Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Where is your application stuck?

Shailabh Nagar
IBM India Research Lab
`nagar1234@in.ibm.com`

Balbir Singh Vivek Kashyap Chandra Seetharaman
Narasimha Sharoff Pradipta Banerjee
IBM Linux Technology Center

`{balbir@linux.vnet,vivk@us,sekharan@linux.vnet}.ibm.com,`
`nsharoff@beaverton.ibm.com, bpradipt@in.ibm.com`

Abstract

Xen migration or Container mobility promise better system utiliztion and system performance. But how does one know if the effort will improve the workload's progress? Resource management solutions promise optimal performance tuning. But how does one determine the resources to be reallocated and the impact of the allotment? Most customers develop their own benchmark that is used for purchasing a solution, but how does one know that the bottleneck is not in the customer benchmark?

Per-task delay accounting is a new functionality introduced for the Linux kernel which provides a direct measurement of resource constraints delaying forward progress of applications. Time spent by Linux tasks waiting for CPU time, completion of submitted I/O and in resolving page faults caused by allocated real memory, delay the forward progress of the application being run within the task. Currently these wait times are either unavailable at a per-task granularity or can only by obtained indirectly, through measurement of CPU usage and number of page faults. Indirect measurements are less useful because they make it harder to decide whether low usage of a resource is due to lack of demand from the application or due to resource contention with other tasks/applications.

Direct measurement of per-task delays has several advantages. They provide feedback to resource management applications that control a task's allocation of system resources by altering its CPU priority, I/O priority and real memory limits and enable them to fine tune these parameters more quickly to adapt to resource management policies and application demand. They are also useful for accurate metering/billing of resource usage which is particularly useful for shared systems such as departmental servers or hosting platforms. For desktop users, these statistics provide a quick way of determining the resource bottleneck, if any, for applications that are not running as fast as expected.

In this paper, we describe the design, implementation and usage of per-task delay accounting functionality currently available in the Linux kernel tree. We demonstrate the utility of the feature by studying the delay profiles of some commonly used applications and how their resource usage can be tuned using the feedback provided. We provide a brief description of the alternative mechanisms proposed to address similar needs.

1 Introduction

Linux is supported on a wide range of systems, from embedded to desktop to mainframe, running a diverse set of applications ranging from business and scientific to office productivity and entertainment. Linux provides multiple metrics and "knobs" for achieving optimal performance in these various deployments. There are a myriad of well known benchmarks and tools available to help gather and analyze and then tune the systems for desired performance.

However, after elaborate tuning as well the applications or system may still perform unsatisfactorily. The resource bottlenecks blocking the progress must be addressed through additional provisioning, resource reallocation or even load-balancing by migration of the application/service to another system.

There is on-going work such as resource groups and 'containers' to re-allocate resources to desired workloads for better system utilization and performance. Xen and Linux containers support the notion of migrating live workloads to consolidate systems when load is low and migrate to other systems for better load-balancing. However, all these mechanisms will work 'blind' without a good method to pinpoint the resources that need tuning.

The delay-accounting framework is built on the observation that: since ultimately the application's progress is most important, the bottlenecks impeding the applications progress must be easily identifiable. The delay-accounting framework therefore gathers the time spent on behalf of the task (or task group) in queues contending for system resources. This information may then be utilised by workload management applications to dynamically increase the desired applicaton's access to the bottlenecked resource by raising its priority, or share of the resource pool, or even initiating service or application migration to a different system.

This paper discusses the design, implementation and utility of the per-task delay accounting framework developed to address this issue. The following sectons outline the use cases

2 Motivation

The traditional focus of operating system accounting is the time spent by a Linux task doing a particular activity e.g. time spent by a task executing on a cpu in system mode, user mode, in interrupts etc. There is also a measure of the system activity that happens e.g. number of system calls, I/O blocks transferred, network packets sent/received. While these metrics are often useful for measuring overall system utilization or for high level detection of performance anomalies, they are not directly usable for determining what is delaying forward progress of a specific application, except perhaps by skilled and experienced system administrators.

Hence there is a need for answering the simple question: what resource, managed by the operating system, is my application waiting for? To start with, the resources of interest can be high level such as cpu time, block I/O bandwidth and physical memory. These, respectively, translate to knowing how long a Linux task (or groups of tasks) spend in, i.e. get delayed, in

- waiting to run on a CPU after becoming runnable waiting for block
- I/O operations inititated by the task to complete waiting for page
- faults to complete

The information is useful in at least two distinct scenarios:

Simple hand tuning: If a favorite task seems to be spending most of its time waiting for I/O completion, raising its I/O priority (compared to other tasks) may help. Similarly, if a task is mainly waiting on page faults, increasing its RSS limit (or running other memory hogs at a much lower cpu priority) may help alleviate the resource bottleneck.

