LUC: Limiting the Unintended Consequences of power scaling on parallel transaction-oriented workloads

Hung-Ching Chang, Bo Li, Godmar Back, Ali R Butt, Kirk W Cameron
Department of Computer Science, Virginia Tech
{hcchang, bxl4074, gback, butta, cameron}@vt.edu

***Abstract*—Following an exhaustive set of experiments, we identify slowdowns in I/O performance that occur when processor power and frequency are increased. Our initial analyses indicate slowdowns are more likely to occur and more acute when the number of parallel I/O threads increases and the variability between runs is high. We use a microbenchmark-driven methodology to simplify isolation of the root causes of I/O performance loss. We classify the observed performance loss into two categories: file synchronization and file write delays. We introduce LUC, a runtime system to Limit the Unintended Consequences of power scaling and dynamically improve I/O performance. We demonstrate the effectiveness of the LUC system running on two platforms for two critical parallel transaction-oriented workloads including a mail server (varMail) and online transaction processing (oltp).**

Keywords: parallel and distributed processing, I/O performance, power, runtime systems

I. INTRODUCTION

Systems continue to grow in complexity. Servers must be designed to handle hundreds and thousands of user requests for a finite number of resources. High-performance systems residing in data centers are designed to support the growing use of software services and mobile platforms. It is widely acknowledged that designing future systems that operate efficiently in increasingly complex environments is a grand challenge for the community [1].

To address inefficiencies, the components of these complex systems perform a growing set of tasks themselves. For example, processors [16], memory [5], and disks [18, 22] are capable of independently managing their power consumption. Such devices attempt to balance performance and energy use dynamically to match the changing resource demands of applications and services.

The combination of device independence with unprecedented degrees of parallelism in software and hardware at system scale leads to more complex interactions among operating systems, middleware, parallel and distributed applications and services, and hardware. Thus, identifying the optimal performance configuration at runtime is exceedingly difficult because the variance among data points may exceed the best average performance operating point.

Figure 1 provides an example of this phenomenon. The figure plots the performance for 64 threads of a parallel transaction workload at various power/frequency settings from 1.6 GHz to 3.1 GHz. Performance is the ratio of speedup versus the slowest frequency and the performance gets worse with higher frequencies. This slowdown[1] is difficult to observe in practice because the standard deviation (shown as the gray/blue area surrounding the line plot) often exceeds the performance difference between two data points.

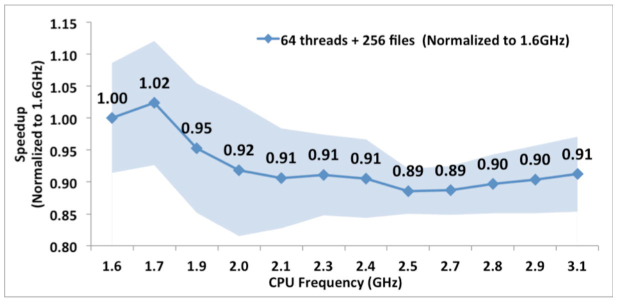


Figure 1 Filebench varMail speedup ratios normalized to the lowest available CPU frequency (1.6 GHz) – higher is better. The grey/blue area shows one standard deviation from the mean. *Nehalem (HDD)* System: Dell T3500 using a W3550 3.00 GHz quad-core with 6 GB of DDR3 RAM and a 250 GB 7200 rpm hard drive.

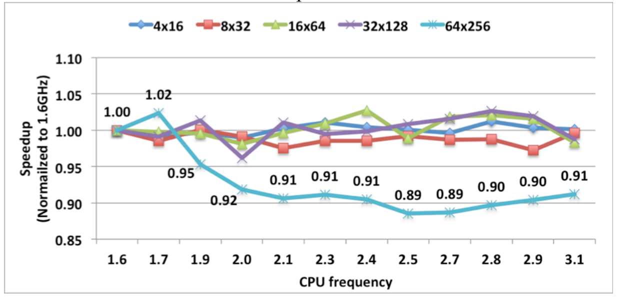


Figure 2 Filebench varMail speedup ratios normalized to the lowest available CPU frequency (1.6 GHz) – higher is better. Each line shows speedup for (number of threads) x (number of files). For 64 threads, performance drops significantly at higher frequencies.

For example, Figure 1 shows the 2.0 GHz frequency has a standard deviation of 12% while 1.7GHz on average performs 10% better. Thus, taking a single measured sample at the low performance range of 1.7 GHz and a single sample at the high performance range of 2.0 GHz results in the flawed conclusion that 2.0 GHz should be selected for better performance. Therefore, a runtime system attempting

[1] Characterization of the root causes of this slowdown is a key contribution. For now, we motivate the need to isolate causation and address these slowdowns later in the paper.

to adapt processor power/frequency settings [7] and optimize based on sampled runtime performance information could be ineffective and result in significant performance loss.

Additionally, the same code may perform differently at scale. Figure 2 shows the same example from our transaction workload for varying the number of threads from 4 to 64. The slowdown at higher frequencies does not occur until 64 threads where variance between runs increases. Beyond 64 up to 256 threads (not shown) the performance is similar to the 4→32 thread cases.

While dynamic concurrency throttling (DCT) [4] has received much attention, Figures 1 and 2 show the assumption that increasing power/frequency improves (or at least doesn't hurt) performance is flawed. Without a deeper understanding of the causes of these types of slowdowns and isolation of the root causes, runtime systems cannot avoid these types of slowdowns altogether. A runtime DCT system using sampling to schedule threads runs the additional risk of falling victim to the previously discussed variance issue.

Unfortunately, at the same time complexity threatens system efficiency, performance has never been more critical to the worldwide economy. According to Amazon's Greg Linden and Google's Marissa Mayer[2]: "Amazon found every 100ms of latency cost them 1% in sales. Google found an extra .5 seconds in search page generation time dropped traffic by 20%." High frequency trading also relies on data centers where power management is considered critical yet a few milliseconds of latency loss can result in millions of lost profits[3].

