HashFIT: Hashed-based File Index Table

Michael Bartling   Behzad Boroujerdian

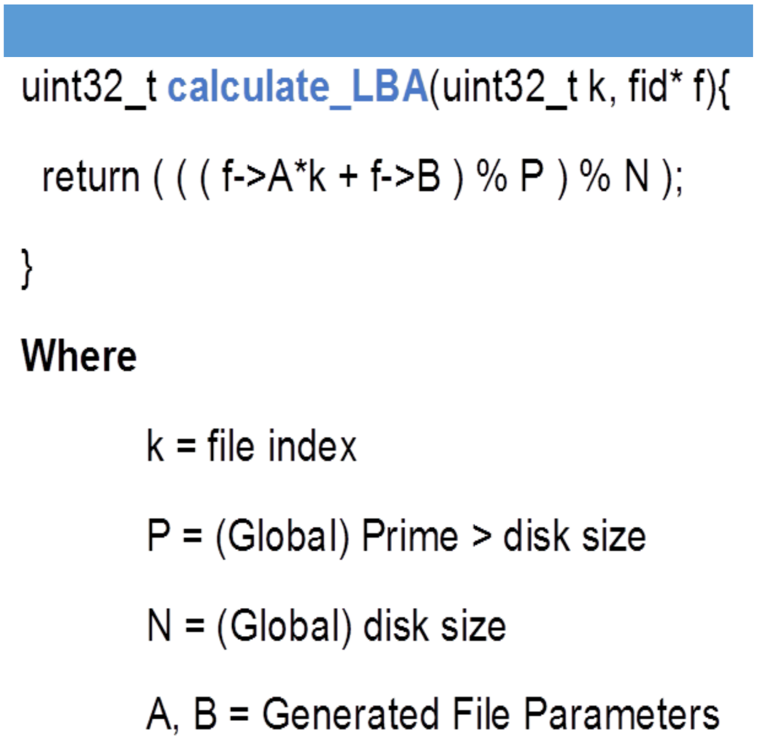
Abstract

This paper introduces HashFIT, a file allocation scheme for embedded platforms based on a class of universal hash functions. Unlike traditional index schemes, in HashFIT data block indexing is file-relative and the logical block addresses are computed from a files indices. In addition to simple code, hash based indexing has theoretically predictable performance which can be used for performance optimizations. To the best of our knowledge, this is the first attempt at a hash-based file system.

Introduction

We start with the class of universal hash functions described in [1]. We choose a universe of size P for the keys, where P is a prime greater than the largest allocation ID. Allocation ID is used rather than block ID since we may choose to index either individual blocks or groups of contiguous blocks. Since P is prime, by Fermat’s Little Theorem for every integer *a*, $a^p \equiv a \left(mod\ p \right)$ hence every unique group in the keyverse is expressible mod p. Consequently, any natural number described by a linear relationship can be expressed mod p as a result of arithmetic operations on rings. Finally, by constraining the keyverse to the disk allocation size, m, we arrive on the class of universal hashes described in [1].

$$\begin{cases} h_{ab}(k) = \left((ak+b)\, mod\ p \right) mod\ m \\ H_{(a,b)} = \{ h_{ab} : a \in Z_p^{¿} \wedge b \in Z_p \} \end{cases}$$



```
uint32_t calculate_LBA(uint32_t k, fid* f){
  return ( ( ( f->A*k + f->B ) % P ) % N );
}
```

Where

k = file index

P = (Global) Prime > disk size

N = (Global) disk size

A, B = Generated File Parameters

Where *a, b* are arbitrary random integers with b> 0. As long as (a,b) are chosen uniformly at random then the probability of collision between two unique keys and a single message is at most 1/m. However, in [1] the authors only considered a universe with replacement, but this does not make sense for systems with memory. We therefore extend the probability of collision to include a dependence on disk utilization.

