Proc. 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM:PDS'03)
at the Third International SIAM Conference on Data Mining, San Francisco, CA, May 2003

Understanding Filesystem Performance for Data Mining Applications

Bouchra Bouqata, Christopher D. Carothers, Boleslaw K. Szymanski, and Mohammed J. Zaki
{bouqab,chrisc,szymansk,zaki}@cs.rpi.edu

Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180, U.S.A.

Abstract

Motivated by the importance of I/O performance in data mining efficiency, we focus this paper on analyzing data mining performance across different file systems. In our study, we consider three of the most popular filesystems available under the Linux distribution. These include: Ext2[3] (non-journaled), Ext3[16] (journaled), and Reiser[12] (journaled). We conclude that full data and metadata journaling (Ext3) appears to dramatically slow down the performance of a data mining engine. For I/O intensive cases, data mining execution time was double when run over Ext3 as compared to Reiser. We found that the write speed of Ext3 was 35 times slower than Reiser, and file address references display only a short-range dependence in data mining, indicating high degree of locality in data references.

Keywords

Performance Mining, File Systems (Reiser, Ext2, Ext3), Data Mining, Association Rules, Frequent Itemsets, Journaling

1 Introduction

The work presented in this paper is conducted under the PerfMiner project at RPI which aims at developing the PerfMiner engine for performance mining of large-scale data-intensive next generation distributed object applications, such as a simulation of the entire World Wide Web or modeling of protein folding. We will not address a detailed description of the Perfminer engine in here because it is out of the scope of this paper.

The ultimate goal of PerfMiner is the *robust performance* of complex distributed object systems which enables them to adapt to and optimize for a dynamic critical path as well a dynamic run-time environment when executed on today's high-power "grid" computing platforms. PerfMiner will detect opportunities for both the critical path optimization and the beneficial application of speculative execution using data mining on a persistent, run-time generated database.

Data mining places heavy demands on both computational and I/O resources. Typically the search for patterns and models is over very high dimensional spaces (with hundreds to thousands of attributes per object), that may have temporal and spatial attributes along with the regular observed attributes in a given application, and over very large databases (with millions to billions of objects).

Motivated by the importance of I/O performance in data mining efficiency, which will affect directly the Perfminer performance, we focus in this paper on analyzing data mining performance across different file systems. While there have been a number of filesystem performance studies conducted, such as [5, 13, 10, 11, 7, 12], we would like to understand how data integrity techniques, such as *vvjournaling*, affect data mining performance. To the best of our knowledge, no study has examined this aspect before.

In our study, we consider three of the most popular filesystems available under the Linux distribution. These include: Ext2[3] (non-journaled), Ext3[16] (journaled), and Reiser[12] (journaled). Other filesystems, such XFS[6] and JFS[4] where omitted since those filesystems are not yet considered to be part of base, stable Linux 2.4.X distribution and required an external kernel patch to make operational.

We chose the VIPER algorithm (Vertical Itemset Partitioning for Efficient Rule-extraction) [14] for association mining, because of its intensive I/O operations and memory usage which will enable us to evaluate well the performance of the different filesystems. Moreover, VIPER exhibits both read and write rather than read only access.

The reminder of this paper is organized as follows. An overview of the three filesystems is presented in Section 2. Viper[14], the data mining engine used in this study is discussed in Section 3, followed by the performance results in Section 4 and conclusions in Section 5.

2 Filesystems

Linux filesystems can be categorized into the following classes. First, there are device based filesystems which typically store data on hard disks. Example filesystems in this category include Ext2 [3], Ext3 [16], JFS [4], Reiser [12], and XFS [6]. Resting on top of these device file systems are network based filesystems that allow machines not having physical access at the device layer to provide seamless access to a filesystem across a local-area network. Example filesystems in this class include NFS [9], Coda[8] and Samba[15]. Finally, Linux supports special filesystems such as */proc*, which is an read-only

filesystem that provides user-level access to system information and statics.

A journaling file system extends the capabilities of device based filesystems by ensuring data integrity at the device layer. This is accomplished by immediately storing updates to a files' metadata (and data depending on the filesystem used) to a serial log on disk before the original disk blocks are updated. When the system crashes, the recovery code analyzes the log and attempts to clean up inconsistent files by replaying the log file. In this paper, our performance study will center on three different filesystems: Ext2, Ext3 and Reiser, each of which has different capabilities and performance properties. These filesystems where chosen in this experimental study because of their popularity, stability and wide availability.

2.1 Ext2

Ext2 is a block based file system, which speeds up the execution of I/O operations by keeping metadata and data temporally allocated in RAM. Inconsistency of the whole system may be caused by a system crash if all modified data are not written to disk. To recover, a file system check (i.e., `fsck`) is performed which scans the entire storage device. This device could be 100s of gigabytes to even terabytes in size, making a scan of such a storage system extremely time consuming. By providing journaling capability, this costly operation can be avoided for most system crashes.

