Controlling Impact while Aggressively Scavenging Idle Resources

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

Abstract

As personal computers become ubiquitous, more and more idle resources are available at workplaces. Scavenging (or resource borrowing) is a common approach to harness unused resources to perform useful computation. Because resource owners donate these contributions, it is vital to minimize the impact of scavenging activities on these owners' native workloads. To this end, existing impact control methods are either overly conservative in stopping scavenging altogether (such as executing scavenging applications only in the screen-saver mode), or lack the flexibility and user autonomy in regulating resource usage (such as priority-based scheduling).

In this paper, we propose a systematic impact control framework for resource scavenging, by *quantifying* the performance impact a scavenging application imposes on a set of tasks stressing different system resources. The framework monitors the native workload and adaptively throttles the scavenging application to keep the impact below a user-defined impact level. This novel approach has unique benefits of (1) making impact control explicit to resource owners with an intuitive tuning "knob" and (2) automatically adapting to different scavenging applications and native workloads. Our experiments with multiple 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 induces less than 1% overhead on native workloads.

1 Introduction

Personal computers are pervasive in today's workplaces. While PC's applications and configurations have both been growing, a typical personal computer is under-utilized most of the time [8, 19]. This has led people to build *scavenger* systems, such as Condor [16], SETI@home [24], Folding@home [11], and Entropia [8], to harness idle resources. Such scavenging is very successful and desirable, for aggregating existing, distributed idle resources into massive computing power.

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

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

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.*, controlling the slow-down of native tasks due to concurrently executing foreign, resource scavenging applications. Performance impact itself is a complex problem, 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 one 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 approach overreacts. 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 users do not feel obvious performance impact when performing a set of typical desktop tasks, even when a significant amount of CPU, memory, and I/O resources are consumed by scavenging processes. Second, most personal computers have bursty workloads as users are often idle and frequently switch between different tasks. Suspending and resuming scavenging applications causes wasteful disturbance to the system, and has significant additional overhead to scavenging 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 [8, 19]. This approach is convenient to deploy, and automatically adapts to a native workload. Nevertheless, it has several limitations. First, priority-based methods are only effective for traditional "cycle-stealing" systems that exploit idle computing power. Recently, there have been efforts in disk storage space aggregation [1, 7, 26]. 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. Furthermore, 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 seamlessly and autonomically restricts 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 and network. The Governor throttles resource utilization by periodically inserting "slack" into the scavenging process, thereby reducing the time the scavenger competes with the native workload for resources. 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 have chosen to study the *objective* performance impact, *i.e.*, how much a native workload is slowed down by the scavenger process. This objective impact may not reflect the actual discomfort of resource owners caused by the scavenging. The *subjective* performance impact is a complex topic involving issues such as the length, interactive nature, and frequency of native tasks, as well as the sensitivity and personality of the individual resource owner. For example, a short editing task that normally takes 0.1 second may take 0.3 second when the personal computer is scavenged. This is 200% objective slowdown, but may seldom cause any user annoyance. In contrast, a 10% or 20% objective slowdown can cause significant discomfort when the personal computer owner is performing visualization tasks. Besides the obvious reason that objective performance impact is much easier to quantify and measure, we also believe that limiting the objective impact serves as a sufficient method for limiting the subjective impact. A resource owner may not perceive subjective performance impact when there is objective performance impact, but one will not when there is no objective performance impact.

We list the major contributions of our work below.

- We designed 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 a 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.

[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

- 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. Furthermore, the framework's monitoring overhead on native workloads is very minimal.
- We evaluated our impact control scheme using multiple types of scavenging applications: CPU-intensive, memory-intensive, and 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 is by Gupta *et al.* [12], which studied real workstation owners' discomfort due to performance impact from resource scavenging. 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 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 control 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 in the presence of any user activity (Condor [16], SETI@Home [24], and Entropia [8]). 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 [15, 23]. 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.

Many studies exist on job co-scheduling in parallel or distributed environments (*e.g.*, [4, 29]). Although existing work analyzes multiple workloads' resource usage patterns and optimizes the system's overall resource utilization, our framework is unique by controlling the interference *between* workloads in a quantitative manner.

Peer-to-Peer file sharing systems (Kazaa [14], *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 [21]. This is very similar to QoS-based discrimination in networks and is performed at the router level [10, 28]. 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" [14, 20]. 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 [23] 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 [3]. An additional approach to containing resource consumption for scavenging applications is through the use of virtual machines [8, 19]. While virtual machines are well known for sophisticated resource isolation through partitioning [22], concurrent execution, *etc.*, they also result in unsatisfactory performance, heavy-weight implementations, and lack of performance guarantees [6]. 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 the resource consumption of an application. 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 [18]. 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 [27], Remos [17] and dproc [2] 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 [9] 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 benefit 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 performance 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 the impact control strategy of the Governor, we define several key terms used throughout the paper.

First, we define *performance impact*. This project uses the slowdown factor. Suppose a task 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 the slowdown, defined as $(t_{scavenged} - t_{original})/t_{original}$. 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 mentioned earlier. However, Section 2 shows that 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) 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 a scavenging process a low priority, and relying on the operating system to schedule it unfavorably, we throttle the intensiveness of resource scavenging by periodically stopping the scavenger process—making it sleep for a short time. 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 time spent running and sleeping. 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 executes with no restrictions, as if Governor is not present.

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. Because a 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, disk bandwidth, *etc.* We use a set of micro-benchmarks that stress each individual resource in R. For each resource, we record the impact on the individual resource micro-benchmark due to concurrently executing a given scavenger process at various throttle levels. Given enough data points, we can estimate the function $impact_i(\beta)$. Suppose the Governor 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 $\beta_i = impact_i^{-1}(\alpha)$.

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, where β_i has been computed for the target impact level α using the impact benchmarking results. The trigger vector is determined for each workstation, while the β vector is unique for each scavenger. 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 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 apply the most restrictive throttling level.

