A Hybrid Filesystem for Hard Disk Drives in Tandem with Flash Memory

October 23, 2011

Abstract

The traditional hard disk drive (HDD) is often a bottleneck in the overall performance of modern computer systems. With the development of solid state drives (SSD) based on flash memory, new possibilities are available to improve secondary storage performance. In this work, we propose a new hybrid SSD-HDD storage system and a selection of algorithms designed to assign pages across an HDD and an SSD to optimise I/O performance. The hybrid system combines the advantages of the SSD's fast random seek speed with the sequential access speed and large storage capacity of the HDD to produce significantly improved performance in a variety of situations. We further improve performance by allowing concurrent access across the two types of storage devices. We show the drive assignment problem is NP-complete and accordingly propose effective heuristic solutions. Extensive experiments using both synthetic and real data sets show our system with a small SSD can outperform a striped dual HDD and remain competitive with a dual SSD.

Keywords: File system, Flash memory, Solid state drives, Storage.

1 Introduction

Desktop computers and server systems today often suffer from a well identified problem, the input/output (I/O) bottleneck. Hard Disk Drives (HDD) are the current dominant secondary storage device for these systems because they have a low cost/capacity ratio, offer high capacity storage and are non-volatile. However, recently flash memory has been introduced as secondary storage in the form of Solid State Drives (SSD) designed to emulate HDD. In this paper, we will use the terms flash memory and SSD interchangeably. SSDs are an emerging technology and while they still have a high cost/capacity ratio, smaller overall capacities than HDD and limited erase-based lifetimes, they provide much faster random access, lower power usage and are much more shock resistant than HDD.

This research aims at using SSDs and HDDs together in an attempt to obtain the best performance from both devices while still keeping costs low and storage capacities large. We assume a system which utilises a small limited capacity SSD and a relatively large HDD. To maximise performance, we assume the two storage devices can be accessed concurrently to fulfil a request. We propose a drive assignment algorithm which determines which device to place data on in order to take advantage of their desirable characteristics while trying to overcome some of their undesirable characteristics. This decision is deceptively complex, since the files can be accessed both sequentially, randomly and a mixture of the two. SSDs perform far better than HDDs for random access but this difference is reduced for long sequential accesses. High end, very expensive SSDs are now capable of outperforming an HDD for sequential access as well. Our algorithm needs to balance these concerns when determining the assignment of pages to drives. We do this offline, using a cost model which estimates the cost/benefit of the pages being stored on one device over the other. Since pages can be related (accessed sequentially in the same request), we show that the data assignment problem is NP-complete and use heuristics to achieve a near optimal solution. Our initial drive assignment first roughly segments the data into related segments and then partitions the segments into the two drives. This is then improved via a refinement algorithm that uses an exact cost formula to produce the initial drive assignment. We propose three segmentation algorithms to assign pages between the two drives. Extensive experimental studies revealed the relative merits of the different algorithms and compare them with simple competitors.

There has been considerable existing work on the design of database and file systems that use flash memory in combination with HDD to reduce I/O costs [17, 38, 12, 24, 28]. These works either use flash memory as a small buffer [24, 28] or proposes simple ad-hoc policies for determining the assignment of individual files to the different devices [38, 12] or assumes the flash memory device can fit all the data [17]. In contrast to the above work, this paper is the first to propose a file system that contains *all* of the following features:

1. Treats flash memory at the same level of the storage hierarchy as HDDs.
2. Proposes theoretically sound workload-driven algorithms to assign data across flash memory and the HDD at the *page* instead of *file* grain.
3. Performs intra-file concurrency. Intra-file concurrency refers to reading or writing different parts of a file concurrently.
4. Allows for the capacity of the flash memory to be smaller than the total data size. This allows us to use a smaller amount of flash memory compared to many hybrid systems. This in turn keeps additional costs low to convert from an existing storage system.

The first feature allows better utilisation of larger flash memory devices such as the SSD. Using large SSDs as a cache for HDDs is not ideal since it would mean that all data kept on the SSD will need to be duplicated on the HDD. Updates to the SSD will also eventually need to be applied to the HDD to keep the cache in sync with the HDD.

The second feature allows our system to systematically minimise the total I/O cost of a given typical workload across the SSD and HDD at the finer page grain. The existing algorithms [38, 12] which use ad-hoc policies to assign data at the coarser file grain across the two types of drives can not make fine grained assignment decisions and do not factor in the precise cost of assignment decisions. An example of where our system would offer superior performance is partitioning of a B+-tree across SSD and HDD. These indexes are typically stored in one file which means the systems proposed by [38, 12] will not be able to partition the index across the two drive types. In contrast, our system can make optimal use of the limited space on the SSD by placing the pages of the index that are used more often onto the SSD and leaving the rest on the HDD.

The third feature allows our system to satisfy a request to a single file which is partitioned across the two drives concurrently. This can potentially half the time taken to satisfy a request compared to file systems that do not support intra-file concurrency. To our knowledge, none of the existing work on hybrid flash memory/HDD file systems support intra-file concurrency across storage devices while limiting the flash memory capacity.

The fourth feature allows our system to be useful for varying SSD sizes. In contrast, the work by Koltsidas et al. [17] (the only work to assign data at the page grain) assumes the SSD can fit all the data. This makes their work inappropriate for realistic situations where a small SSD, which can not fit all the data, is coupled with a large HDD.

Other research on file systems optimisation focuses on alternative file systems [26, 25, 32, 39] or improving data placement within an existing file system [2, 14, 27]. In both cases, the systems assume a single type of secondary storage for data. In contrast, our work focuses on utilising two different types of secondary storage to improve I/O performance.

We use both real traces and synthetic workloads to test our system in a simulated environment. The synthetic workloads allowed us to precisely vary the characteristics of workloads to show how our algorithms perform under various workload characteristics. We compared our algorithms against dual non-hybrid systems using two HDDs or two SSDs. In the dual non-hybrid systems, we striped the data across the two devices at the page grain. The results using both real and synthetic workloads show our system, when given just a small SSD, can outperform dual HDD in all situations tested and come close to a dual SSD system in most situations.

The following is a list of our key contributions:

- Formally defines the drive assignment problem and proves it is NP-complete via a mapping to the Multiple-Choice Knapsack problem.
- Proposes three heuristic solutions for the drive assignment problem and compares them to simple alternative algorithms.

- Conducts a thorough experimental evaluation and analysis of the proposed system against likely competitors of striped dual HDD and dual SSD. The results show that our system, with just a small limited-sized SSD, depending on the situation can outperform and achieve similar performance to likely competitors in a variety of situations.

2 Flash Storage

Flash memory has two primary types, NAND and NOR, with NAND being cheaper, becoming more prevalent and having a greater storage/size ratio and NOR flash the older of the two and being primarily used for eXecute-In-Place (XIP) applications. In NOR flash, the memory is directly addressable by the processor and generally each byte can be read or written individually. On the other hand, NAND flash memory is divided into blocks which are further divided into pages and a flash controller is required to access the data. Blocks in NAND flash memory typically contain about 64 pages and pages are normally 2KB or 4KB in size. Due to the extra interfacing required by NAND flash, its seek times suffer in comparison to NOR flash; on the other hand NAND flash performs much better during larger sequential write operations. In both types of flash memory, bits can only be cleared via an erase process done at the block level. Erase procedures for both types are energy and time consuming. Due to the benefits of using NAND flash for secondary storage, we will focus on NAND flash memory. Therefore, flash memory from here on will refer to NAND flash memory.

Since flash memory can only clear data at a block level, it performs out-of-place updates during which the new data is written elsewhere and the old data is flagged invalid. This style of updating data requires a type of garbage collection or cleaning during which invalidated data is reclaimed. Many different garbage collection algorithms exist [7, 20].

Wear leveling is a procedure to balance the usage of all erase blocks of the flash memory evenly. Since flash memory has a limited lifetime, usually around 1,000,000 erases, it is important to ensure that any file system using flash memory applies wear leveling techniques. Wear leveling techniques use heuristics, algorithms or a history of usage to ensure certain erase units are not worn out well before the rest of the storage.

The flash translation layer (FTL) is a controller placed between the file system and a flash drive. It hides the extra requirements of the flash memory from the host by making the flash chip look like a standard disk (an indexed array of relatively small fixed-sized blocks that can be rewritten arbitrarily). Underneath, it provides a mapping of logical pages/blocks to physical page/blocks. This is done to allow wear leveling algorithms to spread the usage evenly across the blocks and allow data to appear to the file system to be in the same place. There has been much research in the area of designing FTLs that provide efficient logical to physical mapping at different grains: page [15], page/block hybrid [19, 21, 18, 23] and cluster of contiguous pages [6].

3 Comparison of Hard Disk Drives and Flash Memory

HDD and flash memory have quite different characteristics and must be managed differently for optimal performance. An overview of some basic differences between flash memory and HDD is given in Table 1. We use the specifications for a 1TB Seagate Barracuda 7200.12 [33], **a 120 GB OCZ Vertex 2[42] and the much faster and more expensive 240 GB Vertex 3 [43] SSD.** As can be seen, the average latency of a flash memory SSD is much lower than that of the HDD. The relative transfer rate difference between the HDD and two SSDs are much smaller than the latency difference. This means for sequential access the two types of devices have much more similar performance.

When a block on flash memory reaches its erase limit it is marked as failed and will be unusable so the usable size of the disk will be decreased. HDDs, on the other hand, have no set lifetime restriction. However, HDDs have the risk of head or platter damage during a head crash. This means that HDDs are not very durable under shock, especially if they are performing an operation at the time.

Flash memory is much faster at random access than HDDs because it accesses data electronically, while HDDs suffer from latency and long seek times due to the mechanical process involved. However, during a sequential access, the difference between the two technologies is reduced, with HDDs utilising the fast transfer rates as can be seen in Table 1. Flash memory suffer from long erase times when all clean blocks are

Table 1: Comparison of Barracuda 7200.12, OCZ Vertex 2 and UltraDrive GX SSD

Attribute	**Barracuda HDD**	**OCZ Vertex 2**	**OCZ Vertex 3**
Capacity(GB)	1000	**120**	**240**
Price(US$)	89[13]	**200** [40]	**565.00** [41]
Latency(ms)	4.17	**0.1**	**0.1**
Transfer Rate (MB/s)	125	**285/275 (read/write)**	**550/520 (read/write)**
Price per GB (US$/GB)	**0.089**	**1.67**	**2.35**

exhausted and all dirty blocks are near capacity with valid data. Waiting for this operation to complete can delay a pending read/write request.

Energy consumption for flash memory is much less than that for HDDs. It not only consumes less energy for each of its major operations: read, write and erase, but also during idle time. This is because flash drives are almost completely idle when not in use, while HDDs either keep spinning or, in the event of power saving options, slow down the spin when idle but then suffer a warm-up time when the next request comes through for the disk to spin back to speed.

4 Related Work

In this section we will discuss different works in the fields of hybrid systems and file systems for individual devices. There has been a great deal of research already in using additional non-HDD storage device (which includes flash memory) to improve I/O performance (hybrid systems). We will discuss these works in two sections. The first is where the additional non-HDD storage device is used at the same level of the storage hierarchy. The second is where the additional non-HDD storage device is used as an additional cache or buffer between the existing HDD and the traditional RAM buffer. We will also discuss works which look at log-based file systems to improve I/O performance and flash specific file systems.

4.1 Non-HDD storage device at the same level as the HDD

The research covered in this section focuses on the use of additional non-HDD storage devices at the same level as the HDD. These works use a different approach than works which focus on using non-HDD storage devices such as flash memory as small caches above an HDD. Primarily these systems either require unbounded amounts of fast non-HDD storage space (which can be expensive) or uses ad-hoc assignment policies.

The work by Koltsidas and Viglas [17] presents algorithms for database systems that use the SSD at the same level of the storage hierarchy as the HDD. As the paper focuses on using the hybrid system specifically for databases, they assume a large proportion (80%) of their workload as random access and do not consider the cost difference between sequential and random access on the HDD. The system they propose is designed to work online and migrate high read heat pages into the flash while high write heat pages are kept on the HDD. As they work online, they include a buffer replacement policy which takes advantage of the properties of the two devices. The page migration between the devices is done as the pages are evicted from the buffer. Three algorithms are proposed to implement the page migration onto the SSD, as well as a page replacement policy for the memory buffer.

The researchers test their algorithms on their hybrid system versus both a single HDD and a single SSD setup. They note that their algorithms always perform better than the single HDD or SSD as long as the pages have been placed appropriately. This is because they assume that they have enough capacity to store as many pages as they need on the SSD. This research differs specifically from ours since it assumes an SSD that fits the entire database; it does not differentiate between random and sequential access and it does not allow for concurrency between the SSD and HDD.

Payer et. al. [31] proposes a hybrid SSD and HDD system that offers two distinct solutions dependent on the type of SSD drive used. They classify SSD drives into two categories: low performance (low latency, low throughput) and high performance (low latency, high throughput). For low performing SSDs Payer et. al. propose two heuristics: 1) place executables and program libraries onto the SSD; and 2) place random

accessed files onto the SSD. In both cases the remaining files are placed on the HDD. For high performing SSDs Payer et. al. proposes moving the most frequently access files onto the HDD since there is little throughput difference between the SSD and HDD. This work contrasts from ours in that it uses ad-hoc policies for placement of files onto the SSD and HDD and also does not take any advantage of intra-file access concurrency because it places entire files on the SSD or HDD. In contrast our system uses a cost model to place individual pages of files on storage device that results in the greatest benefit in terms of access costs.

Soundararajan et. al. [34] proposes a hybrid SSD and HDD system that accumulates a log of changes on the HDD before writing them in bulk onto the SSD at a later time. The recently updated pages are also cached in RAM which allows faster access to update pages. Experiments show the proposed system cuts down the write costs to the SSD by approximately 49% and thereby extends the lifetime of the SSD by close to a factor of 2. This system contrasts from ours in that we do not rely on an in-RAM cache to reduce the write costs and also we optimize concurrent access between the SSD and HDD whereas they do not.

Micro-electric mechanical systems (MEMS) is another technology which has been the focus of hybrid storage research. MEMS is an upcoming technology with many applications. As a storage unit, MEMS exhibits fast I/O speeds, small physical size to storage capacity ratio but a high cost to storage capacity ratio. This has motivated research into improving the performance of secondary storage by creating hybrid systems utilising both HDD and MEMS. There has been a great deal of past research into the practical applications of MEMS [9, 10, 22, 44]. One important distinction is that sequential access in MEMS is faster than random access whereas for flash memory this is not the case.

A specific example is the research by Uysal et al. [37] on MEMS used as secondary storage in large storage arrays. In this work, they cover a wide range of possible architectures in which MEMS and HDD could be used together to gain the advantages of both. Four of the methods proposed utilised hybrid ideas, however all except the MEMScache required large amounts of MEMS storage to allow for large storage sizes. In all these cases, the amount of MEMS storage required had to equal the total storage desired, since the HDD array acted as a supplement for the data storage. A preferable system price-wise would have allowed for lower capacities of MEMS storage to be used, albeit for a possible performance and redundancy degradation.

