How Processor Speedups Can Slow Down I/O Performance

[1]Hung-Ching Chang, [1]Bo Li, [2]Matthew Grove, and [1]Kirk W. Cameron

[1]Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
{hcchang, bxl4074, cameron}@cs.vt.edu

[2]Rackspace
Blacksburg, VA 24060
mat.grove@rackspace.com

***Abstract*—Power states in power-scalable systems are managed to maximize performance and reduce energy waste. Power-scalable processor capabilities (e.g., Intel Turbo Boost) embrace a "faster is better" approach to power management. While these technologies can vastly improve performance and energy efficiency, there is a growing body of evidence that "faster is not always better". For example, in some I/O intensive benchmarks, we observe up to 47% performance loss when running codes at faster (higher power) frequencies versus slower (lower power) frequencies. To the best of our knowledge, this is the first work to systematically and accurately pinpoint the root cause of these types of slowdowns. The lack of such studies is likely due to three challenges we overcome in this work: 1) high runtime system variance; 2) bottleneck isolation across user- and system-space boundaries; and 3) non-determinism in parallel codes. Our analytical model-driven approach identifies Atomic Batch Transactions (ABTs) in the Linux kernel as the cause of slowdowns at higher processor speeds. We propose and evaluate the use of power-aware ABT's that can increase performance more than 3-fold over the default Linux kernel while maintaining comparable reliability. Our work motivates the need for more studies that potentially reconsider the "faster is better" design paradigm**.

***Keywords*—Performance, energy efficiency, atomic batch transactions.**

I. 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 (KB/sec) for the same system running at two different (fixed) processor frequencies (1.6 GHz and 3.3 GHz). Normalizing to the performance at the lowest available frequency for this IOzone run (1.6 GHz), the IOzone benchmark, which is typically used to compare vendor I/O performance across systems, runs 47% slower at the highest available frequency (3.3 GHz)[1].

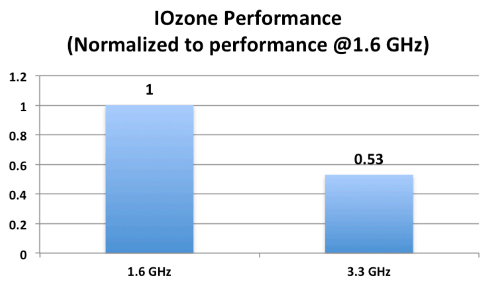

Fig. 1. IOzone performance on a Dell T1100 using a Xeon E3-1270 3300 MHz (SandyBridge) quad-core with 8GB of DDR3 RAM and a 250 GB 7200 rpm hard drive. The system runs the CentOS 6.4 Linux distribution (kernel version 3.4.2).

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 [1-21] have independently confirmed these types of slowdowns.

In some cases [1-6, 10-12, 14-21], 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. The goal of our work is to explain the root cause of significant slowdowns we've observed in I/O performance.

Challenges. Unfortunately, isolating the cause of such slowdowns is non-trivial. First, I/O system measurements are noisy which makes simply identifying slowdowns problematic. Second, even the simplest multi-threaded I/O codes are extremely complex with parallel critical paths

[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.

that repeatedly cross user- and kernel space boundaries due to numerous calls to user and system libraries. Isolating the critical path requires intimate knowledge of both the application being studied and the performance path and nuances of the operating system kernel. Third, thread arrival times and their use of resources in parallel I/O codes are inherently non-deterministic. Nonetheless, highly parallel codes with high variance, complicated critical paths, and inherent non-determinism are exactly the codes that exhibit slowdown behavior. Furthermore, experimental variance, the complexity of critical paths, and non-determinism are steadily increasing in emergent systems exacerbating the challenge of isolating slowdowns and their root causes.

Contributions. Our main contribution is the identification of resource contention among I/O threads as the root cause of the observed I/O slowdowns. With the use of code and kernel instrumentation, exhaustive experiments, and deep insight to the inner workings of the Linux I/O subsystem, we overcome the aforementioned challenges of variance, complexity and non-determinism. We derive an analytical model to explain the behavior of two parallel I/O benchmarks that exhibit significant slowdown when processors speed up. We verify our findings experimentally and propose an adaptive runtime system to avoid slowdowns during processor speedups.

II. EXPERIMENTAL APPROACH

We began our work with exhaustive testing for significant slowdowns across a large number of applications suggested by the literature [1-6]. Figures 2 and 3 show the average speedup from select IOzone and Metarates experiments[2]. *IOzone* is a file system benchmark that generates and measures a variety of file operations. This benchmark is part of the Phoronix Test Suite[3] used by review sites including Tom's Hardware[4]. 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).

Figures 2 and 3 demonstrate the aforementioned variance challenge. 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 11-55%; Metarates for a given number of threads (4) and files (4) on the system running at all available frequencies exhibits RSDs from 13-32%. We overcome these significant variances using exhaustive testing. We run all of the combinations of benchmark parameters mentioned across all the available P-states for each processor. 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 [23]. A full sweep across these parameter sets takes about one and a half months to complete.

