Fence: Protecting Device Availability With Uniform Resource Control

Tao Li[†]   Albert Rafetseder[†]   Rodrigo Fonseca[*]   Justin Cappos[†]
[†]*New York University*   [*]*Brown University*

Abstract

Applications such as software updaters or a run-away web app, even if low priority, can cause performance degradation, loss of battery life, or other issues that reduce a computing device's *availability*. The core problem is that OS resource control mechanisms unevenly apply uncoordinated policies across different resources. This paper shows how handling resources – e.g., CPU, memory, sockets, and bandwidth – in coordination, through a unifying abstraction, can be both simpler and more effective. We abstract resources along two dimensions of fungibility and renewability, to enable resource-agnostic algorithms to provide resource limits for a diverse set of applications.

We demonstrate the power of our resource abstraction with a prototype resource control subsystem, Fence, which we implement for two sandbox environments running on a wide variety of operating systems (Windows, Linux, the BSDs, Mac OS X, iOS, Android, OLPC, and Nokia) and device types (servers, desktops, tablets, laptops, and smartphones). We use Fence to provide system-wide protection against resource hogging processes that include limiting battery drain, preventing overheating, and isolating performance. Even when there is interference, Fence can double the battery life and improve the responsiveness of other applications by an order of magnitude. Fence is publicly available and has been deployed in practice for five years, protecting tens of thousands of users.

1   Introduction

Unfortunately, it is still common for end-user devices to suffer from unexpected loss of *availability*: applications run less responsively, media playback skips instead of running smoothly, or a full battery charge lasts hours less than expected. The cause might be a website running JavaScript [5, 6], a software updater overstepping its bounds [10], an inopportune virus scan [20, 27], or a file sync tool such as Dropbox indexing content [3, 7]. A buggy shell that leaks file descriptors can prevent other applications from running correctly [14]. Moreover, the cause may be unwanted apps or functionalities bundled with a legitimate application, such as advertisements [71] or hidden BitCoin miners [18]. Malicious applications may even attempt to overheat the device to cause permanent damage [54].

Although dealing with resource contention is a problem as old as multiprogrammed operating systems, today's platforms present renewed challenges. Key properties of a system such as battery life, application performance, or device temperature depend on multiple resources. This, along with the resource-constrained nature of mobile devices and the number and ease of installing applications from "app stores," exacerbates the problem.

In many cases, one cannot simply identify and kill processes that consume too many resources, because many programs execute useful-but-gluttonous tasks. For example, a web browser may execute inefficient JavaScript code from a site that slows down the device. However, the browser overall consists of many interrelated processes that render and execute code from different sites. Selectively containing resource usage is much better than killing the browser.

Our work addresses the problem of improving device availability in the presence of useful-but-gluttonous applications that can consume one or more types of resources. We introduce a non-intrusive mechanism that mediates and limits access to diverse resources using *uniform resource control*. Our approach has two parts: a unifying resource abstraction, and resource control. We abstract resources along two dimensions, allowing uniform reasoning about all system resources, such as CPU, memory, sockets, or I/O bandwidth. The first dimension classifies each resource as *fungible* (i.e., interchangeable, such as disk space) or *non-fungible* (i.e., unique, such as a specific TCP port). The second dimension classifies resources as either *renewable* (i.e., automatically replenished over time, such as CPU quanta) or *non-renewable* (i.e., time independent, such as RAM space). Using only these two dimensions, we are able to fully define a policy to control a specific resource. Adding a mechanism that quantifies and regulates access to each resource, either through interposition or polling, provides the desired level of control.

We demonstrate the feasibility of our approach by designing and building a resource control subsystem for sandbox environments, which we call Fence. Fence allows arbitrary limits to be placed on the resource consumption of an application process by applying a resource control policy consistently across resource types. This allows it to provide much better availability, when faced with a diverse set of resource-intensive applications, than would resource-specific tools. As we show in §5, when the system is under contention from a resource-hungry process,

our current user-space implementation of Fence provides double the battery life and a performance improvement of an order of magnitude relative to widely used OS-level and hardware solutions.

Fence is the first approach for controlling multiple resources across multiple platforms. By arbitrarily limiting resource consumption, Fence bounds the impact that applications can have on device availability as well as on properties such as heat or battery drain. Fence runs in user space, which makes it usable even within Virtual Machines (VMs) and on already deployed systems, where privileged access might be problematic and low-level changes could be disruptive.

2 Scope

Fence's controls work insofar as it can interpose on applications' requests for resources. As a proof of concept, Fence is targeted at application hosts, such as sandbox environments, VMs, browsers, or plugin hosts. The same unified policies could be implemented in an OS.

Most modern operating systems have mechanisms for identifying and limiting the resource usage of applications and processes [11, 15, 25]. However, as the prevalence of availability problems indicates, and we further demonstrate in §5, existing mechanisms do not adequately isolate resources among applications. A major cause of device availability problems is that there is no single resource where contention happens, which implies that any mechanism that controls a single resource will not be effective. Furthermore, because existing mechanisms use different abstractions and control points, it is very hard, if not ineffective, to write coordinated policies across different resources. Priority-based allocation and resource reservation systems (along with hybrids of the two) offer partial solutions in some cases, but the combined properties and guarantees are far from sufficient in practice. They apply ad-hoc and uncoordinated resource control across diverse resource types, and do not offer uniform functionality, abstractions, or behavior.

Recent developments in Linux's cgroups (cf. §7) provide a more unified approach to resource management, but are still restricted to one operating system. We implement Fence's resource control mechanisms – polling and interposition – at the user level, making it easily portable across many different platforms.

In this paper we assume that hog applications are "useful-but-gluttonous." That is, a hog application may attempt to consume a large amount of resources, but is otherwise desirable. Specifically, we do not attempt to protect against applications that perform clearly malicious actions, such as deleting core OS files, killing random processes, or installing key loggers (existing work addresses such issues). The desired outcome is that the user has control over properties such as the responsiveness, battery

#	Condition	Fungible	Renewable	Result
1	quantity $\leq$ quota	Yes	either	reduce quota, grant access
2	quantity $>$ quota	Yes	Yes	block until replenished
3	quantity $>$ quota	Yes	No	error
4	unallocated resource	No	either	allocate, grant access
5	busy resource	No	Yes	block until replenished
6	busy resource	No	No	error

Table 1: Accessing resources with different characteristics

life, and heat for any set of applications, especially hogs.

Our main focus in this paper is how to provide the needed *mechanisms* for resource-control, not on providing a specific resource control *policy*. As a first step, we demonstrate the effectiveness of Fence with two simple policies, both assuming that we can identify and have the privilege to control the hog process: one in which we manually provide static limits (most scenarios in §5), and one in which the policy sets dynamic resource limits to achieve a desired battery lifetime (§5.5). We anticipate, however, that more sophisticated policies are possible and applicable in other scenarios.

Lastly, a note about who interacts with Fence. Fence's mechanisms are integrated into a platform, e.g., a new sandbox environment, by the authors of the environment. Policies, on the other hand, can be written by the same authors, by third parties such as machine administrators, or by end-users. This paper does not specify higher-level interfaces for specifying policies. Fence should be transparent to applications running within the environment.

