Binmaps: hybridizing bitmaps and binary trees

Victor Grishchenko
Delft University of Technology
victor.grishchenko@gmail.com

Johan Pouwelse
Delft University of Technology
peer2peer@gmail.com

Abstract

This paper addresses the classical problem of keeping huge bitmaps predominantly consisting of long ranges of zeros and ones. The problem is most often encountered in filesystems (free space tracking) and network protocols (transmission progress tracking).

Three classical solutions to the problem are plain bitmaps (NTFS), extent lists (TCP SACK) and extent binary trees (XFS, Btrfs). Bitmaps are simple but have high fixed space requirements. Lists are able to aggregate solid ranges, but they don't scale well with regard to search. Extent binary trees are able of aggregation, allow scalable search, but have high overhead and extremely bad worst case behavior, potentially exploding to sizes a couple orders of magnitude higher than plain bitmaps. The latter problem is sometimes resolved by ad-hoc means, e.g. by converting parts of an extent tree to bitmaps (Btrfs). Another possible workaround is to impose a divide-and-conquer multilayered unit system (BitTorrent).

We introduce a new data structure named "binmap", a hybrid of bitmap and binary tree, which resolves the shortcomings of the extent binary tree approach. Namely (a) it has lower average-case overhead and (b) as it is tolerant to patchy bitmaps, its worst-case behavior is dramatically better.

1 Introduction

The problem of very large bitmaps appears quite often in systems programming. One high-profile case is file systems that need some means of tracking free space on disks. Another example is network protocols that track the state of transmission. This particular work was motivated by the necessity to track the per-datagram state of transmission in a multiparty transport protocol without resorting to unit stacking and similar workarounds.

Normally, the problem is resolved by using plain bitmaps, extent binary trees, unit layering or a combination of those approaches. Plain bitmaps, the simplest approach, are bit vectors where each bit stays for a unit of data, taking values 0 or 1. Older filesystems, e.g. NTFS used bitmaps for free space tracking; that allowed for greater scalability than the file allocation table (FAT, [2]). Bitmaps scale poorly as well, namely as $O(N)$, which problem becomes critical as volumes become larger [1]. A straightforward enhancement is to employ a kind of run-length encoding and thus to process solid extents, not single units. An example from a different domain, TCP Selective Acknowledged Options (TCP SACK [5]) employ a list of extents to communicate which data already made its way through the network. As the amount of state is small, 4 extents at most, extent arrays are used both as a transmission and storage format. File systems must handle millions and billions of extents. A further enhancement is to put extents into a binary tree to ease search and modification. That approach is used by most modern filesystems (like XFS [3], ZFS [1], Btrfs [6]). ZFS tracks free space using "space maps"; those are log-structured append-only lists of allocation/deallocation events; once the kernel reads a space map into RAM, it is transformed into an AVL-tree of extents. The new Linux filesystem Btrfs uses a red-black tree of extents. The problem of binary trees is their extremely bad behavior in the worst case. The perfect worst case is the extremely fragmented allocation pattern $101010\ldots$; it has $\frac{N}{2}$ extents and thus the extent tree needs a node per every two bits of the bitmap. It has to be noted, that a binary tree node is typically much larger than two bits.

Thus, implementors search for workarounds. Btrfs converts parts of the extent tree to bitmaps once the layout becomes too patchy. Ideally, those bitmaps have to be converted back to extents once the layout solidifies. Another popular workaround is unit layering. Namely, the entire space is divided into some kind of pieces and those pieces are handled separately, thus making the worst case not so bad. An extreme example of this divide-and-conquer workaround is the BitTorrent [8] protocol; each transmission (a torrent) is divided into pieces, pieces are further divided into chunks; at the TCP level, chunks are further subdivided into IP packets. Packet-level handling is delegated to the kernel, and just a limited number of pieces/chunks are processed simultaneously to minimize state complexity. Naturally, these workarounds trade state complexity for code/protocol complexity as those ephemeral entities, once introduced, start forming complex interrelations.

As neither approach is completely satisfactory, we take an effort to hybridize binary trees and bitmaps to have all four advantages at once: easy search, aggregation, lower footprint and graceful degradation.

