

ParaTrac: A Fine-Grained Profiler for Data-Intensive Workflows

Nan Dun
Department of Computer
Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-Ku
Tokyo, 113-5686 Japan
dunnan@yl.is.s.u-
tokyo.ac.jp

Kenjiro Taura
Department of Information and
Communication Engineering
The University of Tokyo
7-3-1 Hongo, Bunkyo-Ku
Tokyo, 113-5686 Japan
tau@logos.ic.i.u-
tokyo.ac.jp

Akinori Yonezawa
Department of Computer
Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-Ku
Tokyo, 113-5686 Japan
yonezawa@is.s.u-
tokyo.ac.jp

ABSTRACT

The realistic characteristics of data-intensive workflows are critical to optimal workflow orchestration and profiling is an effective approach to investigate the behaviors of such complex applications. *ParaTrac* is a fine-grained profiler for data-intensive workflows by using user-level file system and process tracing techniques. First, *ParaTrac* enables users to quickly understand the I/O characteristics of from entire application to specific processes or files by examining low-level I/O profiles. Second, *ParaTrac* automatically exploits fine-grained data-processes interactions in workflow to help users intuitively and quantitatively investigate realistic execution of data-intensive workflows. Experiments on thoroughly profiling Montage workflow demonstrate both the scalability and effectiveness of *ParaTrac*. The overhead of tracing thousands of processes is around 16%. We use low-level I/O profiles and informative workflow DAGs to illustrate the vantage of fine-grained profiling by helping users comprehensively understand the application behaviors and refine the scheduling for complex workflows. Our study also suggests that current workflow management systems may use fine-grained profiles to provide more flexible control for optimal workflow execution.

Categories and Subject Descriptors

D.2.5 [Software]: Testing and Debugging—*tracing*; D.4.8 [Software]: Performance—*modeling and prediction*

General Terms

Performance

Keywords

Profiling, workflow, tracing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HPDC '10 Chicago, Illinois USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

1. INTRODUCTION

With the advance of high performance distributed computing, users are able to execute various data-intensive applications by harnessing massive computing resources [1]. Though workflow management systems have been developed to alleviate the difficulties of planning, scheduling, and executing complex workflows in distributed environments [2–5], optimal workflow management still remains a challenge because of the complexity of applications. Therefore, one of important and practical demands is to understand and characterize the data-intensive applications to help workflow management systems (WMS) refine their orchestration for workflow execution. Research has been conducted to elaborate the characterization of a wide variety of scientific workflows using synthetic approaches [6, 7]. However, these methods require application knowledge from domain researchers and do not include runtime information extracted from realistic execution. Thus, it will be helpful for workflow researchers if they can acquire the workflow essentials simply by running their own original applications.

Profiling is an effective dynamic analysis approach to investigate complex applications in practice. By using user-level file system and process tracing techniques, we designed and implemented a fine-grained profiler called *ParaTrac*, for data-intensive workflows. *ParaTrac* allows users to trace unmodified applications and investigate them in different aspects by automatically generated workflow profiles. First, *ParaTrac* produces valuable low-level I/O profiles that allow users to quickly understand the I/O characteristics of from entire application to specific processes or files. In addition, *ParaTrac* is novel because it can exploit fine-grained data-process interactions in realistic execution to help users intuitively and quantitatively investigate data-intensive workflows. *ParaTrac* automatically generates workflow DAGs annotated with runtime information and the characteristics of workflow can be explored by applying graph manipulations and graph-theoretic algorithms to corresponding DAGs. Besides, users are allowed to customize those profiles or create new ones by querying raw trace logs stored in SQL database.

The significance of this work is twofold. First, *ParaTrac* provides an effortless way to extract and generate informative and comprehensive workflow profiles from fine-grained profiling data, which enables users to intuitively and quantitatively analysis, debug, and refine their own

workflows. Second, fine-grained profiling implies its potential support for fine-grained and realistic scheduling of workflows. Therefore, fine-grained profiling by ParaTrac suggests not only the vantage of more accurate study of workflows, but also the feasibility of enabling more flexible workflow controls for future workflow management systems.

2. RELATED WORK

File system tracing, either in kernel-level or user-level, is widely used for file system performance and application I/O analysis [8–11]. Though kernel-level tracing provides a deeper probe of file system status with less overhead, user-level tracing demonstrates more advantages in practice. First, user-level approaches can be used by non-privilege users. Second, user-level interfaces (e.g., FUSE APIs [12]) are more portable than kernel-level interfaces in most architectures. For example, DFSTrace [8] and Tracefs [10] uses kernel trace module that has to be modified when the kernel changes. Third, kernel approaches target on tracing the activities of the whole file systems thus introducing overhead to the whole file system. Different from DFSTrace and Tracefs, ParaTrac is designed for tracing individual user applications and is non-obtrusive for other users in the time-sharing system. Additionally, ParaTrac logs both file system calls and process runtime to generate various workflow profiles, which distinguishes itself from other user-level file system tracer, such as LoggedFS [11].

The understanding of I/O characterization of workloads can benefit many areas of data-intensive computing practice [13–16]. The static analysis [14] can be used if the knowledge of application (e.g., source code, documentation, data structure and algorithms, etc.) is available. However, this approach requires non-trivial effort and even users may not be able to acquire enough knowledge of applications (e.g., no source code but only binaries). Instead, the dynamic approach, generally trace-based, is more practical and effective even for parallel applications [13, 15].

Data-job interactions in workflow execution are critical to scheduling and management of data-intensive workflows [17–19]. Besides the problem of acquiring application knowledge, static analysis is unable to capture the runtime behaviors of workflow, especially with constraints encountered in dynamic environments, such as data congestion due to unexpected usages of shared network links by other applications. In response to this problem, recent workflow study turns to integrate realistic execution information into workflow analysis and synthesis [6, 7]. However, this information is usually provided at a coarse-grained level (i.e., job-level) because it comes from either the experiences of domain researchers [7] or specific interfaces of workflow management systems [6, 20, 21]. Therefore, ParaTrac is designed to be a general-purpose tool that allows users to acquire fine-grained workflow information by effortlessly profiling any workflows.

Fine-grained profiling also suggests its potential for future workflow management techniques. Current workflow management systems (e.g., Pegasus [2, 3]) uses coarse-grained level (i.e., job and data) scheduling to manage workflow execution [5]. Thanks to the advance of multi-core and distributed data storage technologies, we argue that fine-grained scheduling is a worthy alternative for optimal workflow management. For example, the workflow makespan can be minimized by scheduling tasks with high parallelism on computing resources with more concurrent

processing power or by striping data for transfer instead of passing the whole data. Besides, fine-grained scheduling is demonstrated to be able to improve the execution accuracy in high volatility environments [22].

As the abstract of workflow execution, profile can be used for head-to-head comparisons between different workflows for many purposes. For example, a public repository of scientific workflow benchmarks will contribute to the evaluation of workflow systems [6]. For this reason, profile data generated by ParaTrac is designed to be portable and extensible so they can be reused by other researchers.

