2015 IEEE International Conference on Cluster Computing

Practical Resource Monitoring for Robust High Throughput Computing

Gideon Juve*, Benjamin Tovar†, Rafael Ferreira da Silva*, Dariusz Król*‖, Douglas Thain†, Ewa Deelman*, William Allcock‡, Miron Livny§

* University of Southern California, Information Sciences Institute, Marina Del Rey, CA, USA
† University of Notre Dame, Notre Dame, IN, USA
‖ AGH University of Science and Technology, Department of Computer Science, Krakow, Poland
‡ Argonne National Laboratory
§ University of Wisconsin Madison, Madison, WI, USA

***Abstract*—Robust high throughput computing requires effective monitoring and enforcement of a variety of resources including CPU cores, memory, disk, and network traffic. Without effective monitoring and enforcement, it is easy to overload machines, causing failures and slowdowns, or underutilize machines, which results in wasted opportunities. This paper explores how to describe, measure, and enforce resources used by computational tasks. We focus on tasks running in distributed execution systems, in which a task requests the resources it needs, and the execution system ensures the availability of such resources. This presents two non-trivial problems: how to measure the resources consumed by a task, and how to monitor and report resource exhaustion in a robust and timely manner. For both of these tasks, operating systems have a variety of mechanisms with different degrees of availability, accuracy, overhead, and intrusiveness. We describe various forms of monitoring and the available mechanisms in contemporary operating systems. We then present two specific monitoring tools that choose different tradeoffs in overhead and accuracy, and evaluate them on a selection of benchmarks.**

***Index Terms*—High-Throughput Computing, Profiling, Monitoring.**

I. Introduction

High-throughput computing (HTC) applications seek to maximize the quantity of results produced over long time periods, such as months or years. Hosted computing infrastructures such as grids, and more recently clouds, have been widely used by the research community to address the needs of such applications [1]–[3]. These systems are becoming increasingly complex: where clusters were once typically single-core machines that ran single-process applications, they have become constellations of many-core machines that run many applications simultaneously. HTC applications are also becoming more complex. Individual tasks are often grouped into larger structures such as workflows [4], or MapReduce applications, which allow users to express multi-step computational tasks such as retrieving data from an instrument or database, running an analysis, and extracting statistics.

Efficient and robust resource provisioning and scheduling strategies are required to handle this category of applications. Scheduling and provisioning algorithms typically assume that resource usage information such as wall time, file size, and memory requirements, are all available in advance or can be reliably estimated [5]–[8], but in practice this information is rarely available. Without detailed and accurate resource information, it is virtually impossible to make even a simple decision such as how many tasks to run simultaneously on a single machine.

In this work, we aim to gather information about the resource usage of high-throughput scientific applications so that systems can make better scheduling and provisioning decisions, and thereby improve overall throughput. Our approach focuses on monitoring from the user perspective, which implies different mechanisms, resolution, and privileges than those available to a system administrator. We first collect resource usage data as applications are executed, and then use this historical data to develop models that can be used to estimate the resource usage of future executions of the application [9], [10]. These estimates can be used during provisioning to select appropriate resources for the application, in scheduling to ensure that sufficient resources are available and that resources are used efficiently, and at runtime to enforce limits on resource usage and to detect failures that are caused by overconsumption. Note that the monitoring is done at a task, and not at a system level, as a task may be misbehaving even if it does not have a noticeable impact on the host system.

Although there are many operating system monitoring and profiling mechanisms that can be used to collect resource usage information, there is no single mechanism that meets our needs. This is partially due to the diversity of available architectures and operating systems, but also due to the fact that many monitoring mechanisms were designed for entirely different purposes. For example, many operating systems provide whole-system summaries (such as the global load average) and per-device statistics (such as free blocks on a file system), but neither of these is appropriate for measuring the independent resources consumed by each job currently running on the machine. Collecting the desired information requires a combination of techniques and involves trade-offs between accuracy, overhead, and complexity. If resources are exhausted, it is important to know which process misbehaved. Therefore it is critical that the chosen mechanism can be

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.115

deployed for continuous use during production operation to make suitable resource allocation decisions.

In this paper, we survey the large number of mechanisms available in current operating systems for gathering resource information, and discuss the advantages and drawbacks of each approach for monitoring the resource usage of HTC workloads. We then select several mechanisms appropriate for continuous monitoring of production jobs, and evaluate tradeoffs in overhead and accuracy on a selection of benchmarks.

II. Monitoring Mechanisms

We define an HTC workload as a set of semi-independent tasks (or batch jobs), each of which involves the invocation of one or more operating system processes. In this paper, we focus entirely upon the design and implementation of a resource monitor which runs alongside each task in an HTC workload. Future work will address the other components of resource management, such as estimation, scheduling, and enforcement.

There are three general methods by which the monitor may attempt to learn about the resource usage of a process. It may be *interposed* between the process and the resource in order to learn precisely what the process is doing by intercepting its actions. It may *query* the properties of the resource that are tracked by the OS. Or it may request that the OS send *notifications* when the state of the resource changes.