Using the Governor, the smallest impact a scavenger has is when $\beta = 0$, *i.e.*, $\alpha_{min} = impact(0)$. For *transient* resources such as CPU and network, α_{min} is essentially zero. However, for *persistent* resources such as memory, α_{min} can be significant. Considering memory, a scavenger throttled down to 0 still owns its virtual memory. If there are too few free pages, the dirty pages belonging to the scavenger will have to be paged out of memory onto the disk to make room for the working set of the native application. The impact due to this paging can be significant on short running applications. Because α_{min} can be greater than 0, when a user requests an impact level less than α_{min}, the Governor will abort the scavenger process. Later this paper discusses special cases associated with

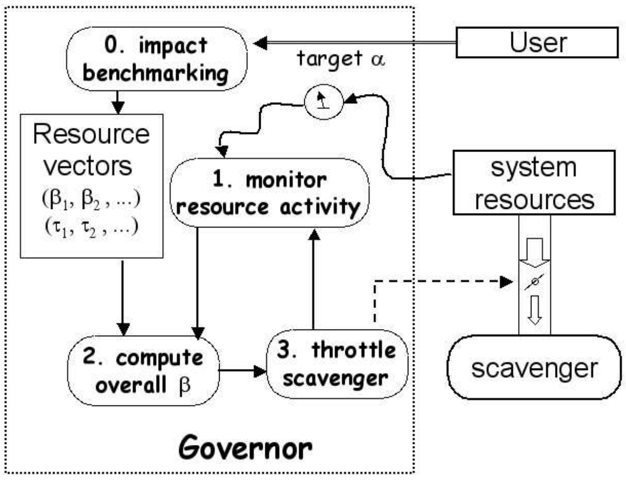


Figure 1: The Governor architecture.

the memory resource, where a β of 0 is insufficient to control the performance impact.

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. Results from our empirical experiments in Section 5 show that this assumption holds for the composite workload we tested. Furthermore, the function used to decide the overall throttle level, currently *minimum*, can easily be replaced with a more sophisticated model if there is complex interaction between resources.

Figure 1 depicts the Governor impact control framework. Step 0 is 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” indicates 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 extensible. 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 trigger (τ_i) and throttle level (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

This section describes the Governor implementation in three parts. First, it explains the implementation of the throttling mechanism. Then it describes how various resources are monitored. Finally, it describes the *micro-benchmarks* that characterize a specific scavenger resource usage.

4.1 Scavenger Throttling Mechanism

Governor uses a straightforward scheme of restricting the execution time of a scavenger process. The Governor operates in fixed, discrete *throttle intervals*, of I seconds. At a given throttle level of β a scavenger will execute for βI seconds. This scheme is preferred over using priorities because it precisely controls the resource usage. Moreover, as we show in Section 5.4, this mechanism is more effective than using priorities.

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 blocks 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 of the net activity. 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 $\beta = \beta_{max} < 1$. The 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 monitor period should be short to avoid overly restricting the scavenger, but long enough to detect native workload activity. Similarly, the throttle interval I needs to be long enough to reduce Governor's overhead, but short enough to react quickly to changes in the native activity. This paper uses the values at $I = 1$ second and $M = 0.1$ seconds. (Section 5.5 justifies these parameter values.) Therefore, β_{max} is 0.9.

The Governor process sends signals, using the *kill* system call, to block and unblock the scavenger. In particular, it sends `SIGINT/SIGCONT` to block/unblock. It also 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

The current prototype implementation of the Governor process monitors four resources, namely CPU, disk, network, and memory. It tries to determine how much the native workload is using each resource. These are the primary system resources. However, more resource types can be easily added to the Governor framework.

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 generally less than 1%. We have hence chosen a trigger value of 2% CPU utilization. Although not included in this paper, experiments have shown that the CPU utilization during the monitor period is almost always either less than or much greater than the trigger value. Therefore, the performance of Governor is not highly dependent on the CPU trigger level.

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

The Governor distinguishes between reading and writing to the disk. While Governor monitors disk writes, it is likely superfluous, 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 user workloads, as we found in our previous experiments [26].

The network activity is similar to the disk. Statistics are obtained from */proc/net/dev*, which shows the total number of bytes either sent or received since the system startup. The trigger, τ_{net}, is 0 bytes. We ran four sets of tests with both native and scavenger processes as either an intensive network reader or writer, while also varying Governor throttling. While the impact on the native process is slightly greater when both processes move data in the same direction, the difference is not significant. Therefore, the Governor does not distinguish between reading and writing to the network, rather it aggregates reading and writing on the network into one trigger.

Our initial prototype, introduced in [25], did not monitor memory usage for three reasons. First, Gupta *et al.* [12] shows that memory usage has little user impact. Second, 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, memory is not a scarce resource until all pages are allocated. Moreover, just because all pages are allocated does not mean all are in use, nor does it mean that there is contention for memory. Thus, it is hard to determine the actual memory pressure exerted on the system by either the native workload or the scavenger process.

However, we have found that scavenger memory usage can have an significant impact on memory-intensive native workloads, and our CPU-based monitoring and throttling along may be inadequate in handling memory-intensive applications. We have hence investigated incorporating the memory as a resource type in our Governor design and implementation.

Therefore, the latest Governor has a memory trigger. Scavenger memory usage becomes an issue only if all physical memory is being used, in which case the scavenger may cause extra paging in the native workload. Therefore, the memory trigger is based on the amount of free pages as reported by */proc/meminfo*. The current memory trigger, τ_{memory}, activates when the amount of free pages falls below 5% of main memory. This value has been sufficient for the experimentation conducted so far.

4.3 Impact Micro-benchmarks

This section discusses 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 that stresses this resource *exclusively*, in so far as possible. Given these micro-benchmarks, the impact benchmarking process concurrently executes the scavenger and each benchmark in turn. 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. These tables enable us to interpolate the $impact_i(\beta)$ functions and their inverses.

The CPU micro-benchmark is EP from the NAS benchmark suite [5]. This test generates pairs of Gaussian random deviates. It is CPU-intensive, uses very little memory, and has no disk or network activity. The I/O micro-benchmarks simulates a large sustained read and write, respectively. A 1GB file was used to simulate a usually intensive case of a user's file retrieval from the disk. The network benchmark simulates a user downloading a web page using *wget*. A single web page is accessed and downloaded from a server within the same subnet. Each page is requested hundreds of times back to back in order to make it cached by the web server but not by the requesting client.