2.2 Ext3

Ext3 is Ext2 file system enhanced with journaling capabilities. Ext3 shares all structures with Ext2 file system, and adds transactions capabilities to Ext2. Ext3 adds two independent modes on top of the writeback mode present in Ext2: a transaction and a logging mode. These modes can be described as follows:

- **Writeback mode:** In writeback mode, Ext3 doesn't do any form of data journaling at all (unlike XFS, JFS, and Reiser, which do journaling). Despite the fact it could corrupt recently modified file, this mode should give, in general, the best Ext3 performance.
- **Ordered mode:** Ext3 by default journals metadata only, but it logically groups metadata and data blocks into a single unit called a transaction. When it's time to write the new metadata out to disk, the associated data blocks are written first. In general, this mode performs slightly slower than writeback but significantly faster than journal mode.
- **Journal mode:** it provides full data and metadata journaling. All new data is written to the journal first, and then to its final location. In the event of a crash, the journal can be replayed, bringing both data and metadata into a consistent state.

Clearly, journal mode provides the highest degree of reliability. Here, not only filesystem structural data or metadata, but also file content stored in RAM, is logged to the journal. However, even with data journaling loss of content is still possible.

In Linux, journaling support is provided by the journaling block device layer (JBD). The JBD is not Ext3 specific. It was designed to add journaling capabilities to a block device. The Ext3 filesystem code informs the JBD of modifications that it performed (called a transaction). The journal handles the transactions start and stop, and in case of crash, the journal can replay the transactions to put the partition on a consistent state faster.

Because Ext3 reuses the data structure and filesystem layout of Ext2, Ext3 is backwards compatible. This interoperability is provided by reserving a special place within the Ext2 file structure for the journal log. From the point of view of Ext2, the journal log appears as any other file. Additionally, it is possible to have multiple file systems sharing the same journal. The journal file's job is to record the new contents of file system metadata blocks while the file system is in the process of committing transactions.

2.3 Reiser

In contrast with other filesystems, Reiser has been designed completely from scratch and reports high-performance on small files. It employs the use of fast balanced trees (B+ trees) (one per filesystem) to organize file system structures such as file state data or directories (directory item). Reiser also eases artificial restrictions on filesystem layouts by dynamically allocating filesystem structures as needed rather than creating a fixed set at filesystem creation. This helps the filesystem to be more flexible to the various storage requirements that may be thrown at it, while at the same time allowing for some additional space-efficiency. Reiser, unlike Ext2, doesn't allocate storage space in fixed 1kB or 4kB blocks. Instead, it can allocate the space of the exactly needed size. In order to increase performance, Reiser is able to store files inside the B+ tree leaf nodes themselves, rather than storing the data somewhere else on the disk and pointing to it. Thanks to that the small file performance is dramatically increased because the file data and the statistics-data information are stored right next to each other and can normally be read with a single disk I/O operation. In addition, Reiser is able to pack the tails together, saving a lot of space. In fact, a Reiser with tail packing enabled (the default) can store 6 per cent more data than the equivalent Ext2 filesystem. However, this technique would increase external fragmentation, since the file data is now further from the file tail. For this reason, Reiser tail packing can be turned off, allowing the administrator to choose between good speed and space efficiency, or opt for even more speed at the cost of some storage capacity.

3 Data Mining Engine: VIPER

Association rule discovery is one of the most important data mining tasks; it aims at uncovering all frequent patterns among

transactions composed of data attributes or items. Results are presented in the form of rules between different sets of items, along with metrics like the joint and conditional probabilities of the antecedent and consequent, to judge a rule's importance.

The problem of mining association rules was introduced in [1]. Let $\mathcal{I} = \{1, 2, \cdots, m\}$ be a set of items, and let $\mathcal{T} = \{1, 2, \cdots, n\}$ be a set of transaction identifiers or *tids*. The input database is a binary relation $\delta \subseteq \mathcal{I} \times \mathcal{T}$. If an item i occurs in a transaction t, we write it as $(i, t) \in \delta$. A set $X \subseteq \mathcal{I}$ is called an *itemset*. The *support* of an itemset X, denoted $\sigma(X)$, is the number of transactions in which it occurs as a subset. An itemset is *frequent* if its support $\sigma(X) \geq$ *minsup*, where *minsup* is a user-specified minimum support threshold.

An *association rule* is an expression $A \xrightarrow{p} B$, where A and B are itemsets; it measures the dependence of B on A. The *support* of the rule is $\sigma(A \cup B)$ (i.e., the joint probability of a transaction containing both A and B), and the *confidence* $p = \sigma(A \cup B)/\sigma(A)$ (i.e., the conditional probability that a transaction contains B, given that it contains A). A rule is frequent if the itemset $A \cup B$ is frequent. A rule is *confident* if $p \geq minconf$, where where *minconf* is a user-specified minimum threshold. The problem of mining association rules is to find all rules that satisfy the minimum support and minimum confidence thresholds. Applications include cross-marketing, attached mailing, catalog design, loss-leader analysis.

