Practical Resource Monitoring for Robust High Throughput Computing

Gideon Juve[1], Benjamin Tovar[2], Rafael Ferreira da Silva[1], Casey Robinson[2]
Douglas Thain[2], Ewa Deelman[1], William Allcock[3], Miron Livny[4]

[1]University of Southern California, Information Sciences Institute, Marina Del Rey, CA, USA
{gideon,rafsilva,deelman}@isi.edu

[2]University of Notre Dame, Notre Dame, IN, USA
{dthain,crobins9,btovar}@nd.edu

[3]Argonne National Laboratory
allcock@alcf.anl.gov

[4]University of Wisconsin Madison, Madison, WI, USA
miron@cs.wisc.edu

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 underload 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 develop a model to describe various forms of monitoring and map the available mechanisms in contemporary operating systems to that model. Based on this analysis, we present two specific monitoring tools that choose different tradeoffs in overhead and accuracy, and evaluate them on a selection of benchmarks. We conclude by describing our experience in collecting large quantities of monitoring data for complex workflows.

1. 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 [37, 16, 31]. 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 [41], or Map-Reduce, 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 [2, 4, 1, 42, 23], but in practice this information is rarely available. As middleware layer get information from the user, without detailed 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 to 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 [11]. 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 here that the monitoring is done at a task, and not at a system level, as a task may misbehave without having 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 filesystem), but neither of these is appropriate for measuring the **independent resources consumed by each job** currently running on the machine. Collecting

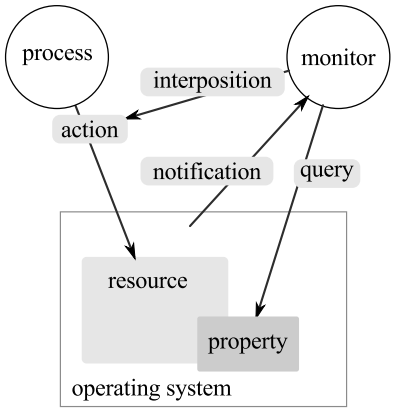


Figure 1: Resource Monitoring Model

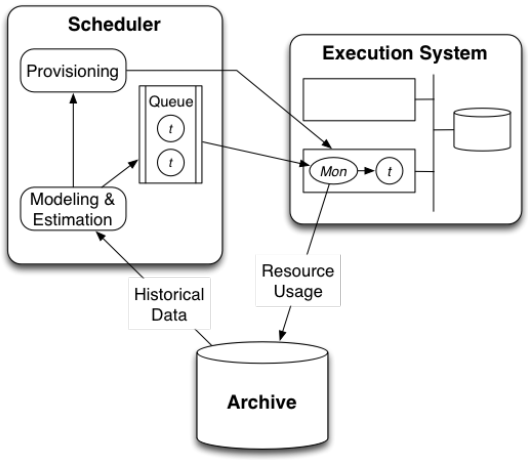


Figure 2: Resource Monitoring Loop

the desired information requires a combination of techniques and trade-offs between accuracy, overhead, and complexity. If resources are exhausted, it is important to know which process misbehaved, so that the mechanism can be used continuously during production operation to make suitable resource allocation decisions.

In this paper, we first develop an abstract model of resource monitoring and establish what information is necessary to make higher-level resource management decisions. We survey the large number of mechanisms available in current operating systems for obtaining the needed information, and discuss the advantages and drawbacks of each approach. We then select several mechanisms appropriate for continuous monitoring of production jobs, and evaluate tradeoffs in overhead and accuracy on a selection of benchmarks. We conclude by describing our experience in collecting a large amount of monitoring data from complex workloads.

2. RESOURCE MONITORING MODEL

We begin by describing a model of resource monitoring that is independent of the available mechanisms, so that we can clearly lay out what we intend to measure, and then compare our intent to the many available mechanisms.

2.1 Resource Monitoring Loop

Resource monitoring is one component of the overall resource management problem in HTC. Broadly speaking, we assume that an HTC system consists of a *scheduler* with a queue of tasks that must be executed, an *execution system* with resources for running tasks and storing data, and an *archive* for storing information about the resource usage of completed tasks (Figure 2). The scheduler queries the archive for historical data that can be used to construct a model for predicting the resource requirements of queued tasks. Based on the model, each task is labeled with an estimate of the resources it requires for a successful run (runtime, cores, memory, disk, etc.). The scheduler provisions resources from the execution system based on the resource requirements of queued tasks, and schedules the tasks to be executed on the provisioned resources. A *resource monitor* runs alongside each task, observing its resource consumption, and enforcing usage limits set by the scheduler based on its resource estimates. If the task stays within the specified limits, then it runs to completion. If it exceeds its resource limits, then the monitor forcibly halts it. When the task finishes, the monitor sends a report of the resources actually used to the archive.

