Study on the File System Calls of Desktop Applications

Nodir Kodirov, RJ Sumi
University of British Columbia

***Abstract*—The ways in which modern applications interact with file systems are complex. Previous research shows that modern software uses a myriad of calls to the file system (FS) for operations that are simple in the abstract, such as saving a text document. This is because files are more than just files; many have complex internal structure, and often files are used as a medium for complex, higher-level application behaviour. What is expected of a modern application has changed a great deal since the FS API has changed last. Thus, we have investigated the text editing program gedit and the libraries it relies upon, gtk and glibc. Using tracing tools to guide source inspection, we build comprehension of the higher-level behaviours expressed through multiple system calls that are currently present in gedit, and based on these analyses identify recurring patterns that can be simplified if better FS support were available. We discuss potential modifications to the FS API that would better facilitate the behaviour desired by programmers of today's applications.**

I. Introduction

A. Motivation

Efficient use of available resources is critical to the satisfactory completion of any computing workload. The file system (FS) is no exception to this rule; indeed, FS design, implementation and performance have been of concern to academics and industry for a long time [2]. However, the vast literature concerning itself with the measurement and improvement of FSes is typically focused on industrial or academic contexts, ignoring the usage of commodity file systems by software targeting end users. Recently, this area has received some attention, and it has been observed that modern software makes many FS calls for conceptually simple tasks [2], performing multiple reads and writes for simple workloads such as text editing of a document.

For example, consider the case of writing a simple note to yourself in your favourite text editing program. This note might be only a few words, and will take you all of ten seconds to write and save. One might assume that such a basic task would correspond to a simple pattern of usage of your system's resources, but this is not the case. Table I shows the number of the four most frequently used FS API calls made during such a workload with a basic GUI text editor.

It is clearly visible that the usage of the FS API goes far beyond what the simplicity of the workload in question would suggest. This is because today's files are more than just files. Many of the files manipulated by modern end-user software have complex internal structure, and, conversely, many files do not represent complete entities by themselves, but are part of implementing functionality spanning the use of multiple files.

What a modern end-user application is and does have evolved a great deal since the FS API has changed last. A consequence of this is that the complex file behaviours required of modern applications are entirely the responsibility of said applications and the libraries upon which they rely; they receive little support from the simplistic FS API. Prior work has not investigated what portion of the burden of complexity is shouldered by the program logic itself, and what part is shouldered by libraries. However, the haphazard usage of the existing FS API suggests functionality desired by application developers may not readily provided by the FS. whether or not this is the case has not been examined, which is a principal concern of this work. We wish to assess *syscall bloat* in modern end-user applications, where syscall bloat refers to the the generation of many syscall invocations at some point in the application or library code. More specifically, we wish to identify whether bloat is necessary to implement the behaviour applications or libraries wish to express, and if so, whether its necessity is due to limitations of the design of the file system.

By conducting experiments to identify the sources of complicated FS API usage, we aim to address the question as to what modern end-user applications want to use files for, and thus shed light on how the file system might be able provide applications and libraries with a better interface to suit these goals. We believe this to be an important endeavour. Not only does the expressiveness of the FS API directly influence the clarity of the code that is written on top of it, but also its ability to implement its intended behaviour in a performant way. Given the amount of attention the optimization of file systems has received historically, this is of some import.

B. Our Contributions

This work presents a number of contributions to the study of the FS in the context of end-user applications. We enumerate what our work offers as follows:

1) *Logs* - We provide a body of logs consisting of traces taken from a typical modern end-user application's interaction with the FS API.
2) *Analysis Scripts* - We publish the Perl scripts we use to aid our analysis. These scripts can generate

syscall	Count
`read`	501
`fstatat64`	435
`close`	348
`write`	143

TABLE I: Number of calls

aggregate statistics about the logs, such as file access counts, as well as facilitate more complicated tasks, such as assembly and dissection of the call graph expressed in a log.

3) *Validation of Prior Results* - Prior work [2] has examined the FS API usage of applications in other contexts. We observe similar results to this prior work in a different environment, thereby generalizing their results further.
4) *Analysis of FS Usage* - We analyze the usage of the FS API as observed in the logs we produce, and determine some points at which syscall bloat occurs. We propose modifications to the FS API that would allow for this bloat to be avoided.