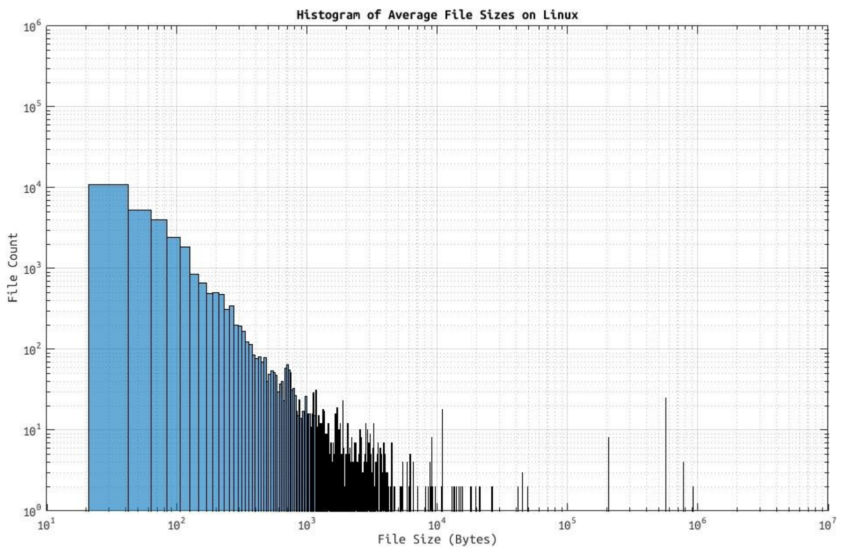


Figure 1 File statistics across Linux Machines

File System Model

We began building the HashFIT model in Matlab using file statistics collected on 7 Linux machines. The results are summarized in Figure 1. Note how most files are smaller than 16KB; this makes sense in a Linux environment given directories are 4KB as well as the large number of drivers and configuration files.

Next we modeled the HashFIT file system on a small 512 MB disk with 512 Byte blocks and 16 block contiguously-addressable clusters without collision avoidance. We chose the Mersenne Twister pseudo random number generator due to its speed and long period. On creation of a new file, A and B are chosen at random and the first logical block address is computed with the index *i = 1*. File sizes where chosen according to the distribution in Figure 1, and the memory allocation tested against various disk utilization levels (see Appendix for Collision heat maps). The probability of collision vs. disk utilization is shown in Figure 2. As expected the probability of collision is approximately linear with respect to disk utilization.

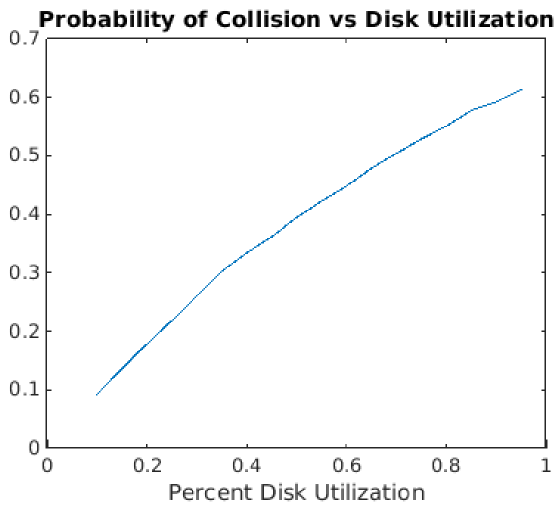


Figure 2

Implementation on the TM4C123gxl

We implemented a bare bones version of HashFIT on the TI TM4C123gxl microcontroller. Our real time operating system provides calls for file creation, opening, reading, and writing. Due to licensing limitations, our design was limited to a code size of 32kB in SRAM. Despite this limitation, we could create a file in linear time, open a file in approximately constant time, and append to a file in constant time.

The file system consists of a look up table containing the file coefficients (A and B), the files first index, and a files last index in addition to a data region. For a file j, its corresponding LUT entry is LUT[j] and its first index is generally 1 for low disk utilization. The first block of every file contains its meta-information. This includes file name, size, ID, etc. and is used by the file driver to calculate offsets into the file and predict collisions. Block size is 512 Bytes, and files are indexed by 8-block clusters. The SD card is a 16GB Kingston drive with a 4GB HashFIT partition and 4GB FAT32 partition. Due to time constraints we were unable to compare FAT32 performance to HashFIT. However, our FAT32 implementation generally takes 3 disk accesses per cluster allocation (on write append) whereas HashFIT only needs 1.

Conclusion

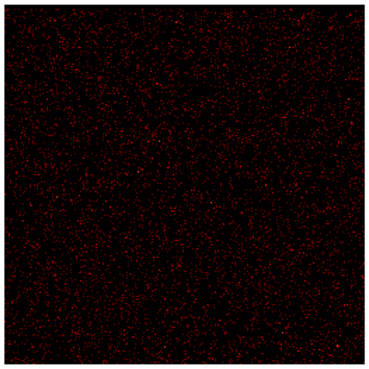
Our model has several limitations. First, it naively assumes file sizes based on Linux systems without considering read/write/execute permissions. The current HashFIT implementation performs best when reading and writing near the start and end of a file but suffers from the random access problem for intermediate clusters. The random access problem is as follows: given an arbitrary index and only the current disk utilization, how can we reliably access byte b in a file. We approach this problem by including meta-information such as number of index collisions, most recently used clusters, and the disk utilization for at a known index, in the first block of every file. Given different meta-information we can determine a cluster index within a predictable range thereby minimizing disk access.

