A Bit-Parallel Search Algorithm for Allocating Free Space

Randal Burns and Wayne Hineman
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
$\{$*randal,hiney*$\}$*@almaden.ibm.com*

Abstract

File systems that allocate data contiguously often use bitmaps to represent and manage free space. Increases in the size of storage to be managed creates a need for efficient algorithms for searching these bitmaps. We present an algorithm that exploits bit-parallelism, examining all bits within a processor word at the same time, to improve search performance. Measurements of our implementation show that these techniques lead to a 14 times increase in the rate at which bitmap pages can be searched on a 64-bit processor. Trace-driven experiments indicate that overall allocation performance increases by a factor of 3 to 6 on a 32-bit processor. As processors mature, registers become wider, and the degree of bit-level parallelism increases, which makes the performance improvements of our search algorithm more substantial.

1 Introduction

We describe a method for representing free space in a storage system with bitmaps and present an efficient algorithm for searching bitmaps for contiguous allocations. While bitmaps have desirable properties for allocating data contiguously, they have a performance shortcoming because searching bitmaps for free-space is less efficient than searching other data structures. We address this shortcoming by providing improved algorithms for searching bitmaps.

Our algorithms employ bit-parallel techniques, computing results on all bits of a processor word concurrently, to improve performance. Therefore, as processors grow faster and their registers become wider, the benefits of our algorithm increase. Our treatment includes a comparison of asymptotic complexity results with performance measurements of our implementation. The algorithm has time complexity in $\Theta(n \log |W|/|W|)$ to search n bits in registers of size $|W|$ as compared to $\Theta(n)$ for algorithms that consider bits sequentially. This translates to an increase in the search rate by a factor of over 14 on a 64-bit processor.

We also present a trace-based simulation of the operation of our algorithm. The experiment is derived from our file system implementation. We use a trace-based approach to more realistically characterize performance against real workloads. We play system-call traces through a file system simulation to create bitmaps that have been "aged", so that the contents of the bitmap represent the internal state of a production file system. Experiments run against these traces indicate a 3 to 6 times performance increase for allocations.

Bit-parallel algorithms are commonly used to achieve a speed-up proportional to the register width in processor and arithmetic unit design. Recent work on massively-parallel processing exemplifies these techniques [14].

Algorithmic and architectural problems increasingly employ bit-parallelism. Pattern matching algorithms to search for sub-strings [8, 10] probably have the most in common with free-space bitmap search. For problems that can be mapped onto bit-parallel operations, increasing processor register widths promise to improve time performance by large factors.

Bitmaps are frequently employed to represent free space, particularly when storage needs to be allocated and liberated at a fine granularity [16, 7, 2, 4]. Long presents an analysis that establishes the scalability of bitmaps to large storage systems [6]. In a free space bitmap, each bit represents one block of storage. A high value of a bit indicates that the corresponding block of storage is available and a low value of the bit indicates that it is allocated. To search for a contiguous allocation for a data object in a bitmap, the allocator takes as input the size of the object and looks for enough adjacent free bits to contain that object in one contiguous region of storage.

2 Representing Free Space with a Bitmap

In our system, we have chosen to represent free space using bitmaps because of their advantages, and we address performance limitations by developing improved search algorithms. Bitmaps have many useful properties including *two-way addressing*, self organization, and parallelism. The primary shortcoming of bitmaps is that searching for an allocation is typically less efficient than with other techniques.

Extent lists are the most common alternative to bitmaps. Extent lists are linked data structures that organize free space into contiguous regions described by location in storage and length of available blocks. They are sometimes preferred for compression, representing free space using less memory and storage. However, they can grow to be larger than bitmaps, particularly when storage is heavily utilized by small allocations.

Bitmaps have two-way addressing – a bit offset in the bitmap directly indexes a block offset in storage, and the block offset in storage indexes a bit offset in the map. The free-space bitmap for a storage volume consists of contiguous bits that describe the allocation state of contiguous storage blocks. The reverse indexing (block address to bit) makes *liberation* of a range – returning an existing allocation to free space – trivial and, therefore, very efficient. In contrast, liberating a range in an extent list requires searching to locate the neighboring free space extents that need to be merged with the liberated space. Searching becomes prohibitively expensive when an extent list is large.

Two-way addressing also makes it possible to extend an existing allocation in place. When extending a file, it is easy to discover (without searching) whether free space adjacent to the current allocation is available.

Bitmaps are self-organizing, whereas extent lists are not. Self-organizing refers to the placement of the data structure in memory and on physical storage. In a bitmap, adjacent blocks in storage always correspond to adjacent bits in the bitmap. Searching a range of storage for an allocation is localized to the few pages of bits that describe that range. In contrast, as extents are inserted and removed, the extent list becomes "disorganized" – extents adjacent in storage are not necessarily on adjacent or even nearby memory addresses in the extent list. Searching an extent list has no spatial locality guarantees. Lists that are larger than the available memory can result in heavy paging traffic. Extent list search performance and organization may be improved by keeping extents ordered using an indexing data structure, like a B-tree [15].