3 Managing Resources

In this section, we present Fence's resource abstraction, and discuss how using a simple classification of resources, along the dimensions of *fungibility* (§3.1) and *renewability* (§3.2), makes it possible to control their usage in a unified way. We assume a simple policy of limiting resource consumption by imposing a quota, given in terms of either an absolute quantity or a utilization.[1] Following that, §3.3 explains how control is enforced on resources, and §3.4 discusses choosing resource limit settings.

3.1 Fungible and non-fungible

One way to characterize low-level resources is by whether or not they are *fungible*, i.e., whether one instance of the resource can stand in for or indiscernibly replace any other instance. A fungible resource is something like a slot in the file descriptor table. It does not matter to the application that accesses a file where exactly the slot maps into the kernel's table. However, overconsumption of the resource can cause stability issues [14]. Fungible resources only need to be counted. The maximum allowed utilization by an application is capped by a quota. Fungible resources can be managed by maintaining, according to the resource's usage, an available quantity relative to a quota.

---

[1] As we show in §5.5, this quota needs not be static.

Lines 1, 2 and 3 of Table 1 summarize how gaining access to fungible resources works. As long as the requested resource quantity is within the quota, access is granted and the consumed quantity is logged. If a request exceeds the quota, granting access depends on the renewability of the resource (defined in §3.2 below), and will either fail or block until the required quantity becomes available. This will happen either over time (for a renewable resource), or through explicit release by the caller (for a non-renewable resource).

For a *non-fungible* resource, like a listening TCP or UDP port, each resource is unique. Each application may request one specific instance and will effectively block all other uses of this resource instance. Non-fungible resources need to be controlled on a system-wide basis. In most cases, non-fungible resources are assigned by Fence and reserved by applications, even when the application is not executing. Thus, even if a webserver application is not running, it may reserve TCP port 8080 to prevent other applications from using the port (closing a common security hole [83]).

Non-fungible resource access is summarized in lines 4, 5 and 6 of Table 1. A non-fungible resource that is not currently allocated can be accessed (after allocating it to the caller). Otherwise, access to a busy resource will either block or raise an error. Similar to fungible resources, non-fungible resources will either replenish over time, or are released explicitly by the caller.

3.2 Renewable and non-renewable

Some low-level resources, such as CPU cycles and network transmit rate, have an innate time dimension wherein the operating system continually schedules the use of the device or resource. These low-level resources are *renewable resources* because they automatically replenish as time elapses. Put another way, one cannot conserve the resource by not using it. If the CPU remains idle for a quantum, this does not mean there is an extra quantum of CPU available later. The control mechanism for a renewable resource is to limit the rate of usage, or utilization. Utilization is controlled over one or more periods, where the application's use of the resource is first measured and then paused for as long as required to bound it below a threshold, on average.

For example, to bound data plan costs, one may set a per-month limit for an application. Preventing the application's data usage from impacting the responsiveness of other network applications may also require a per-second limit. If an application attempts to consume a renewable resource at a faster rate than is allowed, the request is not issued until sufficient time has passed. (An extreme version of this involves batching requests together [17, 58, 76].)

Lines 1 and 4 in Table 1 show that access is granted when there is sufficient quota or unallocated resources. When a renewable resource is oversubscribed or busy (as shown in lines 2 and 5), it will block access until the resource is replenished, which will happen when the caller refrains from using the resource for an interval.

Non-renewable resources like memory, file descriptors, or persistent storage space are acquired by an application and (ignoring memory paging) are not time-sliced out or shared by other applications. As a result, granting access to a non-renewable resource is a conceptually permanent allocation from the application's perspective. For resources other than persistent storage space, the allocation usually coincides with the lifespan of the application instance, although the application may often choose to voluntarily relinquish the resource at any time. Short of forcibly stopping the application instance, there is little remedy for reclaiming most non-renewable resources once they are allocated.

To monitor and control the usage of non-renewable resources, it suffices to keep a table of resource assignments that track and update requests and releases. Once resource caps are set, an application can never request more non-renewable resources than the cap. A request can be met if the requested quantity is no greater than the cap on a fungible resource (see line 1 of Table 1), or, for a non-fungible resource, if the resource is not already allocated (line 4). Trying to exceed the cap will result in an error signal (line 3), as will trying to access an already allocated resource (line 6).

3.3 Enforcing Resource Controls

In the preceding discussion it was assumed that resource consumption can be measured and controlled in some way. While the restrictions Fence intends to place upon resources are fully specified by their fungibility and renewability, the method of enforcing resource controls is not, and will potentially vary across implementations of Fence, on different platforms. We describe two specific types of enforcement that allow Fence to control resources: (1) call interposition and (2) polling and signaling (indicated in Figure 1).

Call interposition. Interposition allows Fence to be called every time a process consumes a resource. However, there are different strategies that one may choose to control resources. For example, consider the case of restricting network I/O rate (a renewable resource). Suppose that a program requests to send 1MB of data over a connected TCP socket. The fundamental question is when Fence's control of this resource should be enforced.

Pre-call strategy. If an application is only charged for sent data before the call executes, then often times it will be overcharged. Given our example of a 1MB send, the send buffer may not be large enough to accept delivery of the entire amount and so much less may be sent. In such
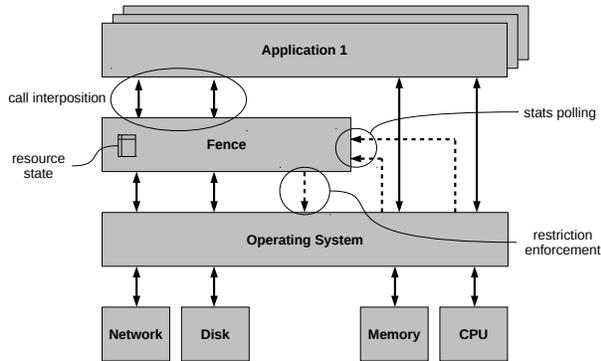
Figure 1: Fence's resource control via call interposition and polling/signaling.

a scenario, Fence would block the program for a longer time than is actually needed.

Post-call strategy. If the application is only charged after the call executes, however, it is possible to start threads that make huge requests to transmit information. This could allow the user to monopolize resources in the short term and to consume a large amount of resources.

Pre-and-post-call strategy. To better meter renewable resources, one can decouple the potential delay of access (done to adjust utilization based on past usage), from accounting, which is best performed after access. With this method, the pre-call portion of control will block until the use from prior calls has replenished; after that, the function is executed and the post-call portion charges the consumed amount.

Micro-call strategy. Finally, another option is to break a large call up into smaller calls and then allow Fence to handle them separately. However, this has the drawback of changing the original call semantics in some cases. For example, suppose that an application is only permitted to send 20kB of data per second. However, the caller wants to send a UDP datagram of 60kB. Instead of sending one 60kB datagram, the application could send three separate 20kB datagrams. Since each datagram is a "message" and the loss / boundaries between messages are meaningful (unlike the stream abstraction of TCP), sending three 20kB datagrams has a different meaning than does sending one 60kB datagram.

In practice, Fence is largely used with the *pre-and-post-call* strategy for renewable resources. For non-renewable resources, the *pre-call* strategy is predominant, because it will block access to an unauthorized resource before it occurs. Other strategies can be used, depending on how Fence is integrated into a platform.

Polling and signaling. For resources on which it cannot interpose, Fence uses polling, coupled with some form of limit enforcement, such as signaling, as a complementary mechanism to control resource usage. For example, for CPU scheduling, Fence does not modify the OS; rather,

the OS scheduler manages process scheduling directly. Fence needs to poll to understand how often a process has been scheduled and must signal the scheduler to stop or start executing the application. This has a number of implications [80, §5.5.3], [78, §18.3] that call interposition does not present.

