Hash Systems for Single Disk Allocation

Chris Riley, Christian Scheideler, Jonathan Shapiro
SRL Technical Report SRL2003-01
Department of Computer Science
Johns Hopkins University
{chrisr,scheideler,shap}@cs.jhu.edu

Abstract

Single- and multiple-choice hashing strategies have long been studied by the theoretical computer science community. In this paper we consider a new application of these hashing strategies, that of block placement on a single hard disk drive. We present a new method of viewing this placement process motivated by recent systems research, and overview a system based on these observations, one which breaks the traditional direct map to blocks on the physical hard drive and uses instead a very large logical address space and a randomized mapping to the physical level. We analyze hashing strategies for their practicality in handling this new mapping, targeting the conflicting goals of reducing collisions and reducing expected block access time.

1 Introduction

Current disk file systems use complex, deterministic placement strategies to decide the location of file blocks on the disk. Conceptually, we may view this as a two-phase process in which the file system places blocks into a "file system block address space" (FSBA), and a separate placement policy maps this space into the logical block address (LBA) space of the disk. Because of latencies resulting from the mechanics of disk drives, mapping sequential file blocks to sequential locations of the disk has significant performance advantages. As a result, only direct (1:1) mappings between the FSBA and LBA spaces have been attempted in existing file system designs. The resulting placement strategies lead to fragmentation, poor file system aging, and complex sequentially dependent update strategies that must balance achieved placement locality against file system recoverability.

If the direct map requirement can be broken, many of these problems can be eliminated. In particular, if a sufficiently large FSBA space can be mapped by a repeatable randomized method to the disk LBA space, responsibility for placement can be offloaded to the disk subsystem and entirely new design strategies for file systems become possible. This would significantly improve disk manageability and scalability, reduce access latency variance, and if done with care might improve perfomance. The PARAID project within the Systems Research Laboratory is designing a file system and disk storage subsystem based on these concepts. The key challenge in doing so is to maximize the number of blocks that are correctly located with the first disk rotation at high disk utilizations while operating within a severely limited main memory overhead. This paper studies at length a number of randomized hashing strategies for mapping the FSBA space to the disk LBA space that meet this objective.

The primary concerns for the mapping process are that the number of collisions is minimized, and that the expected block access time is very low. Naturally, collisions imply that further techniques are required to manage the storage of those blocks (since they must be kept somewhere), and thus are to be avoided. However, strategies which increase the block access time excessively (for example, by requiring two independent drive accesses with high probability to locate a block) in order to reduce collisions are entirely unacceptable for performance considerations. Standard single- and multiple-choice hashing algorithms are insufficient for handling this mapping process; neither is capable of supporting both objectives. We therefore construct extensions of these which are capable of simultaneously reducing collisions and access times, and compare our extensions to the originals both through theoretical analysis and through simulation.

To the best of our knowledge, randomization has not been studied as a tool for assisting in local disk block placement. There are many difficulties inherent in this strategy, including maintaining good locality, avoiding excessive complexity (and metadata), providing online adaptivity, and reducing and resolving overflows. With a properly designed system, all of these challenges can be met with a low operational overhead. The complete allocation system outlined in this paper maintains low fragmentation, uses little storage overhead, allows for high adaptivity while requiring few updates, maintains sufficient locality, and runs efficiently.

1.1 Related Works

Randomization in disk placement strategies has been used by other researchers to help reduce variance in I/O latencies [8], increase scalability [3, 9, 10], and to provide online adaptivity [13]. All these works have focused on multiple-disk systems, and many strategies are complex and/or impossible to apply to the problem of allocation to a single disk.

Research on disk-level traces [13] has indicated that only a small amount of locality is actually observed in general-purpose servers (due to the number of parallel requests for a disk). These results seem to indicate that increases in locality achieved in the FSBA space have rapidly declining value for multiprocess workloads. Individual requests seen at the disk layer involve a relatively small number of sequential blocks, and successive requests from different processes cause the disk arm to move. This defeats the benefit of further FSBA-level locality. As a result, constant locality may in practice be sufficient.

Random allocations have been studied as *balls and bins games* or hashing by theoretical computer science researchers for many years now [1, 2, 4, 6, 5, 7, 12]. These are generally studied in an online, sequential model, where balls are thrown one at a time and decisions for placement must be made based only on the decisions of the previous balls. The classic single-choice algorithm, where each ball picks a single bin uniformly at random and is placed there, has been shown to have a bin of size $\frac{m}{n} + \mathcal{O}(\sqrt{\frac{m}{n}\log n} + \frac{\log n}{\log\log n})$ with high probability, where m and n are the number of balls and bins respectively [11]. Other strategies were developed which involve choosing multiple bins and deciding between them, and these improve the overhead of the largest bin by an exponential factor, to $\frac{m}{n} + \mathcal{O}(\log\log n)$ [1, 2].

2 The Model

We deal with a different view of the balls and bins problem in this paper. The "bins" in this case are fixed regions on the disk drive; we consider them to be fixed-size buckets, and let the number of buckets and the number of bins increase at the same rate (maintaining a constant linear ratio which is the bucket size). Unlike traditional analysis, we do not concern ourselves with the size of the largest bucket; we are instead concerned with the total number of overflow elements in the system (all blocks allocated to a bucket that is already full, assuming a sequential block assignment), since these are the blocks which must be dealt with separately.

We will abstract a disk to a linear set of buckets. A bucket will be read and written by the disk in a single operation; therefore placement within a bucket is trivial. When considering the proximity of two buckets on the disk to evaluate access times, we will consider only their proximity on the line, not taking into account required seek or rotational movements necessary on a physical disk to travel from one to the other.