To address these *Unintended Consequences* and determine the best system configuration or operating frequencies, we need a deeper understanding of the root cause of these types of slowdowns on representative systems and benchmarks. In this paper, we significantly improve our understanding of the complex interactions between power scaling and parallel performance for the varMail and oltp transactional workloads from the Filebench suite [24]. Our contributions include: detailed identification of file system characteristics that contribute to slowdowns in parallel transactional workloads; classification of slowdowns into file write and synchronization delays; design and evaluation of a runtime system to address multiple causes of slowdowns due to file write and synchronization delays.

[2] http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it

[3] http://dealbook.nytimes.com/2014/07/07/no-need-to-demonize-high-frequency-trading/

II. Related Work

To illustrate the growing threat of complexity to efficiency, in this work our focus is on isolating the root causes of performance issues in parallel transactional workloads on a file system with power scalable components. Anecdotal evidence in the extant research literature suggests that latency loss occurs with increasing incidence [6, 9, 11, 12, 14, 19, 20, 23] and magnitude [8, 12, 17] when using common power management techniques such as dynamic voltage and frequency scaling (DVFS). Recent work [3, 17] shows that the use of power management is accompanied by acute increases in latency for certain file system benchmarks.

Our previous work was the first to show that these types of performance slowdowns could be traced to synchronization delays due to global journal commit operations in the Linux ext4 (or ext3) file system [3]. However, this work was limited in several ways. First, the focus on microbenchmarks led to an incomplete understanding of slowdowns. While the microbenchmarks correctly implicated journal commit operations, there was an implicit assumption that there were no other causes of slowdown on the performance critical path. However, when we attempted to extend our previous approach to more representative transactional workloads such as varMail and oltp, some slowdowns were unaffected. In this work, we explain why journal commits introduce file write delays (FWD) and file synchronization delays (FSD) and cause significant slowdowns in two different ways on parallel transactional workloads on power scalable systems. These new findings lead to the development of a runtime system that uses two different optimization approaches to address both write and synchronization delay contributions to slowdowns.

Work on dynamic concurrency throttling [4], dynamic power scaling [7], and their combination [10] are tangentially related to this work. These approaches assume that increased processor power and frequencies lead to better performance. Our proposed solutions can remove file write delays and reduce file synchronization delays that lead to slowdowns making power scalable systems better adhere to this assumption. Thus, our work and the proposed LUC system increases the likelihood that previous DCT and DVFS approaches will be effective.

III. File Synchronization and Write Delays

Figure 3 provides a brief explanation and summary of our previous findings. We refer the reader to the full paper for detail [3]. Herein, we give some background to facilitate a discussion of why our previous approach was not sufficient to address slowdowns in more representative transactional workloads of varMail and oltp. Though we provide details in our experimental results section, for all of our experiments, results, and conclusions, we performed repeated testing to achieve 95% statistical confidence on

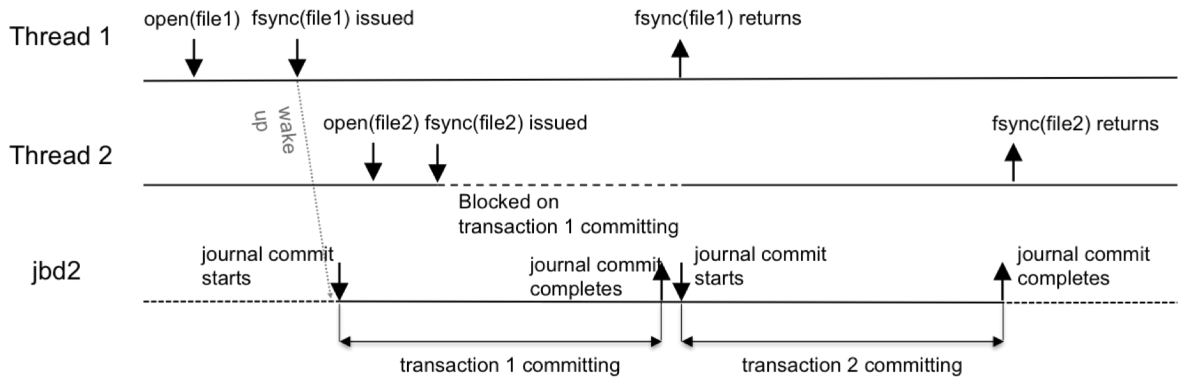


Figure 3 Synchronization delays occur when two threads in different transactions write to their respective files and at some later time flush the data and metadata to nonvolatile storage. In the Linux ext3 and ext4 file systems, the fsync operation causes a journal commit (JC) transaction that ultimately updates the meta-data on the nonvolatile storage. In this example, an fsync operation in Thread 1 triggers JC transaction 1 in the file system waking up the jbd2 system process. After JC transaction 1 starts, Thread 2 opens file2 and performs an fsync operation triggering JC transaction 2 in the file system that blocks and waits on transaction 1 to commit. This effectively serializes the atomic journal commit transactions in the file system.

several different systems running the ext3 and ext4 file systems default settings common to nearly all Linux distributions.

A. *File Synchronization Delays and Slowdowns*

In our previous work, we showed that when power scaling is enabled, the change in arrival rates of fsync requests for journal commits at the system level at higher frequencies can cause *file synchronization delays*. Figure 3 shows a detailed example where two separate, independent threads each trigger separate journal commit transactions in the Linux ext3 and ext4 file systems.

We observed that at slower processor frequencies, fsync requests would queue on the same journal commit transaction and be serviced in parallel since there were no true dependences among the files other than the system requirement that they update the metadata. We also showed that at higher processor frequencies, fsync requests were more likely to queue separately on the journal commit transaction and be serviced sequentially, because the open(file2) operation had occurred by the journal commit start triggered by the fsync(file1) call, then fsync(file2) would not have triggered a separate commit since file2's dirty metadata would have been considered part of transaction 1, as depicted in Figure 3. Therefore, an independent transaction (transaction 2 in Figure 3) is unable to start its journal commit due to the journal commit of the previous transaction. We measured performance losses up to 47% in the IOzone and Metarates microbenchmarks at higher processor frequencies which also wasted significant amounts of energy.