This situation is discussed by Hong [16]. In this research, MEMS storage is simulated either within the disk itself or on the disk controller as another level in the storage hierarchy, allowing the proposed architectures to appear as a single disk. This is proposed in two architectures, with MEMS acting as a cache and with MEMS acting as a write buffer. Both these architectures address the issue of the large MEMS requirements of the Uysal et al. MEMS arrays, thus keeping the cost/size ratio low and still providing an increase in I/O performance, although they do not offer the same levels of redundancy.

Both of these research methods showed ways in which MEMS storage could be used to improve standard HDD performance. The capacities of extra storage used were either large, when used as bulk storage, or smaller, when used as a cache or buffer. In contrast, our research aims to use a small limited capacity SSD for bulk storage to achieve near optimal performance. That is, we do not use the SSD as a cache but also do not assume the existence of a large cache. We also focus on using flash memory which is currently much more popular than MEMS. We can not reuse algorithms developed for MEMS since flash has different characteristics compared to MEMS.

The work by Wang et al. (Conquest)[38] and Garrison et al. (Umbrella)[12] also use different types of memory at the same level of the storage hierarchy. Conquest uses a simple partitioning approach in which they place all small files and metadata (e.g. directories and file attributes) in NVRAM (Non-Volatile Random Access Memory) while the remaining large files and their associated metadata are assigned to the HDD. Umbrella allows the user to set policies for file placement across devices. Our work differs fundamentally from Conquest and Umbrella by allowing different pages of the same file to be placed across different storage devices. This enables our system to support intra-file concurrency. We also assign pages to devices by automatically analysing representative workloads to determine the assignment that will minimise the total I/O costs. In contrast, their work uses ad-hoc assignment policies.

4.2 Non-HDD storage device as cache or buffer

In this section we cover hybrid research which uses small amounts of additional memory as a new layer in the storage hierarchy generally between the existing HDD and RAM. This is normally in the form of a cache or buffer where a new working copy of the data is made in the additional memory. This extra copy is redundant

for reads and if updated must eventually be flushed to the HDD to keep the data in sync.

There is an existing type of HDD called hybrid drives. These drives supplement the HDD with a small amount of flash memory built inside the HDD itself, acting as a cache along with the traditional RAM cache. Microsoft Windows Vista utilises hybrid drives with an option known as ready drive [28]. It allows the flash memory in a hybrid drive to act as a buffer or cache and was primarily developed to improve the power usage of laptop systems by allowing the HDD to spin down during low workload times. When the HDD is spun down, the flash memory buffer is used to absorb all read and write requests. They also use the cache to improve performance by using simple ad-hoc cache insertion policies. These include only caching requests smaller than a certain threshold and inserting requests to the page file.

Adam Leventhal [24] discusses how flash can be used to enhance Sun's enterprise-class file system called ZFS. In ZFS flash can be used as a second level adaptive replacement cache (L2ARC), sitting between the RAM cache and the HDD. L2ARC is used to cache both reads and writes. L2ARC maintains a directory which is used in the event of a system failure to instantly warm the cache to reduce the slow performance ramp caused by system reset. ZFS can also use flash as a fast log device. Some flash devices can perform writes very fast by inserting a DRAM write cache and then treating that write cache as non-volatile by adding a superconductor to provide enough power to flush outstanding data from DRAM into flash when there is a power loss.

Baker et al. [3] proposed the use of a small amount of NVRAM in the form of battery-backed RAM to act as a small write buffer to reduce disk accesses. Their aim is to prevent losing recent updates to file caches without having to continuously write data back to the disks as soon as updates occur. A write buffer in the form of battery-backed RAM is ideal for this since it operates at the same speed as RAM and is also non-volatile which means data will not be lost due to machine failure.

Miller et al. [30] designed a system called HeRMES which uses a form of non-volitive RAM called Magnetic RAM (MRAM) to act as a persistent cache for an HDD. They use the MRAM to cache the file system metadata and also buffer writes to the HDD. Caching the metadata allows the frequent metadata requests to be satisfied very fast. Buffering the writes allows the file system to order the writes to achieve a better layout of pages on the HDD.

These works differ from our approach in that they all aim to produce a buffer or cache to temporarily store some data on a small amount of additional memory above the HDD. SSDs are now larger and therefore an approach using the flash for bulk storage is more appropriate since it avoids required redundancies and synchronisation costs.

4.3 Log and Flash Based File Systems

In this section we will discuss research done for file systems to improve the placement on individual devices (non-hybrid file systems). Our solution to the drive assignment problems is designed to work with any of these file systems to manage the individual devices below our system.

A popular file system alternative is the log-based file system (LFS) which has undergone a large amount of research for both HDD and SSD [32, 39, 45]. A log-based file system stores all writes as log entries in a sequential structure on the disk. As a result, there is a reduced amount of seek performed by the disk during write operations since multiple random writes can be replaced by a single sequential write to the log. A primary concern for LFS is the way in which we reclaim or clean areas of the disk which are filled with old versions of data, since the log requires a long sequential segment to take advantage of the logging feature. This reclamation is, in some ways, similar to flash garbage collection.

Many of the traditional file system assumptions change when using flash memory mainly due to asynchronous read/write times and no update in-place. Gal and Toledo present a review paper [11] on existing research and patents covering these aspects of flash file systems. Among the topics included in the paper are the following: mapping within the Flash Translation Layer (FTL), garbage collection, wear leveling techniques and the structure of file systems. Each of these topics is discussed in detail with references to research already completed on that topic. Of particular interest is the background work completed with NAND flash memory file systems.

A detailed description of Yet Another Flash File System (YAFFS) was covered as a specific file system for NAND flash. YAFFS is a log-based file system which requires all of its file maps to be stored in main memory. To reduce the amount of memory required for this storage, YAFFS file pointers only point to the

area in the storage where the i-node is stored, then each of the file headers in that area must be checked until the file is found. To avoid updating the file headers while extending a file (avoiding in-place updates), the file headers contain information on the size of the file. When a file is extended, a new tail portion that contains information on the size of the tail section is written to the file. The tail size and the original file size in the header then give the total size of the file.

Choi et al. [8] propose an FTL which is designed to work efficiently under journaling file systems. Journaling file systems duplicate the same file system changes in both the journal region and the home locations of the changes to guarantee consistency. However, the duplication degrades the performance of the file system. Choi et al. use a journal remapping technique to efficiently eliminate redundant data in the flash memory as well as preserving the consistency of the journaling file system.

Recent work by Caulfield et al. [5] present a system called Gordon, which is designed to use multiple arrays of flash memory to build fast, power efficient clusters for data-intensive applications. They designed an FTL which offers three types of striping: vertical; horizontal; and 2D, which is a combination of vertical and horizontal. Their results show that Gordon can outperform disk-based clusters by 1.5x and deliver up to 2.5x more performance per watt.

Our work differs from the work on single device file systems because we aim to develop a system which uses two different technologies together rather than a single one. In doing so, it is expected that systems such as YAFFS, CFFS or log-based file systems can be used beneath our system to manage the data on the individual devices specifically.

5 Problem Definition

Having given an explanation of the characteristics of the pertinent technologies and recent research into optimising them, we are now in a position to precisely define the problem we want to study. In doing so, we will also highlight the areas of the problem which are of extra significance.

In this research, we aim to produce a drive assignment algorithm which assigns pages to reside in either the HDD or SSD in order to improve I/O performance of the system. The system will have two stages: online and offline. During the offline stage, the drive assignment algorithm will decide which pages to place on the SSD to maximise performance. The pages are then placed in specific locations on their respective drives using some underlying file system (placement algorithm), the aim being that when the system comes back online, we will gain improved performance by taking advantage of the different device characteristics and concurrency. Both the offline and the online stage have two main components, the assignment of pages to a drive and the placement of pages within each drive.

The offline drive assignment algorithm will use a predicted workload which can be the previous online period's workload. However, this is not a restriction and the predicted workload which the offline algorithm uses may be compiled in any way. Developing the drive assignment algorithm is our primary problem. We can summarise it as follows:

> For a system utilising both an HDD and an SSD, what is the optimal drive assignment of pages for a data set in order to perform a set of requests in minimum time?

The workloads we will be considering will include read, write and growth requests, where a file growth request occurs when data is appended to the end of an existing file. File deletion is not considered because once a file has been deleted, it is no longer in our data set. After the drive assignment is performed, a placement algorithm is used to arrange the location of the pages within each drive in order to further improve I/O performance. In this research, we do not propose a new placement algorithm but rather assume a simple algorithm which groups pages of the same file together and stores them sorted by logical page number.

5.1 Assumptions

Our problem takes page grained request sets (or workloads) as input where each request is to a part or all of a file. In the situation where an operating systems buffer exists the requests model the cache misses to consecutive pages stored on either or both the SSD and HDD.

One of the primary assumptions we have made is that we are only able to achieve concurrency within each request. That is, we assume that there is no inter-request concurrency, and hence no concurrency between

files since each request can only be for all or part of one file. We make this assumption because in order to take advantage of inter-request concurrency, we would need to be able to predict the order by which requests are issued. However, the order is often unpredictable. It does not mean our proposed algorithms will not work in a setting where there is concurrency among file accesses. It just means they will not be optimized for this situation.

When describing pages on HDD and SSD, it is important to note that the two devices may have different sized pages. The sizes are typically 2KB or 4KB for SSD and 4KB or 8KB for HDD. Since the commonly used sizes for HDD are multiples of those used for SSD, we assume that the page size of the HDD is a multiple of the SSD page size. We then use the HDD page size as the system page size, meaning that we require multiple SSD pages for each system page. This avoids any issues of external fragmentation that could occur if the sizes were not multiples of each other.

We assume the existence of a flash transition layer beneath our system. This is so the garbage collection and arrangement of the pages on the SSD does not interfere with the decision algorithm. These costs are then incorporated into the write cost of the SSD.

We use average data transfer costs on the two devices (SSD and HDD) to represent the costs for transferring data from the two devices to RAM. For example if there are a lot pages with content that has not been erased then writing may result in many erasures which in turn leads to much higher costs. However, if the SSD is very close to a clean state with many pages already erased then writing would be much faster. In order to avoid modeling the SSD using a complex model that includes the erase write cycle, we have model the SSD using average data transfer costs instead. This approach also has the added advantage of allowing our model to abstract over the flash translation layer(FTL) and thereby making our solution generic with respect to the FTL.

We model the HDD using average seek cost rather than a page placement dependent seek cost. In a real system seek costs on HDDs are dependent on the exact relative locations of the sought pages on the disk. Namely seeking a shorter distance will incur a lower cost. Therefore constant average seek cost is not as accurate as placement sensitive seek costs. However since our focus is on the drive assignment problem instead of the page placement problem we do not want the performance to be heavily influenced by the placement algorithm used. Therefore we have chosen to use a constant average seek time rather than one that is dependent on the exact distances between successive seeks.

It is assumed that the HDD will always be of sufficient size to hold the entire data set if necessary. This is a reasonable assumption since existing systems would be running off HDD alone.

5.2 Formalisation

This section describes in formal notation the problem space we will be exploring and what bounds are given. We will require the following definitions in order to describe the system:

Request: $\langle fileID,\ start,\ length,\ type \rangle$, where *start* is page id of the 1st requested page, *length* is the number of pages requested and *type* is the type of request and can be *read*, *write* or *growth*.

Drive Assignment: $\langle fileID,\ pageID,\ drive \rangle$, where *drive* is the storage device the page is stored in and can be either SSD or HDD

Placement: $\langle fileID,\ pageID,\ drive,\ position \rangle$, where *position* is the physical page id where the logical page id $pageID$ of file $fileID$ is stored.

System Setup: $\langle HDDSize,\ SSDSize,\ HDDSeek,\ HDDTransfer,\ SSDRead,\ SSDWrite \rangle$, where $HDDSize$ is the size of the HDD in the terms of number of pages, $SSDSize$ is total capacity of the SSD, $HDDSeek$ is the average cost to perform a seek on the HDD, $HDDTransfer$ is the cost of transferring one page from the HDD to RAM or vice versa, $SSDRead$ is the cost of reading one page from SSD into RAM and $SSDWrite$ is the cost to perform one write onto the SSD. All costs are in units of seconds.

We will also need the following to describe the various states the data are in during different stages of processing the requests:

- Let R denote a chronological sequence of requests, with r_i denoting the i^{th} request.

- Let D denote the set of drive assignment sets for all requests, with d_i being the drive assignment preceding the i^{th} request where a drive assignment is a total mapping from every page of every file to either the SSD or the HDD.
- Let P denote the set of page placement sets for all requests, with p_i being the placement preceding the i^{th} request where a placement is a total mapping from every page of every file to a unique location on the SSD or HDD.

In our problem, the following are given:

- A chronological sequence of requests R.
- A system setup S.
- An initial set of files E.
- An initial placement algorithm. Using the initial drive assignment, this algorithm places the pages onto their associated devices.

$$F(d_0) \rightarrow p_0 \tag{1}$$

- A dynamic placement algorithm. This algorithm takes the previous placement and the current drive assignment and places the pages onto their associated devices.

$$G(p_i, d_{i+1}) \rightarrow p_{i+1},\ i \geq 0 \tag{2}$$

- A dynamic drive assignment algorithm. This algorithm takes as input the current drive assignment and the requests which follow it and outputs a new drive assignment. This is needed since, in the case of a growth request, we require the new pages to have an assigned drive.

$$B(d_i, r_i) \rightarrow d_{i+1},\ i \geq 0 \tag{3}$$

- A cost model $C(P, R) \rightarrow cost$, defined in Equation 5.

We assume the above algorithms are given because we want to focus on the offline drive assignment problem which is the primary concern when using multiple storage technologies. The problem of where to place data within individual devices has already been addressed in file system research (as discussed in Section 4) and so we assume a simple algorithm, since it is not the primary focus of our research.

5.3 Drive Assignment Problem

Using the given algorithms and system information, we will aim to produce a drive assignment algorithm A(R, F, E, S) which outputs the initial drive assignment d_0 as follows:

$$A(R, F, E, S) \rightarrow d_0 \tag{4}$$

such that the *cost* is minimised. Using the d_0 from A, we can then use F to determine p_0 which allows us to compute all the subsequent d_i and p_i using B and G respectively. Once all of these sets are computed, we can calculate the desirability of the outputted initial placement d_0 by computing the cost of the requests R using the cost model C and the set of placements P generated by taking d_0 as input.