IOzone and Metarate were selected for detailed study and modelling for several reasons. First the benchmarks themselves showed acute sensitivity to processor speeds as observed in Figures 2 and 3. Second, each thread performs a short sequence of I/O read and write operations. This reduces the amount of complexity in the critical path for each thread. This allowed us to account for non-determinism in parallel thread execution by grouping threads based upon their arrival order at shared resources. Third, the benchmarks are used to compare hardware I/O implementations which means slowdowns could be exploited to game results that affect user adoption. Furthermore, these benchmarks represent many types of common database transactions with high I/O frequency.

In Figures 2 and 3, 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 47% slower than running on the same system at 1.6GHz; Metarates for a given number of threads (4) and files (4) on the system running at 3.1GHz is 20% slower than running on the same system running at 1.6GHz.

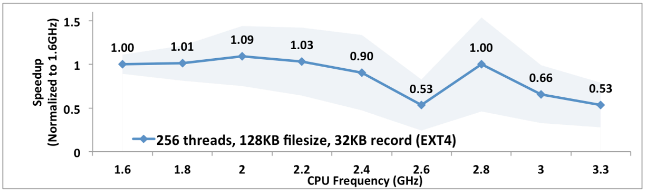

Fig. 2. Select results from IOzone benchmark on *SandyBridge (HDD)*. Findings are comparable on the other systems though not included due to space limitations. *SandyBridge HDD* is a Dell T1100 using a Xeon E3-1270 3.3 GHz quad-core with 8 GB of DDR3 RAM and a 250 GB 7200 rpm hard drive.

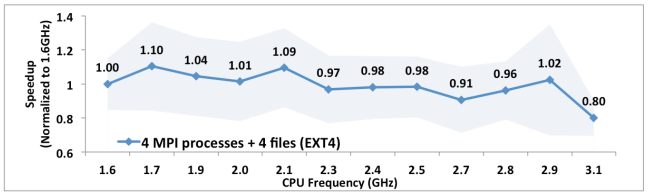

Fig. 3. Select results from Metarates benchmark on *Nehalem (HDD)*. Findings are comparable on the other systems though not included due to space limitations. *Nehalem HDD* is a Dell T3500 using a W3550 3.00 GHz () quad-core with 6 GB of DDR3 RAM and a 250 GB 7200 rpm hard drive.

[2] We disable the turbo boost and hyper-threading features so we can manually isolate performance at each static frequency and isolate slowdowns.

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

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

III. MODELING IOZONE AND METARATES

Following a detailed analysis of the IOzone benchmark and a deep dive on the Linux I/O subsystem particulars, Figure 4 shows our conceptual view of the critical path of a single IOzone thread.

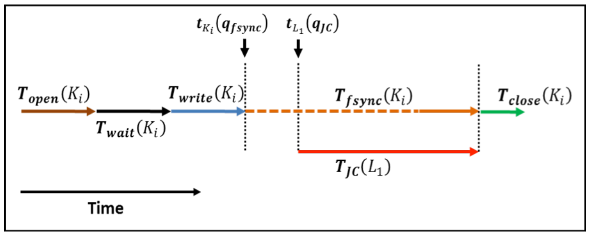

Fig. 4. The interaction between a thread and a journal process for the IOzone benchmark.

We observe that at a high-level Figure 4 shows where time is spent in a given thread. Every thread in IOzone (from left to right in the figure) begins by opening a file, then waiting, then writing to a file. Following the write, an fsync operation occurs causing the critical path to enter kernel-space by invoking a kernel function called a journal commit (JC). When the kernel finishes its commit, control returns to the thread which closes the file. To ease description of what happens when many such threads run in parallel, we use an analytical model. While the model is used to isolate the cause of slowdowns, users may jump to the end of this section for a high-level explanation of our derived findings.

A. Analytical Model of IOzone Performance

We now present an analytical model of atomic batch transaction for IOzone performance. The model uses the parameters measured from the aforementioned Linux kernel instrumentation (see Tables 2 and 3).

1) User-space I/O Threads

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, $t_{K_i}(q_{op})$

TABLE 1. 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 2. 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

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.

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

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 (We assume ordered mode journaling; the default configuration) 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. For now, we assume that the kernel journal process issues only one operation: the journal commit (**JC**) op, but we will remove this constraint shortly (see Figure 6).

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.

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

The kernel-space journal processes may influence the user-space I/O threads during their write ops and fsync ops. The IOzone benchmark threads test the file system's read-write performance by using POSIX system calls (e.g., open(), write(), fsync(), and close() functions). Specifically, IOzone 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. The time spent writing the data to a file by each thread is measured as the result, and the benchmark performance metric is calculated by accumulating the results from all threads. From (1) we have:

$$\text{IOzone Throughput}_{\text{initial\_write}} = \sum_{i=1}^{N} \frac{S}{\mathrm{T_{write}}(\mathrm{K_i})}. \tag{3}$$

From the IOzone benchmark metric, we observe that variance in reported results occurs in the measured time for the write operation ($\mathrm{T_{write}}(K_i)$). The aforementioned high variances come from this operation and, as mentioned, we use a brute force exhaustive approach to obtain statistical confidence of 95% in all of our experiments.

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 4 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.