Bitmaps are also inherently parallel. To synchronize multiple processes acting against a bitmap, we require only latches against bitmap pages, simple synchronization of physical resources based on shared memory [9]. To synchronize access to linked data structures requires locks – access control to logical objects using a lock manager. Latches have been shown to outperform locks by a factor of 10 in a database [9].

Another alternative to both bitmaps and extent lists is the buddy system [5]. While efficient for allocation and more efficient than extent lists for liberation, the buddy system is not self-organizing and externally fragments storage, leading to poor utilization for fine-grained allocations.

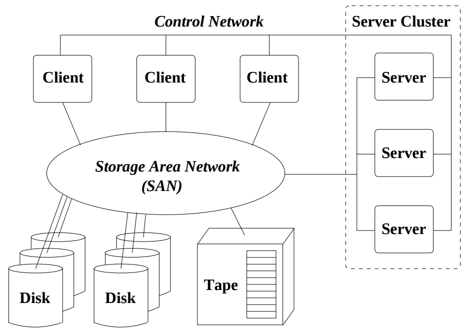

Figure 1: Schematic of the Storage Tank client/server distributed file system.

2.1 Implementation in the Storage Tank File System

A brief digression into the file system architecture in which we implement bit-parallel search for data allocation helps to motivate our solution and its advantages. In the Storage Tank project at IBM research, we are building a distributed file system based on storage area network (SAN) technology. A SAN is a high speed network designed to allow multiple computers to have shared access to many storage devices.

One goal of Storage Tank is to take advantage of SAN hardware to provide computers with the scalability and management benefits of a distributed file system without the performance penalty expected from a client/server architecture. To achieve performance, clients with SAN attachments access data directly over a high-speed network (Figure 1). The direct data access model is similar to the file system for Network Attached Secure Disks [1] that uses shared disks on an IP network and the Global File System [11] for SAN attached storage devices. Clients communicate with Storage Tank servers over a logically separate network, called a control network, to exchange protocol messages for locking, recovery, naming, metadata, and file allocation (based on bitmaps). Servers are clustered for parallelism, high availability, and scalability. Many servers cooperate to serve a single file system namespace, and they seamlessly migrate workload when servers fail or to perform dynamic load balancing.

A bitmap representation of free space is important because Storage Tank relies heavily on the contiguous reallocation of data to preserve read performance. Storage Tank uses data clustering to keep file data contiguous or nearly contiguous. Contiguous allocation allows a file system to read a file with fewer I/Os and, therefore, realize high data rates. In contrast, files fragmented across a disk drive require many more I/Os and incur costs when seeking between fragments. For data clustering, Storage Tank performs many allocations that extend

Memory Words	$W_{31}W_{30}W_{29}\cdots\cdots W_2\ W_1\ W_0$	$W_{31}W_{30}W_{29}\cdots\cdots W_2\ W_1\ W_0$	$W_{31}W_{30}W_{29}\cdots\cdots$
Bitmap Addresses	$B_0\ B_1\ B_2\ \cdots\cdots B_{29}\ B_{30}\ B_{31}$	$B_{32}\ B_{33}\ B_{34}\cdots\cdots B_{61}\ B_{62}\ B_{63}$	$B_{65}\ B_{66}\ B_{67}\cdots\cdots$

Figure 2: Mapping the bitmap address space onto a big-endian memory (or storage).

current file allocations in place or allocate data "near" (in terms of block addresses) the existing allocation. Two-way addressing and efficient liberation in bitmaps makes frequent re-allocations perform well. Also, the self-organization property prevents performance from degrading from loss of locality over time.

Storage Tank also benefits from the high-degree of parallelism that bitmaps support. A single server is expected to serve hundreds of clients at a time (this type of scalability is possible because server's do not provide data). Concurrent transactions from many clients are executed on the server, and the lightweight synchronization of latches in bitmaps allows us to allocate data in parallel to different pages of the same bitmap.

3 The Allocation Problem on Bitmaps

We now look at the general allocation and liberation problem on bitmaps. Allocation consists of searching for a suitable region and marking that region occupied in the bitmap. An algorithm is given an input length k and finds a string of k consecutive bits valued 1. To make the allocation, the algorithm changes the value of the found bits to 0 and returns a pointer (or index) to the start of the run. For liberation, we are given an offset in the bitmap r and a length k and set the corresponding bits to 1. Liberation is algorithmically trivial and will not be treated further.