Our memory micro-benchmark is a synthetic program that allocates and touches 125% of the main memory. It obviously is out-of-core and ensures that all main memory is used. Therefore, when the scavenger application is active, there will be an impact proportional to its memory usage. This benchmark represents a worst-case situation because it allocates a huge amount of memory, touches it only once, and performs no computation beyond what is needed to walk the memory.

Unlike with other resources, where cycles and bandwidths are transient, the impact due to a scavenger using memory depends on the number of dirty pages it owns in main memory. Writing back these pages may significantly slow down the memory access of a new native task. To capture this behavior, memory benchmarking executes the scavenger application (in the Governor) for some time before starting the micro-benchmark. This delay allows the scavenger to allocate and write to pages.

5 Experimental Results

This section presents our experimental results. These experiments feature two scavengers, SETI-@home [24] and FreeLoader [26], that consume system resources in dramatically different ways. SETI@home is a well known resource-borrowing program that uses Internet-connected computers in the Search for Extra Terrestrial 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 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 subtask can be executed on any idle machine, FreeLoader requires data serving be provided from where the requested data is stored. Impact control is especially important for FreeLoader, as it is not feasible to refuse serving data or to migrate data away whenever any native activity is detected.

First, we present the impact benchmarking results for SETI@home and FreeLoader with three resource types: CPU, disk, and network. Because we have found that the memory resource possesses certain unique impact behavior, the second section is devoted to benchmarking memory usage and its impact memory-intensive workloads. Third, we validate the Governor impact control mechanism by measuring the performance impact of two workloads. Our results show that the overall impact can be closely contained by employing appropriate throttle levels dynamically. Next, we show that the Governor framework outperforms existing impact control mechanisms in achieving two goals simultaneously: lowering impact on the native workload *and* improving the performance of the scavenger. Finally, we evaluate the system and justify the selection of the default parameter values.

Most 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 the average is displayed. When small variances are observed, error bars are omitted. The memory tests in

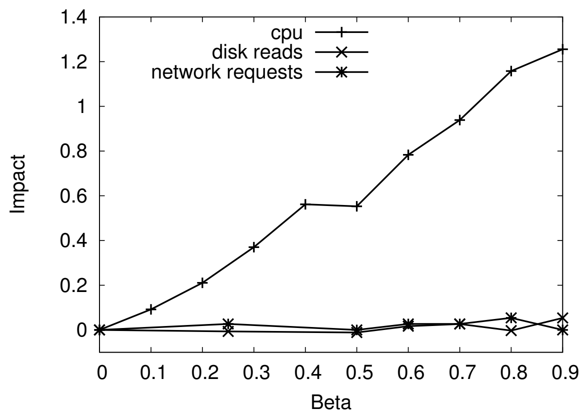


Figure 2: SETI@home impact on multiple micro-benchmarks.

	Impact level α			
Resource	0.05	0.10	0.20	0.25
β_{CPU}	0.02	0.05	0.10	0.20
β_{IO}	1.00	1.00	1.00	1.00
β_{net}	1.00	1.00	1.00	1.00

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

Section 5.2 were performed on an AMD machine with 1GB of DDR memory, a 2GHz Athlon-64, a 40 GB EIDE, and a 512KB L2 cache.

A quick side note before examining the 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%.

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. These scavengers do not use much memory or write to the disk. Therefore, we set $\beta_{IOwrite} = 1.0$ and $\beta_{memory} = 1.0$ for all values of α. In general, disk writing is not an issue, but memory usage can be a problem, as discussed in the next section.

Figure 2 shows the impact curves for SETI@home. The x-axis shows the throttling level, β. The $\beta = 0$ implies the scavenger process is not running at all, thus the impact is zero at that point. The y-axis shows the performance impact measured for each micro-benchmark at these throttle

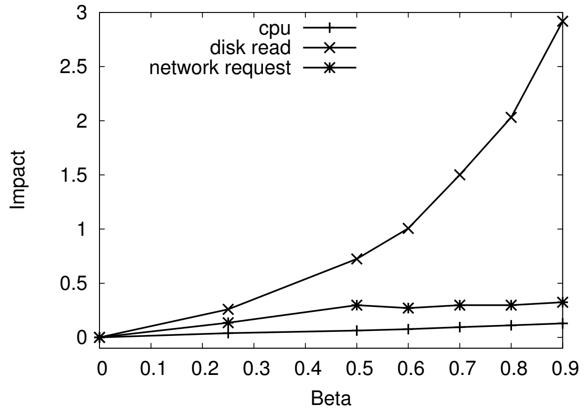


Figure 3: FreeLoader impact on multiple micro-benchmarks.

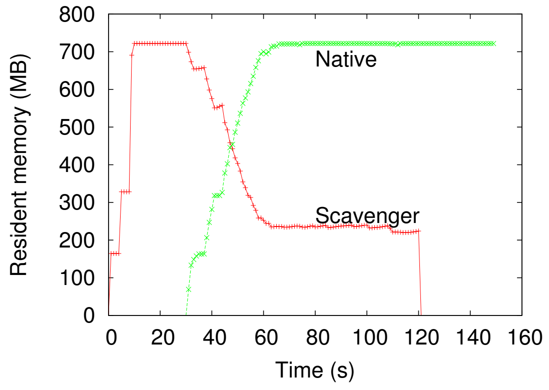
	Impact level α			
Resource	0.05	0.10	0.20	0.25
β_{CPU}	0.30	0.40	0.70	0.90
β_{IO}	0.05	0.10	0.20	0.25
β_{net}	0.10	0.20	0.30	0.50

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

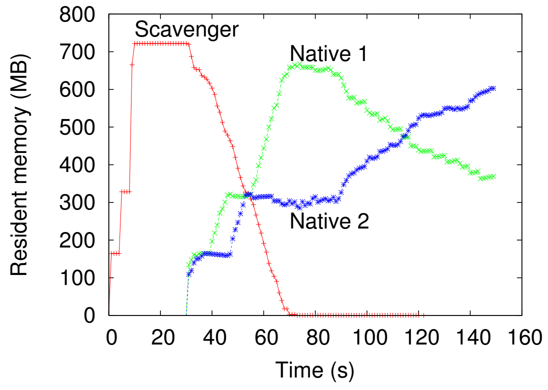
levels. From this we can see that for SETI@home, heavy restriction is necessary in order to keep the impact on the CPU low: for a target impact level of 10%, we must use a β_{CPU} value of 0.05, shown in Table 1. However, the impact SETI@home has on I/O or network resources is always low, as expected. Therefore, both β_{IO} and β_{net} are 1 for all α values. (As mentioned above, the β for writing is always set to 1 in our tests. Therefore, in this section β_{IO} implies the read throttle.)