Workload management: One of the objectives of modern workload management tools is to manage the forward progress of aggregates of tasks which are involved in a given business function. The idea being that if business function A is more important than another one B, the applications (tasks and processes from an OS viewpoint) involved in the former should get preferential access to the OS resources compared to B.

To achieve this, workload managers need to periodically know the bottlenecked resource for all tasks running on the system and use that information, in conjunction with policies that determine prioritization, to adjust the resource priorities like nice values, I/O priorities and RSS limits. Workload managers may also make a decision to shift a given application off a system if it determines that priority boosting isn't helping.

Per-task delay accounting was designed and implemented to meet the above needs. We next describe the design considerations in detail.

3 Design and implementation

The design and implementation of per-task delay accounting had to take the following three major factors into consideration.

3.1 Measurement of delays

The design objective was to measure the most significant sources of delay for a process to the highest accuracy possible, preferably at nanosecond granularity. The approach taken was to take high resolution timestamps at the appropriate kernel functions and accumalate the timestamp differences into per-process data structures.

The delay sources chosen and the manner in which they are collected are:

1. Waiting to run on a cpu after becoming runnable: Here we needed to take timestamps when a process was added to a cpu runqueue and again when it was selected to be run on a cpu. The schedstats codebase, added to help gather cpu scheduler statistics for development and debugging purposes, came in handy since it already had code to gather these timestamps and take their differences. The schedstats functions `sched_info_arrive` and `sched_info_depart` which gathered other statistics that were not of interest to delay accounting had to be refactored to minimize the performance impact. Another decision made to keep the performance impact low was to stick to the jiffie level timestamp resolution used by schedstats instead of higher resolution ones available in the kernel.

2. Waiting for block I/O submitted to complete: Here we had a choice of trying to accurately measure the entire delay in submitting block I/O as well as the delay incurred in the block device performing the I/O. However, it was finally decided that we would measure only the time spent by a process in sleeping for submitted I/O to complete. Measuring the delays in the I/O submission path as well turned out to be quite complex, given the diversity of functions that are involved in block I/O submission as well as the difficulty of correctly attributing the delays to the right process in all these paths. These factors would have necessitated a number of timestamps being collected all over the kernel code which affects maintainability. Moreover, much of the delays seen in I/O submission cannot be changed by the user (other than indirectly by affecting the cpu scheduling of the submitting process) whereas delays incurred after I/O submission can be affected by tweaking the I/O scheduling priority of the process. Hence it was decided to only measure the time spent in `sched.c/io_schedule()` using high resolution timestamps.

3. Waiting for a page fault to complete We had briefly considered measuring the delays seen in both major and minor page faults but later concentrated only on the former to minimize impact of delay collection code on the virtual memory management code paths. Delays for major page faults are a subset of the block I/O delays which were already being measured so the only change needed was to record the fact that a process was in the middle of a major page fault and use this information at block I/O delay recording time.

Delay accounting also measures and returns an interval-based measurement of time spent running on a cpu. This is more accurate than the sampling-based cpu time measures normally available from the kernel (via a task's utime, stime fields in task struct). Accurate cpu usage times are valuable in accurate workload management. One factor that has to be considered for cpu time is the possibility of the kernel running as a guest OS in a virtualized environment. These "guest OSes" are scheduled by the hypervisor in accordance with its scheduling policies and priorities for the OS instance. As a result, the times spent waiting for a particular resource need to be measured in wall-clock times. In contrast the the utilization of the resource occurs only when the Guest is executing. This differentiation enables an educated assessment of resource allocation (such as additional CPU share) needed by the "guest" or the resources within the the "guest OS."

The delay accounting framework retuns both the real and virtual cpu utilization on virtual systems (currently implemented for LPARs on ppc64) as well as the delays experienced by the individual task groups in the OS instances.

3.2 Storing the delay accounting data

Each of the delay data recorded above needed to be stored taking into account two important constraints:

1. The data structures should be expandable to add other per-task delays that were deemed useful in

future. While the current per-task delay accounting only looks at wait for cpu, block I/O and major page fault delays, it is well possible that other delays associated with kernel resource allocation would be of interest and measured in future.