2 Binmaps

A *binmap* is a binary tree consisting of (say) 32-bit *cells*, each made of two 16-bit *halves*. Each half is either a bitmap of *layer* L or an index pointing to a cell at layer $L-1$. "Layer" of a bitmap means that its every bit stands for an aligned 2^L-long *base* bit range, i.e. a range in a virtual plain bitmap storing the same data. Layer 0 is the base layer. Data is always *aggregated* to the highest layer possible. Consider an example. Once a cell at layer L has two leaf (bitmap) halves 11001111 00110000 and 11111100 00000000, then it is immediately aggregated into its parent half at layer $L+1$, which becomes a leaf with a value of 1011 0100 1110 0000. Thus, long ranges of zeros or ones stay well aggregated into logarithmic bins. Considering a file system, if in some area the space is allocated consistently in multiplies of 4MB (aligned), a binmap for that area will not be more detailed than 1 bit for 4MB, plus another bit of overhead.

A binmap is naturally hosted in a vector of integers. Every 16th cell hosts flag bits for the previous 30 halves; flag bits are used to distinct "deep" halves (carrying offsets) from "leaf" halves (with bitmaps). Given that a half has 16 bits, a binmap may have up to 2^{16} cells. It is easy to imagine a case when more cells are needed to describe some layout. While it is always possible to resort to the brute force solution of using 64-bit cells and, correspondingly, 32-bit offsets or pointers, we will employ the half-size pointer technique. To extend the maximum size of a binmap and also to adapt the data structure to paged storage, a binmap might be chunked, i.e. subtrees might be hosted at different vectors/chunks. By reserving one bit of an offset-carrying half as a flag, we may point either directly at a child cell or to another vector/chunk, where the child cell is a root. Thus, we extend the capacity to $2^{15} \times 2^{15} = 2^{30}$ cells.

Binmaps bear some resemblance to tries using implicit $\{0, 1\}$ alphabet; those are sometimes used to keep sparse bitmaps. A paged binmap has block storage adaptations very similar to those of B$\pm$trees. The log-bin aggregation approach is also popular, e.g. three-layer aggregating bitmaps were used in [7]. Still, to the best of our knowledge, the combination of all three principles was not used before.

All binmap algorithms are implemented through iterators; to compensate for the lack of keys and parent pointers, iterators both calculate the current position and keep the backtracking stack. Reading/writing is implemented in the straightforward way, by navigating to the target location and reading/writing the bitmap. Search for a free spot of a given size is implemented as sequential iteration by the means of depth-first search; at

this point, binmaps have an interesting ability to prune search branches that definitely do not have a bin of necessary size. Bitwise operations with binmaps are possible and take $O(N+M)$ operations where N and M are sizes of respective binmaps. Once both bitmaps are extended to the same level at every single point, every pair of leaf nodes might be subjected to a bitwise operation in the normal way. Practically, that is implemented by parallel iteration of all three binmaps.

3 Performance and footprint

Regarding asymptotical big-O performance, binmaps are not special; they feature $O(\log_2 N)$ element lookup and other common characteristics of binary trees. One apparent shortcoming of binmaps, compared to most binary tree flavors, is the lack of rebalancing. It is not essential; as we are dealing with bounded integer keys, a tree cannot degenerate into a linked list.

Good thing about binmaps is their memory footprint being better by linear factor. There are five reasons for that. First of all, differently from extent trees, binmaps do not have to store offsets and lengths, as those are clear from the position of a cell in the tree. Albeit, that comes at a cost of sometimes building a logarithmical "staircase" of cells, where extent trees might use just a single node. Second, a typical red-black or AVL tree implementation employs three pointers per node (left child, right child, parent) while binmaps may live with two, because a binmap iterator has a backtracking stack. Third, a typical red-black/AVL tree implementation has pointers at the leaf nodes as well (those are set to NULL). Binmaps do not have those. Fourth, binmaps may work with half-size pointers (the offsets). Fifth, the last but not the least, binmap leaf nodes are bitmaps, so they could accommodate several intervals, while a node in an extent tree strictly corresponds to a single interval. That saves binmaps in the worst case scenarios.

At bare minimum, a node of an extent tree has a bit budget of three pointers, two long integers and some metadata, $3 \times 32 + 64 \times 2 + 32 = 256$ bits per node for 32-bit architectures. A binmap cell may start at 32 bits plus 2 bits of metadata; 7.5 times less than a node. This factor may vary depending on implementation details, pointer size and other considerations, still it provides a good estimation.

Considering the aforementioned perfect worst case (Sec. 1), binmaps need twice as much memory as bitmaps because of the binary tree overhead; extent binary trees do much worse, starting at $\times 128$. Although the perfect

worst case might be considered improbable or unrealistic, similar situations may naturally occur locally in some parts of a bitmap.

In spite of being completely ignored by big-O approximations, nontrivial multiplicative improvement in storage size is still a worthy objective. Hence, the next question is: when we express a given grand bitmap as a binmap and also as an extent tree, do we normally need more cells than nodes or vice versa?