Figure 5 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

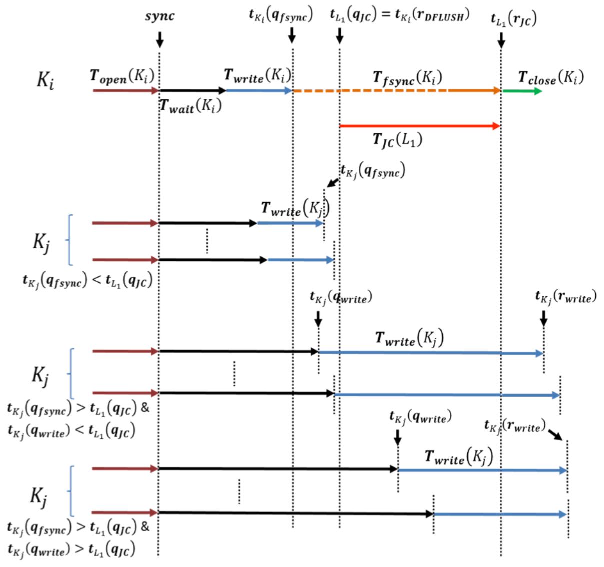

Fig. 5. The runtime profile for the IOzone throughput test with multiple user-space threads and one journal process.

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_1starts its **JC** op, these **write** ops are not blocked, using equation (1):

$$\mathrm{T_{write}}(K_j) = t_{K_j}(r_{write}) - t_{K_j}(q_{write}). \qquad (4)$$

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_1begins its **JC** op and the K_j processes finish their **write** ops after L_1completes its **JC** op, these **write** ops are fully blocked by the **JC** op of L_1, using equation (1) and (2):

$$\mathrm{T_{write}}(K_j) = \left(t_{L_1}(q_{JC}) - t_{K_j}(q_{write})\right) + \mathrm{T_{JC}}(L_1) + \left(t_{K_j}(r_{write}) - t_{L_1}(r_{JC})\right). \qquad (5)$$

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_1begins its **JC** op and the K_i processes finish their **write** ops after L_1completes its **JC** op, these **write** ops are blocked by the partial **JC** op of L_1, using equation (1) and (2):

$$\mathrm{T_{write}}(K_j) = \left(t_{L_1}(r_{JC}) - t_{K_j}(q_{write})\right) + \left(t_{K_j}(r_{write}) - t_{L_1}(r_{JC})\right). \qquad (6)$$

3) *IOzone and Slowdown*

According to the status of **write** ops of K_js when L_p begins its **JC** op, these K_js can be classified into three cases, **X**, **Y**, and **Z**. **X** collects the K_js so that their **write** ops are not affected by the **JC** op of L_p; **Y** and **Z** collect the K_js so that their **write** ops are fully or partially blocked by the **JC** op of L_p, respectively. We replace $\mathrm{T_{write}}(K_j)$ with the three cases:

$$= \sum_{\mathrm{K_x \in X}} \frac{\mathbf{S}}{\mathrm{T_{write}}(K_x)} + \sum_{\mathrm{K_y \in Y}} \frac{\mathbf{S}}{\mathrm{T_{write}}(K_y)} + \sum_{\mathrm{K_z \in Z}} \frac{\mathbf{S}}{\mathrm{T_{write}}(K_z)}. \qquad (7)$$

We then substitute $\mathrm{T_{write}}(K_x)$, $\mathrm{T_{write}}(K_y)$, and $\mathrm{T_{write}}(K_z)$ with the cost functions from (4), (5), and (6) respectively:

$$= \sum_{\mathrm{K_x \in X}} \frac{\mathbf{S}}{t_{K_x}(r_{write}) - t_{K_x}(q_{write})} + \sum_{\mathrm{K_y \in Y}} \frac{\mathbf{S}}{\left(t_{L_1}(q_{JC}) - t_{K_y}(q_{write})\right) + \mathrm{T_{JC}}(L_1) + \left(t_{K_y}(r_{write}) - t_{L_1}(r_{JC})\right)} + \sum_{\mathrm{K_z \in Z}} \frac{\mathbf{S}}{\left(t_{L_1}(r_{JC}) - t_{K_z}(q_{write})\right) + \left(t_{K_z}(r_{write}) - t_{L_1}(r_{JC})\right)}. \qquad (8)$$

The optimal performance of the throughput test can be expected when there is no K_m that falls in the **Y** and **Z** scenarios. In other words, every K_j is not affected by the **JC** op of L_p, and all K_js fall in the **X** category. Thus, $K_j \in \mathbf{K}$ finishes its **write** op before L_p begins its **JC** op, and K_j satisfies $t_{K_j}(q_{fsync}) < t_{L_p}(q_{JC})$.

The slowdown happens when more threads in the **Y** and **Z** categories are at a higher frequency. In other words, more threads are blocked by the **JC** op. These blocked threads spend their time in the **write** op plus the time waiting for the **JC** op and increase $\mathrm{T_{write}}(K_j)$ as shown in (5) and (6). As a result, these blocked threads have a negative impact on the performance measurement of IOzone throughput tests.

