Governor: Autonomic Throttling for Aggressive Idle Resource Scavenging

Jonathan W. Strickland* Vincent W. Freeh* Xiaosong Ma*† Sudharshan S. Vazhkudai†

Abstract

Scavenging (or resource borrowing) is a common approach used to harness unused resources to perform useful calculations. Since these are volunteer contributions from resource owners, it is vital to reduce the impact of scavenging activities on their native workload to a minimum. To this end, existing impact control systems are either overly conservative in stopping scavenging altogether or inflexible and lack user autonomy to regulate resource usage as in some priority-based techniques.

In this paper, we propose a systematic impact control framework for resource scavenging, by quantifying *the performance impact a scavenging application incurs on a set of tasks stressing different system resources. For a user-configurable impact threshold, the framework monitors the native workload, determines the dominating native task, and autonomically and adaptively throttles the scavenging application, to bring the impact below target levels. This novel approach has unique benefits of 1) making impact control explicit to resource owners and an easy-to-tune "knob," and 2) adapting to different scavenging applications and native workloads. Our experiments with two scavenging applications, which use resources in very different ways, demonstrate that this framework allows both* more aggressive resource scavenging *and* less impact on native workloads *at the same time, compared to a priority-based method. Finally, the framework itself is a lightweight user-level process whose monitoring overhead on native workloads averages as low as 1%.*

*Department of Computer Science, North Carolina State University, Raleigh, NC, 27695-7534 {jwstric2,vwfreeh,xma}@ncsu.edu

†Computer Science and Mathematics Division, Oak Ridge National Laboratory {vazhkudaiss}@ornl.gov

1 Introduction

A typical personal computer is under-utilized most of the time [9, 19]. This has lead people to build *scavenger* systems, such as Condor [15] and SETI@home [2], to harvest idle resources. Such scavenging is very successful and desirable, for aggregating existing, distributed idle resources into massive compute power. Meanwhile, because these systems rely on good-will based contributions, their paramount concern is to have little or no negative impact on the workstation contributor or owner.

The impact of scavenging systems on resource owners is complicated and has implications on their computers' performance, storage, security, and privacy. In this paper, we focus on *performance impact control, i.e.*, to control and minimize the slow-down of a resource owner's native tasks by foreign, resource scavenging applications. Performance impact itself is a complex issue, involving both the scavenging application, the native workload and system resources on a particular scavenged desktop. As the native workload varies from computer to computer, and is typically dynamic on each of them, it is difficult for a fixed impact control strategy to incur minimum impact and yet manage to scavenge resources aggressively on all the scavenged computers.

One simple—yet safe—approach is to stop the scavenging application whenever user activity is detected on the scavenged system. However, as commonly recognized, this method over reacts. First, most native tasks can tolerate and co-exist with scavenging applications to a certain degree before deteriorating. In particular, Gupta *et al*. [12] showed through experiments of personal computer users that most of them, when performing a set of typical desktop tasks, do not feel obvious performance impact even when a significant amount of CPU, memory, and I/O resources are consumed by scavenging processes. Second, most personal computers have bursty work loads as users are often idle and frequently switch between different tasks. Stopping and resuming scavenging applications causes wasteful disturbance to the system, and has extra overhead to scav-

enging systems that choose to migrate the scavenging application to another machine (*e.g.*, Condor). The latter is especially expensive and sometimes undesirable for applications scavenging persistent resources such as disk space. Finally, in performing impact control this way, such as running the scavenging application in the screen saver mode (*e.g.*, SETI@home), a large amount of system idle time fails to be utilized because it typically takes minutes for the screen saver to be turned on.

Another commonly used method for performance impact control is based on assigning the scavenging application a lower scheduling priority [9, 19]. This approach is convenient to deploy, and automatically adapts to a native workload. Nevertheless, it has several limitations. First, priority-based methods work are only effective for traditional "cycle-stealing" systems that exploit idle compute power. Recently, there have been efforts in disk storage space aggregation [4, 8, 22]. A disk scavenging system behaves much differently than a CPU scavenging system, and causes a different impact to the same native workload. As we will show later in this paper, priority-based impact control does not work well for this type of scavenging system. In addition, for a given scavenging application and native workload, using a low priority for a scavenging process performs implicit, "best-effort" performance control, without observing the actual performance impact of this particular process on native workloads. Further, although priority adjustment is possible, it is confined by the range and increments offered by the operating system, hence sometimes not able to deliver the desired impact level or tuning granularity.

In this paper, we present a performance impact control framework, Governor,[1] that can seamlessly and autonomically restrict a scavenging application's resource consumption, and adapts to the ever-changing native workload. The main idea is to characterize the performance impact of a given scavenging application on a set of micro-benchmarks, each of which intensively uses one type of system resource, such as CPU, network, and I/O. Performance control is based on assigning the scavenging processes time slices to run, therefore inserting "slack" into their execution. For a mixed, dynamic native workload, our method monitors its activities and dynamically determines how much slack is necessary to *bound* the performance impact (to be formally defined in Section 3) of this workload within a given threshold.

We consider the major contribution of our work as follows:

- We proposed and implemented a novel framework for controlling scavenging induced performance impact. This framework systematically *quantifies* performance impact and resource restriction, and provides an execution framework to control a scavenging application's pace at an *arbitrary* level. Our impact benchmarking scheme allows this framework to accommodate diverse scavenging applications and native workloads at very affordable cost.

- We designed an *explicit* impact control method that derives the target level of restricting resource scavenging from a pre-specified impact level. This provides the basis for future user interfaces for impact feedback and fine tuning.

- We exploited user level impact control that does not require any kernel modification or administrator permission. It is operating system independent and can be easily dispatched along with a resource scavenging application. Further, the framework's monitoring overhead on native workloads is very minimal.