Our logs and scripts are available at the project repository, located on GitHub[1]. In this work, we chose gedit[2] as our representative of a modern end-user application, and examine both its behaviour and its usage of the libraries upon which it is built–namely gtk[3], and glib[4]. Our logs are produced from a fully open-source environment, meaning that unlike prior work, we have produced an artifact that aids the comprehension of syscall bloat that is usable by any motivated party.

We produce logs by gathering traces of typical application usage using DTrace[5] to determine how gedit uses the FS API. We parse these traces and use source code comprehension to discover that gedit and the libraries it uses cannot provide the functionality they want to without excessive usage of the FS, and that simple tasks that tangentially involve actions such as directory traversal can instigate large numbers of calls to the FS API. We find some of these behaviours to be unnecessarily expensive; the limited expressiveness of the FS API is at fault. In these cases, we identify possible extensions to the FS API that would enable applications and libraries to implement their existing functionality with many fewer calls to the FS API.

While our work is motivated by and concerns itself exclusively with the FS API, our methods of analysis and the tools we publish that facilitate it are not domain-specific; we use no special knowledge related to the FS to generate the information from which we conduct our analysis. We believe that our methods and tools can easily be applied to other domains, such as networking, though we do not investigate this possibility ourselves.

II. Methodology

In this work, we use gedit as a representative example of a modern end-user application. While prior work [2] makes observations across multiple applications, in our work to do so is not necessary. This is because one of our principal aims is to assign responsibility for syscall bloat to the portions of code that comprise a running application, including both application code and libraries. Since libraries remain the same across the many applications that use them, a single application is sufficient to gather the data we require to perform our analysis. Additionally, in Section III we observe that gedit expresses similar aggregate FS usage to the programs examined in [2], further justifying this decision.

To comprehend the file access behaviour of gedit, we employ a three-stage process. First, we use DTrace while performing a few simple file manipulation tasks, such as opening, editing, saving, and closing documents. This generates a log consisting of a trace for each FS API access from the syscall layer up to the caller's top-level function. We then process these traces using Perl scripts to produce aggregate statistics about the file access patterns, as well as graphs that show subsets of the call graph encoded in the logs. Finally, we use the edge weights on the graphs to identify probable sites of call bloat and use the function names to guide source code inspection.

Dynamic analysis is necessary for our work, because our focus is to examine the actual usage of the FS API by applications. Singly examining source code without logs would not only be extremely difficult, it would also not allow us to distinguish the portions of the code that do generate a bloat in FS usage from those that could. Our use of DTrace allows us to gather logs while avoiding any modification to the code we are observing.

A. Data Collection

As described, the first step in our comprehension process is to collect logs of gedit's file usage. To do this, we use the DTrace tool available on Solaris. This lets us interpose on all the syscalls made by the application we attach DTrace to. Additionally, we can use DTrace's declarative D language to specify for which types of syscalls we wish to record traces, and for each trace what additional information we wish to record as well.

In our data collection, we (for obvious reasons) traced only syscalls triggered by gedit, and only those targeting the FS API. Additionally, traces originating from process pipes, network calls and DTrace's own probes were all filtered out. Thus, our log contains traces pertaining only to gedit's interaction with the file system. These traces inform us as to the sequences of system calls being made by the program.

Since prior work [2] suggests that diverse workloads may be necessary to capture a representative slice of application behaviour, we take traces while executing three different tasks:

1) OneLineText: Open a new file, type "Hello World.", save the file as "hello_world.txt", close the file.
2) OnePageAutosave: Open a new file, save it as "one_page.txt", write 400 words of text, save the file again, close the file.
3) OnePageManual: As `OnePageAutosave` workload, except `CTRL+S` is pressed after every 4-5 sentences to simulate usual typing behaviour (about 14 times during the whole workload).

Both (2) and (3) take around 15 minutes to complete. The reason to separate `OnePageAutosave` from `OnePageManual` is that in the first we expect to exercise gedit's autosave feature, which is triggered every 30 seconds by default, while in the latter we aim to simulate the frequent saving behaviour of some users.

[1] `http://www.github.com/knodir/proj_538b`
[2] https://wiki.gnome.org/Apps/Gedit
[3] http://www.gtk.org/
[4] https://wiki.gnome.org/Projects/GLib
[5] http://dtrace.org/

B. Identifying Access Patterns

Our observation of gedit's file access patterns is driven by log processing. We do two major types of log processing in this work: generating aggregate statistics, and building subsets of the call graph. All of our processing is done in Perl using scripts available through the project's GitHub repository.