4 Experiments

We implemented a simulation to test how binmaps compare to extent trees in realistic scenarios. The simulation was mostly inspired by the problem of a file system tracking its free space. Namely, a simulation starts with a range of 2^N *units* being "free". Every tick, a *block* of 2^k units is "allocated", where k follows discrete normal distribution. The distribution of block sizes is thus discrete lognormal. Each block is allocated in the leftmost aligned logarithmical bin that is able to accommodate it. Each block has lifetime $L = 2^l$, again assigned according to discrete lognormal distribution. After L ticks, the block is "freed" back. Thus, the allocation algorithm tries to solidify the layout by packing blocks to the left; while deallocation constantly introduces random "holes".

See Fig. 1 for the correspondence between the number of allocated *blocks*, the number of solid *intervals* they form and the number of binmap *cells* needed to describe the layout. As the number of intervals defines the number of nodes in the extent tree describing the layout, cells vs intervals comparison is the most interesting indicator. Depending on the parameters, binmaps performed slightly better or slightly worse than extent trees. The number of binmap cells used was within factor $\frac{1}{2}$ to 2 compared to the number of solid intervals (see Fig. 2), with a notable exception of "bad" cases when the number of intervals exploded (see Fig. 3).

Due to the properties of the discrete lognormal distribution, allocation to deallocation ratio was different at different stages of the model's life. Empirically, cells-to-intervals ratio went down during periods when deallocation prevailed over allocation. The effect has a straightforward explanation: binmaps deal better with holey layouts created by deallocation, the extreme example being the perfect worst-case of every second unit being allocated. To the contrary, extent trees deal better with perfectly-packed solidified layouts created by allocation. This correlation is illustrated by the Fig. 2, derived from the same model run as Fig. 1; the blocks-to-intervals ratio

shows how holey the current layout is.

Thus, we conclude that the number of binmap cells needed to describe a layout is comparable to the number of nodes in an extent tree describing the same layout, for this particular model. It is definitely possible to craft a layout that biases the result in favor of either data structure. Still, the worst case for extent tree is much worse. Considering the tree node size to binmap cell size relation (see Sec. 3), the footprint of binmap is expected to be smaller by some significant factor, although the particular number depends on the implementation and the conditions.

5 Conclusion

Binmaps represent a lighter-footprint alternative to extent binary trees for storing/processing huge amounts of well aggregatable binary state. Binmaps are resilient to the worst case of highly-fragmented bitmaps which leads to "explosion" of extent trees. Compared to popular workarounds, binmaps are simple, reliable and neat.

This work was supported by the P2P-Next project, EC FP 7, grant no 216217, http://p2p-next.eu.

References

[1] Jeff Bonwick: "Space maps" http://blogs.sun.com/bonwick/entry/space_maps

[2] Standard ECMA-107 "Volume and File Structure of Disk Cartridges for Information Interchange"

[3] A. Sweeney et al: "Scalability in the XFS File System", USENIX'96

[4] Microsoft TechNet: "How NTFS Works"

[5] RFC 2018: "TCP Selective Acknowledgment Options"

[6] Btrfs wiki http://btrfs.wiki.kernel.org

[7] E. Witchel et al: "Mondrian Memory Protection", ASPLOS-X

[8] Bittorrent Protocol Specification v1.0 http://wiki.theory.org/BitTorrentSpecification