3. PROFILING APPROACHES

3.1 User-Level Tracing Techniques

ParaTrac traces all file system calls invoked during the workflow execution, as well as files and processes that refer to these file system calls. Table 1 summarized the logged runtime information by ParaTrac.

3.1.1 File System Call Tracing

FUSE (Filesystem in Userspace) [12] is a framework that lets users develop their own user-level file systems without touching the kernel. By implementing FUSE APIs, users can map file system calls to specific userspace interfaces and unmodified binaries can run above the corresponding FUSE file systems. For example, the `read` file system call in ParaTrac is implemented as shown in Figure 1, where logging procedures are shown in lines marked with “+”.

However, there are two limitations when using FUSE to trace file system calls. First, only those file system calls accessing files or directories under the directory mounted by FUSE program (i.e, FUSE mount point) can be traced. Files outside the trace directory (e.g., shared libraries and header files) are out of the tracing coverage. Using `chroot` [23] is a straightforward workaround, but it usually requires superuser privileges. For non-privileged users, `fakechroot` [24] and `fakeroot` [25] utilities can be used instead. Second, old FUSE implementation (version < 2.7.x) [12] does not allow accesses of character devices (e.g., `/dev/null`) from user-space. CUSE (Character Device in Userspace) [26] coming with latest versions of FUSE 2.8.0 [12] enables the control of character devices in user space and thus solves this problem.

3.1.2 Process Tracing

The `/proc` file system [27] is a standard interface in Unix-like system via which a wealth of information of kernel and running processes can be obtained. The overall information of process with `#Pid` can be retrieved by parsing corresponding files under the directory `/proc/#Pid/`.

Besides `/proc` file system, Linux kernel provides another interface called `Taskstats` [28–30] to acquire process runtime statistics information. It sends process statistics from the kernel to userspace via `netlink` when process exits. ParaTrac uses `taskstats` interface because the begin time and elapsed time of process can not be retrieved from `/proc` file system.

3.1.3 Interface to Workflow Management Systems

Allowing the scheduling information to be passed from workflow management systems to ParaTrac is also important, because it helps users understand how the workflow was planned and scheduled. For example, ParaTrac can

Table 1: Summary of traced runtime information

Variables	Interfaces	Usages
<i>Process and Task</i>		
Pid	FUSE APIs	Process logging and statistics
Parent Pid	/proc filesystem	Process relationships analysis, e.g., process tree
Command line	/proc filesystem	Task logging, e.g., task name, execution parameters
Start time	kernel taskstat	Task statistics
Life time	kernel taskstat	Task statistics, performance analysis
Environ	/proc filesystem	Auxiliary interface to WMSs, e.g. job id, scheduling priority
<i>System Call</i>		
Time Stamp	FUSE APIs	Temporal analysis for traced events
File path	FUSE APIs	File logging and statistics
System call	FUSE APIs	System call logging and statistics
Return value	FUSE APIs	Workflow debugging, failure detection
Latency	FUSE APIs	Performance analysis, e.g., throughput
<i>Input and Output</i>		
Bytes	FUSE APIs	I/O volume and throughput
Offsets	FUSE APIs	I/O sequentiality, access pattern
Lengths	FUSE APIs	I/O regularity, e.g, random access
File size	FUSE APIs	I/O throughput, data distribution

```

static int fuse_read(const char *path,
char *buf, size_t size, off_t offset,
struct fuse_file_info *fi) {

/* retrieving file descriptor */
tracker_file_t ff = (tracker_file_t)
(uintptr_t) fi->fh;
fd = ff->fd;

/* invoking real system call
* and timing latency */
+ clock_gettime(CLOCK_REALTIME, &start);
res = pread(fd, buf, size, offset);
+ clock_gettime(CLOCK_REALTIME, &end);

/* logging system call */
+ size_t length = res == -1 ? 0 : res;
+ syscall_logging(&tracker.fs.read,
+ SYSC_FS_READ, &start, &end,
+ fuse_get_context()->pid,
+ res, path, length, offset);

/* set return value and return */
}

```

Figure 1: FUSE implementation of traced file system call

integrate WMS job id or priority into workflow profiles for further analysis. To achieve this purpose, ParaTrac provides an interface to log environment variables that can be exported by other applications. We believe this approach is flexible and requires minimal modification of existing WMSs to adapt to this feature.

3.2 Implementation

ParaTrac consists of a FUSE based tracing program written in C and a profiler generator written in Python. The tracer can be easily ported to other platforms. For MacOS and other UNIX-like system, there are already several ports (e.g., MacFUSE [31]). For windows systems, there are also FUSE-like attempts [32] using IFS (Installable Filesystem KIT) [33] or FIFS [34]. This design maximized the portability of tracer deployment, since only the tracer

needs to be installed wherever workflow executes. The design also allows user to collect application data by their own approach. As long as the log are stored in database using the same format as ParaTrac does, further log analysis and profile generation can be performed by profile generator without modification.

4. PROFILING PROCEDURE

4.1 Application Tracing

Tracing a standalone application using ParaTrac is as easy as running the application as usual but changing its working directory to the traced directory. Figure 2 illustrates the standard steps to start and finish a trace. As discussed in Section 3.1.1, a partial tracing (i.e., without “+” marked steps) only records those accesses through the mounted directory. For many applications, excluded files in partial tracing are system-wide shared files and can be ignored for many purposes. Nevertheless, a full tracing (i.e., without “-” marked steps) can also be performed using `chroot/fakechroot` utilities.

The scalability of both trace of parallel applications and offline trace log analysis are important for efficient large-scale profiling [35]. Since there is no global knowledge that needs to be shared among tracer instances during tracing time, tracing parallel application is straightforward by starting one tracer for every node where application is running. In practice, we use GXP (Grid and Cluster Shell) [36,37] to deploy ParaTrac, concurrently initialize and finalize the tracer instances in distributed machines.

4.2 Log Persistence

After tracing is finished, the raw data of traced events as shown in Table 1 are stored in four plain text CSV (Comma Separated Values) logs: runtime log, file log, process log, and system call log.

As the preamble of profiles, runtime log includes the runtime environment, configurations and other general tracing information. File log stores a index of files that have been accessed. Process log, like the file log, stores a index from process id to process information. System call

```

#!/bin/bash
wdir=/full/path/of/working/directory

# Create mount point and start tracing
mkdir mnt && ./ftrac mnt

- # Partial tracing initialization
- # cd working directory via mount point
- cd mnt/$wdir

+ # Full tracing initialization
+ # using fakechroot
+ fakechroot
+ /usr/sbin/chroot mnt

# Run the application
./run_application

# Partial tracing finalization
- cd $wdir

# Full tracing finalization
+ exit # exit /usr/sbin/chroot
+ exit # exit fake-chroot

# unmount and stop tracing
fusermount -u mnt

```

Figure 2: Standard steps to trace an application

log is the essential part of the logs, which includes all system call events with their parameters, return value and elapsed time. Therefore, the whole activities and information of a specified file or process can be figured out by cross-indexing these logs.