- We evaluated our impact control scheme using two types of scavenging applications, one CPU-intensive and one network/disk-intensive. Our results show that this new impact control scheme can successfully confine the actual performance impact within various specified levels, and works especially well for network/disk-intensive scavenging applications.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents our impact control model and methodology. Section 4 discusses implementation details and Section 5 presents experiment results. Finally, Section 6 lists potential future work and concludes the paper.

2 Related Work

An investigation very closely related to ours was by Gupta *et al*. [12], which studied real workstation owners' discomfort from impacted performance when resource scavenging applications run on their computers. In comparison, our work studies *objective performance impact*. We demonstrate that we can establish the correlation between resource usage throttling and measured performance impact. Although this objective impact

[1] Governor: A feedback device on a machine or engine that is used to provide automatic control, as of speed, pressure, or temperature. – dictionary.com

does not directly translate into *subjective user discomfort*, these two projects can be connected by GUI tools used in Gupta's work for resource owners to provide feedback and limit performance impact dynamically.

As mentioned earlier, one approach to impact control is to adopt a highly conservative mode of operation wherein resource consumption is stopped altogether upon user/process return (Condor [15], SETI@Home [2], and Entropia [9]). These systems scavenge CPU cycles and operate during intermittent intervals of user inactivity when the screen saver is activated. The basic premise therein is if the resource owner experiences the slightest discomfort they might withdraw their donations altogether. A variation is to have the scavenging process co-exist with resource owner's native workload by assigning it a low priority [14, 21]. However, as stated above, such a mechanism is limited to cycle stealing scavenging systems and has coarse-grained impact control. In contrast, our proposed framework provides a generic method for resource usage throttling for continuous and full-range impact control, while allowing aggressive resource scavenging.

Peer-to-Peer file sharing systems (Kazaa [18], *etc.*) are space and I/O bandwidth scavenging instances that extend resource borrowing beyond a conservative realm. These client programs run alongside native user workload to download and exchange files (in the background) with other peers in the Internet. However, this is made feasible due to a strong incentive of "exchanging data files of interest" as opposed to the "goodwill" based contributions in SETI-style systems. Impact control in such P2P file sharing networks has taken the following forms. First, due to its deployment across millions of desktops and potential to generate a lot of traffic, organizations employ tools such as PacketWise that tag P2P packets and limit their bandwidth [20]. This is very similar to QOS-based discrimination in networks and is performed at the router level [11, 24]. Alternatively, several P2P clients come with desktop tools with which resource owners themselves can specify higher level policies such as "maximum number of simultaneous downloads/uploads", "enable/disable sharing with other P2P client users", "do not function as supernode" or "maximum bandwidth to be used while transferring files" [18, 1]. However, such resource restrictions perform coarse-grained impact control only and lack dynamic adaptation capabilities. Our work explores adaptive fine-grained impact control, and can potentially be applied to existing P2P systems.

In a similar vein, Rate Windows [21] is a technique to limit disk and network I/O bandwidth consumption by throttling I/O usage based on job classes, albeit at the kernel level. Another approach to network I/O throttling is to perform rate-based clocking of network traffic by sending out packets at a preset interval, but requiring kernel-level modifications [6]. An additional approach to containing resource consumption for scavenging applications is through the use of virtual machines [9, 19]. While virtual machines are well known for sophisticated resource isolation through partitioning [3], concurrent execution, *etc.*, they also result in unsatisfactory performance, heavyweight implementations, and lack of performance guarantees [7]. Compared to such approaches, our proposed impact control framework is lightweight, operates at user level, and does not require the modification of either the scavenger programs or the operating system.

In addition, there are numerous projects on adaptive methods for controlling applications' resource consumption. For example, Odyssey used predictive resource management for dynamically choosing a quality of computation for wearable computers, so that a task's resource consumption can be bound by current supplies and its latency meets certain specifications [17]. Also, there is a wealth of projects on power-preserving computation that adaptively monitor applications' behavior and reduce power consumption (*e.g.*, [13]). In our work, we use resource restriction as a means for controlling resource owner perceivable performance impact, and aim at *maximizing* resource scavenging while meeting a given impact target.

There exists rich literature in the area of system resource monitoring. Well-known tools such as NWS [23], Remos [16] and dproc [5] monitor real resource information by deploying resource-specific sensors. However, these are highly geared towards an HPC or a distributed setting. In addition, tools like NWS and Remos also delve into predicting resource availability which might be onerous for our purposes. The DGMonitor [10] from Entropia monitors resource consumption and availability in a desktop Grid setting. It monitors several resource metrics on desktops periodically so that global job scheduling decisions can be optimized. While, we can derive significantly from the experience of several of the aforementioned monitoring schemes, our monitoring is intended for use by the local throttling system to limit resource consumption. In addition to the above, operating system tools such as *perfmon* provide a wealth of information on system performance counters that can be used for our purposes.

Finally, to the best of our knowledge, no research yet has taken a quantitative approach to systematic perfor-

mance impact control for resource scavenging systems.

3 Performance Impact Model

This section describes the overall model used in our impact control scheme. Before discussing the model and rationale for Governor's impact control strategy, we define several key terms used throughout the paper.

First of all, what is performance impact? Here we use simple metrics to define the performance impact: the slow-down factor. Suppose a set of tasks takes time $t_{original}$ to complete without a scavenging application running concurrently, and time $t_{scavenged}$ to complete with such an application running on the same workstation, the *performance impact* (or just "*impact*") is $(t_{scavenged} - t_{orginal})/t_{original}$. The goal of this work is to enable the self-configuration of a resource scavenging system to achieve a given performance impact, say 10% or 5%. Note that this objective impact metric does not immediately reflect the *resource owner perceived* impact or discomfort caused by resource scavenging, as this is a complex object involving issues such as the length, interactive nature, and frequency of such tasks, as well as the resource owners' sensitivity and personality. However, as discussed in Section 2 these two metrics can be connected by a user interface.