Atomicity. While trivially implementable for call interposition (e.g., by guarding calls with semaphores), resource access is not guaranteed to be atomic. This raises potential Time-Of-Check To Time-Of-Use (TOCTTOU) issues between threads of the same application as well as other applications wanting to access the resource.

Interference due to load. Fence's polling and signaling impose a certain amount of overhead that depends on the rate of checking and control. When the machine is under load, Fence might not be able to keep up with its planned check and control schedule. As a result, a process under its control can consume a resource for longer than the time scheduled by Fence. Thus, in the worst case, this causes overconsumption and reinforces the overload condition.

Rate of checking and control. The fidelity and overhead of resource control depend on the rate at which control is enforced. The minimum rate (i.e. the longest interval between interruptions) is given by the minimal fidelity desired. If the rate is too low, a process might overspend a resource between checks. The maximum rate is bounded by the maximum acceptable overhead – each check and interruption causes additional cost – and the granularity of checking and control (which defines the minimum possible length of interruption).

Polling and signaling granularity. Granularity is the smallest unit of measurement and control of a resource, e.g., the resolution of the system clock, or the unit job that control functions can handle (process, thread). If the granularity is too coarse, Fence might misapprehend the resource consumption of a process under its control during polling or enforce undue restrictions on the process.

3.4 Choosing Resource Control Settings

A question that follows from the previously discussed framework is how to choose which resources to allocate to a process. This can be viewed along two axes. First, policies can set direct per-resource usage limits or quotas, or indirectly establish limits based on the effect of resource usage on device availability. Second, such limits may be static, set manually and *a priori*, or dynamic, where policies continually adjust limits to achieve availability goals[2].

For most non-fungible resources (like TCP / UDP port

---

[2]We also envision that different stakeholders desire different policies: For example, a programmer could include Fence in their application in order to tame it; a sysadmin may specify policies based on user groups, time of day etc.; the end user sets values with respect to their current workload and usage requirements.
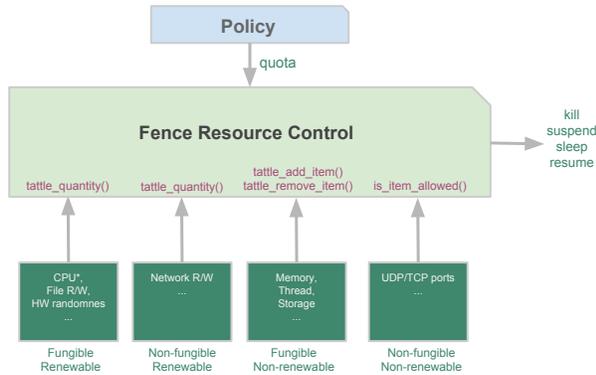
Figure 2: Fence's uniform control APIs for resources based on fungibility and renewability

```
def sendmessage(destip,destport,msg,localip,localport):
  ...
  # check that we are permitted to use this port...
  if not fence.is_item_allowed('messport',localport):
    raise ResourceAccessDenied("...")

  # get the OS's UDP socket
  sock = _get_udp_socket(localip, localport)

  # Register this socket descriptor with fence
  fence.tattle_add_item("outsockets", id(sock))

  # Send this UDP datagram
  bytessent = sock.sendto(msg, (destip, destport))

  # Account for the network bandwidth utilized
  if _is_loopback_ipaddr(destip):
    fence.tattle_quantity('loopsend', bytessent + 64)
  else:
    fence.tattle_quantity('netsend', bytessent + 64)
  ...
```

Figure 3: Fence additions to the Seattle sandbox's sendmessage call. Added lines of code are in bold text.

numbers) or for items that are hard to quantify (like privacy, cf. §6), choosing appropriate resource settings is typically done manually during packaging or installation.

However, the relationship between a limit (say, 30% of CPU or disk bandwidth), and user-observable properties (such as response time, battery life, or heat) is non-obvious and dependent on a number of factors. Accurate resource modeling is very difficult and in general requires advanced techniques [35, 45, 68, 75, 89], but determining a workload that approximates the maximum negative impact is an easier task.

Fence provides the capability to automatically choose resource limits, in particular, for fungible resources. When Fence is installed, it can benchmark the platform and check the usage impact on different resources, such as performance degradation, battery usage, or temperature. By understanding the impact of individual resources, one can choose resource settings that provide device availability in the face of contention. In §5.5 we show a policy that sets limits both indirectly and dynamically, based on a target battery lifetime.

4 Implementation

Fence's fully functional open source implementation that runs across various hardware platforms and OSses includes the features and functionality described in the previous sections. The core implementation is 790 lines of Python code (as counted by `sloccount`). There are two major portions of the Fence code: the uniform resource control code (140 LOC) and the operating system specific code (650 LOC).

4.1 Uniform Resource Control

Because the uniform resource control code only differs in its characteristics of fungibility and renewability, it fits within 140 LOC. Fence is informed about resource consumption by performing a set of four calls as is shown in Figure 2.

The specific call made depends on the type of re-

source being consumed. For example, the Seattle sandbox (described in more detail in § 6) required an additional 79 lines of code to support Fence, 68 of which were direct calls to Fence Figure 3 shows some of the code changes made to `sendmessage`, Seattle's API call for sending UDP datagrams. The first call, `is_item_allowed()`, checks whether the UDP port (a non-fungible, non-renewable resource) can be consumed. The `tattle_add_item()` call is used to charge for entries in the socket descriptor table to prevent the kernel from being overloaded with active sockets. The `tattle_quantity()` call charges for the consumed bandwidth (a renewable resource), depending on the destination interface.

In addition, there is an API that can be used to set high level policy. This is done by setting low-level resource quotas for the different resources that Fence manages. For example, if energy is the primary concern, resource quotas can be set to restrict the maximum expected energy consumption over a polling period. Actual energy consumed can be measured and the resource quota values updated appropriately.

4.2 Operating System Specific Code

The bulk of the Fence code (650 LOC) is operating system specific and involves supporting polling or enforcement across various platforms, including Windows XP and later, Mac OS X, Linux, BSD variants, One Laptop Per Child, Nokia devices, iPhones / iPods / iPads, and Android phones and tablets.

While all these platforms have the necessary low-level functionality, a convergence layer is still required because polling and enforcement semantics differ from platform to platform. For example, resource statistics are gathered in a fundamentally different way across many types of devices. However, the operating systems for many platforms

are derived from similar sources and thus share some subset of resource control functionality. For example, the Android port of Fence is based on the convergence layer of the Linux implementation of Fence and reuses almost all of its code.

4.3 Operating System Hooks Utilized

At the lowest level, Fence polls OS specific hooks for performance profiling to obtain statistics about the resource consumption of an application. To gain control over scheduling, Fence uses the job control interface on most OSes (SIGSTOP and SIGCONT), or the `SuspendThread`/`ResumeThread` interface on Windows. This informs the scheduler when to suspend or continue a process, in order to impose a different scheduling policy than the underlying OS scheduler. This can give a program less CPU time than it would ordinarily have to limit its performance impact, rate of battery drain, or heat.