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 interpolate $impact_i^{-1}(\alpha)$, allowing us to derive all βs for a 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 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



(a) Governor ($\beta = 0.1$) and native.



(b) Governor ($\beta = 0.1$) and two natives.

Figure 4: Memory tests using SP class C, with a throttle level $\beta = 0.1$. Plots show the resident set size (RSS) of each process over time. The scavenger SP process is started at time $t = 0$ and terminated at $t = 120$. The native workload, which consists of one SP process in (a) and two SP processes in (b), is started at $t = 30$.

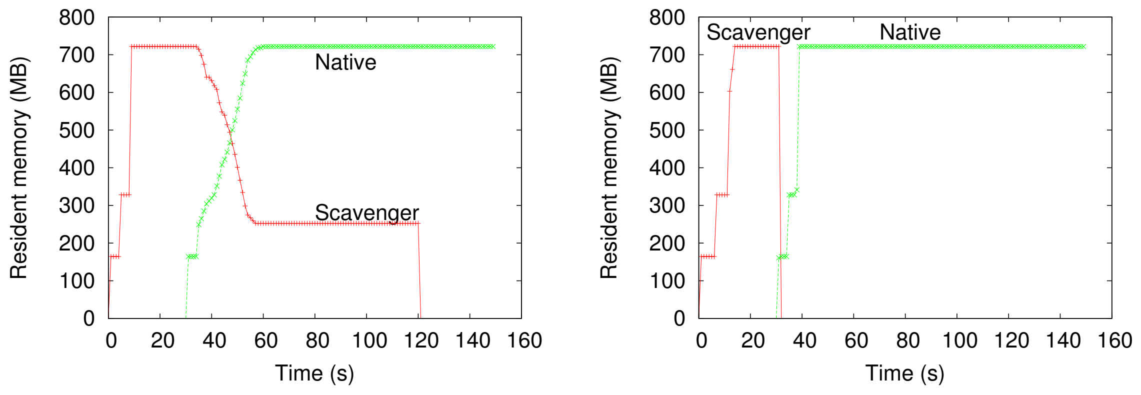
from Figures 2 and 3, respectively.

5.2 Memory Benchmarking

Gupta *et al.* argue that memory usage has little user impact [12]. Moreover, there is a positive correlation between CPU and memory usage for memory-intensive tasks because a task using memory also uses the CPU. The consequence of these observations is that it is not necessary to monitor and react to scavenger memory usage. Our experiments show that memory usage by the scavenger does have an impact on the native workload when (a) total memory usage is in excess of the main memory size and (b) the scavenger has a large number of dirty pages in memory. When these events occur, the impact due to scavenger memory use is very high. So contrary to popular opinion, scavenger memory usage must be monitored and controlled.

Before giving memory impact benchmarking results, we discuss memory contention in a scavenger environment. We use SP from the Class C NAS benchmarks because it fits in core, using approximately 75% of the main memory. SP is a computational fluid dynamics application that solves systems of equations resulting from a discretization of three-dimensional Navier-Stokes equations. Running alone, its resident set size (RSS) reaches 739MB out of the 1GB physical memory. Figures 4 and 5 show several plots of the SP memory usage over time in terms of RSS. The data for the RSS comes from the *stat* file belonging to the individual process in the *proc* file system.

In these experiments, we use SP both as the scavenger (controlled by the Governor using different throttle levels) and as the native workload. The scavenger SP process is started at time $t = 0$ and stopped after 2 minutes (before it is finished executing). It runs alone for 30 seconds,



(a) Governor ($\beta = 0.0$) and native.

(b) Governor ($\beta =$ abort) and native.

Figure 5: Memory tests using SP class C, with throttle levels $\beta = 0.0$ and $\beta = ABORT$ respectively.

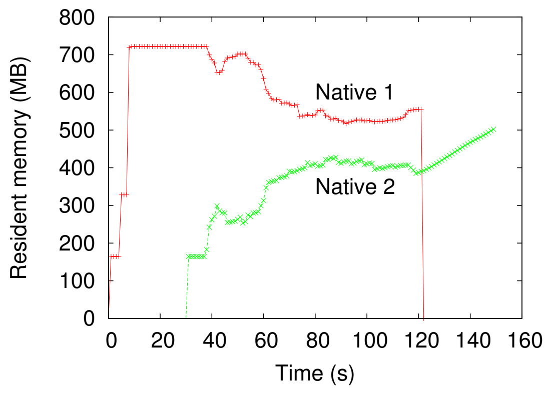
which is sufficient time for the application to copy its entire working set into main memory. At time $t = 30$ seconds the native SP process is started (in Figure 4(b) a third process is also started). After the RSS of the second process exceeds about 235MB, there are no free pages and the system begins swapping out pages belonging to the scavenger SP process. Both plots are quite similar up to this point.

This scenario is used because it represents the situation where a scavenger has its working set in memory when the native workload begins. Notably, in all tests where both the scavenger and the native process begin together the memory trigger is not activated because the CPU trigger activates first and throttles the scavenger, preventing it from acquiring too much memory. For example, in this situation as long as the throttle level is no larger than 0.3, the native SP keeps its working set in memory (*i.e.*, its RSS is 739MB).