The computationally challenging step in mining such rules is to find the frequent itemsets, since rules can be easily generated from them [2]. There have been several efficient algorithms proposed for frequent itemset mining, differing mainly in their search strategy and database layout[1, 2, 14, 17]. In a horizontal layout [1, 2], the database is organized as a set of rows, with each row representing a transaction in terms of the items that are present in the transaction. In a vertical layout [14, 17], each item is associated with a column of values representing the transactions in which it is present. The latter format appears to be a natural choice for the association-rule mining's objective – discovering correlated items – in addition to contributing to fast and simple support counting (reduction of the effective size, compact storage, better support for databases, and asynchronous computation of frequent itemsets).

3.1 VIPER Algorithm

For our performance evaluation, we use the VIPER algorithm (Vertical Itemset Partitioning for Efficient Rule-extraction) [14] for association mining. VIPER is especially designed for very large datasets, since it performs its own memory buffer management. It writes out patterns and data to disk when memory is exhausted, and reads in the information later as required. Thus, VIPER is a good representative of a large-scale data mining algorithm. It stores data in compressed bit-vectors and integrates a number of novel optimizations for efficient bit-vector generation, intersection, counting and storage. VIPER provides good performance for large databases since it scales up linearly with database size.

VIPER uses a vertical bit-vector format for representing an item's occurrence in the tuples of the database. A bit at position i is set to 1 if the item occurs in transaction i. The bit-vector is stored in a compressed form, taking advantage of the sparseness that is typically exhibited in large databases in which, typically, an item occurs in some fraction of transactions, so that most of the entries in the bit-vector are 0's. Consider a database with the following six transactions over the items space $\{A, B, C, D, E\}$: $T_1 = ABDE, T_2 = BCE, T_3 = ABDE, T_4 = ABCE, T_5 = ABCDE, T_6 = BCD$. The bit-vector for A is 101110, for C it is 010111, and so on. Support of longer itemsets can be found by bit-wise AND operations on their subsets' bit-vectors. For example, to compute the support of the set AC we AND the bit-vectors of A and C to get 000110. To compute support of ACD we perform AND operations on the bit-vectors of its subsets, AC and AD, and so on.

In general, to compute the support of an itemset of length i, VIPER performs AND operations on the bit-vectors of any of its $i/2$ length subsets that cover all its items. For instance, assume we know the frequent itemsets of size 2 along with their bit-vectors. To compute the support of $ACDE$, we can intersect the vectors of AC, CD, and DE. Another possibility is to intersect the vectors of AC, AD, AE, or even just AC, DE, and so on. These are all examples of covering subsets of $ACDE$ (among several others). VIPER chooses the covering subsets in decreasing order of their supports.

VIPER is a multi-pass algorithm, wherein data (in the form of compressed bit-vectors) is read from and written to the disk in each pass. In the first pass, the frequent single items are found by scanning the original horizontal format database, and vertical bit-vectors are created for each item and stored on disk in a common file. In the second pass the horizontal dataset is scanned for a second time and all frequent sets of size 2 are identified by direct counting. From pass 3 onward, VIPER uses bit-vector ANDing for support counting.

In each pass the bit-vectors to be read from disk and written to disk are identified and the corresponding read/write operations are performed, depending on the amount of main-memory available. For memory management, VIPER keeps a page in memory for the bit-vector of each itemset of interest. That is, a page is maintained for each itemset to be read and each itemset to be written. The bit-vectors of i-length itemsets are read from disk and are intersected to find the support of all covered itemsets of length up to $2i$. Thus VIPER counts all candidates of size k, with $i \leq k < 2i$ in a single pass and determines which of these are really frequent. When the page buffer for an itemset being counted fills up the page is written out to a common file. All the pages associated with each individual bit-vector are chained together using a linked list of pointers. The frequency computation progresses until all frequent sets (of all possible lengths) have been found.

There are several note-worthy points of VIPER which may impact its performance on different filesystems:

- Support counting in VIPER involves simultaneous counting of several levels of candidates in a single pass via intersections of bit-vectors. In pass i, VIPER counts candidates of size i to $2i-1$, thus it counts itemsets of size 1 in pass $i=1$, sizes $2,3$ in pass $i=2$, sizes $4,5,6,7$ in pass $i=3$ and so on.
- While the candidates of size i are determined from the *frequent* itemsets in pass $i-1$, candidates of length l are determined by taking all pairwise combinations of *candidates* of sizes from i to $2i-2$. Thus, the set of potential candidates in VIPER is a superset of the candidates that would be generated if always frequent sets were used to generate new candidates. The size of the new candidate set cannot be determined in advance, and it may happen that at a lower minsup threshold one would generate fewer candidates than at a higher minsup value. This impacts the amount of buffer space used for active itemsets.
- VIPER writes intermediate results to disk, which may appear as an overhead. However, it can improve its performance since bit-vectors keep on shrinking as the passes increase, significantly speeding up the subsequent mining passes. Also, bit-vectors from previous passes that are no longer relevant to the remainder of the mining process are deleted, resulting in space reduction.

4 Performance Study

We have conducted an evaluation study of the performance of a data mining algorithm on several file systems. In particular, we compared the performance of VIPER [14] on three Linux file systems, namely Ext2, Ext3 and Reiser.