To control resources other than the CPU, Fence constructs a uniform interface to low-level functionality that operates in an OS-specific manner. Consider memory as an example: On Linux and Android, Fence can use `proc` to read the memory consumption (Resident Set Size) for the process. On Mac OS X, similar actions are performed by interfacing with `libproc`. On Windows, calls to `kernel32` and `psapi`, the process status API, reveal the required information. Inside of its resource measurement and control routines, Fence can then use a single high-level call to gather the memory usage of a process, no matter what underlying OS or platform is used.

5 Evaluation

We evaluated Fence's software artifact and its deployment to investigate the following questions.

In situations with resource contention, how effectively do uniform resource control and legacy ad-hoc techniques provide device availability? (§5.2)

How well does uniform resource control function across diverse platforms? (§5.3)

How much overhead is incurred when employing uniform resource control during normal operation? (§5.4)

Can realistic, high level policies be expressed with Fence? (§5.5)

How diverse are the resources that can be metered by uniform resource control? (§6)

How time consuming and challenging is it to add a new resource type to Fence? (§6)

5.1 Experiment Methodology

To understand the tradeoffs between customized solutions and uniform resource control, we compared Fence to well-known, deployed tools found on common OSes. These included `nice` (which sets the scheduling priority of a process), `ionice` (similarly for I/O priority), `ulimit` (which imposes hard limits on the overall consumption of resources like file sizes, overall CPU time, and stack size), as well as a combination of these tools. We also included `cpufreq-set`, which changes the CPU frequency within the device and slows down all processing.

To create a model hog application that stresses resources, we created a series of processes that were intended to consume the entirety of a specific type of resource. For example, a CPU hog will simply go into an infinite loop, while a memory hog will acquire as much memory as possible and constantly touch different parts of it. We also created an 'everything hog' process that would use all of the memory, CPU, network bandwidth, and storage I/O it was allowed to consume.

For those experiments that looked at power consumption and temperature, we used the devices' built-in ACPI interfaces. Measurements were taken after a machine had been in a steady load state for ten minutes, to account for heating and cooling effects between load changes.

5.2 Availability of Fence vs Legacy Tools

5.2.1 Performance Degradation

Setup. To evaluate all of the tested tools' abilities to contain a hog process, we ran an experiment where the 'everything hog' interfered with VLC playback of a 1080p H.264 video [24] stored on disk. The results presented here were generated on a Dell Inspiron 630m laptop running Ubuntu 10.04; however, our results were similar across different device types and operating systems. We set all of the tools, including Fence, to their most restrictive settings[3] in order to contain the hog processes. For `nice` and `ionice`, VLC was additionally set to have the highest possible priority.

Result. Figure 4 shows the results[4] in terms of the proportion of frames decoded by the player when competing with hog processes. Existing tools performed very poorly when competing with the 'everything hog' – using `nice` (19.2%), `ionice` (17.7%), or `ulimit` (16.4%) showed not much more impact than not using them (16.7%). Even using a combination of all these tools showed little effect (21.8%). Setting the CPU frequency lower (7.9%) slowed down the entire system, including the video player, causing even more dropped frames. Because Fence limits all types of resources, even the everything hog has very limited impact (Fence delivers 99.8% of the frames).

One surprising finding was that `nice` was highly ineffective in protecting the video player against a CPU hog.

---

[3] I.e. `nice` level +20, `ionice` class *idle*, lowest CPU frequency setting, and `ulimit` memory to 10 MB; for Fence, 1% of the CPU, 10kBps disk rate, 10 MB of memory

[4] An anonymized video that shows the actual playback quality in this experiment is available [8].
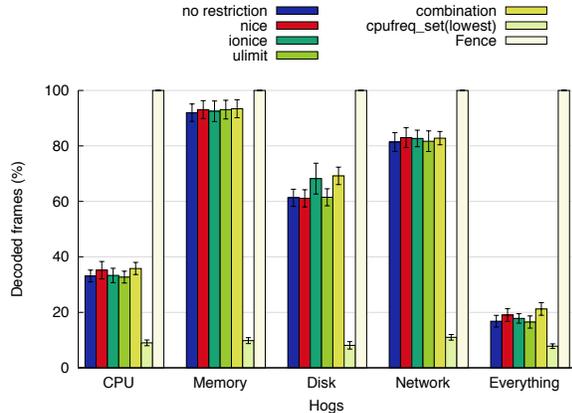
Figure 4: Proportion of decoded frames during video playback when competing with different resource hogs (averages of ten runs $\pm\sigma$, larger is better).
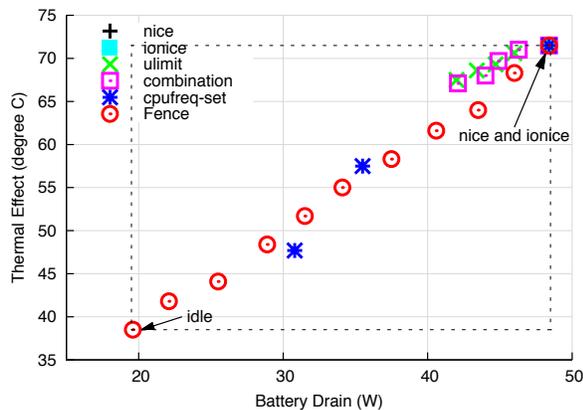

Figure 5: Battery drain and heat control ranges for various tools. The dashed rectangle bounds the minimum observed (when idle) and maximum observed battery drain / heat of the device. Fence effectively sweeps the full range.

It provides little benefit for a high-priority program that exhibits bursty behavior with implicit deadlines for work activities. However, tasks that constantly consume CPU, such as benchmarks, benefited substantially from `nice`.

Evaluations of the 'everything hog' showed that heavy use of one resource tended to cause a performance problem for other resources. Thus, resource conflicts seemed to compound, yet defenses were about the same as the strongest individual defense. This implies that uniformly strong defenses are essential to preventing an application from degrading performance.

5.2.2 Control Of Power And Heat

Setup. We next compared the effectiveness of popular tools in controlling the rate of power consumption and thermal effect (heat) caused by an application. To make this comparison, we ran an 'everything hog' process on the laptop for ten minutes and examined the ACPI battery and temperature sensors. We then applied Fence,

`ulimit`, `cpufreq-set`, `nice`, and `ionice` on the hog process, for ten minutes each, and read the battery and temperature sensors again. To understand the impact of the tool, we varied the hog's priority or resource settings by choosing a variety of settings, including the lowest to highest settings permitted by the tool. Intermediate settings were used to further understand the tool's effective range of control. An ideal tool would allow the temperature and battery drain of the 'everything hog' to be precisely controlled in a range from idle (no impact from the hog) to full system utilization (full performance of the hog).

Results. Figure 5 shows the ability of tools to control the rate of battery consumption and heat. Existing tools, such as `ulimit` and `nice`, showed no measurable impact on a hog's ability to consume power (48.4W) or to raise the temperature of a device (71.5°C). This is because when there are available resources, any program (no matter how low its priority) can exhaust the battery and thermal capabilities of a device. In comparison, `ulimit` can at least slightly improve battery life (41.9W) and thermal effect (67.4°C) because it will reduce memory use. Using all these tools simultaneously on a hog produced a similar effect to `ulimit` (42.2W, 67.1°C). Thus `ulimit`, `nice`, and `ionice` are not effective in controlling battery drain or heat.