Within this loop, a large number of challenging problems may arise, such as how to generate resource estimates, how to allocate tasks to machines, and so forth. In this paper, we focus entirely upon the design and implementation of the resource monitor which runs alongside each task. With accurate information and an enforcement mechanism in place, future work will address the other components of resource management.

The monitor plays both a measurement and enforcement role. It must observe the *resources requested* by the task and compare them to the *resources provided* by the OS. Together, these form the *resources contract*. A monitor must examine both sides of the contract, so that it determine if the task exceeds its stated needs and terminate it in an appropriate fashion. Upon completion, the monitor must produce a report that indicates the resources consumed by the application, and whether the contract was satisfied. The report could be a detailed summary of resources consumed over time, or simply the maximum of each during the run.

2.2 Monitoring Mechanisms

Figure 1 shows our basic model of resource monitoring. For clarity, we start with the problem of monitoring a single *process*, and extend it to multiple processes later.

An application process performs *actions* that affect the state of *resources* managed by the OS. Each resource has *properties* that summarize the current state of the resource. A *monitor* observes the behavior and results of the process to measure its resource usage. There are three general methods by which the monitor may attempt to learn about the resource usage of the 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.

For example, if we apply this model to a process using a file, then the resource is the file itself, the actions of the process are I/O system calls that manipulate the file, and the properties are information such as the size of the file. The

monitor may learn about the process's behavior by querying the file properties with `stat()`, by receiving notifications about file changes via `inotify()`, or by interposing I/O system calls with `ptrace()`.

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 generally has significant overhead and complexity and must be implemented 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. But, reliance on queries can result in incorrect 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 the damage has already been done, and other processes on the machine may have failed as a result.

Notifications from the operating system, where 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 operating systems and conditions may call for different approaches. For example, interpositions may substitute when notifications are not available, or coarse notifications might be used to trigger fine-grained queries for more detailed information.

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. For example, the monitor may be able to query the OS to get a list of processes, or ask the OS to deliver notifications when processes start and stop, or interpose functions involved in process management such as `fork()` and `exit()`.

2.3 Resource Types

We wish to monitor resources in three general categories: computation, memory, and I/O. For each of these resources, it is important to clearly distinguish between the *behavior* of the process and the *resources* provided by the OS, which are not the same thing. Broadly, interposition methods capture behavior, while notifications and queries observe resources provided by the OS.

Computation We model computation as one or more concurrent threads, each of which can be described by a set of increasing counters that yield useful measurements such as number of instructions completed. The computation resources actually delivered by the OS include the number of cores allocated to the task, the number of GPUs allocated, the cumulative time resident on each core, and, the hardware performance counters associated with each core.

For example, if a task consists of one process with eight concurrent threads, we would describe application behavior as the instruction counts for each individual thread, while we would describe the resources provided as the number of cores actually used which could vary between zero and eight over time.

Following our model of actions, properties, and notifications, information regarding computation resources can be obtained by interposing on actions that affect concurrency (`pthread_create`) by querying the operating system for wall and CPU time consumed by a process, or by observing the busy/idle transitions of a given core.

Memory We model memory consumption as the total virtual address space allocated by the process. Processes on modern OSes have a complex virtual memory space consisting of many logical segments dedicated to the stack, the heap, memory mapped files, and so forth. For each of those segments, we may describe the resources actually consumed as the number of resident pages in physical memory (resident set size) and the swap space in use.

A complication arises with applications that rely on the presence of virtual memory to optimize physical memory consumption. For example, an application might allocate a very large virtual address space, and then only touch a few pages. Or, it may memory map a large file, but never read it. In these cases, a complete description of the application's behavior must include both the virtual address size and the physical memory consumed. In addition, since memory use changes over time, it is important for resource allocation purposes to capture the maximum, or *peak*, memory usage.

Information regarding memory resources can be obtained by interposing actions that modify allocation (e.g., `malloc()` and `free()`), by directly querying the operating system for the amount of memory used (e.g., `getrusage()`), or by observing page fault events.

I/O We model the I/O behavior of an application as the complete list of operations (read, write, delete) applied to named objects (files or network connections). Of course, logging every single operation is expensive and rarely necessary. Instead, summaries of the number of operations and total data transferred are usually sufficient.

Again, there is a distinction between what the application requests and what the OS provides. For example, a process's read from a file may result in data being fetched from the file system cache, or from a local disk if the data is not cached. The former case does not involve any disk I/O while the latter does.

