and connected to the driving system via Firewire 400. The Data Acquisition (DAQ) unit used was a National Instruments cRIO-9215 with a National Instruments USB-9161 USB Carrier.

In order to accurately measure power of the test drives, we isolated the mechanical components of the drive. All of our test drives were selected to have separate 5 Volt and 12 Volt lines on the

internal IDE power supply. The power consumption of the mechanical components of the drive was measured by sampling the voltage drop across a 0.01 Ω resistor in series with the 12 Volt line. These samples were taken at 20,000 samples per second using a DAQ system [21]. Figure 61 diagrams our prototype system.

Following trace replication suggestions by Gray and Shenoy [118], all experiment runs were scripted in order to reduce timing errors. Typical experiments would prep a trace replay script, the *SPORe* engine, simulated track buffer (described in Section 10.1.3), and replay driver on the workload replay system (detailed in Section 10.1.4), then issue a command via `ssh` over the local wireless network to the voltage measurement workstation. This required a slight alteration to our *SPORe* engine to allow a flag to output the necessary I/O operations rather than gather statistics *about* those operations. The voltage measurement workstation would then prepare the DAQ software with appropriate settings, issue an *ssh* command to the workload replay system to begin replaying the trace, and immediately begin recording voltages. While the voltage measurement workstation was responsible for recording voltage drops, the workload replay system was responsible for recording the length of time the workload took to replay using the standard `time` command on Mac OS X. For timing reasons, we would kill any `ssh` command that lasted longer than 20 seconds and restart the experiment.

As with our *SPORe* project, all programs were implemented in C or C++ compiled with the default versions of the GNU project `gcc` and `g++` compilers. To ensure correct program behavior, no optimization flags were used during project compiling.

10.1.2 Traces

We selected a subset of *mozart* workloads and parameters from our previous *SPORe* experiment simulations for validation. For hardware testing, again following suggestions by Gray and Shenoy [118], we selected only traces that had sufficient run time to reach a stable state. The shortest trace and parameter set we ran was the *mozart* month trace with simulated 8 KB blocks and 8192 blocks per group. This trace had an average run time more than six minutes. A complete list of workload parameters is given in Table 20.

Most trace sets were run 3 times in order to calculate an average and 99% confidence interval.

Table 20: Trace and parameter set tested on prototype hardware. The drives tested are a 320 GB Hitachi Deskstar (HIT), a 250 GB Samsung SpinPoint (SAM), and a 320 GB Western Digital Caviar Blue (WD).

Trace	Block Size (bytes)	Group Size (# blocks)	# Runs	Drives Tested
mozart, year	512	1024	3	WD
mozart, year	4096	1024	3	WD
mozart, year	8192	1024	3	WD
mozart, year	512	8192	3	WD
mozart, year	4096	8192	3	WD
mozart, year	8192	8192	3	WD
mozart, month	512	1024	3	WD
mozart, month	4096	1024	3	WD
mozart, month	8192	1024	3	WD
mozart, month	512	8192	3	WD
mozart, month	4096	8192	3	WD
mozart, month	8192	8192	30	HIT, SAM, WD

In order to establish that the variance for energy and latency is expected to be low, we ran the shortest, and therefore most variable, trace and parameter set 30 times. This trace and parameter set was test on all three test drives, while all other experiments were tested on the Western Digital Caviar Blue test drive.

10.1.3 Simulated Track Buffer

In order to replicate tracks of various sizes, we implemented a simulated track buffer to sit on the data request stream between *SPORe* and device driver. This simulated track buffer accepted a track size used to determine when a new track was necessary (due to a transition). Every new track was read in its entirety at the time of the first request. While this track size is unlikely to coincide with the test drive's actual track sizes, it is a reasonable approximation, validated by our original tests on *SPORe*'s robustness to track size. Additionally, any track boundary crossed during the read of the simulated track would be sequential in nature, causing minimal latency or power cost.

10.1.4 Avoiding Cache Interference

To avoid interference with existing caching schemes, all requests by *SPORe* were filtered through our simulated track buffer before being issued by a custom replay driver program. Upon receiving a block request, our replay driver would request the entire group containing the request, simulating a track request. These track requests were issued to the raw device; typically, the device path used was `/dev/rdisk2`, although each time a test drive was connected, this path was verified manually. By issuing raw device-level requests, we avoid going to caches or main memory, forcing the physical underlying device to satisfy the request.

10.1.5 Identifying Workload Boundaries

While scripted experiments reduce the variation of results by removing human error, not all network commands take equal time to complete. In particular, we noticed during initial experiment development that most `ssh` commands took between 3 and 7 seconds. Additionally, due to the DAQ software's necessity to fill data arrays, it was necessary to allow additional time for all pertinent voltage history information to be written to the disk. Halting the recorder immediately after

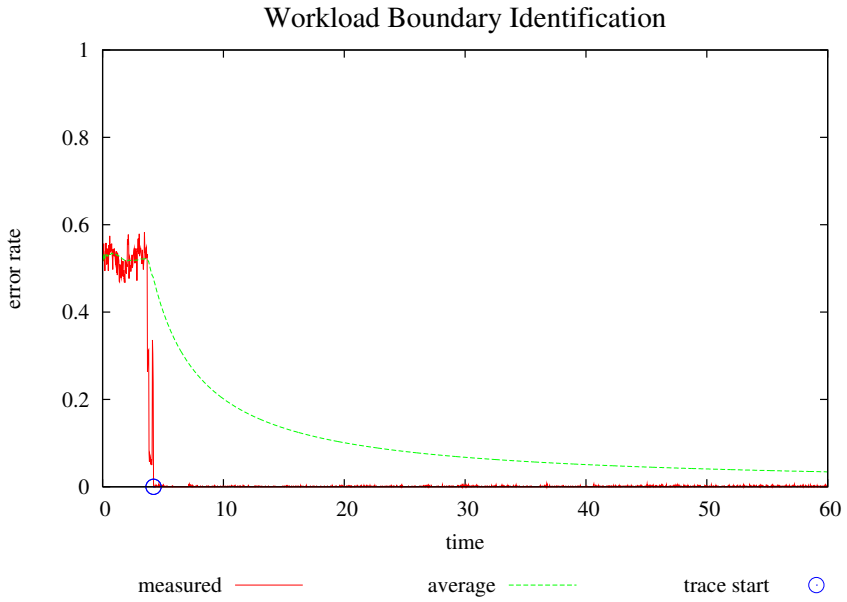


Figure 62: Identifying the beginning boundary of a workload replay by DAQ error rate. In particular, this is the first sixty seconds of the voltage history for one experiment on the Samsung test drive, *mozart* month trace, 8 KB blocks, and 8192 blocks per group.

the workload trace replayer finished would result in missing the very end of the trace. We therefore allowed the recorder to continue operation well after the workload replay had completed. This poses an interesting question: how do we separate the correct piece of the voltage history in order to calculate energy demands? In order to accomplish this, we looked at the error rate of the recorder. A resistor of any size should never have a negative voltage drop, yet they exist throughout the voltage history. While we ignore these in our energy calculation, they can be useful in identifying a more precise beginning and end to our workload.

