Kalpa Publications in Computing
Volume 17, 2023, Pages 81–92
Proceedings of 2023 Concurrent Processes Architectures and Embedded Systems Hybrid Virtual Conference

Is it feasible to identify outputs of an arbitrary process at run time without excessively slowing down workflows?

Philip Shun B. Jensen[1], Iben Lilholm[2], and David Marchant[3]

[1] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
0009-0006-7190-0593
[2] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
0009-0003-6654-3083
[3] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
0000-0003-4262-7138

Abstract

In this study, we explore the feasibility of identifying file events for any process in real-time without significant workflow slowdowns, to aid in generating a data provenance report for the dynamic workflow manager, MEOW. Unlike traditional workflow managers, MEOW's output location isn't pre-defined, and output can initiate another job. We established criteria and examined four Linux tools: strace, perf script, inotify, and fanotify. Our findings suggest that strace meets our requirements, and integrating an strace-based tracer into MEOW is both theoretically and practically viable. While the implemented tracer slows the workflow by approximately 1.3 times, worst-case scenarios show it could be up to 5 times. This research forms the base for constructing MEOW's data provenance report.

1 Introduction

This study is centred around exploring the feasibility of capturing essential information for assembling a data provenance report for the workflow manager when such information is not given by the workflow definitions. Data provenance reports, often used by scientists, offer a comprehensive overview of influences leading to a particular data output [19, p. V],[16, p. 8]. Most workflow definitions will include a complete accounting of inputs and outputs for each step, so it is trivial to compile a provenance report. However, such a complete accounting is not possible in every workflow system, and so some method is needed to identify outputs at runtime. Additionally, considering workflow managers' role in expediting scientific analysis, this identification process must not bog down the system.

This gives us the central research question:
Is it feasible to identify file events of an arbitrary process at run time without excessively slowing down workflows?

To address this, we explore four Linux tools - strace, perf script, inotify, and fanotify - evaluating their alignment with our query and eight essential criteria for future provenance

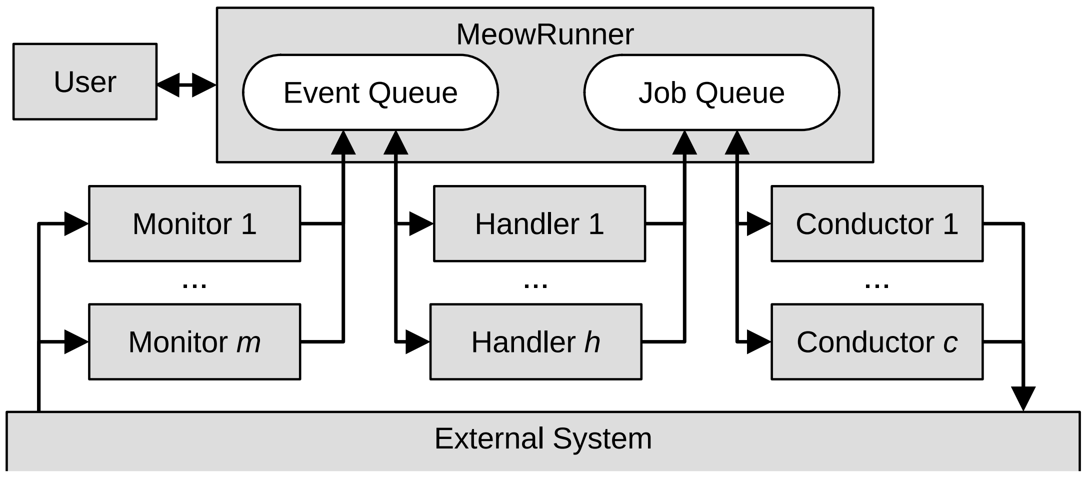

Figure 1: The MeowRunner architecture. An event in the external system may be matched within the monitor, which would then progress through the event queue, handler, job queue and finally conductor before any output is placed back into the external system.

report assembly (detailed in Section 3). Our findings advocate for strace as our primary tool, with discussions on its merits and potential alternatives, such as fanotify. Having integrated strace into MEOW, Section 4 elaborates on the implementation, its scope, and constraints. We then contemplate the implications of our discoveries for future reports, culminating in a summarised conclusion.