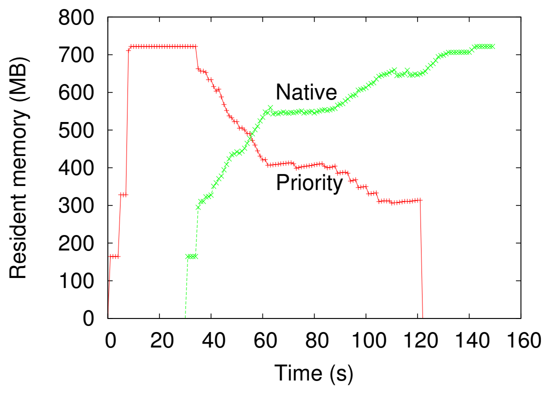
In Figure 4(a), the scavenger process runs with $\beta = 0.1$. As there is no native workload, this process rapidly acquires its full working set. The native process copies its working set into memory in 36 seconds, at $t = 66$. This task took the scavenger process only 8 seconds. Most of this extra time is due to swapping dirty pages to disk.

The next plot, Figure 4(b), shows what happens when the native workload needs all the physical memory. There are two native SP processes that need all the memory and more. The scavenger SP with $\beta = 0.1$ continuously releases memory so that by $t = 73$ it holds only 94 pages, or 376KB, of main memory. It holds these pages until it is aborted at $t = 120$.

The next two plots show the effect of the Governor being more restrictive. Figure 5(a) is similar to 4(a) except that the Governor throttles the scavenger process with $\beta = 0.0$. In this plot, it takes the native SP 29 seconds to copy its entire working set into memory. This is 7 seconds shorter compared to the situation in Figure 4(a), because the scavenger process is not allowed to use any CPU cycles, reducing its competition for the over-committed memory. However, even with the



(a) Two natives.



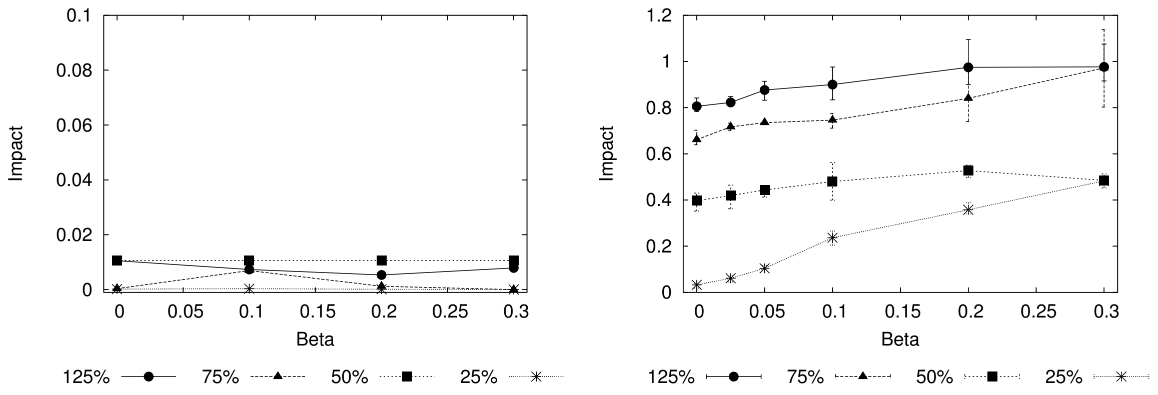
(b) Priority and native.

Figure 6: Memory tests using SP class C with priority control.

smallest β possible, the native workload still takes 4 times as long to grow its working set due to the copying of dirty pages belonging to the scavenger SP process, compared to when it is executed alone. In Figure 5(b), the scavenger is aborted when the native workload starts, at $t = 30$. Because the pages are immediately freed and not copied to disk, the native workload grows its working set in about 7 seconds. In other words, the native process does not suffer a significant performance impact.

The above four experiments reveal that, although a small β effectively controls the memory usage of a scavenger process, it fails to prevent the impact left by "legacy" memory pages when a memory-intensive native workload is started after a memory-intensive scavenger is in place. To deal with this, we add *ABORT* to the Governor's range of throttle levels as an extreme measure. Though harsh on the scavenger, aborting will prevent memory hogs from having an undesirable impact on the native workload. As can be seen from these figures, before the available memory gets low, the native SP process can quickly obtain memory pages (with an almost identical speed pattern as when SP is run alone). Therefore, the ABORT option, invoked by the memory resource trigger, timely terminates the scavenger before a native process experiences excessive performance impact.

Next, we complete the evaluation by looking at what happens in standard Linux with priority control. In Figure 6(a), both SP processes are executed as part of the *native* workload. These processes compete equally for the resources, and the second process never quite catches up to the first. In Figure 6(b), the first process, as the scavenger, executes at the lowest Linux priority, using the *nice* system with a parameter of +20. While the second process (the native process) has a much higher priority, it does not copy its entire working set into memory until 25 seconds after the first process is *terminated* ($t = 145$). In comparison, Governor's throttling, with or without the ABORT option, exerts far less memory pressure.



(a) SETI@home as scavenger. (b) SP C as scavenger.

Figure 7: Impact on the memory micro-benchmark caused by different scavengers.

The above tests show how memory is allocated to competing processes. Below we discuss the overall impact on the performance of these applications. First, we examine the worst-case situation with a synthetic memory-intensive program. The memory micro-benchmark is a simple program that allocates some memory and writes to each word (as *void **). Then it linearly walks the memory performing simple floating point operations on each element. Both the number of times through the memory and the number of operations are parameterized.

Figure 7 shows the impact of scavenging on four instances of this benchmark, allocating 125%, 75%, 50%, and 25% of the main memory (which is 1GB). The parameters were adjusted so that each instance takes a little more than 16 seconds with no scavenger running. The $\beta = 0$ point shows the results when the scavenger is throttled completely but still present. The y-axis shows the impact on the memory micro-benchmark. The scavenger process runs for 10 seconds before the micro-benchmark is started so that it will acquire memory pages. In Figure 7(a) the scavenger process is SETI@home. It has very little impact on the micro-benchmark, and the impact is independent of β in the ranges shown.