B. *Analytical Model of Metarates Performance*

Metarates is a file system benchmark that measures the performance of concurrent aggregate metadata transaction rates in extremely large file systems. Metarates includes the file creation rate (FCR) test that launches MPI processes, $K_m \in \mathbf{K}, m = 1...\mathrm{N}$, where each MPI process in parallel issues a sequence of POSIX system calls (e.g., open(), fsync(), and close() functions) to complete its task. The time each MPI process takes to complete its task is measured, and the average time for all MPI processes to

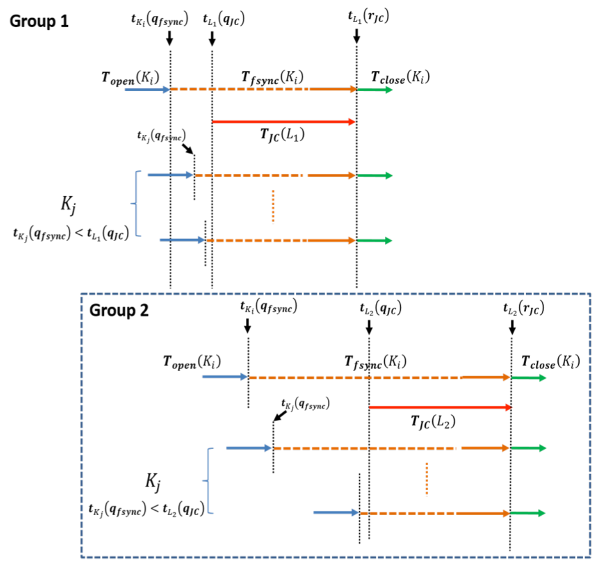

Fig. 6. The Metarates runtime profile for two or more user-space MPI processes and two journal processes.

complete their tasks is used as the performance metric of the Metarates file creation rate (FCR) test. From (1) we have:

$$\text{File creation rate (FCR)} = \frac{\mathbf{N} \times 1}{\sum_{m=1}^{N}\left(\mathrm{T_{open}}(K_m)+\mathrm{T_{fsync}}(K_m)+\mathrm{T_{close}}(K_m)\right)}. \quad (9)$$

1) Metarates and Journal Commits

Figure 6 shows the critical paths for many Metarate threads for two journal commits. 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 journal commits, L_1 and L_2. In Group 1, 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. The K_j processes in Group 1 meet the $t_{K_j}(q_{fsync}) < t_{L_1}(q_{JC})$ condition, and these K_js are not blocked. Group 2 collects the MPI processes that start after the first journal process begins its **JC** op $t_{L_1}(q_{JC})$, Thus, the K_j processes in Group 2 have to wait until L_1 completes its **JC** op before L_2 can start its **JC** op, and therefore the wait time $\left(\boldsymbol{t_{L_1}(q_{JC}) - t_{K_j}(q_{fsync})}\right)$ of K_j is significant.

2) Metarates and Slowdown

In our measurement, we find that $\mathrm{T_{fsync}(K_i)}$ is significantly larger compared with $\mathrm{T_{open}(K_i)}$ and $\mathrm{T_{close}(K_i)}$. This is because an **fsync** op requires storage device accesses, while the **open** and **close** ops involve only memory accesses. From (9), we remove $\mathrm{T_{open}}(K_m)$ and $\mathrm{T_{close}}(K_m)$ since $\mathrm{T_{open}}(K_m)$ and $\mathrm{T_{close}}(K_m)$ are relatively small compared to $\mathrm{T_{fsync}}(K_m)$;

$$\text{FCR} \sim \frac{\mathbf{N} \times 1}{\sum_{m=1}^{N}\left(\mathrm{T_{fsync}}(K_m)\right)}. \quad (10)$$

Since **N** is constant, the optimal performance for FCR exists when $\sum_{m=1}^{N}\left(\mathrm{T_{fsync}}(K_m)\right)$ is minimal. We conclude that every $K_m \in \mathbf{K}, m = 1...\mathrm{N}$ waits for L_p on its **fsync** op. In this situation, all K_ms satisfy $t_{K_m}(q_{fsync}) < t_{L_p}(q_{JC})$. Thus, no K_p suffers from waiting for the second journal process, L_{p+1}.

The slowdown happens when at higher frequencies more threads spend significant amounts of time waiting for the next **JC** op. These threads are blocked and are waiting for the current **JC** op to complete before their **JC** op can start. The additional waiting time for these blocked threads increases $\mathrm{T_{fsync}}(K_m)$ and has a negative impact on the measurement of the Metarates FCR test.

C. Summary Explanation of Slowdowns

Our analytical model shows that slowdowns occur at higher frequencies when the early arrival of a single thread (among many) causes the atomic 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 journal commit locks, more threads are batched and overall less atomic batches occur. Slower processor frequencies effectively increase the number of threads that access the shared resource while reduce the overall commits required at higher processor frequencies.

IV. OPTIMIZING ABT PERFORMANCE

The parallel threads of the I/O codes studied use a Journal Commit (JC) 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. Our analytical models cast blame for slowdown on atomic batch transactions. 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. ABT's as implemented can hurt performance at higher frequencies) to a real system to remove the performance penalty when combining power scaling with the IOzone and Metarates benchmarks. Second, we design and implement a runtime system to reduce the number of threads blocked by the ABTs to improve the performance of IOzone and Metarates. Third, we quantify the impact of the proposed approach on the performance and reliability of IOzone and Metarates.

