Extμ: Using the Inode to Store File Content

Thawan Kooburat, Kevin Liu

Abstract

Hard disks are random access devices that can only exhibit good throughput if the workload consists mostly of large sequential accesses. As a result, application developers tend to avoid storing data in small files because it will degrade file system performance. For example, application specific parameters are usually stored in a single file with a complex format. This complexity can be avoided if the file system is better tuned to handle small files. In this paper, we present the extμ file system, which is a modified version of ext2 file system that has two major modifications to improve small file interactions. The first is to use the available space in the inode to store the content of a tiny file (less than equal to 128 bytes). The second is to change the pre-allocation policy to allow related small files to be stored adjacently to each other. In one of the workloads, extμ shows that it has almost 2.5 times the performance over other file systems. In addition, the size of the configuration parser module can be reduced by 50% when storing each configuration parameter as a file.

1 Introduction

Disk access times have made very modest improvements compared to other computer components such as the CPU. What has dramatically improved is disk storage capacity. Unlike other random access media, hard disks cannot be treated entirely as a random access device. Peak disk throughput can be achieved only if the workload is dominated by sequential accesses. File system designers use various techniques such as principle of locality [9] and batched write operations [1] to increase the file system throughput by amortizing the cost of initial seek time by the bandwidth for subsequent data transfers.

Unfortunately, working with small files still causes major performance problems because the cost of accessing and updating the metadata take up a large proportion of the total access time. Since small files make up a sizeable percentage of all files on UNIX-based operating system [10], it is worthwhile to investigate methods to improve interactions with this type of files.

Application designers tend to use a small database or a large configuration file to store all configuration parameters. Unfortunately, this causes the configuration parsing code to be unnecessarily complex. We argue that if the file system can handle small files better, then the application designer will have a strong incentive to store each parameter in a separate file. This will greatly simplify the configuration parsing code and perhaps even lead to increased application performance.

We propose two methods to improve a file systems' handling of small files. The first method involves using the available space in the inode to store the file content. This available space exists because many major operating systems allocate additional space for each inode to store extended attributes for fast access. By using this space to store the file's data, only one disk access is required to read or write to

files that can fit in its inode. The second method involves changing the existing pre-allocation policy to reduce the overhead of using small files. Existing policy is designed to prevent large files from interleaving with each other when created simultaneously. However, this also means that small files will not be stored adjacently on disk, leading to potential read/write penalties.

We implemented extμ by modifying the ext2 file system to confirm the benefits of these ideas. On normal usage scenarios, our file system performance will be equal to or slightly slower than that of the ext2 file system. However, when interacting with tiny files, our file system can perform two and a half times faster than other file systems. In addition, we also re-implemented Python's configuration parser to take advantage of extμ's small file performance. Each parameter is stored in a separate file, so there is no need for complex parsing logic. The size of this configuration parser module is 50% smaller than the standard Python configuration parser.

The rest of this paper is organized as follows: Section 2 presents how various file system handle small file workloads. Section 3 discuses the motivation behind our work. Sections 4 and 5 describe our design and implementation ideas. In Section 6, we show our evaluation methods. We talk about the limitations of the extμ file system in Section 7. And finally, we state our conclusions in Section 8.

2 Related Works

In this section, we will highlight the work done with existing file systems to improve small file interactions.

2.1 ZFS

ZFS [2] tries to take advantage of the principle of locality by compressing data using the LZJB algorithm. The compression allows additional data to fit into a block size, and thus improve I/O throughput. However, the cost is increased CPU usage for compression and decompression operations when writing/reading to the blocks. ZFS also uses the copy-on-write model to write files to disk. Similar to LFS [1], random writes of small files are grouped into large sequential writes. Finally, ZFS uses variable block size ranging from 512 B to 128 KB [11] to minimize internal fragmentation.

2.2 C-FFS

In Ganger and Kaashoek's C-FFS [3], they dealt with small files using embedded inodes and explicit grouping. With embedded inodes, the inodes for small files (less than one block in size) are stored in the directory with the corresponding name. This change removes one level of indirection to improve performance.

