VSense: Virtualizing Stateful Sensors With Actuators

Navin K. Sharma, David E. Irwin, and Prashant J. Shenoy
{nksharma,irwin,shenoy}@cs.umass.edu
University of Massachusetts, Amherst

Abstract

High-power sensors, such as steerable radars and pan-tilt-zoom cameras, expose programmable actuators to applications, which actively control them to dictate the type, quality, and quantity of data collected. Since networks of high-power sensors are expensive to construct, maintain, and deploy, they represent a scarce resource that multiple applications with conflicting goals often share. However, existing mechanisms are too coarse-grained to satisfy these conflicting goals. To address the problem, this paper presents VSense, a virtualization approach that augments virtual machines with virtual sensors to enable fine-grained multiplexing between unmodified applications. VSense represents each virtual sensor as a state machine, interleaves state transitions (i.e., acutations) to balance fair access to the physical sensor with its efficient use, and employs an actuation-aware proportional-share scheduler to ensure performance isolation between virtual sensors while optimizing actuation overheads. We implement VSense in Xen and prototype it using one example of a high-power sensor with actuators—a PTZ camera—to multiplex concurrent applications. Our results show that VSense efficiently isolates the performance of virtual sensors, allowing concurrent applications to satisfy conflicting goals. In a case study we conduct, we show that concurrent applications receive tolerable, but degraded performance—ranging from 1.5x less to 8x less in our examples—than seen with a dedicated sensor.

1 Introduction

An emerging class of sensor networks (sensornets) consists of nodes with high-power sensors that comprise one or more programmable actuators. To achieve their goals, sensing applications programatically actuate these sensors to control the type, quality, and quantity of data collected. As an example, consider a network of pan-tilt-zoom (PTZ) camera sensors used for both monitoring and surveillance: the monitoring application continuously scans each road at an intersection in a fixed pattern, while the surveillance application intermittently steers the cameras to track suspicious vehicles moving through its field of view. Each application must alter the settings of three distinct actuators—pan, tilt, and zoom—in different ways at different times to satisfy its goals.

Other examples of sensors with actuators include steerable radars and weather stations. Scientists use networks of steerable weather-sensing radars to track tornadoes, detect rainfall intensity, and estimate three-dimensional wind direction and velocity by steering the radar to scan specific regions of the atmosphere [13]. Weather stations operate multiple atmospheric sensors and also expose simple actuators, such as setting the sensing resolution, that weather monitoring applications use in different ways to balance energy, wireless bandwidth, and storage.

Since high-power sensors are often costly to construct, deploy, and maintain, they represent scarce resources that users must share. Simple multiplexing approaches dedicate entire sensors, including control of the actuators, to individual applications for a period of time, and schedule applications in a coarse-grained batch fashion [3]. However, such coarse-grained reservations prevent the fine-grained multitasking—at the level of individual actuations—required for the camera sensornet example above, and force a choice between either monitoring the intersection or tracking the suspicious vehicle during each coarse-grained time period.

In general, fine-grained time-sharing benefits any application that values continuous access to sensor data and is willing to tolerate a higher latency or lower resolution than possible with a dedicated sensor. While coarse-grained reservations are akin to early batch processing systems, fine-grained concurrent access to a sensor is similar to time-shared computing systems, where multiple applications share a machine's physical resources at a fine time-scale. As with early batch computing systems, the deployment cost of high-power sensors limits their number, but also increases the potential benefits from fine-grained sharing. To realize this potential, this paper presents *VSense*, a system for fine-grained multiplexing—at the level of individual actuations—of high-power sensors.

VSense uses *virtualization* as a key enabler for providing fine-grained concurrent access to sensors with programmable actuators. VSense extends virtual machine monitors (VMMs) to include *virtual sensors* (vsensors) that applications, running within guest virtual machines (VMs), actuate to drive data collection based on their own distinct needs. The VMM mediates access to the physical sensor to ensure safe operation and prevent physical damage, while balancing efficient use of the sensor with the enforcement of per-VM fairness bounds. Three key attributes of high-power sensors distinguish them from other devices, and impacts VSense's approach to virtualizing them.