Before the profile generation, all logs are aggregated into a SQLite [38] database because it is much easier and more portable to query records from database than plain text files. In parallel tracing, all dispersed local trace logs are collected and integrated into one database for further processing.

5. PROFILE GENERATION

ParaTrac uses two complementary approaches to workflow. Statistical analysis is one of standard ways to examine large amount of data and discover their trends and correlation. Temporal and causal analysis is effective to explore the orders and dependencies of events. Automatically generated profiles include data tables, charts, and graphs to help user intuitively and quantitatively explore the essentials of a workflow.

5.1 Data Access Characteristics

ParaTrac has integrated most of standard statistical tools to automatically produce statistics on specific file system call to help user effortlessly examine the data access characteristics of application.

For example, the histograms of system call count can reveal the dominant file operations in an application. The average and standard deviation of system call latency enable users to find out the volatility of a specific file operation, or quickly locate the bottlenecks of data access in execution.

For data-intensive applications, we are especially interested in understanding their I/O behaviors since I/O is a dominant factor affecting workflow execution performance. By conventional I/O analysis approaches [14–16], ParaTrac is able to discover the I/O throughput, and I/O access

patterns, such as the regularity and sequentiality of I/O operations.

5.2 Process Tree

ParaTrac uses a directed tree to illustrate the parent-child (i.e., `fork`) relationships between processes. It highlights the processing patterns of applications from the viewpoint of process hierarchy. Here, the *causal order* between processes is defined as follow:

$$p_i \text{ is the parent process of } p_j \Rightarrow C_{fork} : p_i \prec p_j \quad (1)$$

Accordingly, a process tree G_{proc} can be defined as:

$$\begin{aligned}
G_{proc} &= (V, E) \\
V &= \{p_i | p_i \text{ is the process spawned in application}\} \\
E &= \{(p_i, p_j) | p_i, p_j \in V \text{ and If } p_i \prec p_j\}
\end{aligned}$$

By recursively reasoning each parent-child relationship that appeared in process log, a process tree can be constructed.

There are a few potential usages of process tree. For example, the makespan of the entire or partial workflow can be approximately estimated by finding the longest path in the tree if using process life time as the edge weight. The processing pattern can be also identified by clustering nodes into subtrees with the same pattern.

5.3 Workflow DAG

A workflow is usually represented by a DAG (Directed Acyclic Graph) [6, 7, 20] for the convenience of studying the dependencies/interactions between data and processes. As the essential part of workflow profile, we first elaborate the construction of fine-grained workflow DAG by using tracing data, and then describe how to apply graph-theoretic algorithms to extract useful workflow information for various purposes.

5.3.1 DAG Construction

A workflow DAG is constructed similarly to the way of a process tree being constructed. Besides Equation 1, additional causal orders are required to define a workflow DAG.

$$p_i \text{ reads } f_j \Rightarrow C_{read} : p_i \succ f_j \quad (2)$$

$$p_i \text{ writes } f_j \Rightarrow C_{write} : p_i \prec f_j \quad (3)$$

$$p_i \text{ creates } f_j \Rightarrow C_{creat} : p_i \prec f_j \quad (4)$$

When there are multiple causal orders can be applied, the following order decides the final relationship between a file and a process in workflow.

$$C_{creat} > C_{write} > C_{read} \quad (5)$$

In practice, a single event can be extended to a series of continuous identical events without changing the order but significantly saving reasoning effort.

Therefore, by exploiting the causal orders from system call log, a complete workflow DAG $G_{workflow}$ can be created

using following definition.

$$G_{workflow} = (V, E)$$
$$V = \{p_i | p_i \text{ is the process spawned in application}\}$$
$$\cup \{f_j | f_j \text{ is the file accessed in application}\}$$
$$E = \{(v_i, v_j) | v_i, v_j \in V \text{ and If } v_i \prec v_j\}$$

Then, the attributes of nodes and edges are assigned with values that are extracted or calculated from tracing log according to different analysis purposes.

5.3.2 Manipulation and Analysis

Once the workflow DAG structure is created and annotated by required information, users are able to apply graph algorithms to analysis it. There are two typical manipulations to reduce the complexity of workflow analysis.

Subgraphing A subgraph of the entire workflow DAG can be extracted by selecting a subset of nodes for the detailed study of a sub-workflow. For example, the subgraph of a specific phase in a workflow consists of processes in that phase and files accessed by those processes.

Aggregation A fine-grained workflow DAG can be aggregated into a coarse-grained DAG to reduce the number of nodes and the number of edges. For example, a process-level workflow DAG can be aggregated to a job-level workflow DAG according to the job id of each process. Similarly, a host-level workflow DAG allows user quickly understand the data interaction or load-balance among different computing hosts.

We also list several graph attributes that can be used for graph-theoretic analysis.

Node weight Many types of values are available for different study interests. For example, the weight of node can represent the data volume passed from/to this node if users would like to know which process/file is I/O-intensive. Or the weight of a process node can be its life time and the weight of a file node can be its size in order to find the dominant job or data in workflow.

Node degree For a process node, its degree is a summation of the number of sub-processes it has spawned, and the number of files it has accessed. For a file node, its degree tells the number of different I/O operations conducted by different processes. Furthermore, the degree can be classified as either in-degree or out-degree. The node degree indicates the different roles and importance of a node in workflow.

Edge weight Similar to node weight, different analysis results can be conducted depending on the types of values mapped to the edge weight. If the edge weight is data transfer size, then the I/O-intensive processes can be found. If the edge weight is data transfer rate, then those edges with lower rate below average may indicate potential bottlenecks in workflow execution.

Path Each path in the workflow DAG corresponds to a real execution sequence. Critical execution paths of

workflow can be found. If the edge weight is defined as the elapsed time between different jobs, then the makespan of entire workflow or any sub-workflow can be estimated.

Clustering Clustering analysis can be used to identify the processing patterns in workflow. For example, jobs that spawn parallel processing processes can be detected and then scheduled to computing resources having high parallel processing power.

5.4 Usages of Profiles

The workflow profiles can help users either understand or tune their workflows in many ways.

Performance tuning Since our target applications are data-intensive, the performance of data access plays an important role in overall throughput of workflows. Using file system call and I/O profiles provides opportunities for users not only to understand the I/O behaviors of their applications, but also to guide the tuning of underlying I/O subsystem (e.g., a distributed file system or staging system).

Workflow visualization Visualization is the most straightforward and comprehensive way to let users understand the workflow. By using various visualization techniques [39–41], ParaTrac generates workflow DAG annotated with critical workflow information in various popular image formats (e.g., JPEG, PNG, PS, PDF, and SVG). In addition, ParaTrac also exports other graph exchange formats (e.g., DOT, GML, GraphML, and Cytoscape [40] description file) for other graph analysis library and interactive tools.

Workflow auto-generation Manually writing the DAG description file (e.g., DAX [6, 20] and DAGMan [4, 5]) for a complex workflow is non-trivial and error-prone for users. By utilizing profiled workflow DAG, an accurate workflow specification can be effortlessly generated for further workflow planning.