To improve performance and energy efficiency, we proposed two solutions. First, we removed the file system journaling for metadata. While this can hurt the reliability of the system and shouldn't be used generally, it provided a performance target for solutions that maintain reliability. Second, we proposed statically increasing the time the system allowed fsync requests to queue on the journal commit transaction. We then studied the impact of various static settings for the queuing of fsync requests and found settings that improved the performance of IOzone and Metarates when scaling processor frequencies and power.

B. *The Limits of File Synchronization Optimizations*

In our previous work, we focused on the IOzone and Metarates microbenchmarks. IOzone [13] is a file system microbenchmark that generates and measures a variety of file operations including read/write latencies. Metarates is a file system microbenchmark that measures the performance of concurrent aggregate metadata transaction rates in extremely large file systems [15].

These benchmarks were carefully selected for their relative code simplicity and the fact that they exhibited significant power scaling slowdowns similar to those we observed in full applications. This made the task of isolating the root cause of slowdowns practicable despite the measurement variability previously discussed and the number of parallel threads involved (up to 256). We traced the slowdowns to the fsync operations called directly in Metarates and indirectly (via a write operation) in IOzone.

As follow on work, and in response to reviewer comments, in this work, we attempted to determine the effectiveness of our proposed static optimization methods to address the synchronization delays identified in Metarates and IOzone on full benchmarks such as varMail and oltp. These highly parallel benchmarks represent common tasks in the datacenter workloads of service providers from Google to Amazon where slowdowns of the type we've observed encourage users to disable power management entirely despite the potential energy savings.

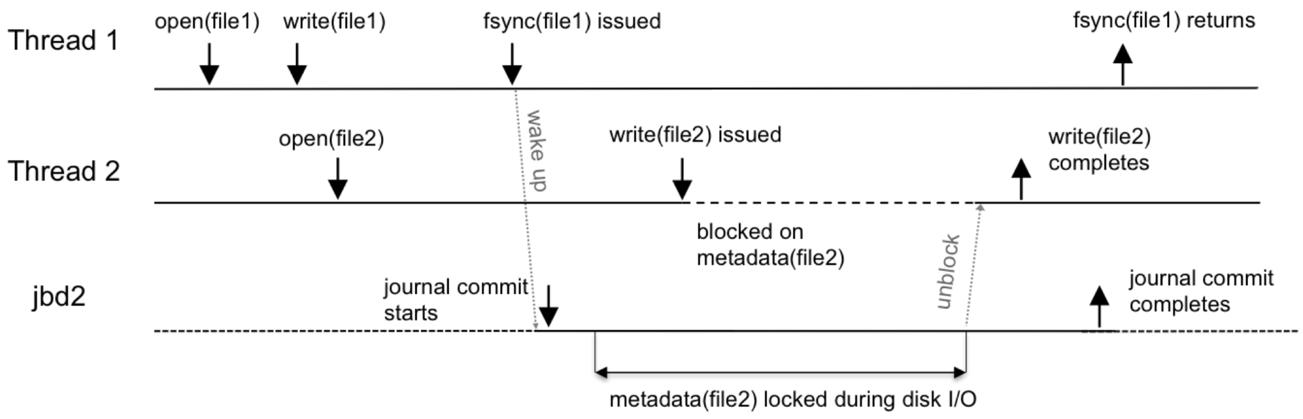


Figure 4. Much like synchronization delays, write delays occur when two threads write to their respective files and at some later time flush the data and metadata to nonvolatile storage. In the Linux ext3 and ext4 file systems, the fsync operation of Thread 1 causes a journal commit (JC) transaction that ultimately updates the metadata on the nonvolatile storage. In this example, an fsync operation in Thread 1 triggers JC transaction 1 in the file system waking up the jbd2 system process. After JC transaction 1 starts, thread 2 attempts to write to file 2. This write operation requires write-access to update file 2's metadata, which is however locked by the ongoing JC transaction 1. As a result, thread 2 is delayed until the transaction has written file2's metadata to disk and released the lock. As before, power scaling slowdowns affect the arrival time of file system requests and impact the likelihood of such delays. The oltp workload suffers from write delays during power scaling experiments, not synchronization delays. Thus, optimization techniques (such as those in previous work) that only address synchronization delays did not improve the performance of oltp.

Our earlier static techniques [3] were measurably effective on varMail slowdowns but had little to no effect on slowdowns in oltp. These mixed results and the complexity of tracing parallel thread performance in oltp caused us to revisit our analyses of the Metarates and IOzone microbenchmarks. Generally, we found that while our previous analyses (summarized in Figure 3) were correct, we missed a subtlety that results in a different optimization approach and requires a deeper discussion of the Linux ext3 and ext4 file systems.

C. *Write Delays and Slowdowns*

For both historical and reliability reasons, Linux ext3 and ext4 file systems default to journal ordering mode[4] which refers to the protocol for pushing metadata from memory to nonvolatile storage. During a journal commit transaction, dirty metadata is written to nonvolatile storage via the disk journal. Figure 3 illustrates how two fsync operations from two independent threads cause performance synchronization delays. In contrast, Figure 4 demonstrates how an fsync and write operation from each of two independent threads cause *file write delays (FWD)*.

To illustrate file write delays, consider the example in Figure 4. Thread 1 opens file1, writes to file1, and then calls an fsync operation. This results in a change to the file1's metadata and the jbd2 processes is awakened to lock the metadata. Further modifications to the metadata must block until the journal commit transaction finishes atomically writing the dirty metadata to nonvolatile storage.

[4] We have experimented with write-back mode for both data and metadata. Slowdowns were not eliminated. We will explore this further in future work.