1.1 MEOW

MEOW[27] is a system for constructing rules-based workflows, and which is fully adaptable at runtime with workflow steps being added, changed, removed or otherwise reacted to. This is a non-standard way of constructing scientific workflows but allows for far greater adaptability, human-in-the-loop interactions, and error handling. As part of this design philosophy of MEOW, the output of any given job is not specified ahead of time, with jobs completed regardless of what files they do or do not produce. For most operations within MEOW, this is not a problem, but it becomes one with the scientific requirement for provenance reporting. This is especially true within MEOW as a record of what event triggered a job, which then produced output and in turn, triggered a further job would be a necessary part of any provenance report of the MEOW system.

MEOW is a system of definitions, with an implementation of these definitions available in `meow_base`[13]. This provides classes for creating rules and matching against file events, as well as a `MEOWRunner`, for managing the lifetime of MEOW definitions and any analyses that result from them. This can be seen in Figure 1, and is comprised of three sub-components. The *Monitor* will match any events, the *Handler* will create job scripts from these events, while the *Conductor* will execute the job. As part of this, the *Conductor* will also keep a log of when the job is started, under what conditions, and record any errors that take place. As there is nothing in this system that records the outputs of the jobs, the *Conductor* would be an ideal place to implement some system for doing so.

2 Criteria for a tracer

2.1 Need-to-have criteria

In line with our objective, a requisite tool should trace file events (R1) while discerning which process manipulated each file (R2). Additionally, it must ascertain the file linked to an event (R3). Given MEOW's unpredictable output locations, the tool needs a broad scope, tracing file events system-wide and not just at set locations (R4). With MEOW's ability to function multi-threadedly or across a distributed system, tracing must occur robustly, ensuring all file events are accurately captured for data provenance (R5). Moreover, it's essential that this tool efficiently captures file events without drastically impacting workflow speeds, given the pivotal role of workflow systems in expediting scientific analyses (R6).

Summarizing, our ideal tool must:

R1. Trace file events.
R2. Identify the initiator of a file event.
R3. Specify the affected file's path.
R4. Monitor events across the entire file system.
R5. Ensure race-condition-free operations.
R6. Maintain workflow speed.

While it might seem straightforward—like merely tracking new or altered files within set intervals—there's depth to our needs. Since MEOW doesn't designate output folders, there's ambiguity in location tracking. Moreover, frequent directory checks may overlook swiftly executed events, rendering the data provenance report incomplete. Ideally, the tool should either trace file-linked system calls to the kernel or be notified after each successful file event, ensuring race-condition-free interactions with the kernel. It must also sustain a connection to the user space for crucial data such as process IDs and file paths.

Among these criteria, R1 to R5 are binary and resolvable with a "yes" or "no." R6, assessing workflow speed impacts, is more qualitative, influencing tool choice without being strictly binary like its predecessors.

2.2 Nice-to-have criteria

Beyond the initial six criteria, practical application considerations suggest two additional features. First, a tool operable without root access would be advantageous because such access might not always be feasible. Second, considering the rising trend of containerised processing in scientific analysis and ongoing efforts to make MEOW compatible with such processes, the ability to register file events from a mount becomes relevant [25].

Therefore, the two desirable features are:

O1. Can the tool operate without root access?
O2. Is it equipped to trace file events from a mount?

These supplementary criteria, like the initial ones, are binary.

3 Investigation

We've examined four Linux tools for our investigation: strace, perf script, inotify, and fanotify. These were selected based on their documented abilities to robustly monitor file events [12], [10], [9], [8]. The upcoming section provides an overview of these tools, evaluates them against our defined criteria, and delves into their performance impacts. We conducted tests on two specific dual boot machines, detailed in Table 1:

Model:	Lenovo Thinkpad X1 Carbon Gen 8	Windows Surface Pro 4
Memory:	16 GB LPDDR3 2133 MHz	16 GB LPDDR3 1867 MHz
CPU:	Intel®Core™ i7-10510U CPU @ 1.80 Ghz × 8	Intel® Core™ i7-6650U CPU @ 2.20GHz × 4
GPU:	Mesa Intel® UHD Graphics	Mesa Intel® Iris® Graphics 540 (SKL GT3)
OS:	Ubuntu 22.04.2 LTS 64-bit	Ubuntu 22.04.2 LTS 64-bit

Table 1: Computer specifications

Initially, we contemplated addressing this challenge on Windows OS. But its complexity, combined with the effort required to navigate it, made Linux a more pragmatic choice. Nonetheless, for those curious about the Windows angle, a summary can be found in section 6.

