The Performance Effects of Power Scaling on Kernel-based Atomic Batch Transactions

[author names redacted]

[affiliation redacted] [affiliation redacted]

ABSTRACT

The need to balance performance and power is essential to computer system efficiency. Today's server-class systems commonly support autonomous power scaling of processors, memory, and disks. While processor power scaling self-governance (e.g. Intel's Turbo Boost) can improve both performance and efficiency, there is growing evidence that at times boosting processor power (and speed) actually harms performance. In this paper, we identify clear cases where processor power scaling can reduce performance by up to 68% on two I/O intensive benchmarks. We describe a methodology for isolating the performance effects of power scaling in server-class systems. We propose a new model to explain the root causes of performance loss in the Linux kernel due to power scaling for two I/O intensive benchmarks. Using the model, we are able to identify global system locks that cause slowdowns at higher processor power (and speed) in the Linux kernel and eliminate the potential performance loss (up to 68%) from power scaling for the benchmarks studied. We provide a detailed case study of the effects of power scaling on one type of Linux kernel-based lock (i.e. atomic batch transactions) and we discuss future performance challenges for power scalable systems.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Concurrency, Multiprocessing, Mutual exclusion, Scheduling, Synchronization, Threads. D.4.8 [**Performance**]: Measurements, Modeling and prediction, Monitors

General Terms

Performance, measurement, experimentation.

Keywords

Performance, energy efficiency, atomic batch transactions.

1. INTRODUCTION

Energy efficiency is now a driving force in system design. One consequence is a notable increase in autonomic power management. For example, nearly all Intel and AMD processors now feature dynamic voltage and frequency scaling (DVFS) "governors" that promise to boost thread and ultimately application performance.

These governors generally boost processor power and speed in response to utilization and operate on the principal that higher power and frequency typically results in improved performance. While memory or I/O issues may stifle performance gains, the assumption is that higher power and frequency do no harm aside from perhaps at times wasting energy.

Figure 1 contradicts the conventional wisdom that higher power and frequency do no harm to performance. The figure shows IOZone benchmark performance for the same system running at two different (fixed) processor frequencies (1.6 GHz and 3.3 GHz). Normalizing to the best available performance for this IOZone run (1.6 GHz), the IOZone benchmark, which is typically used to compare vendor I/O performance across systems, runs 68% slower at the highest available frequency (3.3 GHz)[1].

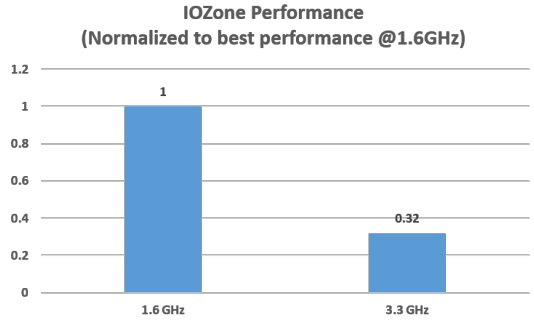

Figure 1. IOZone performance on a Dell T1100 using a Xeon E3-1270 3300 MHz (SandyBridge) quad-core with 8GB of DDR3 RAM and a 250 GB 7200 rpm hard drive. The system runs the CentOS 6.2 Linux distribution (kernel version 2.4.2).

To the best of our knowledge, no work has attempted to isolate and model the root causes of the slowdowns that Figure 1 exemplifies. Nonetheless, other researchers in several different contexts [1, 2, 4, 6-9, 15-21, 23-27, 29, 30] have independently confirmed these types of slowdowns.

In some cases [2, 4, 6-9, 15, 16, 18-21, 23-26, 29], the slowdowns were not significant enough to warrant further study. In other cases, the slowdowns were significant but the authors proffer unconfirmed theories as to the root cause.

Additionally, as we experimentally confirmed, isolating slowdowns is non-trivial. For example, before our work, there were no discernible patterns indicating slowdowns and the slowdowns could have been occurring at the application layer, in the operating system, or in the hardware itself. Thus, just isolating slowdowns is a cumbersome process that begins with exhaustive measurements across a very large experimental design space.

Contributions. Despite the challenges, we have isolated and modelled the root cause of significant slowdowns in two I/O benchmarks running on 3 different systems with power scalable

[1] Astute readers will undoubtedly want more details for these experiments. Though this is shown as a singular result, this is the average result of over 50 runs. We discuss variance and other related details in our experimental results section.

processors.[2] We found that slowdowns manifest when threads compete for shared resources in power scalable systems where thread throughput varies. The parallel threads of the I/O benchmarks we studied use a Journal Commit operation (global locked resource) to ensure the consistency of file metadata by batching accesses. These *Atomic Batch Transactions* (ABTs) use global locks that queue requests for a resource, lock the resource, service the queued requests, and release the lock. ABTs are used extensively in programs, operating systems, and databases. The use of an ABT for the Journal Commit is common across most Linux distributions.

Specifically, we identify the following key contributions of this work:

1. A methodology for isolating the effects of atomic batch transactions when power scaling varies throughput.
2. A novel model for understanding the performance costs of atomic batch transactions when power scaling varies throughput.
3. A detailed case study where we eliminate slowdowns of kernel-based atomic batch transactions when power scaling varies throughput.

The remainder of this paper details our experimental method for isolating the effects of power scaling on performance. We then introduce analytical models of the applications studied and use the resulting analyses to identify the root cause of slowdowns. We follow with a case study to demonstrate the tradeoffs between slowdowns and reliability by disabling the causal atomic batch transactions in the Linux kernel. We close with implications for the broader set of synchronization constructs and a detailed discussion of related work.

2. Isolating Slowdowns

We began our work with exhaustive testing for significant slowdowns across a large number of applications including the SPEC benchmarks[3], the NAS Parallel Benchmarks[4], TPC-C[5] and others suggested by the literature [4, 6-8, 18, 29]. While we were able to confirm slowdowns in a number of cases, the slowdowns were isolated and revealed no discernable pattern. Additionally, the benchmarks were sufficiently complicated that identifying the root causes of the slowdowns without any indication of where they resided was impracticable.

Though patterns and causes were not apparent, the slowdowns we found seemed most pronounced in I/O-heavy workloads. This inspired us to focus first on the IOZone benchmark[6] and later on the Metarates[7] benchmark. With initial data indicating pronounced slowdowns in these benchmarks, we began an exhaustive study to identify patterns and narrow causes to either application, operating system or hardware.

[2] We focus on power-scalable processors since the systems are readily available, the results are directly measureable, and the isolation to a single component simplifies the problem somewhat. The benchmarks were selected due to their common use in I/O performance comparisons between systems and their regularity.

[3] http://www.spec.org/

[4] http://www.nas.nasa.gov/

[5] http://www.tpc.org

[6] http://www.iozone.org

[7] http://www.cisl.ucar.edu/css/software/metarates/

2.1 Experimental Setup

IOZone is a file system benchmark that generates and measures a variety of file operations. This benchmark is part of the Phoronix Test Suite[8] used by review sites including Tom's Hardware[9]. In our tests, we measured the maximum achievable throughput for read-write operations on an exhaustive combination of threads (up to 256), file sizes (up to 16 MB), and record sizes (up to 16 MB).

Metarates is a file system benchmark that measures the performance of concurrent aggregate metadata transaction rates in extremely large file systems [22]. In our tests, we measured the maximum achievable throughput for read-write operations on exhaustive combinations of threads (up to 64) and files (up to 256).

We select systems based on diversity, local availability, and the presence of power-scalable processors. The first system we refer to as *SandyBridge (HDD)*. This is a Dell T1100 using a Xeon E3-1270 3.3 GHz (SandyBridge) quad-core with 8 GB of DDR3 RAM and a 250 GB 7200 rpm hard drive. The second system we refer to as *Nehalem (HDD)*. This is a Dell T3500 using a W3550 3.00 GHz (Nehalem) quad-core with 6 GB of DDR3 RAM and a 250 GB 7200 rpm hard drive. The third system we refer to as *Nehalem (Raid10)*. This is a Dell T7500n using two Xeon E5620 2.40 GHz (Nehalem) quad-core processors with 12 GB DDR3 RAM and four 146 GB 15K rpm hard drives with a RAID10 configuration. We disable the turbo boost and hyper-threading features so we can manually isolate performance at each static frequency and isolate slowdowns.

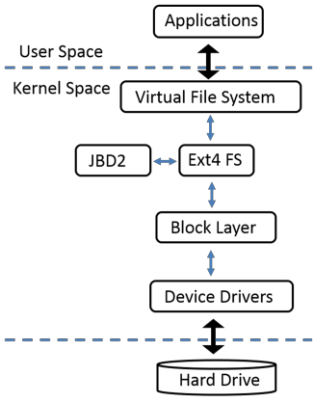

Figure 2. The Linux kernel IO subsystem.

All three systems run the latest CentOS Linux distribution (6.2) with the latest kernel version (2.4.2) available at the time of this writing. For the exhaustive benchmark tests, we run all of the combinations of benchmark parameters mentioned across all the available P-states for each processor. We instrumented the source code (e.g. hardware counters, gettimeofday(), etc.) and later the operating system source code. For the data shown herein, the number of total repeat experiments (>50 in all cases) for a given data point was selected to achieve 95% statistical confidence [28]. A full sweep across these parameter sets takes about one and a half months to complete.

[8] http://www.phoronix-test-suite.com/

[9] http://www.tomshardware.com/

Table 1. Linux components instrumented along critical path of IOZone and Metarates benchmarks. Modelling nomenclature shown for correlation with modelling parameters defined in Table 2. See Section 3 for modelling details.