When the open operation is called by Thread 1, a single, running transaction is launched and records a change to the metadata for file1. Next, the running transaction records a write to file1 and an open to file2 in the metadata. When the Thread 1 fsync operation occurs, the running transaction becomes a committing transaction and the journal commit thread starts writing the changed metadata to nonvolatile storage.

In this scenario, Thread 2 attempts to write to file2 after the committing transaction has started. All file writes now block (including the write by Thread 2 to file2) until the full metadata write to nonvolatile storage is complete. More precisely, Thread 2's write to file2 is handled by the kernel which places the request in a wait queue until the metadata write back to nonvolatile storage is complete. The kernel then wakes up Thread 2 so the write to file2 is allowed to proceed. This can result in a significant performance delay for Thread 2's write to file2. The file write delay is likely to be significant and will depend on the amount of metadata that must be written back to the nonvolatile storage.

In our previous work, we identified journal commit transactions triggered by fsync operations as the root cause of slowdowns. Our static approach to disabling or delaying the journal commit transaction, based on this analysis, improved performance in both IOzone and Metarates.

After revisiting the analyses of IOzone and Metarates herein we determined that Metarates performance was driven by file synchronization delays while IOzone was driven by file write delays. For Metarates, the static change to the timing of the journal commit transaction ensured that the fsync operations in Thread 1 and Thread 2 of Figure 3 queued up for a single journal commit transaction.

There was a subtly different effect in the IOzone microbenchmark. IOzone consists of a number of parallel threads following the scenario depicted in Threads 1 and 2 of Figure 3 – namely, file open followed by one or more writes followed by an fsync operation (which triggers the journal commit transaction). With a statically-determined, reasonably long delay of the journal commit transaction, all of the writes complete before the start of the first fsync operation and all of the fsync operations are serviced by a single journal commit transaction.

However, in IOzone there are few interleaving operations, all the threads perform similar amounts of work and operate independently of one another. Thus, upon revisiting the code and digging deeper on the kernel file system, we determined the slowdowns in IOzone are caused by the unintended interaction of file writes with ongoing journal commits, rather than by interactions between concurrent journal commit requests..

D. *Ineffective Copyout Optimization*

Write delays occur due to the journal commit that writes dirty meta-data to nonvolatile storage. In the worst case, any write or fsync can block (i.e. be kept from updating the meta-data) until all the dirty meta-data is finished writing back to nonvolatile storage. It is not inherently necessary for a committing transaction to prevent further updates to a file's metadata since only a snapshot of the metadata is written to disk. In fact, the kernel already provides a "copy-out" mechanism by which a snapshot of a file's metadata is created in a separate page. The journal commit writes this snapshot to disk without preventing further updates to the metadata. Unfortunately, in the current implementation, copyout is done only when write attempts are detected before the start of the journal commit's I/O operations, because the file system designers mistakenly assumed that concurrent write attempts during the journal commit would be an infrequent case.

For example, in Figure 4, after Thread 1 issues the fsync command, the kernel begins the journal commit transaction. To improve performance, the kernel will issue the copyout command at the start of the journal commit transaction to copy the current state of dirty meta-data to kernel memory. However, this only effectively copies the metadata that is about to be modified by write operations. Prior to the time when the I/O operation metadata(file2) locks in the jbd2 process (see Figure 4). As a result, the copyout mechanism does not make a copy of the meta-data for file2 arriving after the metadata(file2) lock, because the write(file2) is issued after the copyout command is issued and the metadata(file2) has submitted for I/O. Any operations (such as the write to file2 by Thread 2) will block until the nonvolatile write to storage completes. This causes unnecessary delays in write operations. We address this shortcoming in our runtime system discussed in the next section.

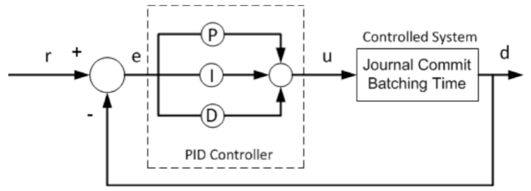


Figure 5 A PID controller for journal commit batching transactions in the ext3 or ext 4 file system.

IV. The LUC Runtime System

We propose the LUC (pronounced "luck") runtime system to improve performance by addressing the file synchronization delays and file write delays that cause slowdowns. LUC is designed to *Limit the Unintended Consequences* that result from the complex interactions of power scaling and parallelism. The proposed system contains two components: a PID batching component to limit synchronization delays and a heuristic copyout component to reduce write delays.

A. *A PID Controller for Journal Commit Batching*

The Linux kernel developers anticipated that frequent journal commits might be expensive and implemented a naive mechanism that would delay journal commits under certain circumstances. Yet, as discussed in Section III, this mechanism turned out to be largely ineffective, resulting in the slowdowns due to synchronization delay we observed. Our first attempt [3] to reduce synchronization delays required trial and error to determine how long to delay journal commit transactions for each application. This manual process was cumbersome and required constant retuning.

To address these deficiencies, we propose a dynamic controller that can automatically adapt to changes in system usage at runtime. We chose to implement this component as a proportional-integral-derivative (PID) controller [2] because we can use direct system measurements as inputs and we have had success with these types of controllers in runtime systems previously.

Ideally, a journal commit batch control system would maximize the frequency of journal commits to approach the number of flush operations while minimizing the number of synchronization events that cause delays.

Figure 5 shows the three major components of our controller. The set point is the target number of synchronization events per journal commit (r value). The set point can be updated dynamically at runtime. We instrument the kernel to observe the number of synchronization events per journal commit (d). The difference between the set point and the current observation is the error (e=r-d).

To affect the length of time the journal commit allows synchronization events to queue, we designed a control mechanism that delays the journal commit. Increasing the delay time can increase the number of synchronization events per journal commit while decreasing delay may have the opposite effect.