I/O monitoring involves interposing actions such as opening, reading, and writing to files, querying the operating system for properties such as files sizes, and observing file creation, modification, and deletion events.

2.4 Monitoring Challenges

There are several challenges and constraints that we encounter when trying to measure the resource usage of high-throughput computing applications. These challenges range from fundamental resource monitoring issues, to system- and application-specific challenges. They include:

- **Variability.** Resource usage may vary widely over time. For example, I/O may occur at the beginning and end of a process, but not in the middle. As a

result, it may be more useful to look at quantities such as total, peak or average resource usage, which may be more useful for resource planning than a time series.

- **Aliasing.** Aliasing refers to a single entity being accounted more than once. For example, hard links may appear as different files in the file system hierarchy, such that the file size is counted more than once. Similarly, the memory used by two concurrent processes in a task may be double counted if both use the same shared library.
- **Timing.** Some resource monitoring mechanisms can provide the data required, but provide no way to determine when it will be available. For example, the size of a file can be determined with `stat()`, but `stat()` gives no indication as to when a process is finished writing to the file. In those cases it may be necessary to combine multiple monitoring approaches, such as a query triggered by a notification, to obtain the desired information.
- **Parallelism / Concurrency.** Many scientific applications are parallelized using threads or message passing to improve performance. In some cases it may be difficult to develop a scalar measure of resource usage for parallel tasks. For example, it is relatively straightforward and intuitive to measure the total I/O of a parallel task, but it may be more complicated and difficult to measure the peak memory usage of a parallel task.
- **Operating Systems.** Many, if not most, scientific applications run on Linux, but other operating systems are frequently used for development, such as Mac OS X, and some proprietary scientific applications only run on Windows. In addition, many high-performance systems, such as IBM Blue Gene and Cray supercomputers run simplified kernels for performance reasons. These kernels lack many useful monitoring features found in other operating systems. They often have no virtual memory and are constrained in terms of RAM per core. This means limiting resource consumption for monitoring, and enforcing resource usage limits, is even more critical.
- **Permissions.** Because they provide access to sensitive data about a process (data that could lead to security vulnerabilities), many monitoring mechanisms require superuser (root) privileges. Unfortunately, many scientific applications run on systems owned and maintained by universities, supercomputer centers, and government laboratories. Getting root permissions on these systems for application-specific monitoring is usually not possible.
- **Scripts.** Many high-throughput applications and workflows are composed of a mix of binary programs and scripts in shell, Python, R, Matlab and other interpreted languages. Monitoring mechanisms that require users to recompile or relink their applications with tracing functions is usually not feasible for such applications because it would require them to have access to source code for the interpreters and the knowledge to recompile them.

3. MONITORING MECHANISMS

As mentioned in Section 2 we distinguish between three general mechanisms for obtaining process and task-level resource usage information: queries, notifications, and interpositions.

3.1 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, special on Linux systems, is `procfs`. `procfs` is a virtual file system (typically mounted at `/proc`) that exports data about the state of the operating system, including system-level and process-level information about memory, CPUs, disks, and filesystems. 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 by the process.

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 [33] is a popular library for querying performance counters, and Linux provides a tool called `perf` [32] to record 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 [30], provides utilization statistics, such as the number of threads, processor time, and memory consumed.

3.2 Notification Mechanisms

In notification mechanisms the operating system delivers messages to the 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 recieve 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` [40] 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 [25, 18]. 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 [8] on Solaris and SystemTap [39] 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.

3.3 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 task 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 a context switch 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 [26]). 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 [9]. `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 application binary at runtime to insert profiling instructions and software breakpoints. Projects such as DynInst [10] and Intel PIN [17] provide libraries for writing monitors that use DBI.

3.4 Monitoring on the Blue Gene Supercomputer Family

In this section we introduce some of the particular challenges when monitoring resources for Blue Gene supercomputers at the Argonne Leadership Computing Facility. For presentation purposes, rather than classifying the available mechanisms into interpositions, queries, and notifications, we give a brief description on some unique aspects of the Blue Gene architecture, together with their respective monitoring tools.

There are three general source of data on Blue Gene supercomputers: 1) The control system database and the cobalt scheduler database for basic job data, 2) autoperf, a memory and performance counter data tool, and 3) Darshan, an I/O monitoring tool.

Unlike many supercomputers, which have relatively commodity blade-like nodes, the Blue Gene uses a System-on-a-Chip (SoC) design where all the coordination, monitoring, control, etc., take place on a separate "service node" and the compute nodes are stripped down and optimized for pure computation. The service node is a DB2 (IBM's commercial database offering) server, running on commodity hardware, that maintains detailed state and history of the system. The service node interacts directly with the schedule, and is the definitive source of data for job (or task) start, stop, run time, and the number of nodes allocated to the job.