Workflow debugging A wrong workflow dependency can be found more easily from DAG than from runtime debugging. For example, the incorrect dependencies mistakenly stated by users in a makefile or workflow description file can be discovered from actually profiled workflow DAG.

Workflow comparison Since the profiles contains overall and detailed runtime information of a workflow, users are able to do head-to-head comparisons between workflows executed in different environments with realistic metrics. In addition, profiles of different workflows can be collected as a study repository for domain researchers.

6. EVALUATION

6.1 Experimental Settings

6.1.1 Environments

Our experimental environments consist of machines with homogeneous hardware and software configurations. Each machine has Intel Xeon E5410 8-Core 2.33GHz CPU, 32GB

Table 2: Montage data sets used in experiments

Data	Dimension	.fits Files	Size (MB)
A	640×480	6	12.08
B	2259×2199	47	90.79
C	10000 ²	609	1212.07

main memory, WD RE3 SATA 1TB hard disk. The unified software includes Linux kernel 2.6.26 with FUSE kernel interface 7.8, FUSE library 2.7.4, Python 2.5.2, GXP 3.06, and ParaTrac 0.4.

6.1.2 Application and Data

We used ParaTrac to trace and profile the Montage astronomy scientific application [42], a popular real-world benchmark that has been widely used in workflow studies [2, 6]. Table 2 shows the summary of the data sets processed by Montage workflow in experiments, which were retrieved from 2MASS repository [43].

6.1.3 Settings

We use `makefile` to describe the dependency of Montage workflow and utilize the GNU `make -j` option [44] to execute the workflow with specified concurrent jobs.

To seamlessly simulate the network latency introduced by staging or distributed file system, we used SSHFS [45] to mount a distant machine where the data are stored. The RTT (Round Trip Time) between local processing node and this remote data storage node is 24.412 msec and the data transfer rate using SSHFS is around 2.8MB/sec. By this approach, we are able to demonstrate the effect of network latency and bandwidth on profiles of applications.

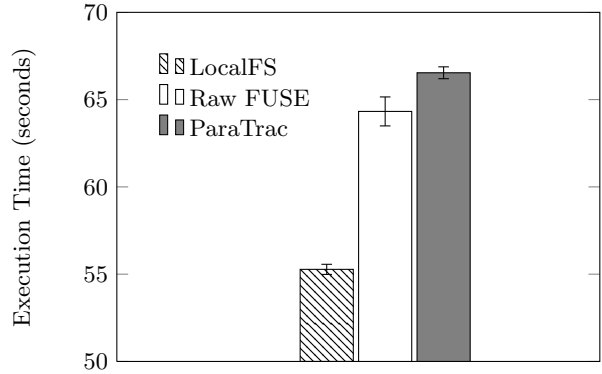
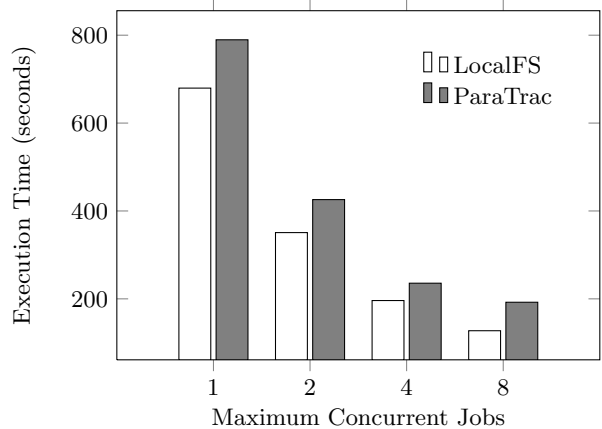
6.2 Tracing Overhead

6.2.1 Run-time Overhead

To investigate the tracing overhead, we compared the serial execution time of Montage workflow on three different file systems: the local ext3 file system, a raw FUSE mounted file system without tracing routines, and a ParaTrac mounted file system. The raw FUSE file system is created by the standard FUSE program (i.e., `fusexmp_fh.c` included in FUSE source [12]). This program is similar to the implementation in Figure 1 but without those tracing routines (i.e., lines with “+” mark). Comparing to this raw FUSE file system allows us to distinguish the inherited overhead by using FUSE from the overhead introduced by ParaTrac tracing routines.

On each file system, we performed 10 identical runs and calculated the mean of their execution time. As shown in Figure 3, ParaTrac causes about 16% more than the execution time on local file system. However, most of tracing overhead is due to the context switch cost in FUSE and the tracing routines take only a small fraction of total overhead. Though kernel-level file system tracing shows only about 6-7% overhead for I/O intensive applications [8, 10], the overhead of user-level tracing is acceptable with respect to its flexibility, portability, and implementation effort.

To investigate the tracing scalability of ParaTrac, we traced the executions of Montage workflow on data C with ascending number of concurrent jobs monitored by one tracer (i.e., via one ParaTrac mounted file system).

**Figure 3: Comparison of serial execution time****Figure 4: Scalability on concurrent jobs**

As shown in Figure 4, the tracing overhead remains 16%–21% when the number of concurrent jobs ranges from 1 to 4, and reaches 50% when the parallelism becomes 8. The tracing scalability of ParaTrac mainly depends on the capability of FUSE handling concurrent requests [46]. However in practice, ParaTrac is used to trace applications running on distributed nodes, which allows ParaTrac to scale with constant overhead on a large number of computing resources. Even for tracing multiple jobs on one single machine (e.g., machine with multi-cores), users can start multiple trackers for each job as a workaround.

Note that when profiling parallel application, concurrent writing trace logs to the shared file system where the workflow is executed might conduct non-trivial overhead to system performance. ParaTrac provides an option to allow the logs to be flushed to other separated file systems, such as ramfs or other disk devices.

6.2.2 Trace Log Size

The size of trace log primarily scales with the number of system calls events, and is secondarily related to the number of processes and files that are traced during the execution. Table 3 shows the data size of logs produced by profiling Montage execution on all three data sets. Comparing with the values in Table 2, we find that, for Montage workflow, the size of trace logs is approximately proportional (about 0.1 time) to the size of application data.