The correlation between the journal commit delay and the number of batched synchronization events exhibits a proportional relationship that maps well to a proportional function for an input signal u(t):

$$d(t+1) = d(t) + c \cdot u(t+1). \quad (1)$$

Here, t is the time step and d is the measured value of synchronization events per journal commit. d(t) is the number of synchronization events we can collect in a single journal commit at time t. d(t+1) is calculated by the previous value d(t) plus the change specified by our PID controller. We assume that such change follows a proportional relationship (c) to the output of the PID controller u(t) at time t. The value of d(t) is in practice constrained by the total number of synchronization events for a certain amount of time we set as threshold ($1 \leq d(t) \leq d_{max}$).

A PID controller is generically described by the following equation in the continuous time domain:

$$u(t) = K_P e(t) + K_I \int e(t)dt + K_D \frac{de(t)}{dt}. \quad (2)$$

u(t) is the output of the controller at time t based on the measured error within the system (e=r-d). From Equation (2), the output of the PID controller is the sum of three terms:

$K_P e(t)$: The first term is proportional to the error. This causes the system to respond to the error value and direction.

$K_I e(t)dt$: The second term is proportional to the integral of the error. This aggregates error over an interval to eliminate the steady-state error.

$K_D e(t)dt$: The third term is proportional to the rate of change in the error, reducing overshoot by directionally damping the response.

The terms K_P, K_I ,and K_D are control gains that are analytically determined through stability analysis. Poor selection of these parameters can cause system instability manifested as output signal oscillation—which in our case would cause thrashing between overshooting and undershooting the journal commit delay and potentially resulting in performance loss. Careful selection of control gains results in a controller with desirable convergent, performance properties. The first step in determining these parameters is to identify the controller transfer function.

Since we monitor the number of synchronization events per journal commit at runtime, our system is inherently discrete, thus we use the discrete form of transfer functions. The z-transform of the discretized, closed-loop, PID controller transfer function is

$$\frac{d(z)}{r(z)} = \frac{c(K_P + K_I + K_D)z^2 - c(K_P + 2K_D)z + cK_D}{(cK_P + cK_I + cK_D + 1)z^2 - (cK_P + 2cK_D + 2)z + cK_D + 1}. \quad (3)$$

This closed-loop transfer function is derived from the z-transform of the PID controller transfer function and our controlled system transfer function (Equations 1 and 2). In other words, the controller parameters are determined by considering the interaction of the controller function and the controlled system function. Relating the transfer functions of the feedback control system and the controller, we can select and analyze control gains for stability.

We experimentally validated stability of our PID controller by looking at poles of the closed-loop transfer function (Equation 3). The poles are defined as the roots of its denominator. Theoretically, the system is stable if all the poles locate strictly within the unit circle. We use the following stable parameter set in the LUC system: $K_P = 0.8, K_I = 0.25, K_D = 0.25$.

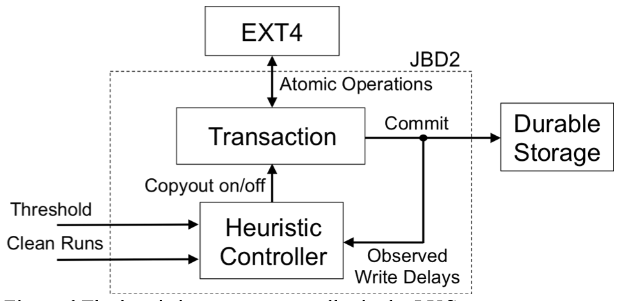


Figure 6 The heuristic copyout controller in the LUC system.

B. An Improved Copyout Controller

Our analysis indicates that the default copyout mechanism in the kernel is useful just not applied during the write delay slowdowns observed in oltp. We propose a heuristic feedback controller to dynamically enable the kernel copyout mechanism when it senses unusual write delay activity at runtime. This feedback controller forces copyout when the write delays affect performance and disables copyout when write delays are infrequent.

In the kernel default ordered mode journaling, all metadata modifications are treated as atomic operations and attached to the running transaction. After some time (e.g., 5 seconds by default) or upon user-driven synchronization events, metadata changes are written to nonvolatile storage.

Figure 6 shows the proposed heuristic copyout controller for the LUC runtime system. The controller is implemented in the kernel JBD2 process that handles journal commit transactions. The controller adapts to three inputs: write delays per journal commit (threshold); observed write delays per journal commit; and the number of runs without a write delay (Clean Runs). Threshold is user-defined and can be changed dynamically at runtime.

The heuristic controller attempts to eliminate the performance penalty due to excessive write delays, when the percentage of write delays to total writes exceeds a given threshold. This is accomplished by enabling the copyout mechanism before the metadata I/O begins. Enabling copyout in this manner removes the lock to the metadata in the file system and eliminates write delays during the journal commit.

The heuristic copyout controller uses the difference between the threshold and the number of observed write delays to determine when to enable and disable the copyout mechanism. When the number of write delays exceeds the write delay threshold, the controller signals the kernel jbd2 thread to enable copyout. For example, the controller works well for the oltp workload with the configuration of enabling copyout as soon as a threshold of 5% of total writes were delayed during the previous journal commit transaction.

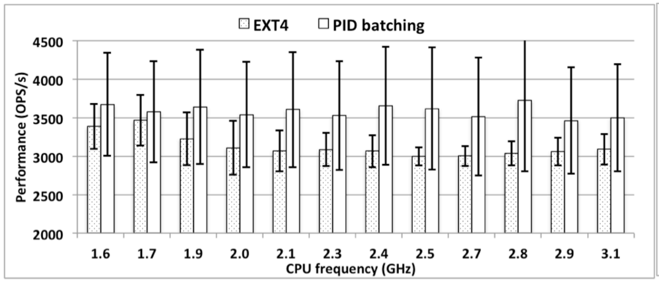
A lower percentage of total write delays disables the copyout mechanism. For example, a Clean Run value of 4 journal commits where no write delays occurred for 4 consecutive journal commits worked well as a default value after extensive experiments, and takes up to 20 seconds to disable the copyout when no jbd2 thread is triggered by a user.