Next, in our system there are three entities. First, the *scavenging application* (also called "*scavenger*" for brevity) runs on distributed workstations and executes at the invitation of resource owners. The second entity is the *governor*, a process that monitors the machine and controls the scavenger. All other processes are grouped together and referred to as the *native workload*.

The goal of the governor is to limit the impact the scavenger has on the native workload to a desired level, while maximizing the throughput of the scavenger. Instead of assigning scavenging processes low priority, and relying on the operating system to schedule these processes unfavorably, we throttle the intensiveness of resource scavenging by inserting sleeping time between time intervals in which a scavenging process can execute. This reduces the demand on resources and hence reduces the impact on the native workload. The Governor framework performs fine-grain impact control by choosing and adjusting the ratio between "run time" and "sleep time". We define the *throttle level*, β, to be $(run\ time)/(total\ time)$. β varies from 0 to 1. $\beta = 0$ means the scavenger is not running at all, while $\beta = 1$ means the scavenger is running at full speed, without being slowed down.

The big question is: how do we find the appropriate β for a given impact level? We approach this through two mechanisms: *scavenger specific impact benchmarking* and *real-time native workload monitoring*, as outlined below. Implementation details will be presented in the next section.

Impact benchmarking helps us to characterize the effect of a scavenger on major resource types. Since native workload can be viewed as a combination of resource consumption components, we establish a *resource vector*, $R = (r_1, r_2, ..., r_n)$, where each r_i is a system resource, such as CPU, memory, disk bandwidth, network bandwidth, etc. We design a set of micro-benchmarks that stress each individual resource in R. Given a scavenger, it is executed at various throttle levels and the impact values on the micro-benchmark are recorded. Given enough data points, we can estimate the function $\mathrm{impact}_i(\beta)$. Suppose the box wants to restrict the maximum impact on any resource to a target impact level, α. The corresponding β to use in throttling the scavenger is determined as $\beta_i = \mathrm{impact}_i^{-1}(\alpha)$.

Now we know how to restrict the scavenger for a single-minded micro-benchmark. How do we decide the appropriate throttle level for a complex, and ever-changing native workload? We attack this problem through periodic monitoring of the native workload: for resources that have non-trivial native consumption detected, we activate the corresponding β. More formally it works as follows. A *trigger vector*, $T = (\tau_1, \tau_2, ..., \tau_n)$, is defined on the resource vector R. Each (τ_i) defines a threshold where the native consumption of resource r_i is considered non-trivial and a corresponding β_i needs to be activated. Note that here β_i has been computed for the target impact level α using the impact benchmarking results. The governor monitors the machine to determine the user activity, a_i, for each resource. Then it determines the effective β for each resource r_i:

$$\bar{\beta}_i = \begin{cases} \beta_i & a_i \geq \tau_i \\ 1 & a_i < \tau_i \end{cases}$$

The overall throttle level, β, is the smallest $\bar{\beta}$:
$\beta = \min(\bar{\beta}_1, ..., \bar{\beta}_n)$. This means that when the native workload is using two or more resources simultaneously, multiple throttling triggers will be turned "on", and the governor will choose the most restrictive throttling level to slow down the scavenger.

Note that our throttling is based on real-time periodic resource usage monitoring on the native workload. Therefore, if the performance impact brought by resource scavenging causes certain changes in the native workload's resource consumption pattern, due to the interdependency among tasks consuming differ-

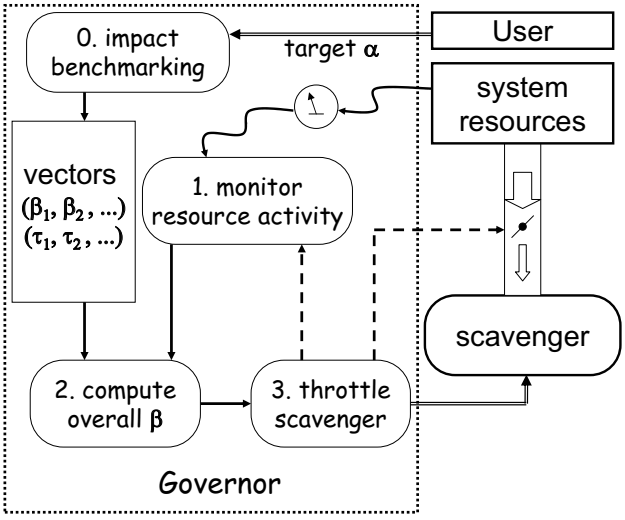

Figure 1. Governor architecture

ent resources, the governor will automatically adapt to such changes. However, by simply taking the most restrictive throttling level when multiple throttling triggers are turned on, we assume that this individual throttling level (measured using single-resource-consuming benchmarks) achieves the target impact level when the native workload is consuming multiple resources. Our empirical experiment results in Section 5 show that this assumption holds for the composite workload we tested with. Further, the function used to decide the overall throttle level, currently $\min$, can easily be replaced with a more sophisticated model if necessary.

Figure 1 depicts the Governor impact control framework. Note that Step 0 is likely to be performed when a scavenger is first installed in a donated workstation, while Steps 1-3 are periodically repeated whenever the scavenger is running. Details such as the frequency of executing this loop will be discussed in the next section. The dotted arrow from inside the Governor box to the "resource valve" shows that the Governor is able to control resource consumption implicitly, by assigning execution time slices to the scavenger.