4.1 Experiment Setup

Our experiments were conducted on a dual processor 400 MHz Pentium-II system with 512 MB of RAM and 4, 9 gigabyte Western Digital, 7200 RPM, SCSI-II hard disks. The operating system was Linux, version 2.4.17 configured with the Ext2, Ext3 and Reiser filesystems. One of the four hard disks was used for the base operating system and swap space. The other three disks were given single partitions and a different filesystem. /dev/sdc1 was given Reiser, /dev/sdd1 was given Ext3 and /dev/sde1 was given Ext2. Thus, the underlying hardware was identical for each disk as well as directory structure on each disk was the same. So, the only difference among the three is the filesystem structure, thus making a true "apples-to-apples" comparison possible.

Our experiments cover a range of data mining workloads. Four different data sets were generated using the technique described in [2] with increasing sizes. This technique attempts to mimic the customer purchase behavior seen in retailing environments. The parameters used in the generator and their default values are described in Table 1.

Different minimum support thresholds were used for every data set as shown:

- **T20I8 (T=20 and I=8):** 0.002, 0.0025, 0.003, 0.004, 0.005, 0.0055, 0.006, 0.0065, 0.007, 0.0075
- **T30I8 (T=30 and I=10):** 0.002, 0.0025, 0.003, 0.004, 0.005, 0.0055, 0.006, 0.0065, 0.007, 0.0075
- **T40I10 (T=40 and I=10):** 0.004, 0.005, 0.0055, 0.006, 0.0065, 0.007, 0.0075, 0.0085, 0.0095, 0.01.

We note that the VIPER algorithm manages its own internal buffers and thus it writes or reads consistent buffers at 256 byte strides. This size was experimentally determined to provide the best overall performance. Thus, the filesystems are being accessed for medium size write and read operations.

4.2 Viper Performance

In this first series of experiments, we compare the performance of the VIPER data mining engine running on the different filesystems for each of the data sets. The performance results for the T20I8 data set is shown in Figure 1. The execution time in seconds is shown as a function of support, which ranges between 0.0020 to 0.0075. We observe that for the smaller levels of support, VIPER running over Ext3 takes considerably longer than VIPER running over Reiser and Ext2. In the 0.0020 support case, VIPER takes almost twice as long to execute when run over Ext3 as it does when run over either Ext2 or Reiser. We attribute this performance difference to additional journaling overheads incurred by Ext3 for providing full data integrity and not just journaling the filesystem metadata, as Reiser does. Additionally, we believe Reiser is able to compete with Ext2 despite its journaling overhead because of its compact filesystem structure.

However, as the support level increases, the performance gap narrows because of the actions taken by the data mining engine. Typically, when support is high, the data mining engine has less work to do, since the search is effectively broader. That is to say, smaller support is the amount to "find a needle in a haystack". With a reduction in the amount of searching, the amount of disk I/O performed is reduced which results in the data mining engine shifting from being I/O-bound to being more compute-bound. Thus, the overall execution times of VIPER running over all three filesystem is substantially reduced and converge to similar values.

The exception to this case is when the level of support allows the VIPER algorithm to make a large jump in the itemset search space. Recall, that the algorithm will jump from i to $2i$ frequent itemsets. If a level of support is such that the search can be pruned early, the algorithm will do so. This results in a situation where for small increments in support, the run-time performance will vary by a substantial amount.

We observe similar phenomena for both the T30I10 and T40I10 data sets, as shown in Figures 2 and 3 respectively. First, we observe a more pronounced performance difference. For the T30I10 data set, the execution time is almost 2.5 time longer for Ext3 than for either Ext2 and Reiser in the 0.0040

Parameter Symbol	Parameter Meaning	Default Value
N	# of Items	1000
T	Mean transaction length	10
L	# of frequent itemsets	2000
I	Mean frequent itemset length	4
D	# of transactions	100K

Table 1: Parameters used in data mining workload generations.

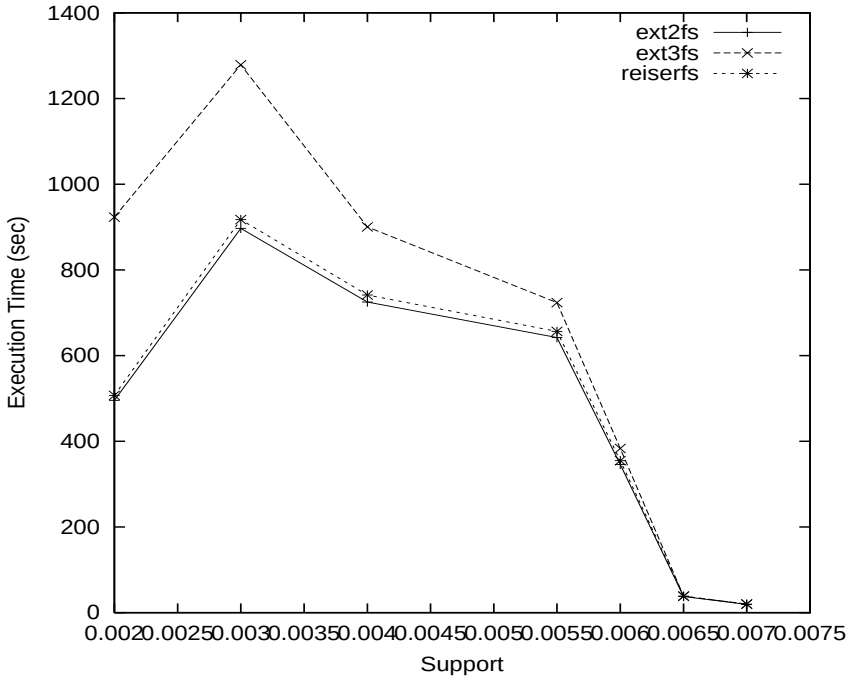

Figure 1: VIPER performance of T20I8 data set under Ext2, Ext3 and Reiser.