In Figure 7(b) the scavenger process is SP class C. The impact is much larger due to memory contention. Almost all additional time occurs in the phase of the benchmark that initially touches the memory. In this phase, the operating system is swapping pages to disk and the benchmark is waiting for dirty pages to be written to disk. In the three largest cases, the time in this phase increases by about 10 seconds. In the 25% case, this initialization phase increases by at most 1.2 seconds. The RSS of SP is about 75% of main memory, so there is sufficient memory to support both processes. Consequently, the impact of the 25% case is due almost entirely to CPU sharing.

Next, we look at the impact on SP class C (Figure 8). The impact here is primarily due to the increase in initialization time. The instance of SP we measured only runs for one iteration, which executes for 30.2 seconds, with 21.8 seconds of initialization. Note that the standard SP

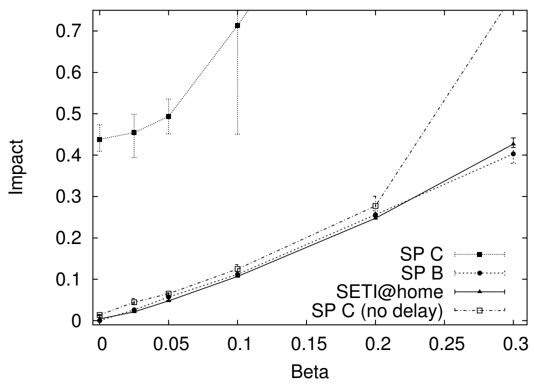


Figure 8: Impact on the SP C native process when running with various workloads

C benchmark executes for 400 iterations, taking nearly 10 minutes. In that case, the increase in initialization has little overall impact. Unaware of the potential length of a native workload, the Governor chooses to be cautious by utilizing the most sensitive memory benchmark to measure the throttle levels. An application that takes the time to create and initialize a large amount of memory will probably use that memory for some time. Therefore, this worst-case scenario that we guard against may not be an issue with real-world memory-intensive workloads.

We measured the impact on SP with four types of scavengers: two instances SP C itself, SP class B, and SETI@home. SP B uses one-fourth of the SP C RSS, and SETI@home uses very little memory, with an RSS of approximately 400KB.

The worst happens when the scavenger SP C gets started 10 seconds earlier than the native process, which causes the initialization time to double (the "SP C" line). The other three curves in Figure 8 are similar. These include SP B, SETI@home, and an "SP C (no delay)" line depicting an instance of SP C which is started at the same time as the native process. Because the working sets of SETI@home and SP B are small enough, they have moderate impact on the native workload. The "SP C (no delay)" curve shows that if the scavenger does not hold memory pages, its memory usage has far less dramatic effect compared to when the scavenger is started early. In fact, the scavenger process here never gets more than about 25% of the memory, which is essentially the amount the native workload is not using.

Finally, Table 3 reports the memory benchmarking results obtained using our memory micro-benchmark described in Section 4.3. Because both SETI@home and FreeLoader use little memory, the table lists only the results from SETI@home. The other two scavengers we benchmarked are SP B and SP C. Not surprisingly, since SETI@home has little impact on the memory benchmark, we get $\beta = 1.0$ for all values of α. The memory-intensive scavenger SP C, on the other hand, forces the β to be ABORT for any α. Between the two is SP B, which is allowed to run with a small β with the two more relaxed α values, but has to abort if the target impact level gets lower.

	Impact level α			
Scavenger	0.05	0.10	0.20	0.25
SETI@home	1.00	1.00	1.00	1.00
SP B	ABORT	ABORT	0.08	0.16
SP C	ABORT	ABORT	ABORT	ABORT

Table 3: Memory throttle, β_{memory}, required to maintain given impact level on memory micro-benchmark from three scavengers. β values are obtained by linear interpolation between measured values.

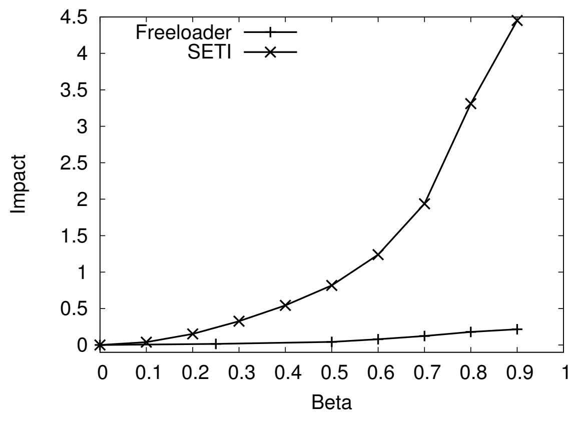
In summary, this section shows that a scavenger's memory usage can impact the native workload. It also demonstrates that Governor is effective in reducing the memory pressure and the resultant impact due to memory usage. However, because it takes time to page memory to disk, the only way to have no impact when both the scavenger and the native workload are memory-intensive is to abort the scavenger. This immediately frees the pages and obviates the need to page.

5.3 Workload Tests

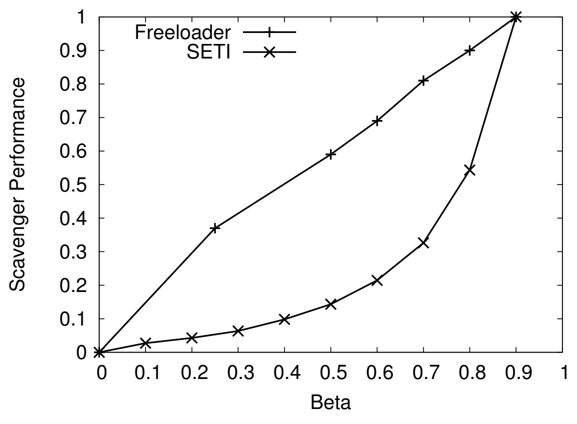
The micro-benchmarks only stress a single resource each, and are not representative user programs. This section shows the above impact benchmarking results applied to impact control for realistic user workloads. Therefore, we tested both FreeLoader and SETI@home executing alongside two user workloads. Since these sample user workloads are not memory-intensive enough, the memory trigger is never turned on and we limit our discussion to the CPU, disk, and network resource.