Table 3: Comparison of trace log size

Trace Log	Data		
	A	B	C
runtime.log	177B	171B	177B
file.log	3.8KB	25KB	303KB
proc.log	5.0KB	22.6KB	284KB
sysc.log	1.3MB	7.2MB	133MB
trace.db	930KB	10MB	97MB

Table 4: Comparison of system call statistics

System Calls	Count (times)		Latency (seconds)	
	sum	ratio	sum	ratio
<i>Data B</i>				
lstat	4292	0.0254	8.19	0.026
fstat	308	0.0018	3.16e-4	1.01e-06
access	21	1.24e-04	6.46e-05	2.07e-07
truncate	1	5.9e-06	4.58e-05	1.46e-07
opendir	3	1.77e-05	6.658e-05	2.13e-07
readdir	86	5.09e-04	0.1347	4.3e-04
closedir	3	1.77e-5	1.1e-05	1.1e-05
creat	308	0.0018	0.012	3.73e-05
open	2385	0.0141	90.86	0.2902
close	2693	0.0159	1.97	0.0063
read	15550	0.092	206.9	0.6611
write	140613	0.832	2.562	0.0081
flush	2819	0.0164	2.369	0.0076
Total	169082	1.0	313.04	1.0
<i>Data C</i>				
lstat	57393	0.0255	360.37	0.0484
fstat	3981	0.0018	0.0038	5.09e-07
access	21	9.34e-6	7.49e-05	1.0e-08
truncate	1	4.45e-7	4.71e-05	6.3e-09
opendir	3	1.33e-06	6.71e-05	9.0e-09
readdir	662	2.9e-04	0.6982	9.37e-05
closedir	3	1.33e-06	1.23e-05	1.65e-09
creat	3981	0.0018	0.2175	2.92e-05
open	31061	0.0138	2128.32	0.286
close	35042	0.0156	79.27	0.0106
read	222985	0.0992	4754.6	0.638
write	1856039	0.826	37.99	0.005
flush	36864	0.0164	90.37	0.012
Total	2246286	1.0	7451.84	1.0

6.3 Profiles of Workflow

6.3.1 File System Call Statistics

File system call statistics shown in Table 4 compare the overall behaviors of Montage workflow executions with 8 concurrent jobs using SSHFS for data B and C, respectively.

Referring to Table 2, we observe that the number of system calls scales proportionally to the data size, but the ratios of most dominant operations (e.g., `stat`, `read`, and `write`) remain constant. This fact suggests that I/O optimization (e.g., reduce the number of `write`) for Montage can be done on small-scale data first, then the same strategy will be also effective for large-scale data.

As for file system call latency, less than 10% `read` operations take over 60% of total latency while more than 80% `write` operations only cost less than 1% of total latency. This is because data was fetched first from remote host over the high-latency network link in our experimental configuration. But the processed data can be written to local cache and thus output operations have much lower latency. From the latency histogram of system calls shown

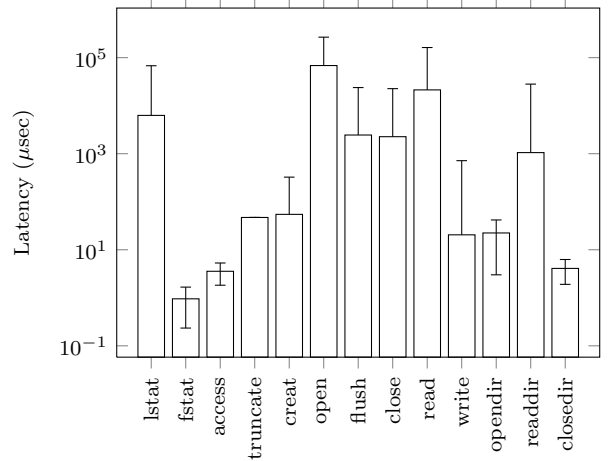


Figure 5: Variance of system call latency

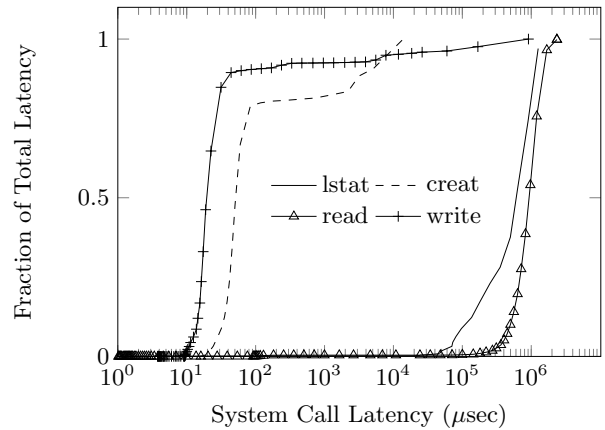


Figure 6: CDF of system call latency

in Figure 5, both high-latency and low-latency operations can be easily discovered.

To investigate the detailed behaviors of system calls, we used ParaTrac to generate CDF (Cumulative Distribution Function) of four common operations: `lstat`, `creat`, `read`, and `write`. Result in Figure 6 confirmed our observation. Most of `lstat` and `read` system calls are high-latency because initially they have to be manipulated at remote host. Instead, `creat` and `write` only touched local cache. It also suggest that users could optimize those high-latency file operations that take the major part of system calls to improve overall performance. For instance, using replication or pre-fetch will improve the read performance of Montage workflow in wide-area environments.

6.3.2 I/O Characteristics

The detailed I/O activities of processes for each phase in workflow are listed in Table 5.

Basically, I/O-intensive jobs in Montage workflow includes `mProjectPP`, `mDiffFit`, `mBackground`, `mAdd`, and `mShrink`, and the number of spawned processes in these phases increases with the size of initial input data. It is obvious that proper data or I/O optimization should be used in these phases. However, we argue that optimal data

Table 5: Comparison of I/O summary of Montage workflow on data A and B

Phase	Jobs	I/O Size (MB)		Number of Files				Number of Processes			
		read	write	ro	wo	rw	none	ro	wo	rw	none
<i>Data A</i>											
mProjectPP	28	57.77	221.07	30	0	56	0	1	0	28	0
mDiffFit	63	1023.87	25.32	58	126	63	0	64	63	63	0
mConcatFit	1	0.533	0.0127	61	1	0	0	1	0	1	0
mBgModel	1	0.0672	0.0144	3	1	0	0	1	0	1	0
mBackground	28	223.63	221.07	59	0	56	0	1	0	28	0
mImgtbl	1	0.644	0.0156	30	0	1	0	1	0	1	0
mAdd	1	133.93	75.81	59	2	0	0	1	0	1	0
mShrink	1	37.95	4.22	2	1	0	0	1	0	1	0
mJPEG	1	4.28	0.23	2	1	0	0	1	0	1	0
Total	125	1481.93	547.75	33	65	243	0	64	63	125	0
<i>Data B</i>											
mProjectPP	308	637.87	2472.25	310	2	614	0	1	0	308	0
mDiffFit	913	15114.33	675.64	618	1804	927	0	914	913	909	0
mConcatFit	1	0.34	0.18	915	1	0	0	1	0	1	0
mBgModel	1	0.314	0.015	3	1	0	0	1	0	1	0
mBackground	308	2524.48	2472.25	619	2	614	0	1	0	308	0
mAdd*	4	248.14	1587.56	613	8	0	0	1	0	4	0
mShrink	4	793.85	7.96	5	4	0	0	1	0	4	0
mImgtbl	1	0.085	0.0315	6	0	1	0	1	0	1	0
mAdd*	1	8.02	12.27	7	2	0	0	1	0	1	0
mJPEG	1	7.67	0.357	2	1	0	0	1	0	1	0
Total	1542	21568.07	7231.48	322	897	3084	0	914	913	1538	0

*mAdd is invoked twice in different phases.