We refine our formulation to reveal the effect of processor words and define a more specific allocation search problem. The algorithm is given a bitmap consisting of some number of words, each of size $|W|$, a target length k, and a suffix length s initially equal to zero. Starting with the first word, it performs:

1. Determine whether the first $\min\{|W|, k-s\}$ bits are high. If not, go to step 2.
   (a) If $k-s > |W|$, then set $s \leftarrow s + |W|$. Move on to the next word and repeat step 1.
   (b) If $k-s \leq |W|$, then return the found allocation.
2. If $k \leq |W|$, search for k consecutive bits valued 1 within the word. If k consecutive bits exist, return the found allocation.
3. Find the longest suffix of the word consisting of bits valued 1 and set s equal to this length. Move on to the next word, and go to step 1.

This algorithm searches sequentially through words trying to find a consecutive run of bits valued 1 with length k. In step 1, the algorithm determines whether it can extend the suffix of the previous word with a prefix in the current word to meet the target. Failing to meet the target, the algorithm attempts to fulfill the target within the current word in step 2. If the target cannot be met within the current word, the algorithm finds the longest suffix of high bits in the current word that it attempts to extend in the next word. For long targets, the algorithm can construct a prefix that spans many words.

A straightforward implementation of this algorithm would consider the bitmap one bit at a time. However, we provide a more efficient implementation by taking advantage of bit-level parallelism.

4 Allocation Algorithm

We describe in detail a bit-parallel allocation algorithm that follows the steps described in Section 3. Sections §4.1, §4.2, and §4.3 correspond directly with steps 1,2, and 3 presented above.

In describing our algorithms, we employ the following terminology. We continue to use variables s (suffix), k (length), and $|W|$ (word size). We introduce variable p that describes the prefix length in the current word required to fulfill the target. We label the free-space bitmap $\mathcal{W}$, consisting of a series of words, $\mathcal{W} = W^1, W^2, \ldots, W^n$, each of size $|W|$, so that a processor can conduct logical and arithmetic operations against words of size $|W|$ in unit time. We assume $|W|$ to be a power of two.

For ease of understanding bit operations and displaying bitmap contents in debuggers and programs, we describe our algorithms in a big-endian memory (most significant byte first) with the bitmap bits ordered so that they are sequential and increasing. Figure 2 shows the relationship between processor words and bitmap bits. Bitmap bits that have adjacent indices are adjacent in memory, which facilitates viewing and debugging bitmap contents. More importantly, in the big-endian format the left and right shift operators ($\ll$ and $\gg$) correspond to left and right shifts in registers and memory. Note that $\ll$ and $\gg$ are specified to shift registers toward the most and least significant bit, not necessarily to the left and right.

4.1 Finding the Prefix

The first step of the algorithm, determining if a prefix of the word added to the suffix of the previous word fulfills

																Shift	Gap Size
	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	$\dots$	$a_{\lvert W\rvert-1}$		1
$\wedge$	0	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	$\dots$	$a_{\lvert W\rvert-2}$	1	
	0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	$\dots$	$b_{\lvert W\rvert-1}$		2
$\wedge$	0	0	0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	$\dots$	$b_{\lvert W\rvert-3}$	2	
	0	0	0	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	$\dots$	$c_{\lvert W\rvert-1}$		4
$\wedge$	0	0	0	0	0	0	0	c_3	c_4	c_5	c_6	c_7	c_8	$\dots$	$c_{\lvert W\rvert-5}$	4	
	0	0	0	0	0	0	0	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	$\dots$	$d_{\lvert W\rvert-1}$		8
$\wedge$	0	0	0	0	0	0	0	0	d_7	d_8	d_9	d_{10}	d_{11}	$\dots$	$d_{\lvert W\rvert-2}$	1	
	0	0	0	0	0	0	0	0	e_8	e_9	e_{10}	e_{11}	e_{12}	$\dots$	$e_{\lvert W\rvert-1}$		9

Figure 3: Gap-search routine.

the target, can be calculated directly. Given target k and suffix of the previous word of length s, the prefix length p must be of length $p = k - s$ to fulfill the requested allocation. To check for a length p prefix in word W^i, for any p less than $|W|$, calculate if

$$(\vec{1} \gg p) \vee W^i = \vec{1}, \qquad (1)$$

where $\vec{1}$ indicates a word containing all high bits. If true, an allocation has been found at the target size starting at offset $|W| - (s \mod |W|)$ in word $W^{i-(\lceil s/|W|\rceil)}$. The suffix s can be much larger than $|W|$, so that the allocation spans many words. If the expression evaluates false and if $k < |W|$, the algorithm continues with the next step (Section 4.2) to find a run of consecutive ones within the current word. Alternatively, if k is too large to fit within a single word ($k \geq |W|$), it skips searching within the word and find the longest suffix of all ones (Section 4.3).

Our algorithm treats any p larger than $|W|$ exceptionally. In this case, targets larger than $|W|$ need to construct an allocation across multiple words. We check to see if the current word describes available space, *i.e.* $W^i = \vec{1}$, and can contribute to such an allocation. If so, it increments s by the word size, $s \leftarrow s + |W|$, and advances to repeat the prefix search on the next word. Otherwise, the algorithm starts anew, finding the suffix of the current word (Section 4.3) before returning to the prefix routine.