When a drive is idle, there is very little power on the supply line to the mechanical components. Intuitively, this increases the likelihood that the DAQ will read an erroneous negative value, since voltages closer to zero are more difficult to detect. We can promote an idle device by specifically waiting for several minutes between experiments. We can then identify a precipitous drop in error

rate in order to identify an accurate beginning of the workload. To calculate this error rate, we used a window of 0.03125 seconds ($\frac{1}{32}$ of a second).

To find the "start" of a trace, we first find the last position H between 0 and 60 seconds that has error rate higher than 0.5. We then find the lowest point L between H and 60 seconds, with the restriction that the error rate at point L be less than 0.05. If no such point exists, we return the first point L' with error rate less than 0.25. Figure 62 shows a typical drop and consequential workload boundary identification around 4 seconds. In some cases, the test drive had not gone to sleep in time. In these cases, we simply use the very beginning of the voltage history as the beginning of our workload. With workload replays lasting several minutes, and a maximum wait of 20 seconds for `ssh` commands, we consider the error introduced in this way to be marginal.

Since we know when the workload began, and we have timed how long the workload took to complete, we can easily identify the end. This piece of the voltage history was extracted and used in our power calculations.

10.1.6 Calculating Power

In order to closely approximate the energy cost of the mechanical components of our test drives, we used Equation (17) and 18. We denote p as power, e as energy, v as the voltage drop measured and recorded in the voltage history, and r as the resistance of the small resistor (0.01 Ω) in series with the device on the 12 Volt line.

$$p(v) = \frac{v}{r} \times (12 - v) \tag{17}$$

$$e(v) = p(v) \times \frac{1}{20000} \tag{18}$$

Total energy, E, was calculated as a sum of each $e(v)$ from the voltage history between the identified workload boundaries.

$$E = \sum_{i=1}^{n} e(v_i) = \sum_{i=1}^{n} p(v_i) \times \frac{1}{20000} = \sum_{i=1}^{n} \frac{v_i \times (12 - v_i)}{20000 \times r_i} \tag{19}$$

Replacing r_i with 0.01 Ω in Equation (19) yields the following.

$$E = \sum_{i=1}^{n} \frac{v_i \times (12 - v_i)}{20000 \times 0.01} = \frac{1}{200} \sum_{i=1}^{n} v_i \times (12 - v_i) \tag{20}$$

10.1.7 Modeling System Energy and Latency

In order to estimate the system-wide impact of *SPORe*, we used a straightforward system model with three components: a processor, main memory, and the underlying storage device. For the processor, we considered both processor speed and maximum wattage as parameters, as well as a base percentage of processor usage. This percentage represents how CPU-bound a workload might be. For main memory, we considered total wattage as a function of wattage per gigabyte and number of gigabytes. The storage device parameters we used were identically those gathered by our hardware prototype, simulating three different hard drives.

Total system time was calculated by using the mean time for each storage device for each workload. For example, if the mean run time for some workload was sixty seconds for the Hitachi test drive, we used a system run time of sixty seconds.

Total system energy was calculated using Equation 21.

$$E_{sys} = P_{cpu} \times u_{cpu}^2 \times t_{sys} + P_{RAM} \times t_{sys} + E_{drive} \tag{21}$$

The value P_{cpu} is the peak power of the processor, u_{cpu} is the utilization of the processor expressed as a percentage. P_{RAM} is the power of main memory, or the product of wattage per gigabyte and the number of gigabytes in the system. E_{drive} was the amount of energy calculated for the trace.

To calculate u_{cpu}, we used Equation 22.

$$u_{cpu} = \frac{t_{cpu} \times \frac{F_{cpu}}{F_{base}}}{t_{real}} + u_{base} \tag{22}$$

The value t_{cpu} is the measured amount of time each workload spent in the CPU (the sum of system time and user time from the `time` command). F_{cpu} is the processor speed in GHz, and F_{base} is the speed of the processor used in our experiments (2.4 GHz). The value t_{real} is the amount of time each workload took to complete (real time from the `time` command), and u_{base} is the base utilization percentage. If u_{cpu} was found to be greater than 1 (greater than 100% CPU utilization), the value t_{sys} was adjusted accordingly. Thus, if we calculate u_{cpu} to be 1.1, we added an additional 10% to t_sys.

We tested multiple configurations, varying processor wattage between 25 and 35 W, processor speeds of 2.4 GHz and 3.3 GHz, memory power per gigabyte between 2.334 W [28] and 9.555 W [76], main memory size of 2 and 4 GB, and base utilization of 5% and 95%.

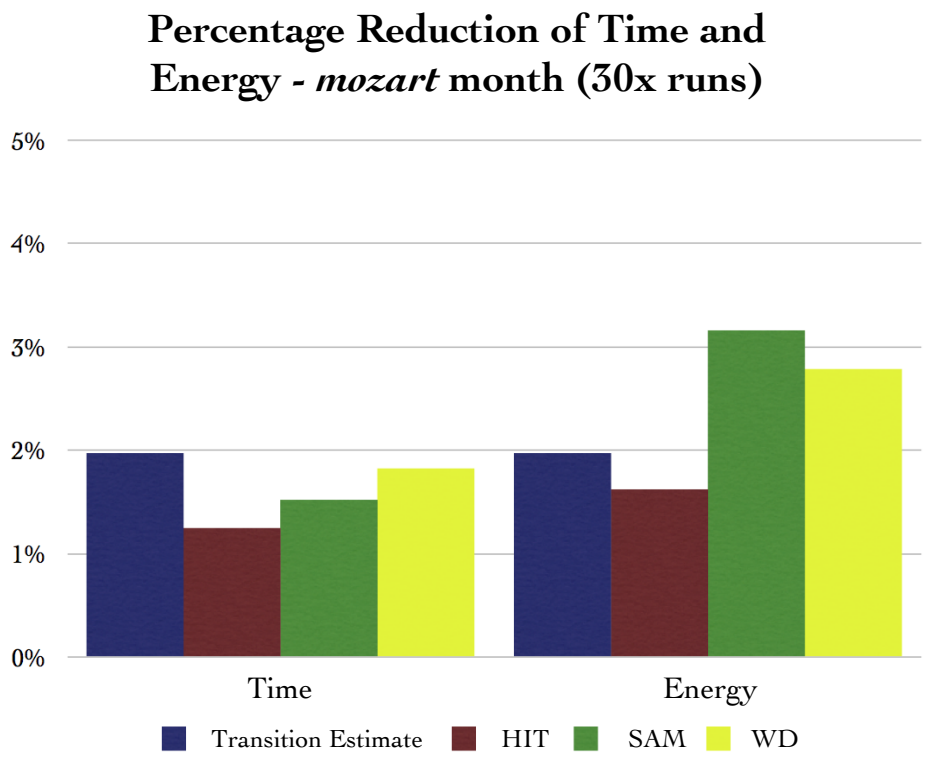
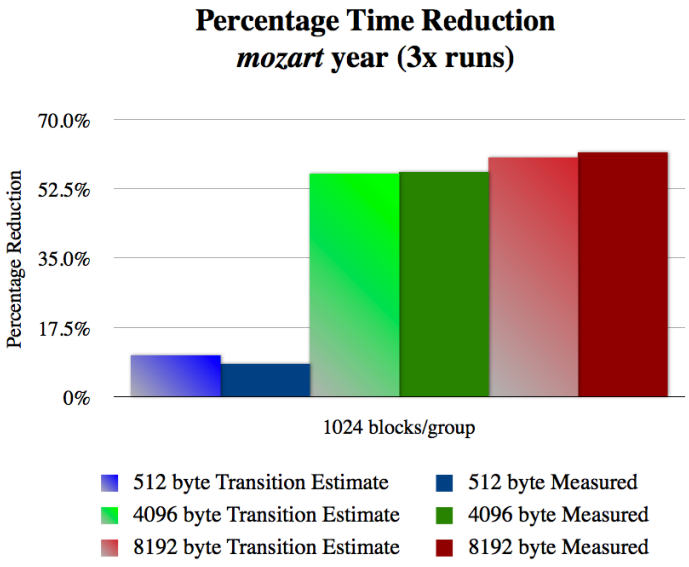