1. Since applications need the ability to *control actuation* in order to drive their data collection, high-power sensors must expose these actuators directly to the application. Thus, unlike other virtualized I/O devices, such as virtual disks or NICs, VSense cannot completely hide a physical device's actuators from its virtual counterparts.
2. High-power sensors are *stateful*: each actuation changes the physical state of the device. Further, the current state of the actuators determines the cost, measured in *time*, to transition to a new state. Thus, handling both state management and the inherent uncertainty of actuation costs are key challenges.
3. Mechanical actuators are *slow*—sometimes incurring latencies on the order of seconds. Consequently, VSense must optimize actuation overheads while multiplexing concurrent applications—somewhat analogous to disk controllers that must optimize seek times to reduce head movement and increase I/O throughput.

Existing techniques for virtualizing I/O devices do not map well to stateful high-power sensors. Such techniques either hide much of physical device's capabilities—its actuators—from VMs to support a wide-range of devices, or virtualize a layer beneath the device (*e.g.*, its bus), which increases performance but prevents multiplexing between VMs. In contrast, VSense must expose the same actuators present on a physical sensor to each of its vsensors. Additionally, while recent mote OS techniques augment sensor platforms with explicit resource control (PixieOS [14]) and concurrency (ICEM [11]), they focus on "conventional" resources (CPU, bandwidth, energy) and not control of sensor actuators. In designing VSense, this paper makes contributions in three areas.

- **Virtualizing Stateful Sensors.** VSense employs a novel finite state machine approach to track the state of each virtual sensor as it actuates. VSense uses these state machines to implement a request emulation mechanism that efficiently multiplexes independent streams of actuation requests from multiple vsensors. Together these mechanisms intelligently group requests within each request stream to optimize state restoration overheads incurred when switching from one vsensor to another.
- **Fair Actuation-aware Sensor Scheduling**. In addition to mimicking the physical sensor's interface, VSense employs a fair proportional-share scheduler, based on Start-time Fair Queuing (SFQ), to allocate *shares* of stateful physical sensors to each VM. However, a strict proportional-share scheduler, while fair, can be very inefficient, since it ignores actuation costs when scheduling sensor requests. We propose Actuator-aware Fair Queuing (AFQ) to optimize sensor actuation costs and expose an explicit tradeoff between fairness and efficiency.
- **Implementation and Experimentation**. We implement a prototype of VSense using the Xen VMM for a PTZ camera and use it to conduct a detailed experimental evaluation. Our results show that VSense is able to efficiently isolate the performance of stateful vsensors, allowing multiple concurrent applications in our case study to satisfy conflicting goals. As one example, we show that VSense is able to photograph an object every 23 feet moving at nearly 3 miles/hour along its trajectory at a distance of 300 feet, while simultaneously supporting a security application that photographs a fixed point every 3 seconds.

The rest of this paper is organized as follows. Section 2 presents background on virtualization and its relationship to high-power sensors. Section 3 through Section 6 present VSense's design and implementation. We then evaluate VSense in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Background

We first define a system model to guide our work and outline the challenges to fine-grained time-sharing of stateful sensors. We then describe the advantages of virtualization as the foundation for multiplexing sensors with actuators between concurrent applications.

2.1 System Model

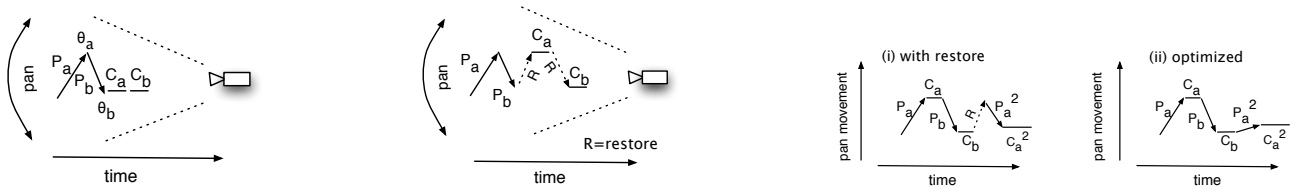
Our work assumes a network of high-power sensors, where each sensor (i) consists of one or more programmable actuators that applications may control, and (ii) attaches to a node that has local processing, storage and communication capabilities. For the purposes of this work, we assume each sensor attaches to an embedded x86-class node capable of running modern VMMs. VSense multiplexes the sensor across multiple applications that run on this node. We model each application as a stream of actuation and sensing requests of the form: $[A_1 A_2 \ldots A_n S_1 \ldots S_m]^+$, $n \geq 0$, $m > 0$, where A_i and S_i denote an individual actuation and sensing request, respectively.