In our experiment, `cpufreq-set` was much more effective than were other off-the-shelf tools (30.8W, 47.7°C). However, as described in the previous section, `cpufreq-set` negatively impacted all applications. It also failed to control access to resources other than the CPU, which resulted in high residual battery drain and temperature (e.g., from wireless network adapters).

Fence was much more effective in controlling battery consumption; it brought these values down to within .2% of idle. The uniform resource control impacted all power consuming resources, which led to effective control of battery drain and heat.

5.3 Effectiveness on Diverse Platforms

Setup. We ran a series of benchmarks to investigate whether uniform resource control is effective on diverse platforms. We tested Fence on smartphones (Samsung Galaxy Y, Nokia N800), tablets (Samsung Galaxy Tab, Apple iPad), laptops (MacBook, Inspiron, Thinkpad), desktop PCs (Alienware, Ideacenter), and also the commercial Amazon EC2 cloud computing platform. Operating systems tested included Windows 7, Ubuntu 10.04 to 12.04, Mac OS X 10.8.4, Nokia's OS2008, Android 2.3.5 and 4.0.3, and a jailbroken iOS 5.0.1

We ran five benchmarks for the Seattle testbed, including an HTTP server serving a large file, an Amazon S3-like block storage service, an HTTP server benchmark with small files / directory entries, a UDP P2P messaging
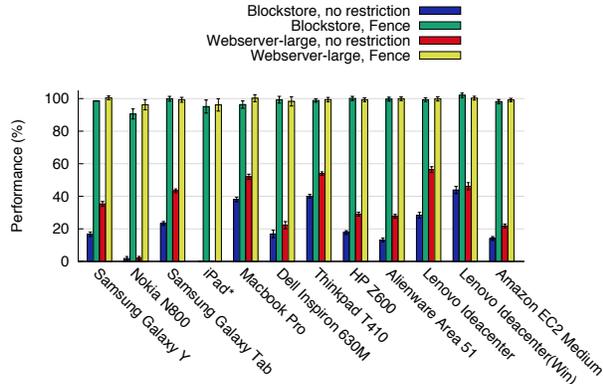
Figure 6: Benchmark performance across devices and OSes in the face of a hog (average of eight runs $\pm\sigma$, larger is better). *The iPad crashed when run with an unrestricted 'everything hog'.

program, and the Richards benchmark [19], set up against an 'everything hog'.

Note that the last four benchmarks were qualitatively and quantitatively similar, so we only present the results from the S3 blockstore. Figure 6 shows the results of these benchmarks with both an 'everything hog' with no restrictions and an 'everything hog' under Fence's control. The values are all normalized by dividing by the benchmark time on an idle device.

Results. The benchmark performance when the hog was unrestricted ranged from about 60% (Lenovo IdeaCentre running Ubuntu 11.04) down to 2% (Nokia N800). Note that performance results for the unrestricted hog on a jailbroken iPad are not represented because parts of the system crashed when we instantiated an unrestricted hog process.

If the hog process is bound by Fence to consume at most 1% of a device's resources, a typical VM size in the Seattle testbed [67], the hog's impact on a benchmark is minimal. If the hog is restricted to 1%, then the benchmark should run with about 99% of the original performance. The lowest value (90%) was recorded on a Nokia N800 that was purchased in 2007. On very low power devices such as this, Fence's overhead for polling and signaling is significant. However on other platforms, the benchmark's execution time was within 98% of the original performance. Our tests show that uniform resource control provides strong device availability with low overhead across a diverse array of modern platforms.

5.4 Overhead of Fence

Both interposition and polling can incur overhead on applications. Although the exact overhead will depend on where and how uniform resource control is implemented, it is important to have a rough understanding of the cost. We measured the overhead from both pre-post call interposition (§3.3) and polling / signaling using
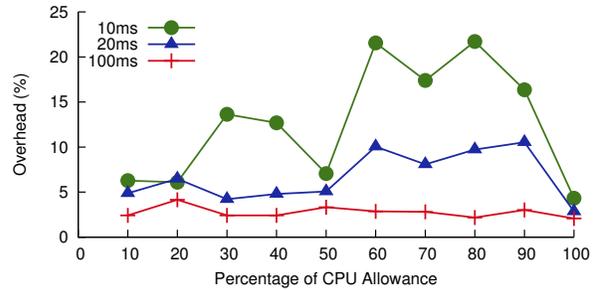

Figure 7: Overhead of CPU limiting which Fence inflicts on a CPU-intensive benchmark, at different polling rates.
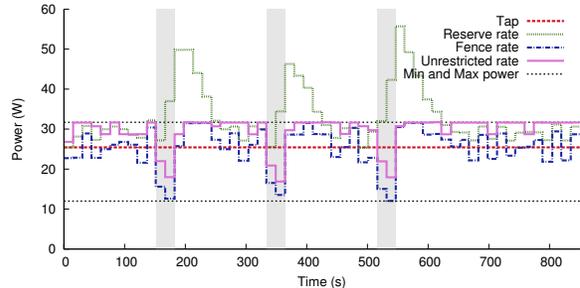

Figure 8: Fence restricts a benchmark's power draw using a simple token-bucket high-level policy [75]. *Tap* is the long-term power target; *reserve rate* the maximum currently allowed power. Benchmark idles in shaded regions.

Pulse-Frequency Modulation [37]. Figure 7 shows the relative overhead incurred by the CPU polling mechanism[5] used by Fence when compared with an unrestricted benchmark on a Lenovo IdeaCentre K330 desktop PC running Ubuntu 11.04. Results for other operating systems and platforms were qualitatively similar. Fence's current implementation uses a 100ms polling interval for Linux on a desktop machine, which causes an overhead of less than 4% across the range of CPU allowances. If the polling interval was reduced to 20ms, control would be finer-grained, but the overhead would exceed 10% for allowance values above 50%. (Only at 100% allowance, *i.e.*, no CPU restrictions, does the overhead drop again. In this case, Fence never detects that the CPU allowance was exceeded; thus, only polling causes overhead and no signaling overhead is incurred.) Results for many shorter polling rates (e.g., 1ms) are omitted because they do not fit on the graph. Using a 100ms polling interval for Fence results in low overhead across a wide variety of resource restriction settings.

5.5 Expressing a High-Level Policy

We now explore how a high-level policy can be expressed with Fence. To demonstrate this we implemented a simple policy from Cinder [75, Fig.1] to limit the long-term power draw of an application. Cinder uses *reserves* that

---

[5]Resource types other than CPU have an impact on the order of 100 *ns* to 8 μs per interposition.

store energy and *taps* that transfer energy between reserves per unit time to control energy use. In this scenario, a fixed-rate tap replenishes a reserve at fixed intervals, and the application can only draw energy from the reserve. (This is analogous to a token bucket where the tap is the filling rate of the bucket and the reserve is the number of tokens in the bucket.) The tap limits the long-term power draw, which is useful to provide battery lifetime guarantees, while the reserve allows for bursts. We implemented the policy as a control loop that reads the battery information from ACPI to determine the power draw for the previous interval, computes the necessary limits for processes, and sets the quota for Fence to enforce.

We examined the power draw on a Dell M1210 laptop. The long-term power draw goal from the battery was set to 25.4W, corresponding to about 69% between idle (12.0W) and maximum (31.5W) power draw on this machine. The "battery tap" replenished a reserve from which the application drew energy. To read the battery information from the system, we used the power supply API that updated approximately every 15 seconds. The theoretical maximum power draw in the reserve for one period was tracked at the same temporal resolution.