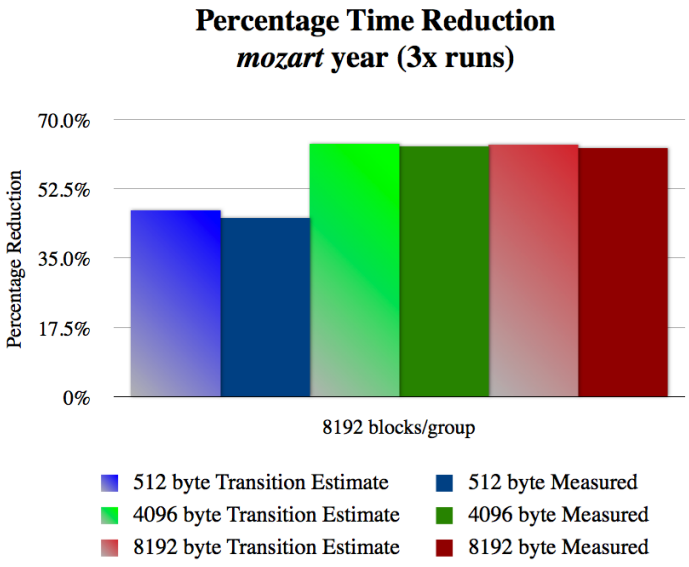
Figure 63: Comparison of *SPORe* estimates and real-world disk measurements.

10.2 RESULTS

Our results are very encouraging in that they indicate a strong correlation between transitions, latency, and energy costs. Figure 63 shows a comparison between our *SPORe* reduction estimates using the number of transitions and the mean of our actual measured reductions for all three test drives. These tests were performed on the *mozart* month length trace with 8 KB blocks and 8192 blocks per groups, and were performed thirty times each, in order to form tighter confidence intervals in Tables 21 and 22. Figure 64 shows the latency results for the *mozart* year length trace, with a group size of 1024 blocks shown in Figure 64(a) and a group size of 8192 blocks in Figure 64(b). Figure 65 shows the comparable energy results. These tests were performed three times each on the Western Digital (WD) test drive.

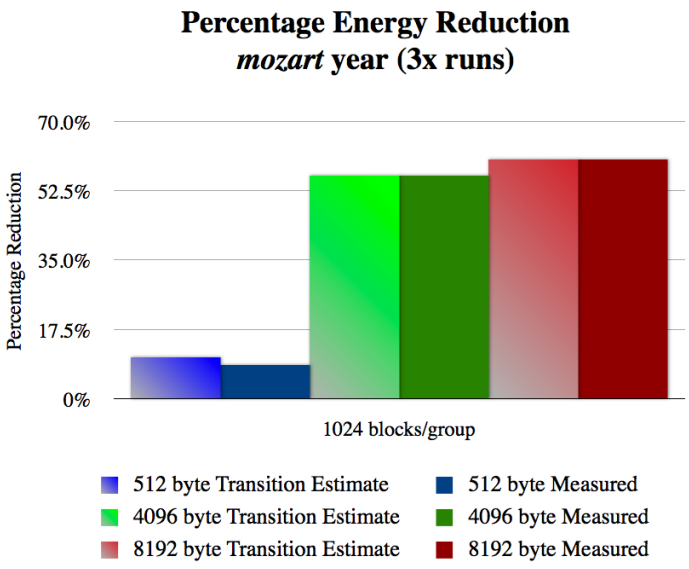


(a) *1024 blocks per group*

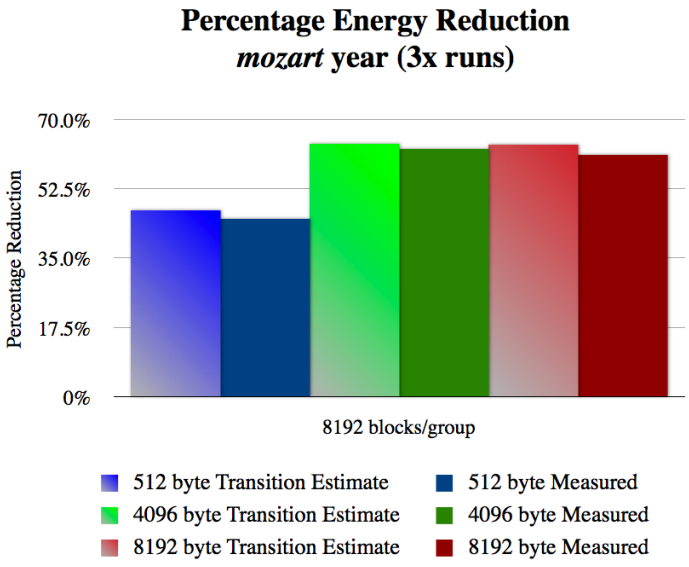


(b) *8192 blocks per group*

Figure 64: Comparison of *SPORe* latency estimates against measured *mozart* year trace.



(a) *1024 blocks per group*



(b) *8192 blocks per group*

Figure 65: Comparison of *SPORe* energy estimates against measured *mozart* year trace.

Our results support our intuition that transition reduction closely approximates latency and energy reductions. In particular, we see that the *mozart* year length traces were very closely approximated, especially for larger block sizes.

Table 21: *mozart* hardware latency reduction results.