ro: read only; wo: write only; rw: read+write; none: create/open only

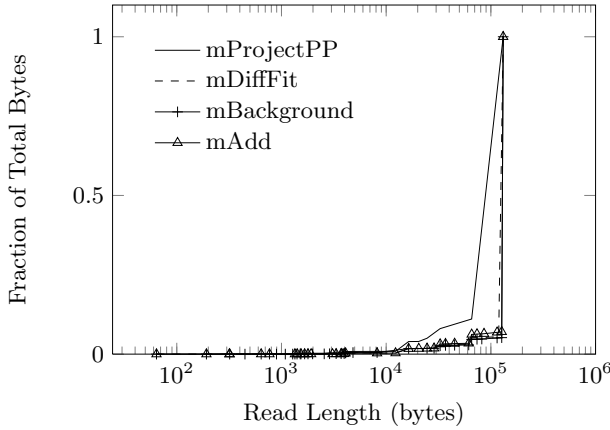


Figure 7: CDF of read length

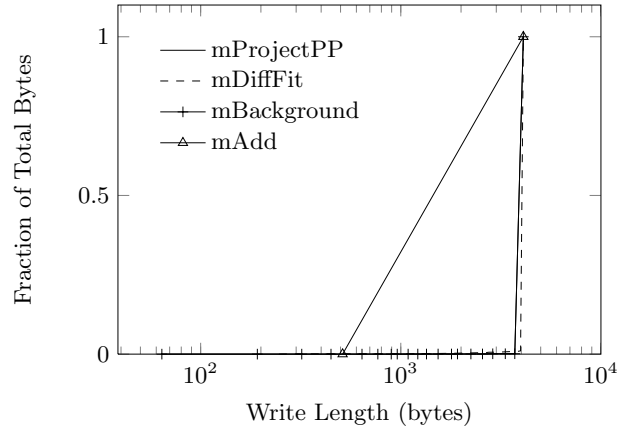


Figure 8: CDF of write length

scheduling strategies are possible to made for those I/O-intensive jobs. Taking mConcatFit as an example, it reads a small amount of data but from a wide range of files that are generated in mDiffFit phase. If mDiffFit and mConcatFit are executed in distributed computing resources in distance, then transferring specific data portion required by mConcatFit will be more efficient than replicating or prefetching entire input files from mDiffFit.

The I/O access patterns can be illustrated by CDFs of I/O request sizes. For file-specified I/O patterns, as shown in Figure 7 and Figure 8, mDiffFit, mBackground, and mAdd share the same read/write pattern. For process-specified I/O patterns, as shown in Figure 9 and Figure 10, processes of mDiffFit prefer large read size and relative small write size, while processes of mBackground request the same size both for read and write.

6.3.3 Process Profiles

The process tree shows the process hierarchy and processing patterns in Montage workflow. Figure 11 illustrates a example of process tree of the small-scale Montage workflow. As a pattern in the upper-right cluster of processes, the series of nodes with two child leaves are mDiffFit processes that spawn two subprocess: mDiff and mFitPlane.

Figure 12 shows the distribution of process life time of four data-intensive phases. Among them, mProjectPP have the longest running time than other three phases, while mDiffFit has the most number of processes but those processes are all short-life.

6.3.4 Workflow DAG

We use the intuitive workflow DAG chart of the Montage

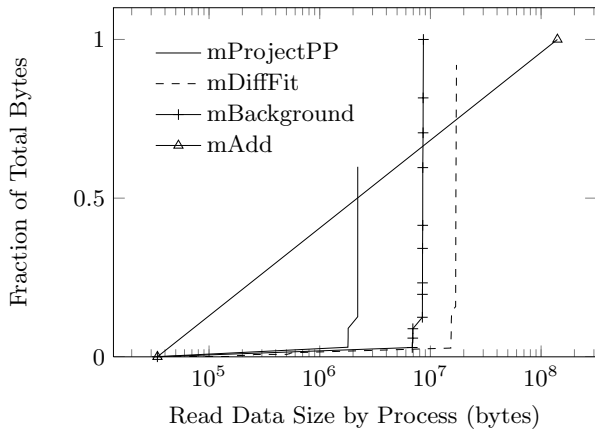


Figure 9: CDF of data size read by process

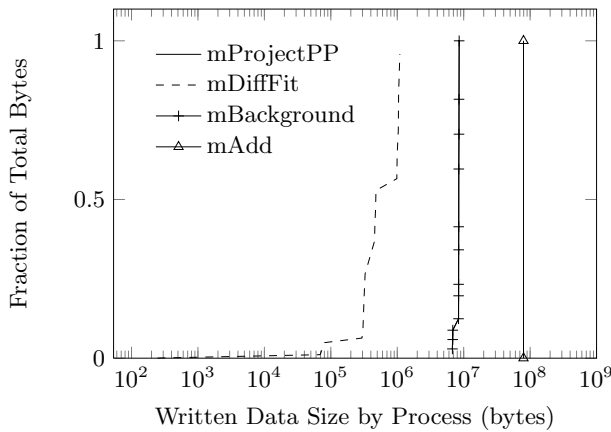


Figure 10: CDF of data size written by process

execution on data A as an example to illustrate the effectiveness of fine-grained workflow analysis. Besides the visualization of workflow DAG, the meta DAG information can be used as addressed in Section 5.4.

The complete workflow DAG is shown in Figure 13. For clear representation, an enlarged sub-workflow DAG is also given in Figure 14. The ellipse represents process node and the box denotes file node. The dependencies between nodes are drawn as edges, where dot-line edge is inter-process relationship and real-line edge is file-process relationship. Different arrowheads are used to distinguish different I/O operations. Nodes and edges are annotated with runtime information extracted from application profiles. The label of process node shows the *command line* that started the process and the *life time* of the process. The label of file node shows the *base filename* of the file. The label of process-file edge are annotated by the data size transferred between file and process and corresponding transfer rate.

From Figure 13, users are able to quickly obtain the entire structure of workflow, such as different phases in workflow, parallel jobs in each phase, input/output files in each job, and so on. One of critical paths (i.e., edges in red) in workflow is also automatically marked for users. Detailed data-process interactions are also intuitively illustrated. For example, `mProjectPP` processes initially fetch data (i.e., *.fits*