5.4 Cost Model

The cost model we present uses the request sequence and calculates the cost of each request based on the placement of the pages at that point. This calculation is based on the average access times of the storage devices involved. We use this cost model as the way of determining the system performance. We will use a simplified version of this cost model to make efficient cost (albeit less accurate) estimations when determining the benefit of using a certain initial drive assignment during the offline stage.

We will need the following definitions to assist in describing the cost model.

- Let CSR be the cost of reading one page in the SSD.
- Let CSW be the cost of writing one page to the SSD.
- Let CHT be the cost of transferring one page of data to or from the HDD.
- Let CHS be the cost of seeking in the HDD. It is the average cost of seeking to a location in the HDD including the rotational latency.
- Let $S_{(i,j)}$ be the set of SSD pages for request r_i according to p_j.
- Let $H_{(i,j)}$ be the set of HDD pages for request r_i according to p_j.

We can now define the cost model $C(P, R)$ as the sum of all read, write and growth requests for the set of placements P as follows:

$$C(P,R) = \sum_{r_i \in R} CostOfRequest(r_i, P) \tag{5}$$

$$CostOfRequest(r_i, P) = \begin{cases} CostOfRead(r_i, p_i) & \text{if } r_i \text{ is a read request.} \\ CostOfWrite(r_i, p_i) & \text{if } r_i \text{ is a write request.} \\ CostOfWrite(r_i, p_{i+1}) & \text{if } r_i \text{ is a growth request.} \end{cases} \tag{6}$$

Equation 5 sums the cost of all the read, write and growth requests separately, allowing us to multiply through the actual costs of performing the particular action on each device. Any growths are considered as the cost of writing to the pages where they are placed by the dynamic placement algorithm. This placement is selected as the request is run so we are required to use p_{i+1} page placements after the current growth request r_i (rather than the preceding placement which is the norm for read/write). This is because unlike read and writes, we only know where the pages will be after the request is complete rather than before.

$$CostOfRead(r_i, p_j) = \max(CSR \times \left|S_{(i,j)}\right|,\ CHT \times \left|H_{(i,j)}\right| + CHS \times \sum_{q \in H_{(i,j)}} IncurSeek(q, r_i, p_j)) \tag{7}$$

$$CostOfWrite(r_i, p_j) = \max(CSW \times \left|S_{(i,j)}\right|,\ CHT \times \left|H_{(i,j)}\right| + CHS \times \sum_{q \in H_{(i,j)}} IncurSeek(q, r_i, p_j)) \tag{8}$$

We define the cost of writing and the cost of reading as the maximum of the costs to read/write off the SSD and HDD. This simulates the intra-request concurrency we achieve by reading/writing off both devices concurrently. We only incur a seek cost for any HDD page if the page placed directly before this on the disk does not belong to the same request. In order to calculate whether we include a seek or not, we simply look to see if the page before the current HDD page belongs to the same request.

$$IncurSeek(q, r_i, p_j) = \begin{cases} 0 & \text{if the page before q on the HDD is in the same request.} \\ 1 & \text{otherwise} \end{cases}$$

6 Methodology

In the last section, we described the drive assignment problem. In this section, we will detail our solution to this problem and the system built around it.

6.1 System Overview

Our system is configured to optimise the usage of an SSD and an HDD used in combination. Our system will typically be used with a relatively small SSD compared to the HDD. Typically, the SSD will only be large enough to store a portion of the data. Both storage devices will be used to store the actual data of the system without redundancy, and not be used as a buffer or cache.

Once the drive assignment is determined the system makes use of existing algorithms to choose where to place the data on the devices. In the case of the SSD, it is intended to work above the flash translation layer and so the actual physical locations on the SSD will be completely unknown to us. In both cases, it is intended that this system would "hijack" the inode (index page used to locate data pages in NTFS or ext3 file systems) lookup table of the file system in order to make the conversion between logical pageIDs to the physical location of the page on the SSD or HDD. More specifically we would reserve one bit of the inode pointer to indicate whether the page being pointed to is on the SSD or HDD. The remainder of the pointer would describe the location of the page on that specific device. This allows us to make the changes on the storage devices transparent to the operating system. Also, there is no extra lookup cost involved when using the system since any lookups we make would have been completed anyway.

6.2 Offline

We perform drive assignment offline during system down time. That is when the system is not being used by users. Typically this would happen at night or some other period of time when the system is not required. The reason we perform the drive assignment offline is that we want to achieve a near optimal global solution. This requires the possible migration of many data pages from the SSD to the HDD and vice versa. The only way to achieve this without major disruptions to users is to do the drive assignment offline.

We tackle the offline drive assignment as a cost-based search problem. That is, we aim to find the assignment that minimises the cost formula defined in Equation 5. The primary challenge is that we cannot calculate a fixed cost for each of the pages in the data until we have already placed all *related* (accessed in the same request) pages, hence making it difficult to create an algorithm that incrementally builds an assignment one page at a time without any initial placements. The reason for this is that moving one page into the SSD or HDD or even just changing its physical location within the same device is likely to affect the cost of logically nearby pages, since those pages may be related. Also compounding this is the fact that some pages are accessed by multiple requests which overlap. In this case, it is not clear which related pages involved in the overlap should be grouped together, given they can be accessed according to different access patterns by the different requests. Finally, by allowing intra-request concurrency, the pages in requests with overlap incur a much more dramatic cost impact on the system. This is because the overlapping requests will want to use concurrency, but their choice of drive assignments to achieve this may differ for the pages in the overlap. All these issues show us that our problem is not trivial to solve. In fact, it turns out the drive assignment problem is an instance of the NP-complete multiple choice knapsack problem, hence making our problem NP-complete.

The multiple choice knapsack problem (MCK) is a variant of the well known knapsack problem. Both of these problems are NP-complete. The basic knapsack problem involves the selection of items from a set where each of the items has an associated size and cost. The aim is to find the set of items for which we maximise the cost while not exceeding a pre-imposed total size limit. In MCK, the set of items is sub-divided into disjoint classes from which we choose exactly one item from each to place in the knapsack while maximising cost.

Theorem 6.1 *The drive assignment problem defined in Section 5.3 is NP-complete.*

Proof *There exists a polynomial map from MCK to the drive assignment problem. See Appendix A for details of the proof.* ∎

Completing the assignment algorithm in optimal run time is not our primary concern because the offline stage happens during down time for the system. However, we still need it to complete in reasonable time since a system's down time is usually bounded. In order to cope with the NP-complete problem, we propose a pipeline approach to the offline stage which uses heuristics and cost estimations to make the drive assignment choice. Experiments in Section 8 show that our algorithms can solve the drive assignment problem in less

than 10 seconds for a data set of 20000 pages. In addition to solving the drive assignment problem, the system also needs to move the data from their original location to new locations. The time needed for this will depend on the amount of data moved. We can minimize this migration time by using IO scheduling algorithms so that pages are written into the HDD with the minimum amount of seek time.

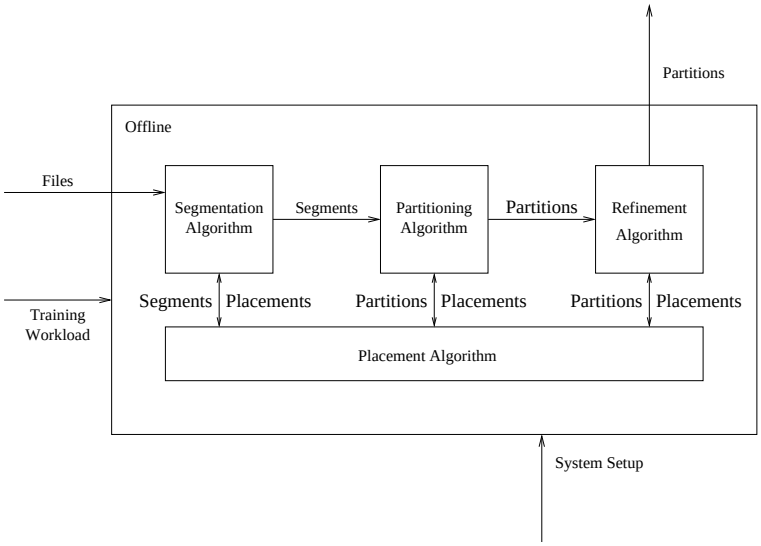


Figure 1: Offline Overview

The offline stage is run as a pipeline which consists of three parts which interact with the fourth component, the placement algorithm, in order to determine the cost of partial/full drive assignments as seen in Figure 1. These stages work together to decide which pages will be on the SSD since it is limited in size while the HDD will always have enough room. The entire offline stage can be seen as the drive assignment algorithm $A(R, F, E, S)$ (Equation 4) designed to solve the drive assignment problem described in Section 5.3. The three pipeline stages require the placement algorithm to be able to check the cost of a certain segment or partition. An outline of the tasks each stage performs is described as follows:

6.2.1 Segmentation Algorithm

We cannot initially determine the exact cost of the different pages or requests because the placement of one of the pages affects the cost of all requests which use it and the costs of related pages. Because of this, it is difficult to begin by incrementally choosing a page at a time for the SSD. Instead, we begin with the segmentation algorithm which breaks up the pages within each file into contiguous and non-overlapping segments which we will consider as independent for our initial choices. These disjoint segments are intended to be representative of requests which are made to the system. Although this will not always be exact since sometimes requests will overlap. if requests overlap then segments cannot both exactly represent the overlapping requests and remain disjoint.

6.2.2 Partitioning Algorithm

The partitioning algorithm then chooses which segments are most worthy of being placed in the relatively small SDD. The benefit of choosing a segment for the SSD is estimated using the cost difference per SSD page between placing the entire segment on the HDD versus placing the optimal portion of the segment on the SSD. The calculation for this optimal portion is via the optimal split which is given in Equation 12 (Section 6.5). The optimal split determines which portion of the segment will be placed on the SSD, with the remainder being placed on the HDD. The optimal split is defined such that the cost of loading the segment is minimised by taking maximum advantage of concurrency. Placing only the optimal portion of the segment on the SSD has the added benefit of reducing the amount of data placed on the SSD and thereby preserving a larger remaining capacity for other segments.

6.2.3 Refinement Algorithm

In the segmentation algorithm, we made the assumption that the segments were completely independent. We also used the optimal split as a way of moving pages across to the SSD in bulk. However, situations exist where the segments do not accurately represent the requests. We use the refinement algorithms to fix the performance degradation caused by these inaccuracies. The refinement algorithm can now calculate a much more accurate cost for all the pages since we have an initial placement to work with. The refinement algorithm is designed to use the more accurate calculations to incrementally move individual pages in and out of the HDD and SSD in order to further improve the I/O performance. As the refinement algorithm makes changes to the drive assignment, it also has to update placements via the placement algorithm in order to compute the cost of the changed partition. When the refinement algorithm has finished, it simply passes the final partitioning to the online stage as the drive assignment.

6.2.4 Placement Algorithm

The placement algorithm uses partitions from the partitioning or refinement algorithms or segments from the segmentation algorithm to decide the location on the devices to place each page. These placements can then be used to calculate costs for the different segments or partitions. We use a simple placement algorithm which places all files in an arbitrary order and with each page in order within each file. It is at this stage that any spatial or temporal localities may be taken advantage of for the placement of pages in the HDD and SSD. For future work, we can design placement algorithms which place frequently written pages together to reduce the cost of erasures , as well as other placement algorithms already established in existing research. However, in this paper, we focus on the drive assignment problem and therefore assume a simple placement algorithm which we model using constant HDD seek cost and SSD write cost.

6.3 Online

The online stage of the system takes the place of the normal inode lookup that might occur in a file system such as NTFS. Its primary purpose is to map logical pages to physical pages on the two devices and issue these requests to the devices. It does this using the file system's existing inode functionality. There is a cost involved in merging concurrently run requests back into a single sequence of pages for the user. This cost would be variable depending on the needs of the operating system and the way in which the pages are merged into a single response. We do not count this cost in our system measurements since we cannot easily predict the exact nature of this cost without developing an algorithm to join the responses, however we make the reasonable assumption that such costs are negligible compared with the I/O costs.

What the online stage adds to the system is the ability to translate the offline stage's partitions into placements using the initial placement algorithm $F(d_0)$ (Equation 1). It is also required to handle growth requests from the operating system and use the dynamic drive assignment $B(d_i, r_i)$ (Equation 3) and dynamic placement $G(p_i, d_{i+1})$ (Equation 2) to make a decision as to the physical location of the new pages.

For our system, we use a dynamic drive assignment which assigns new pages (from growth requests) onto the HDD and never moves any other pages. We also use a simple dynamic placement algorithm which simply searches the HDD for the first available free page to place new pages and never reassigns the positions of existing pages. The reason for these two simple algorithms is we want to focus on the offline drive assignment problem and leave the complex online algorithms for future work. We also use an initial placement algorithm which places the pages on the devices grouped by files and sorted by logical pageIDs.

Finally, it is expected that the online stage would collect any data required to generate the predicted workload to be used by the next offline stage. This is not a requirement but would likely be a common style of implementation since we can use the current online stage as a prediction for the next.

6.4 Segmentation

As described earlier, the segmentation algorithm is the first step in the offline pipeline. It is used to create groups of pages to make a fast initial selection of pages for the SSD. These groups are considered for the SSD in the partitioning algorithm which follows in Section 6.5. The segments are designed to be representative of the requests given in the training workload. We propose three variations for this stage of the pipeline: file;

begin-end and join segmentation. The three algorithms are described here and some examples are given in Figure 2. The arrow heads in the diagram indicate the location at which the file is cut into segments. In the figure, we show the segments that would be produced by file and begin-end segmentation and two examples of possible segments produced by join segmentation.

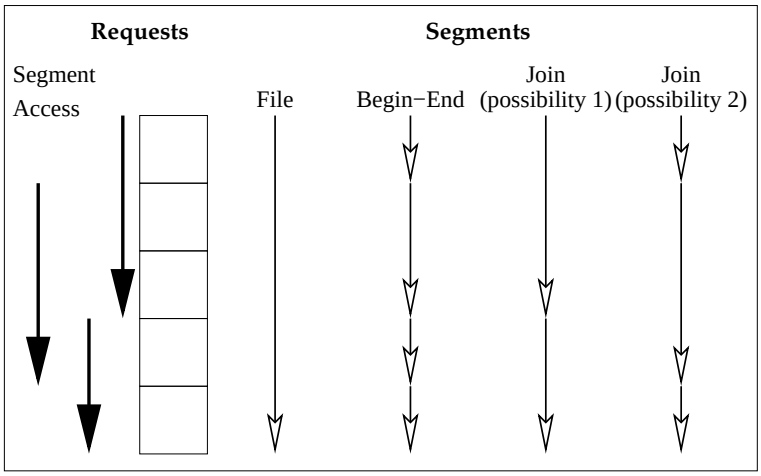


Figure 2: Segmentation Examples

6.4.1 File Segmentation

File segmentation is the simplest possible segmentation algorithm. It simply sends through segments which are the whole file. This algorithm is expected to work well, mainly with full file accesses since the pages in the file are treated equally and have equal expectations.