Trace	Block Size (bytes)	Group Size (# blocks)	Disk	Runs	Raw $\overline{Time}$ (s)	*SPORe* $\overline{Time}$ (s)	99% CI (% of Raw $\overline{Time}$)
month	8192	8192	HIT	30	389	384	0.96 – 1.55
month	8192	8192	SAM	30	489	482	1.16 – 1.90
month	8192	8192	WD	30	401	394	1.57 – 2.09
month	8192	1024	WD	3	809	621	23.04 – 23.28
month	4096	8192	WD	3	463	398	13.45 – 14.37
month	4096	1024	WD	3	851	668	21.21 – 21.84
month	512	8192	WD	3	868	729	15.46 – 16.69
month	512	1024	WD	3	1180	1090	7.45 – 7.73
year	8192	8192	WD	3	9270	3450	62.32 – 63.28
year	8192	1024	WD	3	14700	5630	61.41 – 62.08
year	4096	8192	WD	3	10500	3850	63.19 – 63.67
year	4096	1024	WD	3	17300	7440	56.65 – 57.27
year	512	8192	WD	3	17800	9760	44.93 – 45.31
year	512	1024	WD	3	24100	22000	8.05 – 8.93

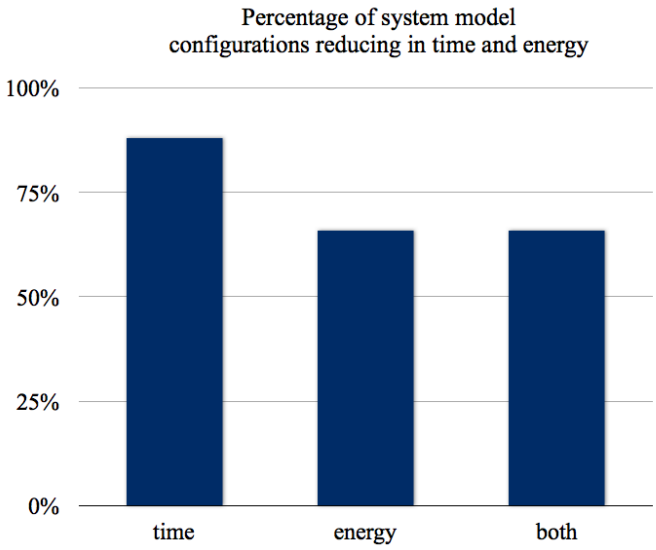
Table 21 summarizes the results for latency measurements, while Table 22 summarizes the energy costs. These tables show the mean raw and reduced times for each trace, as well as the 99% confidence interval of the difference between the means. These confidence intervals are expressed as a percentage of the raw trace measurements. For example, the latency reduction for the *mozart* year trace with 8 KB blocks and 8192 blocks per group exhibited, with 99% confidence, between 62.3% and 63.3% reductions on the Western Digital (WD) test drive, as shown in Table 21. The same test shows, with 99% confidence, a 51.8% to 70.7% reduction in energy, as shown in Table 22. Each of these tested cases, even those with small test set sizes, show statistically significant differences between the raw trace set and the reduced trace set for both energy and latency reductions.

Interestingly, our latency reduction confidence intervals appear significantly tighter than energy reduction confidence intervals. However, for larger test set sizes as well as larger trace sizes, the energy reduction confidence intervals remain reasonably small.

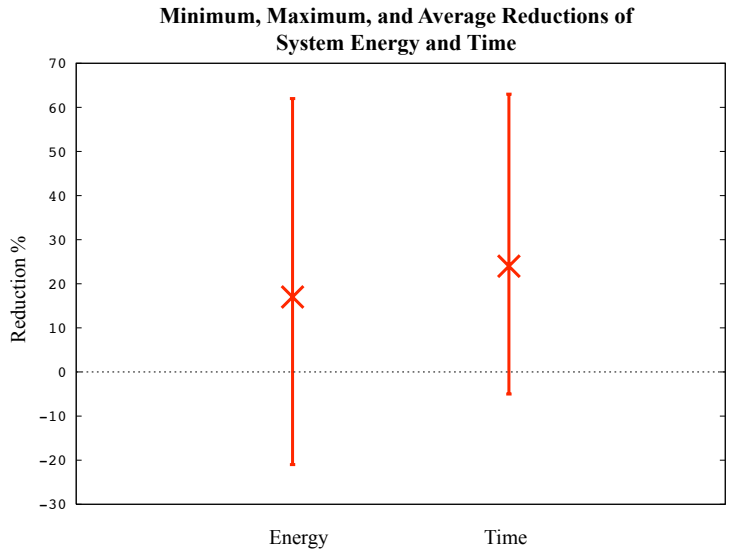
Table 22: *mozart* hardware energy reduction results.

TRACE	BLOCK SIZE (BYTES)	GROUP SIZE (# BLOCKS)	DISK	Runs	RAW $\overline{Energy}$ (J)	*SPORe* $\overline{Energy}$ (J)	99% CI (% OF RAW $\overline{Energy}$)
month	8192	8192	HIT	30	1550	1520	0.85 – 2.42
month	8192	8192	SAM	30	1200	1160	2.41 – 3.93
month	8192	8192	WD	30	982	954	1.59 – 3.99
month	8192	1024	WD	3	2030	1610	12.52 – 29.20
month	4096	8192	WD	3	1140	1010	5.00 – 18.03
month	4096	1024	WD	3	2130	1670	15.66 – 28.15
month	512	8192	WD	3	2160	1840	11.09 – 18.48
month	512	1024	WD	3	2970	2700	1.86 – 16.74
year	8192	8192	WD	3	22500	8730	51.80 – 70.65
year	8192	1024	WD	3	34800	13700	49.25 – 71.78
year	4096	8192	WD	3	25300	9450	58.69 – 66.70
year	4096	1024	WD	3	41300	18000	54.18 – 58.76
year	512	8192	WD	3	43100	23600	41.31 – 48.84
year	512	1024	WD	3	57700	52800	4.76 – 12.29

Using our system model from Equations 21 and 22, we found that *SPORe* to exhibit a reduction of time for 87.5% of configurations (see Figure 66). For energy, we found reductions in 66.1% of configurations. Every configuration that showed energy reductions were found to also reduce time. On average, energy was reduced by 17.4% and time by 23.8%. Table 23 shows these results separated by trace, block size, group size, and drive tested.



(a) *Percentage of system configurations where SPORe shows a benefit*



(b) *Average, max, and min system reductions*

Figure 66: System model results.

Table 23: Reductions of system time and energy for *SPORe* according to our system model, broken down by trace, block size, group size, and test drive.

Trace	Block Size (Bytes)	Group Size (blocks)	Drive Tested	Percentage (for Energy)	Average Energy Reduction	Percentage (for Time)	Average Time Reduction
mozart, month	512	1024	WD	75.0	3.0	100	6.6
mozart, month	512	8192	WD	37.5	-0.6	75.0	8.7
mozart, month	4096	1024	WD	100	18.6	100	21.3
mozart, month	4096	8192	WD	25.0	-4.4	75.0	5.9
mozart, month	8192	1024	WD	100	19.7	100	22.9
mozart, month	8192	8192	HIT	25.0	-5.1	50.0	-0.9
mozart, month	8192	8192	SAM	25.0	-3.7	75.0	0.3
mozart, month	8192	8192	WD	25.0	-4.1	50.0	0.2
mozart, year	512	1024	WD	37.5	0.4	100	5.5
mozart, year	512	8192	WD	75.0	16.0	100	33.7
mozart, year	4096	1024	WD	100	54.6	100	56.5
mozart, year	4096	8192	WD	100	44.4	100	56.0
mozart, year	8192	1024	WD	100	59.7	100	61.4
mozart, year	8192	8192	WD	100	45.2	100	55.8

11.0 CONCLUSIONS AND FUTURE WORK

With the rate of data generation at an all-time high, and increasing at an alarming pace, storage system maintenance is an increasingly crucial task. Persistent predictive replication differs from prefetching in several key ways; most critically, layout maintenance strategies are able to benefit from prior work on behalf of the system, while prefetching, be it accurate or error-prone, is inherently temporary, even evanescent, tied tightly to the data path. Such predictive layout maintenance is an area that remains largely unutilized, with only the simplest of techniques that are prevalent. Yet, in the face of shifting and uncertain workloads, sufficient underlying patterns can be extracted and utilized towards optimized replication, and can be done so efficiently and opportunistically.