Explicit grouping maximizes the use of the principle of locality. Since there is a high cost of accessing a single block, it makes sense to access several blocks even if some will not be used. Therefore, data blocks of multiple small files for a given directory are allocated adjacently and moved to and from the disk as a unit.

2.3 Linux ext2

The Linux ext2 [4] file system is a descendent of the FFS, which made numerous improvements over the original UNIX file system to handle sequential workloads. One of the key improvements is sequential access, specifically interactions with file metadata. Related data are grouped in the same cylinder block, while unrelated ones are spread across different blocks. The new arrangement greatly reduces the rotational and positioning costs associated with file/metadata accesses. This is especially true for small files, where the cost of metadata access dominates the overall file access time.

2.4 NTFS

NTFS is a file system on the Windows operating system. 12.5 percent of disk space on each partition is preserved for the Master File Table (MFT). This table stores file metadata and special information for the file system. Everything, including directories, has its own entry in the MFT table. The size of a MFT entry (MFTE) ranges from 1KB to 4KB, and is dependent on the size of the data block. This entry stores file information as attributes, and includes the file name and pointers to data blocks containing the actual data. The MFTE can also store the actual file content if the data is small enough to fit into a single entry. With this layout, there is no need for a separate data block read request if the file is sufficiently small.

3 Motivation

One of the many differences between Windows and UNIX-based operating systems is how configuration files are stored. On Windows machines, all OS and application configurations are stored in the Windows registry [7] that acts like a small database. On the other hand, UNIX-based operating system stores configuration parameters in a separate file. The sizes of these configuration files vary depending on how many parameters are stored. Although a centralized configuration file can be a blessing at times, it can also complicate the parsing code that reads in these configuration files. This is especially true if complex formats such as XML are used. In addition, the whole configuration file must be read in from disk even if only a single parameter is required.

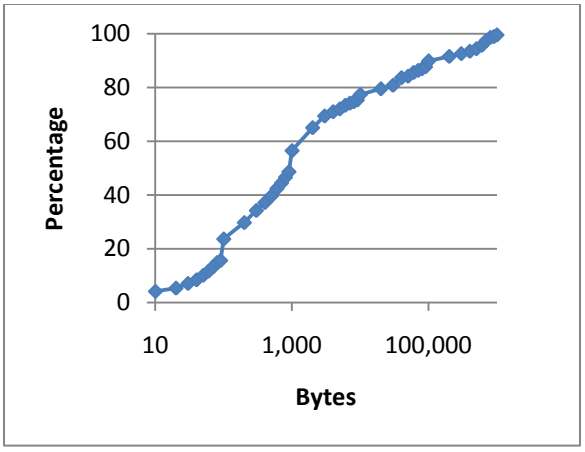

Figure 1: Size distribution of configuration files in /etc folder of CentOS 5

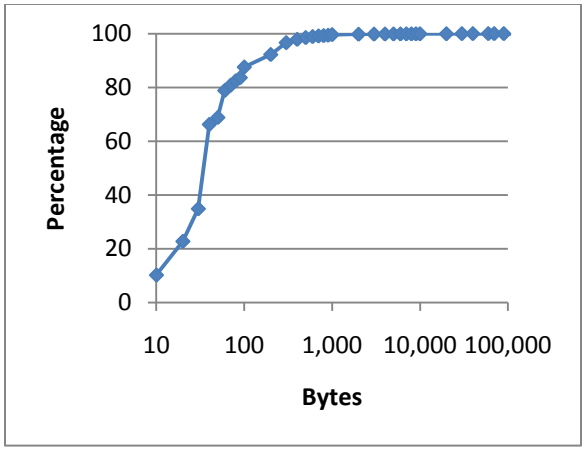

Figure 2: Size distribution of Windows 7 registry's branch

In Figure 1, we evaluated the configuration files for UNIX-based systems. We looked at the configuration file sizes for a newly installed CentOS 5 file system, and found that the majority of the files were quite large since they stored numerous parameters in order to reduce file system overhead.

To investigate how applications interacted with their configuration parameters if they are maintained in a database-like environment, we turned to the Windows Registry. This registry provides a central location to evaluate the configuration parameters of all Windows applications. We were interested in evaluating the size of these parameters and proceeded to split each branch of the registry into individual files. In Figure 2, we found that 87% of the files were less than 100 bytes in size after the split. Based on this information, we were confident that 128 bytes would suffice in holding a large percentage of the configuration parameters.