In contrast, the cobalt scheduler database provides insight into how jobs are organized. Some examples of this are:

- Due to the Blue Gene network architecture, the number of nodes that can be allocated to a job is not arbitrary. They have to be allocated in "blocks". When a job asks for a number of nodes that is not equal to a block size, they are allocated the next larger block size. The Blue Gene control system will report the number of nodes in the block. The cobalt scheduler database will report the number requested and the number actually used by the job. In theory, this could also be determined by he FLOP count (discussed below), which will be zero for unused nodes.
- Script jobs and ensemble jobs can have many Blue Gene jobs (or tasks) for a single scheduler job.

The control system database records every reliability, availability, and serviceability event generated during a job. This information can be easily use to detect jobs showing unusual variations in computational performance. The history of events is preserved, such that it is also possible to check for events that could have impacted the performance. For instance, on the BG/P correctable memory errors cause a performance loss, while on the BG/Q, the errors can be corrected without performance penalty.

The autoperf tool is a joint development between the Argonne Leadership Computing Facility and IBM research. There are two mechanisms for gathering data in autoperf. The first is relatively straight-forward: a pre-job script configures and zeros out the performance counters of interest on all the nodes for the job, and a post-job script aggregates and calculates statistics on those performance counters. There are over 200 data points available via this mechanism. For instance, this is how we can obtain total FLOPs, as well as the FLOPs that were executed in the quad floating point unit (QPX). This approach has effectively zero overhead from a job performance perspective, however it can fail if the user reprograms the performance counters for their own use.

The second mechanism used by autoperf is the standard PMPI interposition mechanism. Our initial configuration under test has been stripped down to a minimal set of functions and data to minimize the impact on performance. For instance, for each MPI call, it performs two adds–one for the number of times this routine has been called and one for the number of bytes (where that makes sense)–two compares–for max and min–and a store if there is a new max or min. When `mpi_finalize()` is called, the data is aggregated and additional statistics are calculated. This method can fail if another tracing or debugging tool also uses the PMPI interposition mechanism.

We do not currently know of mechanisms for obtaining data on peak memory usage, though we have discussed a mechanism that we can add to the autoperf tool in order to obtain estimates. That said, there are library calls available that can query various aspects of memory usage, such as the size of the text, data, and BSS segments, together with the maximum amount of memory consumed on the heap.

We also use Darshan [3], a resource monitoring tool with two major design points. First, it was explicitly aimed at parallel I/O, since there are no well accepted tools for doing so. Second, it was design to consume a small amount of resources, with a fixed maximum, and as close to zero overhead as possible. Darshan also uses the standard PMPI profiling interface to interpose MPI I/O and other techniques for interposing POSIX I/O functions. To meet the second design point, Darshan captures a statistical cross section of the I/O rather than full stats. For instance, for the write size, which is a critical parameter in I/O, it keeps a histogram of sizes rather than a complete list of every file size. However, it does keep track of the 4 most common exact write sizes. Data is kept at each individual MPI rank and Darshan does no internode communication until the `mpi_finalize()` call is made. At that point the ranks do a gather of all the individual rank data sets and write out the input. Output is on a per job basis and tools are provided to assist in analyzing the data. The data obtained for each call is exact, so there is no estimation or error introduced here, though obviously information is lost by the statistical reduction of the data.

3.5 Comparison of Mechanisms

Table 1 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()`.

3.6 Detailed Example: Memory-Mapped I/O

Memory-mapped I/O is an interesting case, because it combines multiple resources and multiple access types. The `mmap()` system call establishes a memory segment that corresponds to a file, and I/O is performed and physical memory is allocated when addresses within the mapped segment are accessed by the process.

- After an `mmap` call, only the virtual memory size is modified, to fit the memory addresses of the mapped file. The value of the peak virtual memory might be accessed by **querying** the OS when the task finishes, or estimated by querying/polling. The intent to reserve a virtual memory segment can be captured by **interposing** the call to `mmap()`. Note that if only a small portion of the file is ever accessed, then virtual memory may greatly overestimate memory usage.
- When the task tries to read from a memory address in the mapped file, a memory page fault occurs and I/O operations occur in the disk. The number of bytes read therefore is a function on the total count of page faults. This count could come from a **query** to the OS, or by directly counting the number of page fault **events**. Similarly, to account for bytes written, the number of pages modified (i.e., dirty pages) that need to be synchronized with the file on disk may be found, We note that this information is unavailable, or restricted in most operating systems implementations.
- As the free resident memory decreases, some of the memory pages might be removed from resident memory. The peak resident memory used by a memory mapped file could be computed from counting the **events** of page faults/discards, or from a **query** to the OS. As mentioned above, this information is often unavailable (for example, a query in Linux provides the current resident memory usage by a memory mapped file, but not peak usage).