We consider the input to be a set of block addresses chosen arbitrarily from a large address space (with no repetitions), though the selected addresses must be capable of being placed within the physical address space. We require that the utilization on the system is bounded by $(1 - \epsilon)$ for some $\epsilon > 0$, or that of the available n blocks on the disk, only $(1 - \epsilon) \cdot n$ of them need to be filled to store all the data. We also assume that each used block is to be assigned a single unique block on the hard disk, and that all blocks are identical in size.

The simulation results presented in this paper are based on C programs which treat buckets on the hard drive as a simple linear array. It uses as its hash function the built-in pseudo-random number generator random(), which can be seeded with specific values using srandom() and then called to return a repeatable pseudo-random hash of the seed value. It is non-linear and has a period of approximately $16 * (2^{31} - 1)$, with output as integers $\in [0, 2^{31} - 1]$. This has shown itself to be by far the most random of several tested hash functions; we will not discuss in this extended abstract the other functions or the performance comparisons. Input block addresses are first XOR'd with a randomly chosen hash key, which allows the function to be replaced if necessary (by choosing a new hash key). The process of XOR with a single key ensures that input values are merely permuted, which implies that the input to the hash function contains no collisions.

The simulations compare the overflow performances of several allocation strategies. They are not meant to evaluate the compactness of the data on disk, the linearity of placement, or the efficiency of computing block location - these objectives are sufficiently analyzed by mathematical methods and by the system layout, as they do not depend on the random choices made by the process.

3 System Objectives

The system is designed to address the following objectives:

1. The used data should be compactly stored on the disk.

2. The position of a block must be efficiently determinable.

3. Linearity of data placement should be sufficiently preserved.

4. The number of overflows should be very low.[1]

The first of these objectives is addressed in section 4.2 which discusses a division of the buckets into superbuckets and the use of previous research to manage this division. The second is met by the system as a whole. The third is addressed in section 4.1, which discusses some preprocessing of the input to the system, referencing related systems research.

This paper is focused primarily on the fourth objective, the most challenging from an algorithmic perspective. To the best of our knowledge, the problem of counting overflows in hashing algorithms has not been studied. Strategies which produce a low maximum bin size likely produce an even

[1]A small number of overflows can be managed by means of a victim cache variation developed by Professor Shapiro, enabling all blocks in the system to have a unique location. For performance reasons this victim cache must reside in memory, which is the ultimate source of our need for a very low probe miss rate.

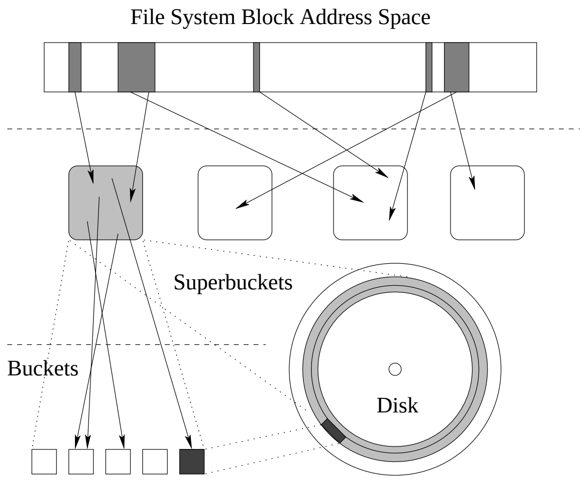

Figure 1: System Overview

load and few overflows, but the mathematical results of varying bin size, utilization, and number of blocks on overflows have not been previously analyzed.

4 System Overview

In this section, we will give an overview of the entire allocation system, explaining each of its components and demonstrating that the desired objectives are maintained. The purpose of this paper is not the complete description of nor the defense for the proposed allocation system; a brief description is given merely to motivate the work in later sections.

The system begins at the file system level with a very large virtual block address space. It collects blocks at the file system level into groups called extents, which are then mapped to a superbucket (a large contiguous disk region) at the physical level via the cut-and-paste strategy described below. Within a superbucket, the extents assigned to it are hashed to a smaller region on the disk (a subset of the region covered by the superbucket) called a bucket, which corresponds to a small set of contiguous blocks on the disk capable of storing a small constant number of extents. The bucket is read and written as a single unit by the disk, so placement within the bucket is trivial.

4.1 System Input and Extents

As mentioned previously, our system involves a very large, sparse virtual block address space (the FSBA space). Full details of the management of this space are omitted, but a larger space allows for fairly trivial management at the file system level, most notably in that it removes the need for logical address reuse for different files completely. We will treat the input to the allocation system as merely a set of logical addresses which need to be assigned physical addresses.

Rather than being divided into blocks, each file is divided into extents, or small groups of blocks. These are treated merely as larger blocks in the later stages of the analysis, and will be referred to simply as blocks. The extents are specifically constructed to be large enough to preserve most of the expected run-time locality of access as evaluated by the disk-level traces by Wilkes [13].

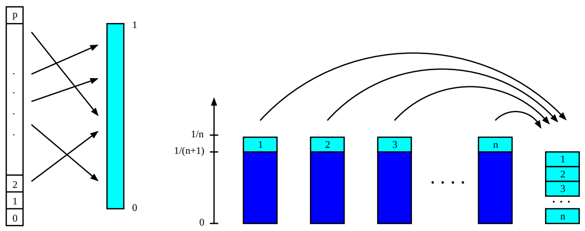

Figure 2: The cut-and-paste strategy: adding a disk.