Based on the above observations, we propose splitting up the large configuration files in the UNIX-based systems into numerous small ones, with each option consisting of one file. This approach will simplify the parsing logic and allow each configuration to be modified individually. However, the astute reader will point out that this will greatly increase the application startup time since it has to read in numerous files. To minimize this shortcoming, we will modify our file system such that there is little to no penalty in handling small files. This setup allows the application to store configuration parameters in the file system as if they were database entries.

4 Design

4.1 Storing Data into the Inode

Current on-disk data structure size of inode of all ext file system is 128 bytes [8]. However, many Linux distributions such as Debian, Ubuntu and Fedora are shipped with default parameters that will create an ext file system with 256 bytes for the inode size. This is done to improve the performance because extended attributes can be stored in the available space adjacent to each inode. Typically, the extended attributes are used by the kernel or SELinux to store additional ACL information.

We believe that for a small file, extended attributes are rarely used. Thus, using these available spaces in the inodes to store the file content can improve the performance of accessing small files in the extμ file system. The performance improvement is the result of reduced number of disk accesses. Any read/write to a small file can be achieved by accessing just the inode disk block.

In the rest of this paper, we will refer to the available space adjacent to each inode as inode data area. With the standard configuration, the inode data area size is 128 bytes. As a result, the definition of a tiny file in this paper is a file whose size is smaller than 128 bytes. However, it is also possible to create an ext file system with inode size larger than 256 bytes, so that we have more space for inode data area. Nevertheless, our initial finding shows that 128 bytes is large enough to hold the majority of configuration

parameters within the inode data area. In addition, further increases to the inode size may lead to wasted space and degraded system performance.

We pick the ext2 file system as the base for our implementation. Ext2 is not capable of using inode data area to store extended attributes, so it simplifies our implementation. In our design, every file created in the file system will be treated as a tiny file. Customized read/write operations will access inode data area to service their request. When file grows larger than 128 bytes, it will be converted to a normal file and the standard ext2 file operations will handle all the subsequent requests.

4.2 Modifying the Pre-allocation Algorithm

Both ext2 and ext3 use a reservation-based allocator [13, 14] to pre-allocate data block during a write operation. The main mechanism is to pre-allocate at least 8 consecutive data blocks for each file when it needs to grow. These pre-allocated blocks will be freed if they are still unused by the time the file closes. This is used to prevent file fragmentation especially when large files are created simultaneously. However, this policy has an adverse effect on tiny files. In particular, configuration files will suffer degraded read/write performance since they might be located 8 blocks apart if they were created in parallel. Ext4 uses multiple allocators [13] to handle various file sizes more efficiently.

We propose a simple modification to the existing ext2 allocator. We noticed that even if a tiny file grows beyond 128 bytes, it rarely exceed one disk block (4KB). This is especially true for configuration files where each parameter usually has a fixed size value that does not change over time. Thus, we should disable pre-allocation for the first allocated block. The next allocation request will be accompanied by the standard pre-allocation logic. This setup will allow tiny files that are created simultaneously to be stored adjacently on disk while preserving the standard pre-allocation policy for larger files.

5 Implementation

5.1 Modifying the Data Structures

For tiny files, we expand the ext2 inode data structure to 256 bytes by utilizing existing space reserved for extended attributes. We can determine if a file is tiny simply by checking that it owns zero data blocks. Based on this design, there is no explicit need to modify the existing ext2 inode data structure. Thus, we can use mke2fs to create extμ file system without any modification.

To interact with tiny files, we developed customized read/write operations that are inserted into the file operations table which exist only within the in-memory inode data structures. Therefore, the in-memory inode data structures are the same size as those of ext2 inodes.