Towards understanding how these techniques must change to fit modern systems, we have collected our own long-term, block-level file system traces, complete with cache activity. These traces, used in tandem with classic file system traces, are used throughout our work for validation and testing, representing real-world request streams. To further understanding in storage system layout maintenance, we have defined and optimally solved a general and relaxed grouping problem. Our solution, *DrNO*, is optimal in terms of transitions and distance.

With *Optimal Expansion, Maximized Expectation*, we demonstrate adaptive, efficient, robust strategies for simultaneously reducing energy and latency costs for storage systems. We have developed three such strategies; a generalizable, block-level strategy called *OE ME*, a distance-aware strategy called *OE ED*, and a variable-size strategy called *OE ESS*. Our primary technique, *OE ME*, operates at the block-level, maximizing applicability and generalizable utility with minimal necessary information. This method is shown to greatly outperform existing predictive methods in terms of group transitions as well as distance, tested against real-world workloads. A trend of diminishing returns is clearly observed with respect to increasing group size in existing methods. Our technique exhibits particular resilience to this trend, showing up to 70% reduction in both

estimated latency and energy while forming far fewer groups.

In order to progress from static grouping to dynamic grouping, our methods must address several key issues. We have presented these challenges and our solutions to them, including how to deal with an explosion of predictive metadata. Our metadata storage structure, *SESH*, greatly reduces the size of necessary predictive information without sacrificing information, often reducing the volume of data by several orders of magnitude. This strategy consistently provides first-successor information for less than one half of one percent of the total volume of data stored, with minimal impact on the CPU. A straight-forward augmentation, tracking predecessor as well as successor information, we have shown to exhibit perfect working set and working sequence reconstruction.

We have also presented *LRDU*, or *Least Recently, Distantly Used*; when combined with an *LRU* filter, this strategy is shown to outperform competing strategies for selecting high-frequency block offenders while maximizing the utility for prediction. This allows for a highly-sortable, fast, efficient, adaptive tracking of potential roots upon which to form predictive groups, capable of approximating optimal strategies for some realistic workloads.

In *SPORe*, we have presented the culmination of these predictive efforts into a unified, dynamic, sustainable engine for adaptive layout maintenance and replication. This engine is shown to be robust and resilient to low-confidence workloads as well as incorrect or unknown track size. Further, we demonstrate our recurring claim of persistent prediction utility and demonstrate that our predictive groups act as low-confidence pattern buffers, tackling the areas of a workload that are difficult to characterize. This ability allows for increased utility of sequential, "raw" groups, even when predictive groups have low average accesses per use. For workloads with high predictability, our predictive groups show much higher utility than these sequential groups. Further, we exhibit reduction of necessary group updates, or generated writes to the underlying storage system, by group comparison, and show increased predictive group use through scanning. Additionally, we present a variety of system parameters, including very large group sizes, in order to project in the future of hardware storage devices. We have presented an augmented version of our original *Optimal Expansion, Maximized Expectation* algorithm that forms *supergroups* as a way to combat diminishing returns that we observed in our study of static grouping.

With *SPORe*, we demonstrate reductions for both transitions as well as distance in order to

generalize across systems. These reductions we have verified through hardware validation using accurate energy calculation. These measurements were gathered using a data acquisition unit measuring voltage drops across a low-impedance resistor in series with test drives' mechanical power supplies. We measured voltages at a high sample rate, and have presented straightforward techniques to accurately identify workload boundaries. Energy and latency reductions indicate a close correlation to transition reductions shown in our simulation study on *SPORe*; further, the difference between raw and optimized device-level workloads is shown to be statistically significant, even for small workloads, with high confidence. These hardware validations are also shown to have low variance across multiple test drives.

11.1 FUTURE WORK: AUGMENTING SPORE

Several areas within our work on *SPORe* present themselves for further study. Default track sizes in modern disk drives remain significantly smaller than the largest tested sizes in our work. However, with advances in hardware, this trend may not continue to hold, and larger group sizes we have demonstrated to demand higher CPU utilization. In addition, all predictive information presented is *replicated*; therefore, we foresee possible extensions to write-oriented techniques using low-confidence predictions within our predictive groups.

11.1.1 Increasing Throughput

Future augmentations of *SPORe* include various improvements to increase throughput and further reduce CPU demand. Many of our techniques greatly help reduce this cost, including use of *supergroups*, priority queue "short circuiting", and especially the use of a compact successor history structure in *SESH*. But larger group sizes from possible future devices will require more CPU cycles to handle. We can increase throughput further in a number of ways. First, we have demonstrated that *SPORe* is robust to incorrect track size. This means that we are able to reduce the size *observed* group sizes if we wish, resulting in lower CPU strain. We may also intentionally translate smaller blocks seen by the system into larger blocks to be processed by *SPORe*.

Without intentionally misinforming our data layout management engine, we may also devise a means of decreasing the number of predictive groups formed. While we have demonstrated the effectiveness of avoiding predictive group *operations*, or writes generated to the underlying storage device, we have not avoided forming the group. An early detection mechanism for these uncommitted groups would greatly reduce the computational cost.

Another way of reducing the computational complexity would be to dynamically decide to turn off the regrouping portion of *SPORe*; we have alluded to this augmentation in Section 9.4.1. This could be done for several reasons, including heavy workload, low observed or expected improvement, low confidence in predictions, or little observed change in workload pattern, among others.

Finally, the structure of *SESH* easily lends itself to custom caching techniques. Predictions on a block cause a *Dynamic Bitmap* node to be accessed; with high probability, the same node will be accessed next. We currently implement a new hash table look up on each access; augmenting this structure may involve even a simple one-node sized buffer to be checked *before* any hash table operation.

11.1.2 Location of *SESH*

Our own collected traces are shown to have low confidence patterns. While reductions are presented, we anticipate improvements by moving the data collection up in the storage hierarchy. Further study would require long-term file system traces complete with caching information, similar to our own traces, along with a translation from *cache address* to *device address*. Using this translation would enable better interaction with cache and memory management. Specifically, we anticipate much higher confidence if data is captured pre-cache. Our structure is expected to maintain high sequentiality necessary for size reduction, the tree structure used has been shown to be resistant to system noise [8].