Another limitation of our model is it does not account for directory entries nor a directory structure. For simplicity, we assumed a flat directory with fixed file permissions and relations. However, extending HashFIT to include directories might include structures similar to FAT32 or ext2. However, we imagined HashFIT would be most useful in a heterogeneous disk structure. HashFIT has optimum performance when reading from and writing to the end of a file, making it a strong candidate for log file writing/reading as well as small file reads and writes. In particular, driver modules can benefit from the high performance disk access when reading and writing to large streams. The OS will place streams and high integrity logs on a separate HashFIT partition accessible only by privileged devices and system calls, while the rest of the system runs in a traditional extN, FAT32, or NTFS formatted partition.

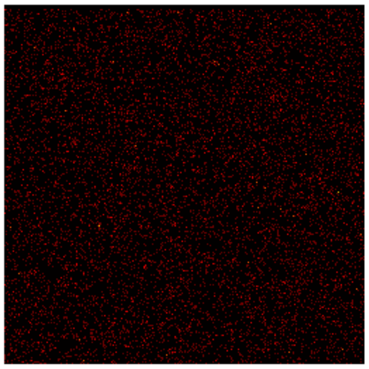
Bibliography

[1] T. H. Cormen, C. E. Lieserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Cambridge: MIT press, 2009.

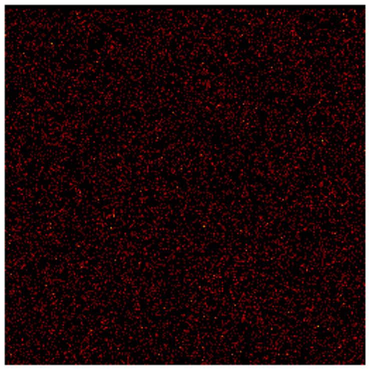
Memory Collisions\nFor Disk Util:0.1



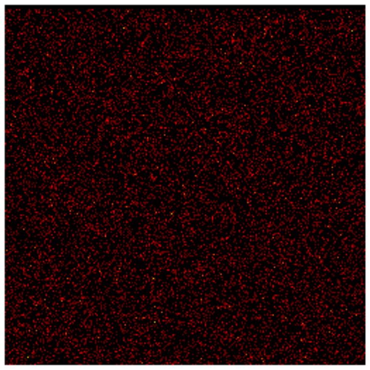