Fence operates at the user level and as such does not have low-level information about the energy consumption of individual hardware components. To enforce energy consumption restrictions, we requested that Fence set the quota on every renewable resource based on the amount of energy in the reserve. For example, if the reserve allowed for 28W for the next interval, then all renewable resources were set to 82% of their maximum value, as $28W = 12.0W + 0.82\% \cdot (31.5W - 12.0W)$. While this policy has many simplifying assumptions, we find that in practice it works.

We ran a Richards benchmark [19] in the following fashion. It ran for 10 periods (where a period is the amount of time between power supply readings), and then slept for 2 periods. After doing this three times, it ran continually for 20 periods. The benchmark was run according to this schedule, both with and without Fence.

Results of this benchmark are shown in Figure 8. The shaded areas indicate when the benchmark was sleeping. The tap rate, representing the long-term power draw goal, was 25.4W for each period. The unrestricted rate shows the benchmark's power draw in absence of any power restrictions: For the majority of time that the benchmark test was active, it caused the battery to drain at approximately the maximum rate.

The Fence rate shows the benchmark's power draw when run under Fence. Note that unlike the unrestricted benchmark, the behavior of this line is limited by *two* power rates: The maximum power draw that the system allows, and also the amount in reserve in each period. When the reserve is large (for example, right after a sleep period), Fence behaves similarly to the unrestricted rate, and uses energy at approximately the maximum rate. When the reserve is small, Fence restricts power use to approximately the rate of the tap. Cinder's policy uses Fence to enforce that the overall power use stays within budget, while allowing the application to have flexibility in when it consumes its energy budget.

Fence's power restrictions are inaccurate, due in part to our implementation working at user space. As such we do not directly account for complex issues of power use in the underlying platform (e.g. tail power consumption [34]). However, Fence will read the new battery level during the next period and will drain the reserve based upon what was actually consumed. Fence's adjustment in subsequent periods mitigates the effect of this inaccuracy.

Fence made implementing Cinder's policy very straightforward. The implementation is 150 lines of code and did not require any detailed knowledge of the underlying resource types or control mechanisms. The implementation simply reads the battery level and adjusts the values in the resource table based upon the reserve and tap settings. This demonstrates that uniform resource control may make policy implementation easier, which we hope will lead to more application developers and system designers using such mechanisms.

6   Practical Fence Deployments

Seattle's Use of Fence.   Fence is deployed as a part of the Seattle testbed [21]. Seattle runs on laptops, tablets, and smartphones, with more than twenty thousand installs distributed around the world [88]. The Seattle testbed is used to measure end user connectivity, to build peer-to-peer services, and as a platform for apps that measure and avoid Internet censorship [16, 44, 46, 73, 81]. Seattle is also widely used in computer science education where it has been used in more than fifty classes at more than a dozen universities [39, 41, 67].

Each device running Seattle uses Fence to allocate a fixed percentage (usually 10%) of the device's CPU, memory, disk, and other resources to one or more VMs. When a Seattle sandbox [40] is started, it reads a text file that lists the resources allocated to the program. Each line contains a resource type and quantity (for fungible resources) or the name of the resource (for non-fungible resources). For example, `resource memory 10000000` sets the memory cap to 10 million bytes, and `resource udpport 12345` allows the program to send and receive UDP traffic on port 12345.

Seattle's categorization of resources and enforcement mechanisms are shown in Table 2. In the Seattle platform's sandbox, the calls to network and disk devices are routed through Fence, whereas usage statistics on memory and CPU are polled from the operating system. Unfortunately, there is not a clean, cross-platform way

Resource	Fungible	Renewable	Seattle Control	Lind Control
CPU	Yes	Yes	Polling	Polling
Threads	Yes	No	Interposition	Interposition
Memory	Yes	No	Polling	Interposition
Storage space	Yes	No	Polling	Interposition
UDP / TCP ports	No	No	Interposition	Interposition
Open sockets	Yes	No	Interposition	Interposition
Open files	Yes	No	Interposition	Interposition
File R/W	Yes	Yes	Interposition	Interposition
Network R/W	No	Yes	Interposition	Interposition
HW randomness	Yes	Yes	Interposition	Interposition

Table 2: The user-space Fence implementation's resource categorization used in Seattle [40] and Lind [65].

for Fence to reclaim memory used by an application. As such, Seattle enforces a hard maximum allowed memory limit for an application, so a process trying to exceed the limit will be forcefully killed.

Lind's Use of Fence. In addition to being used in Seattle's sandbox, Fence is also used in a Google Native Client sandbox called Lind [65]. Since this sandbox provides a different abstraction (a POSIX system call), some of the low-level resource characteristics (and thus, means of controlling consumption) vary from Seattle's deployment. This sandbox has a hook that allows Fence to interpose on memory requests and avoid polling. (This also allows Fence to control native programs that cannot be executed in the Seattle sandbox.) The rightmost column of Table 2 overviews Lind's resource categorization.

The Sensibility Testbed's Use of Fence. Fence is not limited to controlling traditional computational resources – it is currently being integrated by the Sensibility Testbed [22] developers. Sensibility Testbed consists of smartphones and tablets where researchers get access to dozens of diverse sensors including WiFi network information, accelerometer readings, battery level, device ID, GPS, and audio. Sensibility Testbed uses the same sandbox as the Seattle testbed. However, in addition to limiting a program's access to resources for performance reasons, Sensibility Testbed allows users who pass an IRB review to get access to devices' sensors at a rate meant to preserve user privacy (e.g. by limiting the rate or accelerometer queries to prevent sniffing keystrokes [70]). The demonstrated ability of Fence to provide privacy guarantees across "sensor resources" validates the generality of uniform resource control as a technique.

Experiences / Limitations. Our experience is that Fence's effectiveness depends on the amount of resource information available to the developer. E.g., it is relatively straightforward for a sandbox operating in user space to limit disk I/O when interposing on a `read` or `write` call on a file descriptor for a regular file – the number of disk blocks accessed is fairly easy to predict. However, it is very hard to provide performance isolation for a call like `mount`: It may perform a substantial number of disk I/O operations in the kernel, which are not predictable by a sandbox in user space. (For similar reasons, a hardware

resource like L2 cache may be difficult to meter using software in an OS kernel.)

The time it takes a developer to understand the resources consumed by a call depends a lot on the implementer. In our experience, adding calls into Fence in the appropriate parts of the code only takes a few minutes per API call. In fact, outside groups have used Fence to provide resource controls on many platforms (Android, iOS, Nokia, Raspberry PI, and OpenWrt) each necessitating only a few days' worth of effort. The bulk of the effort lies in understanding what resources a call will consume.

7 Related Work

Our work follows a substantial amount of prior work that has recognized the need to prevent performance degradation, enhance battery control, and manage heat. Fence is unique in that it works across diverse platforms and presents a portable, user-space solution that unifies resource control across resource types.

