# IMPLEMENTING B* TREES IN THE EXTENDED FILE SYSTEM (XFS)

An Independent Study

Presented to the

Graduate Faculty of the

Computer Studies Division

Ateneo de Davao University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Information Technology

By

Wilfredo B. Badoy Jr.

ATENEO DE DAVAO UNIVERSITY

GRADUATE SCHOOL

MAY 2009

# TABLE OF CONTENTS

## ABSTRACT

As users demand higher disk capacities and larger single files, the need to address issues concerning speed is imperative. The fastest file systems today use B+ tree algorithms to arrange files on disk and this study aims to introduce B* tree structures into that type of file system. A comparison between two B+ tree-based file systems is made, one for standard B+ and one for B*. Random reads and writes were done and their speed measured. The measurements are in bytes per second. This interval data is then analyzed by using Paired t-test.



*Keywords:*

B+ tree, B* tree, random read, random write

# Chapter 1

# INTRODUCTION

## *1.1 Background of the Study*

According to Silberschatz, Galvin, and Gagne (2002), for most users the file system is the most visible aspect the operating system. It provides the mechanism for on-line storage of and access to both data and programs. It determines how many files can be created and how they should be placed in a disk.

Different file systems use different file allocation methods. The three major methods of allocating disk space are contiguous, linked, and indexed. Each allocation method has its advantages and disadvantages.

Contiguous file systems include UnixWare BFS and ISO 9660. (Filesystems-HOWTO: Introduction, n. d., Section 1.6) Contiguous allocation suffers from the difficulty of finding space for a new file. Determining the space needed for the file can be a major problem. (Silberschatz, Galvin, & Gagne, 2002)

The FAT and FAT32 file systems, which are widely used in Windows 98 and NT, are variations of linked allocation. It creates a table upon formatting where rows are directory entries pointing to the starting block of a specific file. Linked allocation can also have problems, one such problem stems from its sequential nature. To access the middle of the file may mean accessing every bit from the start until the middle is reached. Linked allocation also wastes space because it may use space for pointers instead of information. Another problem is reliability, a pointer may be lost due to a bug or failure,

which in turn will result in pointing linking to free-space or the wrong file. (Silberschatz, Galvin, & Gagne, 2002)

Indexed allocation solves the problem of inefficient direct access by bringing all the pointers to a single location. The popular Ext2FS uses a variation of the indexed allocation scheme. It uses direct and indirect pointers to access data. Indexed allocation suffers the same performance problem as linked allocation. (Silberschatz, Galvin, & Gagne, 2002)

Another data structure is now being implemented in various modern file systems. Among the file systems that use this data structure is Silicon Graphics Incorporated's Extended File System. This filesystem was chosen to be the experimental filesystem because of source code availability. This data structure is a variation of the indexed allocation. This is the B+ tree structure. The B+ tree is one the variations of the balanced tree. Balanced trees have the characteristic of having rapid random access. It accomplishes this by having all the nodes that point to data reside in the same level, so that the number of nodes accessed to find a particular item is small. The B-tree suffered from the difficulty of traversing data sequentially. The B+ tree later solved this, by having all keys reside in the leaves. This retains the rapid random access of the B-tree and adding rapid sequential access. (Langsam, Augenstein & Tenenbaum, 1996, p. 458)

To improve storage efficiency, the B* tree and the compact B-tree were introduced. The B* guarantees a minimum storage utilization of 67% is opposed to 50% by the B-tree. The figure is higher in actual practice. The compact B-tree on the other hand has the maximum storage allocation that reaches 98% to 99%. Unfortunately, there

is no known efficient algorithm to insert a key into a compact B-tree. (Langsam, Augenstein, & Tenenbaum, 1996)

The advantage of compression is that it enables more keys to be retained in the node, so the depth of the tree and the number of nodes required can be reduced. Traversing from root to leaf, would therefore require less time and an increase in seek speed should be evident. (Langsam, Augenstein, & Tenenbaum, 1996)

## *1.2 Statement of the Problem*

Can B* trees improve random read and write speed in XFS?

*Sub-problems:*

1. Can B* data structures be integrated into the XFS filesystem?
2. How can the XFS code support B* structures?
3. How can random read and write speeds be measured in XFS?

## *1.3 Objective of the Study*

The objective of this study is to be able to determine if there is an increase in random read and/or write speed when B* data structures are integrated into SGI's Extended File System.

*Specific Objectives:*

1. To be able to investigate how the Extended File System allocates inodes and files.
2. To be able to incorporate the B* trees into the Extended File System.
3. To measure read and write speeds of the XFS file system on different file sizes.
4. To be able to determine if the implementation of the B* trees will improve read and write speeds of the XFS file system.

## 1.4 *Hypothesis*

*Null Hypothesis*

Ho1: There is no significant increase in random read speed when implementing B* trees in XFS.

Ho2: There is no significant increase in random write speed when implementing B* trees in XFS.

*Alternative Hypothesis:*

Ha1: There is a significant increase in random read speed when implementing B* trees in XFS.

Ha2: There is a significant increase in random write speed when implementing B* trees in XFS.

## 1.5 *Significance of the Study*

Different computer systems have different mixes of reads and writes. For example, a study by Heath & Rouser (1995) showed the student file servers have a read/write ratio of 0.39 while administration file servers are higher at 1.0.

The knowledge of file read and write speeds will help in approximating the overall speed of the system using a particular implementation of the B+ tree structure, thereby helping in choosing which file system to use in database servers, client pc's, standalone computers, and others.

This study will give us a glimpse of how compacting keys in file systems using B+ tree structures affect random read and random write speeds.

## *1.6 Scope and Limitations of the Study*

This study solely involves the traditional B+ tree, and the more compact B* tree. Another possible implementation can be done by further compaction of the index keys of the B+ tree as is done in compact B-trees.

For the reason that XFS source code is readily available, it was chosen as the experimental file system. XFS source code is then modified to incorporate B* trees.

During test, files of different sizes are created, specifically from 512 megabytes to 2 gigabytes in graduations of 512 megabytes. Reads and writes of 512 kilobytes were then done and speeds were measured in bytes per second. The record size was fixed at 512 kilobytes for the reason that it is the experiment machine's processor cache size.

## *1.7 Definition of Terms*

Balanced tree – is an optimization of a tree which aims to keep equal numbers of items on each subtree of each node as to minimize the maximum path from the root to any leaf node.

B+ tree – is a balanced tree where the leaves are also linked sequentially that allows fast random access and sequential access to data.

B-tree – is a kind of balanced tree that can have more than two sub-trees in each node as opposite to a binary tree.

B*trees – a variant of the B+ tree where nodes are kept 2/3 full by redistributing keys to fill two child nodes then splitting them into 3 nodes.

File system – is a system for organizing files and directories, generally in terms of how it is implemented in the operating system.