The design of the Governor framework is unique in three ways. First, it is fine-grained and adaptive. A scavenger can effectively utilize idle cycles in very small bursts, whereas existing scavenging systems rely on long periods of user inactivity. Second, it builds a two-dimensional impact relationship between diverse scavenging applications and critical system resources. This enables a scavenger to be restricted in different ways when it collides with the native workload on different resources. Third, this framework is generic and extendible. For example, the "stop" strategy used by scavenging systems such as Condor and SETI@home can be viewed as a special case of Governor's strategy: where both the triggers (τ_i) and throttle levels (β_i) are 0s for all resources in R. Governor does not modify or analyze the internals of scavengers, making it trivial to handle new scavenging applications. Meanwhile, new resources can be easily accommodated by extending the resource vector and adding new impact micro-benchmarks.

4 Implementation Details

4.1 Scavenger Throttling Mechanism

Governor uses a straightforward scheme of restricting the execution time of a scavenger process. It operates in fixed, discrete *throttle intervals*, of I seconds. At a given throttle level of β a scavenger will execute for βI seconds.

The Governor entity itself is a user-level process. The scavenger is a child of this process. After forking the scavenger, the Governor executes the following steps every throttle interval. First, it unblocks the scavenger, and unblocks it after βI seconds. Then, it monitors machine activity twice, at the beginning and the end of the remainder of the interval, $(1-\beta)I$ seconds. Using these activity levels, it determines which resources are *active* based on their differences. Because the scavenger is blocked during this period, all the monitored activities belong to the native workload. Finally, it calculates the overall throttle level, which is the minimum β of the resources that have logged activity beyond their corresponding trigger level.

If no resources have triggered, then $\beta = \beta_{max} < 1$. Note that maximum β value has to be less than one, because Governor requires the scavenger to sleep for a certain period to perform accurate native activity monitoring. Therefore, if M is the minimum time we must monitor for user activity, then $\beta_{max} = 1 - M/I$. The throttle interval I needs to be long enough to reduce Governor's overhead, and short enough to react quickly to changes in the native activities. For this paper, we have fixed these values at $I = 1$ second and $M = 0.1$ seconds. Therefore, the maximum β is 0.9. We have tested that these are reasonable choices. However, the optimal values for I and M are open questions.

The Governor process sends signals, using the *kill* system call, to block and unblock the scavenger. It sends `SIGINT`/`SIGCONT` to block/unblock, and uses the *usleep* system call to delay the scavenger between

throttle intervals.

This prototype of Governor does not track the children of the scavenger. Future versions will suspend groups of scavenger processes, whether forked by the scavenger or by the Governor itself. We do not intend to deploy any devices to track ill-behaving scavengers, because a workstation owner should not host such scavengers.

4.2 Resource Usage Monitoring

In this section, we explain which system resources we choose to monitor, how we monitor them, and how we decide the triggers for them.

Our current prototype implementation of the Governor process monitors three resources, namely CPU, disk read, and network write. We believe they are the most important resource types, as explained below. However, more resource types can be easily added to the Governor framework.

We do not monitor disk write because most writes are asynchronous and the operating system is very good scheduling such actions "in between" synchronous reads. Therefore, a scavenger writing to a disk has little impact on a user workloads, as we found in our previous experiments [22]. Also, we do not monitor network read because we assume a scavenger will not download a significant amount of data. Memory usage is not monitored for three reasons. First, Gupta et al. [12] shows that memory usage has little user impact. Second, we believe there is a positive correlation between CPU and memory usage for memory-intensive tasks. By using such applications in impact benchmarking, throttling aimed at controlling CPU consumption will control memory consumption as well. Third, it is difficult to get an accurate measurement of memory usage for the native workload. We plan to monitor more resources if it becomes necessary or useful. However, at this point the three resources monitored appear to be sufficient for the scavenging applications we experimented with.

The Governor process collects CPU usage from */proc/stat*, which lists total cycles and cycles active since the system startup. These values are recorded at the beginning and the end of the monitor phase, from which we calculate the CPU utilization during the monitor phase. This reflects the amount of CPU activity from the native workload. To decide the trigger, τ_{CPU}, we measure an idle system and determine that the CPU utilization of system daemons and other background processes is less than 1%. We have hence chosen a trigger value of 1% CPU utilization. Although not included in this paper, experiments have shown that the performance of Governor is not significantly dependent on the CPU trigger level.

The disk read activity statistics can be obtained from */proc/partitions*, which shows the total number of blocks read since the system startup. Similarly, the network write activity statistics can be obtained from */proc/net/dev*, which shows the total number of bytes written since the system startup. Similar to the way we calculate CPU activity, we calculate the disk and network activity at the end of the monitor phases. The idle activity level for both the disk and the network is essentially zero. Therefore we set the τ_{IO} and τ_{net} to be 0 blocks and 0 bytes, respectively. In other words, any native disk read or network write activity will activate the corresponding β for throttling the scavenger.

4.3 Impact Micro-benchmarks

Finally, we discuss the impact micro-benchmarks used to characterize the impact of a specific scavenger on the resource vector. For each resource monitored by the Governor, there is a micro-benchmark code that stresses this resource *exclusively*, in so far as possible. Given these micro-benchmarks, the impact benchmarking process concurrently executes each of them and the scavenger. The scavenger is controlled by the Governor to perform at a variety of throttle levels. The outputs from the benchmarking tests are tables (one for each resource) that list the measured impact of the scavenger on each individual resource's micro-benchmark, at a series of β values.

The CPU micro-benchmark is EP from the NAS benchmark suite.[2] This test generates pairs of Gaussian random deviates. It is CPU-intensive, uses very little memory, and has no disk or network activities.