There are inherent tradeoffs between each monitoring technique. Generally, interposition offers the greatest accuracy in determining the intent of a process, because it sees each operation before the OS has an opportunity to act upon it. This also permits the interposer to prevent an action before a resource is overconsumed. But, interposition often has significant overhead and complexity, and must be used carefully to avoid changing the behavior of the monitored process. Queries are usually the easiest mechanism to implement, but the information returned is immediately out of date. Repeated queries at a rapid rate will result in more timely information at the cost of increased overhead. However, reliance on queries can result in inaccurate monitoring: a temporary resource spike between measurements might be missed. Even if a query indicates overconsumption of a resource, the monitor can act to stop the process, but other processes on the machine may have failed as a result. Notifications from the operating system, when available, are more accurate than repeated queries because they rely upon the OS to detect key events and report them reliably, using much less traffic. However, like queries, they report upon completed events and do not permit the monitor to prevent problems before they happen.

Given these considerations, practical resource monitoring requires the use of all three techniques. Different resource types may call for different kinds of monitoring, and different OS facilities and operating conditions may call for different approaches. A task may also involve multiple processes arranged in a process tree. In such cases it is necessary for the resource monitor to track and measure the resource usage of all the processes and sum the results appropriately. In order to do this, the monitor needs to use mechanisms that enable it to observe the creation and termination of processes, and determine the relationships between them.

We wish to monitor resources in three general categories: computation, memory, and I/O. For each of these resources, it is important to capture the *behavior* of the process and the *resources* provided by the OS. Broadly, interposition methods capture behavior, while notifications and queries observe resources provided by the OS.

A. Query Mechanisms

There are many system calls and library functions that can be used to query resource usage. `getrusage()` is a standard UNIX system call that can be used to get information about the computation, memory and I/O of a task. Unfortunately, information available from `getrusage()` varies between implementations, as the POSIX standard only requires the report of CPU times (for example, Linux and Darwin populate the I/O fields differently). The `stat()` family of functions can be used to get information about file sizes. `statfs()` provides information about mounted file systems, such as the number of used and free inodes and blocks. This information could be used to estimate the amount of disk space used by a task, or to ensure that there is enough disk space available to run a task.

A common source of resource usage information is `procfs`. `procfs` is a virtual file system (typically mounted at `/proc`) that exports data about the state of the operating system, including system- and process-level information about memory, CPUs, disks, and file systems. The information available in `procfs` varies widely among UNIX systems, but on many systems there is a directory for each process with files for different types of information about the process. For example, Linux provides `/proc/[pid]/stat`, which contains CPU usage information (utime, stime) and current memory usage, `/proc/[pid]/status`, which contains information about peak memory usage, and `/proc/[pid]/io`, which contains information about the number of bytes read and written.

Hardware performance counters can provide information about the computation resources used by a process. These counters track the number of hardware operations performed by a CPU core in special-purpose registers. The types of counters available on different systems varies widely, but typically there are counters for cycles, instructions, floating-point operations, cache hits, cache misses, branches, loads, stores, and many other CPU operations. PAPI [11] is a popular library for querying performance counters, and the Linux `perf` tool [12] records performance counters at the process level.

For GPUs, while the interface may vary amongst vendors, most drivers provide a mechanism for inquiring about the utilization of GPU resources by a given process. For example, `nvmlDeviceGetAccountingStats`, included in NVIDIA's Management Library [13], provides utilization statistics, such as the number of threads, processor time, and memory consumed.

B. Notification Mechanisms

With notification mechanisms, the operating system delivers messages to the resource monitor when the state of a resource changes. A simple example is the `wait4()` system call found on most UNIX systems, which blocks the caller until one of its children exits and returns information about the resource usage of the exiting child (the same information as `getrusage()`).

Linux provides `inotify()` for monitoring file system events. The monitor registers to receive notification when files and directories are opened, closed, modified, deleted, or moved. Unfortunately, the events reported are not associated with a process ID, so it is difficult to use for monitoring the files accessed by a specific task, unless each task has a unique working directory or only one task is allowed to run at a time.

`ptrace()` is a UNIX system call that is used to implement debuggers. Linux provides an extension to `ptrace()` for observing process creation and exit events. This extension is useful for tracking the genealogy of a task's process tree, and, because `ptrace()` stops the traced process on `exit()`, for observing the final state of a process before it is cleaned up (e.g. peak memory usage from `procfs`).

`taskstats` [14] is a query/notification interface for collecting information about processes on Linux. It uses a netlink socket to deliver resource usage data for processes and threads from the kernel to the monitor. This data includes values returned by `getrusage()`, such as utime and stime, as well as information available in `procfs`, such as bytes read and written and peak memory usage. The monitor can use taskstats to query for data about all processes/threads, about a specific process/thread, or register to receive events whenever a process/thread exits.

Kernel probes are another category of notification mechanisms. Probes are implemented as tracing points in the kernel that can be turned on and off by the monitor [15], [16]. Probes are placed at key locations in the kernel where they can report useful information about the system. Events are reported to the monitor every time a kernel thread encounters a probe that the monitor is interested in. For example, probes in the kernel VFS layer can report information about operations on files and directories. DTrace [17] on Solaris and SystemTap [18] on Linux are similar approaches for using kernel probes. Both systems provide a scripting language that enables users to define actions to associate with different probes, such as incrementing a counter or printing information. Because they have access to sensitive information about the entire system, most kernel probe implementations require the monitor to have superuser privileges.