Additionally, we may get some benefit from using knowledge or simulation of cache or memory management. Predictive groups become useful if some request is *not* satisfied by cache or memory. We may be able to exploit this by specifically *not* including items that are likely to exist in the cache, given that some request (most likely, the root) has been generated. Such answers

may be refined by looking backward from this request, *e.g.* using our augmented *SESH* structure that tracks predecessor information. We envision a backwards prediction, with any predecessor occurring with confidence lower than some threshold being added to a "black list" of IDs that are not to be added to the predictive group.

11.1.3 Extensions to Write Strategies

Read requests occur when data has already been committed; the underlying system, presumably, has the information, while a write request is acting upon information yet to be witnessed. This simple but fundamental difference is the driving impetus behind our focus on read requests. Future augmentations of our predictive engine may take greater care of write requests in a number of ways. For example, upon a block's write request, how might we update copies held in predictive groups? A simple mechanism to solve this coherence problem is to simply free all copies from predictive groups; the primary benefit is that this operation need not access the track within which the copies exist.

This "free block" strategy, coupled with *replicated* predictive groups, presents an additional possible augmentation. Using freed blocks, possibly along with blocks predicted with low confidence, we may provide an area for pending writes, allowing for reduced write-triggered seeks. We envision the free and low-confidence areas of predictive groups serving as write offloading locations, as used in [92], but at the device level rather than the data center level.

11.2 FUTRE WORK: TRACE GATHERING AND USAGE

We have previously discussed the merits of trace-driven simulation in Chapter 4. With our observations on reduced confidence patterns existing post cache in our custom collected traces, we foresee a need for similar traces in the future. Accurate, detailed, long-term traces seem to be uncommon; indeed, many research papers use short benchmarks lasting 8 hours or less [118]. Additionally, many benchmarks leave out cache information. We predict that traces similar to our own, with cache translations included, will be in high demand if storage systems and devices are to keep pace

with demand trends.

With this in mind, trace gathering need not be a complicated process. We propose the use of simple tools, such as the `fs_usage` command, used in gathering our own traces, to encourage trace collection from various public sources. Additionally, we envision a possible system using our own augmented *SESH* structure to recreate working sequences, wherever they are gathered from, for large or short trace generation. Upon a simple trace request, the working sequence could be generated and saved within moments, rather than requiring specific gathering scripts to be initiated. While recreating a trace in this way has no guarantee of trace length, it often can generate very large traces, and would do so within moments, rather than hours or days. Even in the case of intentional trace collection, a working sequence reconstruction might be used for additional sequencing to be added at the beginning of the workload history.

BIBLIOGRAPHY

[1] Hitachi travelstar 5k250 disk drive technical specifications. Retrieved October 6, 2010. http://www.hitachigst.com/tech/techlib.nsf/techdocs/E1F385C6ABA70364862572F00067C59B/$file/5K250_DS.pdf.

[2] U.S. Environmental Protection Agency. Greenhouse gas emissions from the U.S. transportation sector (1990–2003). Technical Report EPA 420-R-06-003, Office of Transportation and Air Quality (6401A), 2006.

[3] U.S. Environmental Protection Agency. Report to Congress on server and datacenter energy efficiency. Public Law 109–431, 2007.

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

[5] Ahmed Amer. *Predictive Data Grouping Using Successor Prediction*. PhD thesis, University of California, Santa Cruz, Septemeber 2002. Long, Darrell D.

[6] Ahmed Amer and Darrell D. E. Long. Noah: Low-cost file access prediction through pairs. In *Proceedings of 20th International Performance, Computing, and Communications Conference (IPCCC 2001)*, pages 27–33, Phoenix, Arizona, April 2001. IEEE Computer Society.

[7] Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. Group-based management of distributed file caches. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, 2002. IEEE Computer Society.

[8] Ahmed Amer, Darrell D. E. Long, Jehan-François Pâris, and Randal C. Burns. File access prediction with adjustable accuracy. In *Proceedings of 21st International Performance, Computing, and Communications Conference (IPCCC 2002)*, pages 131–140, Phoenix, Arizona, 2002. IEEE Computer Society.

[9] Ahmed Amer, Alison Luo, Newton Der, Darrell D. E. Long, and Alexander Pang. Visualizing cache effects on i/o workload predictability. In *Proceedings of the International Performance Conference on Computers and Communication (IPCCC '03)*, pages 417 – 424, Phoenix, Arizona, April 2003. IEEE Computer Society.

[10] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive caching using multiple experts. In *Distributed Data & Structures 4, Records of the 4th International Meeting (WDAS 2002)*, volume 14, pages 143–158, Paris, France, March 2002. Carleton Scientific.

[11] Jinsuk Baek, Paul S. Fisher, and Min Gyung Kwak. Fi-based file access predictor. In *ACM-SE 47: Proceedings of the 47th Annual Southeast Regional Conference*, pages 1–4, Clemson, South Carolina, March 2009. ACM.

[12] Sung Hoon Baek and Kyu Ho Park. Prefetching with adaptive cache culling for striped disk arrays. In *ATC '08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 363–376, Boston, Massachusetts, 2008. USENIX Association.

[13] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[14] Cullen Bash and George Forman. Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[15] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. Borg: block-reorganization for self-optimizing storage systems. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 183–196, Berkeley, CA, USA, 2009. USENIX Association.

[16] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case for power management in web servers. *Power aware computing*, pages 261–289, 2002.

[17] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and Contributors. The disksim simulation environment. Retrieved on August 19, 2009 http://www.pdl.cmu.edu/DiskSim/, 2008.

[18] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and Contributors. The disksim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon, 2008.

[19] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high i/o data rates. *Computing Systems*, 4(4):405–436, 1991.

[20] Scott D. Carson and Paul F. Reynolds Jr. Adaptive disk reorganization. Technical Report UMIACS–TR–89–4 and CS–TR–2178, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, and Department of Computer Science, University of Virginia, January 1989.

[21] Matthew Craven and Ahmed Amer. Predictive reduction of power and latency (PuRPLe). In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 237–244, Monterey, California, April 2005. IEEE Computer Society.

[22] Igor Crk and Chris Gniady. Context-aware mechanisms for reducing interactive delays of energy management in disks. In *ATC '08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 71–84, Boston, Massachusetts, 2008. USENIX Association.

[23] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. *SIGMOD Record*, 22(2):257–266, 1993.

[24] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, Redmond, WA, USA, 1994.

[25] Yuhui Deng. Exploiting the performance gains of modern disk drives by enhancing data locality. *Information Sciences*, 179(14):2494–2511, 2009.

[26] Peter J. Denning. Effects of scheduling on file memory operations. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 9–21, Atlantic City, New Jersey, April 1967. ACM.

[27] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: exploiting disk layout and access history to enhance i/o prefetch. In *ATC '07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Santa Clara, California, 2007. USENIX Association.

[28] Bruno Diniz, Dorgival Guedes, Wagner Meira, Jr., and Ricardo Bianchini. Limiting the power consumption of main memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 290–301, San Diego, California, June 2007. ACM.

[29] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, volume 8, pages 121–137. USENIX Association, April 1995.

[30] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *WTEC '94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 293–306, San Francisco, California, January 1994. USENIX Association.