4.2 Cut-and-paste

In order to preserve compactness of storage of data on the disk, the cut-and-paste strategy of [3] is used to divide the disk space into multiple superbuckets of a fixed size, each of which corresponds to a fixed (large) region of blocks on the disk. This strategy has the advantage of maintaining a uniform distribution of blocks to superbuckets while increasing or decreasing the number of superbuckets, while performing very few block movements. We then use only as many superbuckets as we need to store the data with sufficient utilization (increasing or decreasing the number of disks used as the number of blocks used increases or decreases), thus keeping the amount of data on the disk compact.

The algorithm works by hashing all input addresses to the real interval $[0, 1]$ (though it is also possible to discretize the range of the output function while achieving similar results). For n disks, the subintervals assigned to each disk will total exactly $1/n$ (thus the interval is evenly distributed), and this division will be maintained over the addition or removal of disks. Initially, the entire interval is assigned to disk 1. Then, to advance from n disks to $n + 1$, the top $\frac{1}{n(n+1)}$ of the set of intervals assigned to each disk is removed, and these are concatenated to form the interval assigned to disk $n + 1$, with the higher-indexed disks' portions placed lower in the set of intervals assigned to disk $n + 1$ (see figure 2). Thus the new size of the set of intervals assigned to disk $n + 1$ is $\frac{n}{n(n+1)} = \frac{1}{n+1}$, and the size of the previous disks is $\frac{1}{n} - \frac{1}{n(n+1)} = \frac{1}{n+1}$. The correct disk can be computed efficiently for any real number in $[0, 1]$ with no disk queries and only knowledge of the current number of disks n.

All following analysis examines only the situation within a single superbucket. The number of blocks allocated to a superbucket can be bounded tightly, and under the assumption that our hash function is (nearly) random, it does not matter which blocks are assigned to a superbucket (for the purposes of analyzing the collisions within the disk). Also, the superbuckets operate independently, which has the added advantage that a hash function could be replaced within a single superbucket (in the event of poor performance) without needing to rearrange blocks in the other superbuckets.

4.3 Hashing

Within a superbucket, we then must hash all logical blocks assigned to that superbucket to a set of buckets which cover the corresponding portion of the physical disk. As mentioned before, each bucket is designed to contain a fixed number of blocks (its capacity). Our objective in selecting an appropriate hash system is to minimize the number of overflow elements produced in the hashing process while keeping the process of accessing an element efficient so that we may handle all elements as easily as possible.

We use the term *hash system* to emphasize further work beyond the basic hash function, which accepts a single numerical input and produces a single repeatable random numerical output, and

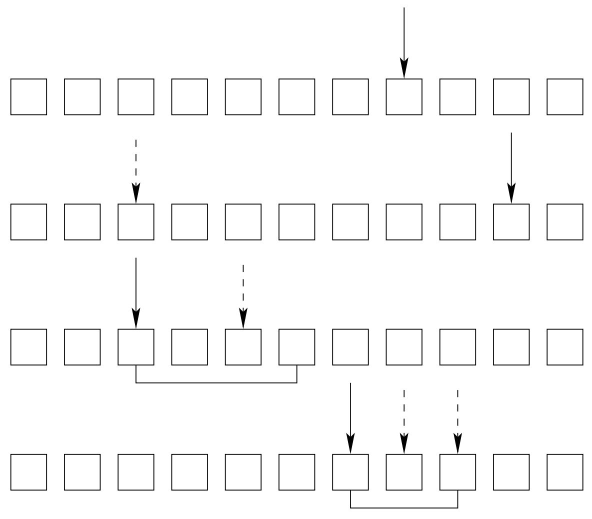

Figure 3: A sample hash in each system: single-choice, greedy multiple-choice, limited range multiple-choice, and bucket windowing

which we use as a primitive operation. Multiple-choice hashing algorithms are hash systems according to this definition, where two hash function outputs are produced and compared to determine the actual placement of the block.

The hash systems we consider for use are:

1. Standard single-choice: The block is assigned to a uniformly chosen bucket across the entire output bucket space.
2. Greedy multiple-choice: Two uniform random buckets are selected from the entire output bucket space, and the block is placed in the one with the smallest current load.
3. Limited-range multiple-choice: A first-choice bucket is selected from the entire space, and then a second-choice bucket is chosen at random within a certain range after the first.
4. Bucket window: A bucket is selected from the entire space uniformly at random to be the beginning of a small sequential bucket window. All buckets within the window are examined, and the block is placed in the least loaded bucket in the set.

5 Analytical Results

This section outlines and analyzes the above-mentioned hashing systems both for their feasibility in reducing overflows and expected access time. In the following section simulation results for the algorithms will be presented and discussed.

5.1 Single-choice hashing

The most basic, simple hashing technique involves picking a single bucket from the set of all buckets uniformly independently at random. This can be efficiently implemented by computing a hash value with an integer output (uniformly distributed) between 1 and the number of buckets in the system.

The size of the largest bucket is $m/n + \Theta(\sqrt{m \log n / n} + \frac{\log n}{\log \log n})$ [2] with high probability (where m is the number of blocks in the system and n is the number of buckets).

Theoretical analysis of hashing techniques is usually concerned with the problem of the largest bucket size. This has applications in such areas as scheduling theory (where the greatest concern is which of a set of processors assigned a set of jobs will finish last, for example), but has only limited relevance to our analysis, since any height over capacity requires equivalent additional work. Thus we consider now the problem of counting the number of overflows in a single-choice hashing system.