Deployed Solutions For Improved Availability. The need for improved device availability has produced strategies for preventing performance degradation, with different levels of required privileges, including per-application, per-user, OS-wide, and hypervisor-based approaches. For example, modern web browsers monitor the run time of their JavaScript engine to detect "runaway" scripts, i.e., programs that take excessive time to execute and allow the user to stop the script. A malicious script can fool the timer however, by partitioning its workload, or by using Web Workers [69]. The runaway timer also ignores other resources that the JavaScript program may take, such as network and memory. Another application-level example, the Java Virtual Machine [12], supports setting a limit on the amount of memory that can be allocated by a process. However, much like Lua [56], Flash [1], and other programming language VMs, the Java Virtual Machine (JVM) does not support limiting the rate of some fungible/renewable resources, such as CPU — the primary cause of energy drain and heat on many devices.

Operating system virtual machine monitors control many different resource types, depending on the implementation [13, 26, 28, 29, 36, 72]. However, the resource controls are ad hoc and specific to the type of resource. Bare-metal hypervisors [13, 36] require kernel changes, whereas hosted hypervisors [26, 28] have substantial per-VM resource costs. As a result, none of these are practical to deploy on a per application basis, especially on devices like smartphones and tablets.

More recently, the `cgroups` [4] infrastructure in Linux addresses many of the issues discussed in this paper (CPU, memory use, and disk I/O). `cgroups`, however, is specific to newer versions of Linux. Due to its location in the kernel, it has better resource granularity than Fence. However, `cgroups` focuses on point solutions for spe-

cific resources, rather than a more uniform and general solution. In contrast, Fence provides a user-space uniform resource control solution that works across a wide array of devices without kernel modifications.

Operating Systems Research. There are many clean slate OS approaches that would achieve the same improvements as Fence. Since many deployed OSes are ineffective at preventing performance degradation, Fence focuses on providing this property in user space. Several OS-level efforts aim at managing processes' low-level resources consumption in a system-wide manner to control battery life. ECOSystem [89], Odyssey [48], Cinder [75], and ErdOS [85] focus on extending battery life. Their reasoning about resources is fixated on the energy consumed by renewable resources. Scheduling decisions are made exclusively on this foundation. Fence also supports energy-aware resource limiting, but we measure resource consumption as "unit of resource," which enables interesting use cases, such as control strategies based on device responsiveness or service availability.

Research projects [42, 43, 54, 60, 61, 66, 87] try to manage thermal effects through temperature-aware priority-based scheduling and thread placement on the CPU. Our work demonstrates that priority-based scheduling is ineffective at upper-bounding a process.

The efforts presented above require deep changes to applications to support resource aware operation, are based on new kernels, or use forms of priority-based scheduling that do not succeed in limiting resource consumption. In contrast, Fence requires no changes to OSes or applications, and operates entirely in user space. Furthermore, Fence puts boundaries on any type of resource consumption and can affect more than battery drain. This is achieved by reasoning about, tallying, and controlling multiple different resources in a uniform way.

Controlling resource consumption is well researched in the real-time OS community [31, 64, 74]. Our research focuses on techniques to enhance general-purpose OSes with minimal disruption to existing systems.

There have been a substantial number of complimentary user-space techniques for improving security that involve system-call interposition [52, 53, 59], host intrusion detection [47, 50], and access control [55, 77, 86]. These mechanisms aim to permit or deny access to resources requested by applications based upon how they will impact the security of the system. Fence's goal is fundamentally different: It limits the rate of resources consumption so that the use of allowed resources does not impact the availability or correct operation of a device.

Idle Resource Consumption A variety of frameworks, such as Condor [63], SETI@Home [23], and Folding@Home [9] allow trusted developers to consume idle resources on a users device. These wait for the user's system to be idle and then run, so as to not interfere with performance. However, once they run, these programs may fully utilize the CPU, GPU, and similar resources on the device, often leading to significant power drain [2]. Fence may also operate in such an on-off manner, but it is flexible enough to allow more advanced policies.

One related effort to Fence in this domain was the construction of an idle resource consumption framework by Abe et al. [30]. This system runs in user space and leverages special functionality from OS-specific hooks in Solaris revolving around `dtrace` [38]. While this provides easy control of native code (which Fence lacks), equivalent techniques do not exist across platforms. As such, this will only work for a few environments, such as BSD, that support similar hooks. As a result, Abe's work cannot be deployed on many systems. (For example, Windows lacks a non-bypassable method for interposing on an untrusted application's operating system calls.) Additionally, as this work seeks to enable background execution only when foreground execution is idle, many of the detection and scheduling results from this work do not apply to our domain.

Distributed Systems Research Controlling resource utilization is also an important problem in distributed contexts [32, 33, 49, 51, 57, 62, 79, 82, 84]. Significant prior work has focused on efficiently allocating available resources between multiple parties. While managing distributed resources is orthogonal to our goals, Fence embraces richer semantics to reason about resource consumption, rather than utilization; we believe that our approach towards uniform resource control would apply well as a heterogeneity-masking technique in distributed contexts.

8 Conclusion

This paper introduces uniform resource control by classifying resources along the dimensions of renewability and fungibility. Our system, Fence, demonstrates that uniform resource control provides flexibility by controlling multiple heterogeneous resources across almost a dozen diverse operating systems. Furthermore, we demonstrate that this technique is particularly adept at addressing issues of performance degradation, heat, and battery drain that many users face today.

In addition to the experimental validation presented in this paper, Fence has been deployed and adopted to provide resource containment of untrusted user code in several testbeds. As a result, tens of thousands of smartphones, tablets, and desktop OSes around the world rely on Fence to prevent device degradation. Beyond our deployment, we believe Fence's abstractions and mechanisms could be used to provide better resource control for sandboxes, web browsers, virtual machine monitors, and operating systems.

References

[1] Adobe flash player. http://www.adobe.com/software/flash/about.

[2] BOINC wiki: Heat and energy considerations. http://boinc.berkeley.edu/wiki/Heat_and_energy_considerations.

[3] Box Sync ruins my rMBP. https://support.box.com/entries/21766312-Box-Sync-ruins-my-rMBP.

[4] cgroups. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[5] Crash Internet Explorer 9 in one line of Javascript! http://www.roslindesign.com/2011/04/08/crash-internet-explorer-9-in-one-line-of-javascript.

[6] Death to Javascript: CNN Edition. http://www.goodbyemicrosoft.net/news.php?item.708.4.

[7] Dropbox using 100% of each core when I turn my machine on. https://forums.dropbox.com/topic.php?id=62054.

[8] Fence vs nice, ionice, ulimit, cpufreq-set demo video. https://www.youtube.com/user/fencedemo.

[9] Folding@Home. http://folding.stanford.edu/.

[10] installd took 130% of CPU and sent temperatures through the roof. Why? https://discussions.apple.com/thread/3738340.

[11] ionice. http://linux.die.net/man/1/ionice.

[12] Java Virtual Machine. http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[13] Kernel-based virtual machine. http://www.linux-kvm.org/page/Main_Page.

[14] Linux - File descriptors exhausted, how to recover. http://www.linuxquestions.org/questions/linux-newbie-8/linux-file-descriptors-exhausted-how-to-recover-4175417070/.

[15] nice. http://www.kernel.org/doc/man-pages/online/pages/man2/nice.2.html.

[16] Open3GMap. https://o3gm.cs.univie.ac.at/.

[17] Optimizing downloads for efficient network access. https://developer.android.com/training/efficient-downloads/efficient-network-access.html.

[18] Potentially Unwanted Miners – Toolbar Peddlers Use Your System To Make BTC. http://blog.malwarebytes.org/fraud-scam/2013/11/potentially-unwanted-miners-toolbar-peddlers-use-your-system-to-make-btc/.