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 9(a) 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 nearly 4.5 on the kernel compile at $\beta = 0.9$. This is primarily 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, yielding the CPU to 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. Figure 9(b) 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



(a) Impact of scavenging on kernel compile.



(b) Normalized scavenger performance during compile.

Figure 9: Scavenging with kernel compile as native workload.

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 a 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@home poses a challenge. Analysis of a data block takes several hours; therefore, using completion time is cumbersome. Initially, we tried measuring the CPU time used by SETI@home. 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 based on these ideal seconds.

Composite workload The second workload simulates typical user activities. To simulate common intermittent user activities, this synthetic workload sleeps for a short set period of time (1-3 seconds) between the following stages: (1) writing 80 MB of randomly-generated data to files in a specific directory, 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; and (5) removing all data files written from the beginning of the simulation. This composite workload takes 148 seconds, with 6 seconds of idle due to *sleep*,

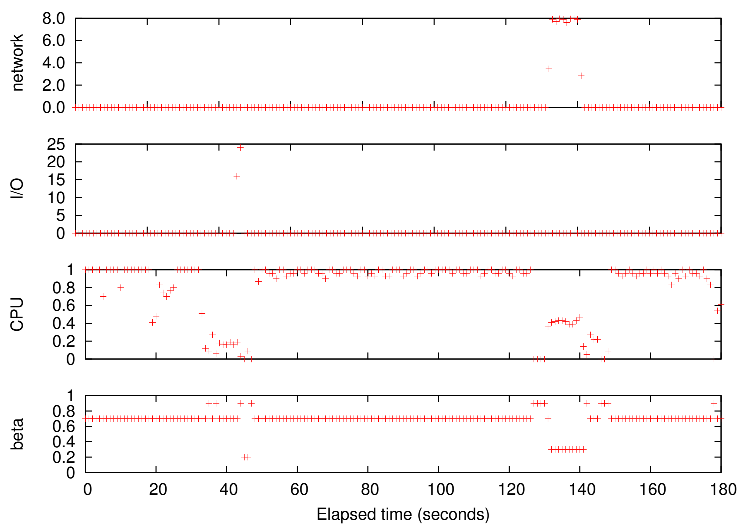


Figure 10: Time line showing resource utilization levels and the corresponding β chosen by Governor.

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 the Governor switches between different throttle levels dynamically as the workload progresses. Figure 10 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. The figure shows the measured value three resources and the β value for each 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. Neither disk writes nor memory are shown because they are never triggered. Thus, for simplicity, these triggers are not discussed below. Furthermore, the file system cache masks subsequent reads to the same data blocks. Therefore, I/O activities were only detected occasionally.

From Figure 10, 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, for $\alpha = 0.2$.

Table 4 shows the impact of our two scavengers at four target α values. Two additional columns are shown. The performance with no scavenger is labeled $\alpha = 0.0$ and that where the scavenger is an unrestricted peer is labeled as $\alpha = \infty$. We show both the measured execution time of the composite workload, and the performance impact calculated accordingly. The β values used can

Scavenger	Impact level α					
	0.0	0.05	0.10	0.20	0.25	∞
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 4: Overall execution time and impact on the composite workload

Scavenger	Impact level α					
	0.00	0.05	0.10	0.20	0.25	∞
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 5: Scavenger absolute (ideal seconds or MB/s) and relative performance .

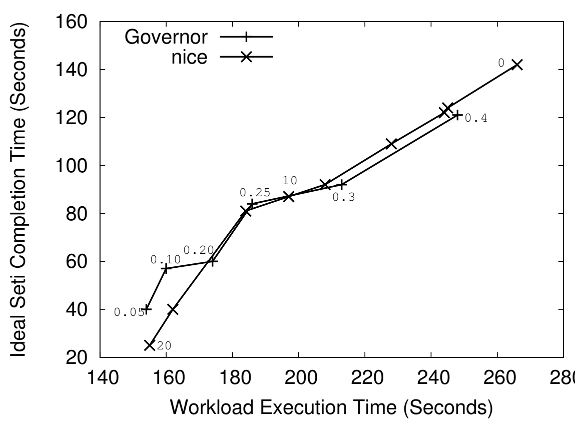
be found in Table 1 and 2. We have excluded the 6 seconds of total sleep time in all measured completion times, as obviously that time is not affected by the scavengers.

Table 4 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 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.

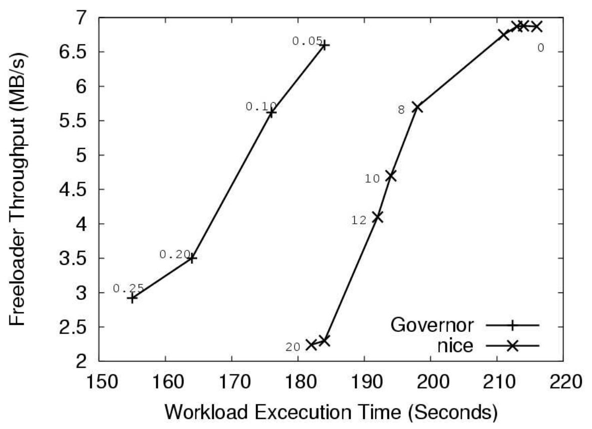
Table 5 shows the performance of the scavenger processes executing concurrently with the composite workload at various α values. We see from this table that with a relatively low α value of 0.1, SETI@home and FreeLoader can work at around half speed, compared to the unrestricted case. At a more restrictive 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.

5.4 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 nearly unrestricted. Governor also allows SETI@home to run when the workstation owner is idle but the screen saver has not



(a) Impact and time for SETI@home.



(b) Impact and throughput for FreeLoader.

Figure 11: Scatter plots of impact and performance.

been turned on. On the other hand, when the workstation 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 in these situations.

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 11(a) and 11(b) 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.

Figure 11(a) shows the results for SETI@home. For readability the origin is not show in this graph. 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 out-performs *nice* when a small impact is desired (allowing greater SETI@home throughput).

Figure 11(b), which plots FreeLoader results, tells a different story. The origin is also not show in this graph. 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% less time 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.