3.1 Tools Overview

strace is a command-line utility for tracing system calls and signals using the Linux kernel's ptrace feature [12]. It's found in most Linux distributions and is actively updated, with the latest commit in May 2023 [6]. While strace serves various purposes, like diagnostics and debugging, we only utilized its command-line version in our project (detailed in section 4). There's also a Python wrapper named pystrace that offers enhanced functionality [29].

Perf tools, kernel-based performance counters for Linux, offer a broader analysis spectrum than strace [4]. Originating from the Linux kernel version 2.6.31, users need to install them separately, commonly through the linux-tools package for Ubuntu [30]. Our focus lies on perf script, a tool that traces system calls into an intelligible report [11]. This tool shares similar functionalities with strace. Unlike strace, we found no Python wrappers for perf tools, probably because their core purpose is performance analysis rather than software integration.

inotify is a kernel API that effectively monitors file and directory alterations by issuing notifications [9]. Integrated since Ubuntu version 2.6.13, it's a tool currently maintained, with the latest commit in April 2023 [14]. Beyond its fundamental role, libraries such as inotify_simple and inotify exist in Python, aiding in file monitoring. One can also use the inotifywait command-line tool by obtaining the inotify-tools package [2].

Akin to inotify but with enhanced capabilities, fanotify was introduced in the Linux kernel version 2.6.36 and supports comprehensive file system event monitoring [8]. This tool is updated actively, as evident from the recent commit in April 2023 [15]. It's superior to inotify in functionalities like monitoring an entire mounted file system and accessing files before other applications. fanotify's functionalities can be harnessed via programming languages, either through C or Python wrappers like pyfanotify [18]. No command-line invocation for fanotify is known to us.

3.2 Criteria Evaluation for the Tools

3.2.1 Register File-related Events/System Calls (R1)

While strace and perf script trace system calls initiated by specific processes, inotify and fanotify respond to notifications upon file events in monitored directories. Therefore, all four tools are equipped to register file-related system calls or event notifications.

3.2.2 Trace Process Initiating File Event (R2)

Identifying the process ID responsible for a file event is pivotal when handling concurrent tasks. Both strace and perf script can provide the process ID when correctly configured, as can fanotify [12], [10], [8]. Conversely, inotify cannot offer the process ID or the user details associated with an event [9], complicating the distinction between file events from various origins, significantly when jobs overlap in output locations.

Although alternate methods might distinguish different tasks and resources, such as custom IDs, we found no straightforward solution. Given that the other three tools can trace process IDs and considering our project's time and resource limits, further exploration was postponed.

3.2.3 Path Provision for File/Directory Events (R3)

For thorough provenance documentation, it's crucial to identify the path of adjusted files. Since 2012, strace can decode file descriptor paths by using ptrace to get the memory address of the file linked to a system call and then using PTRACE_PEEKDATA to try to retrieve the file path in string format. In contrast, perf script doesn't provide paths directly; it gives hexadecimal path names because it operates at the kernel level [21]. To acquire paths in a string format using perf script, additional procedures involving probes are necessary, as referenced by De Melo in 2015 [20, slide 28]. Inotify provides a relative path when a file event is detected within a watched directory; users familiar with the directory's position in the filesystem can extrapolate the absolute path. Fanotify, on the other hand, gives a metadata structure during event interception, which can lead to the file path if there's no queue overflow, as illustrated in the fanotify manual.

3.2.4 File Event Registration Across File System (R4)

For accurate detection, tools need to register file events throughout the file system without predetermined locations. Strace allows users to focus on specific system calls. When set to the trace=file option, it observes all related file actions. Given that strace tracks the system calls of any process, it identifies events at any location within the system. Perf script, resembling strace, logs a process's system calls regardless of their location in the file system. In contrast, inotify has limitations. It can't monitor the entirety of a system recursively but only works within directories, excluding their subdirectories. While it's feasible to watch sub-directories separately, it introduces complications like race conditions. However, fanotify, when utilizing the "file system" or "mount" settings, can observe directories on a recursive basis, encompassing the whole system. Conclusively, except inotify, all tools can survey file events across the complete file system.