4. 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 [5], and Kickstart [6], which is part of the Pegasus Workflow Management System [7]. Table 2 describes the resource and process information these tools record for each task.

4.1 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.

4.2 Kickstart

Kickstart [6] 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 [7], 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 used for events and interposition because, unlike other interposition mechanisms, it does not require the application code to be recompiled, and because 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 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

Table 1: 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	[c]
GPU Libraries	C	Medium	Linux	Low	Any	Low	Process	[d]
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	[e]
wait4()	C,M,I	Low	UNIX	Low	Owner	Low	Process	[f]
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	[g]
LD_PRELOAD	F,I,M,T,P	High	UNIX	Medium	Owner	High	Process	[h]
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] Some systems do not populate memory and/or I/O
[d] With supervisor privilege scope is expanded to System, Process
[e] Does not associate events with processes
[f] Some systems do not populate memory and/or I/O
[g] Requires re-compiling or re-linking
[h] Only works for dynamic libraries

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.

4.3 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 5. 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 (see Table 2), 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 the resource exhausted.

5. EVALUATION

In this section we evaluate our implementation of the different monitoring levels described in Section 4.1. 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 ptrace events with kickstart; finally, for level 3, we also interpose system calls using ptrace with kickstart. We first evaluate the accuracy of these tools while measuring CPU, memory, and I/O consumption on mock processes, and then we evaluate the performance impact (overhead) of performing monitoring. The experiments were conducted on a 12-core Intel Xeon 2.67GHz with 40GB of RAM. For each configuration,

Table 2: Resources measured

Field	Notes
cores	Number of cores used by the task
gpus	Number of gpus used by the task
start	The timestamp when the process started, seconds since epoch
end	The timestamp when the process exited, seconds since epoch
exit type	*normal* indicates that it called exit() with a zero OR non-zero exitcode, *signal* indicates it exited on an uncaught signal, and *limit* indicates it was killed for exceeding one of its resource limits
signal	The number of the signal that terminated the process, if any
exit status	The status returned by the task
limits	Comma-separated list of all of the resource limits that were exceeded, in the form field: peak > limit, field: peak > limit, etc., if any
concurrent procs.	The maximum number of processes that ran concurrently
cpu time	The total CPU time of the process (utime+stime)
wall time	end_time - start_time
virtual memory	Maximum sum of virtual memory peaks of all the sets of concurrent processes
resident memory	Maximum sum of resident memory set peaks of all the sets of concurrent processes
swap memory	Maximum sum of swap memory peaks of all the sets of concurrent processes
bytes read	Count of all of the bytes read in I/O operations.
bytes written	Count of all of the bytes written in I/O operations.
number files/dirs	The peak count of files and directories in the working directory.
footprint	The peak value of the size of all files and directories in the working directory.

5 repetitions of the experiment were performed, which were sufficient to obtain average values with less than 2% error.

5.1 Accuracy

Table 3 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 3(a) shows the average CPU time (utime+stime) for each configuration when executed without monitoring (Baseline), and the error ratios of these times reported by the monitoring tools when compared to the actual values. Positive error ratios (resp. negative) mean that the monitoring tool overestimates (resp. underestimates) the CPU consumption. 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 `ptrace()` in cases where there are few system 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 3(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 when 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. Both experiments use the O_DIRECT flag to reduce the impact of the file system cache on the results.

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. Table 3(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 1GB of data with buffer sizes ranging from 4KB to 32KB. Average error ratios for bytes read are shown in Table 3(d). Again, the results for bytes written were similar to bytes read, 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.

5.2 Overhead

The overhead of the monitoring tools was measured for the same experiments described in the previous section.

Table 4(a) shows the CPU overhead in seconds, and the percentage overhead in brackets, for all the experiments from the previous section. A relatively large overhead is observed for very short executions, but as the execution time increases, the overhead becomes less than 1%. This reflects a small, approximately constant overhead for all the tools.

The impact of memory monitoring is shown in Table 4(b). In most cases the overhead is less than 5% regardless the amount of data written. The overhead is slightly larger in the case of LD_PRELOAD. We belive there are two reasons for this: First, resource_monitor writes to disk the library to be preloaded; and second, there is extra overhead caused by the monitor synchronizing the end event of the task when intercepting `exit()`.

Both tools 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 Table 4(c), the average overhead is above 1000% for a small amount of data, but this is likely just a result of the very short runtime of the program. As the amount of read/written data increases, the overhead ratio progressively decreases to less than 2% in most cases. In the case where Kickstart interposes system calls, however, the overhead remains high. This is a result of the large number of system calls that are intercepted, which each impose an overhead on the process.