Intuitively, each application continuously issues one or more actuation requests to "prepare/steer" the sensor, followed by one or more sense requests to gather data. In our camera sensornet, for instance, a monitoring application might issue a repeating pattern of *pan* and *tilt* requests to steer the camera, followed by one or more *capture* requests to retrieve images. We assume the stream of actuation and sense requests from different applications are independent of one another. To enable fine-grained time-sharing, VSense must *interleave* these requests on the underlying physical sensor.

2.2 Multiplexing Stateful Sensors

Assuming the above system model, we now outline the primary challenges to fine-grained time-sharing of sensors with actuators. Consider two users—Alice and Bob—that time-share a single PTZ camera. Assume that Alice issues a pan, followed by a capture, denoted by $P_a C_a$ and that Bob issues a similar sequence $P_b C_b$, where the subscripts a and b denote the user issuing the commands, respectively. Consider a naïve schedule that interleaves these requests in the following order on the camera: $P_a P_b C_a C_b$. In this case, the camera pans to position θ_a, as requested by Alice, and then pans to a position θ_b, as requested by Bob (see Figure 1(a)).

Thus, executing Alice's capture command C_a next results in an erroneous picture, since the camera's lens is at pan position θ_b when Alice expects the camera's lens to be at pan



(a) Naïve interleaving of $P_aP_bC_aC_b$ yields incorrect results.

(b) Correct interleaving of $P_aP_bC_aC_b$ with state restorations.

(c) Optimizing state restoration costs.

Figure 1: Examples showing why request interleaving is challenging in stateful sensors.

position θ_a. The problem arises since the camera is stateful—Bob's actuation leaves the camera in a different state than Alice left it. This simple example also illustrates why naïve time-slicing using time quanta is not appropriate for stateful devices. Since both Alice and Bob alter the camera's state during their time-slice, they have no guarantee about the device's state at the beginning of any time-slice.

A straightforward solution to dealing with stateful devices is to *restore* the previous state prior to switching from one user's request stream to the next. The approach has been proposed previously to address virtualizing stateful devices in general [8], and is similar to CPU schedulers that perform state restoration during a context switch, by saving one thread's program counter and CPU registers before restoring those of another. Thus, the sensor would restore Alice's state prior to executing the capture command C_a, ensuring that she captures a meaningful image.

However, in this example, state restoration is clearly wasteful, since it involves re-executing the P_a pan command to move the camera back to position θ_a, as shown in Figure 1(b). The naïve interleaving combined with state restoration renders the initial execution of P_a useless. Further, wasteful actuations are still possible when the camera uses a better interleaving, such as $P_aC_aP_bC_b$. In this case, suppose that Alice subsequently issues new requests $P_a^2C_a^2$, which execute after Bob's requests. Restoring the camera to position θ_a is not necessary prior to executing the new pan P_a^2. Instead it is more efficient to directly move from the current position θ_b to the position indicated in P_a^2, as shown in Figure 1(c). These simple examples highlight two important insights.

First, an arbitrary interleaving of actuation commands across users is problematic. Specifically, interleaving any actuation command A_b from Bob is not desirable when executing any of Alice's actuation or sensing requests: Bob's actuation may inadvertently modify Alice's state and trigger wasteful state restoration procedures prior to switching back to Alice. Instead it is more efficient to execute a group of requests from Alice of the form $A_1A_2 \ldots A_nS_1 \ldots S_m$, where each A_i represents a single actuation such as pan or tilt, before switching to Bob, since this avoids needless state restorations. In the above example, this implies executing the sequence P_aC_a for Alice before switching to Bob's requests P_bC_b, and then back to Alice.