C. Interposition Mechanisms

These are mechanisms in which the monitor intercepts actions performed by the process. System call interposition is a commonly used technique where every system call made by a process is intercepted by the monitor. This enables the monitor to observe I/O and file access information by intercepting the system calls associated with those functions, such as `open`, `close`, `read` and `write`. System call interposition can be implemented using `ptrace()` with the PTRACE_SYSCALL flag. On other systems, system calls can be replaced with software breakpoints using `ptrace()` to achieve the same result. System call interposition usually has a very high overhead because `ptrace()` generates a signal to stop the traced process and extra context switches every time a system call enters or returns from the kernel.

In function interposition the monitor provides wrappers that replace and call original functions. These wrappers record information about the parameters and the results of wrapped functions. This can be achieved in a number of different ways. Compile-time techniques require the application code to be modified by either importing a header that redefines the wrapped functions, or by replacing all the function references in the program to be traced with the equivalent wrapped versions. Function interposition can also be performed at link time by telling the linker to consider the wrapper functions before the wrapped functions when resolving symbols. Care needs to be taken when defining the wrappers so that name collisions can be resolved, and the wrapper functions can still call the wrapped functions. This is usually accomplished by providing alternate names for the wrapped functions. For example, the MPI standard specifies an interposition mechanism for profiling MPI applications called PMPI that enables users to specify a profiling library at linking time to intercept MPI function calls (see Chapter 8 of [19]). The specification requires that all MPI implementations provide alternative names for MPI functions by prepending the letter "P" so that profiling libraries can provide their own implementation of the MPI interface, and call the implementation-specific functions, without causing a naming conflict. Alternatively, the GNU linker provides a `--wrap` option that allows arbitrary symbols to be wrapped.

Function interposition can also be implemented for shared libraries with help from the dynamic linker. In this approach, the LD_PRELOAD environment variable is used to tell the dynamic linker to use the symbols from the library with the wrapper functions in place of the symbols from the library with the wrapped functions. The wrapper library then uses `dlsym()` to locate and call the wrapped functions. This approach only works if the wrapped functions are in a shared library and if the program is not statically linked. Despite these limitations, function interposition is a powerful method to monitor vendor-specific devices. For example, it can be used to determine which GPUs are being accessed by the monitored process by preloading a wrapper for the `cuCtxCreate()` function from the NVIDIA's CUDA library.

Interposition can also be achieved at the file system level using a virtual file system that records information about I/O operations before passing them on to another file system that stores the actual data. This approach is used by ParaTrac [20], for example. `chroot` can be used to ensure that all I/O performed by the task passes through the virtual file system transparently. This approach is effective at capturing I/O and file accesses with low overhead, but requires superuser privileges to mount the file system and chroot the task.

Finally, dynamic binary instrumentation (DBI) is a technique for profiling applications that could be used for resource monitoring. In this approach, the monitor modifies the appli-

cation binary at runtime to insert profiling instructions and software breakpoints. Projects such as DynInst [21] and Intel PIN [22] provide libraries for writing monitors that use DBI.

D. Comparison of Mechanisms

Table I compares the various resource monitoring mechanisms described above based on several key characteristics. These characteristics include:

- **Mode** refers to the mode of operation, which is either query, notification, or interposition.
- **Resources** refers to the set of resources that can be monitored with a given mechanism. For example, `procfs` can provide information about the processes (P), threads (T), computation (C), memory (M), I/O (I) and files (F) associated with a task.
- **Effort** refers to the relative amount of work required to use the resource monitoring mechanism. This ranges from simply calling a system call or opening a file, to writing hundreds of lines of intricate code.
- **Overhead** refers to the level of performance degradation imposed on the task when using a given resource monitoring mechanism. This varies from none in the case of `wait4()`, to high in the case of system call interposition.
- **Portability** refers to the availability of the resource monitoring mechanism across different operating systems. Some mechanisms are standard features on all POSIX systems, like `wait4()`, others are available in only one operating system, like `taskstats` on Linux.
- **Privileges** refers to the level of security permissions required to use the mechanism. Some mechanisms can only be used by superusers, some mechanisms can be used by superusers or the owners of a process, and others can be used by any user on the system.
- **Intrusiveness** refers to the degree to which the mechanism interferes with the normal behavior of the task. Some mechanisms, such as interposition, are highly intrusive, while others, such as `stat()` have little or no impact on the task.
- **Scope** refers to the set of objects targeted by a monitoring mechanism. This can range from the whole operating system in the case of kernel probes, to individual files and directories for `inotify()`.
- **Notes** refers to the significant limitations of a mechanism. For example, systems differ in how they populate the fields returned by `getrusage()`.

III. Monitoring Tools

In this section, we describe the implementation of two tools we have developed for monitoring the resource usage of HTC tasks: resource_monitor, which is part of CCTools [23], and Kickstart [24], which is part of the Pegasus Workflow Management System [25].

A. Levels of Measurement

When implementing monitoring mechanisms, we found it helpful to establish *levels* of monitoring. These levels describe how intrusive a tool is when monitoring a task. The levels we defined are:

Level 1: Only query mechanisms and low overhead, non-intrusive notifications such as `wait4()` are used. Since there is no general method for obtaining the full process tree in level 1 (in Linux, one could periodically inspect the contents of `procfs`, but this is error prone as it would miss short running processes), this level is mostly useful for processes that do not fork.