The following formula gives tight bounds for likelihood that a sum of independent uniformly distributed Bernoulli random variables will exceed a certain distance from its expected value [11]:

$$\begin{aligned} \frac{2}{\sqrt{2\pi}(x+\sqrt{x^2+4})}e^{-x^2/2} \leq \mathbf{Pr}[\frac{S_m - \mu m}{\sqrt{m\nu}} \geq x] \\ \leq \frac{4}{\sqrt{2\pi}(3x+\sqrt{x^2+8})}e^{-x^2/2} \end{aligned}$$

We will use the upper limit of this formula to achieve a (fairly) tight value for the expected number of balls overflowing in a single bin. Here, S_m is the random variable representing the size of a single arbitrary bin, and μ and ν represent the mean and variance of each x_i, the Bernoulli components of S_m (i.e. the variable representing whether or not each ball lands in the bin). Here, $\mu = \mathbf{E}[x_i] = p = \frac{1}{n}$ and $\nu = \mathbf{E}[x_i^2] - (\mathbf{E}[x_i])^2 = \frac{1}{n} - \frac{1}{n^2} = \frac{n-1}{n^2}$.

Define a random variable Y to be the number of balls overflowing in a single arbitrary bin y. The range of values for Y are integers between 0 and (m - capacity(y) or $c(y)$), since this maximum would mean that all m balls were placed in this bin. Therefore:

$$\begin{aligned} \mathbf{E}[Y] &= \sum_{k=0}^{m-c(y)} k \cdot \mathbf{Pr}[\text{overflow } = k] \\ &= \sum_{k=0}^{m-c(y)} \mathbf{Pr}[\text{overflow } \geq k] \end{aligned}$$

We assume partial utilization $u \in [0,1)$ and an expected bucket size $c = \frac{m}{n}$. Define $\alpha = \frac{1}{u}$ to be the "acceptable overflow" factor. Then an overflow corresponds to the amount S_m is over $\alpha\mu m$, since we expect S_m to be $\mu m = c$, but we can allow it to be slightly more because the bucket is expected to be only partially utilized. Therefore let us translate the statement "overflow $\geq k$" as follows:

$$\begin{aligned} S_m - \alpha\mu m &\geq k \\ S_m - \mu m - (\alpha-1)\mu m &\geq k \\ S_m - \mu m &\geq k + (\alpha-1)\mu m \\ \frac{S_m - \mu m}{\sqrt{m\nu}} &\geq \frac{k+(\alpha-1)c}{\sqrt{m\frac{n-1}{n^2}}} \\ &\geq \frac{k+(\alpha-1)c}{\sqrt{c}} \end{aligned}$$

The last line is a relaxation for simplicity. The right-hand side, therefore, is the x in (1), and this formula can then be substituted in (1) to give the following summation (where $x = \frac{k+(\alpha-1)c}{\sqrt{c}}$):

$$\mathbf{E}[Y] = \sum_{k=0}^{m-\alpha c} \frac{4}{\sqrt{2\pi}(3x+\sqrt{x^2+8})} e^{-x^2/2}$$

$$= \frac{4}{\sqrt{2\pi}} \sum_{k=0}^{m-\alpha c} \frac{e^{\frac{-(k+(\alpha-1)c)^2}{2c}}}{3(\frac{k+(\alpha-1)c}{\sqrt{c}}) + \sqrt{\frac{(k+(\alpha-1)c)^2}{c} + 8}}$$

This translates to (roughly) an expected constant fractional overflow for the bucket if α and c are fixed. Dividing the formula by c gives the probability that a single ball will overflow in the system, which can be logically extended to be the expected fraction of balls in the total system which overflow. A graph of sample values for this probability based on fixed utilization $u = \frac{1}{\alpha}$ and actual bucket capacity $\alpha\mu m$ is given; the m at the top of the summation is omitted since the higher-order terms contribute little to the total. A larger bucket size produces a flatter increase line at lower utilizations, but as utilization increases the larger bucket becomes less and less capable of helping, and all curves approach the same increase line (where almost all blocks produce a new overflow).

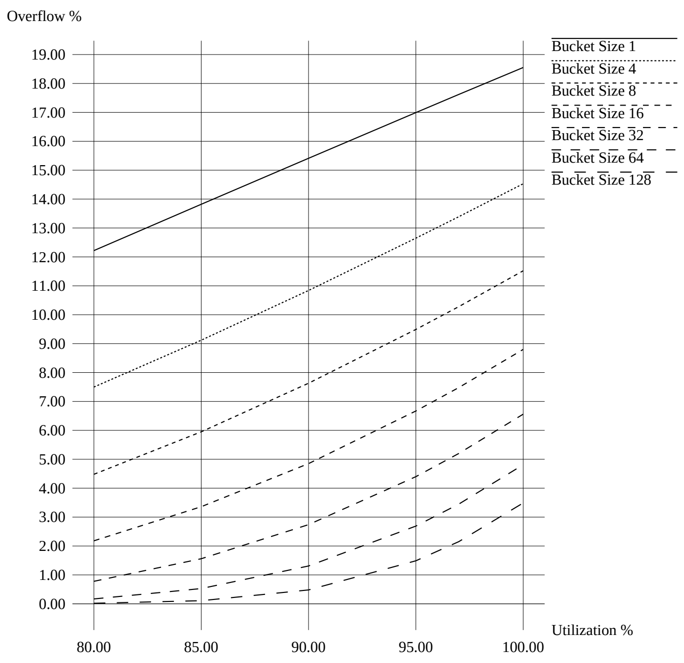

Figure 4: Single-choice hashing overflows, theoretical upper bounds.