V. Evaluations

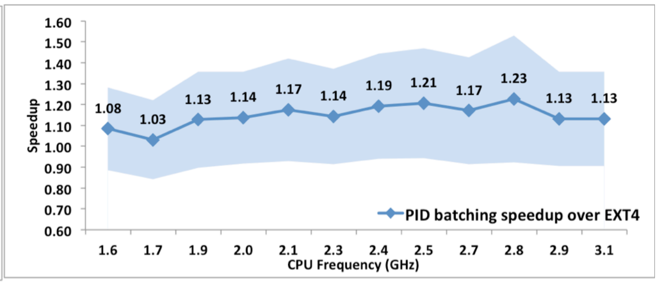
A. Experimental Setup

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

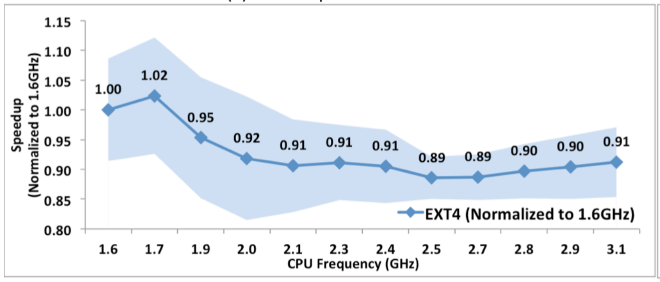
Both systems run CentOS Linux distribution (6.4) with the kernel version (3.4.2). We configure the systems with the ext4 file system and ordered journaling mode. We disable the periodic background data writeback and journal



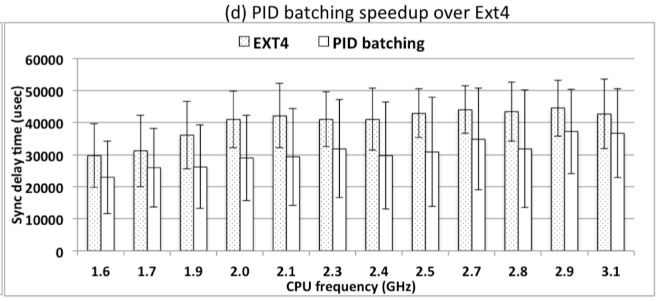
(a) varMail performance



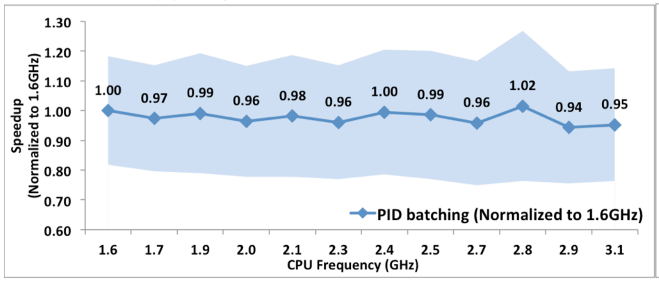
(d) PID batching speedup over Ext4



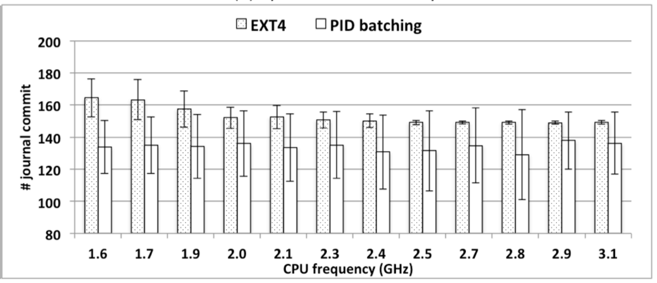
(b) Ext4 performance normalized to 1.6GHz



(e) Synchronization delay time



(c) PID batching performance normalized to 1.6GHz



(f) Number of journal commits

Figure 7. Select results for varMail on Nehalem (HDD). Findings are comparable to other systems though not included due to space limitations.

commit to reduce system noise. 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 [21]. Due to space limitations, we focus on results that reflect the key contributions of this work. [5]

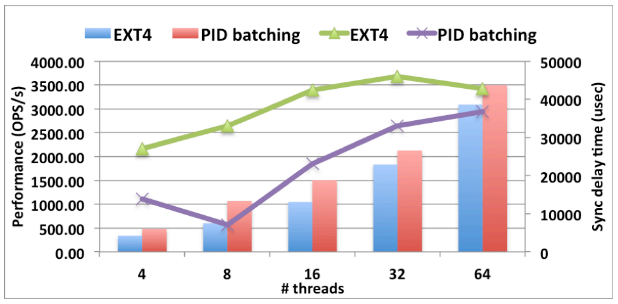


Figure 8 Filebench varMail synchronization delays (lines) and performance (bars) for increasing number of threads at **3.1** GHz.

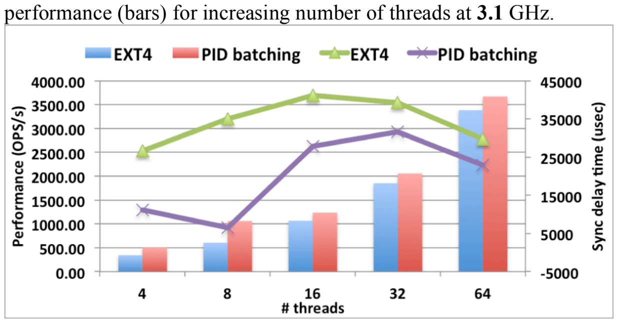


Figure 9 Filebench varMail synchronization delays (lines) and performance (bars) for increasing number of threads at **1.6** GHz.

B. *VarMail Results*

Filebench [24] varMail emulates a multi-threaded mail server workload common to file systems. Figure 7 compares varMail results for the default ext4 configuration to our PID journal commit batch controller. VarMail is configured with 64 threads and 256 files with settings of 1K file size, 1K IO size, and 1K append size. The data points shown here are averaged from >50 runs for a combination of configurations (i.e. ext4, PID batching) and frequencies (1.6GHz, up to 3.1GHz).