2. The data should be collected per-task and also aggregated per-process (i.e. per-tgid) within the kernel. This latter design constraint drew a lot of comments when the delay accounting code was proposed since it was felt that per-process delays could as well be accumulated outside the kernel in userspace. However, we had observed that accurate measures of per-process delays were very useful for performance analysis of bottlenecks and also that accumalating per-task delays in userspace required far too much overhead and reduced accuracy due to presence of short-lived tasks in a process. Hence it was necessary to have a per-process delay aggregating data structure that would keep the delay data collected for an exiting task and make it available until all tasks of the process exited.

   The taskstats data structure, defined in `include/linux/taskstats.h`, took into account these constraints. It was versioned, mandated that new fields would be added at the bottom and also took into account alignment requirements for 32 and 64 bit architectures.

   To meet the second constraint, two copies of the data structure are maintained. One is per-task, maintained within the `task_struct`. The other is per-tgid, maintained within the `signal_struct` associated with a tgid. The fields of struct taskstats are large enough to serve as an accumalator for per-tgid delays.

3.3 Interface to userspace to supply the data

The interface to access delay accounting data from userspace formed a large part of the discussion preceding the acceptance of delay accounting in the kernel. The various design constraints for the interface were:

1. Efficient transfer of large volumes of delay data: Workload managers that are monitoring the entire system for performance bottlenecks need to gather delay statistics from each task periodically. In order to allow this period to be small, it is essential that the data transfer of delay accounting data be efficient while handling large volumes (due to potentially large number of tasks). A similar requirement, albeit less severe, exists even for monitoring a single application if its degree of multithreading is sufficiently high.

2. Handle rapid exit rate without data loss: In order to do workload management at the level of user-defined groups of processes, workload managers need to get the cumulative delays seen by a task (and a thread group) right up to the time it exits. Hence delay accounting data needs to be available even after a task (or thread group) exits. Obviously the kernel cannot keep such data around for a long time so the choice was made to use a "push" model (kernel->user) to send such data out to listening userspace applications rather than require them to "pull" the data. In such a scenario, its important to be able to handle a rapid rate of task/thread group exits, if not on a sustained basis, atleast for short bursts, without losing data.

3. Exporting data as text or not: This is another instance of the classic debate whether such data should be exported as text through /proc or /sysfs like interfaces allowing it be directly read in user space using shell utilities or whether it should be a structured binary stream requiring special user space utilities to parse. Given the volume of data needing to be transferred, the latter option was chosen.

4. Bidirectional read/writes: There was a need for userspace to send commands and configuration information to the kernel delay accounting and hence interfaces like relayfs which lacked a user->kernel write capability were not usable.

Given these constraints, we decided to use the newly introduced genetlink interface. Genetlink is a generalized version of the netlink interface. Netlink, which exports a sockets API, has been used to handle large volumes of kernel<->user data, primarily for networking related transfers. But it suffered the limitation of having a limited number of sockets available for use by different kernel subsystems as well as an API that was too network-centric for some. Genetlink was created to address these issues. It multiplexes multiple users over a single netlink socket and simplifies the API they need to use to effect bidirectional data transfer. Delay accounting was one of the early adopters of the genetlink mechanism and its usage provided inputs for refining and validating the genetlink interface as well.

Delay accounting handles the rapid exit rate constraint by splitting the delay data sent on exit into per-cpu streams. A listening userspace entity has to explicitly register interest in getting data for a given cpu in the system. Once it does so, it receives the exit delay data for any task which exits while last running on that cpu. This design allows systems with many cpus (which will typically have a correspondingly larger number of exiting tasks) to balance the exit data bandwidth amongst multiple userspace listeners, each listening to a subset of cpus. CPUs are used as a convenient means of dividing up the exit data requirements. They also help when the cpusets mechanism is used to physically partition up machines with very large number of cpus and some of the physical partitions have strict performance requirements that prohibit exit data from being processed. Being able to regulate the sending of exit delay data by the kernel by cpu allows fine-grain control over the performance impact of delay accounting.

Finally, virtual machine technology enables a single system to run multiple OS instances. These "guest OSes" are scheduled by the hypervisor in accordance with its scheduling policies and priorities for the OS instance. As a result, the times spent waiting for a particular resource need to be measured in wall-clock times. In contrast the the utilization of the resource occurs only when the Guest is executing. This differentiation enables an educated assessment of resource allocation (such as additional CPU share) needed by the "guest" or the resources within the the "guest OS."

The delay accounting framework retuns both the real and virtual cpu utilization on virtual systems (currently implemented for LPARs on ppc64) as well as the delays experienced by the individual task groups in the OS instances.