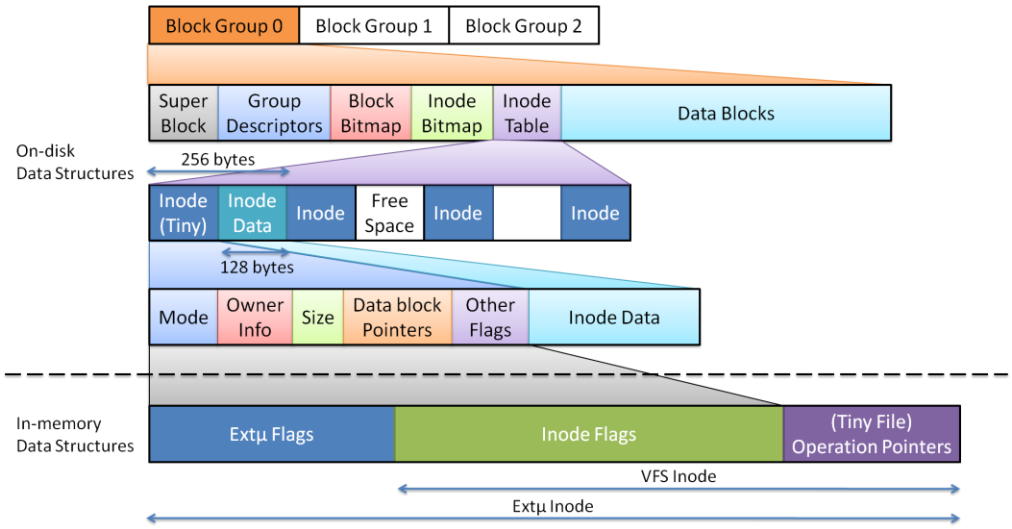

Figure 3: Data structures for the extμ file system

5.2 Modifying the Read/Write Operations

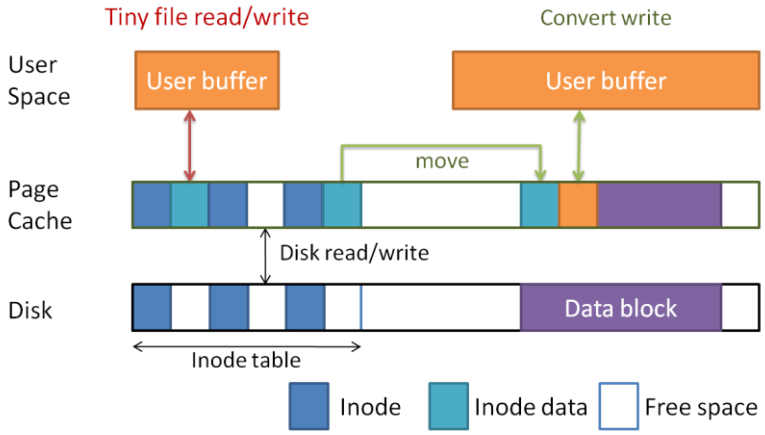

Figure 4: Read and write operations of extμ file system

We will describe the implementation by following its read and write operations. In order to access a file, a filename will be translated into an inode number and extμ will locate its corresponding inode disk block. Extμ will then issue a read request and store the disk block into a page cache. Then, it will initialize the in-memory inode and copy the inode data from the on-disk inode residing in the page cache. At this point, extμ will check if a file owns any data blocks. If a file does not own a block, it is classified as a tiny file with customized file operations assigned to the inode; otherwise, normal ext2 file operations will be used. During read and writes, the kernel will invoke a corresponding method from the file operations pointer associated with each inode. In case of the tiny file, both read/write will check if the file is still considered a tiny file (not owning any data blocks). If so, then the read/write operations will obtain the file data directly from inode data area, which is already in the page cache.

If the operations detect that the file is not longer a tiny file (owning at least one data block), then they will remap the file operation pointers and invoke the default ext2 methods to service the request.

However, the result of write operation might grow the file beyond 128 bytes. In that case, extμ will allocate a new data block and copy existing data from the inode data into it. Then, it will switch the file operation pointers and invoke the normal write operation to write the user data into the newly allocated data block.

Implementation complexity lies in the fact that ext2 uses VFS-provided file operations for most of its file operations. As a result, we have to customize the VFS's read/write-related operations and put them into extμ. Linux kernel provided numerous method calls to facilitate the file system development process. These methods include validating the user buffer, and locating/paging-in a disk block. Example usages of these operations in other file systems seem to be the best documentation on how to correctly invoke these file operations.