3.2.5 Avoid Race Conditions/ Robustness (R5)

To ascertain the tools' robustness against race conditions, we ran a Python script, timing-Python.py, which generated eight million file events by creating and deleting two million files and directories. This script was multi-processed to emulate race conditions. For full script details, refer to our repository meowBa [24]. Throughout this paper, you will see the use of both repositories meowBa and meow_base, the main difference being that meowBa is a working development clone of meow_base that we set up for investigation and implementation. It is also worth mentioning the meow_base now has included an option to use our tracer.

Each tool was operated on this script, and the outputs were analyzed to ensure all events were registered. Strace, perf script, and inotify effectively registered all events. Conversely, fanotify faced challenges.

Fanotify's Limitations Despite using both the Python-based pyfanotify library and a C implementation (based on fanotify's manual [8]), fanotify didn't consistently register all events, especially during simultaneous file creation and deletion. Observations suggest potential race conditions, especially as implementing a sleep() function improves its performance. Monitoring via htop revealed a spike in thread count during fanotify's operation, hinting at possible thread management issues.

Moreover, while the fanotify documentation mentions the possibility of queue overflow, our tests showed inconsistent results even when using the UNLIMITED_QUEUE flag. There were challenges in diagnosing queue overflow due to our recursive tracing implementation, which relies on file handles, making overflow flag checks unreliable.

Concluding Remarks on Race Conditions Fanotify's behaviour implies potential race conditions, thread management issues, or both. In contrast, strace, perf script, and inotify proved robust against these challenges.

3.2.6 Tool Operation without Root Access? (O1)

For our ideal use, we'd prefer tools that don't need root access due to varied execution environments. Strace fits this description, requiring no root when tracing processes owned by the user [12]. Perf script, on the other hand, can run without root after an initial setup, but given that this setup does need root access [5], we lean towards labelling it as root-dependent. Inotify simply informs users about filesystem events for files or directories they can already access, eliminating the need for root. Fanotify primarily demands root access, especially its ability to call functions like mount [1]. Though there's an option to run it without such access, it doesn't fully serve our requirements [22].

3.2.7 Registering File Events from a Mount (O2)

If a tool needs to operate from a mounted directory, strace is capable of tracing both the mounting procedure and any ensuing jobs, as long as the user initiating these processes has the right permissions. Perf script operates similarly to strace, but either requires the user to have root access or an admin's prior setup to function on such directories. Inotify is functional on directories mounted beforehand, but it's limited to watching only those directories without including their subdirectories. Lastly, while fanotify can work once a directory is mounted, it might not immediately start its monitoring. As with its other operations, running fanotify on a mounted directory demands root access. In essence, every tool can function on a mounted directory, but both perf script and fanotify require the user to possess root access on the host system of the mount.

3.2.8 Slowdown (R6)

The slowdown requirement, addressing speed, is a qualitative consideration for dynamic file event tools. We conducted timing tests on two laptops to gauge workflow delays across different hardware, with machine specifications provided in Table 1.

We assessed the four tools on varied scripts:
1. A bash script looped a hundred thousand times, resulting in eight hundred thousand file events, aiming to analyze potential worst-case scenarios. This test is stored in our meowBa repository [24].
2. A Python script, performing similarly to the bash script, looped a million times across two threads, resulting in eight million file events.
3. Another Python script that runs a scientific analysis similar to a known study [28]. It processes 15 .npy data sets of size 1.1 MB, yielding a JupyterNotebook for each. This test mimics MEOW's behaviour for comparative analysis.
Notably, fanotify's race conditions might have affected its timing results. Strace and perf script timings were obtained via the command line. In contrast, inotify and fanotify relied on Python libraries and were timed through their Python implementations, possibly granting them a speed advantage. All tests were executed five times on each device, with averages used for slowdown calculations.

Bash script timings Table 3 showcases the timings for a bash script. The results indicate that strace slows down by x3.26, perf script by x2.20, while inotify slightly speeds up the process. The speedup by inotify can be attributed to the separate process in which the bash script runs.