Figures 7 and 8 show the results of our experiments. In each graph, the x-axis denotes CPU frequency. For line plots, the gray area surrounding the line shows the normalized standard deviation. For bar charts, error bars are provided. ***EXT4*** denotes the Linux default configuration. ***NOJC*** denotes a modified fsync system call implementation where the Linux kernel flushes out the metadata directly to the disk location and altogether avoids use of the kernel journal commit ABT mechanism. ***delayJC*** denotes results from our proposed runtime system that vary the length of time an ABT batches instructions. The delayJC implementation uses the arrival rates of requests to the JC to drive the close of the JC. After testing a number of configurations for delayJC, we settled on a JC wait time between requests of 16ms for IOzone and 1ms for Metarates. In other words, if a JC arrives during this interval, the JC batch timer is reset allowing more requests. If not, timer expires and the JC is closed to further requests.

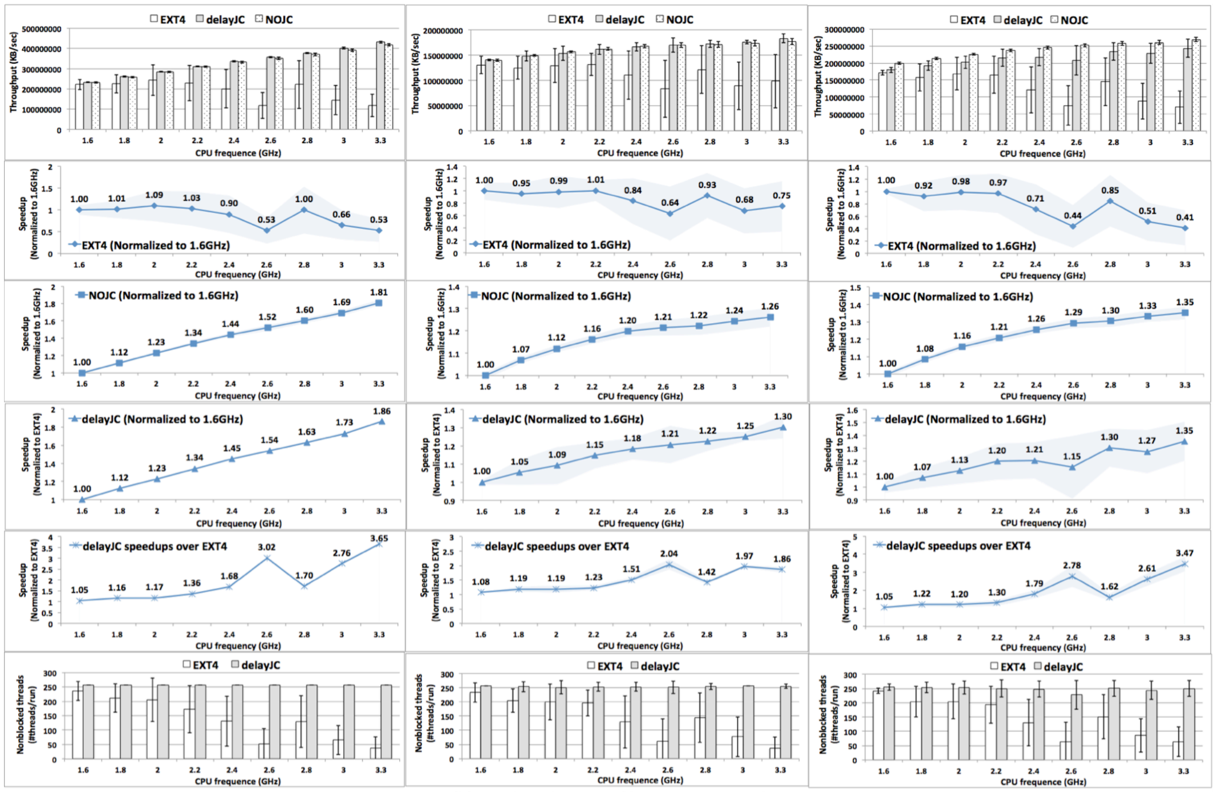

Fig. 7. Select results for the IOzone benchmark on the *SandyBridge (HDD)* system. Findings are comparable on the other systems though not included due to space limitations.