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

Table 3: Monitoring Accuracy

	Baseline	Polling (resource_monitor)		fork/exit LD_PRELOAD (resource_monitor)		fork/exit ptrace (kickstart)		syscall ptrace (kickstart)	
Instr.	**(a) CPU time**								
10^6	0.32 s	+0.04	(12.50%)	+0.02	(4.91%)	0.00	(0.00%)	0.00	(0.00%)
10^7	2.93 s	+0.06	(2.12%)	+0.04	(1.20%)	0.00	(0.00%)	+0.01	(0.14%)
10^8	28.20 s	+0.17	(0.60%)	+0.09	(0.31%)	+0.03	(0.10%)	+0.04	(0.14%)
10^9	279.53 s	+1.29	(0.46%)	+1.32	(0.47%)	+0.20	(0.07%)	+0.41	(0.15%)
Memory	**(b) Memory: resident size**								
1GB	1GB		−13.96%		+0.08%		+0.03%		+0.03%
2GB	2GB		−17.63%		+0.03%		+0.02%		+0.02%
4GB	4GB		−2.25%		+0.02%		0.00%		0.00%
8GB	8GB		−1.89%		+0.01%		0.00%		0.00%
16GB	16GB		−1.99%		+0.01%		0.00%		0.00%
File size	**(c) I/O: bytes read, 4KB buffer**								
1MB	1MB		−13.64%		0.00%		0.00%		0.00%
100MB	100MB		−9.07%		0.00%		0.00%		0.00%
1GB	1GB		−5.84%		0.00%		0.00%		0.00%
10GB	10GB		−2.13%		0.00%		0.00%		0.00%
Buffer size	**(d) I/O: bytes read, 1GB file**								
4KB	1GB		−5.84%		0.00%		0.00%		0.00%
8KB	1GB		−0.82%		0.00%		0.00%		0.00%
16KB	1GB		−15.41%		0.00%		0.00%		0.00%
32KB	1GB		−18.41%		0.00%		0.00%		0.00%

same, but the percentage increases just because the larger buffer size results in a shorter runtime.

6. MONITORING ARCHIVE

Information gathered while monitoring HTC applications can be valuable for our use but also to the community at large, enabling research in resource provisioning, workload scheduling, performance prediction, and others. To this end, we have created a resource summary archive to capture the information gathered by the monitoring tools. The archive is publicly readable at `http://dvdt.crc.nd.edu`, and is build on top of the content management system Drupal with custom PHP and python code, with a database backend running mysql. Users of the archive can submit sets of resources summaries through a web interface, or with a batch job using ssh keys for authentication. When submitting sets of resources summaries, a description of the set may be included. This description, such as the directed acyclic graph of tasks dependencies in a workflow, is used to characterize and compare tasks across different sets. The archive can be queried to produce task summaries that match conditions, such as task name, monitoring tool used, set description, and resource values comparisons.

Emphasising that monitoring from the user perspective is different from a system administrator perspective, instances of the same task may show different resources values; the task may not change, but it is difficult to run the task in the same environment every time. Even if task instances run on the same host, the resources available at the host change (e.g., memory available to the task). This presents two non-trivial challenges: how are the sets of possible resource values characterized?, and how do we design exception handling for such a wide range of valid resource values? We plan to address these questions in our future work, and as of today, we are using the archive to observe and better understand variability of resources usage for a given task running on different available nodes.

As an example of this resource variability, we include in Table 5 some statistics for the task `rmapper`, part of the SHRiMP[35] package, to align genomic sequences to target genomes. The statistics were computed from 96,501 resource summaries, executed using the Condor pool at the University of Notre Dame. The pool has approximately 12,850 nodes, running different versions of Linux for the x86_64 architecture. In Table 5 we only include resources that show some interesting variability; resources such as virtual memory and disk footprint had very dominant peaks (kurtosis in the order of thousands), which as expected for well-behaved tasks, describe very small variability across the different computing nodes. In comparison, high variability in resident memory is reflected in relatively high standard deviation and negative kurtosis.

In future work, we plan to use the archive to bootstrap resource management loop: when executing new tasks, the resources used by previous tasks instances can be queried from the archive, and appropriate resource allocation, with resource limits to be enforced can be determined.

7. 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 [15], which includes `sar` and other tools.