Python script timings When running the Python script, perf script has the least slowdown at x1.27, followed by inotify at x1.55, and strace with the most at x4.88.

Scientific analysis For the timings of scientific analysis, Strace results in the highest slowdown at x1.24, while perf script and inotify have marginal slowdowns of x1.03 and x1.01, respectively.

Fanotify timings Table 4 highlights fanotify timings. For the bash script, fanotify's slowdown is x1.13, less than other tools. For the Python script, its slowdown is x2.06, despite missing half the events. This suggests that, if accurate, fanotify may offer better performance than strace.

3.3 Investigation Results

Our research details the seven binary criteria and timing results for Linux tools as illustrated in the provided tables. Table 2 showcases which tools meet the seven binary yes/no criteria, while Table 3 compares the slowdown from various tools and scripts.

	Need-to-have					Nice-to-have	
	R1	R2	R3	R4	R5	O1	O2
strace	✓	✓	✓	✓	✓	✓	✓
perf	✓	✓		✓	✓		✓
inotify	✓		✓		✓	✓	✓
fanotify	✓	✓	✓	✓			✓

Table 2: Summary of which tools have which of the seven binary criteria

Slowdown			
	bash	Python	Scientific Analysis
strace	x3.26	x4.88	x1.24
perf	x2.20	x1.27	x1.03
inotify	x0.98	x1.55	x1.01

Table 3: Overview of the slowdown from tools and scripts

fanotify timings		
	Slowdown	Average events missed
bash base	x1.00	0
bash trace	x1.13	72
Python base	x1.00	0
Python trace	x2.06	4152756

Table 4: Timing of analysis in workflow manager

Strace stands out as it meets all the binary criteria. Its main drawback is a significant slowdown, especially when monitoring numerous file events, experiencing up to a x4.88 slowdown. However, in more realistic scenarios, the slowdown becomes more tolerable at x1.24. For efficiency, it's recommended to trace the least amount of file events to minimize this slowdown.

Perf doesn't satisfy all criteria, with a notable gap in the path criteria. Although its slowdown is more favorable than strace's, its requirement for root access complicates its implementation. Its failure to track file paths makes it unsuitable as a primary file event tracer.

In terms of slowdown, Inotify is superior. Yet, it misses the mark on two critical criteria: its inability to trace the process initiating a file event and to trace file events across the entire file system. Its functionality is contingent upon tracing events in a pre-specified directory, a limitation for our broader requirements.

Fanotify's primary concerns are its need for root access and its inconsistent registration of all file events. A potential race condition or queue overflow warrants further study. Should its event registration limitation be rectified, fanotify could emerge as a potent rival to strace due to its anticipated lower slowdown.

3.3.1 Choosing strace for our implementation

Strace, meeting all binary criteria, is selected as our tool for MEOW. While the slowdown is a concern, particularly in scenarios with vast file events, our current strategy will only trace file creation events to curb this issue. If future needs demand a more expansive scope, fanotify could be the better pick, assuming its constraints are addressed.

4 Implementation

This section delves into MEOW's deeper aspects, details its implementation, covers black box tests designed for our implementation, and estimates the slowdown from our strace-based file event tracer.

4.1 MEOW

Originating from David Marchant's PhD thesis [27] for the University of Copenhagen's MiG [17], MEOW has evolved for broader applications. Its essence can be accessed on its GitHub page. At its core, MEOW is an event-driven scheduler: users define patterns and recipes, which together form a rule. The recipe encompasses the processing component, while the pattern specifies the conditions for triggering a process.

For a more detailed breakdown of MEOW's requirements, refer to tables 6.1-6.3 in Marchant's work [26].

In its updated form, meow_base consists of four core elements: runner, monitor, handler, and conductor. The runner oversees event flow among these components. Events are watched by the monitor, sent to the runner if they match any rule, processed by the handler into jobs, and executed by the conductor. Crucially, when the conductor writes matching patterns to the monitored filesystem, it triggers a new job execution. This sequence is vital for a comprehensive data provenance report. Thus, our tracer focuses on registering file events as the conductor executes jobs.

4.2 Tracer using strace

Based on our investigation, we've implemented a tracer using strace, noting it met our criteria, albeit with some performance slowdown.