[19] Richards benchmark. http://www.cl.cam.ac.uk/~mr10/Bench.html.

[20] Scan causes CPU to overheat. https://community.norton.com/t5/Norton-Internet-Security-Norton/Scan-causes-CPU-to-overheat/td-p/642743.

[21] Seattle web page. https://seattle.poly.edu/.

[22] Sensibility Testbed. https://sensibilitytestbed.com/.

[23] SETI@Home. http://setiathome.berkeley.edu/.

[24] Sintel trailer. http://www.sintel.org/download.

[25] ulimit. http://linux.die.net/man/1/ulimit.

[26] VirtualBox. https://www.virtualbox.org/wiki.

[27] VirusScan freezes up completely during full scan. https://community.mcafee.com/message/211811.

[28] VMware workstation. http://www.vmware.com.

[29] Windows Virtual PC. http://support.microsoft.com/kb/958559.

[30] Y. Abe, H. Yamada, and K. Kono. Enforcing appropriate process execution for exploiting idle resources from outside operating systems. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 27–40, 2008.

[31] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating multimedia applications in hard real-time systems. In *In Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[32] Y. Agarwal, S. Savage, and R. Gupta. Sleepserver: a software-only approach for reducing the energy consumption of pcs within enterprise environments. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.

[33] A. AuYoung, B. Chun, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Bellagio: An economic-based resource allocation system for planetlab. `http://www.sysnet.ucsd.edu/~aauyoung/bellagio/about.php`.

[34] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.

[35] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *the nineteenth ACM symposium on Operating systems principles*, 2003.

[37] M. Broughton. Pulse-frequency modulation applied to the digital control of a thyristor. *IEEE Trans. Industrial Electronics and Control Instrumentation*, 24(2):173–177, May 1977.

[38] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, 2004.

[39] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: a platform for educational cloud computing. In *40th ACM technical symposium on Computer science education*, 2009.

[40] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *the 17th ACM conference on Computer and communications security*, 2010.

[41] J. Cappos and R. Weiss. Teaching the security mindset with reference monitors. In *45th ACM technical symposium on Computer science education*, 2014.

[42] J. Chen, C. Hung, and T. Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 236–248. IEEE, 2007.

[43] J. Choi, C. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 international symposium on Low power electronics and design*, pages 213–218. ACM, 2007.

[44] L. Collares, C. Matthews, J. Cappos, Y. Coady, and R. McGeer. Et (smart) phone home! In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, pages 283–288, New York, NY, USA, 2011. ACM.

[45] S. S. Craciunas, C. M. Kirsch, and H. Röck. I/O resource management through system call scheduling. *ACM SIGOPS Operating Systems Review*, 42(5):44–54, 2008.

[46] J. Eisl, A. Rafetseder, and K. Tutschku. Service architectures for the future converged internet: Specific challenges and possible solutions for mobile broad-band traffic management. *Future Internet Services and Service Architectures*, 15:49, 2011.

[47] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 194–208, 2004.

[48] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. In *ACM Transactions on Computer Systems (TOCS)*, 2004.

[49] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *the 8th USENIX conference on Operating systems design and implementation*, 2008.

[50] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128, 1996.

[51] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, 2003.

13

[52] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *IN NDSS*, 2003.

[53] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, 1996.

[54] J. Hasan, A. Jalote, T. N. Vijaykumar, and C. E. Brodley. Heat stroke: Power-density-based denial of service in SMT. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 166–177, Washington, DC, USA, 2005. IEEE Computer Society.

[55] T. L. Hinrichs, D. Martinoia, W. C. Garrison III, A. J. Lee, A. Panebianco, and L. Zuck. Application-sensitive access control evaluation using parameterized expressiveness. In *Computer Security Foundations Symposium, 2013. 26th IEEE*, 2013.

[56] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua - an extensible extension language. In *Software: Practice & Experience*, 1995.

[57] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing networked resources with brokered leases. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, 2006.

[58] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical report, Technical report HPL–CSP–91–7rev1, 1991.

[59] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[60] R. Jayaseelan and T. Mitra. Temperature aware scheduling for embedded processors. In *VLSI Design, 2009 22nd International Conference on*, pages 541–546. IEEE, 2009.

[61] A. Kumar, L. Shang, L. Peh, and N. Jha. Hybdtm: a coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd annual Design Automation Conference*, pages 548–553. ACM, 2006.

[62] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3), Aug. 2005.

[63] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, 1988.

[64] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 61–70, New York, NY, USA, 2012. ACM.

[65] C. Matthews, J. Cappos, R. McGeer, S. Neville, and Y. Coady. Lind: Challenges turning virtual composition into reality. In *Freeco 2011 Onward! Workshop: Towards Free Composition of Software Modules*, 2011.

[66] A. Merkel and F. Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. *ACM SIGOPS Operating Systems Review*, 42(4):1–12, 2008.

[67] Monzur Muhammad and Justin Cappos. Towards a Representative Testbed: Harnessing Volunteers for Networks Research. In *The First GENI Research and Educational Workshop*, GREE'12, 2012.

[68] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 67–72. IEEE, 2001.

[69] Nicholas C. Zakas. Responsive Interfaces. http://de.slideshare.net/nzakas/responsive-interfaces.

[70] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: Password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM.

[71] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[72] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 351–366, 2006.

[73] A. Rafetseder, F. Metzger, D. Stezenbach, and K. Tutschku. Exploring youtube's content distribution network through distributed application-layer measurements: a first view. In *Proceedings of the 2011 International Workshop on Modeling, Analysis, and Control of Complex Networks*, Cnet '11, pages 31–36. ITCP, 2011.

[74] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 150–164, 1998.

[75] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *EuroSys 2011*, 2011.

[76] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer Volume 27 Issue 3*, 1994.

[77] M. Sherr and M. Blaze. Application containers without virtual machines. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 39–42, New York, NY, USA, 2009. ACM.

[78] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[79] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. Technical report, Old Dominion University, Norfolk, VA, USA, 1996.

[80] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[81] K. Tutschku, A. Rafetseder, J. Eisl, and W. Wiedermann. Towards sustained multi media experience in the future mobile internet. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–6. IEEE, 2010.

[82] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol.*, 9(1), Feb. 2009.

[83] Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE. Accessed April 14, 2012. http://msdn.microsoft.com/en-us/library/ms740621%28VS.85%29.aspx.

[84] M. Valero, A. Bourgeois, and R. Beyah. Deep: A deployable energy efficient 802.15.4 mac protocol for sensor networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–6, 2010.

[85] N. Vallina-Rodriguez and J. Crowcroft. Erdos: achieving energy savings in mobile os. In *Proceedings of the sixth international workshop on MobiArch*, MobiArch '11, pages 37–42, New York, NY, USA, 2011. ACM.

[86] H. Vijayakumar, J. Schiffman, and T. Jaeger. Process firewalls: Protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 57–70, New York, NY, USA, 2013. ACM.

[87] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin. Dynamic thermal management through task scheduling. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 191–201. IEEE, 2008.

[88] Yanyan Zhuang and Albert Rafetseder and Justin Cappos. Experience with Seattle: A Community Platform for Research and Education. In *The Second GENI Research and Educational Workshop*, GREE'13, 2013.

[89] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *10th international conference on Architectural support for programming languages and operating systems*, 2002.