Kernel Component	Function	Instrumentation	Measurement
VFS	sys_open () (source: fs/open.c) sys_close () (source: fs/close.c) sys_write () (source: fs/read_write.c)	open begins and ends: $t_{K_i}(q_{open})$ and $t_{K_i}(r_{open})$ close begins and ends: $t_{K_i}(q_{close})$ and $t_{K_i}(r_{close})$ write begins and ends: $t_{K_i}(q_{write})$ and $t_{K_i}(r_{write})$	$\mathrm{T}_{\mathrm{open}}(K_i) = t_{K_i}(r_{open}) - t_{K_i}(q_{open})$ $\mathrm{T}_{\mathrm{close}}(K_i) = t_{K_i}(r_{close}) - t_{K_i}(q_{close})$ $\mathrm{T}_{\mathrm{write}}(K_i) = t_{K_i}(r_{write}) - t_{K_i}(q_{write})$
EXT4	ext4_fsync_file () (source: fs/ext4/fsync.c)	fsync begins and ends: $t_{K_i}(q_{fsync})$ and $t_{K_i}(r_{fsync})$ writeback begins and ends: $t_{K_i}(q_{wb})$ and $t_{K_i}(r_{wb})$	$\mathrm{T}_{\mathrm{fsync}}(K_i) = t_{K_i}(r_{fsync}) - t_{K_i}(q_{fsync})$ $\mathrm{T}_{\mathrm{wb}}(K_i) = t_{K_i}(r_{wb}) - t_{K_i}(q_{wb})$
JBD2	jbd2_journal_commit_transcation () (source: fs/jbd2/commit.c)	journal commit begins and ends: $t_{L_p}(q_{jc})$ and $t_{L_p}(r_{jc})$	$\mathrm{T}_{\mathrm{JC}}(L_p) = t_{L_p}(r_{jc}) - t_{L_p}(q_{jc})$

2.2 Isolation Methodology

Figure 2 provides an illustration of the potential locations where slowdowns can occur. Applications operate in user space which could contribute to slowdowns. The operating system provides a platform for program execution that consists of numerous software components (file system, devices drivers, etc.) where the slowdowns could manifest. The hardware (CPUs, memory, disks) could also be the root cause of slowdowns since the power scaling itself is a hardware function that dynamically changes the hardware throughput.

Once we identified the applications for study (IOZone and Metarates), we created a platform where every nuance of performance could be captured. Table 1 lists the instrumented functions of the Linux kernel. When complete, all three platforms studied ran a fully instrumented software stack. We controlled the granularity of measurements and tested to ensure none of the reported results were perturbed with overhead.

Initially we used our timing data, hardware counter data, and principal component analysis [3] to identify any correlations between slowdowns and hardware configuration data (cache hits, IPC, etc.). We found none. While this did not rule hardware out completely, it indicated any hardware trigger of slowdowns would be much more difficult to isolate and could require simulation.

We next focused our attention on software. We instrumented all the functions in the Linux kernel on the critical path of IOZone and Metarates (see Table 1). By comparing instrumented runs without slowdown to those with slowdown (where the workload is identical), we were able to narrow the cause down to the *write* and *fsync* I/O system calls in the Linux kernel. These functions are used by both benchmarks.

Since the benchmarks consist of parallel threads making system calls in a non-deterministic sequence, it was not initially clear whether the delays occurred during user-space operations or during kernel-space operations. To isolate the cause further, we analyzed and instrumented the GNU C Library (glibc) wrapper interface to separate timings by user- and kernel-space. The resulting traces showed kernel time dominated user time in cases of slowdown.

Next, we manually analyzed the source code and instrumentation for the virtual file system (VFS) and subsequently the fourth extended file system (EXT4). This led us to the Journaling Block Device (JDB2) which provides a file system independent interface for journaling. Journaling makes file systems more reliable by tracking all changes before they are committed to storage.

The platform independent JDB2 harbors an Atomic Batch Transaction (ABT) called the Journal Commit (JC) triggered on a user-driven fsync system call[10]. The JC locks the file system metadata allowing no further modifications until all existing modifications are fully committed to the on-disk journal. Though the JC batches accesses, multiple batches occur in sequence.

From these observations, we instrumented the kernel further and found a direct correlation between slowdowns and the number of journal commit operations. Simply put, the observed slowdowns were a result of additional serialized journal commits that occurred when processor frequency was increased.

2.3 Measurements and Analysis

Upon further analysis, we found that the excessive journal commits occurred with high regularity but at times the additional delays were masked by other performance phenomenon such as high variability in the I/O subsystem. This meant that the delays were always there for certain applications and workloads, but that the high system variance masked the behavior making slowdowns appear at times and not appear at other times depending on the severity of the delay. The larger the slowdown, the less likely it could be masked. This helps explain the need for exhaustive experimentation to isolate the phenomenon and why other researchers may have failed to notice the potential significance of the slowdowns.

[10] Linux experts may attest that there are other scenarios (e.g. cron jobs) where fsyncs may occur periodically. Since fsyncs in our benchmarks occur with very high frequency and periodic system fsyncs typically occur at intervals such as 5 or 10 seconds, we ignore this complication.

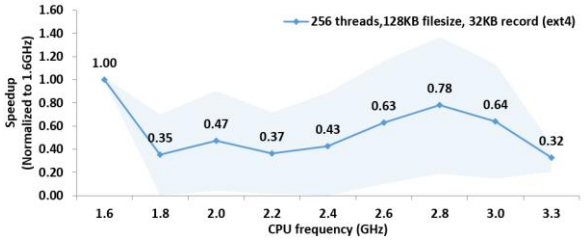

Figure 3. Select results from IOZone benchmark on *SandyBridge (HDD)*. Findings are comparable on the other systems though not included due to space limitations.

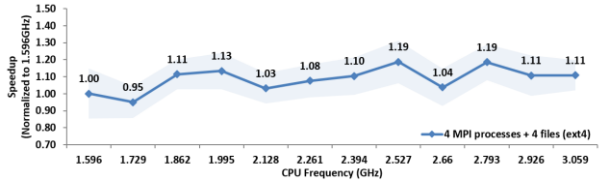

Figure 4. Select results from Metarates benchmark on *Nehalem (HDD)*. Findings are comparable on the other systems though not included due to space limitations.

Figures 3 and 4 show the average speedup from select IOZone and Metarates experiments. Average speedup refers to the ratio of the average performing high power state to the average performing low power state for a single experimental combination of benchmark parameters and system. In the figures, we use the results obtained at the frequency of 1.6 GHz (the lowest available frequency) as the baseline to calculate the speedup ratio for the results from higher power states. Speedups less than 1 indicate slowdown versus the 1.6 GHz case; i.e., increased processor speed hurts performance. For example, IOZone for a given number of threads (256), file size (128 KB), and record size (32 KB) on the system running at 3.3GHz is 68% slower than running on the same system at 1.6GHz. (In other words, the code completes 3.08x faster at 1.6 GHz versus running at 3.3 GHz.); Metarates for a given number of threads (4) and files (4) on the system running at 2.66GHz is 12.7% slower than running on the same system running at 2.527 GHz. (1.14x faster time to completion at 2.527 GHz versus 2.66 GHz.)

The gray area in each figure shows the relative standard deviations (RSDs) for the select IOZone and Metarates experiments. For example, IOZone for a given number of threads (256), file size (128 KB), and record size (32 KB) on the system running at all available frequencies exhibits RSDs from 2-108%; Metarates for a given number of threads (4) and files (4) on the system running at all available frequencies exhibits RSDs from 7-17%.

Measurement uncertainties have several consequences. First and foremost, to obtain significance greater than 95% we typically must run experiments more than 50 times each and use the statistical average. Second, while any conclusions can be made with reasonable statistical confidence (95%) a small amount of uncertainty (<5%) is practically unavoidable. Third, while variance is not the direct cause of the slowdown phenomenon, we observe that samples with low variance often indicate the best performing case. For example, IOZone performs best at 1.6 GHz (see Figure 3) which also shows the low RSD.

We must stress that we are not identifying any system or application bugs. What we identified are certain combinations of applications and systems that cause resource contention that drastically affects performance. This is a fundamental issue with system design. The system itself has a fixed amount of resources while demands for these resources ebb and flow. Processor scaling technologies (often for power-performance efficiency management) ignore these effects and operate autonomously – in many cases to good effect. However, during times of resource contention, the speedup gained from accelerating throughput is often nullified by the creation of additional contention on resources. It is this contention that wreaks havoc on performance and results in the data found in Figures 3 and 4.

3. MODELING ATOMIC BATCH TRANSACTIONS (ABTs)

The correlation of slowdowns to atomic batch transactions (i.e. journal commits) in the Linux operating system for the IOZone and Metarates benchmarks was a critical step toward understanding slowdowns. However, while empirical measurements indicate where time is spent during slowdowns, they do not fully explain why the slowdowns (or additional journal commits) occur.

We now present an analytical model of atomic batch transaction performance. The model uses the parameters measured from the aforementioned Linux kernel instrumentation (see Table 1). These parameters are further defined in Tables 2 and 3 in the next subsection.

The resulting analytical model shows that slowdowns occur at higher frequencies when the early arrival of a single thread (among many) causes the atomic batch transaction (i.e. journal commit) to lock with less batched threads than in the lower frequency case. In the lower frequency case, the difference between the lead thread and other threads is much smaller – so, when the atomic batch transaction (i.e. journal commit) locks, more threads are batched and overall less atomic batches occur. When we apply the analytical model to the specific I/O benchmarks where journal commits are extremely expensive, the number of journal commits identified in the model further explains the slowdowns.

3.1 Analytical Model of ABT Performance

3.1.1 User-space I/O Threads

We use the notations in Tables 2 and 3 throughout the following subsections. Suppose a parallel I/O workload launches **N** threads, where each thread starts at the same time and simultaneously makes POSIX I/O system calls to its own file. Let **K** represent a set of I/O threads. The total number of threads is $|\mathbf{K}| = \mathbf{N}$. Each element in **K** represents one individual thread, from $K_1, K_2, \cdots, to\ K_N$. K_i indicates the *i-th* thread in **K**, using i as an index to select a thread from **K**.

Each thread receives a task with size **S** and requires a series of operations (ops) to complete. The ops that a thread can execute are the **open**, **write**, **close**, and **fsync** POSIX I/O calls. Each thread requires more than one op for its task. Each op is a blocking system call, so each thread needs to complete its current op before the thread can progress to its next op.