6.4.2 Begin-End Segmentation

Begin-end segmentation is a more complex algorithm in which the file is broken into segments at the start and end of every request issued. We propose this as an alternative to the file segmentation algorithm because file segmentation does not take into account files having only part of their data accessed. Begin-end segmentation, in comparison, gives us segments which match exactly with the requests given for non-overlapping workloads. Overlapping requests, on the other hand, will yield new segments starting whenever a request starts or ends.

6.4.3 Join Segmentation

Join segmentation is the most complex of the algorithms and works in a similar way to begin-end segmentation. Join segmentation uses the start and end points of the requests as possible join or split points. In this way, it can create the same output as begin-end segmentation or file segmentation for any type of access. It can also create longer segments than the begin-end segments by joining consecutive segments together. We use join segmentation because begin-end segmentation treats individual segments as independent. When the size and frequency of request overlaps becomes more significant, begin-end segmentation will be insufficient to produce the optimal amounts of concurrency. Join segmentation can join two segments so that we treat them together rather than individually. It is hoped that these situations for the begin-end and file segmentation will be fixed by the refinement algorithms but we can do it much more efficiently here.

The join section can be represented by a binary number where a 1 represents a break and a 0 represents a join at a possible join position. The possible join positions are the begin and end points used by the begin-end segmentation. This representation gives us the number of possible join segmentations equal to 2^{n-2} where n is the number of unique join points including the start and end of the segment as shown in Figure 3. The figure shows a set of requests, the possible join points and the resulting segments. Notice that while the start and end of the file are considered as possible join points, they will always be included as cuts since we cannot extend past the bounds of the file.

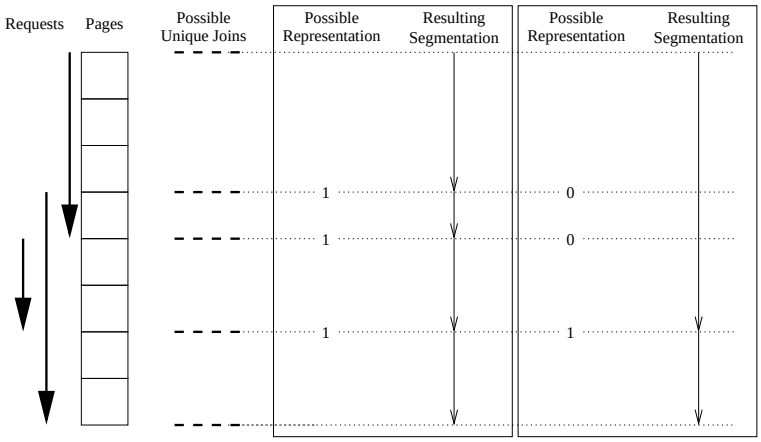


Figure 3: Join Representation Example

We use the simulated annealing process as a way of searching the possible segments for the best places to join. A specific choice of joins d is compared with another choice of joins e by computing $\sum_{s \in S(d)} costSegment(s)$ where $S(d)$ is the set of all segments produced by the choice of joins d and $costSegment(s)$ is defined according to Equation 20 below. The joins d are considered better than the joins e if $\sum_{s \in S(d)} costSegment(s) < \sum_{s \in S(e)} costSegment(s)$.

The way the join decision is made is through a simulated annealing process and is described in Algorithm 1. We use simulated annealing so that we may pick the joins that will give us the optimal segments to consider for the SSD. This improves our segmentation in the case where an overlapping section may be better considered as part of one of the adjoining segments rather than independently as is the case in begin-end segmentation (L.4). We begin with the begin-end segmentation and explore the neighbourhood of this choice by joining some of the segments in order to improve our initial guess (L.13-17). We use the cost difference of HDD only versus optimal split of segments per page used for optimal split as our measure of cost to be optimised. The optimal split of a segment is the number of pages we place on the SSD in order to obtain maximum concurrency with the HDD and will be discussed in Section 6.5. This allows us to try and make segments which will have the best possible optimal split for the number of pages they take in the SSD.

6.4.4 Segment Cost

For a single segment, we only calculate the costs of the requests which interact with it because in this stage, we consider the placement of each segment in isolation to other segments. It is the refinement stage where we consider the global cost of all segment placements as a whole.

The cost advantage of a segment is the cost difference between all the segment pages being placed on the HDD and the segment having the optimal portion of its pages placed on the SSD. This difference is then divided by the number of pages used in the optimal portion. This allows us to rate segments based on the amount they can improve the system performance with the least use of SSD capacity. Equation 20 is computed for segment s where R is the set of requests which interact with s which have already been restricted to the bounds of s, p^h is the placement with s on the HDD and p^o is the placement where the optimal split is used.

$$costSegment(s) = \sum_{r \in R} \frac{costRequest(r, p^o) - costRequest(r, p^h)}{numPagesInOptimalSplit(s)} \tag{9}$$

6.5 Partitioning Algorithm

This algorithm is the second component for the offline stage. In this algorithm, we choose which of the segments we will use to fill the SSD and which will remain wholly on the HDD. The costs that we are using assume no sequential links between segments. This assumption is not necessarily true since begin-end

```
   input  : Files, Requests, Temp, terminateTemp, terminateInner, neighbourhood
   output: Segments
 1 foreach File f in Files do
 2     //Setup simulated annealing
 3     T← initialTemp;
 4     JoinsA← Begin-End segmentation;
 5     costA← costJoin(JoinsA);
 6     bestCost← costA;
 7     bestJoins← JoinsA;
 8     //Start simulated annealing
 9     while T > terminateTemp do
10         s← 0;
11         while s < terminateInner do
12             s ← s+1;
13             //Pick a random neighbourhood point
14             JoinsB← JoinsA;
15             index← random join point in file;
16             JoinsB[index]← NOT JoinsB[index];
17             costB← costJoin(JoinsB);
18             //Use the new point if it is better
19             if costA < costB then
20                 costA← costB;
21                 JoinsA← JoinsB;
22             end
23             //Give a chance to use the new point anyway
24             else if  rand() < e^(-|costA-costB|/T)  then
25                 costA← costB;
26                 JoinsA← JoinsB;
27             end
28             //Update global best
29             if bestCost > costA then
30                 bestCost← costA;
31                 bestJoins← JoinsA;
32             end
33         end
34         T← updateTemp(T);
35     end
36     //Now use the best join we found to create segments
37     JoinsToSegments(bestJoins, StartEnds);
38 end
```

Algorithm 1: Join Segmentation Algorithm

and join segmentation produce segments which break up overlapping requests. Because of this, we do not worry too much about trying to gain an optimal solution at this point. To this end, we simply use a greedy algorithm which incrementally selects the segment with the largest cost improvement per SSD page used. The cost improvement per page is given in Equation 20 (Section 6.4.4).

This means the best cost improvement is the largest negative. If a segment is considered not to be an improvement (cost improvement ≥ 0), the segment is not considered and the greedy algorithm stops since we have already placed all the better segments on the SSD.

When considering a segment for the SSD, we may not wish to place all pages on the SSD because we can gain concurrency by leaving some pages on the HDD. How many pages we use is a simple concept, but a more complex equation. The basic idea is that we want to put pages on the SSD until the cost to read all those pages is less than or equal to the seek cost of the HDD. Once we reach this balance point, we simply need to add pages to the two devices at the ratio between SSD read time and HDD transfer time. The problem lies in the fact that we may also need to be writing to the SSD in the same segment and so the balance point and subsequent ratio after that are different for read and write. We could adopt a 50/50 approach but then this would disadvantage segments that have a higher number of reads than writes or vice versa. So, we calculate the read split and the write split and then take the weight of them according to the average read and write heat of the segment and the read and write costs. The optimal split is calculated using the following calculations:

$$\begin{aligned} splitRead = min\Bigg(&max\left(segmentLength - \frac{SeekTime}{SSDRead}, 0\right) \\ &\times \frac{HDDTime}{SSDRead + HDDTime} + \frac{SeekTime}{SSDRead}, segmentLength\Bigg) \end{aligned} \tag{10}$$

$$\begin{aligned} splitWrite = min\Bigg(&max\left(segmentLength - \frac{SeekTime}{SSDWrite}, 0\right) \\ &\times \frac{HDDTime}{SSDWrite + HDDTime} + \frac{SeekTime}{SSDWrite}, segmentLength\Bigg) \end{aligned} \tag{11}$$

$$optimalSplit = \lfloor splitWrite + \frac{segmentReadHeat \times SSDRead \times (splitRead - splitWrite)}{segmentReadHeat \times SSDRead + segmentWriteHeat \times SSDWrite} \rfloor \tag{12}$$

In Equations 10 and 11 $\frac{SeekTime}{SSDRead}$ (or similar for write) represents the number of pages which can be read from the SSD while waiting for the HDD seek. If this exceeds the size of the segment, then placing all the pages on the segment in the SSD is the optimal split. Otherwise, for the remaining pages, we must compute the number of pages we can read/write off the SSD while we are reading/writing off the HDD.

Once we have calculated the size of the optimal split x, we place the first x pages into the SSD if this segment is chosen and there are at least x page slots remaining in the SSD. We do not use a more complex algorithm for this determination since the segments are the same as the requests (unless we join them in join segmentation or there is some overlap) and so there is no advantage in choosing one page over another within the segment. In the other cases, we expect the refinement algorithm to be able to rearrange any out-of-place pages.

We give an example of computing the above formulas. In this example we will look at a single segment of 30 pages with 1000 sequential reads and 100 sequential writes across the length of the segment. The hardware specifications we use are the same as those given in Table 2, namely HHDTime = 0.0313ms, SeekTime = 4.17ms, SSDRead = 0.149ms, SSDWrite = 0.198ms. Please all times units for the example below are in ms.

$$\begin{aligned} &splitRead = min\left(max\left(30 - \frac{4.17}{0.149}, 0\right) \times \frac{0.0313}{0.149 + 0.0313} + \frac{4.17}{0.149}, 30\right) \\ &splitRead = 28.34 \end{aligned} \tag{13}$$

$$splitWrite = min\left(max\left(30 - \frac{4.17}{0.198}, 0\right) \times \frac{0.0313}{0.198 + 0.0313} + \frac{4.17}{0.198}, 30\right)$$
$$splitWrite = 22.28 \tag{14}$$

$$optimalSplit = \lfloor 22.28 + \frac{1000 \times 0.149 \times (28.34 - 22.28)}{1000 \times 0.149 + 100 \times 0.198} \rfloor$$
$$optimalSplit = \lfloor 27.63 \rfloor \tag{15}$$
$$optimalSplit = 27$$

This means that for our example segment the optimal (according to the training workload) placement of pages on the SSD would be 27 pages with 3 pages left to be used concurrently on the HDD. If the requests had been only reads or writes then the optimal split would have been 28 or 22 pages respectively.

To extend this example we will also calculate the cost of this segment according to Equation 9 and also Equations 7 and 8. In this case our set of requests R is the 1000 reads and 100 writes. Each read and write write access all 30 pages in the segment. We are comparing the optimal placement of this segment (p_o) (27 pages on the SSD) with that of all 30 pages on the HDD (p_h).

$$CostOfRead(r_i, p_o) = \max(0.149 \times 27,\ 0.0313 \times 3 + 4.17 \times 1)$$
$$CostOfRead(r_i, p_o) = 4.264 \tag{16}$$

$$CostOfWrite(r_i, p_o) = \max(0.198 \times 27,\ 0.0313 \times 3 + 4.17 \times 1)$$
$$CostOfWrite(r_i, p_o) = 5.346 \tag{17}$$

$$CostOfRead(r_i, p_h) = \max(0.149 \times 0,\ .0313 \times 30 + 4.17 \times 1)$$
$$CostOfRead(r_i, p_h) = 5.109 \tag{18}$$

Assuming the read and write cost to the HDD are the same:

$$CostOfWrite(r_i, p_h) = 5.109 \tag{19}$$

Substituting the above to the Equation 9 we have the following:

$$costSegment(s) = 1000 \times \frac{4.264 - 5.109}{27} + 100 \times \frac{5.346 - 5.109}{27} \tag{20}$$
$$costSegment(s) = -32.222 + .877 \tag{21}$$
$$costSegment(s) = -31.344 \tag{22}$$

In the end we have a total cost benefit of 31.344ms per page used in the SSD.

6.6 Refinement

This is the final step in the offline phase and aims to correct any mistakes made by the segmentation and partitioning algorithms. Since we can now calculate the exact cost difference for each page, we can address the finer grain of the more complex problems of concurrency and request overlap. We use a straightforward hill climber, given in Algorithm 2. The hill climber searches for a solution by continuously moving in the direction of a lower cost drive assignment. It terminates when it cannot find any further improvement. Note this may lead to local optima since the global optima maybe only reachable via first going to a higher cost

solution. Empty pages obviously have a cost of 0 since they store no data, but they can be used to swap out pages which disadvantage the overall cost.

We sort both SSD and HDD according to the cost difference of swapping (L.1-2) according to Equation 23. If the top pages of each list are worth swapping with each other, we do it (L.8-16) according to Equation 24. If they are not worth swapping, we check the next best SSD page but keep the top HDD page. We continue trying to swap until the number of consecutive failures > cutoff (L.6). This allows us to avoid terminating simply because we compared pages which were related, for example, swapping the first page of a request out of the SSD for the second page of the request. If the pages were swapped with any other pages, it would be worth swapping, but swapping them with each other is not.

6.6.1 Page Cost

Page cost is used by the refinement algorithms to rank pages based on the performance improvement resulting from placing a page on the SSD rather than the HDD. It is the cost difference between completing all the requests with the page on the HDD rather than SSD. This page cost is defined formally as follows:

$$costPage(p) = \sum_{r \in R} costRequest(r, p^s) - costRequest(r, p^h) \tag{23}$$

where R is the set of requests which includes p, p^s is the placement of p on the SSD and p^h is the placement of p on the HDD.

6.6.2 Swap Cost

Swap cost is used by the refinement algorithms to determine the cost of swapping an SSD page with an HDD page. This is used to ensure that before a swap is made, that it is actually profitable. The following is the cost of swapping $p1$ in the SSD with $p2$ in the HDD:

$$\begin{aligned} swapCost(p1, p2) = &\sum_{r \in R1} costRequest(r, p^a) - costRequest(r, p^b) \\ &+ \sum_{r \in R2} costRequest(r, p^a) - costRequest(r, p^b) \end{aligned} \tag{24}$$

where $R1$, $R2$ are the sets of requests which include $p1$ and $p2$ respectively, p^b is the original placement (with $p1$ on the SSD and $p2$ on the HDD) and p^a is the placement after $p1$ and $p2$ have been swapped.