Second, the multiplexing mechanism must take into account each user's state before switching contexts, since state restoration may be necessary if the underlying sensor is in a different state than the expected state. However, both actuations and state restorations incur high overheads because (i) mechanical actuation is slow and, more importantly, (ii) the time to restore the previous state is not fixed—it is a function of the sensor's current state and its next state. Consequently, limiting both actuation and state restoration overheads is important.

2.3 Why Virtualization?

The scenarios above highlight the challenges that arise when multiplexing a sensor at the level of individual actuations between applications with conflicting demands. Addressing the multiplexing problem does not necessarily require virtualization—a custom application-level scheduler could solve the same problem. However, virtualization is an attractive mechanism for fine-grained multiplexing of stateful sensors that has two key benefits:

- **Device Abstraction.** Virtualization provides the abstraction of a dedicated vsensor to each user. The vsensor provides the same functionality, albeit slower, as the underlying physical sensor. Specifically, vsensors expose slower "virtual actuators" that are equivalent to the physical sensor's actuators. Thus, applications control each vsensor and its exposed actuators as if it were dedicated, and remain oblivious of the underlying multiplexing.
- **Strong Isolation.** The device abstraction promotes strong software isolation: multiple instances of unmodified sensing applications use their virtual sensor as if it were a dedicated physical sensor. Strong isolation also extends to performance: each user's performance is a function of its share of the device's resources. Shares guarantee a minimum fraction of a vsensor's resources—in this case actuation time. For instance, a fair proportional-share scheduler would allocate a fraction $w_i/\sum_j w_j$ of the physical sensor's time to each vsensor with weight w_i.

These benefits motivate a virtualization layer that supports fine-grained multiplexing of high-power sensors with actuators. However, existing methods for virtualizing traditional I/O devices, such as disks and NICs, do not apply to sensor devices. These techniques focus on either emulating a complete physical bus to support running unmodified device drivers inside a guest VM, or using a generic virtualized driver for each class of device that forwards I/O requests to

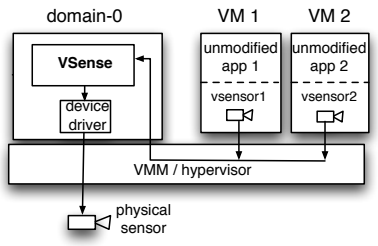


Figure 2: VSense architecture overview.

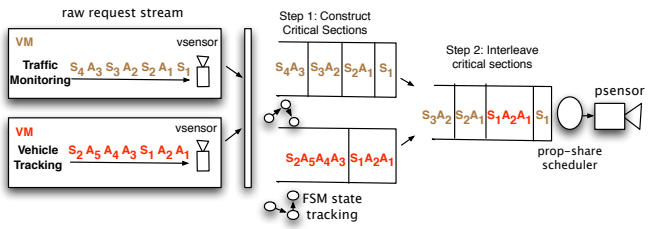


Figure 3: Constructing and interleaving critical sections.

a corresponding back-end driver for the physical device. The former case does not support multiplexing since it attaches each device to the virtual bus of only a single guest VM. The latter case hides the physical device's actuators to ensure the virtualized driver applies to a range of different physical devices.

Fair-share sensor multiplexing also differs from fair-sharing of traditional I/O devices, since sensor applications control actuators directly to influence the data they gather. For instance, even I/O-intensive applications use the filesystem interface to read and write data, and do not concern themselves with the low-level details of the disk head's position or the placement of data on disk. As a result, virtual block devices attached to VMs need not expose the physical disk head's position. Additionally, the fine time-scales, mechanical instability, and jitter associated with adjusting a disk head makes accounting problematic, which is not an issue for relatively large and slow sensors.

Hence, a virtualization layer for sensors with actuation capabilities must differ from traditional methods for virtualizing I/O devices. The following sections address this problem by presenting the design and implementation of VSense.

3 VSense Design Overview

VSense assumes each node runs a modern VMM, such as Xen [1] or VMware [18]. Each node is capable of running multiple concurrent VMs, one for each user or application. Traditional VMMs virtualize a node's physical resources, such as the processor and memory, and provide performance isolation across VMs. VSense extends traditional VMMs by adding virtualization support for sensors with actuators.