For each op, q_{op} is used to indicate the event where the op begins, and r_{op} is used to refer to the event where the op ends. When K_i (the *i-th* thread in **K**) begins its op, $t_{K_i}(q_{op})$ is used to indicate the time stamp of this event, and $t_{K_i}(r_{op})$ is used to refer to the time stamp of when K_i finishes its op. Let $\mathrm{T_{op}}(K_i)$ represent the time elapsed for the *i-th* thread to complete its op. Then $\mathrm{T_{op}}(K_i)$ can be calculated by subtracting the time stamp when the

i-th thread ends its op from the time stamp when the *i-th* thread begins its op.

$$\mathrm{T_{op}}(K_i) = t_{K_i}(r_{op}) - t_{K_i}(q_{op}) \quad (1)$$

3.1.2 The Journal Process

When a parallel I/O workload requires user-space I/O threads to execute **fsync** ops before their completion of the tasks, a set of journal processes will be activated in the Linux kernel. There is only one active journal process running at a time in the Linux kernel. The job for a journal process is to issue a journal commit operation, which synchronizes the in-memory metadata in the virtual file system of the Linux kernel with the permanent (non-volatile storage).

Let **L** represent the journal processes that are activated by the user-space I/O threads via their **fsync** ops. The total number of the journal processes is $|\mathbf{L}| = \mathbf{O}$. Each element in **L** represents one individual journal process, from $L_1, L_2, \cdots, to\ L_O$. Let L_p indicate the *p-th* journal process in **L**, using p as an index to specify a journal process from **L**. For each op, q_{op} is used to indicate the event that the op begins, and r_{op} is used to refer to the event that the op ends. When the *p-th* journal process L_p issues its op, $t_{L_p}(q_{op})$ is used to indicate the time stamp of this event, and $t_{L_p}(r_{op})$ is used to refer to the time stamp of when L_p finishes its op. We assume that the kernel journal process issues only one operation: the journal commit (**JC**) op. We will remove this constraint when we apply the general model to the two benchmarks studied.

Let $\mathrm{T_{op}}(L_p)$ represent the elapsed time for the *p-th* journal process to complete its op. Then $\mathrm{T_{op}}(L_p)$ can be obtained by subtracting the time stamp when the *p-th* journal process ends its op from the time stamp when the *p-th* journal process begins its op.

$$\mathrm{T_{op}}(L_p) = t_{L_p}(r_{op}) - t_{L_p}(q_{op}) \quad (2)$$

3.1.3 I/O Thread and Journal Process Interaction

The kernel-space journal processes may influence the user-space I/O threads during their fsync ops and write ops. We provide a separate discussion for each scenario. Our descriptions for user-space I/O threads, kernel-space journal processes, and the fsync- and write-op scenarios will be used to construct analytical models of IOZone and Metarates in the next section.

FSYNC-OP SCENARIO. In this scenario, a user-space I/O thread may be delayed if it arrives too late to be serviced by the journal commit servicing the previous user-space I/O thread. Let K_i be the first thread to start its **fsync** op and wake up the journal process L_p in the Linux kernel. L_p then begins its **JC** op and synchronizes the metadata from the physical memory with the permanent storage.

At this point, K_i (which issues the **fsync** op and wakes up L_p) will be waiting for its **fsync** op until L_p completes its **JC** op before K_i can progress to its next op. Let K_j be another I/O thread that reaches the phase of its **fsync** op and requests the kernel to launch another journal commit to sync K_j's in-memory metadata with the storage. Let L_q be the journal process assigned by the kernel for K_j's metadata synchronization, and since a journal process is limited to run one at a time in the Linux kernel, K_j has to wait until L_p completes its **JC** op before L_q is actually woken up by the kernel and issues its **JC** op for K_j. As a result, K_j will be waiting for its **fsync** op for two **JC** ops, L_p's **JC** op and L_q's **JC** op. We consider a user-space I/O thread as *delayed* when its **fsync** op includes more than one **JC** op issued by the kernel-space journal processes. Section 3.2 illustrates this fsync-op scenario in detail using the Metarates benchmark.

WRITE-OP SCENARIO. In this scenario, a journal process influences user-space I/O threads during their **write** ops. Let K_i be the first thread to start its **fsync** op, which wakes up the journal process L_p in the Linux kernel. L_p then begins its **JC** op, syncing

Table 2. Thread/journal-dependent notations.

Notations	Definition
K	Set of threads, $\mathbf{K} = \{K_1 + K_2 + \cdots + K_N\}$
K_i	The *i-th* thread in **K**
S	Size of the workload
N	Total number of threads, $\|\mathbf{K}\| = \mathbf{N}$
q_{op}	The event when op begins
r_{op}	The event when op finishes
$t_{K_i}(q_{op})$	The timestamp when the *i-th* thread begins its op
$t_{K_i}(r_{op})$	The timestamp when the *i-th* thread finishes its op
$\mathrm{T_{op}}(K_i)$	Time elapsed for the *i-th* thread to complete its op
L	Set of journal processes, $\mathbf{L} = \{L_1 + L_2 + \cdots + L_O\}$
O	Total number of journal processes, $\|L\| = O$
L_p	The *p-th* journal process in **L**
$t_{L_p}(q_{op})$	The timestamp when the *p-th* journal process begins its op
$t_{L_p}(r_{op})$	The timestamp when the *p-th* journal process finishes its op
$\mathrm{T_{op}}(L_p)$	Time elapsed for the *p-th* journal process to complete its op

Table 3. Operation(op)-dependent notations.

op	Definition
User-space ops	
open	Open a file via the POSIX system call open()
write	Write data to a file via the POSIX system call write()
close	Close a file via the POSIX system call close()
fsync	Synchronize data and metadata from memory to storage device via the POSIX system call fsync()
Kernel-space ops	
DFLUSH	Flush in-memory data to storage device
JCFLUSH	Flush in-memory metadata to storage device via journal
JC	Journal commit

the in-memory metadata with the permanent storage. Thus, the metadata is locked by L_p for its **JC** op, and any modification request to the ongoing metadata is put on hold and has to wait until the lock to the metadata is released when L_p completes its **JC** op.

Let K_j be another I/O thread either in its **write** op or about to enter its **write** op, attempting to update the locked metadata. Then K_j will be blocked on its **write** op and will have to wait until L_p completes its **JC** op. The wait time for K_j on its **write** op is expensive in this scenario since metadata synchronization involves the memory (e.g., **write** op) and the storage device (e.g., wait on the **JC** op), while a regular **write** op requires accesses only to the memory. We consider the user-space I/O thread K_j as delayed when its **write** op includes the wait time for a journal process on the **JC** op that involves storage accesses. Section 3.3 illustrates this write-op scenario in detail using the IOZone benchmark.

3.2 Metarates Baseline Performance Model

To test the file system's metadata performance, Metarates includes the file creation rate (FCR) test that launches MPI processes, where each MPI process in parallel issues a sequence of POSIX system calls (e.g., open(), fsync(), and close() functions) to complete its task. Specifically, for the FCR test, each MPI process completes its task by creating a file, flushing metadata to the storage device, and then closing the file.

Let **K** be a set of MPI processes launched by Metarates where each of the elements in **K** represents an individual MPI process, from $K_1, K_2, \cdots, to\ K_N$. The total number of MPI processes is $|\mathbf{K}| = \mathbf{N}$. When Metarates starts the FCR test, each MPI process in $K_1, K_2, \cdots, K_N$ is given a task to issue **open**, **fsync**, and **close** ops on its own file. The time each MPI process takes to complete its task is measured, and the average time for all MPI processes to complete their tasks is used as the performance metric of the Metarates FCR test.

$$\text{File creation rate (FCR)} = \frac{\mathbf{N} \times 1}{\sum_{i=1}^{N}\left(T_{open}(K_i)+T_{fsync}(K_i)+T_{close}(K_i)\right)} \quad (3)$$

From the Metarates FCR calculation and our direct measurements (see Table 1 "Measurement"), we observe that $T_{fsync}(K_i)$ is significantly larger than $T_{open}(K_i)$ and $T_{close}(K_i)$ for a given $K_i \in \mathbf{K}$ in its **open**, **fsync**, and **close** ops. In addition, $T_{fsync}(K_i)$ has a large variance, while $T_{open}(K_i)$ and $T_{close}(K_i)$ are typically stable and repeatable.

$T_{fsync}(K_i)$ is significantly larger compared with $T_{open}(K_i)$ and $T_{close}(K_i)$ because an **fsync** op requires storage device accesses, while the **open** and **close** ops involve only memory accesses. Thus, the $T_{fsync}(K_i)$ term dominates the measured variance in the FCR value for the Metarates benchmark.

We instrument the Linux kernel to provide additional details for $T_{fsync}(K_i)$ by fully tracing K_i in its **fsync** op. Our results indicate the variance in $T_{fsync}(K_i)$ is highly dependent on the kernel journal process executing the journal commit, which synchronizes in-memory metadata with the storage device via the on-disk journal.

Let **L** represent the journal processes that are activated by the MPI processes in **K** via their **fsync** ops. The total number of journal processes is $|\mathbf{L}| = \mathbf{O}$, and each element in **L** represents one individual journal process, from $L_1, L_2, \cdots,$ to L_O. From the kernel trace, we observe that one or several MPI processes in **K** share a $L_p \in \mathbf{L}$.

In essence, several MPI processes are at times queuing up waiting on a single journal process. The key to modeling slowdown in Metarates lies in identifying the number of journal commits necessary to complete all the threads. The number of journal commits needed depends on the arrival times of the user-space threads. Since the user-space threads are racing and competing for access to each journal commit, modeling user-space process arrivals implicitly models the contention for the kernel-space journal commits.

3.2.1 Metarates and Journal Commits

Let K_i be the first MPI process progressing to its **fsync** op, L_1 be the first journal process triggered by K_i, and L_2 be the journal process after L_1. Figure 5 shows the events belonging to K_i, L_1, and L_2, with the time increasing from left to right. K_i issues three ops—**open**, **fsync**, and **close**—to complete its task. K_i starts by issuing an **open** op to create an empty file, then moves to an **fsync** op. When K_i is on its **fsync** op, it is must wait for L_1 on its **JC** op to sync the in-memory metadata. As soon as L_1completes its **JC** op, K_i progresses to its **close** op, and K_i finishes its task after its **close** op is done.