4.2 Searching the Word

Our method uses bit-parallel search techniques to find allocations smaller than $|W|$ that are contained within a given word. If an allocation of the target size t exists, it starts at some offset between 0 and $|W| - t$ in word W^i. To find the allocation, the algorithm repeatedly shifts and "ands" the word, using the word and word-wide processor primitives to compute intermediate results at all bit offsets concurrently. We call this portion of the algorithm the "gap-search" routine.

Figure 3 illustrates the operation of the gap-search routine for an allocation of size 9. The algorithm proceeds in rounds. In each round, the algorithm shifts the result and "ands" the shifted value with the previous result. To search for a gap of size k, the algorithm conducts $\lceil \lg k \rceil$ rounds with each round shifting the bits *minimum*$(2^R, k - 2^R)$ where R is the number of the round, $R \in \{0, 1, 2, ..., \lceil \lg k \rceil - 1\}$. In our example, the algorithm conducts $\lceil \lg 9 \rceil = 4$ rounds with shifts of $1, 2, 4$ and $1 = 9 - 2^3$.

Table 3 includes a "*gap size*" column that describes how large a contiguous gap each bit in the word represents. Obviously, for the original word $\langle a_1, a_2, \ldots \rangle$, each bit corresponds to a unit allocation gap. For later rounds, looking at the example clarifies the *gap size*. The bit d_9 summarizes eight bits in the original word.

$$\begin{aligned} d_9 &= c_5 \wedge c_9 \\ &= b_3 \wedge b_5 \wedge b_7 \wedge b_9 \\ &= a_2 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_6 \wedge a_7 \wedge a_8 \wedge a_9 \end{aligned}$$

Since d_9 is a conjunction of values from the previous rounds, d_9 with value one represents eight consecutive ones in the original word. To calculate the *gap size*, we add the *shift* to the previous *gap size*. When this search routine terminates, the *gap size* always equals the target.

When the gap-search routine terminates, any high bit in the result word ($W(R)$) indicates that an allocation of the target size can be made. To test this, the algorithm merely evaluates $W(R) \neq \vec{0}$. For a non-zero result word, the algorithm performs binary search to find the high bit representing contiguous free space and return its offset. If no gap exists, the algorithm continues to look at the suffix (Section 4.3).

4.3 Finding the Suffix

The routine for finding the suffix draws on the same concepts and reuses the results computed in the gap-search routine. If the algorithm just completed the gap-search routine, we reuse the result words. We label these words by round, *i.e.* $W(0)$ is the original word and $W(R)$ the result of round R. If these results are not available, because the algorithm came directly from the prefix routine, we compute result words $W(R)$ using $|W|/2$ as the target size. We address the choice of $|W|/2$ in Section 4.3.2.

The suffix-finding routine searches backward through the rounds using already computed values to extend the known length of the suffix. The routine starts by considering the result of the latest round with a high rightmost

$W(0)$	...0000000$\underline{\mathbf{0}}$11111111
	...0000000001111111
$W(1)$	...0000000$\underline{\mathbf{0}}$01111111
	...0000000000011111
$W(2)$	...0000000$\underline{\mathbf{0}}$00011111
	...0000000000000001
$W(3)$	...000000000000000$\underline{\mathbf{1}}$
	...0000000000000000
$W(4)$	...0000000000000000

(a) Suffix = 8

$W(0)$	...000$\underline{\mathbf{1}}$111111111111
	...0000111111111111
$W(1)$	...000$\underline{\mathbf{0}}$111111111111
	...0000001111111111
$W(2)$	...0000001$\underline{\mathbf{1}}$11111111
	...0000000000111111
$W(3)$	...000000000011111$\underline{\mathbf{1}}$
	...0000000000000000
$W(4)$	...0000000000000000

(b) Suffix = 13

$W(0)$	...0$\underline{\mathbf{1}}$11111111111111
	...0011111111111111
$W(1)$	...001$\underline{\mathbf{1}}$111111111111
	...0000111111111111
$W(2)$	...0000111$\underline{\mathbf{1}}$11111111
	...0000000011111111
$W(3)$	...000000001111111$\underline{\mathbf{1}}$
	...0000000000000000
$W(4)$	...0000000000000000

(c) Suffix = 15

Figure 4: Examples of the Suffix-Finding Routine

bit. For notation, the latest round with a high rightmost bit is called $W(L)$. The n^{th} bit within $W(L)$ is indicated by $W(L)_n$; the last bit is $W(L)_{|W|-1}$. The best way to describe the operation of the suffix-finding routine is by example.