[31] Dan Duchamp. Prefetching hyperlinks. In *USITS'99: Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, 1999. USENIX Association.

[32] David Essary and Ahmed Amer. Dr NO clustering: Mozart trace analysis. Technical Report TR-04-112, Department of Computer Science, University of Pittsburgh, August 2004.

[33] David Essary and Ahmed Amer. Project (Dr.) NO proof of optimality. Technical Report TR-04-113, Department of Computer Science, University of Pittsburgh, March 2004.

[34] David Essary and Ahmed Amer. Predictive data grouping: Defining the bounds of energy and latency reduction through predictive data grouping and replication. *Transactions on Storage*, 4(1):1–23, May 2008.

[35] David Essary and Ahmed Amer. Avoiding state-space explosion of predictive metadata with SESH. In *Proceedings of the IEEE International Performance, Computing and Communications Conference (IPCCC)*, Phoenix, Arizona, December 2009. IEEE Computer Society.

[36] David Essary and Ahmed Amer. Space-efficient predictive block management. In *Proceedings of the International Workshop on Software Support for Portable Storage (IWSSPS'09)*, Grenoble, France, October 2009. ACM.

[37] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *SIGOPS Operating Systems Review*, 33(5):48–63, December 1999.

[38] Community Office for Resource Efficiency. Carbon dioxide information analysis center frequently asked questions. Retrieved on August 19, 2010 from http://cdiac.ornl.gov/pns/faq.html.

[39] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–17, Anaheim, California, January 1997. USENIX Association.

[40] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. The disksim simulation environment version 1.0 reference manual. Technical Report CSE-TR-358-98, University of Michigan, February, 1998.

[41] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. The disksim simulation environment version 2.0 reference manual. Technical report, Carnegie Mellon, 1999.

[42] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe. IDC white paper, http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf, 2008.

[43] Lieutenant Colonel Tim Gibson and Ethan L. Miller. An improved long-term file usage prediction algorithm. In *CMG '99: 25th International Computer Measurement Group Conference*, pages 639–648, Reno, Nevada, December 1999. Computer Measurement Group.

[44] Binny S. Gill and Dharmendra S. Modha. Sarc: sequential prefetching in adaptive replacement cache. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Anaheim, California, April 2005. USENIX Association.

[45] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, New Orleans, Louisiana, January 1995. USENIX Association.

[46] Richard Golding, Peter Bosch, and John Wilkes. Idleness is not sloth. Technical Report HPL-96-140, Hewlett-Packard Laboratories, Palo Alto, California, 1996.

[47] Jim Gray and Prashant Shenoy. Rules of thumb in data engineering. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, Washington, DC, USA, 2000. IEEE Computer Society.

[48] Paul M. Greenawalt. Modeling power management for hard disks. In *MASCOTS '94: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 62–66, Durham, North Carolina, February 1994. IEEE Computer Society.

[49] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USTC '94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 197–207, Boston, Massachusetts, June 1994. USENIX Association.

[50] Sudhanva Gurumurthi, Youngjae Kim, and Anand Sivasubramaniam. Using steam for thermal simulation of storage systems. *IEEE Micro*, 26(4):43–51, July 2006.

[51] Sudhanva Gurumurthi, Anand Sivasubramaniam, and Vivek K. Natarajan. Disk drive roadmap from the thermal perspective: A case for dynamic thermal management. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, volume 33, pages 38–49, Madison, Wisconsin, May 2005. ACM.

[52] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities*. Cambridge University Press, 2 edition, 1952, reprinted 1983.

[53] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *ACM/Baltzer Mobile Networks and Applications* (*MONET*), 5(4):285–297, December 2000.

[54] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *MobiCom '96: Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 130–142, Rye, New York, November 1996. ACM.

[55] Mark Herbster and Manfred K. Warmuth. Tracking the best expert. In *Twelfth International Conference on Machine Learning*, pages 286–294, Tahoe City, California, 1995. Morgan Kaufmann.

[56] Hai Huang, Wanda Hung, and Kang G. Shin. Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption. *SIGOPS Operating Systems Review*, 39(5):263–276, December 2005.

[57] Lan Huang and Tzi cker Chiueh. Implementation of a rotation latency sensitive disk scheduler. Technical Report CS-TR-283-90, Computer Science Department, State University of New York at Stony Brook, 2000.

[58] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 117–130, Banff, Alberta, Canada, October 2001. ACM.

[59] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Concurrent Computing Department, Hewlett-Packard Company, Palo Alto, California, 1991.

[60] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST '05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 8–8, San Francisco, California, 2005. USENIX Association.

[61] Nikolai Joukov and Josef Sipek. Greenfs: making enterprise computers greener by protecting them better. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 69–80, Glasgow, Scotland UK, April 2008. ACM.

[62] Mahmut Kandemir, Seung Woo Son, and Mustafa Karakoy. Improving disk reuse for reducing power consumption. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, pages 129–134, Portland, Oregon, 2007. ACM.

[63] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[64] Youngjae Kim and Sudhanva Gurumurthi An. Understanding the performance-temperature interactions in disk i/o of server workloads. In *HPCA-12: 12th International Symposium on High-Performance Computer Architecture*, pages 179–189, Austin, Texas, February 2006. IEEE Computer Society.

[65] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *13th ACM Symposium on Operating Systems Principles (SOSP)*, 25(5):213–225, October 1991.

[66] Rachita Kothiyal, Vasily Tarasov, Priya Sehgal, and Erez Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, New York, NY, USA, 2009. ACM.

[67] P. Krishnan. *Online prediction algorithms for databases and operating systems*. PhD thesis, Department of Computer Science, Brown University, Providence, RI 02912, August 1995.

[68] P. Krishnan, Philip Long, and Jeffrey Scott Vitter. Learning to make rent-to-buy decisions with systems applications. In *Proceedings of the Twelfth International Conference on Machine Learning (ML95)*, pages 322–330, Tahoe City, California, July 1995. Morgan Kaufmann.

[69] P. Krishnan, Philip Long, and Jeffrey Scott Vitter. Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments. *Algorithmica*, 23(1):31–56, July 1999.

[70] Thomas M. Kroeger and Darrell D. E. Long. Predicting future file-system actions from prior events. In *Proceedings of the 1996 Usenix Winter Technical Conference*, pages 319–328, San Diego, California, January 1996. USENIX Association.

[71] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, pages 14–9, Rio Rico, Arizona, March 1999. IEEE Computer Society.

[72] Thomas M. Kroeger and Darrell D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *USENIX Annual Technical Conference*, pages 105–118, Boston, Massachusetts, June 2001. USENIX Association.

[73] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997. USENIX Association.

[74] James A. Larkby-Lahet, Ganesh Santhanakrishnan, Ahmed Amer, and Panos K. Chrysanthis. Step: Self-tuning energy-safe predictors. In *MDM '05: Proceedings of the 6th international conference on Mobile data management*, pages 125–133, Ayia Napa, Cyprus, May 2005. ACM.