The I/O micro-benchmark simulates a large sustained read. A large sustained read size of 1GB was used to simulate an usually intensive case of a user's file retrieval from the disk.

Finally, the network benchmark simulates a user downloading a web page using *wget*. A single web page was accessed and downloaded from a server within the same subnet. Each page was requested hundreds of times back to back in order to make it cached by the web server but not by the requesting client.

[2] http://www.nas.nasa.gov/Software/NPB/

5 Experiment Results

This section presents our experimental results. In our experiments we chose two scavengers, SETI@home and FreeLoader, that consume system resources in dramatically different ways. SETI@home [2] is a well known resource-borrowing program that uses Internet-connected computers in the Search for ExtraTerrestrial Intelligence (SETI). This application is embarrassingly parallel, with almost no communication, and is hence ideally suited for such an execution environment. It is also very compute-intensive. A typical execution of SETI@home at a workstation will download KBs of data, and crunch numbers for hours. In contrast, FreeLoader [22] is a storage scavenging system that mainly consumes disk and network resources. It aggregates unused disk space for storing large datasets. Unlike systems such as SETI@home, where a piece of task can be executed on any idle machine, FreeLoader requires data serving be provided, where the requested data is persistently stored. Impact control is especially important for FreeLoader, as it is not feasible to refuse serving data or to migrate data away whenever any workstation owner's activity is detected.

First, we present the impact benchmarking results for SETI@home and FreeLoader respectively. We then validate Governor's impact control approach by measuring the actual performance impact of two workload samples comprising of desktop computer tasks. Our results show that the overall impact can be closely contained by deploying appropriate throttle levels dynamically. Finally, we show that the Governor framework outperforms existing impact control mechanisms in achieving two goals simultaneously: lowering impact on the native workload and improving scavengers' performance.

All tests were conducted on a Linux workstation running kernel version 2.6.1. The machine has a 2.8 GHz Pentium 4 with 512KB L2 cache and 512MB of DDR memory. The hard disk drive is 80G with a SATA interface. All experiments are repeated multiple times and we use the average results. Since small variances are observed, we omit the error bars.

A quick side note before we examine performance impact and scavenger performance results: the impact of the Governor framework itself is small. With the governor monitoring but no scavenger running, the impact on multiple native workloads averages about 1%, and was never more than 2%. We tested several different throttle intervals, as small as 0.5 seconds, and there was not a noticeable difference in terms of impact on the native workloads.

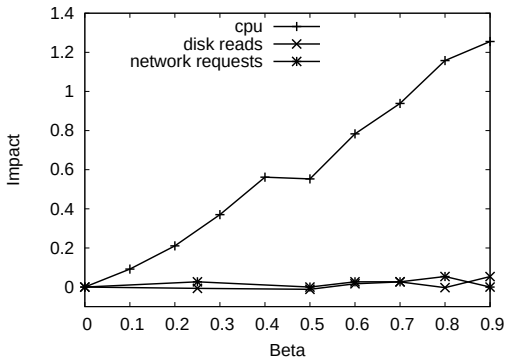

Figure 2. SETI@home impact on multiple micro-benchmark

5.1 Impact Benchmarking Results

This section presents results from the micro-benchmarks described in Section 4.3, yielding impact characterizations of the two scavengers on a set of system resources: CPU, I/O (disk reads), and network.

Figure 2 shows the impact curves for SETI@home. The *x*-axis shows the throttling level, β, where 0 means the scavenger process is not running and 1 means that the scavenger process is running at full speed along with a microbenchmark. The *y*-axis shows the performance impact measured for each micro-benchmark at these throttle levels. From this we can see that for SETI@home, heavy restriction is necessary in order to keep the impact on the CPU low: for a target impact level of 10%, we must use a β_{CPU} value of 0.05. However, the impact SETI@home has on I/O or network resources is always low, as expected. Therefore, both β_{IO} and β_{net} should be 1.

Figure 3 shows results of the same experiments for FreeLoader. As expected, these curves are very different from those of SETI@home. Here, for a 10% impact on CPU, the β_{CPU} value can be relaxed to 0.4, which is much less restrictive than with SETI@home. On the other hand, I/O and network are more restrictive here, for which a 10% performance impact requires a β_{IO} of 0.1 and a β_{net} of 0.2.

As discussed in Section 3, from the data collected in the above experiments we can derive corresponding βs for target impact level, α. In other words, given an α for a particular system resource, we select the βs that will throttle the scavenger to that impact level using the impact curves above. For example, suppose a user selects $\alpha = 0.1(10\%)$ for FreeLoader. We draw a horizontal

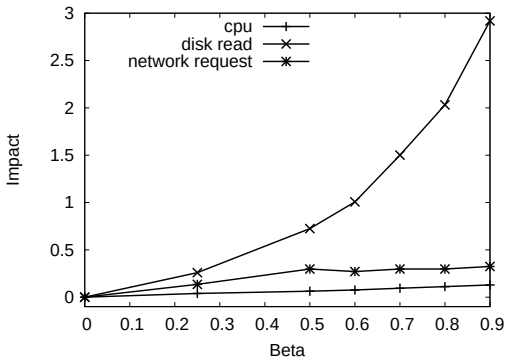

Figure 3. FreeLoader impact on multiple micro-benchmarks

Resource	Impact level α			
	0.05	0.10	0.20	0.25
β_{CPU}	0.02	0.05	0.10	0.2
β_{IO}	1.0	1.0	1.0	1.0
β_{net}	1.0	1.0	1.0	1.0