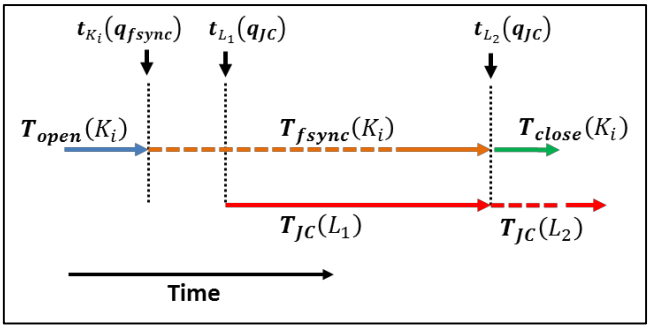

Figure 5. The interaction between a single user-space thread (e.g. MPI process = open + fsync + close) and a single kernel-space journal process (e.g. Journal Commit = JC) for the Metarates benchmark.

The Linux kernel allows one running, active journal process at a time, L_1. If there is a second **JC** op waiting to run, this journal process, L_2, wakes upon completion of L_1. The time needed to execute the remaining user-space MPI processes depends heavily on when they arrive and how long they must wait for their **fsync** (and any associated kernel-space journal commits) to complete.

Let $K_j \in \mathbf{K}$ be another user-space MPI process such that $t_{K_j}(q_{fsync}) > t_{K_i}(q_{fsync})$), i.e. K_j progresses to its **fsync** after K_i progresses to its **fsync**. Since there may be a delay between K_i's **fsync** progression ($t_{K_i}(q_{fsync})$) and the start of the corresponding journal commit $t_{L_1}(q_{JC})$, there are two possibilities for K_j's **fsync** progression: 1) K_j's **fsync** progression occurs before the start of K_i's corresponding journal commit, i.e. ($t_{K_j}(q_{fsync}) < t_{L_1}(q_{JC})$) and K_j must only wait for L_1 to complete; or 2) K_j's **fsync** progression occurs after the start of K_i's corresponding journal commit, i.e. ($t_{K_j}(q_{fsync}) > t_{L_1}(q_{JC})$) and K_j must suffer the

remainder of the L_1 journal commit and then wait for L_2 to complete.

More precisely for the first scenario, MPI processes waiting for L_1 satisfy the following: Let $K_j \in \mathbf{K}$ and $K_j \neq K_i$.

If $t_{K_j}(q_{fsync}) < t_{L_1}(q_{JC})$, then

$$\mathrm{T_{fsync}}(K_i) = \left(t_{L_1}(q_{JC}) - t_{K_i}(q_{fsync})\right) + \mathrm{T_{JC}}(L_1), \text{ and}$$
$$\mathrm{T_{fsync}}(K_j) = \left(t_{L_1}(q_{JC}) - t_{K_j}(q_{fsync})\right) + \mathrm{T_{JC}}(L_1). \quad (4)$$

More precisely for the second scenario, MPI processes waiting for L_2 will suffer variable delay from the L_1 journal commit which depends on the start of the **fsync** progression. These MPI processes satisfy the following: Let $K_j \in \mathbf{K}$ and $K_j \neq K_i$.

If $t_{K_j}(q_{fsync}) > t_{L_1}(q_{JC})$, then

$$\mathrm{T_{fsync}}(K_i) = \left(t_{L_1}(q_{JC}) - t_{K_i}(q_{fsync})\right) + \mathrm{T_{JC}}(L_1), \text{ and}$$
$$\mathrm{T_{fsync}}(K_j) = \left(t_{L_2}(q_{JC}) - t_{K_j}(q_{fsync})\right) + \mathrm{T_{JC}}(L_2). \quad (5)$$

3.2.2 Algorithm for Metarates Performance

Note that for any given run we are measuring all threads for all the parameters described by our methodology in Table 1. Since the arrival order is non-deterministic and the thread times from run to run vary significantly, we need a method to reconstruct the arrival groupings of threads. Algorithm 1 (see Appendix) describes the algorithm for the Metarates FCR test that we use to calculate the time spent in the **fsync** ops when a set of MPI processes **K** are launched and trigger a set of journal processes **L**. We assume that the number of MPI processes **K** is $|\mathbf{K}| = \mathbf{N}$, and the number of journal processes is $|\mathbf{L}| = \mathbf{O}$.

The algorithm begins with a set of MPI processes in **K** and a set of journal processes in **L**; the algorithm finishes when both the **K** and **L** pools are empty. The algorithm identifies the relationships between the MPI processes in **K** and the journal processes in **L** and uses this information to calculate the time spent on each **fsync** op of an MPI process.

The algorithm calculates the time spent in every individual **fsync** op by iterating over all the MPI processes in **K**. The algorithm attempts to determine how many journal commit operations in **L** a given fsync op must suffer. In other words, a **JC** op is able to service multiple **fsync** ops at a time, but since **JC** ops are serialized, we must determine which **JC** ops service which user level **fsync** threads.

On lines 3 to 4 of the algorithm, a pair of K_i and L_p is selected to form a group such that K_i is the MPI process that triggers the journal process L_p. K_i is determined by choosing the smallest $t_{K_i}(q_{fsync})$ in **K**, and is the first MPI process that starts its **fsync** op in **K**. L_p is determined by selecting the smallest $t_{L_p}(q_{JC})$ in **L**, and is the first journal process that begins its **JC** op in **L**. On line 5 of the algorithm, once K_i and L_p are found, the time that K_i spends on its **fsync** op $T_{fsync}(K_i)$ can be calculated by the equation below:

$$\mathrm{T_{fsync}}(K_i) = \left(t_{L_p}(q_{JC}) - t_{K_i}(q_{fsync})\right) + \mathrm{T_{JC}}(L_p) \quad (6)$$

The algorithm then uses K_i and L_p to search through all of the K_js that belong to this group. From lines 8 to 11 of the algorithm, the inner loop is used to check the rest of the MPI processes in **K** and find all the K_j processes that have their **fsync** ops serviced by L_p. These K_j processes meet the condition of $t_{K_j}(q_{fsync}) < t_{L_p}(q_{JC})$, and the time that each K_j spends on its **fsync** op $\mathrm{T_{fsync}}(\mathrm{K_j})$ can be calculated using the following equation:

$$\mathrm{T_{fsync}}(K_j) = \left(t_{L_p}(q_{JC}) - t_{K_j}(q_{fsync})\right) + \mathrm{T_{JC}}(L_p) \quad (7)$$

Lastly, on lines 6, 7, and 11, the algorithm removes K_i and the K_j processes from **K** and the L_p processes from **L**. The algorithm continues to find the next K_i and L_p from **K** and **L** for the next group as well as the K_j processes belonging to the next group. The algorithm repeats until the **K** and **L** pools are empty.

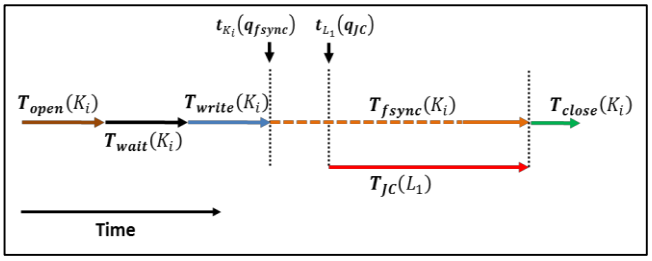

Figure 6. The interaction between a thread and a journal process for the IOZone benchmark.

3.3 IOZone Baseline Performance Model

The IOZone throughput tests use threads to test the file system's read-write performance by using POSIX system calls (e.g., open(), write(), fsync(), and close() functions). Specifically, IOZone includes an initial write throughput test that launches threads, where each thread in parallel issues POSIX system calls and completes its task by creating a file, writing data to the file, flushing data/metadata to the storage device, and then closing the file. For this initial write throughput test, the time spent on writing the data to a file by each thread is measured as the result, and the performance metric is calculated by accumulating the results from all threads. From now on, unless specified, we use the throughput test to refer to the IOZone initial write throughput test.

Let **K** be a set of threads. Each element in **K** represents an individual thread, from $K_1, K_2, \cdots, to\ K_N$. The total number of threads is $|\mathbf{K}| = \mathbf{N}$. When IOZone starts the throughput test, each thread in $K_1, K_2, \cdots, to\ K_N$ is given a task to execute **open**, **write**, **fsync**, and **close** ops on its private file. The size of the workload on each thread for the **write** op is **S**, and the time spent on the **write** op is measured.

$$\mathrm{IOZone\ Throughput_{initial\_write}} = \sum_{i=1}^{N} \frac{S}{\mathrm{T_{write}(K_i)}} \quad (8)$$

From our IOZone throughput calculation and our direct measurements (see Table 1 "Measurement"), we observe that any variance from a number of $\mathrm{T_{write}}(K_i)$ events with significantly large values that skew the average. The results are repeatable and frequent at times so they are not statistical outliers.

For Metarates, our analyses focused on the **fsync** operation that dominates the ***open-fsync-close*** operation sequence for each thread. The arbitration for a fixed resource (the journal commit, **JC**) and the total number of sequential journal commits drives the total time and is essential to understanding the slowdown.

Understanding slowdown in IOZone requires studying the variance in **write** op performance per thread. In IOZone, $\mathrm{T_{fsync}}(K_i)$ is not used in the performance throughput calculation. The IOZone performance metric focuses on $\mathrm{T_{write}}(K_i)$ and the time it takes for $K_i \in \mathbf{K}$ to complete the **write** op. The **write** operation calls **fsync** upon completion indirectly calling the journal commit (**JC**) operation. Thus, the throughput of the benchmark is a function of the size of the data, time to complete the write and any additional delays due to the resulting JC count. Once again, the challenge in modelling IOZone is isolating user-space thread costs while accounting for additional time due to delays in the kernel.