5.3 Modifying the Block Allocator

Ext2 reservation-based allocator uses a goal block to hint where to allocate new blocks. For files that already own a block, the goal block will point to a block next to the already allocated blocks. However, if the block is not available, the allocator will reserve eight consecutive blocks in memory and refer to them as a reservation window. The reservation window prevents other files from allocating a block in the range that will lead to fragmentation and will be freed when the file closes.

However, this might have an adverse effect on the configuration files performance as described previously. Thus, we modify the allocator logic to disable block reservation for the first block allocation of the file. This change improves the file conversion performance when the file content does not grow beyond one block by allowing multiple files to be stored adjacently on disk. We achieved this by modifying just one conditional statement in the current ext2 block allocator's logic.

6 Evaluation

The goal of this paper is to create a file system that can better support read/write operations on tiny files without reducing the performance during general file system usage. We also wanted to explore tiny file benefits at the application level. To achieve these goals, we divided the evaluation into three parts. Firstly, we used micro-benchmark to measure the execution times for basic file operations. Secondly, we used FileBench [6] to simulate application workloads. And finally, we developed a configuration parser that stores each parameter in a separate file. This configuration setup will further evaluate the tiny file support of the extμ file system.

We carried out our tests on a machine with the following specifications: Pentium 4 3.0 GHz CPU, 1GB DDR2 RAM, and 80 GB 7200 rpm SATA hard disk. Our test machine ran Ubuntu 9.10 (Linux Kernel 2.6.31) as its operating system. Extμ is developed as a kernel module that can be loaded without recompiling the entire kernel. In all test cases, we compared the performance of extμ with that of ext2, ext3, and ext4 [12]. Each file system is created with standard configuration parameters on a 10 GB

partition. Unless otherwise specified, we report the results from the ext versions that do not contain the modified block allocator.

6.1 Micro-benchmark

We developed a customized application to measure execution time. Prior to benchmarking, the application creates 1,000 input files (128 bytes each) and flushes the page caches, dentries and inodes to ensure that input files are fetched from the disk. The buffer size of all write operation is 128 bytes, which means that the files are not converted into normal files except when the O_APPEND option is set. The program reports the sequential access times averaged over 1000 iterations. The values reported are the minimums for three test runs and can be seen in Table 1 below.

Table 1: Execution of file operations using various open's flags

Flags	**O_RDONLY**		**O_WRONLY**		**O_TRUNC | O_SYNC**		**O_WRONLY | O_APPEND**		
FS Type	open	read	open	write	open	write	open	write	close
extu	62.0	1.3	60.7	1.7	70.8	224.4	53.1	31.6	1.7
ext2	69.6	1250.3	64.2	654.0	76.2	927.4	-	-	-
ext3	68.0	858.7	68.5	691.4	88.7	1967.6	-	-	-
ext4	105.0	202.3	105.2	178.0	134.2	28691.4	-	-	-

Read and write operation of extμ is fast because the page cache that contains the data is already loaded as part of inode access. In case of synchronous writes, extμ is faster than ext2 because only the inode block needs to be flushed to the disk. In the case of O_APPEND, the result shows the cost of converting a tiny file into a normal file.

Table 2: Execution time when operating with 1KB files

Flags	**O_CREATE**			**O_RDONLY**		
FS Type	open	write	close	open	read	close
extμ (modified block allocator)	68.1	20.9	1.6	64.7	162.9	1.4
extμ	68.4	22.1	1.5	64.9	935.2	1.4
ext2	73.0	19.4	1.7	53.7	636.2	1.4

Note: extμ-pre is version of extμ that implements the modified block allocator.

Here we show the effect of our modified pre-allocation policy. The test scenario is similar to the previous experiment. The buffer size of all file operation is 1K, so this experiment shows the performance of normal file operations. Additionally, files created by O_CREATE run are used as an input to O_RDONLY run. Thus, the input files for this experiment are generated as if 1,000 files are created in parallel.