4.3.1 Suffix Finding Example

We consider the rightmost bits of three different words, with suffixes of 8,13, and 15, for which the gap-search algorithm looked, but did not find, a target of 16. These examples are called E8, E13, and E15, respectively. Figure 4 displays the results of the gap-search algorithms for the rightmost 16 bits for each of the three original words. Also in each figure, the bits that the suffix algorithm considers in each word are underlined.

For all three examples, the rightmost bit of $W(3)$ equals one, implying a suffix of at least 8. In all cases, the algorithm shifts its focus, indicated by the underlined bit, to the previous round $W(2)$ and 8 bits to the left. The algorithm examines bit $W(2)_{|W|-9}$ next, which is high for examples E15 and E13, and low for E8. Since each bit in $W(2)$ represents a gap of size 4, the known suffix length increases to 12 for examples E13 and E15.

Upon encountering a low bit, like E8 does, the focus of the algorithm shifts to the previous round, but remains at the same bit offset. For the remainder of its rounds, E8 remains at known suffix length 8, never again finding a high bit in its focus.

For E13 and E15, the focus shifts four bits to the left and to the previous round ($W(1)_{|W|-13}$). This is where examples E13 and E15 diverge. Bits in round 1 represent gaps of size 2 and the known length of the E15 grows to 14. Example E13 has a low bit and stays at known length of 12.

In the final round, E15 again shifts it focus, this time two bits to the left, and E13 remains on the same bit, only changing its round. Both examples find a high bit and increase their known length by one. The suffix routine terminates after examining the original word $W(0)$, having found suffixes of size 8, 13, and 15, respectively.

4.3.2 Suffix Algorithm

The suffix-finding algorithm works backward through the results words of the gap-search algorithm to determine the length of the suffix. It starts with round $R(L)$, which is the highest round with a high rightmost bit, *i.e.* $W(L)_{|W|-1} = 1$. The rightmost bit in round L indicates that the available suffix has minimum size 2^L. We also know that the suffix is smaller than 2^{L+1} because the rightmost bit in $W(L+1)$ is low. As we will show later, the shift in round L is always a full shift (size 2^L, rather than $k - 2^L$) so that bits represent gaps of length 2^L.

The algorithm iteratively refines the possible length of the suffix by looking in lower rounds. In round $L-1$, if bit $W(L-1)_{|W|-2^L-1}$ is high, the suffix is at least $2^L + 2^{L-1} = 3 \cdot 2^{L-1}$ in length, and if it is low, the suffix is at most $3 \cdot 2^{L-1} - 1$ in length. The bit $W(L-1)_{|W|-2^L-1}$ represents the 2^{L-1} contiguous bits that precede the current 2^L bits of the suffix. Figure 5 provides pseudo-code for this algorithm.

```
size = 0;
round = $L$;
do {
  if ( 0x01 $\wedge$($W$ [round] $\gg$ size) ) {
    size $+=$ 2 $\ll$ round;
  }
  round $-=$ 1;
} while ( round >= 0 );
return size;
```

Figure 5: Finding the suffix length

We argue that the shift of round L must be 2^L by contradiction. If round L contained a partial shift, it must be the last round of the gap-search algorithm. Then, $W(L)_{|W|-1} = 1$ implies that a target size gap exists. However, this is not possible because the gap-search routine would have successfully found an allocation and terminated the algorithm. We conclude that L can be no higher than the penultimate round.

A similar argument explains the choice of $|W|/2$ as the target when the suffix routine calculates the result words, because the gap-search routine was not run. Round $\lg(|W|/2)$ calculates gaps of size $|W|/2$. The following round, round $\lg|W|$, computes gaps of size $|W|$. Were the rightmost bit in the round $\lg|W|$ high, the prefix routine of the algorithm would find a gap of length $|W|$. In this case, the prefix routine would have moved on to the prefix routine at the next word, rather than continuing to the suffix routine (Section 4.1).

5 Performance Measurement

We have two goals in measuring our implementation of the bit-parallel search algorithm: (1) to isolate properties of the method and verify algorithmic analysis; and (2) to determine what gains this technique provides in a file system on real workloads. To achieve these goals, we designed two families of experiments. In both, we compare bit-parallel search methods to a linear search of a file system bitmap that looks at each bit in the map sequentially. First, we look at performance measured against idealized bitmaps and compare our results directly against expectation based on analysis. In the second family, we run file system traces to "age" the allocation bitmaps. After aging, the contents of the bitmaps resemble those of a production file system. Experiments run against aged bitmaps provide insight into the gains our method realizes in practice.

5.1 Analysis and Verification

Our bit-parallel algorithm provides a speed-up proportional to the degree of parallelism, as defined by the word size, as compared to a linear search of a bitmap. A linear search takes a fixed number of steps to examine each bit and uses time in $\Theta(n)$ to search n bits. The bit-parallel search algorithm requires $\Theta(n \log|W|/|W|)$ time to search n bits. For each word in a bitmap, the bit-parallel search looks for a prefix (§4.1), searches within the word for the target (§4.2), and finds the longest suffix (§4.3). The algorithm performs $\Theta(1)$ work to evaluate the prefix. Both the search and suffix take time in $\Theta(\log|W|)$ as they each conduct $\log|W|$ rounds. All steps are conducted on $n/|W|$ words.