Table 1. SETI@home throttle levels at a series of given impact levels

Resource	Impact level α			
	0.05	0.10	0.20	0.25
β_{CPU}	0.30	0.4	0.70	0.90
β_{IO}	0.05	0.10	0.20	0.25
β_{net}	0.10	0.20	0.30	0.50

Table 2. FreeLoader throttle levels at a series of given impact levels

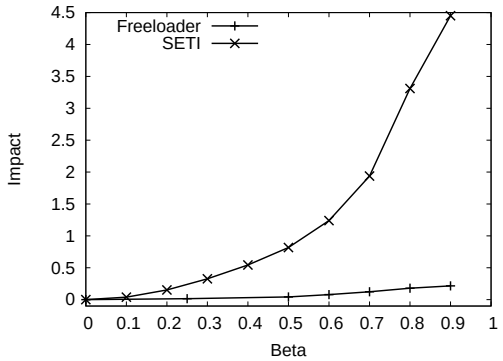

Figure 4. Impact of SETI@home and FreeLoader on kernel compile

line at 0.1 in Figure 3. The points where this line intersects the three curves correspond to the βs for each of these resources. Tables 1 and 2 show the βs determined this way from Figures 2 and 3, respectively.

5.2 Workload Tests

The micro-benchmarks only stress a single resource each, and are not representative user programs. This section shows the above impact benchmarking results applied to impact control for realistic user workloads. Therefore, we tested both FreeLoader and SETI@home executing alongside two user workloads.

Single-task workload. The first workload contains a single task: a kernel compile. This workload is interesting because it uses both the processor and the disk. Figure 4 shows the impact of scavenging on the kernel compile at a series of throttling levels. FreeLoader has very little impact on the kernel compile. Its greatest impact is only 30%. On the other hand, SETI@home induces impact of 4.5 on the kernel compile when unrestricted. This is mainly because SETI@home is CPU-intensive and will execute for hours without blocking on I/O. In contrast, FreeLoader is mostly performing I/O, reading from the disk and writing to the network. It rarely completes its assigned time slice before blocking on I/O, leaving more space for the kernel compile.

Now we take a look at how the scavengers themselves will be affected by the throttle levels, running with the kernel compile. As a companion to Figure 4, Figure 5 shows the performance of the scavenging processes during a kernel compile. The performance of FreeLoader and SETI@home are normalized to their maximum performance, which is achieved when the scavenger process executes unrestricted on the machine as a peer to the kernel compile, without the Governor. Both processes approach their unrestricted performance at $\beta = 0.9$, our maximum β value. Also, at $\beta = 0$ neither process executes, so the performance is 0 as well. FreeLoader's performance scales almost linearly, while the performance improvement of SETI@home accelerates as β grows. This figure shows that with gradually relaxed throttle level, a scavenger will at least steadily get more work done, which rewards using an aggressive β whenever allowed by the pre-specified impact level.

Measuring progress of SETI execution posed something of a challenge. Analysis of a data block takes sev-

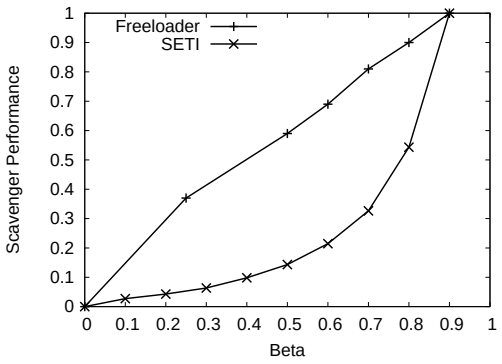

Figure 5. Normalized scavenger performance during kernel compile

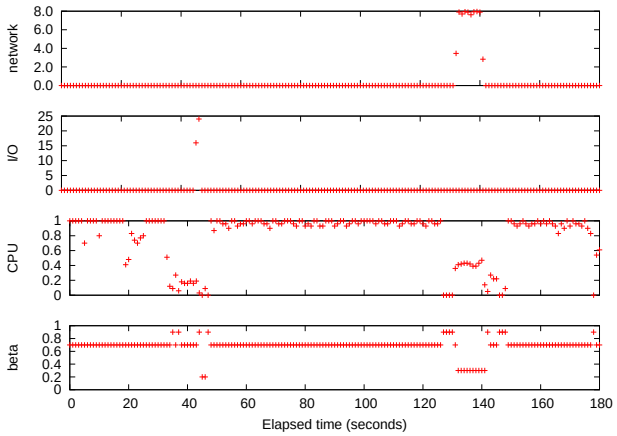

Figure 6. Time-line showing resource utilization levels and the corresponding β chosen by Governor

eral hours, significantly longer then the kernel compile. Initially, we tried measuring the CPU time that SETI@home receives. But the competition for cache and memory made that metric unreliable. In the verbose mode, SETI@home continuously outputs status messages. More lines of output means more throughput. However, the lines are emitted at irregular intervals. To solve this problem, we measured how many lines are emitted in each 10 second interval by SETI@home when executing it without any other processes running. Using this table, we convert the number of lines emitted in a test run into the number of *ideal seconds* that represents the time it took SETI@home to get that far in the best case. All SETI@home performance numbers presented in this paper are on these ideal seconds.

Composite workload. The second workload is to simulate typical user activities. To simulate common intermittent user activities, this synthetic workload sleeps for a short set period of time between 1-3 seconds between executing the following operations in order: 1) Writing 80 MB of randomly-generated data to files in a specific directory. This simulates unzipping a downloaded file into a local directory. 2) Browsing arbitrary system directories in search of a file. 3) Compressing the written data from the first part of the simulation with *bzip* into a file and transferring this file across the network to a data repository. 4) Browsing a few more local directories. 5) Finally, removing all data files written from the beginning of the simulation. This composite workload takes 148 seconds to complete without any other user load on the system.