The result shows that with the modified block allocator, reading back the same data is four times faster. This is because the each data block stays adjacent to each other compared to eight blocks apart in normal ext2 block allocator. However, this experiment also shows that there is an overhead in extμ when

operating with normal files. This suggests that there might be some implementation inefficiencies in our file system.

6.2 FileBench

FileBench is a file benchmark tool that contains standard test suites and also allows programmers to easily develop test scenarios by using the FileBench Workload language. The standard workloads that we used are web server and web proxy scenarios. For the web server workload, a file set of 10,000 files is generated and 1000 threads are started to simulate client requests. Each thread randomly selects 10 files for read operations. Then, it will write to a single large file to simulate logging. Web proxy workload is similar to the web server but only 80% of file set is generated to simulate cache misses. Additionally, a single write is performed to a random file out of every five reads. The result reported is the maximum of four test runs with each test running for 60 seconds.

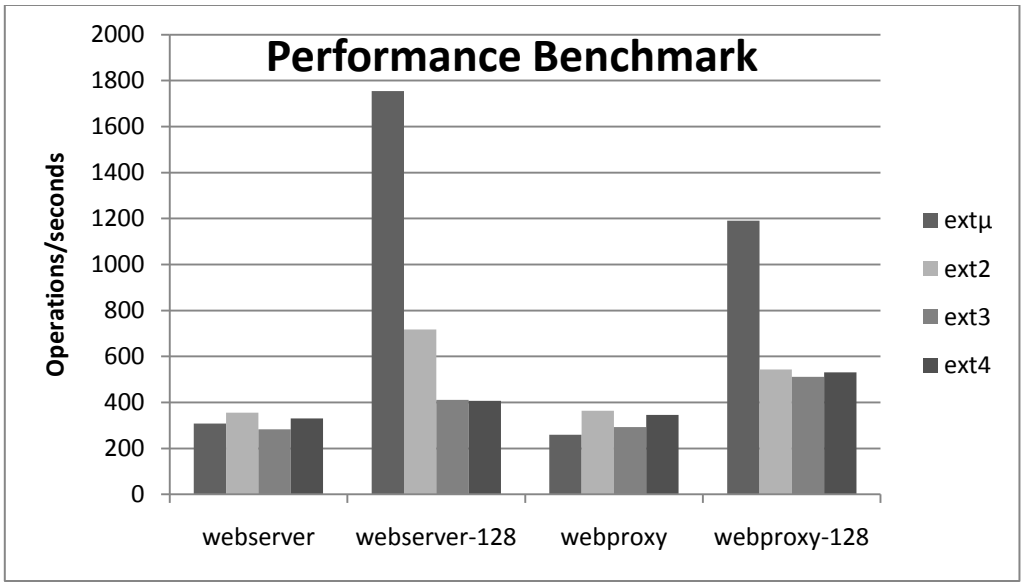

Figure 5: Performance of standard scenarios across different file systems. Each scenario is performed with default file size (16K) and 128 bytes.

By using default file size (16K) for both workloads, extμ shows a slight performance loss between itself and ext2 that corresponds to the previous micro-benchmark experiment. Nevertheless, the benefit of extμ is highlighted here when we change the file size to 128 bytes. Its performance is almost 2.5X compared to the ext2 web server workload.

However, the performance benefit of extμ is noticeable only when the file could fit into the inode data area and the working set is larger than system main memory. To confirm the validity of the second requirement, we reduced the number of files in the server workload from 10,000 to 1000 and observed no performance improvements between extμ and ext2.

6.3 Configuration Parser

One application for the extμ file system is configuration files. Traditionally, these files are stored in the INI file format that follows a basic structure. The basic structure for an INI configuration file consists of parameters and optional sections. Every parameter has a name, value pair and often delimited by the "=" character. Parameters may optionally be grouped into sections for organizational purposes [15].

Applications typically centralize their configurations to a single file. For large complex applications such as Apache and Mozilla Firefox, these configuration files can become quite large. One disadvantage with such a storage approach is that the whole file must be read into memory and flushed back to disk even if a single parameter is changed. Another disadvantage is the code required to parse these INI configuration files can become quite complex. Since there is no definitive standard for configuration files, different applications may use different syntaxes and delimiters for their parameters and sections. As such, developers who write configuration parsers must resort to writing long complex code involving regular expressions to correctly parse and extract the desired values.