To show that our algorithm realizes this bound, we run the two algorithms against a bitmap in which all blocks are allocated. Since there are no free blocks, bitmap search algorithms run continuously, never finding free space. Our performance measure for this experiment is search rate – the number of memory pages that an algorithm can search per second.

The performance of the bit-parallel algorithm also varies with the width of the processor register. To observe this, we varied the fundamental data type used to describe the bitmap from 8 bits (character) to 64 bits (long). This experiment was conducted on a 133 MHz DEC(Compaq) Alpha running Digital UNIX 5.0. We chose this machine because it implements 64-bit words and pointers natively in both the hardware and the operating system. This allows us to vary our experiments over word sizes of 8, 16, 32, and 64 bits and have the machine implement shifts and arithmetic against these data types in hardware.

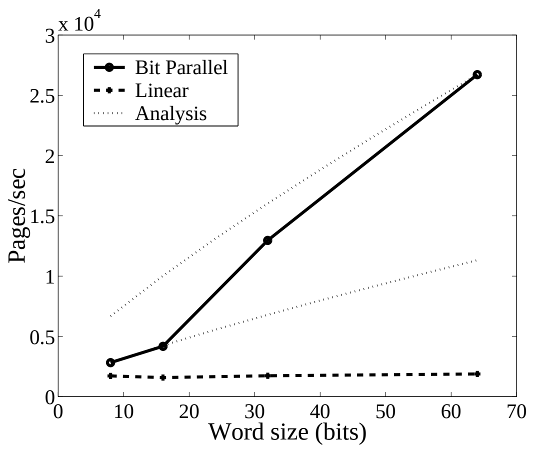

Figure 6: Search performance when no blocks are free.

Varying the data type in the bitmap only approximates different register widths, and system side-effects occur. The Alpha processor addresses memory at 64-bit alignment. Operations against smaller data types (8, 16, and 32 bits) involves shifting, which reduces performance as we will see shortly.

We create a bitmap in a single 8192 byte memory page and initialize this page so that all bits are valued zero, indicating that space is occupied. We then search this page repeatedly (over one million times) using both algorithms. Since we use a single memory page and initializing the page creates the memory and brings it into the cache, our measurements include no latency from page faults.

Figure 6 shows the result of this experiment, comparing the search rate in pages per second as a function of the processor word size for both algorithms. For linear search of a bitmap, the rate does not vary significantly with the word size. There is a slight increase with increasing word size that can be attributed to inefficiencies when doing arithmetic on data types that are not aligned for addressing [3]. However, this effect is not significant when compared with the differences between algorithms.

Bit-parallel search shows a slightly better than expected increase in performance as the word size grows. Based on asymptotic results, when we double the word size, we expect the algorithm to perform less than twice

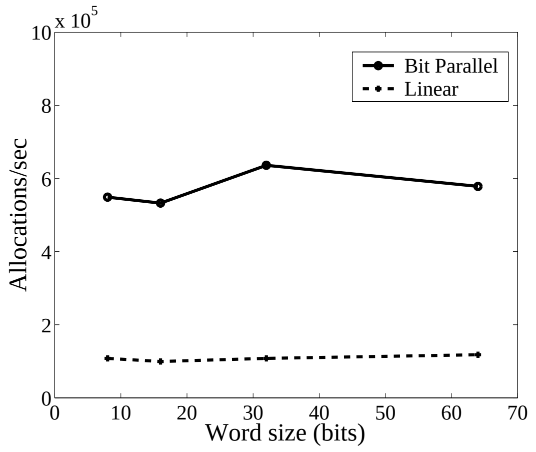

Figure 7: Allocation performance on an empty disk.

as fast. The number of words that need to be considered is halved, but more work (one more shift in the gap-search routine) must be done at each word. In our experiments, doubling the word size more than doubles the performance of the algorithm. For comparison, we have included in the figure two lines that show the asymptotic results normalized to the 8-bit rate and 64-bit rate respectively. The better than expected speed-up results from inefficiencies when operating on nonaligned types (the same effect seen in the linear search algorithm). Experimental results show that for 64-bit words, our algorithm provides more than a factor of 14 increase in the rate at which bitmap pages can be searched, as compared to a increase of about 1.7 times for 8-bit words, 3.3 times for 16-bit words, and 6.9 for 32-bit words.

Making allocations, rather than searching for them, is an important component of the allocation workload that is overlooked by the previous experiment. Often, free space on a disk is immediately available or found after a short search. This is particularly true with large disk drives and file systems that consume blocks sequentially, as does Storage Tank. A good bitmap search algorithm must also perform well in this case.