support case. Similarly, in the 0.0040 support case, the execution time is 2 times longer for Ext3 than either Ext2 and Reiser for the T40I10 data set. Additionally, the same "zig-zag" execution time is observed as support is increased followed by steep decrease in execution time for high supports. With the decrease in execution time, the performance gap between Ext3 and Reiser/Ext2 narrows.

As previously indicated, we attribute the overall performance difference between Ext3 and Reiser to the level of journaling provided. In the next series of experiments, we take a closer look at the I/O behavior of VIPER running under Ext3 and Reiser.

4.3 I/O Performance: Ext3 versus Reiser

In this series of experiments, we trace all `read` and `write` operations for the T30I10 data set with a support level of 0.0040 for both Ext3 and Reiser. Each trace contain approximately 600,000 read operations and 1,100,000 write operations. Thus, there are above twice as many writes as reads for this data set and level of support combination.

For each I/O operation, we capture (i) the wall clock time when the operation occurred, (ii) the file address being referenced, and (iii) the duration of the operation. The timing information is captured using the `rdtsc` machine instruction which records the internal processor cycle counter. The time returned from this operation represents the number of machine cycles that the system has been running since it was last booted. Because of the clock rate, the accuracy of this timer is $1/400,000,000$ seconds plus the memory access time.

In Figure 4, the write overhead measured in clock cycles for each write operation is shown as a function of wall clock time (this is the direction in which time flows) and file address for VIPER running Ext3 and Reiser respectively. We observe here that the write operation overhead for Ext3 is much greater and displays much higher degree of variance than Reiser, as shown by the large range of write times in y-axis.

Looking at the data more closely, the mean time to write is 538,861 clock cycles for Ext3 versus 15,379 for Reiser. Thus, the data mining engine will spend 35 times the amount of time performing write operations when Ext3 is running when compared to Reiser. Additionally, the standard deviation is 10,525,592 clock cycles for Ext3 versus 251,142 for Reiser.

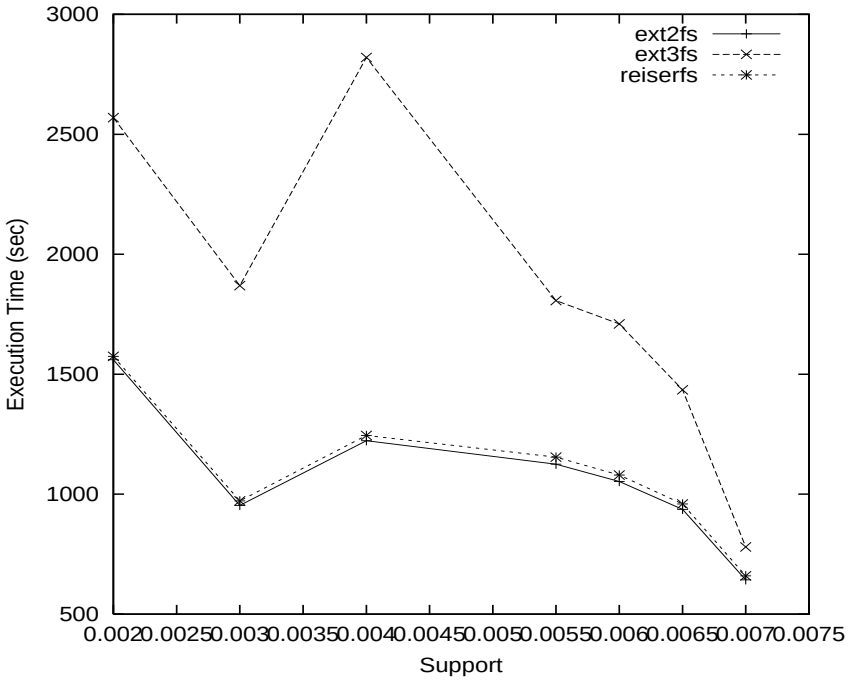

Figure 2: VIPER performance of T30I10 data set under Ext2, Ext3 and Reiser.