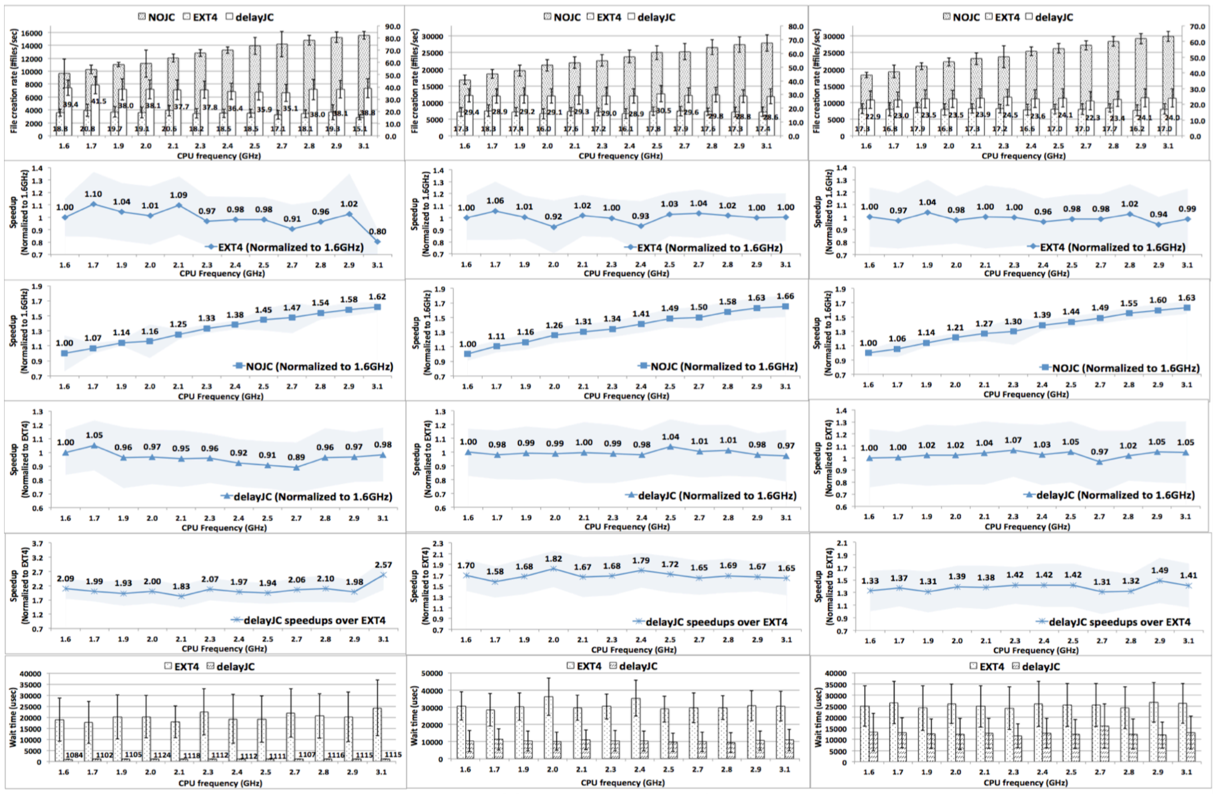

Fig. 8. Select results for the Metarates benchmark on the *Nehalem (HDD)* system. Findings are comparable on the other systems though not included due to space limitations.

The timer allows a maximum number of resets to avoid starvation.

We also consider the amount of time a thread waits to have its metadata committed. If this wait time is on average the same or less than that provided by the default EXT4 Linux kernel, we conclude our runtime techniques are at least as "reliable" as the default scheme.

A. *IOzone Results*

Figure 7 compares IOzone results from the default EXT4 configuration to the NOJC and delayJC runs. Each column of charts refers to a different thread and file size scenario. The y-axis on the topmost graphs in each column denotes the performance of the aggregated throughput in KB per second. In the NOJC configuration where the JC is avoided altogether, higher processor frequencies result in higher throughput. In contrast, the default EXT4 configuration performance varies wildly and exhibits slowdown at higher processor frequencies. But, in most cases, delayJC performance is comparable to NOJC performance and in all cases outperforms the EXT4 case.

Slowdowns (speedup < 1) and speedups versus the slowest CPU frequency are depicted in Figure 7 in the second row of graphs (EXT4), the third row of graphs (NOJC), and the fourth row of graphs (delayJC). Collectively, these graphs confirm performance improves noticeably in both the NOJC and delayJC configurations. Upon deeper inspection, we monitored the number of journal commit operations and found that in all cases, delayJC exhibited less journal commits than EXT4.

The fifth row of graphs from the top in Figure 7 shows the delayJC runtime system significantly improves the performance across all processor frequencies for combinations of threads, file sizes, and records. We observe speedups (computed as the ratio of delayJC throughput to EXT4 throughput at each frequency respectively) up to 3.65x for a combination of 256 threads, 128 KB file size, and 32 KB record running at 3.3 GHz in Figure 7a; up to 2.04x for a combination of 256 threads, 256 KB file size, and 64 KB record running at 2.6 GHz in Figure 7b; and up to 3.47x speedup for a combination of 256 threads, 512 KB file size, and 128 KB record running at 3.3 GHz in Figure 7c.

The sixth row of graphs from the top in Figure 7 depict the average number of threads "not" blocked by the journal commit during the threads' write operations. The EXT4 configuration shows less nonblocked threads at the higher frequency (e.g. 40 nonblocked threads at 3.3GHz versus 240 nonblocked threads at 1.6Ghz in Figure 7a), and thus causes the slowdown to happen as described analytically in equation (8) in Section III. The NOJC configuration disables the journal commit, so no threads get blocked – thus, we remove NOJC from these graphs. For the delayJC configuration, the number of non-blocked threads is close to the number of threads used in the test. In essence, the delayJC runtime system delays the journal commit until all (or nearly all) the threads complete their write operations. For the IOzone benchmark, the delayJC runtime system improves performance by reducing the number of blocked threads waiting on the journal commit. The average wait time per thread also decreases and thus the delayJC runtime system is at least as "reliable" as the EXT4 configuration.