Generating aggregate statistics is the simpler of our processing methods by far. We generate histograms of the files that are accessed in FS-related syscalls, of the syscalls that are being invoked, and the most frequent pairs of top-level function and syscall. These inform further investigation by suggesting to us what syscalls we should be targeting in our graph building, and what types of routines we should be looking for bloat in (e.g. ones that interact with GUI elements if there are many interactions with files ending with .png).

Since the full call graph generated from even the OneLineText workload is large enough to prohibit easy comprehension, we employ filtering techniques to target subsets of the call graph for investigation. The first technique we use is restricting the call graph subset to only contain functions that appear in traces that access a particular function or set of functions in the FS API, for example fstat calls. Second, since gedit is a multi-threaded GUI application, our processing scripts allow us to restrict the call graph subset to only contain functions that appear in traces that originate from a specified top-level function, for example the main function of gedit. This allows us to differentiate between behaviour caused by the window manager and that caused by gedit. Since the previous two filtration techniques still produce graphs that are too complex to easily understand, we employ a threshold-based graph reduction scheme, wherein traces that contain functions that are called less than a specified number of times are culled from the log for the purposes of generating the graph. Since removing a trace from the log alters the number of occurrences of all the functions in the trace, and not just the one that caused the trace to be removed, this culling is done iteratively until convergence. We have found that with our data, a threshold of five occurrences reduces the size and complexity of the graph appreciably, while still maintaining rich information.

In addition to controlling the size of the call graph subset under examination and the information it pertains to, we attempt to model the execution that generated the log through edge weights. Doing this exactly is not possible, since we do not modify the application source, and DTrace gives us only stack traces and file names at the time of a FS API call; it is not possible to exactly determine at what depth into the stack two consecutive identical traces have originated from. However, the analysis we do provides a lower bound on where syscall bloat may be originating, and we find that is practically a useful tool in informing our investigation of the source code.

Our algorithm consists of maintaining a "stack state" for each top-level function while stepping through the log. If a trace contains a difference from the current stack state, all the edge weights of between the function calls in the part of the trace below the point of difference are incremented. If there is no difference, the difference is assumed to take place at the lowest level call. Making this stack state per-top-level function means that the arbitrary interleaving of execution in the multi-threaded gedit application does not affect our data processing and thus does not affect our analysis. Additionally, since the call graph may contain "diamond" patterns, which may obscure which path through the graph generates the syscall-rich executions, we also generate a second edge weight for each edge, which is simply the number of times that function-function pair appears in the log (after filtration). This lets us identify if only one of the parent functions of a bloated function is participating in bloated executions.

C. Identifying Simplifications

Our analysis of application and library source code informed by our call graph subsets allows us to comprehend the intended behaviour of code portions that lead to FS syscall bloat. From the comprehension we derive from this analysis, we generate simple descriptions of these higher level behaviours. For example, if we determine that gtk is making repeated calls to the FS API to observe all the members of a given directory, this would suggest that the library is performing a traversal of that directory. Our identification of such descriptions of behaviours enables us to do our next task, a proposal of a cleaner API. This phase of investigation generates concise descriptions of application behaviour that is not currently supported by the FS API. For example, "gedit's use of the gtk library causes it to make repeated stat calls to the null device."

The final stage of our work is to propose a cleaner FS API that allows for applications and libraries to express the same interactions that they already have with the FS more concisely. Our identification of the problematic behaviours of applications and libraries us to posit extensions to the FS API that would allow applications and libraries to more simply express their desired behaviour and with lesser syscall usage. Again taking the example of directory traversal, we could, in this case, propose that the FS API expose calls that would support different incarnations of this. This might take the form of proposing a call `readdir(pathname)`, which would cause the underlying FS to generate a copy of the directory structure in userspace for the program to query without crossings.