To test performance on allocations where space is available immediately, we initialize a bitmap to contain available blocks and test how many 64 block allocations per second an algorithm can make. We actually left the first byte of bitmap occupied to force the bit-parallel algorithm into the suffix algorithm. We chose 64 blocks so that all variants of all algorithms had to look in multiple words. Figure 7 shows comparative performance of bit-parallel and linear searching. Bit-parallel search improves performance by approximately a factor of 5 for allocations of 64. Allocation rate does not scale with the word size. Note that any variant of the bit-parallel algorithm only conducts the suffix algorithm once. Having found a suffix, the algorithm can extend the suffix by prefixes, which are found immediately (Equation 4.1).

We do note that linear search outperforms bit-parallel search for trivial instances of this experiment. For example, linear search allocates one free bit about twice as fast as parallel search and allocates 8 free bits at the same rate. However, for such small instances the cost of searching the bitmap with any method is negligible when compared with computation for allocation.

5.2 Trace-based Aging

Our second family of experiments runs allocation workloads, both trace-driven and artificial, against "aged" bitmaps that resemble those in a production file system.

To create aged bitmaps, we employed the system call traces developed at the University of California [12], which record three months of activity on HP 700 workstations. We played these traces through a file system allocation simulator designed to emulate the allocation and data placement policies of the FFS file system [7]. After playing three months worth of file system calls, we extract the "oldest" half of the file system allocation bitmap. Since FFS consumes blocks on an empty disk sequentially, the oldest blocks are those that were allocated first. The premise is that the oldest blocks have seen the most activity in terms of deletions/liberation and re-allocations to extend previously allocated files in place.

We conducted file system aging for three different machine traces: a web server (WEB), an instructional machine used in an undergraduate laboratory (INS), and a machine used for research projects by faculty and staff (RES). The INS trace consists of mostly small files that are created and deleted frequently, and the oldest parts of its bitmaps have 51% of space free. The RES workload has slightly larger files that are more stable and has 38% free space. Unlike the previous two traces, the WEB trace contains very stable data that is rarely deleted and has only 4% free space in aged bitmap pages.

We conducted this family of experiments on an AMD Thunderbird Gigahertz processor running Redhat Linux 7.0. We chose this system because it is a target architecture for Storage Tank and representative of current hardware. However, unlike our Alpha, this machine is a 32-bit processor.

Having aged the bitmaps with traces, we replayed the traces again and measured the allocation performance of each workload. In doing so, we only search for allocations and do not alter the aged bitmaps, consuming more disk space. We randomly chose offsets within the bitmap at which we started an allocation search.

Storage Tank's allocation policy, which we used for these experiments, is to only search through 256 Megabytes of storage (8192 bytes in the bitmap) before

Trace and Algorithm	Allocs/sec	99% Confidence
INS: Linear	3.338×10^5	± 1605
INS: Bit-parallel	9.998×10^5	± 921
RES: Linear	1.603×10^5	± 232
RES: Bit-parallel	6.187×10^5	± 570
WEB: Linear	1.4969×10^4	± 299
WEB: Bit-parallel	9.478×10^4	± 301

Table 1: Trace-driven performance results on aged bitmaps (allocations per second).

deciding that the disk is near capacity and extending the logical volume. Storage Tank is not confined to a single disk and does not need to search exhaustively. For this experiment, this policy bounds the search length in nearly full bitmaps.

Table 1 holds the performance results from these trace-driven experiments. Bit-parallel techniques increase allocation rates by a factor of 3 in the instructional workload, a factor of nearly 4 in the research workload, and by more than a factor of 6 in the web-serving workload.

Search performance varies significantly from trace to trace. In fact, the linear allocation rate in either RES or INS exceeds the bit-parallel rate in the web workload. The characteristics of the three workloads account for these differences. The instructional and web workloads both involve many small files that are created and destroyed. Mean allocation sizes are 2.3 blocks for the research workload and 2.6 blocks for the instructional workload. The majority of allocations in these traces are small files or portions of files that need either one or two blocks. Many of these have been deleted, liberating many small chunks of storage. Subsequent allocations of small files find these liberated gaps and populate them. In contrast, the Web workload consists of slightly larger files (mean allocation of 3.6 blocks) that are infrequently deleted. Fewer allocations can be satisfied immediately and algorithms search longer.

Based on the results of the trace-driven experiments, we suspected that allocation performance was highly dependent upon the number of blocks requested. Therefore, we probed our aged bitmaps with allocations of different sizes. In this experiment, we performed many (more than a million) allocations at all target allocation sizes between 1 block and 32 blocks. The starting search offset in the bitmap was randomized for each allocation.