With this synthetic workload designed to stress multiple resources in a short period of time, we can take a closer look at how Governor switches between different throttle levels dynamically as the workload progresses. Figure 6 shows the resource utilization levels and the βs used for a run of the composite workload with FreeLoader, for a target α value of 0.2. Each second along the *x* axis is a throttle interval, generating three resource utilization levels, as well as one β value to be used for the next throttle interval. For CPU, we show the utilization level, which varies between 0 and 1. For I/O, we show number of blocks read, which varies between 0 and 25. For network, we show number of MBs written, which varies between 0 and 8. Note that our current monitoring does not consider disk writes, which were observed to be unaffected by FreeLoader [22]. Also, the file system cache masks subsequent reads to the same data blocks. Therefore, I/O activities were only detected occasionally.

From Figure 6, we can clearly see that Governor switches between four different β levels, from high to low: the maximum β of 0.9 allowed by Governor when no triggers are on (the sleep times), β_{CPU} when the CPU trigger is on (true for most throttle intervals), β_{net} when the network trigger is on (around the 140 second mark), and β_{IO} when the I/O trigger is on (between the 40 and 60 second marks). The values of the resource-specific βs can be found in Table 2.

Table 3 shows the impact of our two scavengers at four target α values, along with the minimum and maximum αs represented as 0.00 and 1.0, respectively. We show both the measured execution time of the composite workload, and the performance impact calculated ac-

Scavenger	Impact level α					
	0.0	0.05	0.10	0.20	0.25	1.0
SETI@home	142	148	154	168	180	261
% impact	0%	4.0%	8.4%	18.5%	26.8%	83.8%
FreeLoader	142	150	157	172	180	211
% impact	0%	5.6%	10.6%	21.1%	26.8%	48.6%

Table 3. Overall execution time and impact on the composite workload

Scavenger	Impact level α					
	0.00	0.05	0.10	0.20	0.25	1.0
SETI@home	0	40	57	60	84	142
% of max	0%	28%	40%	42%	58%	100%
FreeLoader	0	2.94	3.56	5.77	6.66	6.92
% of max	0%	42%	51%	83%	96%	100%

Table 4. Scavenger absolute and relative performance (lines/s for SETI@home and MB/s for FreeLoader as absolute performance)

cordingly. The β values used can be found in Table 1 and 2. We have excluded the 6 seconds of total sleep time in all measured completion times, as obviously that time is not affected by the scavengers.

Table 3 verifies that the impact is indeed controlled by the Governor, and approximately contained within the target impact levels. Both SETI@home and FreeLoader exceed the target αs by less than 2% of overall impact. We are looking into two possible causes. One, the throttle interval might be too long. Two, we have only considered resources individually in our benchmarking. There may be some complex interaction between tasks consuming multiple resources simultaneously, in which case we need to have a composite β rather than simply picking the lowest one. Overall, these results validated that our scheme of selecting target βs with impact benchmarking and real-time workload monitoring works successfully.

Again, Table 4 shows the performance of the scavenger processes themselves catering to various α values running with the composite workload. We see from this table that with a relatively low α value of 0.1, SETI@home and FreeLoader can work at around half speed, compared to the unrestricted case. At a more restrictive α value of 0.05, they still get to progress at a pace that is 28% and 42% of their full speed respectively, performing significant amount of work rather than being suspended.

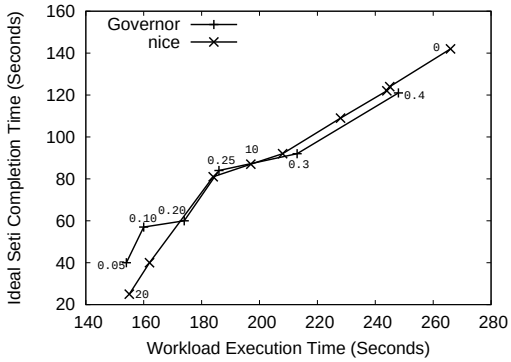

Figure 7. Scatter plot of impact and throughput for SETI@home.

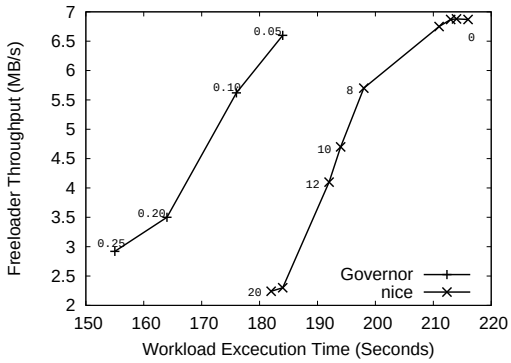

Figure 8. Scatter plot of impact and throughput for FreeLoader.

5.3 Comparison to Existing Methods

The Governor framework is in general more aggressive in resource scavenging than the traditional "stop" (*i.e.,* screen saver) impact control for scavengers like SETI@home. When SETI@home is active the workstation owner is not, Governor lets the scavenger run unrestricted, too. Governor also allows SETI@home to run when the workstation owner is idle but the screen saver has not been turned on. On the other hand, when the work station owner is active, the screen saver method generates both zero impact and zero scavenger performance. However, there is not an obvious way to compare the two schemes for the last case.

On the other hand, we can compare Governor with priority-based impact control schemes, such as using the command *nice* in UNIX. This section shows that priority is not as effective in controlling impact or extracting

performance as Governor. Figures 7 and 8 show scatter plots of user impact and scavenger performance at different throttle levels. For Governor this throttle level is of course the β value, while for nice this is the nice parameter indicating a given priority, specified when starting the scavenging process. The *x* axis shows the execution time of the composite workload. Closer to the origin is *less* impact. The *y* axis plots scavenger performance. Further from the origin is *more* scavenger throughput. So a point to the left and above another point is preferred for generating less impact to the native workload and yielding more scavenger performance. To make the labels more readable, we chose to mark only a subset of nice data points with their nice parameters.