Level 2: Interpositions and events are used for detecting when processes start and stop. By interposing, for example, `fork` and `exit` calls, the process tree can be easily observed. Once the process tree is known it is possible to record CPU times, virtual, resident, and swap memory, and bytes written and read by inspecting sources such as `procfs` for each process.

Level 3: Full system call or function interposition is used. By capturing `open`, `read` and `write` calls, this level provides the most precise measurements for files accessed and I/O.

B. Kickstart

Kickstart [24] is used to launch computing tasks, monitor the behavior of tasks, and report information about tasks and the hosts on which they were executed. Kickstart was originally designed to be used with Pegasus, but it can also be used separately. Kickstart implements all three monitoring levels, with level 1 being the default, and levels 2 and 3 enabled via command-line flags.

For all levels Kickstart uses `procfs` and other query mechanisms to gather basic information about the host, such as the number of CPUs and CPU cores, the amount of used and free system memory, the number of running tasks, the system uptime, and the hostname. It uses `wait4()` to obtain CPU usage (utime and stime), and `gettimeofday()` to compute the wall time of the task. In addition, Kickstart can optionally use `stat()` and a list of the task's input and output files to infer the amount of I/O performed by the task.

Kickstart implements monitoring levels 2 and 3 using `ptrace()`. `ptrace()` is particularly useful for level 3 because, unlike other interposition mechanisms, it does not require the application code to be recompiled, and it works on all binaries regardless of whether they are statically or dynamically linked.

For level 2, Kickstart uses `ptrace()` to intercept only process creation (`fork()`, `vfork()`, `clone()`, `exec()`) and `exit()` events. When a process exit event occurs, it inspects `/proc/[pid]/status` to determine peak memory usage (VM and RSS high water marks) and total I/O (bytes read and written). The process creation events are used to track new processes created by the task, and the exit events are used to observe the state of the processes when they have finished executing their computations, but before they have fully exited. This latter capability is critical for capturing accurate final statistics for the process in procfs. If Kickstart attempts to check procfs after `wait4()` returns, then the process will no longer exist under `/proc/[pid]`. If Kickstart checks before

TABLE I: Comparison of Resource Monitoring Mechanisms

Mechanism	Resources[a]	Effort	Portability	Overhead	Privileges	Intrusiveness	Scope	Notes
Query mechanisms								
perf. counters	C	Low	All	Low	Owner	Low	Process	
procfs	C,F,M,I,T,P	Low	UNIX	Low	Varies	Low	System, Process	[b]
stat()	F	Low	POSIX	Low	Any	Low	File	
statfs()	F,I	Low	UNIX	Low	Any	Low	File System	
getrusage()	C,M,I	Low	POSIX	Low	Owner	Low	Process	Some systems do not populate memory and/or I/O
GPU Libraries	C	Medium	Linux	Low	Any	Low	Process	[c]
Notification mechanisms								
taskstats	C,M,I,T,P	Low	Linux	Low	Owner	Low	Process	
ptrace() events	T,P	Medium	Linux	Low	Owner	Medium	Process	
inotify()	F	Low	Linux	Low	Any	Low	File, Directory	Does not associate events with processes
wait4()	C,M,I	Low	UNIX	Low	Owner	Low	Process	Some systems do not populate memory and/or I/O
Interposition mechanisms								
sys call interp.	F,I,T,P	High	Linux	High	Owner	High	Process	
function interp.	F,I,M,T,P	Medium	All	Low	Developer	High	Process	Requires re-compiling or re-linking
LD_PRELOAD	F,I,M,T,P	High	UNIX	Medium	Owner	High	Process	Only works for dynamic libraries
virtual filesystem	F,I	High	All	Medium	Superuser	Low	File System	
kernel probes	C,F,M,I,T,P	Medium	UNIX	Low	Superuser	Low	System	
DBI	C,F,M,I,T,P	High	All	Medium	Owner	High	Process	

[a] P: processes, T: threads, C: computation, M: memory, I: I/O, F: files

[b] Some information is only accessible by owner and superuser. Availability of data varies among UNIX systems.

[c] With supervisor privilege scope is expanded to System, Process

the process calls `exit()`, then procfs may not reflect the final peak memory and total I/O of the process. By capturing the exit event with `ptrace()`, Kickstart can ensure that the process is finished, but that it still exists in procfs. This approach provides accurate memory and I/O measurements without adding a significant amount of overhead.

For level 3, Kickstart uses `ptrace()` to gather detailed information about the files accessed by a process and the I/O performed on those files. In this mode, Kickstart interposes system calls, and inspects the arguments and return values for I/O system calls such as `open()`, `close()`, `read()`, `write()` and others. In this way it can keep track of exactly which files are opened by the task, and exactly how much I/O is performed on each one. In addition, it can observe I/O performed on terminals, sockets, FIFOs, and pipes. This mode provides more accurate and detailed file and I/O information than the previous list-of-files approach, but adds some overhead in the form of extra context switches on each system call performed by the task.