The complexity of this formula and the lack of sufficiently tight analyses of the probabilities of general overflow amounts in other hashing strategies cause the analysis of expectations of overflows using other strategies to be beyond the scope of this paper.

5.2 Multiple-choice hashing

Multiple-choice hashing techniques involve selecting a small set of buckets for each block and placing the block in the least loaded bucket of the set. To find a block, then, one must store some form of directory information for each bucket to indicate the presence of the block; then in reading, all of a block's possible buckets must be read until the correct one is found. Algorithms for these techniques differ primarily in how they select the set of candidate buckets.

The first algorithm for this is known as the Greedy strategy, which selects the buckets in the set uniformly at random from the entire set of buckets. If there are d buckets chosen for each block, in the case $m = n$ the largest bucket is of size $\log\log n/\log d + \Theta(1)$ with high probability [1]. Another possible strategy which achieves a slightly better load is the Always-Go-Left strategy of [12], which partitions the set of buckets into d regions and selects one from each region. Then the least loaded of the selections is used; in the event of a tie the leftmost bucket of minimum load is selected. This strategy achieves a maximum load of $\ln\ln n/(d * \ln\phi_d) - \mathcal{O}(1)$, where ϕ_d is a function of d which approaches 2 for increasing d (and is always less than 2).

Since these strategies achieve a significantly better maximum bin size than the single-choice strategy, it follows that they would probably correspond to significantly lower overflows. However, they are not efficient enough to be used in a physical disk system - the reading of a single block will use $\frac{d+1}{2}$ disk operations in the expected case (assuming all d choices are equally likely), which represents a large overhead since disk accesses are expensive. One can avoid this to some extent by biasing the selection of the first choice, for example by assigning a block to the first of its choices with room to hold it. This would greatly speed access, but also would increase collisions (approaching the single-choice case).

5.3 Multiple-choice hashing with limited range

While multiple-choice strategies require too much overhead to be feasible, it is possible to modify these strategies to achieve a more realistic system which can approach the load balancing capabilities of multiple-choice hashing while dramatically reducing the expected access time. For example, within the $d = 2$ Greedy strategy, the choice of the second bucket could be restricted to lie within a small fixed-size range of buckets immediately after the first choice bucket (within a range parameter r). Then, after the random second choice is selected, the two buckets can be located in the same pass of the hard disk drive (thus avoiding lengthy arm seeks). Simulations indicate that, for realistic values of n, even small values for r are sufficient to provide comparable results to general unlimited range multiple-choice strategies, though at higher utilizations the limitations of the restricted range become more apparent.

However, probabilistic analysis of this limited range strategy shows that reasonable ranges cannot guarantee small bins. Rather than analyzing directly the situation that a certain bucket receives a load of size k, let us instead examine the probability that $2k$ blocks choose the same two specific choices for buckets. Call these two buckets b_0 and b_1. Clearly one of the two has load at least k. Let us also for simplicity deal solely with the case $m = n$.

By Chernoff's bounds, the probability that bucket b_0 has at least $2k$ blocks choosing it first is at most $\left(\frac{e}{2k}\right)^{2k}$. Then, the probability that all these choose b_1 second is equal to $\left(\frac{1}{r}\right)^{2k}$.

This must be at most $\frac{1}{n}$, so:

$$\begin{aligned}
\left(\frac{e}{2kr}\right)^{2k} &\leq \frac{1}{n} \\
2k - 2k\ln(2kr) &\leq -\ln n \\
2k(\ln(2kr) - 1) &\geq \ln n
\end{aligned}$$

This implies that if $r = poly(\log n)$, then $k = O(\frac{\log n}{\log\log n})$ is required, or the best load we can guarantee is asymptotically identical to single-choice hashing.

5.4 Single-choice hashing with bucket windows

Another systems-focused alternative strategy involves extending the traditional single-choice hashing techniques. Instead of simply using the first bucket selected randomly by a block to be placed, examine the bucket and a few of its successors (the *bucket window*), and use the one with the lightest load. This seems to require an additional operation during the disk write (since it must read the bucket sizes before selecting one to write), but in fact in all of the above strategies, including standard single-choice, a read operation must be performed before writing a block, to determine whether or not the bucket still has available space. Thus the only additional cost is the small extra rotational delay to read a few more buckets during the block read.

This strategy involves an inherent tradeoff. The larger the size of the bucket window, the greater the utilization can be while producing fewer overflows. However, a larger bucket window incurs a greater operational overhead when searching for a block, since more buckets need to be searched to find the block.

The remainder of this section contains two theoretical analyses of the bucket window strategy. The first is a simple argument that it is expected to be better than a single-choice strategy with a larger bucket (one the size of the total size of the bucket window). The second is a proof that for a bucket window of size $o(\log n)$, the best that can be proved is a load near that of the simple single-choice strategy, though with a window of size $O(\log n)$ constant deviation per bucket can be achieved with high probability.