B. *Metarates Results*

Figure 8 compares Metarates results from the default EXT4 configuration to the NOJC and delayJC runs. Each column of charts refers to a different thread and file number scenario. The y-axis on the topmost graphs in each column denotes the performance of Metarates. In the NOJC configuration where the JC is avoided altogether, higher processor frequencies result in higher throughput. The metarate critical path does not include a write operation (as in the IOzone case). Hence, performance is much more sensitive to the journal commit disk access variance. While the delayJC configuration consistently outperforms the EXT configuration, the delayJC misses some opportunities to avoid some slowdowns.

Slowdowns (speedup < 1) and speedups versus the slowest CPU frequency are depicted in Figure 8 in the second row of graphs (EXT4), the third row of graphs (NOJC), and the fourth row of graphs (delayJC). Collectively, these graphs confirm performance improves noticeably in the NOJC configuration. While the performance of delayJC is always better than EXT4, the ability of delayJC to eliminate slowdowns in Metarates is mixed. For higher numbers of threads, delayJC performance is consistent as processor frequency scales. However, for lower numbers of threads, disk noise in the journal commit masks some of the gains from the dynamic runtime system. To confirm the effects of delayJC, we monitored the number of journal commit operations and found that in all cases, delayJC exhibited less journal commits than EXT4.

Despite the mixed results in slowdown effectiveness versus eliminating the journal commit altogether, delayJC is a significant improvement over EXT4. The fifth row of graphs from the top in Figure 8 shows the delayJC runtime system significantly improves the performance across all processor frequencies for combinations of threads and number of files. We observe speedups (computed as the ratio of delayJC throughput to EXT4 throughput at each frequency respectively) up to 2.57x for 4 MPI processes with 4 files at 3.1 GHz in Figure 8a; up to 1.82x speedup for 16 MPI processes with 16 files at 2.0 GHz in Figure 8b; and up to 1.49x for 32 MPI processes with 32 files running at 2.9 GHz in Figure 8c.

The sixth row of graphs from the top in Figure 8 depicts the average time that a thread waits before it is serviced by the journal commit for both the EXT4 and the delayJC configurations. The EXT4 configuration shows longer wait times at higher frequencies (e.g. 25 ms wait time at 3.1GHz versus 18 ms wait time at 1.7GHz in Figure 8a), and thus causes the slowdown to happen as described analytically in equation (10) in Section III. The NOJC configuration disables the journal commit and has an average wait time of zero – thus, we remove NOJC from these graphs. For the delayJC configuration, the wait time is significantly reduced for the combination of 4 MPI processes and 4 files, as shown in 9a. This is because *SandyBridge (HDD)* has four CPU cores, and each of the four MPI processes starts roughly at the same time. The delayJC runtime system delays the journal commit for 1ms to sufficiently queue all the MPI processes for the same journal commit. When the number of MPI processes scales from 4 to 16 and 32, as shown in 9a, 9b, and 9c (respectively), we observe that the average wait time increases. In these configurations, the number of MPI threads exceeds the available number of cores. In such situations, threads are delayed further due to contention and the delayJC runtime adjusts ABT delays to compensate. The increase in average wait time among threads correlates to drops in overall speedups (longer wait time results in less speedup). For example, the average wait times for delays around 1ms show speedups of 1.83x–2.57x for 4 MPI processes (see Figure 8a). Similarly, the average wait times for 12 ms delay show speedups of 1.31x–1.49x for 32 MPI processes (see Figure 8c).

As in the IOzone case, the delayJC configuration reduces the average wait time for each thread on the journal commit. This means that compared to the EXT4 case, our delayJC runtime system reduces the likelihood that that (should a fault occur) metadata is not committed to the disk. We conclude that our delayJC runtime system is at least as reliable as the default Linux kernel EXT4 configuration.

RELATED WORK

Slowdowns at higher frequencies have been observed in various contexts including: during MPI communication phases [1-3]; during parallel I/O phases [4-6]; on Fourier transform codes [7, 8]; on parallel fluid dynamics codes [9]; on vehicle scheduling codes [10-12]; on the MapReduce Sort benchmark [13], on hard disk drive systems [14, 15], on synthetic CPU bound codes [16-18], and in memory architectures [19-21].

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 [25] [3], while others offer unsubstantiated (but reasonable) hypotheses citing bus interaction [13, 26, 27], synchronization [9] and system- or benchmark-specific details [28-30].

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 conclusively isolate the root cause of observed slowdowns at higher processor frequencies. In contrast, we isolate slowdowns, model the performance of atomic batch transactions in power-scalable systems, and implement a runtime system demonstrating the tradeoffs between eliminating slowdown at the cost of reliability.