3.3.1 IOZone and Journal Commits

Let K_i be the first thread that triggers the journal process L_1 via the **fsync** op. K_i issues four ops—**open**, **write**, **fsync**, and **close**—to complete its task. Figure 6 shows a group of events belonging to K_i and L_1, with the time increasing from left to right. K_i starts by issuing an **open** op to create an empty file and waits for all threads to finish their **open** ops before it progresses to its **write** op. This is the synchronization event in the IOZone throughput test that ensures all threads start their **write** ops at the same time. Next K_i issues a **write** op with a workload of size **S** and progresses to its **fsync** op. When K_i is on its **fsync** op, it first waits for the kernel to flush out its in-memory file data, and then it waits for the kernel journal process L_1's **JC** op to synchronize the in-memory file metadata with the storage device. After $\mathrm{L_1}$ completes its **JC** op, $\mathrm{K_i}$ progresses to its **close** op and finishes its task.

During the phase that L_1 is on its **JC** op, the file system's metadata is locked and is synchronized with the storage device. Any **write** op that attempts to update the metadata is blocked and has to wait until the metadata is released by L_1, which is the reason that these blocked **write** ops take a significantly long time to complete. These blocked **write** ops are forced to wait for the duration of time spent in data transmission between the memory and the storage device, while the unblocked **write** ops involve only the memory accesses. Let $K_j \in \mathbf{K}$ and $K_j \neq K_i$, and let K_i be the thread that triggers L_1. If L_1 is on its **JC** op, upon completion of its write op K_j must block until the L_1 **JC** op completes.

3.3.2 Algorithm for IOZone Performance

Note that for any given run we are measuring all threads for all the parameters described by our methodology in Table 1. Since the arrival order is non-deterministic and the thread times from run to run vary significantly, we need a method to reconstruct the arrival groupings of threads. Algorithm 2 (see Appendix) analytically describes how the **JC** op of L_1 affects the **write** ops of the K_j processes in three scenarios: (1) not affected, when the write ops of the K_j processes are not affected by the **JC** op of $\mathrm{L_1}$; (2) fully blocked, when the write ops of the K_js are blocked by the entire **JC** op of $\mathrm{L_1}$; and (3) partially blocked, when the **write** ops of the K_j processes are blocked by the partial **JC** op of $\mathrm{L_1}$. The algorithm uses **X**, **Y**, and **Z** sets to represent these three scenarios, respectively. These three sets are used to categorize the threads in **K**.

From lines 2 to 4, the algorithm starts by looking for K_i, which triggers L_1. K_i is the first thread that completes its data flush and progresses to the second step in its **fsync** op and signals the kernel to launch a journal process to start a journal commit for the metadata synchronization. On line 2, K_i is selected using $t_{K_i}(r_{DFLUSH})$, the smallest time stamp among all of the threads in **K**. This K_i is placed into set **X** because its **write** op is not affected by the **JC** op of L_1

From lines 5 to 15, the algorithm checks every $K_j \in \mathbf{K}$ and $K_j \neq K_i$ and places K_j into set **X**, **Y**, or **Z**, according to the details of the **write** ops of the K_j processes (e.g., start time stamp, end time stamp, etc.), and with knowledge of when L_1 begins its **JC** op. Line 6 describes the "not affected" scenario, when the **write** ops of the K_j processes do not involve the **JC** op of L_1. These K_j processes start their **fsync** ops before L_1 begins its **JC** op. These K_j processes are placed into set **X**. The time a K_j thread spends on its **write** op can be calculated as follows:

$$\mathrm{T_{write}}(K_j) = t_{K_j}(r_{write}) - t_{K_j}(q_{write}) \tag{9}$$

In the "fully blocked" scenario, **write** ops of the K_j processes are fully blocked by the **JC** op of L_1. These **write** ops of the K_j processes are broken into two pieces by the **JC** op of L_1. These K_j processes start their **write** ops; then these **write** ops are blocked and wait for L_1 to complete its **JC** op; and finally they resume and finish. On lines 9 and 10, the algorithm summarizes that such K_j processes start their **fsync** ops after L_1 begins its **JC** op, and the K_j processes start their **write** ops before L_1 begins its **JC** op. These K_j processes are placed into set **Y**. The time a K_j thread spends on its **write** op and can be calculated as follows:

$$\mathrm{T_{write}}(K_j) = \left(t_{L_1}(q_{JC}) - t_{K_j}(q_{write})\right) + \mathrm{T_{JC}}(L_1) + \left(t_{K_j}(r_{write}) - t_{L_1}(r_{JC})\right) \tag{10}$$

Equation (10) demonstrates that in this scenario, each **write** op of K_j is broken into two parts: (1) before the **JC** op and (2) after the **JC** op. $\mathrm{T_{write}}(K_j)$ is assembled by the first part of the **write** op $\left(t_{L_1}(q_{JC}) - t_{K_j}(q_{write})\right)$, $\mathrm{L_1}$'s **JC** op $\mathrm{T_{JC}}(L_1)$, and the second part of the **write** op $\left(t_{K_j}(r_{write}) - t_{L_1}(r_{JC})\right)$.

In the "partially blocked" scenario, the **write** ops of the K_j processes are blocked by the partial **JC** op of L_1. These **write** ops of the K_j processes are blocked as soon as the **write** ops start, then they wait for the **JC** op of L_1 to complete, and finally they resume and finish. On lines 9 and 13, the algorithm summarizes that such K_j processes start their **fsync** ops after L_1 begins its **JC** op, and the K_j processes start their **write** ops after L_1 begins its **JC** op. These K_j processes are placed into set **Z**. The time K_j spends on its **write** op can be calculated as follows:

$$\mathrm{T_{write}}(K_j) = \left(t_{L_1}(r_{JC}) - t_{K_j}(q_{write})\right) + \left(t_{K_j}(r_{write}) - t_{L_1}(r_{JC})\right) \tag{11}$$

Equation (11) demonstrates that in this scenario, $\mathrm{T_{write}}(K_j)$ consists of the time that K_j waits for the **JC** op of L_1 to finish to the time that K_j starts its **write** op $\left(t_{L_1}(r_{JC}) - t_{K_j}(q_{write})\right)$ in addition to the time that K_j spends on its **write** op after it resumes $\left(\mathrm{t_{K_j}(r_{write}) - t_{L_1}(r_{JC})}\right)$.

3.4 Metarates Extended Performance Model

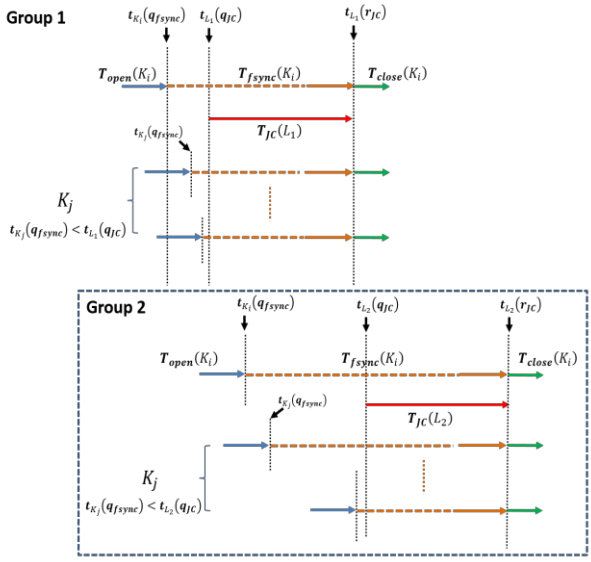

Figure 7. The Metarates runtime profile for two or more user-space MPI processes and two journal processes.

Figure 7 shows the runtime profile for the Metarates FCR test for two or more journal commits. This profile illustrates how one or more user-space MPI processes share a kernel journal process. There are two groups in the profile, Group 1 and Group 2. The two groups are differentiated by two journal commits, L_1 and L_2. In each group, let K_i be the first MPI process to trigger a journal commit. Also in each group, let K_j be one or more MPI processes that call **fsync** after the K_i in their respective group. In each group, K_i and one or more K_j processes issue **open**, **fsync**, and **close** ops for their tasks.

In Group 1, which starts slightly before Group 2, the fastest MPI process K_i progresses to its **fsync** op and triggers L_1. There is slack between when K_i starts its **fsync** op and when L_1 actually begins its **JC** op (from $t_{K_i}(q_{fsync})$ to $t_{L_1}(q_{JC})$). The K_j processes all start their **fsync** ops before L_1 starts its **JC** op, and then the K_j are queued up and serviced by L_1. These K_j processes meet the previously described $t_{K_j}(q_{fsync}) < t_{L_1}(q_{JC})$ condition. After L_1 finishes its **JC** op, K_i and the K_j processes continue their **close** ops, and then all MPI processes in Group 1 finish their tasks.

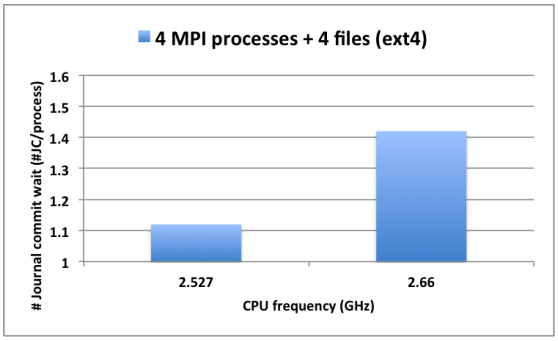

Figure 8. Metarates for a given number of threads (4) and files (4) on *Nehalem (HDD)*, the average number of journal commits each MPI process waits at runtime for 2.527 GHz and 2.66 GHz on the system.

Group 2 collects the MPI processes that start after the first journal process begins its **JC** op $t_{L_1}(q_{JC})$ and before the second journal process L_2 begins its **JC** op $t_{L_2}(q_{JC})$. Again, K_i is the MPI process that triggers L_2 within Group 2. These K_j processes meet the $t_{K_j}(q_{fsync}) < t_{L_2}(q_{JC})$ condition and are queued up and serviced by L_2. After L_2 finishes its **JC** op, K_i and the K_j processes progress to their **close** op, and all MPI processes in Group 2 finish their tasks.