5.4.1 Comparison to single-choice

This analysis of the strategy centers around the effects of two nearby "tall" buckets, or buckets which are chosen as the initial hash of at least h balls each (i.e. without using a bucket combination, bucket window, or any other hashing technique). It disregards the placements and corresponding load induced by all other balls whose initial hash was to some other bucket as well as balls placed after the first h to each of the tall buckets. We will compare two strategies for fixing these tall buckets by redistributing some of their balls. In one, we will combine all buckets into fixed sets or ranges of r buckets each, where any ball hashed to one of the r buckets is distributed evenly among all r buckets (i.e., all hash targets of any of the r buckets are equivalent, essentially treating the set as one bucket r times as large). In the other, we will examine the tall bucket and the $r-1$ following buckets (the "bucket window"), and distribute each of the h balls assigned to the tall bucket across this set uniformly. Intuitively, these strategies appear almost equivalent, where the first strategy will be better whenever the two tall buckets are in separate ranges, and the second will be better whenever they are in the same range. But probabilistic analysis will show a clear preference for the second strategy, regardless of the distance between the two buckets.

We examine the relative loads imposed on the potentially affected buckets in each strategy. The central difference between the two strategies is that if two buckets are in the same range in the combining strategy, then they have more room to spread their elements out in the bucket window case, while if they are in separate ranges they have more room to spread out in the fixed range case. However, the amount of room they have to spread out in the bucket window strategy increases as the distance between them increases. Thus, as it becomes more likely that the buckets are in separate ranges, it also matters less since the bucket window strategy has more space.

Let the fixed range algorithm be A_f, and the bucket window be A_w. Also let the distance between the two buckets be $d \in [1,(r-1)](\in \mathbb{Z})$. If the distance is 0 or at least r (where distance is measured in the number of buckets - two sequential buckets are at distance 1), then the two strategies are equivalent - at distance 0 they have the same space of r buckets, and at distance at least r they cannot affect each other in either strategy.

If the two buckets are in the same range, then the cost to A_f is defined as $\frac{2h}{r}$, since there is a load of $2h$ being produced by the two tall buckets which must be spread out over r buckets. The cost to A_w is then $\frac{2h}{r+d}$, since the separation of d units implies that the two bucket windows overlap by $r-d$, or that the total space covered is $2r-(r-d)=r+d$. Now, the probability of this occurring is determined by the location within the fixed range of the first tall bucket's position. There are $r-d$ positions within the r positions of the range for which the second bucket will be inside the range, and thus the probability of this situation occurring is $\frac{r-d}{r}$.

If the two buckets are in different ranges, then the cost to A_f is $\frac{h}{r}$, since each tall bucket can spread its load over r buckets independently of the other. The cost to A_w is still $\frac{2h}{r+d}$. This situation has probability $\frac{d}{r}$. These two facts lead us to compute:

$$\begin{aligned}
\mathbf{E}[\frac{C(A_f)}{C(A_w)}] &= \frac{r-d}{r}\cdot\frac{r+d}{r}+\frac{d}{r}\cdot\frac{r+d}{2r} \\
&= \frac{r^2-d^2}{r^2}+\frac{dr+d^2}{2r^2} \\
&= \frac{2r^2-2d^2+dr+d^2}{2r^2} \\
&= 1+\frac{d}{2r}-\frac{d^2}{2r^2} \\
&> 1
\end{aligned}$$

The last inequality holds because $d/r<1$. This indicates that for any separation d, one would expect the bucket window algorithm to perform better than the fixed range approach.

5.4.2 Chernoff tail bounds

Let us consider two sequential fixed ranges of r buckets, i and $i+1$. Let S_i be the number of blocks which select a bucket within range i as their first choice. If S_i is at least rc for some value c to be determined, then some bucket in i or $i+1$ has load at least $\frac{c}{2}$, since there are at least rc blocks to distribute across $2r$ buckets. S_i is binomially distributed, and can be bounded by Chernoff's bounds, with mean r (assume $m=n$):

$$\mathbf{Pr}[S_i \ge rc] \le \left(\frac{e^{c-1}}{c^c}\right)^r = \frac{1}{e^r}\left(\frac{e}{c}\right)^{cr}$$

This needs to be upper bounded by $\frac{1}{n}$ to ensure that no bucket has load $\frac{c}{2}$ - this condition is far from sufficient, but is certainly necessary. Thus:

$$\begin{aligned}
\frac{1}{e^r}\left(\frac{e}{c}\right)^{cr} &\le \frac{1}{n} \\
\left(\frac{e}{c}\right)^{cr} &\le \frac{e^r}{n} \\
cr - cr\ln c &\le r-\ln n \\
cr\ln c - c + r &\ge \ln n
\end{aligned}$$

Assuming that $r=o(\log n)$, this requires $cr\ln c = O(\log n)$, which means $c=O(\frac{\log n}{poly(\log\log n)})$. Thus the bucket window strategy offers no significant asymptotic improvement over the single-choice strategy for small ranges, and certainly is not comparable to the unrestricted multiple-choice strategy.

Note that a range of $r = O(\log n)$ is sufficient to achieve excellent results (constant deviation), while in the multiple-choice limited range strategy this is not the case; in fact, even arbitrary range multiple-choice strategies cannot perform as well unless the number of choices is allowed to be larger than a constant.

6 Simulation Results

Simulation results can be found in figures 5, 6, 7, and 8; note that all figures are in log scale. All graphs measure the number of blocks that overflow corresponding to simulations run with a varying utilization. The bucket size is 32, and the number of potential blocks in the system is $2^{22} = 4194304$. This is the total number of positions available for blocks in the system; fixed numbers were used to focus the simulations on the relationships between the algorithms rather than attempting to achieve specific numerical results. Numbers in the range $[0, 4194303]$ are chosen independently with probability equal to the desired utilization to produce the input set of blocks. This is a small input space, but, assuming the hash function is good enough, results are equivalent to using a larger input space with any means deterministic or random of selecting blocks as input. Simulations were run with a fixed number of possible blocks varying in utilization to better represent the different situations possible at a single hard disk drive; the graphs demonstrate the levels of utilization possible in each strategy without allowing more than 1% of the blocks to overflow.