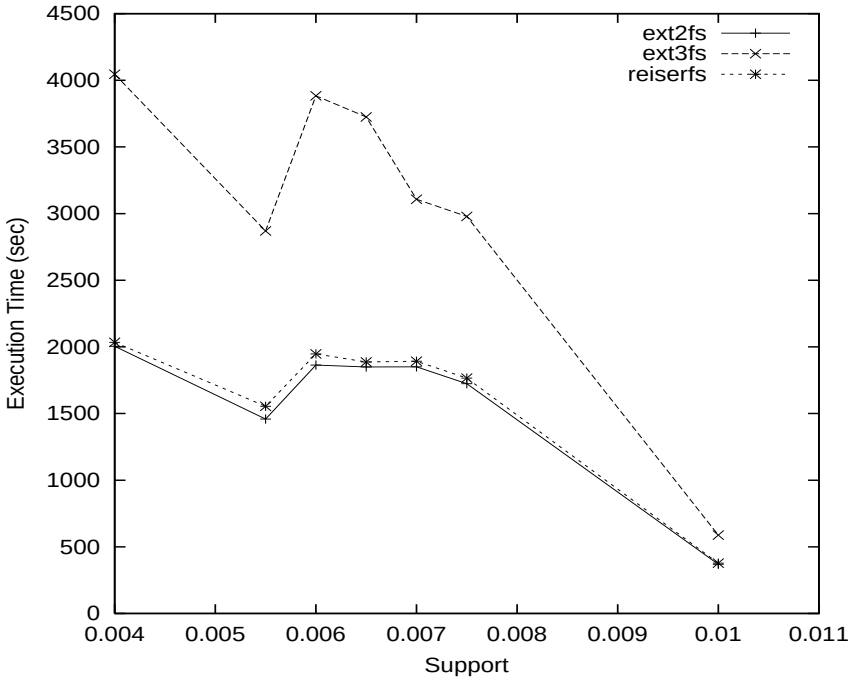

Figure 3: VIPER performance of T40I10 data set under Ext2, Ext3 and Reiser.

So, clearly, the data journaling overhead is exacting its toll on Ext3 performance.

Now, as one would expect, journaling should not perturb read performance. Figure 5 shows the read overheads in clock cycles for each read operation as a function of wall clock time and file address for VIPER running Ext3 and Reiser respectively. We observe little difference in average read performance. The mean number cycles to perform a read is 2867 for Ext3 and 2930 for Reiser.

However, we do see a higher degree of variance displayed by Reiser. The standard deviation for Reiser is 36,026 cycles as compared to only 1,350 cycles for Ext3. We attribute this phenomenon to the structural differences between the two filesystems. In particular, recall that Reiser employs a B+-tree to arrange filesystem objects. While this structure provides fast access to small files, for larger files, the read op-