III. Observations

We started our experiment by looking for similar behavior to that found in [2]. That work was done on Mac OS X, with the HFS+ file system and Cocoa framework using the iWork suite. Because we wanted to comprehend the application and library source code to find the root causes of bloated syscalls, we conducted our experiment on Solaris 11 with open source libraries and gedit. Despite such differences in the experimental framework, the observations we made that were of the same type as those in [2] yielded similar results. We found that gedit makes an enormous number of syscalls to accomplish simple tasks. Figure 1 shows the amount of FS API syscalls made by `gedit` for the three workloads described in Section II. We observed hundreds of `write()`, `read()`, `fstat()`, and `close()` calls being made. This conforms with our previous study on Mac OS X environment and the results of [2]. The power-law-esque syscall histogram motivates us to focus our observations on the four most frequent calls, `write()`, `read()`, `fstat()`, and `close()`,

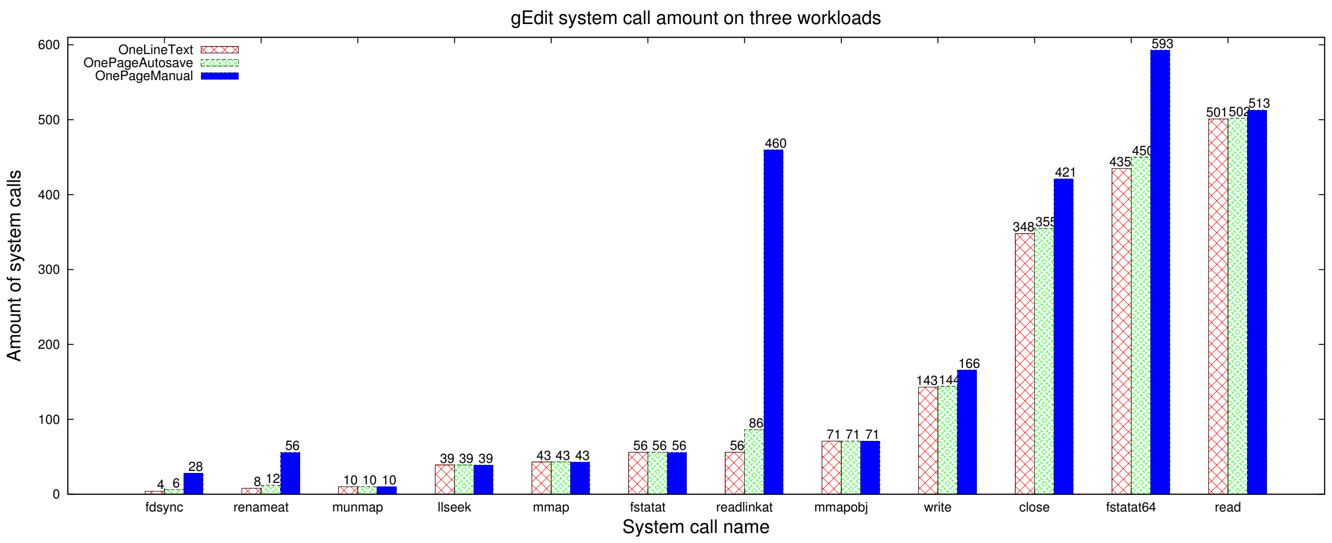

Fig. 1: Amount of system calls for three different workloads.

as identifying inefficient usage of those calls is likely to yield the most impactful simplifications due to their number.

Generating the full call graph implicity contained in the log proved to be a fruitless endeavour. It is too complex to aid comprehension of the application and library behaviour. Even after applying our reduction techniques discussed in Section II, gives us a large graph. Figure 2 shows the call graph for `fstat()`-realted traces rooted in the gedit main function, containing only traces where every function appears at least five times.

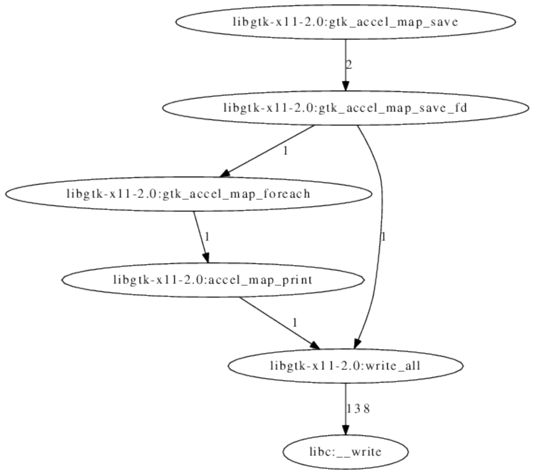
libgtk-x11-2.0:gtk_accel_map_save
2
libgtk-x11-2.0:gtk_accel_map_save_fd
1
libgtk-x11-2.0:gtk_accel_map_foreach
1
libgtk-x11-2.0:accel_map_print
1
1
libgtk-x11-2.0:write_all
138
libc:__write

Fig. 3: Selected area of call graph for `write()`-related traces.