VSense provides a virtual sensor abstraction by attaching a vsensor to each virtual machine. To a VM's applications, a vsensor operates like a slower version of the physical sensor that has identical functionality (including actuation capabilities); an application designed to run on the physical sensor should also run unmodified on the corresponding vsensor. VSense resides in the VMM or a privileged control domain—`domain-0` in Xen—and decides how to interleave requests from each vsensor on the underlying physical sensor (see Figure 2). Sensor virtualization involves two key functions.

- **Maintaining Sensor State Machines**. In Section 4, we describe how VSense tracks the current state of each vsensor as well as the physical sensor (Section 4.1), and uses the information to infer critical sections in the request stream that execute as atomic units (Section 4.2). VSense uses the state information further to perform intelligent state restoration when context switching from one vsensor to another (Section 4.3). Each technique serves to reduce mechanical actuation cost.
- **Actuator-aware Fair Scheduling**. In Section 5, we describe VSense's use of a proportional-share scheduling algorithm to fairly allocate control of the physical sensor's actuators to vsensors according to an assignment of weights (Section 5.1). To optimize mechanical actuation while maintaining fairness, we describe an actuator-aware fair queuing (AFQ) algorithm that defines a trade-off between strict fairness and efficient actuation (Section 5.2).

Following our system model, each vsensor receives a stream of actuation and sensing requests of the form $[A_1 \ldots A_n S_1 \ldots S_m]^+$, and VSense determines the interleaving of these requests on the physical sensor. The challenge is to determine an interleaving that isolates performance, while optimizing both actuation overheads and state restorations cost. VSense employs a two step process to determine an interleaved schedule (see Figure 3). First, VSense groups requests *within* each vsensor request stream to construct atomic units—analogous to critical sections—to issue to the physical sensor. After transforming each request stream into a stream of critical sections, VSense then determines an interleaving of the critical sections to optimize actuation costs.

In the first step, VSense transforms a request stream into a sequence of critical sections by grouping requests that have the form $A_i^* S_i^+$. The regular expression captures a key characteristic: each critical section consists of zero or more actuations, followed by one or more sensing requests, all from the *same vsensor*. Notice that, by executing this group of requests atomically, VSense prevents interleaving of actuations from other vsensors that might modify the current sensor state, which in turn avoids needless state restoration overheads. Importantly, VSense automatically infers these critical sections from each vsensor request stream without requiring any VM-level support—consistent with our goal of running unmodified applications.

VSense then employs an enhanced fair-share scheduler that is *actuation-aware*. Scheduling and interleaving occurs at the granularity of critical sections; our enhanced scheduler determines an ordering of critical sections that reduces actuation costs. The challenge lies in balancing fairness with actuation efficiency: VSense provides a configurable schedul-

ing parameter to determine an appropriate balance between these two objectives. The scheduler also enables performance isolation and fair resource allocation by assigning a user-specified weight w_i to each vsensor and allocating a minimum fraction $w_i/\sum_j w_j$ of the time on the physical sensor to vsensor i.

Finally, VSense must track the state of each vsensor and that of the physical sensor and perform state restoration whenever it detects a state mismatch at the time of a "context switch"—i.e., switching from one vsensor's critical section to another. VSense uses a *finite state machine* (FSM) for the physical sensor and each vsensor to track its current state, and employs an intelligent state restoration mechanism to further reduce the restoration overhead.

4 FSM-driven Interleaving

In this section, we present the state machine approach VSense employs to track the state of each virtual and physical sensor. VSense uses state machines for two key tasks: (i) to automatically infer critical sections from the request stream seen at each vsensor, and (ii) to perform state restoration that minimizes overhead.

4.1 Sensor State Machines

VSense uses finite state machines to track the state of each physical and virtual sensor, where a state is a specific setting of each sensor actuator. We use the term actuator broadly to include both mechanical actuators, as well as other non-mechanical settings of interest. For instance, a PTZ camera's state includes both the pan, tilt, and zoom position of its lens, as well as the image resolution and shutter speed settings. Pan and tilt are true mechanical actuators that require a motor to alter, while zoom, shutter speed, and image resolution are settings of the lens, camera, and CMOS sensor, respectively. Each actuation modifies the state of one or more of these parameters, causing the sensor to transition from one state to another.