Figure 7 shows the results for SETI@home. The nice point with the most impact (furthest to the right) has a nice parameter of 0, while the leftmost point has 20 (the lowest priority possible). In between are points for nice equal to 1, 2, 4, 8, 10, 12, and 16. At a nice value of 0, the scavenger is unrestricted, so this point also represents the Governor with $\beta = 1$. This plot shows that Governor and nice behave similarly at most impact levels, but Governor clearly out-performs nice when a small impact is desired (allowing much higher SETI@home throughput).

Figure 8, which plots FreeLoader results, tells a different story. In this case, Governor is by far superior to the priority-based nice. In fact, the entire Governor curve is to the above and left of the nice curve. Suppose a workstation owner is willing to have a 20% impact. That means the composite workload execution time (*x* axis) can be about 185 seconds. Draw a vertical line at 185, we will discover Governor yields more than twice the FreeLoader performance than nice, 6.5 MB/s versus 2.6 MB/s. Similarly, to get 3.5 MB/s out of FreeLoader, we draw a horizontal line at 3.5 MB/s and find the composite workload completes in about 165 seconds with Governor, 13% lower as compared to in 190 seconds with nice. Further, Governor can deliver low impact levels that nice cannot achieve in its capability range. This plot clearly shows that the Governor framework offers a much better solution than nice for I/O- or network-intensive scavengers such as FreeLoader.

6 Summary

In this paper, we have put forth the design and construction of a novel impact control framework for performance constrained execution of scavenging programs. This framework, Governor, serves as a means to address systematic impact control of resource usage, as well as to proactively consume idle resources, on contributed workstations. Given a user-configurable impact threshold, our framework monitors a scavenged workstation's native workload and autonomically throttles the scavenging program to bring its performance impact on the native workload approximately within the threshold. We demonstrated the effectiveness of our impact control framework for two different scavenging paradigms: cycle stealing, and disk-network I/O scavenging. Experiment results indicate that Governor can perform better than priority-based impact control for both scavenging paradigms. In addition, it is lightweight, operates at the user level and incurs an overhead as low as 1% on native tasks.

Our future work is comprised of designing interface knobs so that users can specify and tune impact tolerance levels, optimizing the throttle intervals and monitor intervals dynamically, evaluating Governor with popular instances of scavenging applications such as peer-to-peer file sharing programs, and memory usage throttling.

7 Acknowledgment

This work is supported in part by an IBM UPP award, the U.S. Department of Energy under contract No. DE-AC05-00OR2275 with UT-Battelle, LLC and Xiaosong Ma's joint appointment between NCSU and ORNL. The authors thank the conference reviewers and David Bernholdt at ORNL for reviewing the manuscript and providing many helpful suggestions.

References

[1] File sharing with peer-to-peer (p2p) applications. http://www.ncsu.edu/resnet/pages/p2p/-p2p.php/.

[2] Seti@home: The search for extraterrestrial intelligence. http://setiathome.ssl.berkeley.edu/, 2003.

[3] Planetlab: An open platform for developing, deploying and accessing planetary-scale services. http://www.planet-lab.org, 2005.

[4] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[5] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf. Resource-aware stream management with the customizable dproc distributed monitoring mechanisms. In *Proceedings of the 12th High-Performance Distributed Computing Conference*, 2003.

[6] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the 17th Symposium on Operating System Principles*, pages 232–246, 1999.

[7] P. Barham, B. Dragovic, K. Frase, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of SOSP'03*, October 2003.

[8] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of Supercomputing*, 2004.

[9] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5), 2003.

[10] P. Cicotti, M. Taufer, and A. Chien. Dgmonitor: A performance monitoring tool for sandbox-based desktop grid platforms. In *Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2004.

[11] T. Faber, L. H. Landweber, and A. Mukherkee. Dynamic time windows: packet admission control with feedback. In *Proceedings of SIGCOMM*, pages 143–135, 1992.

[12] A. Gupta, B. Lin, and P. Dinda. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, 2004.

[13] C. Hughes and S. Adve. A formal approach to frequent energy adaptations for multimedia applications. In *Proceedings of the 31st International Symposium on Computer Architecture*, 2004.

[14] P. Krueger and R. Chawla. The stealth distributed scheduler. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 336–343, 1991.

[15] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[16] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proceedings of the 7th High-Performance Distributed Computing Conference*, 1998.

[17] D. Narayanan and M. Satyanarayan. Predictive resource management for wearable computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[18] SHARMAN NETWORKS. The kazaa media desktop. http://www.kazaa.com.

[19] R. Novaes, P. Roisenberg, R. Scheer, C. Northfleet, J. Jornada, and W. Cirne. Non-dedicated distributed environment: A solution for safe and continuous exploitation of idle cycles. In *Proceedings of the Workshop on Adaptive Grid Middleware*, 2003.

[20] PACKETEER. Control peer-to-peer downloads. http://support.packeteer.com/, 2005.

[21] K.D. Ryu, J.K. Hollingsworth, and P.J. Keleher. Efficient network and I/O throttling for fine-grain cycle stealing. In *Proceedings of Supercomputing'01*, 2001.

[22] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader: Scavenging desktop storage resources for bulk, transient data. Submitted for publication, http://www.csm.ornl.gov/ vazhkuda/Morsels/.

[23] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference*, 1997.

[24] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proceedings of SIGCOMM*, pages 19–29, 1990.