3.4 Delay accounting lifecycle

With the above elements in place, its useful to outline how the delay accounting works in practice. The description is being kept generic without referring to specifics of the interface since that can be obtained elsewhere.

On system startup, the system administrator can optionally start userspace "listeners" who register to listen to exit delay data on one or more cpus. The typical usage is to start one listener listening to all the cpus of the system.

When a task is created (via fork), a taskstats data structures get allocated. If the task is the first one of a thread group, a per-tgid taskstats struct is allocated as well. As the task makes system calls, the delays it encounters get measured and aggregated into both these data structures.

At any point during the task's lifetime, a user can query the delay statistics for a task, or its thread group, via the genetlink interface by sending an appropriate command. The reply contains the taskstats data structure for the task or thread group.

When the task exits, its delay data is sent to any listener which has registered interest in the cpu on which the exit happens. If the task is the last one in its thread group, the accumalated delay data for the thread group is additionally sent to registered listeners.

3.5 Per Task IO Accounting

An important related work are the per-task I/O accounting statistics by Andrew Morton which improve the accuracy of measurement of I/O resource consumption by a task. The CSA infrastructure also supports per-task IO statistics, but the data returned by it, can be incorrect.

CSA accounts for per task IO statistics using data read or written through the `sendfile(2)`, `read(2)`, `readv(2)`, `write(2)`, and `writev(2)` system calls. It does not account for

1. Data read through disk read ahead
2. Page fault initiated reads
3. Sharing data through the page cache. If a page is already dirty and another task T1, writes to it, both the task that first dirtied the page and T1 will be charged for IO.

The following example illustrates the deficiencies of the current per task IO accounting infrastructure. We wroter a sample test application that maps a 1.2GB file and writes to 1GB of the mmap'ed memory. The output below shows the output of `/proc/<pid of test>/io`

```
rchar: 1261
wchar: 237
syscr: 6
syscw: 13
read_bytes: 1048592384
write_bytes: 1317117952
cancelled_write_bytes: 0
```

The gathered statistics indicate that the existing accounting of rchar and wchar present in CSA, failed to capture the IO that had taken place during the test.

The ideal place to account for IO, is the IO submission routine, `submit_bio()`. This works well for accounting data being read in by the task. However, for write operations, we need some place else to account for the following reasons

Writes are usually delayed, which means that it is hard to track the task that initiated the write. Furthermore, the data might become available long after the task has exited.

Write accounting is therefore done at page dirtying time. The routines `__set_page_dirty_nobuffers()` and `__set_page_dirty_buffers()` account the data as written and charge the current task for IO.

A task can also actually cause negative IO, by truncating pagecache data, brought in by another task. Instead of accounting for possible negative write IO, the data is stored in the field called `cancelled_write_bytes`.

Figures 1, 2, and 3 show the call flow graph for read, write, and truncate accounting, respectively.

Communicating the per task IO data to user space is fairly straight forward. It uses the per task taskstats interface for this purpose. The taskstats structure has been expanded to new fields corresponding to the per task IO accounting information. The taskstats interface automatically takes care of sending this data to user space when either a task exits or information is requested from user space.

4 Case studies of performance bottleneck analysis

The following two case studies, taken from a performance analyst who used delay accounting to debug performance issues, illustrate the utility of the mechanism.

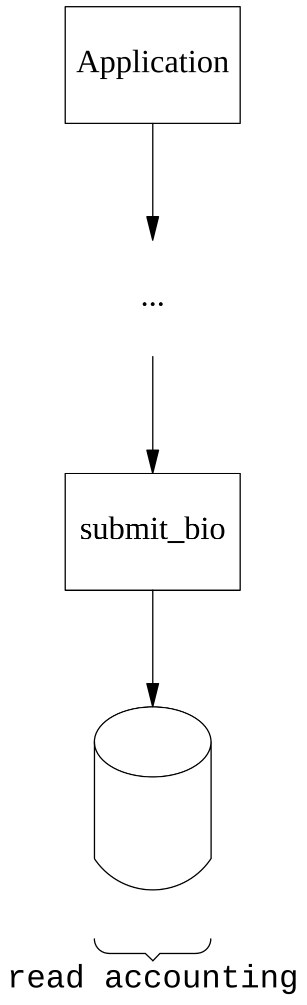

Figure 1: Read IO Accounting

1. Analysing performance issue with an MPI based parallel application:

   This is a brief on how per-task delay accounting was used to find out the root cause for a performance problem of an application running on homogeneous cluster. During the user acceptance test for the cluster this application successfully completed its run within the expected time. However some days later the same application was taking too much time to complete. Nothing changed in the cluster wrt to the configuration. However one strange thing was noted - when the cluster was compleletely isolated from the public network the application completed its run within the expected time. The problem happened only when the cluster was open for use by everyone concerned. This was an important lead but we didn't have any clue on how to proceed. The cluster was pretty large