7 Experimental Setup

In order to measure the performance of the system proposed in Section 6, we have simulated the offline and online stages. We have used both synthetic and real data sets. The synthetic data was created using a data generator which creates distributed workloads according to prescribed distributions of access across files and requests. The synthetic data generator allows us to control the pattern of data access which means we are able to study how our system responds to the different styles of data access. These simulations are all written in C++, and the data generator uses the GNU Scientific Library (gsl) to implement distributed random number generation. For the real data, we used two OLTP Application workloads obtained from the UMassTraceRepository [36]. These are used to demonstrate how the system performs on real data sets which can contain interesting behaviour such as metadata updates.

The online simulation is different from what it would be in a real system. We do not worry about how the lookup of logical pages would actually be done but simply take in the SSD and HDD partitions from the offline system and place them according to the initial placement algorithm (Section 6.2.4).

Our simulator made the following two approximations: 1) we use a constant average seek time for the HDD instead of one which is sensitive to seek distance and 2) we use an average write cost for the SSD which includes average garbage collection costs instead of accurately modeling the garbage collector. As explained in Section 5.1 the reason for approximation 1 is that we want to focus on evaluating the effects of the drive assignment algorithms instead of placement decisions within either device. For approximation 2, it

```
   input  : Drive assignment
   output: Drive assignment
 1 Sort(SSD, pageCost);
 2 Sort(HDD, pageCost);
 3 SSDIndex ← 0;
 4 fails ← 0;
 5 while fails < cutoff AND SSDIndex < SSDSize do
 6 |   cost ← costOfSwap(SSDIndex, 0);
 7 |   if cost < 0 then
 8 |   |   swapPages(SSDIndex, 0);
 9 |   |   calculatePageCosts();
10 |   |   Sort(SSD, pageCost);
11 |   |   Sort(HDD, pageCost);
12 |   |   SSDIndex ← 0;
13 |   |   fails ← 0;
14 |   end
15 |   else
16 |   |   fails ← fails+1;
17 |   |   SSDIndex ← SSDIndex+1;
18 |   end
19 end
```

Algorithm 2: Refinement Algorithm

is dangerous to model any particular garbage collection algorithm since all commercial SSDs use their own garbage collection algorithm whose details are commercial secrets. So as suggested by a landmark paper on understanding flash memory I/O patterns[4], it is better to treat the flash device as a black box instead of trying to model its internals accurately.

We do not simulate a buffer and therefore simulate the situation that every page request results in one page load.

7.1 Synthetic Data Generator

We use probability distributions throughout the data generator to allow us to have certain access or size probabilities for file sets and workloads. The distributions which are implemented are:

Uniform simply gives all items equal probability of access.

Gaussian spreads the items on either side of the specified mean with a specified standard deviation. Given as a parameter in the form *(g, mean, std.dev.)*.

Pareto spreads the items out from a beginning point across a Pareto curve with a specified exponent and scale, given as a parameter in the form *(p, scale, balance point)*. Pareto distributions generalise over the Zipfian distribution by allowing for scaling. As a guide, with a scale (exponent) of 1, the balance point indicates the proportion of data which will contain 50% of the cumulative distribution. That is, for a balance point of .2, we can expect the probability of picking a value in the first 20% of the data will be 50%.

When generating requests for files, we use three different styles of access:

File files are always accessed as a single sequential request for every page in the file.

Random files are always accessed as a request to a single page within the file.

Segment files are always accessed with sequential requests of varying length and starting pages. These requests overlap each other.

7.2 Default Experimental Setup

By default, we use three pure access styles: file, random and segment. In this way, we can see how our algorithms perform for each of the different access styles separately. However, in the varying access style experiment (Section 8.4), we generate workloads that contain different mixtures of the different access styles. We use Gaussian distributions for request size calculations. However, file sizes are designed to match those reported in [1] using a normal distribution on the log two scale. The file size distribution we use are shown in Figure 7.2. Pareto distributions are used for probability of access of files and requests. We allow for approximately 80% read access. The default data parameters are summarised in Table 2.

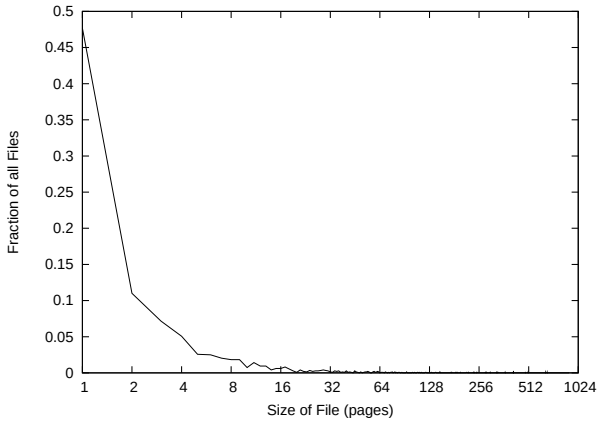


Figure 4: File Size Distribution for Synthetic Data

We do not consider file growth (new data appended to the end of a file) in the synthetic experiments, as the offline drive assignment problem has very limited impact on system performance as a result of file growth. In the real traces file growth is represented by a write request to pages beyond the size of the file. In our future work, we will address optimal file growth when we focus on the online placement problem since the online assignment and placement algorithm determines where the data is grown.

Table 2: Default Data Generation Information

Parameter	Value	Parameter	Value
Min File Size	1 page	Access Dist. (Files)	(p,1,.1)
Max File Size	1000 pages	Access Dist. (Requests)	(p,1,.1)
Total Data Size	20000 pages	Avg. Segment Size	20% file size
Number of Requests	100000	Avg. Overlap Size	20% segment size
Read Requests	80%		

The SSD and HDD systems simulated are described in Table 3. We use the performance data from the 1TB Seagate Barracuda 7200.12 [33] and the Super Talent 64GB MasterDrive EX2 [35]. We have chosen a cheap SSD since one of our aims is to keep costs down. This means we can demonstrate if our algorithms can use a low cost SSD to improve an existing HDD only system. The offline default parameters are described in Table 4. Our system setup assumes 4KB SSD pages and HDD pages, giving us 4KB system pages with no need to use multiple SSD pages for one system page.

7.3 Real Traces

We use two real traces from OLTP Applications running in financial institutions. They both include significantly larger numbers of requests in their workloads compared to our synthetic data sets. The data given is converted to our required format by using the Application Specific Unit as the file descriptor and page sizes of 4KB. Workload 2 is around the size our synthetic data while Workload 1 is approximately 90 times

Table 3: Default System Setup Parameters

Parameter	Value	Parameter	Value
HDD Size	100000 pages	SSD Size	2000 pages
HDD Seek Time	4.17ms	SSD Read Time	0.149ms
HDD Transfer Time	0.0313ms	SSD Write Time	0.198ms

Table 4: Default Offline Parameters

Parameter	Value	Parameter	Value
Segmentation Alg.	Join	Join Neighboorhood	1
Refinement cutoff	10	Join Initial Temp.	10000
Join Temp. Terminator	0.01	Join Inner Terminator	10

larger. Both traces spanned approximately 12 hours. For both sets of data, we have used the first half of the requests as the training workload for the offline stage and the second half as the online workload.

7.4 Algorithms tested

We tested three different variants of our drive assignment algorithms against four other rival systems. The algorithms as described below:

File Segmentation This is our drive assignment algorithm using the file segmentation algorithm described in Section 6.4.1.

Begin-End Segmentation This is our drive assignment algorithm using the begin-end segmentation algorithm described in Section 6.4.2.

Join Segmentation This is our drive assignment algorithm using the join segmentation algorithm described in Section 6.4.3.

Heat Assignment (rival) This is a simple intuitive rival algorithm to solve the drive assignment problem. It sorts pages according to access frequency (heat), and then fills up the SSD with the highest heat pages and assigns the remaining pages onto the HDD.

Random Assignment (rival) This algorithm fills up the SSD with randomly selected pages and assigns the remaining pages onto the HDD.

Dual SSD (rival) Two SSDs unbounded in size stripped at the page level (RAID 0). The SSDs uses the same placement algorithm as the hybrid systems. This represents an expensive rival system which has optimal concurrency.

Dual HDD (rival) Two HDDs unbounded in size stripped at the page level (RAID 0). The HDDs uses the same placement algorithm as the hybrid systems. This represents a cheaper rival system which has optimal concurrency.

By default the three variants of our algorithms use the refinement step described in Section 6.6 with refinement cutoff of 10. By default the heat and random assignment algorithms do not use the refinement step since they represent rival algorithms. However, in experiment 8.1 we test the effectiveness of the refinement step by testing the first five algorithms (including the 2 rival algorithms) with and without the refinement step.

By default the first five algorithms listed above assume an HDD large enough to fit our data, and an SSD which is approximately 10% of the total data size.

7.5 Computational hardware used

In the experimental results we report the execution time of our algorithms. We therefore describe the computational hardware used to conduct the experiments in our paper. The system has a Intel Core i7 860 CPU running at 2.8 GHz. The CPU has a 32KB L1 instruction cache, 32 KB L1 data cache, a 256 KB L2 cache and a 8192 KB L3 cache. The system has 8 GB of DDR3 RAM.

8 Experimental Results

In this section, we present results showing how our system performs under various conditions. We will check various choices of our primary algorithms, vary the kinds of data we use within the system and vary the system constraints. Using the results we will be able to determine the strengths and weaknesses of our proposed system and algorithms. The basic metrics we use to measure either the system or the algorithms are the average time per page (this is the total time divided by the number of pages read/written), the algorithm execution time (time spent in the offline stage assigning pages) and the average size of the requests in the workload.

All the synthetic workloads use the same set of files with the file sizes distributed according to the research by Agrawal et al. [1]. This is discussed in Section 7.2. Although the size distribution remains constant the access pattern changes across workloads.

The request sizes (in pages) for the default data sets are: Full file access - 12.38, Random Access - 1 and Segment Access - 2.87. These are the average request sizes for the different workloads in Experiments 8.1 and 8.2. For the other experiments the average request size is mentioned explicitly.

8.1 Algorithm Comparisons

In this experiment, we aim to test the performance of the various assignment algorithms described in Section 7.4. One of the key objectives of this experiment is to test the effectiveness of our proposed refinement step. Accordingly we test all assignment algorithms (3 proposed and 2 rival) with and without refinement and the two rival non-hybrid systems dual SSD and dual HDD.

Figures 5(a), 7(a) and 8(a) show the online time results of varying the algorithms used. The different algorithms are labeled on the x-axis. In order to aid in the analysis of the algorithms, the graphs are designed to show the proportion of concurrent access versus non-current single device access. This is done by showing the average time per page access divided into 3 sections.

Non-concurrent SSD shows the proportion of time spent on a single SSD with no concurrent access with the other device.

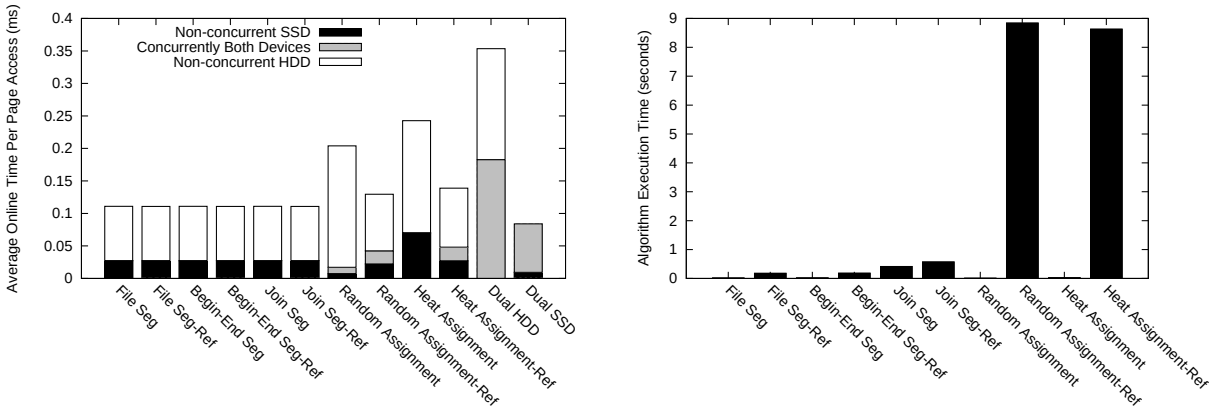
Concurrently Both Devices shows the proportion of time the algorithm spent concurrently accessing data from both devices. For the hybrid system this is SSD and HDD.

Non-concurrent HDD shows the proportion of time spent on a single HDD with no concurrent access with the other device.

It is important to note that the 3 sections of each bar do not individually give the actual average time spent on that device, but rather give a proportion of the time spent. However, the total of the three sections together does give the average online time per page access.

Figure 5(a) shows the online time results for full file access. These results show that the three proposed segmentation algorithms perform exactly the same. This is because in this case there is no variation in the start and end points of any request within a file. This means that begin-end and join segmentation can only create the same segments as file segmentation. Therefore the online performance of these algorithms is identical for file access.

We also note that the performance of our three algorithms are 3 times faster than the dual HDD and only 1.3 times worse than the dual SSD. The reason dual SSD does so well despite the fact it is full file access (longer sequential requests) is because 45% of the requests are still below the threshold (71 pages for these systems) for a request to be faster on a dual HDD or hybrid system than on a dual SSD system. For most



(a) Online Time Per Page

(b) Algorithm Execution Time

Figure 5: Vary Refinement Algorithms - File Access

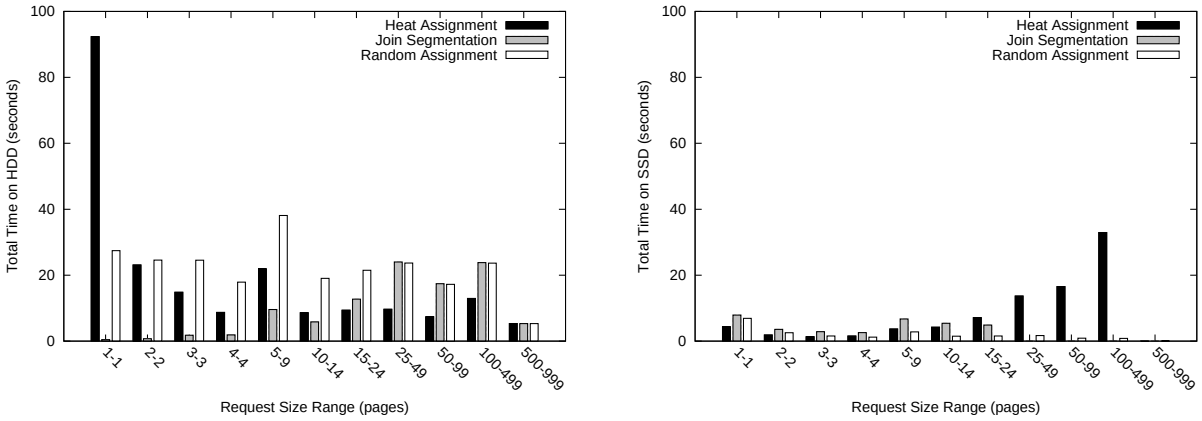
of these requests the difference in performance is much greater than for the remaining 55% which are larger than the threshold.