VSense employs a virtual state machine (VSM) to track the current state of each vsensor and a physical state machine (PSM) to track the state of the physical sensor. The state of a vsensor (and hence the VSM) changes only when the corresponding application actuates it. In contrast, the state of the physical sensor (and the PSM) depends on which vsensor request is currently executing on the physical sensor. Thus, the PSM and VSM state machines allow VSense to track the state expected by each user, as well as the current state of the underlying physical sensor. Whenever VSense switches from one vsensor to another, it compares the states of the VSM and PSM. If there is a state mismtach, VSense must perform state restoration, by issuing actuation commands for each state parameter that is out-of-sync, to synchronize the vsensor's state with that of the physical sensor.

As an example, assume that Alice's virtual camera has the state $pan = \theta_a\ tilt = \phi_a\ zoom = Z_a$ (for simplicity, we ignore other camera settings here). Suppose the PSM of the physical camera has the state $pan = \theta_b\ tilt = \phi_a\ zoom = Z_b$. Thus, the two state machines are out-of-sync along the pan and zoom dimensions but in-sync along the tilt dimension. VSense synchronizes Alice's VSM state with the PSM by issuing a pan command to move the camera from θ_b to θ_a and a zoom command to change the zoom setting from Z_b to Z_a. No synchronization action is necessary along the tilt dimension. We refer to this strategy as *eager* state restoration since VSense synchronizes states eagerly at "context switch" time.

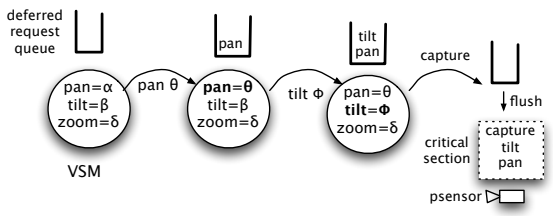


Figure 4: Request emulation and deriving critical sections.

4.2 Inferring Critical Sections

Given a sequence of requests arriving at each vsensor, VSense automatically groups requests that the physical sensor should execute atomically. As explained earlier, to minimize state restoration overheads, VSense attempts to group a sequence of zero or more actuation commands, followed by one or more sense requests from a vsensor into a critical section to avoid interference between actuation commands of competing users. However, automatic derivation of such critical sections is non-trivial. The difficulty stems from the nature of sensing and actuation requests, which are typically blocking system calls executed synchronously on the underlying physical sensor. Since applications block on these calls until completion, vsensors only see a single request at a time, which does not permit batching multiple requests into atomic units.

VSense must enable asynchronous execution of blocking requests in order to "batch" requests into a group and execute it atomically as a critical section. To address the problem, we employ a novel mechanism that *emulates* the execution of requests on the vsensor and defers their actual execution on the physical sensor. Request emulation allows the vsensor to behave as if the request actually executed on the sensor and allows the blocking call to complete. Emulation requires the vsensor to capture all state changes that result from executing the request by transitioning a vsensor's VSM to the new state dictated by the request. From the application's point-of-view, the request appears to complete successfully on the vsensor, allowing it to continue execution by issuing subsequent requests. VSense queues each of these emulated requests and defers execution on the physical sensor until a later time.

To ensure correctness in the presence of request emulation, VSense classifies all sensing and actuation requests as either safe or unsafe. Emulating safe requests does not alter an application's control flow because it does not return data to the application. In general, all actuation commands are safe since applications do not observe their effects until gathering data from subsequent sense requests. For example, emulating a change in the camera's pan position is possible by "moving" the virtual camera's pan setting. Emulating the effect of panning by changing the vsensor's state without changing the physical sensor's state does not alter the application's cor-

rectness since it does not change the application's inputs. In contrast, *unsafe* requests return data to the application that alters its control flow. In general, sense requests are unsafe since they return data which VSense cannot emulate; VSense must execute them on the physical sensor with an appropriate setting of the actuators to return correct data.