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 such as residual slowdowns in Metarate. 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 such slowdowns. Our results indicate that system complexity is introducing unexpected, counter-intuitive performance issues that are increasingly difficult to isolate due to high system variance, critical path complexity, and non-determinism. In future work, we would like to isolate other root causes of slowdown. Though we were able to improve performance up to 3.65x over the EXT4 default configuration, our delayJC approach is somewhat static (fixed interval) and would likely be improved with additional runtime automated tuning. Ultimately, would like to create techniques that enable ABTs without conflict with power scalable features.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0910784 and 0905187.

REFERENCES

[1] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs," in *ACM/IEEE conference on Supercomputing*, Tampa, Florida, 2006, p. 107.

[2] L. Wang, G. v. Laszewski, J. Dayal, and F. Wang, "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS," presented at the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010.

[3] M. Etinski, J. Corbalan, J. Labarta, and M. Valero, "Understanding the future of energy-performance trade-off via DVFS in HPC environments," *J. Parallel Distrib. Comput.,* vol. 72, pp. 579-590, 2012.

[4] R. Ge, X. Feng, S. S., and X.-H. Sun, "Characterizing energy efficiency of I/O intensive parallel applications on power-aware clusters," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1-8.

[5] R. Ge, "Evaluating Parallel I/O Energy Efficiency," in *IEEE/ACM Int'l Conference on Green Computing and Communications (GreenCom) & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010, pp. 213-220.

[6] R. Ge, X. Feng, and X.-H. Sun, "SERA-IO: Integrating Energy Consciousness into Parallel I/O Middleware," in *12th IEEE/ACM*

International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2012, pp. 204-211.

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

[8] Y. Cheng and Y. Zeng, "Automatic Energy Status Controlling with Dynamic Voltage Scaling in Power-Aware High Performance Computing Cluster," in *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2011, pp. 412-416.

[9] N. B. Lakshminarayana and H. Kim, "Understanding performance, power and energy behavior in asymmetric multiprocessors," in *IEEE International Conference on Computer Design (ICCD)*, 2008, pp. 471-477.

[10] E. L. Sueur and G. Heiser, "Dynamic voltage and frequency scaling: the laws of diminishing returns," presented at the Proceedings of the 2010 international conference on Power aware computing and systems, Vancouver, BC, Canada, 2010.

[11] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer, "Efficient interaction between OS and architecture in heterogeneous platforms," *SIGOPS Oper. Syst. Rev.,* vol. 45, pp. 62-72, 2011.

[12] F. Pan, V. W. Freeh, and D. M. Smith, "Exploring the energy-time tradeoff in high-performance computing," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005, p. 9 pp.

[13] T. Wirtz and R. Ge, "Improving MapReduce energy efficiency for computation intensive workloads," presented at the International Green Computing Conference and Workshops, 2011.

[14] T. Saito, K. Sato, H. Sato, and S. Matsuoka, "Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system," presented at the 3rd Workshop on Fault-tolerance for HPC at extreme scale, New York, New York, USA, 2013.

[15] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: understanding the runtime effects of frequency scaling," presented at the 16th international conference on Supercomputing, New York, New York, USA, 2002.

[16] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson, "Characterizing the impact of different memory-intensity levels," in *IEEE International Workshop on Workload Characterization (WWC)*, 2004, pp. 3-10.

[17] R. Kotla, S. Ghiasi, T. Keller, and F. Rawson, "Scheduling processor voltage and frequency in server and cluster systems," in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005, p. 8 pp.

[18] S. Ghiasi, T. Keller, and F. Rawson, "Scheduling for heterogeneous processors in server systems," presented at the 2nd conference on Computing frontiers, Ischia, Italy, 2005.

[19] K. Malkowski, G. Link, P. Raghavan, and M. J. Irwin, "Load Miss Prediction - Exploiting Power Performance Trade-offs," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1-8.

[20] R. Schone, D. Hackenberg, and D. Molka, "Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors," presented at the 2012 USENIX conference on Power-Aware Computing and Systems, Hollywood, CA, 2012.

[21] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," presented at the 8th ACM international conference on Autonomic computing, Karlsruhe, Germany, 2011.

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

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

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

[25] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi, "Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster," presented at the 20th international conference on Parallel and distributed processing, Rhodes Island, Greece, 2006.

[26] V. W. Freeh and D. K. Lowenthal, "Using multiple energy gears in MPI programs on a power-scalable cluster," presented at the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, Chicago, IL, USA, 2005.

[27] C.-h. Hsu and W.-c. Feng, "A Power-Aware Run-Time System for High-Performance Computing," in *ACM/IEEE Supercomputing Conference (SC)*, 2005, pp. 1-1.

[28] R. E. Grant and A. Afsahi, "Improving system efficiency through scheduling and power management," presented at the IEEE International Conference on Cluster Computing, 2007.

[29] R. E. Grant and A. Afsahi, "Improving energy efficiency of asymmetric chip multithreaded multiprocessors through reduced OS noise scheduling," *Concurr. Comput. : Pract. Exper.,* vol. 21, pp. 2355-2376, 2009.

[30] T. Hruby, H. Bos, and A. S. Tanenbaum, "When Slower is Faster: On Heterogeneous Multicores for Reliable Systems," in *USENIX Annual Technical Conference (ATC)*, San Jose, CA, USA, 2013.