Our three proposed algorithms perform 54% better than heat assignment without refinement. In this case it is because heat assignment is unable to take advantage of the sequentiality of the data. For example, let us consider the following situation. An SSD that can fit only 10 pages, a workload consisting of one large file with a size of 10 pages and a heat of 20 and 10 small files with a size of 1 page and a heat of 19. Heat assignment will place the entire large file onto the SSD since each of its pages has higher heat than the pages of the smaller files. However, this results in heat assignment incurring 10×19 HDD seeks, since all the small files would be stored on the HDD. In contrast, our three proposed algorithms would place the 10 small files on the SSD and therefore only place the one large file on the HDD. Therefore, our three proposed algorithms would only incur 1×20 HDD seeks. Our algorithms are able to achieve this more optimal assignment by considering sequentiality in their heuristics and cost equations.

In order to show that this is the situation which is occurring we have provided a trace analysis for the full file access with heat and random assignment and join segmentation. This trace analysis is shown in Figures 6(a) and 6(b). The graphs show the total time spent on requests within a certain range on the HDD and SSD respectively. We use total time here rather than average time per page access so it can be seen how much the system actually gains or suffers within a certain request size range among the two device types separately. The results show clearly the additional time spent by heat assignment on the HDD for smaller requests (compared to join segmentation) which indicates it has not assigned many of them to the SSD. In place of those pages it has assigned pages from the 15-24, 25-49, 50-99 and 100-499 pages request ranges. This gives an improvement on the HDD but performs much worse on the SSD. This is because for this system the SSD is slower at sequential access than the HDD when requests are longer than 35 pages in length.

The result that random assignment without refinement outperforms heat assignment without refinement seems counter-intuitive. However, it can be explained by using a similar explanation as for our three proposed algorithms outperforming heat assignment. Let us consider the same situation as the previous example. In that example random assignment will likely place some of the 10 small files onto the SSD and part of the one large file on the SSD. Let us suppose it places 4 of the small files on the SSD and 6 on the HDD. Then it would incur 1×20 + 6×19 HDD seeks. This is still less than the 10×19 HDD seeks incurred from using the heat assignment. To show that this is the case we refer again to Figures 6(a) and 6(b). For random access we can see that for each of the request ranges it has assigned some of the pages to the SSD. The large number pages of the 1-1 request range placed on the SSD greatly increased the performance of random assignment compared to heat assignment. Additionally since it did not assign many pages from the larger requests it did not suffer from the poor sequential access of the SSD in this system.

We notice a significant improvement to the random and heat assignment algorithms after using refinement. This is because the refinement stage calculates the cost of certain pages (which includes sequentiality and differences in cost of read and write access). In this way it tries to fix some mistakes made by random



(a) Request Ranges Analysis for HDD accesses

(b) Request Ranges Analysis for SSD accesses

Figure 6: Request Ranges Analysis for File Access

and heat placement. However we can see that it is not able to reach the same level of performance as the other three algorithms. Our refinement algorithm is a simple hill climber and is unable to make fundamental changes to the assignment which is necessary to escape a local optimum such as these.

We can see in Figure 5(a) that with refinement the heat and random assignment algorithms have a small amount of concurrency while the concurrency used by our algorithms is negligible. This is because as a general rule it is better to first take advantage of the SSD to avoid HDD seeks on small requests than it is to use the SSD for concurrency (concurrency is a smaller improvement and uses more SSD pages). Our segmentation algorithms take this into account by using a cost-based analysis. Heat and random assignment do not consider this, as a result refinement may introduce concurrency to achieve a local optimum.

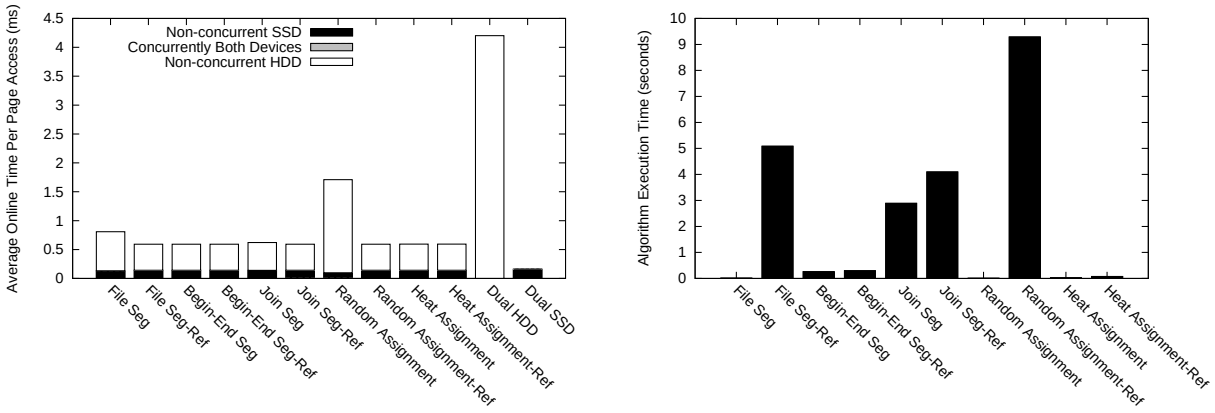
We notice for full file access in Figure 5(a) that by adding refinement to our three segmentation algorithms we get negligible improvement in performance. This is because the initial segmentation algorithms produce an assignment that is very close to the optimal assignment since the requests are not complicated.

In Figure 5(b) we can see the total execution time of all the algorithms are less than 10 seconds. This is quite acceptable given the assignment is done offline. The results show that heat and random assignment with refinement takes 12 times longer than our segmentation algorithms with refinement. This is reflected by the significant increase in execution time when refinement was used. The reason is our segmentation algorithms get very close to the optimal assignment which leaves little work left for the refinement step. In contrast heat and random assignment require refinement to do a lot more work to make the assignment close to the optimum.

Figure 7(a) shows the online performance for a fully random access workload. Similar to full file access our three algorithms with refinement have approximately the same performance. However we see that for random access both file and join segmentation without refinement do not perform as well. For file segmentation this is because the size of requests is not taken into consideration. For join segmentation it is because the join algorithm erroneously believes that joining some of the segments will better represent the requests. This happens when it considers many adjoining pages of similar heat. In this case join segmentation will believe that it can gain extra performance by considering the adjoining pages as sequential access and thus gaining concurrency. Fortunately this error is easily remedied by refinement. Begin-end segmentation can not make this mistake and thus requires virtually no refinement (as shown by the difference between begin-end with and without refinement in Figure 7(b)).

For random access, heat assignment performs as well as our algorithms even without refinement(Figure 7(a)). This is because in this workload there is no sequentiality and hence heat assignment achieves close to optimal assignment. It is not exact since it still does not take into account the asymmetric read and write costs of the SSD.

Figure 7(b) shows the execution time of all the assignment algorithms are less than 10 seconds. We can see that the join segment needs to do more work than for full file access (Figure 5(b)). This is because join segment requires a lot of analysis for the large number different requests. File segmentation requires a long

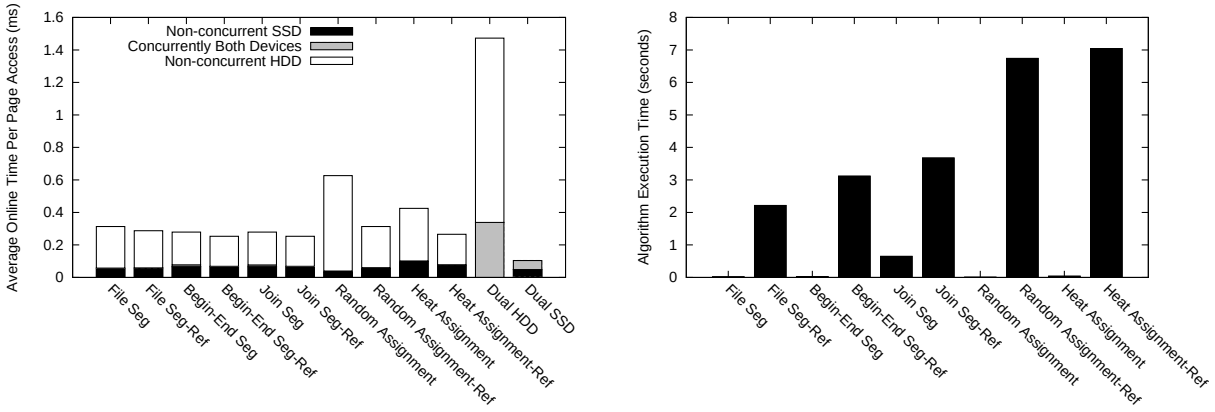


(a) Online Time Per Page

(b) Algorithm Execution Time

Figure 7: Vary Refinement Algorithms - Random Access

refinement step because it is such a coarse algorithm which incorrectly places many pages by assuming all requests span the entire file.



(a) Online Time Per Page

(b) Algorithm Execution Time

Figure 8: Vary Refinement Algorithms - Segment Access

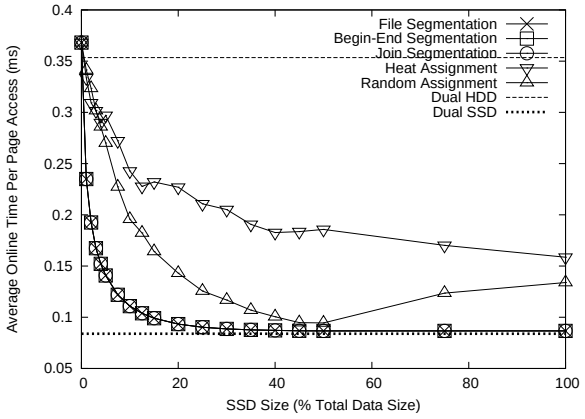
The results for segment access in Figure 8(a) shows similar trends to full file access (Figure 5(a)). The primary difference is the slight variance between the performance of our algorithms. For file and join segmentation the performance differences are for the same reasons as random access described above. Begin-end segmentation now performs worse without refinement because the assumption of the disjoint segments representing requests no longer applies. This is because there is now overlap between the different segments in the workload.

8.2 Varying SSD Capacity

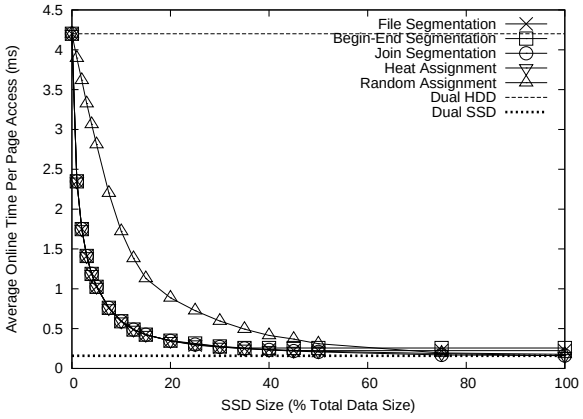
In this experiment, we compare the assignment algorithms and dual SSD and dual HDD as the capacity of SSD is varied. We use the default settings for all the algorithms as outlined in Section 7.4. We vary the SSD size between 0 and the total data size (20, 000 pages), since once the total data size is reached, adding more SSD makes no difference.

Figures 9(a), 9(b) and 9(c) show the results of this experiment. SSD size is the percentage of the total data size on the x-axis. The previous experiment corresponds to the results where the SSD is at 10% of the

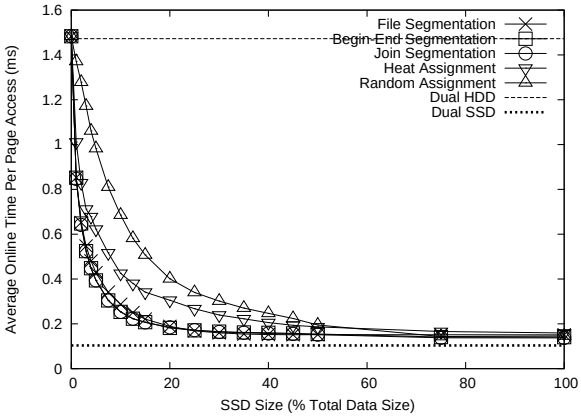
total data size. Full file access has an average request size of 12.38 pages. While random access and segment access have average request sizes of 1 and 2.87 pages respectively.



(a) File Access



(b) Random Access



(c) Segment Access

Figure 9: Vary SSD size

For file access (Figure 9(a)) we can see that our three proposed assignment algorithms perform almost identically regardless of the SSD size. This limited difference in performance is due to the simplicity of the data set as described for file access in Experiment 8.1.

For heat assignment however we notice that it is consistently worse than random assignment. This is again for the same reason as for file access results in Experiment 8.1. The performance of random assignment improves with increasing SSD size until around 50% data set size but then starts to degrade in performance with increasing SSD size. The reason is as more SSD becomes available random assignment places an increasing number of small files and small portions of large files on the SSD. This improves performance. However after the SSD size reaches beyond 50% data set size too much of the large files get placed on the SSD which becomes sub-optimal since it is better to place them on the HDD which has high transfer rates.

For our algorithms they are able to perform within 4% of the dual SSD when the SSD size is over 40% of the total data size. This is because much of the data is sequential (average request size 12.38 pages), which alleviates the disadvantages of using the HDD. However our most notable improvements are achieved when the SSD is smaller since we are able to intelligently place the best pages on the SSD. This is due to the skewed nature of the data in which 50% of the requests are given to 10% of the files.

Figure 9(b) shows the results for random access. As expected heat assignment is now able to perform at the same level as our algorithms since there is no sequential access. However we now need a lot more of the SSD (50%) until we perform within 31% of the dual SSD. This is because for every request the dual SSD is able to perform better than our HDD and the same as our single SSD so we need to be able to place every

page from the working set in order to equal the performance.

For segment access (Figure 9(c)), similar to Experiment 8.1, we see a mixture of the results of file and random access. That is our algorithms have only marginal differences in their performance while heat and random assignment perform worse for small SSD sizes. However we notice that now heat assignment outperforms random placement unlike for file access. This is because the requests are on average 20% of the file size which brings the request size much closer to random access (better for heat assignment). We are only able to get within 32% of the dual SSD's performance since these smaller request sizes are more favourable to dual SSD.

8.3 Varying Request Sizes

In this experiment, we compare the assignment algorithms and dual SSD and dual HDD as the size of request segments were varied. We use only segment access and vary the size of the requests according to the mean in the gaussian distribution of sizes. We vary the Segment Size Dist. = (g, x, .15) with x between 0 and 1 which is a percentage of the files' total size.