Figure 3 shows a small portion of the call graph for `write()`-related traces. Sections like these are of particular interest to us, because they suggest substantial syscall bloat. Observe that the incoming edges' counts to `libgtk-x11-2.0:write_all` amount to only 2, while the outgoing edge count is 138. We investigated the `write_all` function and found that it calls `write()` in a loop that does not terminate until the entirety of the specified buffer is written successfully or failure is encountered. Since the libc `write()` call is allowed to return before having written all requested bytes in the presence of signals, this leads to a large bloat. We discuss this situation further in Section IV.

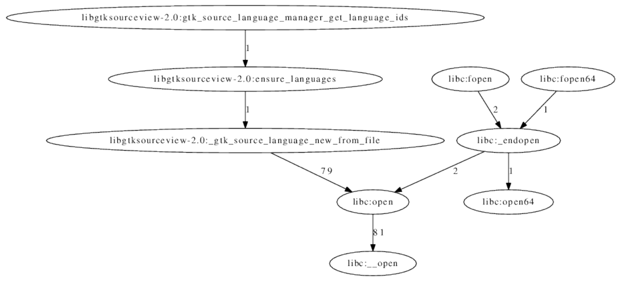
libgtksourceview-2.0:gtk_source_language_manager_get_language_ids
1
libgtksourceview-2.0:ensure_languages
1
libgtksourceview-2.0:_gtk_source_language_new_from_file
79
libc:fopen
libc:fopen64
2
1
libc:_endopen
2
1
libc:open
libc:open64
81
libc:__open

Fig. 4: Selected area of call graph for `open()`-related traces.

Figure 4 shows a small portion of the call graph for `open()`-related traces. This has a point of interest similar to the graph pertaining to writes, in that it shows an incoming-to-outdoing edge count ratio of 1 to 79 for `_gtk_source_language_new_from_file()`. While this seems to suggest it is this function that is the cause of the syscall bloat observed here, it is actually showing a limitation of our method. Our logs do not contain enough information to create the unique execution that produced the log, and so we show the lowest-possible point of bloat as creating the bloat. However, source code inspection reveals that this function cannot be creating syscall bloat. This means that we must traverse the call graph upwards until we find the cause of the bloat. This turns out to be quite easy in this case, as the culprit here is `ensure_languages()`. The reason `ensure_languages()` creates syscall bloat is because it iterates over the files in a directory and appears to do some very cursory processing. This appears to result in many `fstat()` and `open()` calls and small `read()` calls.