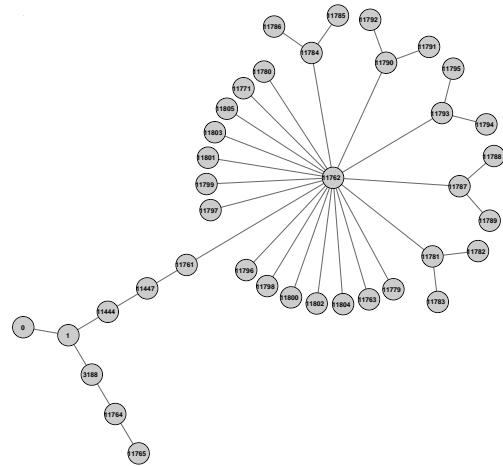


Figure 11: Process tree of Montage on data A

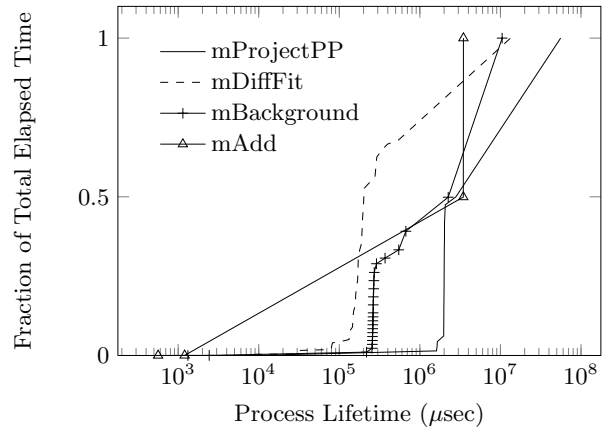


Figure 12: CDF of process lifetime on data B

files) from remote hosts with a low transfer rate (around 1.58MB/sec) and then subsequent data accesses rate become higher ($> 200\text{MB/sec}$) because of data are cached locally, by which we confirmed our observation found in Section 6.3.1. In practice, we can mapping the data transfer rates to the thickness of corresponding edges, thus those bottlenecks can be quickly and visually identified. For another example, as discussed in Section 6.3.2, `mConcatFit` only reading hundreds of bytes of data from several files is recognizable at one glance. By sub-graphing, we can also extract a sub-workflow DAG from the complete one. Figure 14 highlights a sub-processing flow in `mDiffFit` phase, in which the subprocesses (i.e., `mDiff` and `mFitPlane`) spawned by `mDiffFit` are expanded instead of being aggregated as in Figure 13.

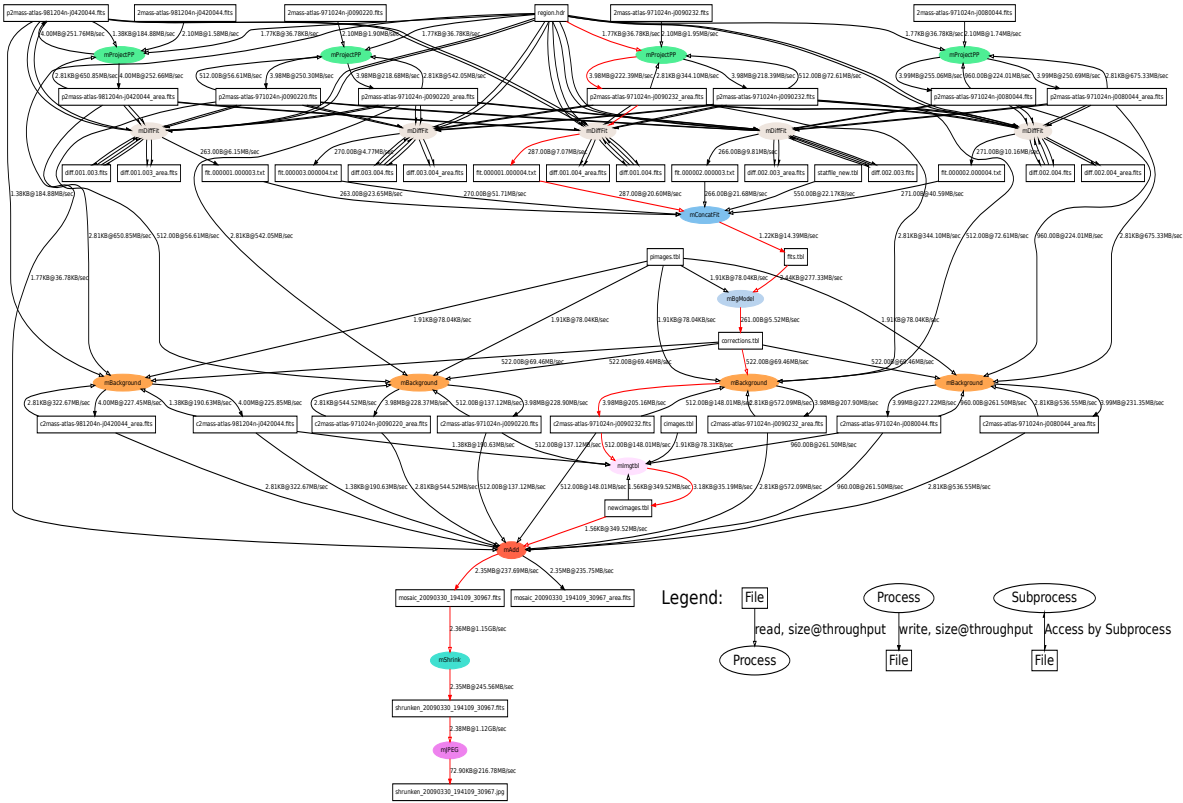


Figure 13: Complete workflow DAG of Montage on data A

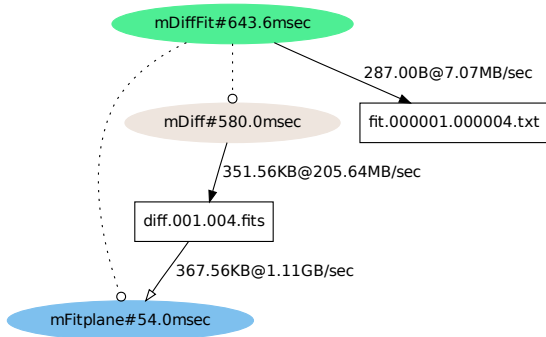


Figure 14: Sub-workflow of Montage on data A

7. CONCLUSIONS AND FUTURE WORK

In this paper, we described the design, implementation, and usages of ParaTrac — a fine-grained profiler targeted for data-intensive workflows. ParaTrac can automatically

generate comprehensive workflow profiles to help users understand the detailed behaviors of workflows execution, which implies the refinement of their orchestration. Since more realistic runtime information can be obtained by fine-grained profiling, we suggest that workflow management systems provide extra controls of specifying resource constraints to allow users to perform the fine-grained scheduling of workflows. For example, a workflow management system can utilize the data size and transfer rates in profiles to achieve a data throttling strategy [47] to improve the throughput of workflows.

In future, we plan to evaluate the realistic scheduling of workflow management systems for large-scale applications by using ParaTrac, and study the potential optimization brought by fine-grained execution profiles. Currently, we have also collected a number of real-world data-intensive applications and prepare to investigate them by ParaTrac. We also plan to extend ParaTrac to trace other workflow information, such as CPU time of tasks, to enrich the profiles of applications. Another direction is to reuse profiles as a macro benchmark for workflow management systems by consistent replaying of profiles.

Finally, ParaTrac is an open source software that is online available at <http://paratrac.googlecode.com/>.

8. ACKNOWLEDGMENTS

We would like to thank our colleagues for their suggestions on efficient implementation of ParaTrac. The authors would like to thank the referees for their efforts and comments. This research is supported in part by the MEXT Grant-in-Aid for Scientific Research on Priority Areas project “New IT Infrastructure for the Information-explosion Era” and Grant-in-Aid for Specially Promoted Research.