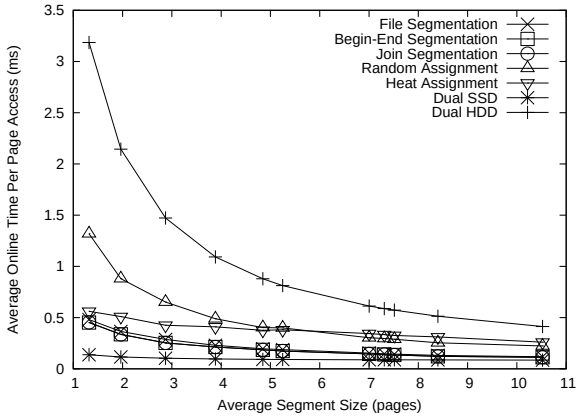


Figure 10: Vary Request Size

We see the results to this experiment in Figure 10. As the average request size increases, the performance of dual HDD improves considerably. This is because as the request size increases the high seek cost of the HDD gets offset by the high transfer rate of the HDD.

We note that for our systems we show a small improvement as the segment size increases while heat assignment gains only a marginal improvement. This is because our systems take the sequentiality of data into account, while heat assignment does not. As the average segment size changes the data set moves from almost fully random access to almost fully file access and so the results at either end reflect the results given in Figures 5(a) and 7(a).

To test what happens when the average request size increases beyond 11 pages (the largest used in this experiment) we conducted some additional experiments with larger file sizes. The results showed that when the average request sizes crossed a certain point the dual HDD became the best performer. We do not show the graphs for those experimental results since the file sizes used in that experiment are not as representative of real file sizes as those of our default synthetic data set.

8.4 Varying Access Style

In these experiments, we aim to discover which system performs best under mixed access styles. This has similarities to Experiment 8.3 except that now the request sizes have up to three primary spikes in sizes, one for each type of access used. We provide an even spread of different access files and vary how likely we are to make a request of that type. Among our algorithms we only include the results of join segmentation with refinement since we tested our other two algorithms and found the results were all very similar. We compare the results of dual SSD, dual HDD and heat assignment against join segmentation in each of the Tables 5,

6 and 7. Tables 5, 6 and 7 give the total online time of the given system divided by the total time of join segmentation. In this way we show how much better or worse this system was compared to our own. For example the .57 in the box for 0 file access and 1 random access in Table 5 means that the dual SSD executed the workload in .57 of the time it took join segmentation hybrid system to complete it. Table 8 shows us the average request size of the different workloads for a reference.

The tables show varying the probability of different access styles according to $0 \leq x \leq 1$, $0 \leq y \leq (1-x)$ and $z = 1 - x - y$ with x, y and z being the probability of file, random and segment access respectively.

Table 5: Vary Access Style - $\frac{Dual\ SSD\ online\ time}{Join\ Segmentation\ online\ time}$

Random Access Probability											
1	0.57										
0.9	0.43	0.57									
0.8	0.41	0.46	0.61								
0.7	0.41	0.44	0.51	0.69							
0.6	0.42	0.44	0.49	0.58	0.74						
0.5	0.43	0.45	0.49	0.56	0.64	0.79					
0.4	0.44	0.46	0.50	0.56	0.62	0.69	0.83				
0.3	0.47	0.48	0.51	0.57	0.62	0.67	0.74	0.88			
0.2	0.49	0.50	0.53	0.59	0.64	0.68	0.73	0.79	0.93		
0.1	0.52	0.53	0.56	0.62	0.67	0.71	0.75	0.80	0.86	0.98	
0	0.69	0.67	0.66	0.73	0.78	0.82	0.86	0.90	0.94	0.99	1.00
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0

File Access Probability

Table 6: Vary Access Style - $\frac{Dual\ HDD\ online\ time}{Join\ Segmentation\ online\ time}$

Random Access Probability											
1	15.13										
0.9	10.53	9.47									
0.8	9.38	7.32	7.90								
0.7	8.75	6.68	6.30	6.82							
0.6	8.32	6.38	5.85	5.60	6.12						
0.5	8.05	6.23	5.63	5.26	5.15	5.60					
0.4	7.89	6.11	5.53	5.10	4.89	4.80	5.22				
0.3	7.86	6.07	5.50	5.06	4.80	4.61	4.56	4.95			
0.2	7.87	6.15	5.54	5.08	4.80	4.58	4.43	4.40	4.70		
0.1	8.09	6.30	5.73	5.24	4.92	4.69	4.49	4.37	4.30	4.54	
0	10.21	7.68	6.51	5.99	5.59	5.29	5.04	4.85	4.67	4.54	4.22
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0

File Access Probability

In Table 5 we see the results of how the dual SSD performs compared to our system with varying types of access. The first thing we can notice is that at worst join segmentation is 2.5 worse than the dual SSD. The best we can do is in the full file access where we perform at the same level as the dual SSD. At this best point the average request size given in Table 8 is 12.06 pages. Join segmentation has only 10% of the total data size as SSD space. This makes a considerable saving since it only needs a single SSD 10% the size of the data, plus a single HDD large enough for the remaining 90%.

From Table 6 we can see clearly that for the file size and access distributions used here we are always able to outperform the dual HDD by between 4 and 15 times. Predictably as the request sizes get larger due to more segment and file access, the dual HDD performs better. The reason it never outperforms join segmentation hybrid is because the request sizes are never large and often enough, to make the superior performance of the HDD at long sequential requests the dominant factor.

In Table 7 we can see that heat assignment is unable to outperform our system. For pure random access it has the same results and while there is no file access it performs reasonably. However we notice it performs

Table 7: Vary Access Style - $\frac{Heat\ Assignment\ online\ time}{Join\ Segmentation\ online\ time}$

Random Access Probability											
1	1.00										
0.9	1.23	2.32									
0.8	1.30	2.13	2.82								
0.7	1.36	2.11	2.59	2.90							
0.6	1.39	2.12	2.50	2.73	2.87						
0.5	1.40	2.13	2.55	2.73	2.71	2.82					
0.4	1.41	2.08	2.52	2.70	2.76	2.67	2.74				
0.3	1.44	2.02	2.46	2.70	2.78	2.72	2.63	2.67			
0.2	1.49	1.94	2.40	2.67	2.76	2.72	2.68	2.57	2.61		
0.1	1.46	1.82	2.33	2.65	2.72	2.69	2.66	2.62	2.51	2.48	
0	1.49	1.85	2.30	2.74	2.76	2.86	2.77	2.70	2.57	2.50	2.13
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0

File Access Probability

Table 8: Vary Access Style - Average Request Size

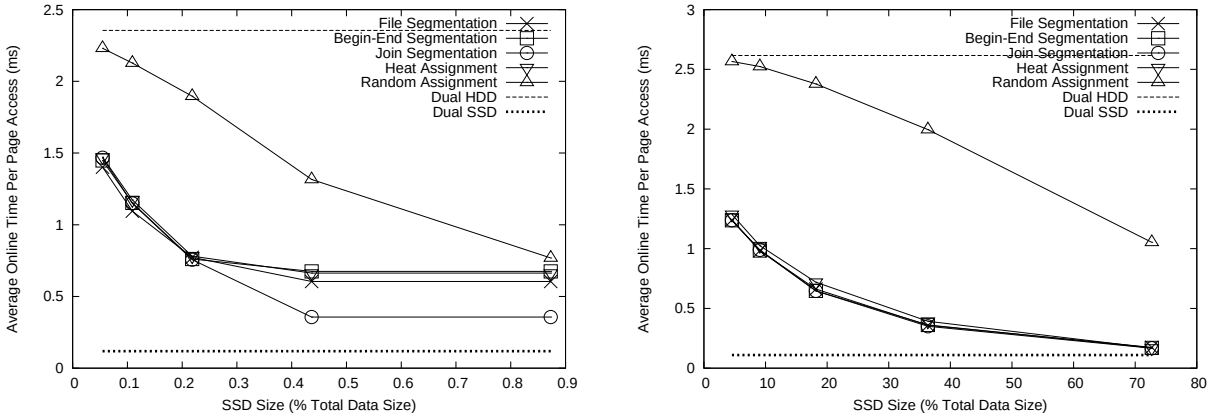
Random Access Probability											
1	1.00										
0.9	1.16	2.21									
0.8	1.34	2.38	3.16								
0.7	1.53	2.55	3.32	4.40							
0.6	1.71	2.74	3.49	4.57	5.53						
0.5	1.89	2.92	3.69	4.74	5.69	6.60					
0.4	2.07	3.10	3.86	4.93	5.86	6.77	7.67				
0.3	2.25	3.28	4.04	5.11	6.06	6.94	7.83	8.69			
0.2	2.42	3.46	4.23	5.29	6.24	7.13	8.00	8.85	9.80		
0.1	2.59	3.63	4.40	5.48	6.42	7.31	8.20	9.02	9.97	10.90	
0	2.76	3.80	4.57	5.65	6.60	7.49	8.38	9.22	10.14	11.06	12.06
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0

File Access Probability

worst when there is an even mixture of file and random access. This is because it is not taking sequentiality into account, as explained in Experiment 8.1.

8.5 Real Traces

Using the real traces, we aim to show that our systems also performs well for real workloads rather than just synthetic ones. In these experiments, we vary the SSD size between 0 and close to the working set size. The working set sizes are 8796 and 3066 pages for workload 1 and workload 2 respectively. This is compared to their total data sizes of 916458 pages for workload 1 and 4402 pages for workload 2. Therefore the largest SSD sizes used for workload 1 is 8000 pages and workload 2 is 3200 pages. The x-axis of the graphs show SSD size used as a percentage of total data size. The average request sizes are 1.79 and 1.61 pages for Workload 1 and 2 respectively. The algorithms and systems tested are the default ones described in Section 7.4.



(a) Workload 1

(b) Workload 2

Figure 11: Real Traces

We can see in Figure 11(b) that for workload 2 the relative performances of our algorithms and heat assignment are very similar. This is because the average request size for workload 2 is 1.61 pages and the request pattern is a skewed distribution which means it performs similar to our random access synthetic data set. For that data set heat assignment performs very well since the fact that it does not take sequentiality into consideration is less important.

In contrast, for workload 1 (Figure 11(a)), join segmentation outperforms the other proposed and rival algorithms. Begin-end segmentation performs quite badly under this workload compared to the other algorithms. The reason is that begin-end segmentation only uses around 1/4 of the total SSD space available, the reason being due to the discrepancy between the training workload and the actual online testing workload. In the training workload, the working set is only 3265 pages leaving the remaining pages in the data set completely unused. Unused data has a cost benefit of 0 to be placed on the flash and so is left on the HDD. However, we note that join segmentation has an odd reaction to this unused space and performs more than 40% better than our next best algorithm (file segmentation). It is still 3 times worse than the dual SSD though. It is able to join small heavily requested segments with large unused segments. This means that join segmentation will be able to place unused pages onto the SSD, while begin-end segmentation will suffer from overtraining and leave the additional pages off the SSD. In this case, it allows join segmentation to utilise the entire SSD. The extra pages placed on the SSD includes some of the heavily-used pages in the actual testing workload (but unused in the training workload) which results in the significantly improved performance. The refinement stage is unable to fix these irregularities between our algorithms because the refinement stage also uses the non-indicative training workload to make the placement decisions.

For both workloads in Figure 8.5 the algorithm execution time was worst for file segmentation which took under 12 minutes. While our other algorithms always completed execution within 3 minutes. This is fast for an offline algorithm which produces such good performance.

8.6 Summary of Results

Through the experiments we were able to see the advantages and disadvantages of the proposed systems and their rivals. The strengths and weaknesses of each are given below.

Dual SSD is the fastest system for the styles of workloads which we tested. It is guaranteed to be fastest on any single request that is less than 71 pages (for the system performance parameters we have used). However it requires that you have enough SSD storage space to fit the entire data set. This is generally an unreasonable assumption since SSDs are expensive especially when dealing with large storage capacities.

Dual HDD is the slowest of the systems we tested. It was normally more than twice as slow as its nearest competitor. However it is the cheapest and simplest option.

Join segmentation is the best of our proposed algorithms. It was able to perform well in all the experiments with only a small overhead in algorithm execution time. It requires only a small proportion (10%) of the total data size to be available in SSD to be a close competitor of the dual SSD system. Like our other hybrid algorithms it requires an offline stage to analyse data and assign pages to the SSD.

Begin-end segmentation is the next best of our algorithms. It performs well in most situations however it can be confused by some complex data sets and suffers in these when the amount of SSD is increased. It achieves its good performance with less algorithm execution time than join segmentation.

File segmentation is the worst of our proposed algorithms. It performs well in many situations but often requires extensive amounts of refinement which causes the algorithm execution time to increase significantly. It is the simplest initial algorithm and requires less detailed workload information to complete.

Refinement is a default step for our three proposed algorithms which thoroughly improves performance. However it is often the most expensive step of the offline stage except for join segmentation which can sometimes spend longer on the initial segmentation step. Refinement unfortunately is a blind hill climber and so is unable to escape local optima. It requires a good initial assignment to work from. The reason we do not propose a more complex algorithm which can potential escape the local optima is that the computation complexity would be much higher.

Random assignment is a blind alternative to our algorithms. It performs the worst in many situations especially when the SSD is small. It does not require any workload analysis and so requires virtually no algorithm execution time or overheads for storing workload information.

Heat assignment is a simple heuristic rival for our algorithms. It performs as well as our algorithms with less overheads only for random access. However for even simple sequential access it suffers from performance degradation especially when the SSD is limited in size. Its algorithm execution time is never expensive and it requires minimal workload information.

9 Conclusions

In this research, we aimed to optimise the I/O performance of a system that uses a large HDD and limited size SSD in tandem to store data. To further improve performance, we allowed the two devices to be accessed concurrently during each request. We focused our research on the drive assignment problem. We proved the problem was NP-complete and therefore proposed heuristic solutions to solve the problem. Our approach was to first make a rough and quick assignment which took us close to a good solution and then a refinement stage which brought us to a near optimal solution. Experimental results showed this approach was very effective at improving I/O performance.

We compared our algorithms against dual SSD and dual HDD systems for both synthetic and real data sets. For the synthetic data set, we outperform the dual HDD system in all cases and achieved close performance to the dual SSD system. We have also shown that, with just 10% of the total data size as additional

SSD storage, we can perform as well as the dual SSD system (which had unlimited size restrictions) in some situations, while outperforming the dual HDD by a factor of between 3 and 16 in all cases.

For the real datasets, we saw our system outperform the dual HDD system but it performed worse than the unlimited size dual SSD system. This was because the dual SSD system can take advantage of the fast random access of the SSD on both drives concurrently whereas in our hybrid system there is only one SSD. The results showed that our system is able to offer an effective middle ground between the slow but cheap dual HDD system compared to the fast but much more expensive dual SSD system.