Kickstart also implements level 3 using the `LD_PRELOAD` mechanism. In this mode, LD_PRELOAD is used to tell the dynamic linker to load a custom library into the application that wraps key function calls such as `open()`, `close()`, `read()` and `write()`. The library writes monitoring data to a log file, which is read and analyzed by the monitoring process. The LD_PRELOAD approach has much less overhead than `ptrace()` for interposing function calls, but does not work on static binaries. In the case of static binaries, however, the library used for LD_PRELOAD can also be statically linked into the application at compile time.

C. resource_monitor

resource_monitor implements monitoring levels 1 and 2. For level 1 it continuously polls different query mechanisms, such as `procfs` on Linux, and the kernel `kvm` interface on FreeBSD. The `getrusage()` system call is used to get CPU usage information such as user and system time, and the peak resident memory size. Additionally, it uses calls from `fts.h` to periodically record the total size and file count of the working directory. This can be made more precise by providing the monitor with a list of directories and files to watch. Level 1 is less intrusive and has lower overhead than Level 2, but results in reduced accuracy as as shown in Section IV. This is because polling causes the monitor to miss peak usage values. In general, the longer the polling interval the less accurate the monitor will be in level 1.

For level 2 resource_monitor uses the LD_PRELOAD mechanism to interpose process management functions such as fork, exit and wait. LD_PRELOAD was chosen because it has less overhead than `ptrace()`, and requires less effort to implement. However, LD_PRELOAD does not work for binaries that have been statically linked. Special care is required using LD_PRELOAD to synchronize the monitor with fork/exit events from the process tree. Ideally, the monitor should measure peak resource usage values just before the task exits; otherwise, when the monitor finally detects that a task has terminated, its information is not available in the kernel anymore. To enable this, if the task was compiled with `gcc`, then the monitor also interposes the destructor attribute, which allows to detect when a process's `main()` completes or `exit()` is called. In addition, in level 2 resource_monitor uses `inotify()` to record which files are accessed by the task, when it is available.

By default, resource_monitor generates up to three report files: a summary file with the maximum values of resource used, a time-series that shows the resources used at periodic time intervals, and a list of files that were opened during execution. resource_monitor can be used as a watchdog by specifying maximum resource limits; when one of the resources goes over the limits specified, the task is terminated, and a report in the summary is made to indicate that the resource was exhausted.

IV. Evaluation

In this section we evaluate our implementation of the different monitoring levels described in Section III-A. For level 1, we use resource_monitor using only queries with a

sample polling period of 1 second. For level 2, we capture fork/exit events using LD_PRELOAD with resource_monitor and using ptrace events with Kickstart. Finally, for level 3, we use Kickstart to interpose system calls using ptrace and library calls using LD_PRELOAD. We first evaluate the accuracy of the tools for measuring CPU, memory, and I/O consumption on mock processes, and then we evaluate their performance impact (overhead). The experiments were conducted on a 12-core Intel Xeon 2.67GHz with 40GB of RAM. For each configuration, 5 repetitions were performed, which were sufficient to obtain average values with less than 2% error.

A. Accuracy

Table II shows accuracy results for CPU, memory, and I/O. To evaluate CPU accuracy, we developed a program to repeatedly compute the sine and cosine of random numbers. We varied the computation size from a million (10^6) up to a billion (10^9) instructions. Table II(a) shows the average CPU time (utime+stime) for each configuration when executed without monitoring (Baseline), and the error ratios of the times reported by the monitoring tools when compared to the baseline values. Positive error ratios (resp. negative) indicate that the monitoring tool overestimates (resp. underestimates) the CPU time. In all cases, the error ratios are positive, which suggests that the monitors are causing the monitored process to use more CPU time to do its job, possibly due to cache interference or context switches. In general, the resource_monitor seems to have more of an impact than Kickstart, probably because polling introduces more overhead than the mechanisms used by Kickstart when there are few function calls.

To measure the accuracy of memory monitoring, we developed a program that allocates 16GB of memory and fills between 1GB and 16GB of it with data. We expect the measurement reported by each tool to be equal to the amount of memory filled with data. Table II(b) shows the average error ratios of memory consumption value reported by the monitoring tools. The values measured by both Kickstart and resource_monitor are reasonably accurate in all cases except for the polling case. The relatively large errors for the polling case are all underestimates, and reflect the fact that the polling approach is not able to detect memory usage peaks because the process exits before the final value can be measured. This is a fundamental limitation of the polling approach.

Finally, I/O accuracy was determined using two different experiments: 1) fixing the buffer size and varying the amount of data read and written, and 2) fixing the file size and varying the buffer size.

For the first I/O experiment, we developed a program to read and write 1MB, 100MB, 1GB, and 10GB of data using a 4KB buffer size. Table II(c) shows the average error ratios for bytes read reported by the monitoring tools. Note that the error values for bytes written were similar to the values for bytes read, so they have been omitted. Again, the values reported by both tools are accurate with the exception of the polling case, which systematically underestimates the amount of data read and written because it is unable to record a measurement right before the process exits.

For the second I/O experiment, we developed a program to read and write 10GB of data using buffer sizes ranging from 4KB to 32KB. Average error ratios for bytes read are shown in Table II(d). Again, the results for writes were similar to the results for reads, so they have been omitted. Like the previous experiment, the values measured by using the monitoring tools are precise with the exception of the polling case.