Figure 7 illustrates typical findings for the varMail benchmark where slowdowns occur at higher processor frequencies. Figure 7a compares the raw performance (Ops/s) at each available processor frequency for the default ext4 configuration and the proposed PID batching described in the previous section. PID batching consistently outperforms the default method.

Figures 7b and 7c show the performance of the ext4 and PID batching configurations for the available processor frequencies normalized to the slowest frequency. These two graphs show that for the varMail workload PID batching removes nearly all slowdowns. Figure 7d shows the amount of speedup achieved by PID batching (from 3% up to 23%) at each frequency.

Figure 7e shows these speedup numbers as the change in measured synchronization delays for ext4 and PID batching. These values are the average time that a user-driven synchronization event delays before it is serviced by the journal commit (in varMail this is the time spent in the fsync system call). Notably, PID batching on average (in every case for this scenario) spends less time in synchronization delays than the ext4 default case. Figure 7f shows that PID batching results in less journal commit transactions than the ext4 case as well. In this case, the trend is more meaningful than the raw numbers since the journal commit transaction time is variable (i.e. synchronization delay time tracks more closely with speedup).

The larger the difference in synchronization delays between EXT4 and PID batching configurations, the better the speedup. For example, the PID batching configuration shows significant speedup of 1.23x over EXT4 configuration at 2.8 GHz for an average 11.6ms reduction in synchronization delay, and shows only 1.03x speedup over EXT4 configuration at 1.8 GHz for an average 5.3ms reduction in synchronization delay.

Figures 8 and 9 show the performance differences between ext4 and PID batching. Each pair of bars denotes the performance (Ops/s) for 4 to 64 threads on the

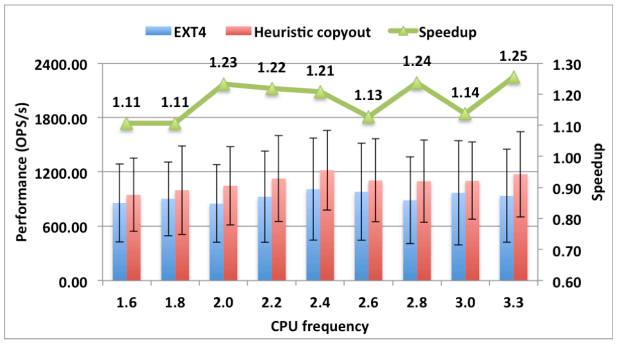


Figure 10 Filebench oltp speedup (line) and performance (bars) for EXT4(default copyout enabled) and heuristic copyout running at all available frequencies on *SandyBridge(HDD)*. The heuristic copyout improves performance for all processor frequencies.

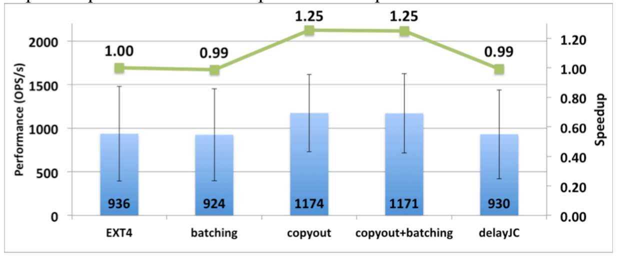


Figure 11 Filebench oltp performance (bars) and speedup normalized to EXT4 (line) on the *SandyBridge(HDD)* running at 3.3 GHz for various configurations. The combined heuristic copyout + PID batching controller provides comparable performance.

[5] The exhaustive testing necessary for 95% confidence takes weeks. We have tested on ext3 and ext4 and various versions and distributions of Linux. In every case, slowdowns were observed and the details of the file systems herein have been confirmed.

Nehalem(HDD) system running at 3.1 GHz. The performance in both cases increases with the number of threads while the PID batching scenario consistently out performs ext4. The synchronization delays increase steadily with the number of threads since the likelihood for lock collisions across threads increases. The sync delay time for PID batching is consistently lower than that of the ext4 default. Currently, we are unable to fully explain the dips in sync delay time at 8 threads (at both 3.1 and 1.6 GHz) and at 64 threads (at 1.6GHz). However, to simplify our experiments we used a PID set point equal to half of the number of threads. Thus, this is likely an artifact of this setting and with further exhaustive testing (which takes weeks) we could likely find set points that further optimize the sync delay time for different combinations of threads.

C. Oltp Results

Filebench oltp emulates a TPC-C workload testing the performance of small random read and write transactions on a file system using the I/O model from Oracle 9i. The oltp benchmark is run with 48 threads and 48 files. The data points are averages over 140 runs.

Figure 10 shows the performance (Ops/s) of the ext4 and heuristic copyout controller when varying the processor frequency from 1.6 to 3.3 GHz on the *SandyBridge(HDD)* system. In the default ext4 case, slowdowns with processor frequency increases are minimal. The same is true for the heuristic copyout results. However, the heuristic copyout controller consistently outperforms the default ext4 case. The classification of write delays as separate from synchronization delays led to the copyout optimizations that contributed these significant increases in raw performance. The line curve in Figure 10 shows the performance improvements vary from 11% up to 25% at 3.3 GHz.

Figure 11 shows oltp performance (Ops/s) in bar graph form for different optimization strategies on the *SandyBridge(HDD)* system. The strategies include: ext4 (default), batching (proposed PID controller), copyout (proposed heuristic copyout), copyout+batching (the LUC system with a combined PID and heuristic copyout controlled), delayJC (our previous static batching technique optimized for oltp). The line graph shows the speedup relative to the ext4 case for each configuration.

Initially, we observe that for the oltp workload, ext4, batching, and delayJC have statistically the same performance. The similarity between ext4 and delayJC confirms our earlier finding that batching has little to no effect on oltp. For batching and delayJC, it is no surprise that their performance is comparable on oltp given the workloads insensitivity to synchronization delays.