When we compared the performance of the different variants of our algorithms, we saw that our refinement algorithm was able to bring any of the segmentation algorithms' placements to almost the same performance level. This indicated that the refinement was a key step to approaching the best performance. However, the segmentation algorithms varied much more in how close they brought the solution to the best performance level which affected the amount of work that the refinement algorithms need to perform. This caused the time taken offline to assign pages to degrade up to a factor of 4.

We show that a heat based assignment algorithm is able to perform well in most situations. However when dealing with workloads which include long sequential accesses the performance is not maintained. In fact, since heat assignment does not consider the sequentiality of data it can make decisions which detriment its performance rather than improve it.

There can be many useful extensions to our proposed work. At present, the system makes drive assignment decisions offline to avoid both online CPU and I/O overheads. Since the offline drive assignment is not disruptive to users, we can use a more complex algorithm which is more likely to bring us close to a global optimal drive assignment. Other directions for future research are the online drive assignment and placement problems. This will enable the system to better react to changes in workload patterns and also handle file growth in a more optimal way.

Redundancy is not currently being included in the system. We predict there would be situations where storing multiple copies of the same data across the two devices could be either beneficial to the performance or required by system administrators to achieve fault tolerance. This research may heavily rely on existing RAID redundancy techniques applied to the new system or specifically investigate new methods of redundancy, taking into account the varied abilities of the two device types.

The system has been designed around the use of only two devices, however designing a system which used multiple devices of each type either running in a RAID-like setup or independently, would extend the research by ensuring that it would be applicable to many of the larger server systems. Alternatively, extending this research to the use of any number of different performance devices could be beneficial to systems which need to grow slowly over time, as different storage technologies can be added as required.

The system relies on the collection of workload statistics for use in the offline drive assignment. It is important that this workload is collected in an efficient manner to reduce the disturbance on the online system. We envisage an efficient system would use in RAM data structures and sequential writes to a log file when RAM is full. This is therefore an important direction of future work.

Finally, a significant area of future work is the placement of data within each device. This would extend the research to implementing both a flash file system and an HDD file system jointly with the drive assignment algorithm. Alternatively, a single file system could be designed to place pages across both devices with an in-built drive assignment.

References

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage*, 3(3), 2007.

[2] Ron Arnan, Eitan Bachmat, Tao Kai Lam, and Ruben Michel. Dynamic data reallocation in disk arrays. *ACM Transactions on Storage*, 3(1):2, 2007.

[3] Mary Baker, Satoshi Asami, Etienne Deprit, John K. Ousterhout, and Margo I. Seltzer. Non-volatile memory for fast, reliable file systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 10–22, 1992.

[4] Luc Bouganim, Björn þór Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[5] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 217–228, 2009.

[6] Li-Pin Chang and Tei-Wei Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *ACM Transactions on Storage (TOS)*, 1(4):381–418, 2005.

[7] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC '07: Proceedings of the 44th Annual Conference on Design Automation*, pages 212–217, New York, NY, USA, 2007. ACM.

[8] Hyun-Jin Choi, Seung Ho Lim, and Kyu Ho Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage (TOS)*, 4(4), 2009.

[9] Ivan Dramaliev and Tara Madhyastha. Optimizing probe-based storage. In *FAST 03: 2nd USENIX Conference on File and Storage Technologies*, pages 103–114, 2003.

[10] K.J. Gabriel. Microelectromechanical systems (MEMS). *Aerospace Conference, 1997. Proceedings., IEEE*, 3:9–43, 1997.

[11] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[12] John A. Garrison and A. L. Narasimha Reddy. Umbrella file system: Storage management across heterogeneous devices. *ACM Transactions on Storage (TOS)*, 5(1), 2009.

[13] getprice.com. Get Price 1TB Baracuda 7200.12 Price. http://www.getprice.com.au/Seagate-Barracuda-7200-12-3-5-1TB-HDD-SATAII-7200rpm-32MB-Cache-ST31000528AS-Gpnc_58--36904358.htm, Last viewed: June 2010.

[14] David D. Grossman and Harvey F. Silverman. Placement of records on a secondary storage device to minimize access time. *Journal of the ACM*, 20(3):429–438, 1973.

[15] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, 2009.

[16] Bo Hong, Feng Wang, Scott A. Brandt, Darrell D. E. Long, and S. J. Thomas J. E. Schwarz. Using MEMS-based storage in computer systems—MEMS storage architectures. *ACM Transactions on Storage*, 2(1):1–21, 2006.

[17] Stratis Viglas Ioannis Koltsidas. Flashing up the storage layer. In *34th International Conference on Very Large Databases (VLDB)*, 2008.

[18] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 161–170, New York, NY, USA, 2006. ACM.

[19] Jesung Kim, Jong Min Kim, S. H. Soh, Sang Lyui Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[20] Jongmin Lee, Sunghoon Kim, Hunki Kwon, Choulseung Hyun, Seongjun Ahn, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Block recycling schemes and their cost-based optimization in nand flash memory based storage system. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, pages 174–182, New York, NY, USA, 2007. ACM.

[21] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions Embedded Computer Systems*, 6(3):18, 2007.

[22] Sukhan Lee, Chungwoo Kim, Sunghwan Jung, In-Sup Song, and Yong Chul Cho. MEMS for IT applications. *Proceedings of 2001 International Symposium on Micromechatronics and Human Science, 2001*, 1:17–23, 2001.

[23] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[24] Adam Leventhal. Flash storage memory. *Communications of ACM*, 51(7):47–51, 2008.

[25] Seung-Ho Lim, Chul Lee, and Kyu-Ho Park. Hashing directory scheme for NAND flash file system. In *Advanced Communication Technology, The 9th International Conference on*, volume 1, pages 273–276, 2007.

[26] Seung-Ho Lim and Kyu-Ho Park. An efficient NAND flash file system for flash memory storage. *Computers, IEEE Transactions on*, 55:906–912, July 2006.

[27] Yan Liu, Chang-Sheng Xie, and Huai-Yang Li. PMSH: a new algorithm for RAID data migration based on stripe unit heat. In *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, volume 6, pages 3421–3427, 2005.

[28] Jeanna Matthews, Sanjeev N. Trika, Debra Hensgen, Rick Coulson, and Knut Grimsrud. Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage (TOS)*, 4(2), 2008.

[29] David S. Johnson Michael R. Garey. *Computers and Intractability. A Guide to the Theory of NP-Completeness*, chapter A6 Mathematical Programming, page 247. W.H. Freeman and Company, 1979.

[30] Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. HeRMES: high-performance reliable MRAM-enabled storage. In *Hot Topics in Operating Systems*, pages 95–99, 2001.

[31] H. Payer, M. Sanvido, Z. Bandic, and C. Kirsch. Combo drive: Optimizing cost and performance in a heterogeneous storage device. In *Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH)*, 2009.

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

[33] Seagate. Seagate 1TB Barracuda 7200.12 Specifications. http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_7200_12.pdf, 2010. Last viewed: June.

[34] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[35] SuperTalent. Super Talent MasterDrive EX2 Specifications. http://www.supertalent.com/products/ssd_category_detail.php?type=MasterDrive, 2010. Last viewed: June.

[36] UMassRepository. Storage system workloads. http://traces.cs.umass.edu/index.php/Storage/Storage, Last viewed: June 2009.

[37] Mustafa Uysal, Arif Merchant, and Guillermo A. Alvarez. Using MEMS-based storage in disk arrays. In *FAST 03: 2nd USENIX Conference on File and Storage Technologies*, pages 89–101, 2003.

[38] An-I Andy Wang, Geoffrey H. Kuenning, Peter L. Reiher, and Gerald J. Popek. The *conquest* file system: Better performance through a disk/persistent-ram hybrid design. *ACM Transactions on Storage (TOS)*, 2(3):309–348, 2006.

[39] Wenguang Wang, Yanping Zhao, and Rick Bunt. HyLog: A high performance approach to managing disk layout. In *FAST 04: 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, 2004.

[40] www.mwave.com.au. OCZ Vertex 2 Price. http://www.mwave.com.au/sku-22140387-OCZ_Vertex_2_120GB_SATA_II_3_5%22_Solid_State_Drive_0_1ms_Seek_Time_Read_285M, Last viewed: August 2011.

[41] www.mwave.com.au. OCZ Vertex 3 Price. http://www.mwave.com.au/sku-22140500-FREE_SHIPPING_OCZ_Vertex_3_2_5%22_Solid_State_Drive_240GB_SATA3_Read_550MB_s_, Last viewed: August 2011.

[42] www.ocztechnology.com. OCZ Vertex 2 Specifications. http://www.ocztechnology.com/ocz-vertex-2-sata-ii-2-5-ssd.html, Last viewed: August 2011.

[43] www.ocztechnology.com. OCZ Vertex 3 Specifications. http://www.ocztechnology.com/ocz-vertex-3-sata-iii-2-5-ssd.html, Last viewed: August 2011.

[44] Hailing Yu, Divy Agrawal, and Amr El Abbadi. Towards optimal I/O scheduling for MEMS-based storage. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems & Technologies*, 2003.

[45] Zhihui Zhang and Kanad Ghose. yFS: A journaling file system design for handling large data sets with reduced seeking. In *FAST 03: 2nd USENIX Conference on File and Storage Technologies*, pages 59–72, 2003.

A NP-complete Proof

In MCK, the set of items is sub-divided into disjoint classes from which we choose exactly one item from each to place in the knapsack while maximising cost. MCK is formally defined as below [29].

Definition Let S be the maximum size of the knapsack. Let there be a set of k disjoint sets $N_1, N_2, \ldots, N_k$ with each item $j \in N_i$ having a profit $p_{ij} > 0$, and size $s_{ij} > 0$. Let $x_{ij} \in \{0, 1\}$ be used to decide whether the item $j \in N_i$ has been selected.

$$Maximise : \sum_{i=1}^{k} \sum_{j \in N_i} p_{ij} x_{ij} \tag{25}$$

$$Such\ that : \sum_{i=1}^{k} \sum_{j \in N_i} s_{ij} x_{ij} \leq S \tag{26}$$

$$(\forall i : 1 \leq i \leq k) \sum_{j \in N_i} x_{ij} = 1 \tag{27}$$

A.1 Mapping

From Definition A, we can see some similarities between MCK and our problem; we have a limited sized SSD and the pages we place in the SSD have a cost and size. If we consider the items as pages, we cannot have a set cost as required by the MCK. Also, all the items would have the same size of 1. Instead, we use the set of all possible partitions as our set of items. An example of the possible partitions for two files is given in Figure 12. Here we can see there are 2^n possible partitions for each file where n is the number of pages in the file. The set of all possible partitions has certain items which cannot both be selected eg. we cannot use both the first and last possible partition for file 0. This mutual exclusion of possible partitions gives us our disjoint classes. As it happens, this can always be described as the items of a disjoint set N_i mapping to the possible partitions of $file_i$. We know we can calculate the cost of each of these possible partitions exactly since there is no interaction with anything in any other file. This means the cost is fixed for that

partitioning regardless of the partitioning choices of other files. Also, each possible partition has a specific size, namely the number of pages in the SSD partition. This shows us how to map MCK into our problem. This is summarised by Table 9.

Table 9: Mapping Summary

MCK equivalent	Initial Drive Assignment
S	SSD size
k	number of files
N_i	possible partitions for $file_i$
p_{ij}	cost of $partitioning_j$ in $file_i$
s_{ij}	size of $partitioning_j$ in $file_i$
x_{ij}	picking a $partitioning_j$ in $file_i$
Equation 25	optimising drive assignment
Equation 26	limited size SSD
Equation 27	picking a partition for each file

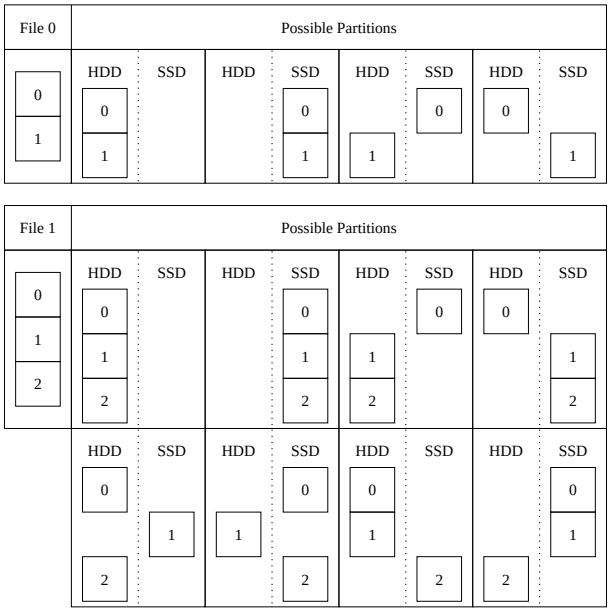


Figure 12: Possible Partitions

A.2 NP-complete Proof

Theorem A.1 *The drive assignment problem in Equation 4 is NP-complete.*

Proof *Let f be a map from MCK (Definition A) into the drive assignment problem such that:*
the set of items N_i maps to the set of possible partitions for $file_i$ using the identity map,
$f(p_{ij}) = -p_{ij} + max(cost) + 1 = cost_{ij}$,
$f(s_{ij}) = s_{ij} - 1 = size_{ij}$,
$f(S) = S - k = SSDsize$,
the maximisation Equation 25 maps to the minimisation of the cost Equation 5,
the size restriction Condition 26 maps to the restriction of SSD size,
and the choice restriction condition 27 maps to the need to choose exactly one partition to represent each file.
From this, we have $(\forall i, j \in \mathbb{Z}^+ \leq k)$ N_i and N_j are disjoint since $file_i$ and $file_j$ are independent and share

no pages,
$p_{ij} > 0$ *since* $max(cost) - cost_{ij} + 1 > 0$
and $s_{ij} > 0$ *since* $size_{ij} + 1 > 0$
Hence, f is a polynomial mapping from the drive assignment problem into MCK, using only the identity map and linear transformations. Finally, checking the solution to the assignment problem can be done in polynomial time since Equations 25, 26 and 27 can all be computed in polynomial time. Therefore the drive assignment problem is NP-complete.

In the proof, we add a single unit of weight to all the sizes of the partitions in order to satisfy the condition that $s_{ij} > 0$. What this means is that even if we choose a partition with no files on the SSD, we will incur a size cost of one. To counter this, we increase the size allowed by k. This extra size is taken up exactly by the extra size added to each of our partition sizes since we must make exactly k choices (one for each class/file). For the costs, to ensure that $p_{ij} > 0$, we need to convert them from differences into positive profits. This is accomplished by taking our cost from the $max(cost) + 1$. This means our final maximum does not represent the max cost difference but instead the max profit (plus the extra k we added to make it positive). However, this is not important since at the offline stage, we are not interested in the actual cost of the optimal drive assignment but rather its composition. ∎