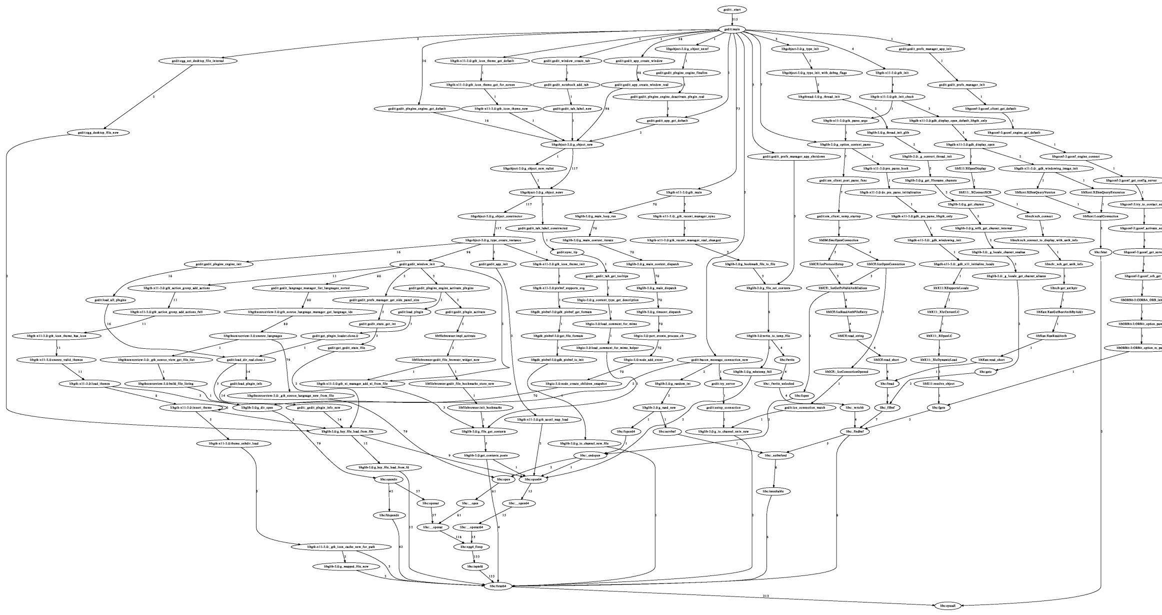

Fig. 2: Call graph for `fstat()`-related traces.

IV. Discussion and Future Work

A. Potential improvements

The FS API and its implementations on the various systems it is used in are products of extensive thought and engineering. For very good reasons, the FS API has been kept fairly simplistic. However, our observations show that, if we were to specialize the FS API to better support the requirements of end-user applications. We have identified a few examples that could be included into the FS API that would increase its expressiveness and potentially reduce the number of syscalls programmers and libraries would need to invoke to create their desired behaviour:

1) `freaddireach(pathname, buffer, count)` - This modification is intended to introduce the ability to fetch many small files from an entire directory at once. This is motivated by the directory traversal patterns we observed in the logs, where each element of a directory was iteratively queried, resulting in a large number of syscalls. This call does not require the userspace code to perform any fundamentally different functionality - the inputs of the path of the directory in question and the location of a buffer in which to record the results of the operation are both required by the existing `fstat()`, `open()`, and `read()` calls. This would require the kernel to populate a larger data structure than it already does, but not with data that it does not already return through repeated syscall usage. This syscall could drastically reduce the call bloat associated with reading many small files in a directory, as each file currently causes at least four calls (`fstat`, `open`, `read`, and `close`). This improvement would be especially effective when using libraries such as glib, which introduces a number of `fstat()` calls to `/dev/null` for compatibility reasons.
2) `write_waiting(fd, buffer, count)` - This modification is intended to reduce the number of syscalls made when an application wishes to write out the entire buffer that is normally specified in a `write()` call. A call to `write()` is allowed to return having only partially written the provided buffer, but we observed in the gtk source code that `write()` was used in a looping condition that only terminated on completion of the whole write or an error return value. The library programmer has demonstrated clearly their willingness to wait for the actual completion or unrecoverable failure of the `write()` call, and so the FS API could be extended to support this behaviour. Signals while in kernel space are the likely cause of the frequent interrupting of the write calls we observed, but modern kernels are typically re-entrant, so the early-exit behaviour should have no real impact on the kernel's behaviour.

These examples are by no means exhaustive, and there is no guarantee that they would benefit every program written on top of them. However, they are motivated by our observations that the libraries applications are built on top of are forced to cause syscall bloat. In this, it is assured that, in the context of gedit and gtk, these modifications would reduce the number of FS-related syscalls made if integrated into the application and library source. We leave it to future work to determine whether this reduced number of kernel crossings would precipitate a performance benefit.

B. Future Work

Due to time constraints, the scope of this project was limited to analyzing a simple graphical text editor's behaviour. Collecting traces of the programs with larger code base and more complex behaviour could further validate our claims of the generality of our suggestions to improve libraries and the

FS API. While we believe our workloads are representative of modern end-user document creation applications due to the similarity of our aggregate syscall observations to those found in [2], it would be useful to validate that our observations hold for other classes of applications, such as browsers, or IDEs such as Eclipse[6].

In this project, we ran our experiments on Solaris 11 mainly because of its DTrace support, as the work inspiring this project ([2]) was based on DTrace and we started by replicating its results. This allowed us to quickly move into the source code comprehension phase of our work, which was main focus of this project. However, the use of Solaris is somewhat detrimental to our goal of investigating the behaviour of applications and libraries in practice, as Solaris is not a commonly used desktop operating system. Using a common Linux distribution would be preferable to Solaris, but this would require the effort of extending the existing rudimentary support for DTrace currently available in Linux distributions, or moving to another tool, such as `strace`, which is available on Ubuntu. Our approach - reasoning about application and library behaviour by collecting execution traces - is generic, however, so aside from the additional work of building the new tracing environment and generating new logs, much of our methodology could be reapplied exactly. Experimenting on other operating systems would also validate our claim that the ubiquity of libraries allows us to generalize our suggestions for FS API and library modifications.