Memory Collisions\nFor Disk Util:0.15



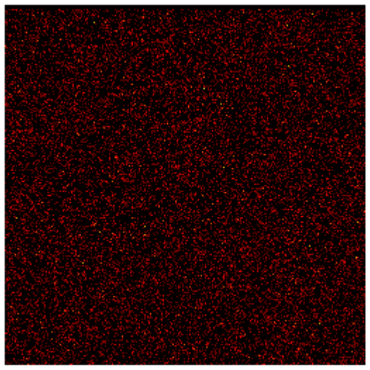
Memory Collisions\nFor Disk Util:0.2



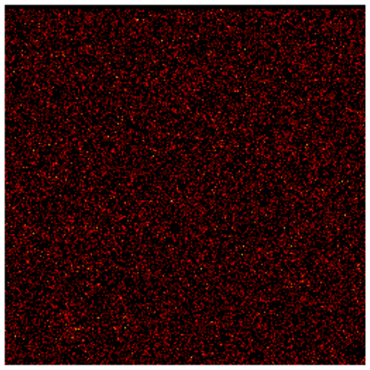
Memory Collisions\nFor Disk Util:0.25



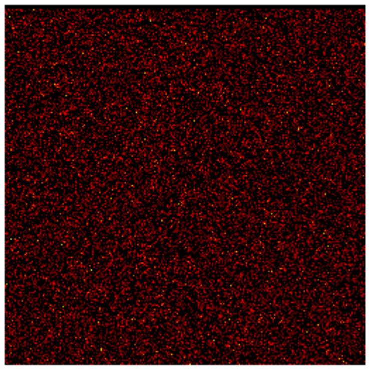
Memory Collisions\nFor Disk Util:0.3



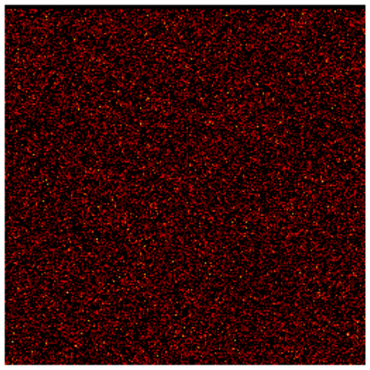
Memory Collisions\nFor Disk Util:0.35



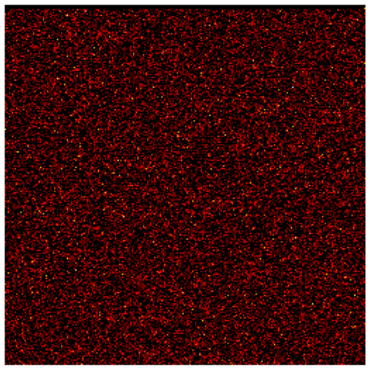
Memory Collisions\nFor Disk Util:0.4



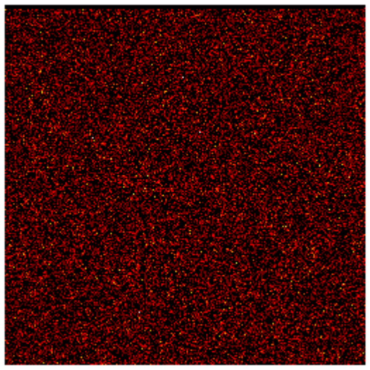
Memory Collisions\nFor Disk Util:0.45



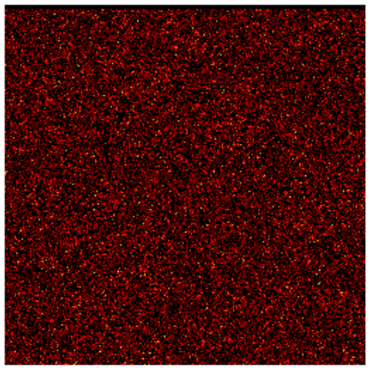
Memory Collisions\nFor Disk Util:0.5



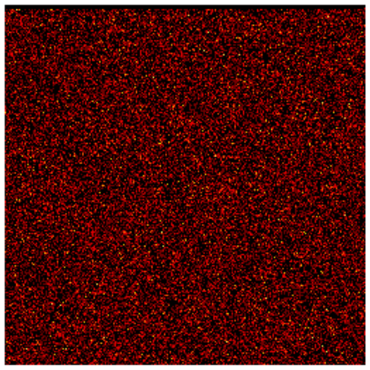
Memory Collisions\nFor Disk Util:0.55



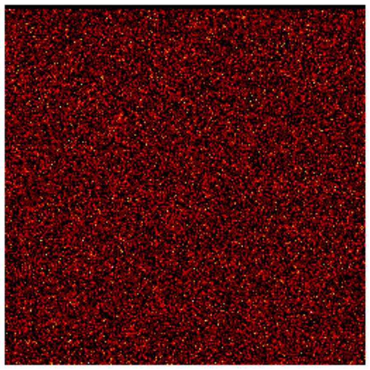
Memory Collisions\nFor Disk Util:0.6



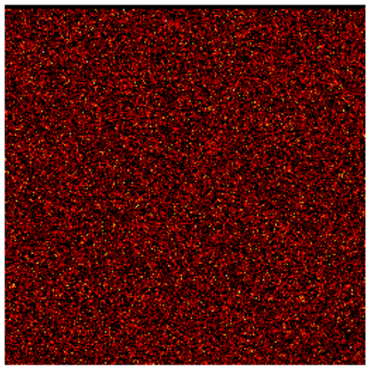
Memory Collisions\nFor Disk Util:0.65



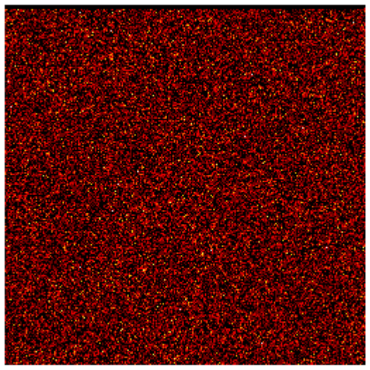
Memory Collisions\nFor Disk Util:0.7



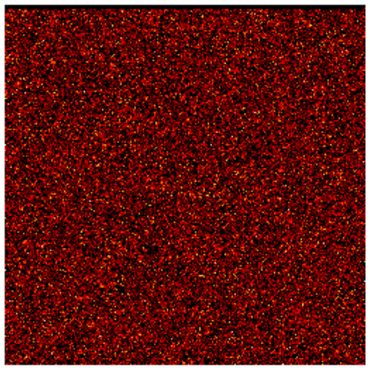
Memory Collisions\nFor Disk Util:0.75



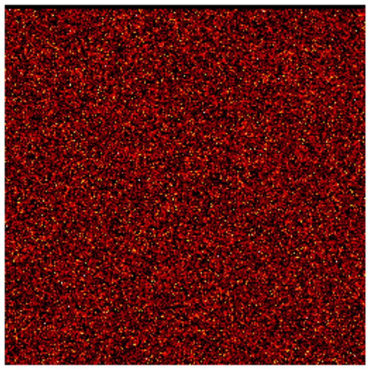
Memory Collisions\nFor Disk Util:0.8



Memory Collisions\nFor Disk Util:0.85



Memory Collisions\nFor Disk Util:0.9



Memory Collisions\nFor Disk Util:0.95