The Metarates FCR test is directly impacted by the number of K_js serviced by L_2, since these K_js have to wait for both L_1 and L_2 to complete (two **JC** ops). Thus, the ideal FCR performance can be achieved when all the K_js are serviced by L_1 and thus suffer only one kernel **JC** op for their **fsync** op. On the other hand, the worse FCR performance occurs when all the K_js are serviced by L_2, and all the K_js have to wait two **JC** op before completing their tasks.

The slowdown in the Metarates FCR test (see Figure 4) when changing the processor operating frequency comes from the change in the average number of journal commits per thread (see Figure 7). As processor frequency changes, the start times $t_{K_i}(q_{fsync})$, $t_{K_j}(q_{fsync})$, and $t_{L_1}(q_{JC})$ will vary affecting the number of K_js serviced by the **JC** op of L_1. Figure 8 shows the average number of **JC** ops per process for K_j waiting at runtime from our kernel profile for 2.527 GHz and 2.66 GHz respectively. The biggest performance drop happens when scaling from 2.527 GHz to 2.66 GHz for a 12.7% slowdown. At 2.527 GHz, K_j waits an average of 1.12 **JC** ops, while at 2.66GHz K_j waits an average of 1.42 **JC** ops. This correlates to Figure 4 where the Metarates FCR performance at 2.527 GHz (**1.19x**) is higher than at 2.66 GHz (**1.04x**) on *Nehalem (HDD)*.

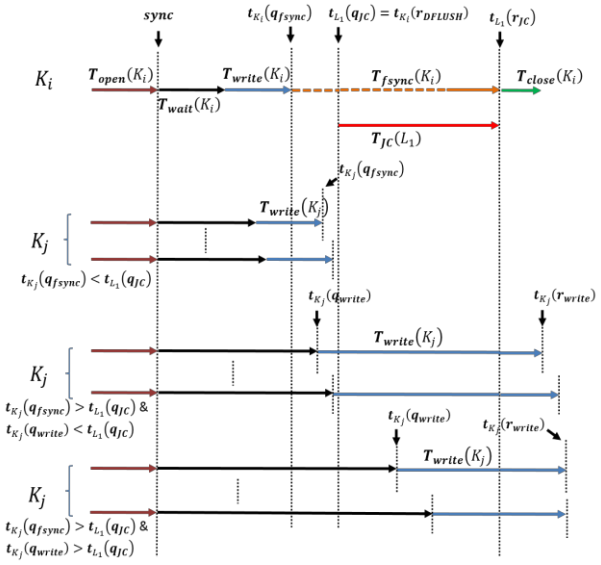

Figure 9. The runtime profile for the IOZone throughput test with multiple user-space threads and one journal process.

3.5 IOZone Extended Performance Model

Figure 9 shows how the **JC** op of L_1 affects the **write** ops of the K_j processes in three scenarios: (1) not affected, (2) fully blocked, and (3) partially blocked. In the figure, K_i and the K_j processes

start their **open**, **write**, **fsync**, and **close** ops in order. A sync event between the **open** and **write** ops is used to force all threads in **K** to wait until all their **open** ops are completed before they continue to their **write** ops. K_i is the first thread to reach the metadata sync phase in the **fsync** op and triggers L_1. L_1 then begins its **JC** op and locks the metadata in order to synchronize the in-memory metadata with the storage device.

In the "not affected" scenario, let $K_j \in \mathbf{K}$ and $K_j \neq K_i$, and when the K_j processes complete their **write** ops before L_1 starts its **JC** op, these **write** ops are not blocked. In the "fully blocked" scenario, let $K_j \in \mathbf{K}$ and $K_j \neq K_i$, and when the K_j processes start their **write** ops before L_1 begins its **JC** op and the K_j processes finish their **write** ops after L_1 completes its **JC** op, these **write** ops are fully blocked by the **JC** op of L_1. Lastly, in the "partially blocked" scenario, let $K_j \in \mathbf{K}$ and $K_j \neq K_i$, and when the K_j processes start their **write** ops after L_1 begins its **JC** op and the K_i processes finish their **write** ops after L_1 completes its **JC** op, these **write** ops are blocked by the partial **JC** op of L_1.

IOZone write throughput performance is directly impacted by the number of K_js that are delayed by the **JC** op of L_1 before the K_j can complete their **write** ops. The ideal write throughput performance is achieved when no K_j is delayed by the **JC** op of L_1, and the write throughput performance become worse as

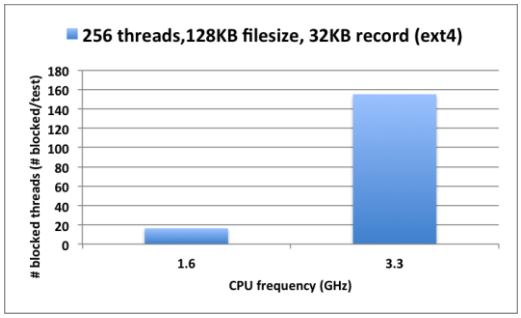

Figure 10. IOZONE for a given number of threads (256), file size (128KB), and record (32KB) on *SandyBridge (HDD)*, the average number of threads blocked by the journal commit at runtime for 1.6 GHz and 2.66 GHz on the system.

number of delayed K_j goes up.

Slowdown occurs in IOZone throughput (see Figure 3) when changes in CPU frequency increase the number of blocked writes (see Figure 9). This affects the distributed of writes across the three categories: not affected, partially blocked, and fully blocked. CPU frequency changes vary $t_{L_1}(q_{JC})$ or the time of data writeback for the first **fsync** op which launches L_1 to begin the **JC** op. Figure 10 shows the average number of K_js blocked in the **write** ops at runtime from our kernel profile for 1.6 GHz and 3.3 GHz respectively. This explains the biggest performance drop for IOZone – a 68% slowdown. At 1.6 GHz, an average of 16.34 K_js get blocked at runtime, while at 3.3 GHz an average of 155.04 K_js are blocked. This correlates to the results in Figure 3 where IOZONE throughput performance at 1.6 GHz (**1.0x**) is significantly higher than at 3.3 GHz (**0.32x**) on *SandyBridge (HDD)*.

4. PERFORMANCE VS. ABTs

Our isolation methodology and analytical models cast blame for slowdown on atomic batch transactions. We observed one example of ABTs (journal commits) cause slowdowns directly and indirectly in Metarates and IOZone respectively. But, ABT's enable journaling which makes file systems more reliable since changes are tracked before they are atomically committed to storage.

This section serves two purposes. First, we apply our modelling conclusions (i.e. dropping ABTs should be good for power-scalable performance) to a real system to remove the performance penalty when combining power scaling with the Metarates and IOZone benchmarks. Second, we quantify the impact of the additional reliability of ABTs on the performance of Metarates and IOZone.

Figures 11 and 12 (see Appendix) provide data for our repeat experiments for the observed slowdowns (labelled "ext4") and two other configurations. The "NOJC" configuration consists of a modified fsync system call in the Linux kernel that flushes out the metadata directly to the disk location and altogether avoids use of the kernel journal commit mechanism. As an alternative, the "NOFSYNC" configuration avoids using the fsync calls entirely which results in no journal commit operations.

4.1.1 Metarates Performance and ABTs

Figure 11 (see Appendix) repeats the experimental results for Metarates with the ext4 configuration with the addition of the NOJC and NOFSYNC results. Figure 11a contains 4 graphs. The second graph from the top depicts slowdowns (speedup < 1) in the default ext4 configuration also observed in Figure 4. The third (NOJC) and fourth (NOFSYNC) graphs from the top depict speedups as the processor frequency increases from left to right. Strikingly, the NOJC and NOFSYNC graphs show speedups correlating with increase processor speed (i.e. no slowdowns). A similar phenomenon is evident in Figures 11b and 11c for other combinations of MPI processes and files.

The topmost graph in each of Figures 11a, 11b, and 11c overlays the Metarates raw performance data for each of the three tests (ext4, NOJC, and NOFSYNC). The throughput is artificially high for both the NOJC and NOFSYNC cases since the elimination of the journal commit penalty altogether significantly boosts performance. In both NOJC and NOFSYNC cases, the performance rises steadily correlated to increases in processor frequency. This confirms the findings in our modelling and analysis: namely, that excessive journal commits lead to poor performance at some frequencies for Metarates.

4.1.2 IOZone Performance and ABTs

Figure 12 (see Appendix) repeats the experimental results for IOZone with the ext4 configuration with the addition of the NOJC and NOFSYNC results. Figure 12a contains 4 graphs. The second graph from the top depicts slowdowns (speedup < 1) in the default ext4 configuration also observed in Figure 3. The third (NOJC) and fourth (NOFSYNC) graphs from the top depict speedups as the processor frequency increases from left to right. Once again, the NOJC and NOFSYNC graphs show speedups correlating with increases in processor speed (i.e. no slowdowns). A similar phenomenon is evident in Figures 12b and 12c for other combinations of threads, file sizes, and records.

The topmost graph in each of Figures 12a, 12b, and 12c overlays the IOZone raw performance data for each of the three tests (ext4, NOJC, and NOFSYNC). In both NOJC and

NOFSYNC cases, the performance rises steadily correlated to increases in processor frequency. This confirms the findings in our modelling and analysis: namely, that excessive journal commits lead to poor performance at some frequencies for IOZone.

5. RELATED WORK

Slowdowns at higher frequencies have been observed in various contexts including: during MPI communication phases [4, 18, 29]; during parallel I/O phases [6-8]; on Fourier transform codes [1, 27]; on parallel fluid dynamics codes [17]; on vehicle scheduling codes [21, 25, 26]; on the MapReduce Sort benchmark [30], on hard disk drive systems [20, 23], on synthetic CPU bound codes [9, 15, 16], and in memory architectures [2, 19, 24].

Despite the number and diversity of these experiments, none of this work conclusively identifies the root cause of the slowdowns. Some leave the investigation to future work [12] [4], while others offer unsubstantiated (but reasonable) hypotheses citing bus interaction [5, 14, 30], synchronization [17] and system- or benchmark-specific details [10, 11, 13].