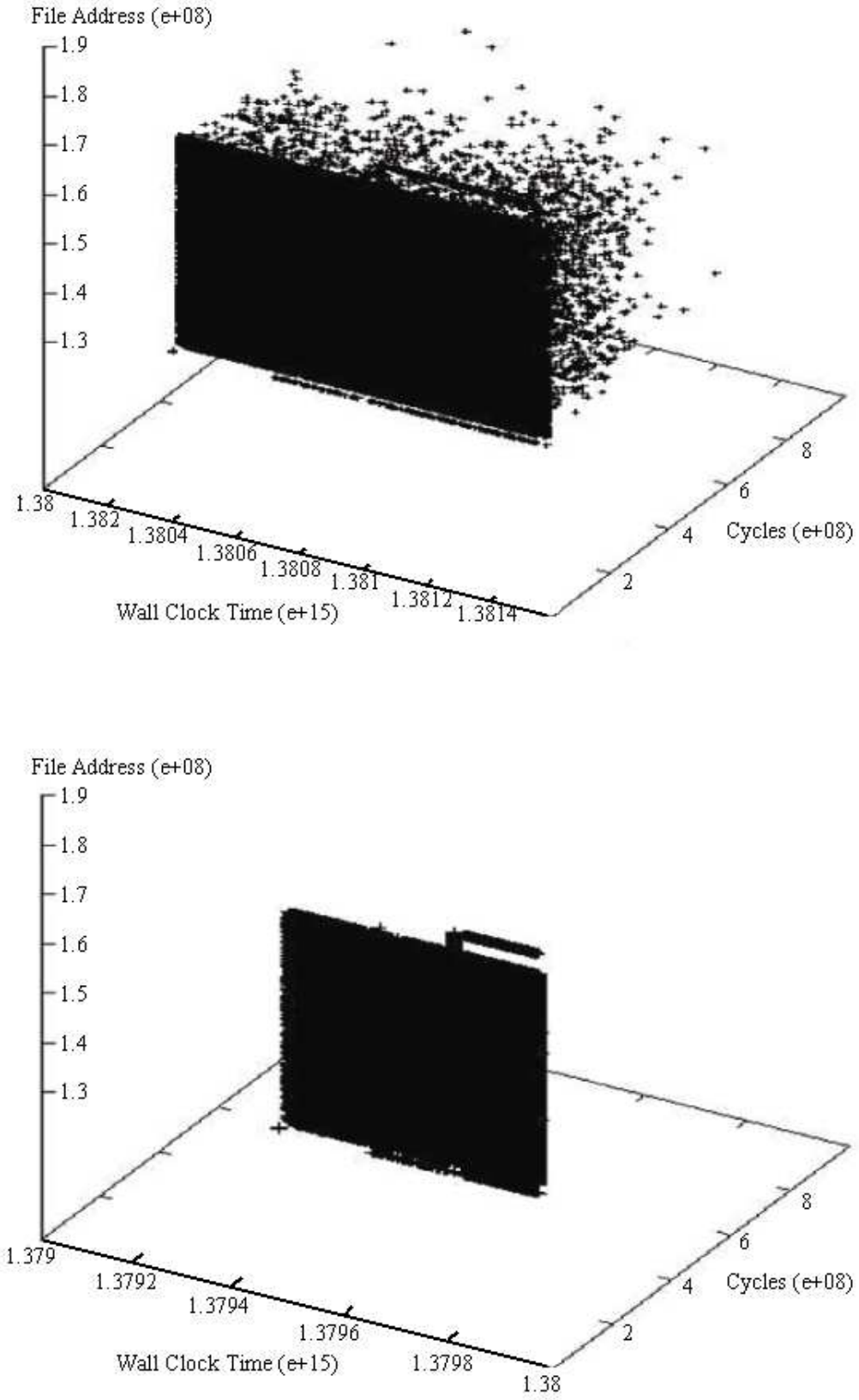

Figure 4: VIPER write overheads measured in CPU clock cycles on Ext3 (top) and Reiser (bottom) as a function of wall clock time and file address.

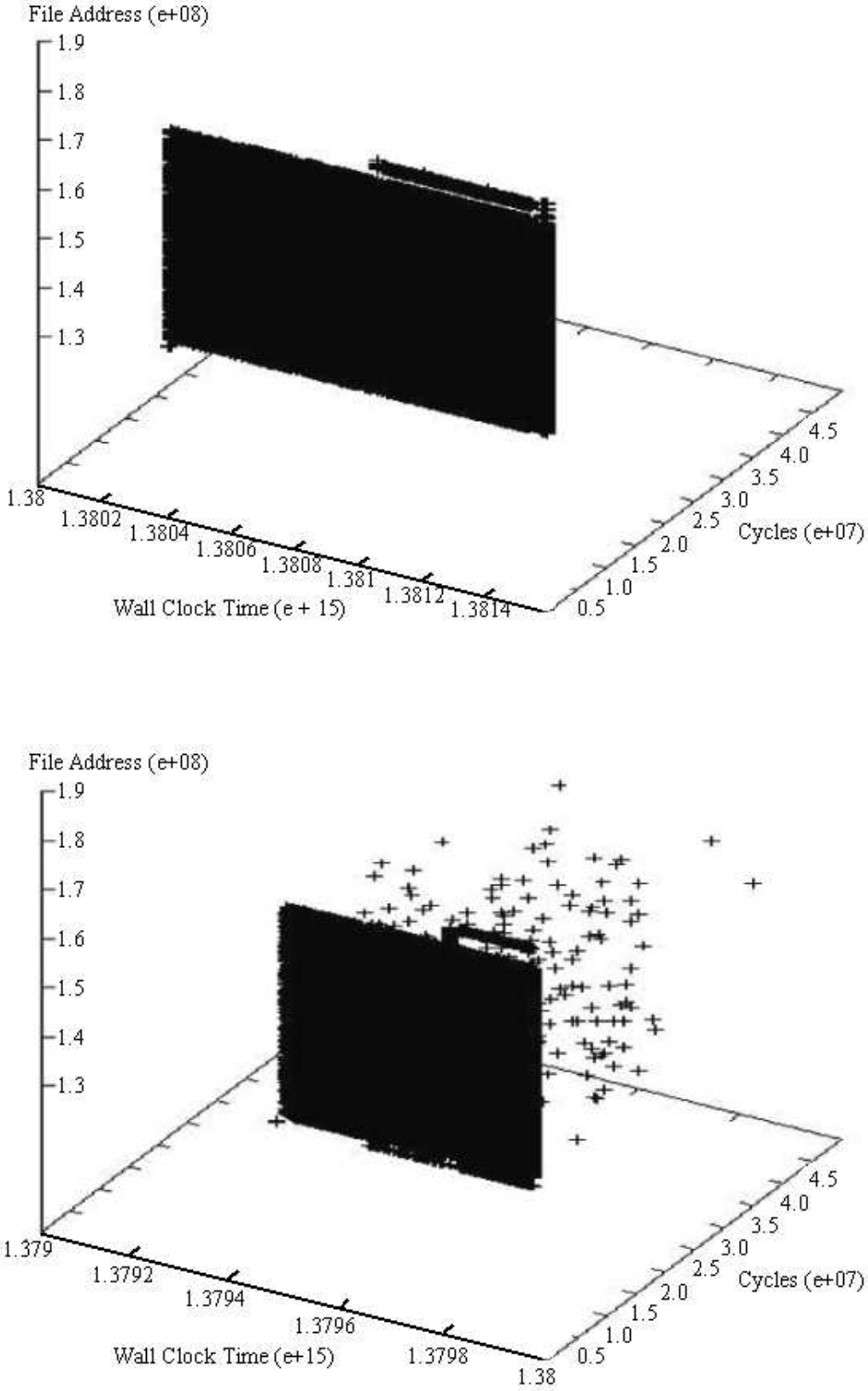

Figure 5: VIPER read overheads measured in CPU clock cycles on Ext (top) and Reiser (bottom) as a function of wall clock time and file address.

eration cost appears to vary depending on the address within the file being accessed. This overhead could come from either accessing metadata for block placement on disk or the arrangement of blocks on the disk. Making the picture more complex is how each filesystem utilizes the Journal Block Device (JBD) and other cache data structures. At this time, the primary source of the differences is unclear and further investigation is required.