Understanding strace and MEOW's base_conductor simplified its application. The base_conductor calls a bash script through subprocess.call(). By setting the shell option to True, we can directly invoke strace [7]. This led to a straightforward alteration in the subprocess call to integrate strace, specifying the type of system calls to be traced [12].

Upon successful subprocess call, the .trace file is parsed, extracting created files that are appended to the job's metadata. The .trace file, depending on its size, can be substantial. For instance, tracing the Python script with eight million file events resulted in files ranging from 1 to 2 GB. However, in a regular MEOW scientific analysis context, these files are typically about 0.5 MB per job. To optimize storage, we promptly delete the .trace files after parsing the necessary data.

4.3 Tests

Our implementation underwent black-box testing to ensure that strace effectively captures file events and to validate the parsing of its output. Using existing pytest instances from the meow_base repository [13], we modified them to verify the correct file logging in a job's metafile. Four tests were crafted to evaluate different scenarios including Python scripts, linked scripts, file creation in varied locations, and multithreaded file creation. Although we missed testing with a bash script, our Python scripts initiate from a bash script, thus it wasn't deemed a priority. These tests can be viewed in our repository at meow_baseStrace/tests/test_tracer.py [24].

4.4 Timings

We conducted a nuanced performance assessment by running MEOW with and without the tracer across 11 different dataset sizes, ranging from 1 to 49. A regression analysis yielded the equation $1.04 + 0.0751 \ln x$ with an R^2 of 0.884. This suggests that the slowdown is both minimal and logarithmically related to the number of datasets.

Table 5: Slowdown by Dataset Size

Size	1	5	10	15	20	25	30	35	40	45	49
Slowdown	0.99	1.2	1.23	1.26	1.3	1.3	1.32	1.29	1.29	1.29	1.29

The regression model captures the overall slowdown trend effectively, although individual dataset sizes may experience varying slowdown rates. Importantly, the slowdown is logarithmically bounded, making the tracer's performance impact manageable even as dataset sizes grow.

4.5 Implementation Discussion

4.5.1 Output File Size

A key concern with strace is its potentially extensive log file. Though we only request strace to trace certain system calls, kernel-level operations might produce a larger log. To manage this, we promptly delete the log post-extraction of essential details. While tools like logrotate might help manage large logs [3], using them with strace is not straightforward due to output redirection constraints [12]. The -P flag in strace could offer a solution, but it posed challenges during our implementation.

4.5.2 Performance Impact

The x1.30 slowdown observed is not negligible. This might be minimized by altering where strace is invoked. Presently, strace monitors all related activities, from running the bash script to the specific analysis. Ideally, invoking strace closer to the analysis initiation might reduce the slowdown. While we integrated strace within the base_conductor for metafile updates, it's conceivable to adjust MEOW to enable updates within the bash script, possibly leading to performance improvements.

5 Results

Our research confirms the feasibility of identifying file events in real time without causing excessive slowdowns in workflows. Specifically, we found strace to be the only tool meeting all our essential criteria from section 2. It is particularly relevant when the necessity arises to trace multiple types of file events. Although fanotify is a promising alternative, it does have limitations such as the need for root access and potential race conditions that may be resolvable.

Our integration of strace into MEOW demonstrates a variable slowdown pattern, as indicated by our regression analysis: $1.04 + 0.0751 \ln x$ with an R^2 of 0.884. This shows that the slowdown isn't constant but increases logarithmically with the number of datasets involved in the analysis. Each traced system call halts the kernel, thus affecting the workflow. Therefore, the slowdown is not merely a static value but is dependent on the complexity of the file events in the workflow. This regression formula offers a nuanced and accurate representation of the performance impact.

6 Future Work

6.1 Data Provenance

The ultimate goal of this study was to set the stage for a data provenance report in MEOW. Subsequent work will harness our insights to devise such a report.

6.2 Tracer Refinements

Improvements in the strace integration can potentially lessen the slowdowns in MEOW. This would involve solving challenges in thread-safe metadata updates deeper in the MEOW architecture, as described in section 4.

6.3 Fanotify Resolutions

Rectifying the issues we've encountered with fanotify may make it a more efficient substitute to strace. Our fanotify codebases, written in both Python and C, are accessible in our repository [24].