Based on our findings in the literature and to the best of our knowledge, no detailed scientific studies of power-scalable slowdowns had been conducted prior to our work. None of the aforementioned work explore slowdowns in much detail nor do they propose: a methodology to isolate slowdowns, a model of slowdown behavior in atomic batch transactions in power-scalable systems, and a case study demonstrating the tradeoffs between eliminating slowdown at the cost of reliability.

6. LIMITS, CONCLUSIONS, FUTURE

Our results are limited to the systems and benchmarks studied. Broader studies could yield broader conclusions. For example, we have measured slowdowns in varMail, MySQL, and TPC-C, but application complexity makes isolation and modelling more difficult. Additionally, our approach only identified a particular cause of slowdown; there may be others. Our modelling efforts focused strictly on performance to isolate the cause of slowdowns. Future versions would be more useful if they integrated the effects of power scaling directly in the models for prediction.

Overall, we have conclusively shown that in some circumstances, higher power and processing speeds can cause harm. Though some have noted this previously, none have isolated the root cause of the significant slowdowns that we observed. Our results indicate that system complexity is introducing unexpected, counter-intuitive performance issues that are increasingly difficult to isolate due to high system variance. In future work, we would like to isolate other root causes of slowdown. Though we were able to improve performance up to 68%, removing the Journal Commit to improve performance is not ideal since it hurts reliability. We would like to create techniques that enable ABTs without conflict with power scalable features.

7. ACKNOWLEDGMENTS

[Redacted to maintain anonymity]

8. REFERENCES

[1] Cheng, Y. and Zeng, Y., 2011. Automatic Energy Status Controlling with Dynamic Voltage Scaling in Power-Aware High Performance Computing Cluster. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*, 412-416. DOI= http://dx.doi.org/10.1109/PDCAT.2011.24.

[2] David, H., Fallin, C., Gorbatov, E., Hanebutte, U.R., and Mutlu, O., 2011. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the Proceedings of the 8th ACM international conference on Autonomic computing* (Karlsruhe, Germany2011), ACM, 1998590, 31-40. DOI= http://dx.doi.org/10.1145/1998582.1998590.

[3] Duda, R.O., Hart, P.E., and Stork, D.G., 2012. *Pattern classification*. John Wiley & Sons.

[4] Etinski, M., Corbalan, J., Labarta, J., and Valero, M., 2012. Understanding the future of energy-performance trade-off via DVFS in HPC environments. *J. Parallel Distrib. Comput. 72*, 4, 579-590. DOI= http://dx.doi.org/10.1016/j.jpdc.2012.01.006.

[5] Freeh, V.W. and Lowenthal, D.K., 2005. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proceedings of the Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (Chicago, IL, USA2005), ACM, 1065967, 164-173. DOI= http://dx.doi.org/10.1145/1065944.1065967.

[6] Ge, R., 2010. Evaluating Parallel I/O Energy Efficiency. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, 213-220. DOI= http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.130.

[7] Ge, R., Feng, X., S., S., and Sun, X.-H., 2010. Characterizing energy efficiency of I/O intensive parallel applications on power-aware clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 1-8. DOI= http://dx.doi.org/10.1109/IPDPSW.2010.5470904.

[8] Ge, R., Feng, X., and Sun, X.-H., 2012. SERA-IO: Integrating Energy Consciousness into Parallel I/O Middleware. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, 204-211. DOI= http://dx.doi.org/10.1109/CCGrid.2012.39.

[9] Ghiasi, S., Keller, T., and Rawson, F., 2005. Scheduling for heterogeneous processors in server systems. In *Proceedings of the Proceedings of the 2nd conference on Computing frontiers* (Ischia, Italy2005), ACM, 1062295, 199-210. DOI= http://dx.doi.org/10.1145/1062261.1062295.

[10] Grant, R.E. and Afsahi, A., 2007. Improving system efficiency through scheduling and power management. In *Proceedings of the Cluster Computing, 2007 IEEE International Conference on* (17-20 Sept. 2007 2007), 478-479. DOI= http://dx.doi.org/10.1109/CLUSTR.2007.4629271.

[11] Grant, R.E. and Afsahi, A., 2009. Improving energy efficiency of asymmetric chip multithreaded multiprocessors through reduced OS noise scheduling. *Concurr. Comput. : Pract. Exper. 21*, 18, 2355-2376. DOI= http://dx.doi.org/10.1002/cpe.v21:18.

[12] Hotta, Y., Sato, M., Kimura, H., Matsuoka, S., Boku, T., and Takahashi, D., 2006. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *Proceedings of the Proceedings of the 20th international conference on Parallel and distributed processing* (Rhodes Island, Greece2006), IEEE Computer Society, 1898825, 298-298.

[13] Hruby, T., Bos, H., and Tanenbaum, A.S., 2013. When Slower is Faster: On Heterogeneous Multicores for Reliable

Systems. In *2013 USENIX Annual Technical Conference (USENIX ATC '13)* USENIX, San Jose, CA, USA.

[14] Hsu, C.-H. and Feng, W.-C., 2005. A Power-Aware Run-Time System for High-Performance Computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 1-1. DOI= http://dx.doi.org/10.1109/SC.2005.3.

[15] Kotla, R., Devgan, A., Ghiasi, S., Keller, T., and Rawson, F., 2004. Characterizing the impact of different memory-intensity levels. In *Workload Characterization, 2004. WWC-7. 2004 IEEE International Workshop on*, 3-10. DOI= http://dx.doi.org/10.1109/WWC.2004.1437388.

[16] Kotla, R., Ghiasi, S., Keller, T., and Rawson, F., 2005. Scheduling processor voltage and frequency in server and cluster systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 8 pp. DOI= http://dx.doi.org/10.1109/IPDPS.2005.392.

[17] Lakshminarayana, N.B. and Kim, H., 2008. Understanding performance, power and energy behavior in asymmetric multiprocessors. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, 471-477. DOI= http://dx.doi.org/10.1109/ICCD.2008.4751903.

[18] Lim, M.Y., Freeh, V.W., and Lowenthal, D.K., 2006. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *Proceedings of the Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (Tampa, Florida2006), ACM, 1188567, 107. DOI= http://dx.doi.org/10.1145/1188455.1188567.

[19] Malkowski, K., Link, G., Raghavan, P., and Irwin, M.J., 2007. Load Miss Prediction - Exploiting Power Performance Trade-offs. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 1-8. DOI= http://dx.doi.org/10.1109/IPDPS.2007.370536.

[20] Miyoshi, A., Lefurgy, C., Hensbergen, E.V., Rajamony, R., and Rajkumar, R., 2002. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the Proceedings of the 16th international conference on Supercomputing* (New York, New York, USA2002), ACM, 514200, 35-44. DOI= http://dx.doi.org/10.1145/514191.514200.

[21] Pan, F., Freeh, V.W., and Smith, D.M., 2005. Exploring the energy-time tradeoff in high-performance computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 9 pp. DOI= http://dx.doi.org/10.1109/IPDPS.2005.213.

[22] Philip, C., Samuel, L., Robert, R., Murali, V., Julian, K., and Thomas, L., 2009. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* IEEE, 1-11.

[23] Saito, T., Sato, K., Sato, H., and Matsuoka, S., 2013. Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system. In *Proceedings of the Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale* (New York, New York, USA2013), ACM, 2465822, 41-48. DOI= http://dx.doi.org/10.1145/2465813.2465822.

[24] Schone, R., Hackenberg, D., and Molka, D., 2012. Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors. In *Proceedings of the Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems* (Hollywood, CA2012), USENIX Association, 2387878, 9-9.

[25] Srinivasan, S., Zhao, L., Illikkal, R., and Iyer, R., 2011. Efficient interaction between OS and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev. 45*, 1, 62-72. DOI= http://dx.doi.org/10.1145/1945023.1945032.

[26] Sueur, E.L. and Heiser, G., 2010. Dynamic voltage and frequency scaling: the laws of diminishing returns. In *Proceedings of the Proceedings of the 2010 international conference on Power aware computing and systems* (Vancouver, BC, Canada2010), USENIX Association, 1924921, 1-8.

[27] Telgarsky, M., Hoe, J.C., and Moura, J.M.F., 2006. SPIRAL: Joint runtime and energy optimization of linear transforms. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (2006).

[28] Traeger, A., Zadok, E., Joukov, N., and Wright, C.P., 2008. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4*, 2, 5.

[29] Wang, L., Laszewski, G.V., Dayal, J., and Wang, F., 2010. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *Proceedings of the Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), IEEE Computer Society, 1845153, 368-377. DOI= http://dx.doi.org/10.1109/ccgrid.2010.19.

[30] Wirtz, T. and Ge, R., 2011. Improving MapReduce energy efficiency for computation intensive workloads. In *Proceedings of the Proceedings of the 2011 International Green Computing Conference and Workshops* (2011), IEEE Computer Society, 2193362, 1-8. DOI= http://dx.doi.org/10.1109/igcc.2011.6008564.

9. APPENDIX

Algorithm 1 For the Metarates FCR test to calculate how much time the threads spend on their fsync ops.