B. Overhead

The overhead of the monitoring tools was measured for the same experiments described in the previous section. Table III(a) shows the CPU overhead in seconds, and the percentage overhead in brackets. A relatively large overhead is observed for very short executions, but as the execution time increases, the overhead becomes less than 1%. This suggests a small, approximately constant overhead for all the tools.

The performance impact of memory monitoring is shown in Table III(b). Similar to the CPU experiments, there is a relatively large overhead for short-running experiments, but in most cases the overhead is less than 5%. There does not seem to be any correlation between memory size and overhead, which is what we would expect. There also does not seem to be much difference in the overhead between different tools.

The I/O overhead experiments were the same as the previous section, except the read experiments used `/dev/zero` as the input file, and the write experiments used `/dev/null` as the output file. This was done in order to isolate the monitoring overhead and minimize the effects of file system caching and disk performance variation.

The different approaches have a more significant impact on performance in the case of I/O monitoring. For the first I/O experiment (variation of the data size) shown in Table III(c), the average overhead is high for small amounts of data, but this is just a result of the very short runtime of the program. As the amount of read/written data increases, the overhead ratio progressively decreases. In 4 out of 5 of the configurations tested, the overhead is only a few hundredths of a second for all the data sizes considered. In the case where Kickstart interposes system calls with `ptrace()` the overhead increases as the data size increases. This is a result of the increasing number of `read()` calls that are interposed as the data size increases (with constant 4KB buffer size), which each impose additional overhead on the process. However, the Kickstart results for LD_PRELOAD interposition show that it has a negligible overhead, even though it is interposing as many function calls as `ptrace`.

For the second I/O experiment, shown in Table III(d), when the size of the buffer is varied from 4KB to 32KB the number of function calls is significantly reduced and, consequently, the overhead of the `ptrace()` interposition case is also reduced. In the other cases, the overhead remains relatively low.

V. Related Work

There is a large number of system monitoring tools that use query and event-based mechanisms. Included among these are common system monitoring tools such as `top`, `ps`, `free` and the `Sysstat` suite [26], which includes `sar` and other tools.

TABLE II: Monitoring Accuracy

	Baseline	Polling (resource_monitor)		fork/exit LD_PRELOAD (resource_monitor)		library call LD_PRELOAD (Kickstart)		fork/exit ptrace (Kickstart)		system call ptrace (Kickstart)	
Instr.	**(a) CPU time**										
10^6	0.32 s	+0.04	(12.50%)	+0.02	(4.91%)	+0.01	(2.17%)	0.00	(0.00%)	0.00	(0.00%)
10^7	2.93 s	+0.06	(2.12%)	+0.04	(1.20%)	+0.01	(0.63%)	0.00	(0.00%)	+0.01	(0.14%)
10^8	28.20 s	+0.17	(0.60%)	+0.09	(0.31%)	+0.07	(0.24%)	+0.03	(0.10%)	+0.04	(0.14%)
10^9	279.53 s	+1.29	(0.46%)	+1.32	(0.47%)	+0.57	(0.19%)	+0.20	(0.07%)	+0.41	(0.15%)
Memory	**(b) Memory: resident size**										
1GB	1GB		−13.96%		+0.08%		+0.06%		+0.03%		+0.03%
2GB	2GB		−17.63%		+0.03%		+0.03%		+0.02%		+0.02%
4GB	4GB		−2.25%		+0.02%		+0.01%		0.00%		0.00%
8GB	8GB		−1.89%		+0.01%		0.00%		0.00%		0.00%
16GB	16GB		−1.99%		+0.01%		0.00%		0.00%		0.00%
File size	**(c) I/O: bytes read, 4KB buffer**										
1MB	1MB		−13.64%		0.00%		0.00%		0.00%		0.00%
100MB	100MB		−9.07%		0.00%		0.00%		0.00%		0.00%
1GB	1GB		−5.84%		0.00%		0.00%		0.00%		0.00%
10GB	10GB		−2.13%		0.00%		0.00%		0.00%		0.00%
Buffer size	**(d) I/O: bytes read, 10GB file**										
4KB	10GB		−5.42%		0.00%		0.00%		0.00%		0.00%
8KB	10GB		−0.91%		0.00%		0.00%		0.00%		0.00%
16KB	10GB		−13.12%		0.00%		0.00%		0.00%		0.00%
32KB	10GB		−16.87%		0.00%		0.00%		0.00%		0.00%