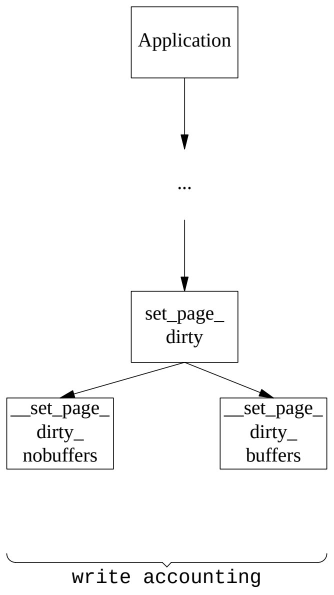

Figure 2: Write IO Accounting

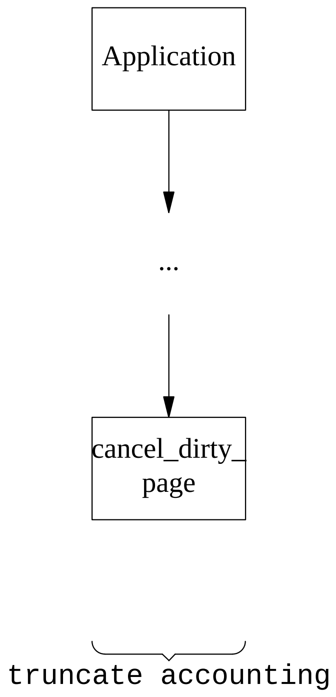

Figure 3: Truncate IO Accounting

and analysing each and every node manually was a tedious task.

Some of the possible things to look for were network communication delays, problems with the job scheduler's resource reservation functionality and system configuration of all the nodes (just to be sure that nothing was changed in between). Even after doing all these activities we were not able to identify the root cause of the problem. We didn't have a clue whether the issue was with any particular node/s or with the entire cluster.

Per-task delay accounting came to our rescue here. We asked the customer not to isolate the cluster and let it be used like in any normal day. The application was run on all the nodes in the cluster and subsequently the getdelays program was run so as to get the delay accounting statistics for this particular application. After completion of the run, the delay accounting statistics from all the nodes were compared and it was found that there was huge IO and CPU delay on one of the nodes in the cluster.

This was affecting the overall performance (execution time) of the application. We then focussed our attention on this particular node. Eventually after monitoring this node for some time, we found that one of the users was directly running interactive jobs on this particular node. This in turn pointed to a security hole in the customer's overall cluster setup which allowed this user to bypass the access control restrictions put in place and run the jobs directly on a particular node.

Per task delay accounting made the job of isolating the problem a lot easier. An added advantage is that of convincing the customer becomes a lot easier and effective.

Current per task delay accounting can be taken a step forward by including per task network IO delay accounting statistics also and writing a tool based on per-task delay accounting for identifying performance problems in a cluster.

2. There was another instance where per-task delay accounting helped us in dealing with a customer. The customer had a single threaded program on a non-Linux platform. The application was later made multi-threaded on the same platform. The program typically read X number of records in n seconds.

The customer ported the multi-threaded version of the program to Linux (using C programming language). The ported program was reading significantly less number of records in n seconds when compared to the original program on the non-Linux platform (hardware configuration was same, only the OS was different). Tools like top, vmstat, iostat were not very conclusive. Since source code access for the program was not there, we asked the customer to provide us with both single threaded and multi-threaded version of the program along with the timings and used per-task delay accounting to get the delay stats for both the single-threaded and multi-threaded version of the program. We found that for the multi-threaded version IO delay was on a higher side when compared with the single-threaded version. Pointed this out to the customer (as source code access was not there). Convinced them that they need to relook at the application code.

Per task delay accounting helped us to get specific data pertaining to the multi-threaded program in question.

5 Summary

Detecting the bottlenecks in resource allocation that affect an application's performance on a Linux system is an increasingly important goal for workload management on servers and on desktops. In this paper, we describe per-task delay accounting, a new functionality we contributed to the Linux kernel, that helps identify the resource allocation bottleneck impeding an applications forward progress. We also describe other important related work that improves the accuracy of CPU and I/O bandwidth consumption. Finally we demonstrate how these new mechanisms help identify where applications can get "stuck" within the kernel.