Using a system optimized for small files such as extμ, we propose a different solution to store configuration files. Instead of grouping all configurations into a single large file, we propose storing each name/value pair to a single file. The option name will be used as the file name, and the option value will be stored as the content for the file. Subdirectories can be introduced inside the configuration directory to incorporate the concept of sections. Finally, each configuration parameter inherits metadata and ACL from its file, gaining access to these functionalities at no additional costs.

Using our decentralized approach, the two main disadvantages with the traditional configuration files can both be eliminated. When configuration changes are made, only a small subset of the configuration files will be read into memory and subsequently flushed back to disk. In addition, the code complexity for configuration parsers will also be reduced as large portions of the parsing code can be eliminated.

However, there are some desired functionalities that are lost when storing the configuration parameters in this manner. By decentralizing the parameters to numerous files, manual inspection/modifications of these parameters becomes slower since the user may have to navigate to several subfolders to change the desired parameters. In addition, applications that read in all configuration parameters at once may find the process slower using our approach. However, we argue that application does not need to read in all parameter at once since there is no parsing overhead and parameters can be accessed on demand.

6.3.1 Configuration Parser Complexity

We rewrote the ConfigParser that is part of the Python 2.6.5 release. By removing large portions of the code dealing with parameter and section parsing, we were able to reduce the complexity of the module. We used CLOC [16] to calculate the actual LOC of both modules and found that our module (364

LOC) is almost half the size of Python's module (669 LOC). Code reduction, without compromising functionality, is always desired as it directly leads to less software bugs.

6.3.2 µConfigParser Performance

To evaluate the performance of working with our proposed approach, we benchmarked the read and write performance of working with a large file versus a group of small files. For our benchmark test, we compared setting and reading back a small number of configuration parameters using Python's default ConfigParser and our µConfigParser. We also took the minimum of our measurements to further abate the effects of background processes. To simulate a large configuration file, we used a file that spanned three data blocks. The benchmark program only accesses five configuration parameters in all test cases. The results of our benchmarking are summarized in Table 2 below.

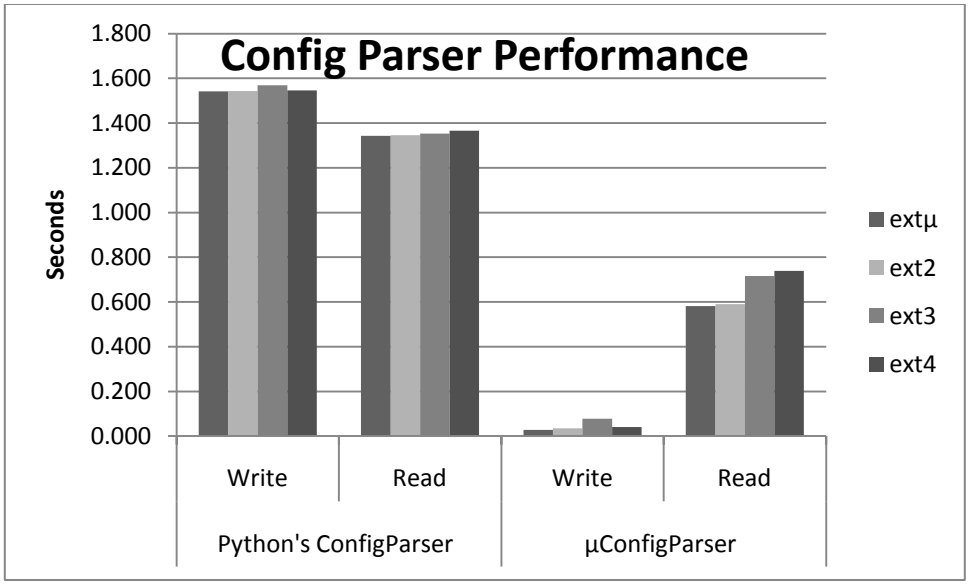

Figure 6: Configuration parser benchmarking

To confirm the integrity of our setup, we notice that the amount of time it takes to read/write to a large file is fairly consistent across the different file systems. Expectedly, the performance between the different file systems is more prominent when working with a group of small configuration files.