Figure 8 contains the results of this experiment and shows the allocation rate as a function of the requested size. All algorithms exhibit performance that degrades as the target size increases. For the linear search algorithm, performance falls quickly to a knee, and then degrades more slowly. The bit-parallel search algorithm displays a more moderate decline that can be attributed

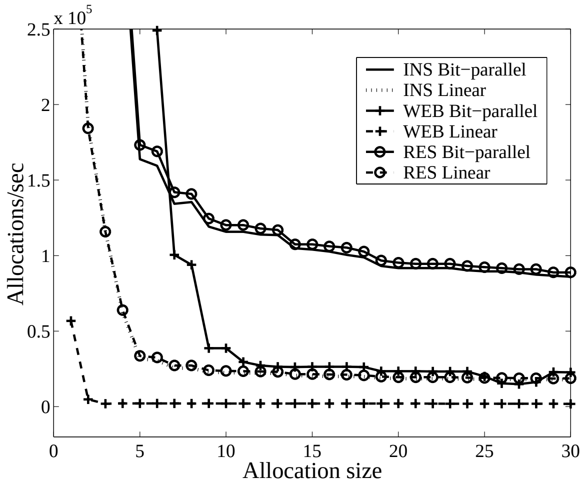

Figure 8: Allocation rates as a function of allocation size.

to its faster search rate. Declines in performance verify that free space is heavily fragmented and that allocations larger than about 5 blocks (corresponding to a file of 20K) require fairly extensive searching. This is true even for the INS trace in which the majority of space is free.

These results make it clear that if a file system intends to allocate data contiguously in a region of a disk that is already partially occupied, it must be prepared to search extensively. Reduced bitmap search performance argues strongly that it is undesirable to allocate data in portions of the disk that have already been used. File systems that never deallocate have been proposed [13]. However, this approach leads to very low storage utilization, as we see in the INS trace in which over half of the blocks used in the previous three months are no longer allocated to files. Most enterprises are unable or unwilling to discard such a large portion of their capacity. Therefore, previously used disk space must be liberated and allocated to new files. In this environment, bit-parallel search improves performance by a factor of 3 to 6, depending upon the workload.

6 Conclusions

The many examples of systems that use bitmaps and their desirable performance properties establish that bitmaps are an important data structure for managing free-space. We selected bitmaps for reasons of efficiency when liberating data or extending contiguous allocations, and because lightweight synchronization constructs allow for concurrent allocations.

We have presented an efficient algorithm for searching free space bitmaps based on the concept of bit-level parallelism. Our algorithm has time complexity in $\Theta(n \log |W| / |W|)$ to search n bits with word size $|W|$, as compared to complexity $\Theta(n)$ for algorithms that consider bits sequentially. This indicates that as processor registers increase in width, the performance of our method increases commensurately.

Experimental results based on idealized bitmaps verify these findings, showing that for 64-bit words our algorithm provides more than a factor of 14 increase in the rate at which bitmap pages can be searched.

Performance measurements of our implementation based on bitmaps that have been aged through file system traces show that bit-parallel search improves performance by a factor of 3 to 6 in practice. We also determined that as file systems age, many small allocated regions are liberated, leaving highly fragmented free space. To effectively use this space, a file system needs to be able to search it efficiently for both large and small allocations.

Acknowledgments

We are indebted to Jim Wyllie of the IBM Almaden Research Center for showing us the fundamental concepts used in the "gap-search" routine. We also extend many thanks to Zachary Peterson of the University of California, Santa Cruz who developed a file system simulator that was the foundation for our trace-based aging. Finally, we appreciate the efforts of Scott Brandt and Darrell Long, both of the University of California, Santa Cruz, whose critical readings of early versions of this paper helped direct our research.

References

[1] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attach secure disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, July 1997.

[2] R. L. Haskin. Tiger shark – A scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2), 1998.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1995.

[4] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the USENIX San Francisco 1994 Winter Conference*, January 1994.

[5] D. E. Knuth. *The Art of Computer Programming, Volume 1*. Addison Wesley Longman, 3rd edition, 1998.

[6] D. D. E. Long. A note on bit-mapped free space allocation. *Operating Systems Review*, 27(2), April 1993.

[7] M.K. McKusick, W.N. Joy, J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3), August 1984.

[8] G. Meyers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3), May 1998.

[9] C. Mohan, D. Haderle, B. Lindsay, H Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transaction on Database Systems*, 17(1), March 1992.

[10] A. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, 1998.

[11] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the 16th IEEE Mass Storage Systems Symposium*, 1999.

[12] D. Roselli and T. E. Anderson. Characteristics of file system workloads. Research report, University of California, Berkeley, June 1996.

[13] D. S. Santry, M. J Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[14] I. D. Scherson, D. A. Kramer, and B. D. Alleyne. Bit-parallel arithmetic in a massively-parallel associative processor. *IEEE Transactions on Computers*, 41(10), October 1992.

[15] A. Sweeney, W. Hu D. Doucette, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 Usenix Annual Technical Conference*, 1996.

[16] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, 1997.