9. REFERENCES

- [1] I. Taylor, E. Deelman, D. B. Gannon, and M. S. (Eds), *Workflows for e-Science*. Springer-Verlag, 2006.
- [2] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.
- [3] Pegasus: Planning for execution in grids. [Online]. Available: <http://pegasus.isi.edu/>
- [4] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, “Workflow management in Condor,” in *Workflows for e-Science*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. Springer-Verlag, 2006, ch. 22, pp. 357–375.
- [5] DAGMan: Meta-scheduler for Condor. [Online]. Available: <http://www.cs.wisc.edu/condor/dagman/>
- [6] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, “Characterization of scientific workflows,” in *Proc. The 3rd Workshop on Workflows in Support of Large-Scale Science*, 2008, pp. 1–10.
- [7] L. Ramakrishnan and D. Ganno, “A survey of distributed workflow characteristics and resource requirements,” Indiana University, Tech. Rep. TR671.
- [8] L. Mummert and M. Satyanarayanan, “Long term distributed file reference tracing: Implementation and experience,” *Software — Practice & Experience*, vol. 26, no. 6, pp. 705–736, 1996.
- [9] D. R. Jacob, J. R. Lorch, and T. E. Anderson, “A comparison of file system workloads,” in *Proc. The 2000 USENIX Annual Technical Conference*, 2000, pp. 41–54.
- [10] A. Aranya, C. P. Wright, and E. Zadok, “Tracefs: A file system to trace them all,” in *Proc. The Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, Mar. 2004, pp. 129–143.
- [11] LoggedFS filesystem monitoring with FUSE. [Online]. Available: <http://loggedfs.sourceforge.net/>
- [12] M. Szeredi. FUSE: Filesystem in userspace. [Online]. Available: <http://fuse.sourceforge.net/>
- [13] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, “Measurements of a distributed file system,” in *Proc. The 13th ACM Symposium on Operating Systems Principles (SOSP ’91)*, vol. 25, no. 5, pp. 198–212.
- [14] B. K. Pasquale and G. C. Polyzos, “A static analysis of I/O characteristics of scientific applications in a production workload,” in *Proc. The 1993 ACM/IEEE Conference on Supercomputing*, New York, NY, 1993, pp. 388–397.
- [15] N. Nieuwejaar, D. Kotz, A. Purakayastha, and C. S. Ellis, “File-access characteristics of parallel scientific workloads,” *IEEE Transactions on Parallel and Distributed System*, vol. 7, no. 10, pp. 1075–1089, 1996.
- [16] A. Ching, A. Choudhary, W. keng Liao, L. Ward, and N. Pundit, “Evaluating i/o characteristics and methods for storing structured scientific data,” in *Proc. International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.
- [17] E. Deelman and A. Chervenak, “Data management challenges of data-intensive scientific workflow,” in *Proc. The 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid ’08)*, Lyon, France, May 2008.
- [18] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, “Examine the challenges of scientific workflows,” in *IEEE Computer*, vol. 40, no. 12, Lyon, France, Dec. 2007, pp. 24–32.
- [19] S. Pandey and R. Buyaa, “Scheduling and management techniques for data-intensive application workflows,” in *Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management*, T. Kosar, Ed. IGI Global, 2009.
- [20] Workflow generator. [Online]. Available: <http://vtcp.isi.edu/pegasus/index.php/WorkflowGenerator>
- [21] T. Shibata, S. Choi, and K. Taura, “File-access patterns of data-intensive workflow applications and their implications to distributed filesystems,” in *Proc. International Workshop on Data-Intensive Distributed Computing*, 2010.
- [22] O. Curran, P. Downes, J. Cunniffe, and A. Shearer, “Fine-grained workflow in heterogeneous environments,” in *Proc. 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2008, pp. 115–119.
- [23] chroot. Linux man page. [Online]. Available: <http://linux.die.net/man/2/chroot/>
- [24] fakechroot. Linux man page. [Online]. Available: <http://linux.die.net/man/1/fakechroot/>
- [25] fakeroot. Linux man page. [Online]. Available: <http://linux.die.net/man/1/fakeroot/>
- [26] CUSE: Character device in userspace. [Online]. Available: <http://lwn.net/Articles/308445/>
- [27] The /proc filesystem. Linux Kernel Documentation. [Online]. Available: <http://www.kernel.org/doc/Documentation/filesystems/proc.txt>
- [28] Per-task statistics interface. Linux Kernel Documentation. [Online]. Available: <http://www.kernel.org/doc/Documentation/accounting/taskstats.txt>
- [29] The struct taskstats. Linux Kernel Documentation. [Online]. Available: <http://www.kernel.org/doc/Documentation/accounting/taskstats-struct.txt>
- [30] getdelays.c. Linux Kernel Documentation. [Online]. Available: <http://www.kernel.org/doc/Documentation/accounting/getdelays.c>
- [31] MacFUSE FUSE for Mac OS X. [Online]. Available: <http://code.google.com/macfuse/>
- [32] E. Driscoll, J. Beavers, and H. Tokuda. FUSE-NT: Userspace file system for Windows NT. [Online].

- Available:
<http://pages.cs.wisc.edu/~driscoll/fuse-nt.pdf>
- [33] Installable file system kit. [Online]. Available:
<http://www.microsoft.com/whdc/devtools/ifskit/>
 - [34] D. Almeida, "Fifs: A framework for implementing user-mode file systems in windows nt," in *Proc. The 3rd USENIX Windows NT Symposium*, 1999.
 - [35] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. Wylie, "A platform for scalable parallel trace analysis," in *Proc. Workshop on State-of-the-Art in Scientific and Parallel Computing*, Umeå, Sweden, Jun. 2006.
 - [36] K. Taura, "GXP: An interactive shell for the grid environment," in *Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA 2004)*, Charlotte, NC, USA, Apr. 2004, pp. 59–67.
 - [37] GXP grid and cluster shell. [Online]. Available:
<http://gxp.sourceforge.net/>
 - [38] SQLite library. [Online]. Available:
<http://www.sqlite.org/>
 - [39] Matplotlib: Python 2D plotting library. [Online]. Available: <http://matplotlib.sourceforge.net/>
 - [40] Cytoscape: Network analysis and visualization platform. [Online]. Available:
<http://www.cytoscape.org/>
 - [41] The DOT language. [Online]. Available:
<http://www.graphviz.org/doc/info/lang.html>
 - [42] Montage: An astronomical image mosaic engine. [Online]. Available: <http://montage.ipac.caltech.edu/>
 - [43] Irsa - two mircon all sky survey (2mass). [Online]. Available:
<http://irsa.ipac.caltech.edu/Missions/2mass.html>
 - [44] GNU make manual. [Online]. Available: <http://www.gnu.org/software/make/manual/make.html>
 - [45] SSH filesystem. [Online]. Available:
<http://fuse.sourceforge.net/sshfs.html>
 - [46] FUSE performance. [Online]. Available: <http://old.nabble.com/fuse-performance-td18271595.html>
 - [47] S.-M. Park and M. Humphrey, "Data throttling for data-intensive workflows," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS 2008)*, 2008.