Furthermore, we notice that copyout performance confirms our findings in Figure 10 that 25% performance improvement is achieved. Upon deeper inspection, our heuristic copyout controller (with tracing enabled) revealed 32% of the total write system calls in oltp suffered write delays with an average delay of 22.2 ms (not shown in Figure 11). In contrast, when we ran the copyout controller with varMail, we measured 0.2% of the total write system calls in varMail suffered write delays with an average delay of 0.9 ms.

In the final LUC runtime system, we combined the copyout and PID batching controllers to provide a single solution and check whether there were any unexpected deleterious effects when they are combined. We noticed none as shown in Figure 11 since the copyout+batching scenario results in 25% speedup (the same as copyout in isolation) compared to the ext4 default case.

VI. Conclusions and Future Work

In this paper, we improved our understanding of performance slowdowns that occur at higher frequencies in power scalable systems. Our novel classification of performance loss into two causal categories (file synchronization delays and file write delays) enabled us to propose the LUC runtime system that leverages a novel PID controller for batching journal commits and an effective heuristic copyout controller for enabling copyout when it will improve performance. Our techniques result in up to 25% performance improvements on the benchmark and systems studied. Since we are operating in lower power modes longer, there are power and energy implications we plan to study further in future work.

There are some limitations to our work. The heuristic copyout controller currently requires us to statically set values for threshold and the number of clean runs. While we found some experimentally acceptable values, we would like to automate this and have the controller adapt these inputs dynamically.

For the PID batch controller, the effectiveness is closely related to the selected set point and while the controller adapts dynamically, the set point is currently static. This could limit the controller's effectiveness in environments where the workloads vary significantly. We hope to address this in future work.

Lastly, while both proposed controllers substantially improved performance and reduced energy waste over the ext4 default case, neither eliminated slowdowns entirely. Even after applying both controllers in the LUC runtime system, we observed remaining slowdowns as high as 6% for varMail and 9% for oltp in some scenarios. We leave identifying these additional types of inefficiencies to future work.

VII. Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1422788, 0910784 and 0905187.

VIII. References

[1] *NITRD LSN Workshop Report on Complex Engineered Networks, 2012*. Available: http://www.nitrd.gov/Publications/PublicationDetail.aspx?pubid=52

[2] K. J. Åström and B. Wittenmark, *Adaptive control*: Courier Dover Publications, 2008.

[3] H.-C. Chang, B. Li, M. Grove, and K. W. Cameron, "How processor speedups can slow down I/O performance," in *Proc. of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'14)*, Paris, France, 2014.

[4] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online strategies for high-performance power-aware thread execution on emerging multiprocessors," in *Proc. of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodes Island, Greece, 2006.

[5] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proc. of the 8th ACM international conference on Autonomic computing (ICAC'11)*, Karlsruhe, Germany, 2011, pp. 31-40.

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

[7] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters," in *Proc. of the ACM/IEEE conference on Supercomputing (SC'05)*, Seatle, WA, 2005, p. 11.

[8] R. Ge, "Evaluating Parallel I/O Energy Efficiency," in *Proc. of the IEEE/ACM Int'l Conference on Green Computing and Communications (GreenCom'10) & Int'l Conference on Cyber, Physical and Social Computing (CPSCom'10)*, 2010, pp. 213-220.

[9] N. B. Lakshminarayana and H. Kim, "Understanding performance, power and energy behavior in asymmetric multiprocessors," in *Proc. of the IEEE International Conference on Computer Design (ICCD'08)*, 2008, pp. 471-477.

[10] D. Li, B. R. De Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid MPI/OpenMP power-aware computing," in *Proc. of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*, 2010, pp. 1-12.

[11] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs," in *Proc. of the ACM/IEEE conference on Supercomputing (SC'12)*, Tampa, Florida, 2006, p. 14.

[12] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: understanding the runtime effects of frequency scaling," in *Proc. of ACM/IEEE conference on Supercomputing (SC'02)*, New York, New York, USA, 2002, pp. 35-44.

[13] W. D. Norcott and D. Capps. *Iozone filesystem benchmark, 2006*. Available: http://www. iozone. org

[14] F. Pan, V. W. Freeh, and D. M. Smith, "Exploring the energy-time tradeoff in high-performance computing," in *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[15] C. Philip, L. Samuel, R. Robert, V. Murali, K. Julian, and L. Thomas, "Small-file access in parallel file systems," in *Proc. of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, Rome, 2009, pp. 1-11.

[16] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, "Ensemble-level power management for dense blade servers," in *Proc. of the 33rd annual international symposium on Computer Architecture (ISCA'06)*, 2006, pp. 66-77.

[17] T. Saito, K. Sato, H. Sato, and S. Matsuoka, "Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system," in *Proc. of the 3rd Workshop on Fault-tolerance for HPC at extreme scale (FTXS'13)*, New York, New York, USA, 2013, pp. 41-48.

[18] P. Sehgal, V. Tarasov, and E. Zadok, "Evaluating Performance and Energy in File System Server Workloads," in *Proc. of the USENIX Conference on File and Storage Technologies (FAST'10)*, 2010, pp. 253-266.

[19] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer, "Efficient interaction between OS and architecture in heterogeneous platforms," *ACM SIGOPS Operating Systems Review,* vol. 45, pp. 62-72, 2011.

[20] E. L. Sueur and G. Heiser, "Dynamic voltage and frequency scaling: the laws of diminishing returns," in *Proc. of the international conference on Power aware computing and systems (HotPower'10)*, Vancouver, BC, Canada, 2010, pp. 1-8.

[21] 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.

[22] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "SRCMap: Energy Proportional Storage Using Dynamic Consolidation," in *Proc. of the USENIX Conference on File and Storage Technologies (FAST'10)*, 2010, pp. 267-280.

[23] L. Wang, G. v. Laszewski, J. Dayal, and F. Wang, "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS," in *Proc. of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, Melbourne, VIC, 2010, pp. 368-377.

[24] A. Wilson, "The new and improved FileBench," in *Proc. of the USENIX Conference on File and Storage Technologies (FAST'08)*, 2008.