TABLE III: Monitoring Overhead

	Baseline	Polling (resource_monitor)		fork/exit LD_PRELOAD (resource_monitor)		library call LD_PRELOAD (Kickstart)		fork/exit ptrace (Kickstart)		system call ptrace (Kickstart)	
Instr.		**(a) CPU overhead**									
10^6	0.32 s	+0.22	(68.75%)	+0.25	(78.13%)	+0.04	(12.96%)	+0.18	(56.25%)	+0.13	(40.63%)
10^7	2.93 s	+0.28	(9.56%)	+2.42	(82.59%)	+0.03	(1.09%)	+0.14	(4.78%)	+0.14	(4.78%)
10^8	28.20 s	+0.17	(0.60%)	+0.22	(0.78%)	+0.04	(0.16%)	+0.10	(0.35%)	+0.12	(0.43%)
10^9	279.53 s	+0.28	(0.10%)	+0.78	(0.28%)	+0.02	(0.01%)	+0.07	(0.03%)	+0.61	(0.22%)
Resident size		**(b) Memory overhead**									
1GB	3.05 s	+0.31	(10.16%)	+0.34	(11.15%)	+0.22	(7.28%)	+0.11	(3.48%)	+0.43	(13.97%)
2GB	6.11 s	+0.11	(1.77%)	+0.17	(2.85%)	+0.04	(0.69%)	+0.03	(0.46%)	+0.06	(1.05%)
4GB	12.76 s	+0.81	(6.36%)	+0.67	(5.28%)	+0.51	(4.03%)	+0.80	(6.25%)	+1.04	(8.18%)
8GB	26.18 s	+0.70	(2.66%)	+1.09	(4.16%)	+1.35	(5.16%)	+0.31	(1.18%)	+1.34	(5.10%)
16GB	54.43 s	+0.07	(0.13%)	+1.63	(2.99%)	+0.25	(0.46%)	+0.26	(0.47%)	+1.75	(3.22%)
File size		**(c) I/O overhead, 4KB buffer**									
1MB	0.01 s	+0.01	(42.86%)	+0.01	(42.88%)	+0.02	(185.71%)	+0.01	(71.43%)	+0.02	(197.14%)
100MB	0.02 s	+0.01	(50.00%)	+0.01	(83.75%)	+0.02	(105.00%)	+0.02	(130.00%)	+0.45	(2812.50%)
1GB	0.13 s	+0.01	(1.22%)	+0.02	(11.89%)	+0.02	(17.07%)	+0.02	(17.07%)	+4.51	(3437.80%)
10GB	1.26 s	+0.01	(0.32%)	+0.02	(1.78%)	+0.03	(2.35%)	+0.03	(2.10%)	+44.67	(3545.24%)
Buffer size		**(d) I/O overhead, 10GB file**									
4KB	1.29 s	+0.01	(0.69%)	+0.01	(0.56%)	+0.02	(1.63%)	+0.01	(0.75%)	+45.59	(3570.33%)
8KB	1.19 s	+0.01	(0.51%)	+0.01	(0.54%)	+0.01	(0.88%)	+0.01	(0.74%)	+22.94	(1935.17%)
16KB	1.12 s	+0.01	(0.81%)	+0.01	(1.15%)	+0.03	(2.51%)	+0.02	(1.50%)	+12.08	(1081.81%)
32KB	1.10 s	+0.02	(1.46%)	+0.02	(1.35%)	+0.02	(2.26%)	+0.02	(1.83%)	+6.42	(585.98%)

Many distributed monitoring systems, including Ganglia [27], Nagios [28], and Munin [29], have been developed to provide system-level monitoring information. These systems are typically used by system administrators for problem detection and troubleshooting. They do not record the detailed, job- or process-level resource usage data that is required to model the resource usage of batch workloads.

Some monitoring tools have been developed for profiling the resource usage of HPC workloads. TACC_Stats [30] collects resource usage information including CPU usage, memory usage, filesystem and network I/O, and hardware performance counters. These values are recorded as a time series from `procfs`, `sysfs` and other sources. The data is correlated with individual jobs for later analysis based on job ID. NCAR has used a similar approach for monitoring CPU usage and floating point operations for HPC jobs [31].

There are several tools that use interposition to collect information about program behavior. The `strace` [32] and `ltrace` [33] tools use interposition to report system calls and library calls, respectively. `LANL-Trace` [34] uses these tools to profile the I/O behavior of parallel applications. `ParaTrac` [20] interposes I/O operations using a FUSE [35] filesystem that records information about I/O operations before passing them on to an underlying filesystem that stores the actual data. The system uses `chroot` to ensure that all application I/O passes through the profiling filesystem transparently. ParaTrac also collects information from `procfs`, `taskstats` [14] and the workflow management system to provide complete application profiles.

Many MPI profiling libraries that use PMPI for function interposition, including Jumpshot [36], mpiP [37], FPMPI [38], Scalasca [39] and others. Function interposition is used by several tools to implement I/O profiling. Darshan [40] uses PMPI and other function call interposition techniques to observe the I/O behavior of MPI applications. IOT [41] uses both PMPI and a GNU linker extension that enables functions to be wrapped at link time to enable I/O tracing. HPCT-IO [42] interposes UNIX I/O calls by either requiring applications

to include a header file that redefines the I/O functions and redirects them to a tracing library, or by using dynamic binary instrumentation to replace the I/O function calls in the application binary. Condor uses link-time interposition for implementing checkpointing and remote I/O for HTC jobs [43]

VI. Conclusion and Future Work

In this paper we presented a study of resource usage monitoring techniques for HTC workloads. We defined several categories of resource usage that are of interest for workload management and planning, including CPU usage, memory usage, storage, and I/O.

Many different mechanisms are available for measuring these resources, but there is a large number of challenges and tradeoffs that need to be considered when using these mechanisms for monitoring. In order to better understand these issues, we grouped the mechanisms into three general categories based on their method of operation (queries, notifications, interpositions), and compared the available mechanisms across a wide range of different characteristics, including portability, intrusiveness, performance impact, level of effort, accuracy, and others. Finally, we described the implementation of two monitoring tools that use several different monitoring mechanisms, and presented an evaluation of the accuracy and overhead of these tools using benchmark applications.