For K and **L,**

#Find the thread that triggers the journal commit

$\boldsymbol{Find}\ i\ \boldsymbol{such\ that}\ 1 \le i \le \mathrm{N}\ \textbf{and}\ \boldsymbol{t}_{K_i}(\boldsymbol{q_{fsync}}) < \boldsymbol{t}_{K_m}(\boldsymbol{q_{fsync}})\ \forall\ 1 \le m \le \mathrm{N},\ \boldsymbol{and}\ \forall\ K_i, K_m \in \mathbf{K}$

$\boldsymbol{Find}\ p\ \boldsymbol{such\ that}\ 1 \le p \le 0\ \textbf{and}\ \boldsymbol{t}_{L_p}(\boldsymbol{q_{JC}}) < \boldsymbol{t}_{L_u}(\boldsymbol{q_{JC}})\ \forall\ 1 \le u \le 0,\ \boldsymbol{and}\ \forall\ L_p, L_u \in \mathbf{L}$

$\boldsymbol{T_{fsync}}(K_i) = \left(\boldsymbol{t}_{L_p}(\boldsymbol{q_{JC}}) - \boldsymbol{t}_{K_i}(\boldsymbol{q_{fsync}})\right) + \boldsymbol{T_{JC}}(L_p)$

$\boldsymbol{K} = \boldsymbol{K} - \{K_i\}$

$\boldsymbol{L} = \boldsymbol{L} - \{L_p\}$

$\forall\ K_j \in \boldsymbol{K}\ \boldsymbol{and}\ i \ne j,$

$\boldsymbol{if}\ \boldsymbol{t}_{K_j}(\boldsymbol{q_{fsync}}) < \boldsymbol{t}_{L_p}(\boldsymbol{q_{JC}})\ \boldsymbol{then}$ *#Thread waits for this journal commit*

$\boldsymbol{T_{fsync}}(K_j) = \left(\boldsymbol{t}_{L_p}(\boldsymbol{q_{JC}}) - \boldsymbol{t}_{K_j}(\boldsymbol{q_{fsync}})\right) + \boldsymbol{T_{JC}}(L_p)$

$\boldsymbol{K} = \boldsymbol{K} - \{K_j\}$

Repeat until $\boldsymbol{K} = \{\emptyset\}, \boldsymbol{L} = \{\emptyset\}$

Algorithm 2 Calculate how much time the threads spend on their write ops for the IOZone throughput benchmark.

#Find thread starts relative to journal commit

$\boldsymbol{Find}\ i\ \boldsymbol{such\ that}\ 1 \le i \le \mathrm{N}\ \textbf{and}\ t_{K_i}(\boldsymbol{r_{DFLUSH}}) < t_{K_m}(\boldsymbol{r_{DFLUSH}})\ \forall\ 1 \le \mathrm{m} \le \mathrm{N}\ \boldsymbol{and}\ \boldsymbol{X} = \{\emptyset\},\ \boldsymbol{Y} = \{\emptyset\}, \boldsymbol{Z} = \{\emptyset\}$

$\boldsymbol{T_{write}}(K_i) = \boldsymbol{t}_{K_i}(\boldsymbol{r_{write}}) - \boldsymbol{t}_{K_i}(\boldsymbol{q_{write}})$

$\boldsymbol{X} = \boldsymbol{X} + \{K_i\}$

$\forall\ K_j \in \boldsymbol{K},\ i \ne j$

$\boldsymbol{if}\ \boldsymbol{t}_{K_j}(\boldsymbol{q_{fsync}}) < \boldsymbol{t}_{L_1}(\boldsymbol{q_{JC}})\ \boldsymbol{then}$ *#Thread not affected by journal commit*

$\boldsymbol{T_{write}}(K_j) = \boldsymbol{t}_{K_j}(\boldsymbol{r_{write}}) - \boldsymbol{t}_{K_j}(\boldsymbol{q_{write}})$

$\boldsymbol{X} = \boldsymbol{X} + \{K_j\}$

$\boldsymbol{if}\ \boldsymbol{t}_{K_j}(\boldsymbol{q_{fsync}}) > \boldsymbol{t}_{L_1}(\boldsymbol{q_{JC}})\ \boldsymbol{then}$ *#Thread affected by journal commit*

$\boldsymbol{if}\ \boldsymbol{t}_{K_j}(\boldsymbol{q_{write}}) < \boldsymbol{t}_{L_1}(\boldsymbol{q_{JC}})\ \boldsymbol{then}$ *#Write starts before journal commit*

$\mathbf{T_{write}}(\mathbf{K_j}) = \left(\mathbf{t_{L_1}}(\mathbf{q_{JC}}) - \mathbf{t_{K_j}}(\mathbf{q_{write}})\right) + \mathbf{T_{JC}}(\mathbf{L_1}) + \left(\mathbf{t_{K_j}}(\mathbf{r_{write}}) - \mathbf{t_{L_1}}(\mathbf{r_{JC}})\right)$

$\boldsymbol{Y} = \boldsymbol{Y} + \{K_j\}$

$\boldsymbol{if}\ \boldsymbol{t}_{K_j}(\boldsymbol{q_{write}}) > \boldsymbol{t}_{L_1}(\boldsymbol{q_{JC}})\ \boldsymbol{then}$ *#Write starts after journal commit*

$\boldsymbol{T_{write}}(K_j) = \left(\boldsymbol{t}_{L_1}(\boldsymbol{r_{JC}}) - \boldsymbol{t}_{K_j}(\boldsymbol{q_{write}})\right) + \left(\boldsymbol{t}_{K_j}(\boldsymbol{r_{write}}) - \boldsymbol{t}_{L_1}(\boldsymbol{r_{JC}})\right)$

$\boldsymbol{Z} = \boldsymbol{Z} + \{K_j\}$

End

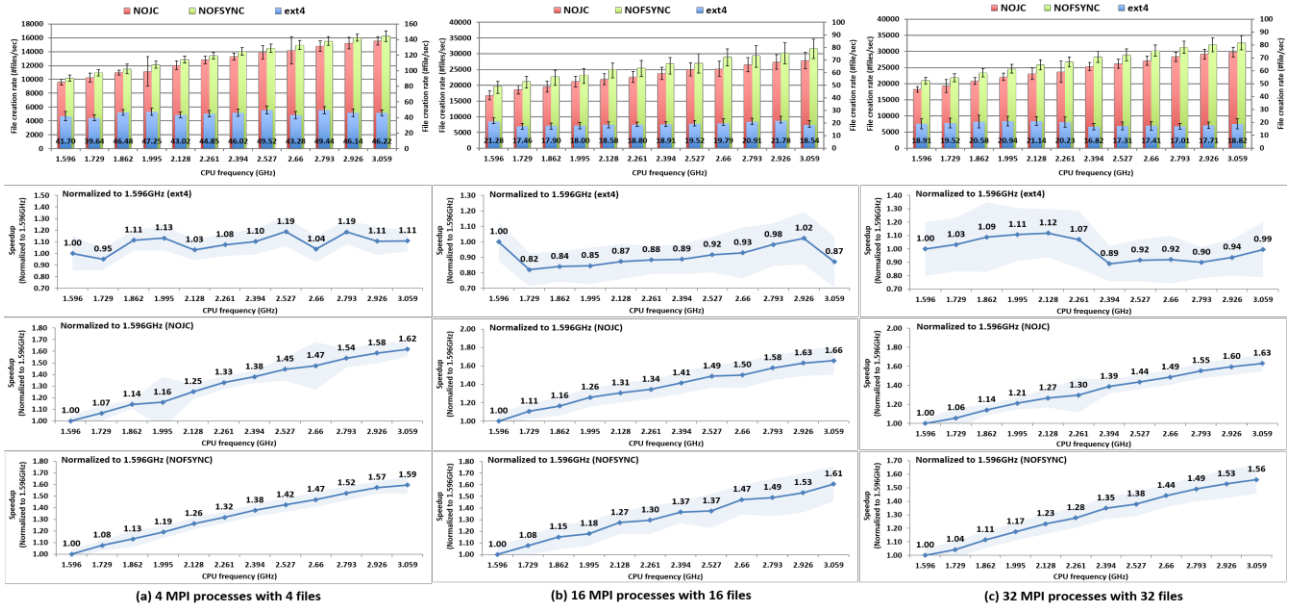

Figure 11. Select results for the Metarates benchmark on the Nehalem (HDD) system. Findings are comparable on the other systems though not included due to space limitations.

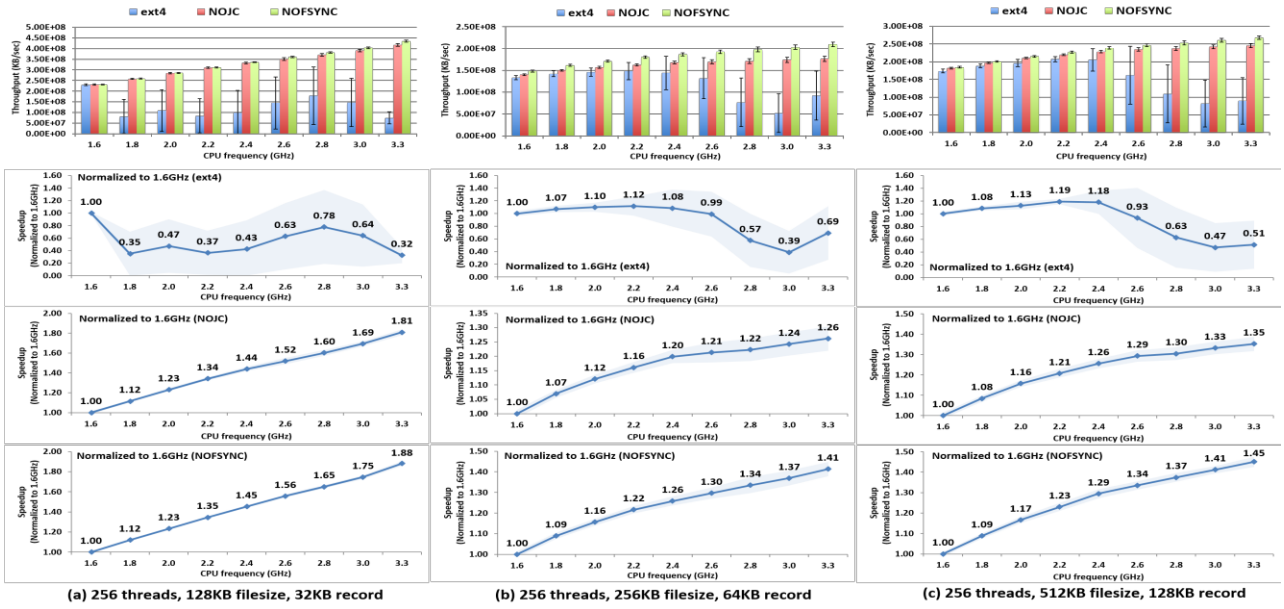

Figure 12. Select results for the IOZone benchmark on the SandyBridge (HDD) system. Findings are comparable on the other systems though not included due to space limitations.