A new heuristic was added in the simulations. Blocks were allowed to select from their choice bins according to the rule "first-fit"; in other words, as long as the first choice bucket had any space left, blocks were placed there, without examining alternate choice buckets. This was applied to bucket windowing by placing the block in the first bucket in the window with any available space. This notion is introduced to attempt to reduce the expected access time to blocks by biasing them towards their first choice. This of course increases the number of overflows produced in the system as a side effect.

The first-fit-only graph studies the number of overflows for the candidate algorithms when using the first-fit metric. It uses a single sample parameter each for the limited range and bucket window strategies. It also includes for comparison the performance of single-choice hashing with comparable bucket sizes. Note that both algorithms outperform the corresponding single-choice results significantly, though the performance curve shadows that of the single-choice (making the algorithmic improvements less significant); this is intuitively supported since the system will not even use the algorithmic modifications until it nears capacity, at which point the system is already unbalanced. Also worthy of note is that when first-fit is used, the bucket window strategy greatly outperforms even the general multiple choice strategy; this indicates that the bucket window strategy is far more capable of repairing an unbalanced allocation than limited range multiple choice.

The first-fit heuristic allows the number of second choice blocks (those successfully placed in the second random choice in multiple choice hashing or a later bucket in the bucket window) to be reduced to 2-5% in both limited range multiple choice and bucket windowing, as opposed to approximately 33% in multiple choice and 45% in bucket windowing (for the other three positions combined) when unrestricted. The system parameters would need to be considered to determine whether the resulting increase in overflows (which is quite considerable) is tolerable.

The no-first-fit graph shows that limited range with a range of 30 well outperforms a bucket window strategy of size 4, but with comparable ranges (limited range 4 actually examines 5 buckets total, which gives it more than the bucket window) the bucket windowing strategy is superior. However, the closeness of the two curves of similar ranged bucket windowing and limited range multiple choice suggests that the most important parameter is not which of these strategies to use but rather how many additional buckets are allowed to be examined (at least when the ranges are small). Both the strategies use the additional space in a manner far superior to simply enlarging the buckets by the same amount.

The limited range multiple choice and bucket windowing graphs note the affects of modifying the range parameters in the two strategies, both with and without first-fit; they also demonstrate the cost incurred by using the first-fit heuristic, as much higher utilizations are possible with the same number of overflows without first-fit.

An observation worthy of note is that the variance in the simulation results across different executions of the same test was extremely small. This is due to the aggregate nature of the performance metric - while individual bin sizes may vary widely, the overflow amount measures the ability of a large number of bins to be significantly over their expected load, enough to exceed their capacity.

It should be clear that these techniques allow overflows to be kept quite small, even at high utilizations, with little algorithmic overhead.

7 Conclusion

The placement strategies described in this paper achieve the stated goals of low fragmentation, small overhead, efficient operation, and maintaining sufficient locality. The new hashing systems introduced are capable of greatly reducing the number of overflows even at very high utilizations. By utilizing additional space in a more intelligent way (rather than simply enlarging output buckets), the balancing abilities of traditional multiple-choice hashing can be approached with more practical algorithms which can maintain low expected block access time. While neither multiple-choice limited range hashing nor single-choice with bucket windows achieve the desired theoretical performance guarantees (such as those for general multiple-choice hashing), both perform very well in simulations and are quite capable of serving as adequate random mappings for the proposed allocation system.

For the purposes of single disk allocation, only strategies using the first-fit heuristic provide sufficient performance. In this application 95%-97% of blocks must be found in the first bucket probed. Both limited range multiple choice hashing and bucket windowing are capable of this at over 90% utilization (with first-fit). The memory cache is capable of managing approximately 10000 overflows. Both limits are met by bucket windowing with a window of size 5 at up to 90% utilization (with second choices under 4%). Limited range multiple choice hashing cannot compete when the first-fit heuristic is used, and produces unacceptable overflow at the lowest tested utilization even with a large range. For applications requiring extremely small overflows, or for applications with different latency parameters, limited range multiple choice hashing and no first-fit are also quite practical.

8 Acknowledgments

We wish to thank Dylan Adams for much initial work on the simulator.

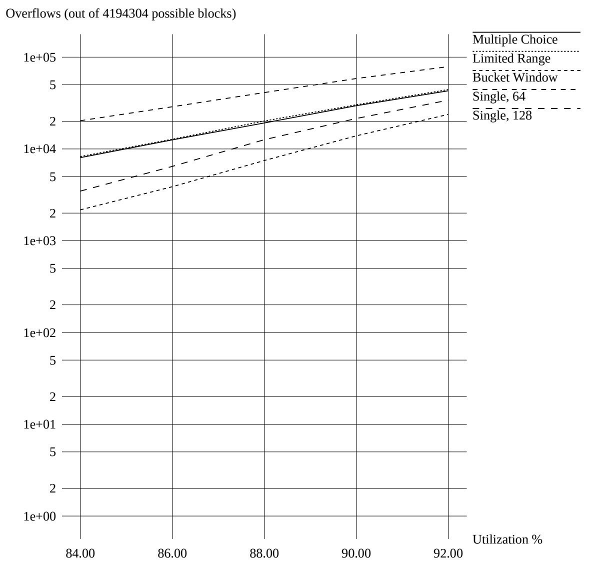

Figure 5: First-fit; greedy multiple-choice, limited range MC with range = 30, bucket window of size = 4, single choice with bucket size = 64, single choice with bucket size 128