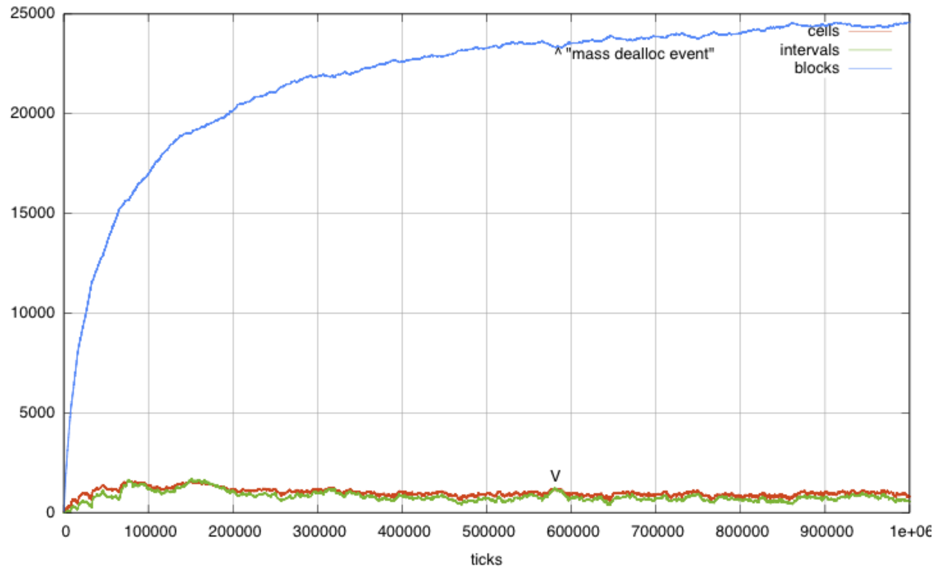

Figure 1: Simulation run, $N = 30$, $k \in [0, 20]$, $l \in [0, 25]$. Notice the slight deallocation just before tick 600,000.

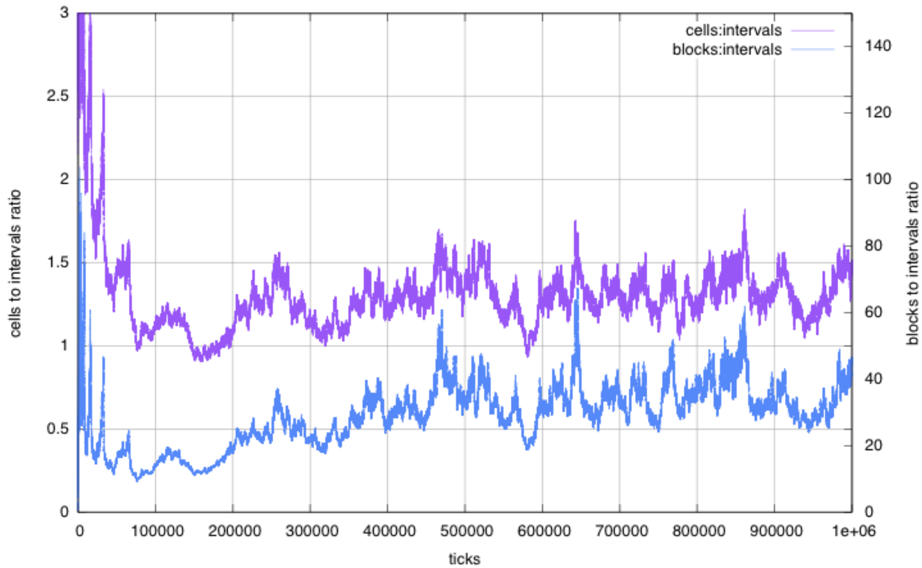

Figure 2: Binmaps work better for holey layouts. Notice the bottom just before tick 600,000; compare to Fig. 1.

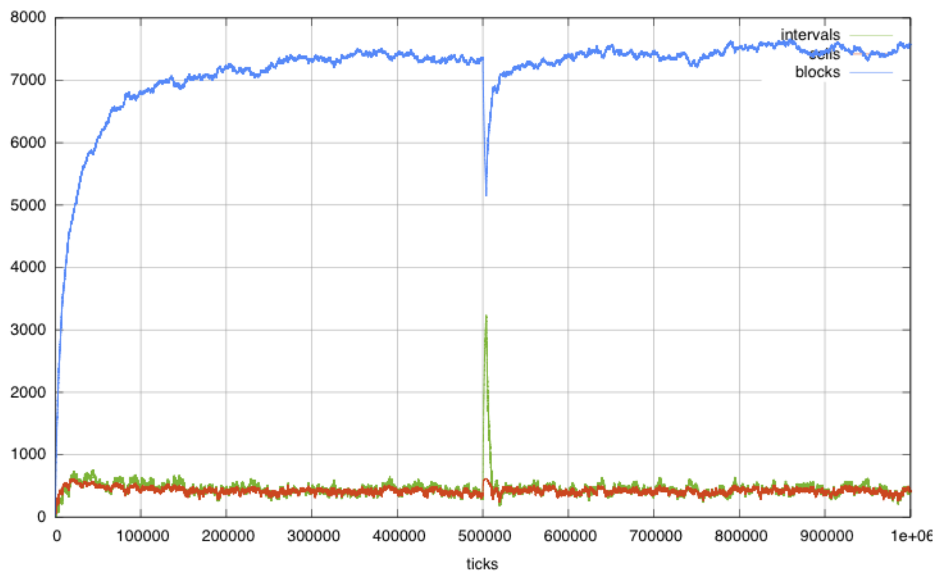

Figure 3: Simulated bad case: massive deallocation fragments the layout, explodes the extent tree.