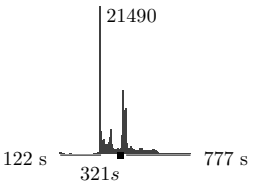
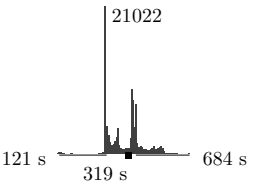
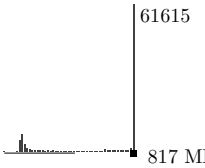
Many distributed monitoring systems, including Gangila [24], Nagios [29], and Munin [28], have been developed to provide system-level monitoring information. These systems are typically used by system administrators for problem detection and troubleshooting. They do not record the detailed, job- or process-level 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 [22] 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 [43].

Table 4: Monitoring Overhead

	Baseline	Polling (resource_monitor)		fork/exit LD_PRELOAD (resource_monitor)		fork/exit ptrace (kickstart)		syscall ptrace (kickstart)	
Instr.		**(a) CPU overhead**							
10^6	0.32 s	+0.22	(68.75%)	+0.25	(78.13%)	+0.18	(56.25%)	+0.13	(40.63%)
10^7	2.93 s	+0.28	(9.56%)	+2.42	(82.59%)	+0.14	(4.78%)	+0.14	(4.78%)
10^8	28.20 s	+0.17	(0.60%)	+0.22	(0.78%)	+0.10	(0.35%)	+0.12	(0.43%)
10^9	279.53 s	+0.28	(0.10%)	+0.78	(0.28%)	+0.07	(0.03%)	+0.61	(0.22%)
Resident size		**(b) Memory overhead**							
1GB	3.57 s	+0.17	(4.76%)	+0.26	(7.28%)	+0.06	(1.68%)	+0.07	(1.96%)
2GB	6.19 s	+0.10	(1.62%)	+0.14	(2.26%)	+0.09	(1.45%)	+0.06	(0.97%)
4GB	12.64 s	+0.50	(3.96%)	+0.86	(6.80%)	+0.24	(1.90%)	+0.43	(3.40%)
8GB	25.06 s	+0.51	(2.04%)	+1.88	(7.50%)	+0.87	(3.47%)	+0.96	(3.83%)
16GB	52.81 s	+1.11	(2.10%)	+4.69	(8.88%)	+1.38	(2.61%)	+2.25	(4.26%)
File size		**(c) I/O overhead, 4KB buffer**							
1MB	0.01 s	+0.17	(1700%)	+0.24	(2400.00%)	+0.13	(1300.00%)	+0.14	(1400.00%)
100MB	1.53 s	+0.09	(5.88%)	+0.10	(6.54%)	+0.09	(5.88%)	+1.82	(118.95%)
1GB	16.02 s	+0.04	(0.25%)	+0.38	(2.37%)	+0.36	(2.25%)	+15.98	(99.75%)
10GB	153.98 s	+0.54	(0.35%)	+0.64	(0.42%)	+0.58	(0.38%)	+143.95	(93.49%)
Buffer size		**(d) I/O overhead, 1GB file**							
4KB	16.02 s	+0.04	(0.25%)	+0.38	(2.37%)	+0.36	(2.25%)	+15.98	(99.75%)
8KB	9.14 s	+0.20	(2.19%)	+0.38	(4.16%)	+0.24	(2.63%)	+8.72	(95.40%)
16KB	6.40 s	+0.23	(3.59%)	+0.34	(5.31%)	+0.30	(4.69%)	+4.13	(64.53%)
32KB	4.37 s	+0.18	(4.12%)	+0.43	(9.84%)	+0.60	(13.73%)	+2.11	(48.28%)

Table 5: Resource Archive Statistics for 96501 Instances of a Single Task in a Workflow

resource	wall time	cpu time	resident memory
histogram			
mean	410.55 s	406.17 s	682.62 MB
std. dev.	79.16	73.86	208.83
skewness	0.42	0.17	-1.11
kurtosis	0.26	-0.10	10.96

There are several tools that use interposition to collect information about program behavior. The `strace` [38] and `ltrace` [21] tools use interposition to report system calls and library calls, respectively. `LANL-Trace` [19] uses these tools to profile the I/O behavior of parallel applications. `ParaTrac` [9] interposes I/O operations using a FUSE [13] 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` [40] and the workflow management system to provide complete application profiles.

Many MPI profiling libraries that use PMPI for function interposition, including Jumpshot [44], mpiP [27], FPMPI [12], Scalasca [14] and others. Function interposition is used by several tools to implement I/O profiling. Darshan [3] uses PMPI and other function call interposition techniques to observe the I/O behavior of MPI applications. IOT [34] uses both PMPI and a GNU linker extension that enables functions to be wrapped at link time to enable I/O tracing. HPCT-IO [36] 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 [20]

8. CONCLUSION AND FUTURE WORK

In this paper we presented a study of resource usage monitoring techniques for a broad spectrum of science applications. 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 different levels of monitoring, and presented an evaluation of the accuracy and overhead of these tools.