The amount of time it takes to set five options in the large configuration file is up to 55 times faster in extµ, and up to 20 times faster in ext3 than setting them in the large configuration file. Furthermore, amount of time it takes to read in these options is about 2 times faster when working with the small files as compared to a single large file.

Based on the above results, we believe that applications that only check their configurations during startup will not likely see much improvement using our configuration setup since all parameters must be read in. However, applications that periodically checked their configurations file for changes could potentially benefit from using our file system and configuration setup.

7 Limitations

Currently, our file system does not support the `mmap` operation on a tiny file. However, we can add this operation by converting the tiny file into a normal file and invoking the default `mmap`. We also did not modify the fsck utility to accommodate our file system layout. Fsck may therefore report file system errors for the non-zero sized files without any data blocks. Nevertheless, since every file without a data block is considered a tiny file in extμ, there should be enough information in the inode for the modified fsck to operate. For the O_DIRECT option, it is impossible to avoid caching on tiny files because its data is co-located with the inode block that is always loaded into the page cache. Therefore, we treat O_DIRECT as O_SYNC in our file system.

8 Conclusion

This work has shown that the inode data area can be utilized to improve the performance of tiny files. Additionally, the inode data area is already available in many major Linux distributions. The complexity of utilizing this space to store data is low compared to the improved performance on tiny file operations. Thus, it is worthwhile to build this kind of functionality into existing file systems. In addition, we also showed that application-level code complexity, such as the configuration parser, could be greatly reduced when tiny files are properly supported by the file system.

9 Acknowledgements

We would like to thank Professor Barton Miller for his guidance and advice throughout the course of our project. We would also like to thank Tony Nowatzki and Chaman Singh Verma for reviewing our paper. We would like to thank these individuals for their contributions to our project.

10 References

[1] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," ACM Transaction on Computer Systems, vol. 10, pp. 26-52, 1992.

[2] J. Bonwick and B. Moore. ZFS – The Last Word in File Systems. Available: http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/zfslast.pdf

[3] G. R. Ganger and M. F. Kaashoek, "Embedded inodes and explicit grouping: exploiting disk bandwidth for small files," presented at the Proceedings of the annual conference on USENIX Annual Technical Conference, Anaheim, California, 1997.

[4] R. Card, et al., "Design and implementation of the second extended filesystem," presented at the Proceedings of the First Dutch International Symposium on Linux, 1994.

[5] Microsoft NTFS Technical Reference. Available: http://technet2.microsoft.com/windowsserver/en/library/81cc8a8a-bd32-4786-a849-03245d68d8e41033.mspx

[6] R. McDougall. FileBench. Available: http://www.solarisinternals.com/wiki/index.php/FileBench

[7] M. Russinovich. Inside the Registry. Available: http://technet.microsoft.com/en-us/library/cc750583.aspx

[8] D. Bovet and M. Cesati, Understanding The Linux Kernel: Oreilly & Associates Inc, 2005.

[9] M. K. McKusick, et al., "A fast file system for UNIX," ACM Transactions on Computer Systems, vol. 2, pp. 181-197, 1984.

[10] M. G. Baker, et al., "Measurements of a distributed file system," presented at the Proceedings of the thirteenth ACM symposium on Operating systems principles, Pacific Grove, California, United States, 1991.

[11] Solaris ZFS Administration Guide. Available: http://docs.sun.com/app/docs/doc/819-5461

[12] A. Mathur, et al., "The new ext4 filesystem: current status and future plans," presented at the Proceedings of the Linux Symposium, 2007.

[13] A. Kumar, et al., “Ext4 block and inode allocator improvements” presented at the Proceedings of the Linux Symposium, 2008.

[14] M. Cao, et al., “State of the Art: Where we are with the Ext3 filesystem” presented at the Proceedings of the Linux Symposium, 2005.

[15] “Cloanto Implementation of INI File Format”. Available: http://www.cloanto.com/specs/ini

[16] A. Danial, “CLOC: Count Lines of Code”. Available: http://cloc.sourceforge.net/#Overview