5.5 Evaluating the System

The Governor monitors the native workload for M seconds every interval of I seconds. This section performs tests to determine what the values for M and I should be. The monitor period needs to be long enough to detect and accurately determine the native workload. The length of the monitoring period does not effect the overhead of the Governor because there is always one period per interval. However, the scavenging process does not execute during the monitoring period, so a long period reduces the throughput of the scavenger process. Therefore, the period should be as short as possible.

Nearly every benchmark program we tested was *continuously* busy. Therefore, activity is detected by almost any monitor period length. Our simulated workload is not always busy. However, when it becomes busy it is active for a long period, so the activity is detected equally well at any period length.

To determine the proper monitor period, we executed a kernel compile for a couple of minutes. We monitored the native workload with the Governor but without a scavenger process. We recorded the number of times that a trigger was activated for different length monitor periods. The interval is kept constant at $I = 1$s, thus the number of events is independent of the length of the monitor period. Figure 12 shows the results using period lengths from 5 to 100ms; it plots the average and range of 5 runs. With a period of 100ms (and all tested periods of greater length) a trigger is activated every interval. At 5ms, only 83% of the intervals result in a trigger. This is because there was insufficient activity detected during that period. Most likely the compilation was blocked on I/O. A monitor period of at least 50ms is required and 100ms is preferred.

The standard benchmarks are insensitive to the interval length, I. The effect of a longer interval is that it takes longer to throttle the scavenger. In the worst case, this throttling is delayed $I - M$ seconds. Because I is on the order of seconds, this delay has little effect on a benchmark with a running time of minutes. Therefore, we created a simple, numeric kernel that only uses CPU resources and executes for a short duration. We executed this at 3 lengths. We then measured the slowdown of this “application” when our CPU-intensive impact test is executing in the Governor

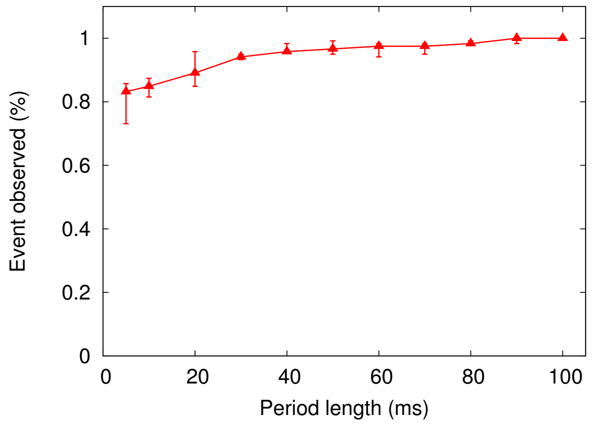


Figure 12: Events observed for different monitor periods.

with $\beta_{CPU} = 0.1$. The data is plotted in Figure 13. Each test consists of 20 runs of the small kernel. We delay between each run to make sure that the scavenger is not throttled, *i.e.*, $\beta = \beta_{max}$. Thus, each run begins execution concurrent with the scavenger process. The impact to the application is a function of the time it takes to throttle the scavenger, which increases as I increases. In these tests, the ratio $I : M$ is kept constant so that there is the same relative amount of throttling for each dataset. There are three datasets plotted for different length kernels. The time of the interval varies from 0.5 to 2.5 seconds. To bound the performance of the numeric kernel, the extreme left and right points in Figure 13 show time without the Governor. The length "0.0" interval is the time with no scavenger process (and no Governor) running. The length "inf" (for infinity) shows the time with both processes run as peers (but no Governor throttling the scavenger process). For all plots the right-most point is slightly more than twice the time of the left-most. However, the impact at finite interval lengths is greater on the shorter running kernels. At $I = 2$ seconds, the impact on the numeric kernel is 60%, 55%, and 33%, from shortest to longest. But at $I = 1$ second, the impact is approximately 25% for all. This figure shows that for short CPU bursts, we need a small interval. We use $I = 1$ second because the overhead is low. Although the slowdown of the shortest running numeric kernel is 26% it is only 111 microseconds.

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

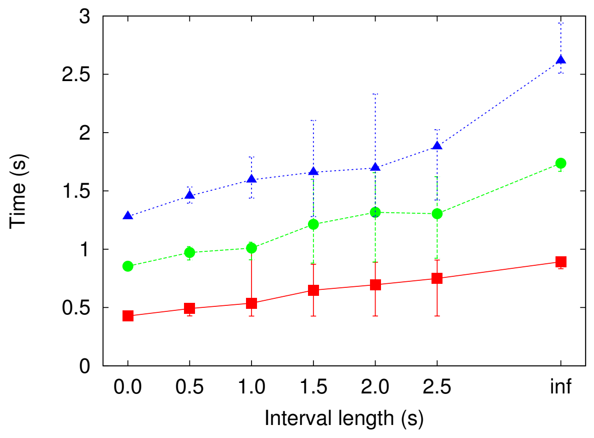


Figure 13: The impact of interval length on short programs, using three instances of a numeric kernel.

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 multiple scavenging paradigms: cycle stealing, memory-stealing, and disk-network I/O scavenging. Experiment results indicate that Governor can perform better than priority-based impact control for all three scavenging paradigms, in terms of the capability of controlling performance impact as well as aggressively scavenging resources. In addition, Governor 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, and evaluating Governor with popular instances of scavenging applications such as peer-to-peer file sharing programs.

7 Acknowledgments

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 reviewers.

References

[1] 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.

[2] 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.

[3] 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.

[4] A. Arpaci-Dusseau, D. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1998.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] Folding@home: distributed computing. http://folding.stanford.edu/, 2005.

[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] The Kazaa media desktop. http://www.kazaa.com.

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

[16] 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.

[17] 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.

[18] 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.

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

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

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

[23] 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.

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

[25] Jonathan W. Strickland, Vincent W. Freeh, Xiaosong Ma, and Sudharshan S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *International Conference on Automonic Computing*, pages 64–75, Seattle, WA, June 2005.

[26] 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/.

[27] 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.

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

[29] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. Impact of workload and system parameters on next generation cluster scheduling mechanisms. *IEEE Transactions on Parallel and Distributed Systems*, 12(9), 2001.