Finally, to truly verify the utility of our suggestions for FS API improvements in increasing expressiveness and decreasing syscall bloat, we need to implement them and collect traces with the same workloads we used here. We leave this as future work.

V. Related Work

We derived inspiration from [2] and started by replicating their results in open source domain. The main difference between this work and theirs is that we do not stop at reporting observations of application FS usage. We extend this work by looking at application and library source code to inspect the root causes of proliferous file system calls. Moreover, [2] suggests measuring desktop file system performance with a number of different workloads, while we believe a sample set of representative applications (such as text editors, graphics rendering and web browsing) to suffice for measurement, since many applications use the same generic libraries (such as gtk, gdk and glibc). Because of the commonalities of libraries between many applications, taking measurements from many different applications are unlikely to observe wildly disparate behaviour.

Our observations are to the results mentioned in [1], where `rename()` and `fsync()` are repeatedly used by most text editors to express the semantics of ordered persistent writes. We agree with the analysis in [1] that suggests that the separation of `fsync()` into two separate primitives - `osync()` for ordered writes, `dsync()` for durable writes - would eliminate a fair number of syscalls and increase the expressiveness of the FS API.

[6]https://www.eclipse.org/

Historically, the database community has had a keen interest in the guarantees provided by file system and its performance. One topic of recent discussion between PostgreSQL, kernel and FS developers is about redesigning `fsync()` [4]. This discussion stems from databases' reliance on a transactional model, wherein each complete transaction is expected to be persistently recorded to the disk in a well-defined order. Since there is only one FS API call - `fsync()` - to guarantee the persistence of writes, it is heavily used by database developers. Although `fsync()` allows developers to achieve both *ordering* and *durability*, it is typically computationally expensive. Moreover, if another application is running along the database and it is also makes frequent use of fsync(), each transaction will take longer because has to wait for previous `fsync()` calls' returns. The database community would prefers to have lighter weight, finer granularity sync primitives, such as a non-blocking `sync_file_range()` to allow database and application developers to flush only content relevant to their application. This would allow parallel execution of multiple syncing operations, reducing latency overall for all applications. Another approach would be to have `dirty_background_bytes()` call, which would expose the amount of data not persistently written to the disk yet and enable developers to issue calls to `fsync()` according to their own policies, such as when a certain threshold of un-flushed bytes is reached. These mechanisms allowing for more precise control would mitigate the cost of ensuring data durability by replacing frequent `fsync()` calls with more targeted ones.

There have been several measurement studies [6], [3], [5] investigating FS usage and file access patterns in BSD, Unix and Windows NT systems. In particular, our study shares conclusions with [5] about the file access patterns of applications. Most applications use memory-mapped interfaces to read/write files and most accessed files are small, temporary files. However, to the best of our knowledge, our work is unique in trying to comprehend the FS operations end-to-end – starting from the application source code through to the low-level libraries used to interact with FS – and in identifying potential causes of and solutions to bloated usage.

VI. Conclusion

We conducted experiments to explore the previously-unexamined role of libraries in generating syscall bloat. Our investigation led us to build a body of logs that provide full stack traces for workloads representative of modern end-user application usage, using gedit, a commonly used text editor built on top of widely-distributed libraries. Our analysis required us to develop scripts to aid the comprehension of complex logs and call graphs, with methods that generalize to other domains. Our experiments validated that prior work extends to the wholly open-source environment we use, and informed our suggestion of modifications to the FS API that would allow for cleaner programming of applications and reductions in the number of FS-related syscalls they make.

References

[1] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[2] T. Harter, C. Dragga, M. Vaughn, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. A file is not a file: understanding the i/o behaviour of apple desktop applications. In *ACM Transactions on Computer Systems 2012*, volume 30. ACM.

[3] J. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. pages 15–24, 1985.

[4] H. Robert. Linux's fsync() woes are getting some attention, 2014. Available at http://rhaas.blogspot.ca/2014/03/linuxs-fsync-woes-are-getting-some.html.

[5] D. Roselli and T. E. Anderson. A comparison of file system workloads. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54. USENIX Association, 2000.

[6] W. Vogels. File system usage in windows nt 4.0. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 93–109, New York, NY, USA, 1999. ACM.