6.4 Windows Exploration

While Linux is the primary focus, extending our research to Windows could be informative. In Windows, similar functionality could be achieved through the development of a file system filter driver, known as a mini-filter. Although our initial assessment found the framework restrictive and time-consuming for development [23], exploring this avenue could provide valuable comparative insights between operating systems. This is not a high priority but would enrich the scope of our research.

7 Conclusion

We set out to explore 'Is it feasible to identify outputs of an arbitrary process at run time without excessively slowing down workflows?'. We formulated five binary and one qualitative "need-to-have" criteria, along with two "nice-to-have" binary criteria to guide our evaluation of potential tools. We systematically examined four Linux tools against these criteria and identified that strace met the most conditions.

Following this, we went beyond theoretical assessment to practical application, implementing an strace-based tracer into MEOW, an existing workflow manager. This implementation was rigorously vetted using black-box testing techniques, ensuring its accuracy and reliability. To quantify the impact on workflow speed, we employed regression analysis. The resulting equation $1.04 + 0.0751 \ln x$ with an R^2 of 0.884 indicates that the slowdown is generally moderate and strongly correlated with the logarithm of the number of files processed.

Our study lays the foundation for a future data provenance report in MEOW, enabled by the tracer's capabilities. With the question of feasibility largely answered affirmatively—subject to the acceptable levels of workflow slowdown—developing this data provenance report seems to be the next natural course of action in our research.

References

[1] capabilities(7) - linux manual page.

[2] inotifywait(1) - linux man page.
[3] logrotate(8) - linux man page.
[4] perf - linux manual page.
[5] Perf events and tool security.
[6] strace github commmits.
[7] subprocess — subprocess management.
[8] fanotify(7) - Linux manual page, 08 2021. publisher: man7.org.
[9] inotify(7) - Linux manual page, 03 2021. publisher: man7.org.
[10] perf-trace(1) - Linux manual page, 06 2021. publisher: man7.org.
[11] perf-script(1) - Linux manual page, 08 2022. publisher: man7.org.
[12] strace(1) - Linux manual page, 10 2022. publisher: man7.org.
[13] Github repository for meow_base project, 2023. being develope, branch `stable`.
[14] linux/fs/notify at master · torvalds/linux · github. 2023.
[15] linux/fs/notify at master · torvalds/linux · github. 2023.
[16] Mehmet S. Aktas, Beth Plale, David Leake, and Nirmal K. Mukhi. *Unmanaged Workflows: Their Provenance and Use*, pages 59–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
[17] Jonas Bardino, David Marchant, Rasmus Munk, and Martin Rehr. migrid.
[18] Alexander Baskikh. pyfanotify · pypi. 07 2022.
[19] You-Wei Cheah and Beth Plale. Provenance analysis: Towards quality provenance. In *2012 IEEE 8th International Conference on E-Science*, pages 1–8. IEEE, 2012.
[20] Arnaldo Carvalho de Melo. Linux Perf Tools (Probe & Trace), 02 2015.
[21] Arnaldo Carvalho de Melo. Re: perf trace: file names, strace groups, 06 2015.
[22] Amir Goldstein. fanotify: support limited functionality for unprivileged users - Patchwork — gmail.com. https://patchwork.kernel.org/project/linux-fsdevel/patch/20181105132816.12241-1-amir73il@gmail.com/, 11 2018.
[23] Lori Whippler Hollasch, Andrew Kim, Sameer Saiya, and Matt. File systems and filter driver design guide - windows drivers, 09 2022.
[24] Iben Lilholm and Philip Shun Jensen. Github repository for our bachelor project, 2023.
[25] J. Luo, M. Abbasi, Q. Zhang, X. Sun, N. Mohamed, M. Mukherjee, M. Chiang, and A.V. Dastjerdi. Container selection processing implementing extensive neural learning in cloud services, 06 2021.
[26] David Marchant. Enabling dynamic scheduling of scientific analysis. 2021.
[27] David Marchant. Events as a basis for workflow scheduling. 2023.
[28] D.M. Pelt, A.A Hendriksen, and K.J. Batenburg. Foam-like phantoms for comparing tomography algorithms. 1(29):254–265, 2022.
[29] Joao Pinto. pystrace 0.0.3.
[30] swift.org. Linux perf.