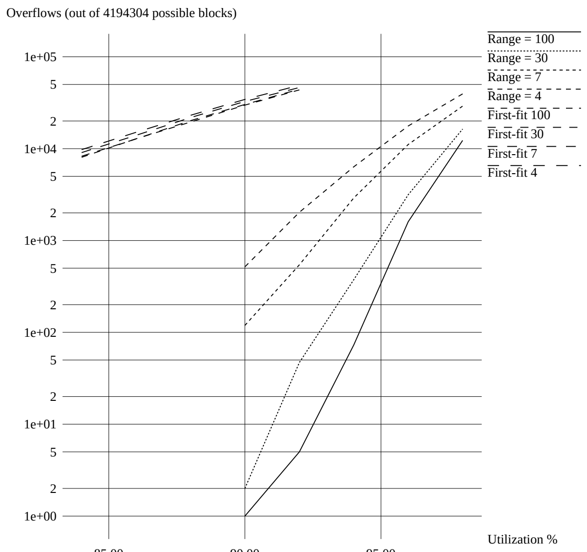

Figure 7: Limited range multiple choice, normal (90-98% utilization) and first-fit (84-92% utilization), with ranges 100, 30, 7, 4

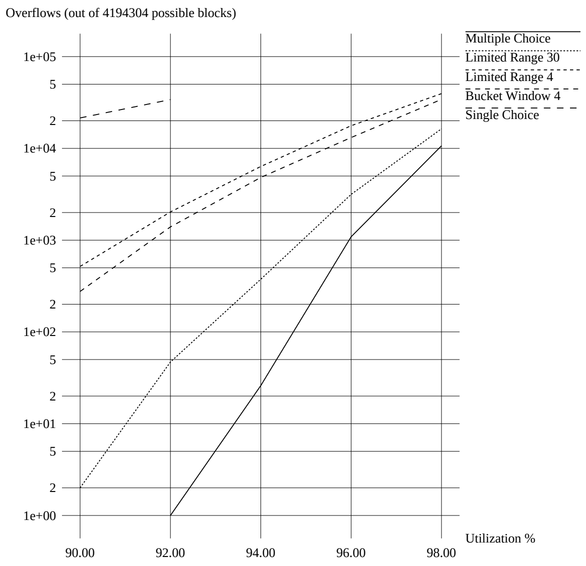

Figure 6: No first-fit; greedy multiple choice, limited range MC with range = 30 and 4, bucket window of size = 4, single choice with bucket size = 128

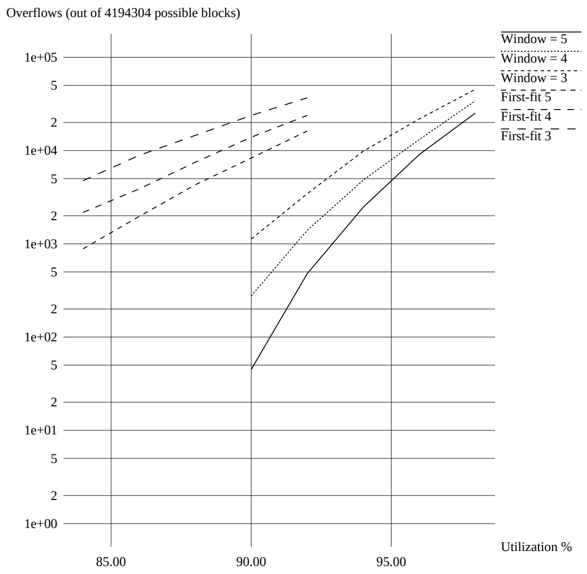

Figure 8: Bucket windowing, normal (90-98% utilization) and first-fit (84-92% utilization), with window sizes 3, 4, 5

References

[1] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, Sept. 1999. A preliminary version appeared in *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 593–602, Montreal, Quebec, Canada, May 23–25, 1994. ACM Press, New York, NY.

[2] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 745–754, Portland, OR, May 21–23, 2000. ACM Press, New York, NY.

[3] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–128, 2000.

[4] Cole, Frieze, Maggs, Mitzenmacher, Richa, Sitaraman, and Upfal. On balls and bins with deletions. In *Proceedings of the RANDOM*, pages 145–158, 1998.

[5] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results.

[6] M. Mitzenmacher and B. Vöcking. The asymptotics of selecting the shortest of two, improved. In *Proceedings of the 37th Allerton Conference on Communication, Control, and Computin*, Urbana, IL, 1999.

[7] M. Raab and A. Steger. "Balls into bins" — a simple and tight analysis. In M. Luby, J. Rolim, and M. Serna, editors, *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science*, Lecture Notes in Computer Science 1518, pages 159–170, Barcelona, Spain, October 8–10, 1998. Springer-Verlag, Berlin.

[8] J. Renato. Design of the rio (randomized i/o) storage server.

[9] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, San Francisco, CA, January 9–11, 2000. SIAM, Philadelphia, PA.

[10] J. R. Santos, R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Measurement and Modeling of Computer Systems*, pages 44–55, 2000.

[11] C. Scheideler. *Probabilistic Methods for Coordination Problems.* University of Paderborn, 2000.

[12] B. Vöcking. How asymetry helps load balancing. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 131–141, New York City, NY, October 17–19, 1999. IEEE Computer Society Press, Los Alamitos, CA.

[13] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems*, 1:108–136, 1996.