In the future we plan to deploy our monitoring tools on production infrastructure to collect resource usage data for real science applications. This data will help us extend our previous work [9], [10] on using historical resource usage data to automatically construct resource usage models for applications. These models can be used to derive estimates of future resource usage, which we plan to use to guide scheduling and provisioning algorithms, and to detect unexpected behavior and set limits for resource usage at runtime.

Acknowledgments

This work was funded by DOE under the contract number ER26110, "dV/dt - Accelerating the Rate of Progress Towards Extreme Scale Collaborative Science", and contract #DE-SC0012636, "Panorama - Predictive Modeling and Diagnostic Monitoring of Extreme Science Workflows".

References

[1] R. Sobie *et al.*, "Htc scientific computing in a distributed cloud environment," in *4th ACM Workshop on Scientific Cloud Computing*, 2013.
[2] C. Hoffa *et al.*, "On the use of cloud computing for scientific workflows," in *3rd International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES '08)*, 2008.
[3] "Open science grid," http://opensciencegrid.org.
[4] I. Taylor *et al.*, *Workflows for e-Science: Scientific Workflows for Grids*, 2007.
[5] T. D. Braun *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, jun 2001.
[6] H. Casanova *et al.*, "Heuristics for scheduling parameter sweep applications in grid environments," in *9th Heterogeneous Computing Workshop*, 2000.
[7] J. Blythe *et al.*, "Task scheduling strategies for workflow-based applications in grids," in *5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, may 2005.
[8] A. Mandal *et al.*, "Scheduling strategies for mapping application workflows onto the grid," in *14th IEEE International Symposium on High Performance Distributed Computing*, 2005.
[9] R. Ferreira da Silva *et al.*, "Toward fine-grained online task characteristics estimation in scientific workflows," in *8th Workshop on Workflows in Support of Large-Scale Science*, 2013.
[10] ——, "Online task resource consumption prediction for scientific workflows," *Parallel Processing Letters*, vol. accepted, 2015.
[11] "Performance application programming interface (papi)," http://icl.cs.utk.edu/papi.
[12] "perf," http://perf.wiki.kernel.org.
[13] "Nvml," https://developer.nvidia.com/nvidia-management-library-NVML.
[14] "taskstats," http://www.kernel.org/doc/Documentation/accounting/taskstats.txt.
[15] A. Mavinakayanahalli *et al.*, "Probing the guts of kprobes," in *Proceedings of the Ottawa Linux Symposium*, 2006.
[16] J. Keniston *et al.*, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *Ottowa Linux Symposium*, 2007.
[17] "Dtrace," http://dtrace.org.
[18] "Systemtap," https://sourceware.org/systemtap.
[19] M. P. I. Forum, "Mpi: A message-passing interface standard," 2003.
[20] N. Dun *et al.*, "Paratrac: A fine-grained profiler for data-intensive workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.
[21] "Dyninst," http://www.dyninst.org.
[22] "Intel pin," http://software.intel.com/en-us/articles/pintool.
[23] "Cctools," http://www3.nd.edu/ ccl/software/download.
[24] E. Deelman *et al.*, "Kickstarting remote applications," in *International Workshop on Grid Computing Environments*, 2006.
[25] ——, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, no. 0, pp. 17–35, 2015.
[26] S. Godard, "Sysstat," http://sebastien.godard.pagesperso-orange.fr.
[27] M. L. Massie *et al.*, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, jul 2004.
[28] "Nagios," http://nagios.org.
[29] "Munin," http://munin-monitoring.org.
[30] C.-D. Lu *et al.*, "Comprehensive job level resource usage measurement and analysis for xsede hpc systems," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE)*, 2013.
[31] D. D. Vento *et al.*, "System-level monitoring of floating-point performance to improve effective system utilization," in *Supercomputing*, 2011.
[32] "strace," http://sourceforge.net/projects/strace.
[33] "ltrace," http://ltrace.org.
[34] "Lanl-trace," http://institute.lanl.gov/data/software/#lanl-trace.
[35] "Fuse: Filesystem in userspace," http://fuse.sourceforge.net/.
[36] O. Zaki *et al.*, "Toward scalable performance visualization with jumpshot," *High Performance Computing Applications*, vol. 13, pp. 277–288, 1999.
[37] "mpip: Lightweight, scalable mpi profiling," http://mpip.sourceforge.net.
[38] "Fpmpi-2 fast profiling library for mpi," http://www.mcs.anl.gov/research/projects/fpmpi/WWW.
[39] M. Geimer *et al.*, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, apr 2010.
[40] P. Carns *et al.*, "24/7 characterization of petascale i/o workloads," in *Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.
[41] P. C. Roth, "Characterizing the i/o behavior of scientific applications on the cray xt," in *Proceedings of the 2nd International Workshop on Petascale Data Storage*, 2007.
[42] S. Seelam *et al.*, "Early experiences in application level i/o tracing on blue gene systems," in *IEEE International Symposium on Parallel and Distributed Processing IPDPS*, 2008.
[43] M. Litzkow *et al.*, "Checkpoint and migration of unix processes in the condor distributed processing system," University of Wisconsin-Madison Computer Sciences Technical Report #1346, Tech. Rep., 1997.