In the future we plan to deploy our monitoring tools on production infrastructure to collect resource usage data for science applications. This data will help us extend our previous work [11] on using historical resource usage data to automatically construct resource usage models for applica-

tions. 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.

9. ACKNOWLEDGMENTS

This work was funded by DOE under the contract number ER26110, "dV/dt - Accelerating the Rate of Progress Towards Extreme Scale Collaborative Science".

10. REFERENCES

[1] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, May 2005.

[2] T. D. Braun, H. J. Siegel, N. Beck, L. L. BÃűlÃűni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, June 2001.

[3] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[4] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop*, 2000.

[5] CCTools. `http://www3.nd.edu/~ccl/software/download`.

[6] E. Deelman, G. Metha, J.-S. VÃűckler, M. Wilde, and Y. Zhao. Kickstarting remote applications. In *International Workshop on Grid Computing Environments*, 2006.

[7] E. Deelman, G. Singh, M. 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*, 13(3):219–237, 2005.

[8] DTrace. `http://dtrace.org`.

[9] N. Dun, K. Taura, and A. Yonezawa. ParaTrac: a fine-grained profiler for data-intensive workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010)*, 2010.

[10] DynInst. `http://www.dyninst.org`.

[11] R. Ferreira da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny. Toward fine-grained online task characteristics estimation in scientific workflows. In *8th Workshop on Workflows in Support of Large-Scale Science*, 2013.

[12] FPMPI-2 fast profiling library for MPI. `http://www.mcs.anl.gov/research/projects/fpmpi/WWW`.

[13] FUSE: Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[14] M. Geimer, F. Wolf, B. J. N. Wylie, E. ÃĄbrahÃąm, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, Apr. 2010.

[15] S. Godard. Sysstat. `http://sebastien.godard.pagesperso-orange.fr`.

[16] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. 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.

[17] Intel PIN. `http://software.intel.com/en-us/articles/pintool`.

[18] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Ottowa Linux Symposium*, 2007.

[19] LANL-Trace. `http://institute.lanl.gov/data/software/#lanl-trace`.

[20] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical report, University of Wisconsin-Madison Computer Sciences Technical Report #1346, 1997.

[21] ltrace. `http://ltrace.org`.

[22] C.-D. Lu, J. Browne, R. L. DeLeon, J. Hammond, W. Barth, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra. 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.

[23] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling strategies for mapping application workflows onto the grid. In *14th IEEE International Symposium on High Performance Distributed Computing*, 2005.

[24] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[25] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of kprobes. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[26] Message Passing Interface Forum. MPI: a message-passing interface standard, 2003.

[27] mpiP: Lightweight, scalable MPI profiling. `http://mpip.sourceforge.net`.

[28] Munin. `http://munin-monitoring.org`.

[29] Nagios. `http://nagios.org`.

[30] Nvml. `https://developer.nvidia.com/nvidia-management-library-NVML`.

[31] Open Science Grid. `http://opensciencegrid.org`.

[32] perf. `http://perf.wiki.kernel.org`.

[33] Performance application programming interface (PAPI). `http://icl.cs.utk.edu/papi`.

[34] 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.

[35] S. Rumble, P. Lacroute, A. Dalca, M. Fiume, A. S. A, and et. al. SHRiMP: Accurate mapping of short color-space reads. *PLoS Compututational Biology*, 5(5), 2009.

[36] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu. Early experiences in application level I/O tracing on blue gene systems. In *IEEE International Symposium on Parallel and Distributed Processing IPDPS*, 2008.

[37] R. Sobie, A. Agarwal, I. Gable, C. Leavett-Brown, M. Paterson, R. Taylor, A. Charbonneau, R. Impey, and W. Podiama. HTC scientific computing in a distributed cloud environment. In *4th ACM Workshop on Scientific Cloud Computing*, 2013.

[38] strace. `http://sourceforge.net/projects/strace`.

[39] SystemTap. `https://sourceware.org/systemtap`.

[40] taskstats. `http://www.kernel.org/doc/Documentation/accounting/taskstats.txt`.

[41] I. Taylor, E. Deelman, D. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., 2007.

[42] H. Topcuoglu, S. Hariri, and W. Min-You. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[43] D. D. Vento, T. Engel, S. S. Ghosh, D. L. Hart, R. Kelly, S. Liu, and R. Valent. System-level monitoring of floating-point performance to improve effective system utilization. In *Supercomputing*, 2011.

[44] O. Zaki, E. Lusk, and D. Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.