[75] Donghee Lee, Jongmoo Choi, Jong hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU (least recently/frequently used) replacement policy: A spectrum of block replacement policies. In *IEEE Transactions on Computers*. IEEE Computer Society, 1996.

[76] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 12, December 2003.

[77] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, Anaheim, California, January 1997. USENIX Association.

[78] A W Leung, M Shao, T Bisson, S Pasupathy, and E L Miller. High-performance metadata indexing and search in petascale data storage systems. *Journal of Physics: Conference Series*, 125(1), 2008.

[79] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. Spyglass: fast, scalable metadata search for large-scale storage systems. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 153–166, Berkeley, CA, USA, 2009. USENIX Association.

[80] Dong Li. *High performance energy efficient file storage system*. PhD thesis, University of Nebraska at Lincoln, Lincoln, NB, USA, 2006. Adviser-Wang, Jun.

[81] Dong Li and Jun Wang. A performance-oriented energy efficient file system. *SNAPI '04: Proceedings of the international workshop on Storage network architecture and parallel I/Os*, pages 58–65, 2004.

[82] Huajing Li, Wang-Chien Lee, Anand Sivasubramaniam, and C. Lee Giles. A hybrid cache and prefetch mechanism for scientific literature search engines. In *ICWE'07: Proceedings of the 7th international conference on Web engineering*, pages 121–136, Berlin, Heidelberg, 2007. Springer-Verlag.

[83] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. TaP: table-based prefetching for storage caches. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, San Jose, California, 2008. USENIX Association.

[84] Jacob R. Lorch and Alan Jay Smith. Software strategies for portable computer energy management. *IEEE Personal Communications*, 5(3):60–73, June 1998.

[85] Peter Lyman and Hal R. Varian. How much information. Retrieved on August 19, 2010 from `http://www.sims.berkeley.edu/how-much-info-2003`, 2003.

[86] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[87] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[88] Nimrod Megiddo and Dharmendra S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[89] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 205–216, New York, NY, USA, 2009. ACM.

[90] Alan G. Merten. *Some quantitative techniques for file organization*. PhD thesis, University of Wisconsin-Madison, Madison, Wisconsin, June 1970.

[91] Lily Mummert and Mahadev Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software - Practice and Experience (SPE)*, 26(6):705–736, June 1996.

[92] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Transactions on Storage*, 4(3):1–23, 2008.

[93] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[94] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficiency through burstiness. *IEEE Workshop on Mobile Computing Systems and Applications*, 2003.

[95] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficient prefetching and caching. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, Boston, Massachusetts, June 2004. USENIX Association.

[96] Jehan-François Pâris, Ahmed Amer, and Darrell D. E. Long. A stochastic approach to file access prediction. In *SNAPI '03: Proceedings of the international workshop on Storage network architecture and parallel I/Os*, pages 36–40, New Orleans, Louisiana, 2003. ACM.

[97] Caleb Phillips, Suresh Singh, Douglas Sicker, and Dirk Grunwald. Applying models of user activity for dynamic power management in wireless devices. In *MobileHCI '08: Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 315–318, New York, NY, USA, 2008. ACM.

[98] Alexander P. Pons. Web-application centric object prefetching. *Journal of Systems and Software*, 67(3):193–200, 2003.

[99] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (das). In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Washington, DC, December 2003. IEEE Computer Society.

[100] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, San Diego, California, June 2000. USENIX Association.

[101] Chris Ruemmler and John Wilkes. Disk shuffling. Technical Report HPL-CSP-91-30, Software Systems Laboratory, Hewlett-Packard Company, October 1991.

[102] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *USENIX Winter Technical Conference*, pages 405–420, San Diego, California, January 1993. USENIX Association.

[103] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition edition, 2003.

[104] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 259–274, Monterey, California, January 2002. USENIX Association.

[105] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: A disk array volume manager for orchestrated use of disks. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 159–172, San Francisco, California, 2004. USENIX Association.

[106] Philip H. Seaman, Robert A. Lind, and Troy L. Wilson. On teleprocessing system design part iv: An analysis of auxiliary storage activity. *IBM Systems Journal*, 5(3):158–170, 1966.

[107] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1992.

[108] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, California, 1990. USENIX Association.

[109] S. W. Sherman and J. C. Browne. Trace driven modeling: Review and overview. In *ANSS '73: Proceedings of the 1st symposium on Simulation of computer systems*, pages 200–207, Gaithersburg, Maryland, 1973. IEEE Computer Society.

[110] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher Stein. Storage management for web proxies. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 203–216, Boston, Massachusetts, June 2001. USENIX Association.

[111] Elizabeth Shriver, Christopher Small, and Keith Smith. Why does file system prefetching work? In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 71–83, Monterey, California, June 1999. USENIX Association.

[112] Keth A. Smith and Margo Seltzer. A comparison of FFS disk allocation policies. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 15–25, San Diego, California, January 1996. USENIX Association.

[113] Carl Staelin and Hector Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, 1990. Revised June 1990.

[114] Carl Staelin and Hector Garcia-Molina. File system design using large memories. In *JCIT: Proceedings of the fifth Jerusalem conference on Information technology*, pages 11–21, Jerusalem, Israel, October 1990. IEEE Computer Society.

[115] Carl Staelin and Hector Garcia-Molina. Smart filesystems. In *Proceedings of the Winter 1991 USENIX conference*, pages 45–52, Dallas, Texas, January 1991. USENIX Association.

[116] David Capppers Steere. *Using Dynamic Sets to Reduce the Aggregate Latency of Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1997.

[117] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 2001.

[118] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *Transactions on Storage*, 4(2):1–56, May 2008.

[119] Stephen Tweedie. Journaling the linux ext2fs filesystem. *Proceedings of the 4th Annual LinuxExpo*, 1998.

[120] Dan Tynan. Prepare for data tsunami, warns Google CEO. *PCWorld*, 2010.

[121] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996.

[122] M. Warmuth and N. Littlestone. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, February 1994.

[123] Andreas Weissel, Björn Beutel, and Frank Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 117–129, Boston, Massachusetts, December 2002. ACM.

[124] Gary A. S. Whittle, Jehan-François Pâris, Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. Using multiple predictors to improve the accuracy of file access predictions. *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 230–240, April 2003.

[125] John Wilkes. Predictive power conservation. Technical Report HPL-CSP-92-5, Concurrent Systems Project, Hewlett-Packard Laboratories, Palo Alto, California, February 1992.

[126] C. K. Wong. *Algorithmic Studies in Mass Storage Systems*. W. H. Freeman & Co., New York, NY, USA, 1983.

[127] Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt. Performing file prediction with a program-based successor model. In *MASCOTS '01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 193–202, Cincinnati, Ohio, August 2001. IEEE Computer Society.

[128] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Wang. Modeling hard-disk power consumption. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 217–230, San Francisco, California, March 2003. USENIX Association.

[129] Qingbo Zhu and Yuanyuan Zhou. Power-aware storage cache management. *IEEE Transactions on Computers*, 54(5):587–602, May 2005.

[130] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. In *IEEE Transactions on Information Theory*, volume IT-24, pages 530–536. IEEE Computer Society, Septemeber 1978.