Additionally, we observe from the reference traces that access to the file appears to have some underlying structure. We suspect that disk references are relatively memory-less because of the pruning done at each level of the data mining search algorithm. To determine this performance characteristic, we need to check for the long-range dependences in the address traces which is a work in progress.

5 Conclusion

In this paper, we have evaluated the performance of a data mining algorithm, VIPER, under three different file systems: Ext2, Ext3 and Reiser. VIPER represents a class of data mining engines that are used for mining large datasets.

The following conclusions can be drawn from this performance study.

- Full data and metadata journaling (Ext3) appears to dramatically slow down the performance of a data mining engine. For I/O intensive cases, VIPER's execution time under Ext3 was double of time needed for the same data execution under Reiser.
- Ext3 achieved write speed 35 times slower than the write speed of Reiser.

It should be noted that the gap between the execution times under these two file systems may grow with the size of processed data sets because larger the sets more I/O bound the data mining engine becomes. Moreover, the results presented in the paper were collected on relatively dated processor technology (Pentium II, 400 MHz). On more recent systems, the processor/disk speed performance gap is much greater than in Pentium II. Thus, the impact of filesystem performance on data mining execution times will be even more pronounced for the most recent processors than it was reported here.

In short, we conclude that full-journaling filesystem significantly slows-down the performance of VIPER and should be avoided. An interesting question, that we plan to explore in the future, is how does Ext3 with ordered mode (metadata-only journaling) impacts the data mining performance. We hypothesize that Ext3 will perform significantly faster in this mode than with full journaling.

An interesting topic of future research is an investigation of statistical properties of addresses in memory traces. We hypothesize that addresses of data mined by the data mining engine are in long-term dependence. If this hypothesis is demonstrated true, then some related questions are of interest. For example, is the degree of long-range dependence varying for different data sets mined? Can long-range dependence be exploited to improve data mining performance?

Another useful direction of future work is to study the level of support provided by current filesystems and the needs of data mining algorithms. For example, are there filesystem features that are required by data mining but currently not available? If so, how they can be provided? Finally, our study can be expanded to investigate how memory mapping impacts filesystems performance in data mining applications.

6 Acknowledgments

This work was supported by NSF EIA-0103708 as part of the Next Generation Software Program.

References

[1] R. Agrawal, T. Imielinski, and A.Swamy. Mining assocication rules between sets of items in large databases. *Proc, of ACM SIGMOD Intl. Conf. on Management of Data*, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proc. of 20th Intl. Conf. Very Large Databases (VLDB).*, September 1994.

[3] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. Linux kernel internals. addison-wesley. 1998.

[4] S. Best. Jfs home page. `http://www-124.ibm.com/developerworks/oss/jfs/index.html`.

[5] R. Bryant, R. Forester, and J. Hawkes. Filesystem performance and scalabitlity in linux 2.4.17. *Proc. Advanced Computing Systems Association (USENIX), FREENIX Track Technical Program*, 2002.

[6] J. Mostek et. al. Porting the sgitm xfs file system to linux. June 2000.

[7] R. Galli. Journal file systems in linux. *Upgrade*, 2(6), December 2001. `http://www.upgrade-cepis.org/issues/2001/6/upgrade-vII-6.html`.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on ComputerSystems*, 10(1):3–25, February 1992.

[9] B. Nowicki. Nfs: Network file system protocol specification. *Internet Engineering Task Force. RFC-1094*, March 1989.

[10] N. Petreley. Reiserfs or ext3: Which journaling filesystem is right for you? *November, 2001.* `http://www.linuxworld.com/site-stories/2001/1120.filesystem.html`.

[11] D. E. Powell. ext3 or reiserfs? hans reiser says red hat's move is understandable. *2002.* *http://www.linuxplanet.com/linuxplanet/reports/3726/1/*.

[12] H. Reiser. Reiserfs v.3 whitepaper. *September, 2002.* *http://www.namesys.com*.

[13] D. Robbins. Filesystem update: Advanced filesystem implementor's guide, part 11. *June, 2002.* *http://www-106.ibm.com/developerworks/linux/library/l-fs11.html*.

[14] P. Shenoy, J. R. Haristsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 22–33, May 2000.

[15] J. Stasko. Using student-built algorithm animationsas learning aids. *Twenty-eighth SIGCSE Technical Symp. on Computer Science Education*, pages 25–29, February 1997.

[16] S. Tweedie. Journaling the linux ext2fs filesystem. pages 25–29, 1998. citeseer.nj.nec.com/288237.html.

[17] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372-390, May-June 2000.