Thus, it is possible to emulate the execution of a sequence of safe (actuation) requests and defer their actual execution. Hence, VSense continues in request emulation mode until an unsafe request arrives, which causes it to flush the queue of deferred (safe) requests. VSense groups all flushed safe requests with a corresponding unsafe request into a single atomic unit, and issues this request block to the physical sensor for execution as a critical section. From an application's point-of-view, unsafe requests block until the critical section completes execution and the result of the unsafe (sense) request returns. Overall, safe requests execute asynchronously and unsafe requests execute synchronously, without any application modifications; the application continues to make blocking calls where each call appears to execute normally as a synchronous operation.

As an example, consider how Alice's virtual camera sensor maps onto a physical camera. Assume that Alice issues a *pan* command to position θ_a. Pan is an actuation command that is safe to emulate. Request emulation involves triggering a state transition in the VSM, causing the virtual camera pan position to change to θ_a, as shown in Figure 4. The figure also shows that VSense queues the request for deferred execution. Once the blocking pan completes, Alice's application continues execution and issues a *tilt* command to position ϕ_a. Since tilt is also a safe command, request emulation continues, triggering another state transition in the VSM to capture its effects. Next, Alice's application issues a capture command.

Since capture is a sensing request, it is unsafe; only the physical sensor is able to capture the correct image and return it to the application. Hence, VSense groups the capture command with the batch of pending requests in the deferred execution queue to form a critical section consisting of a pan, tilt and capture. This group is sent to the physical camera for execution as an atomic unit. Note that, while VSense has already emulated the execution of pan and tilt on the virtual camera, it must still execute these actuation commands on the physical camera in order to capture the correct image. Alice's application blocks until the critical section executes and returns the results of the capture request.

4.3 Intelligent State Restoration

VSense interleaves critical sections from different vsensors on the physical sensor and performs state restoration when switching from one user's critical section to another. Section 4.1 describes a simple eager state restoration strategy that compares the VSM state at the end of the previous critical section to the current PSM and synchronizes the PSM state by issuing the appropriate actuation commands. However, such eager state restoration imposes a higher overhead than necessary.

Recall the example from Section 2.2, where Alice issues a critical section P_aC_a, followed by a second critical section $P_a^2C_a^2$. The pan P_a in the first critical section causes the camera to move to position θ_a. After this critical section completes, suppose that Bob's request P_bC_b executes next, causing the camera to pan to a different position θ_b. Before executing Alice's second critical section, the eager restoration strategy restores the pan state of the camera by moving it from the current position θ_b to position θ_a. As depicted in Figure 1(c), the approach is wasteful, since the second critical section includes a new pan request P_a^2 that moves the camera to position θ_a^2; it may be more efficient to move the camera directly from θ_b to θ_a^2 instead of executing two pan actuations.[1]

VSense employs an intelligent state restoration strategy to reduce this overhead. Let VSM_{prev} denote the VSM state at the end of the previous critical section, and let VSM_{curr} denote the current VSM state. Observe that the set of safe requests in the critical section cause the vsensor to transition from VSM_{prev} to VSM_{curr} during request emulation. Let $VSM_{prev} \cap VSM_{curr}$ denote the set of state parameters *not* modified by the safe requests. Our intelligent state restoration strategy only restores the state of those parameters not modified by the subsequent critical section, which are precisely those in the set $VSM_{prev} \cap VSM_{curr}$.

VSense need only consider these state parameters when comparing and synchronizing with the physical state machine; other state parameters will be modified by actuation requests in the critical section and need not be restored. In the Alice and Bob example, $VSM_{prev} \cap VSM_{curr}$ includes the parameters zoom and tilt, but not pan. Since the second critical section modifies the pan parameter, VSense does not restore it. Thus, our intelligent restoration strategy avoids wasteful actuations by not restoring the settings that VSense will modify in the critical section.

One caveat of the strategy is that it will not work for actuation commands that use relative, rather than absolute, values. A relative actuation specifies an action in relation to the actuator's current setting. For instance, an actuation that pans the camera by 30° to the right or increases the lens aperture to the next f value, instead of specifying an absolute pan position or an absolute aperture value. Such relative actuation commands will not execute correctly if VSense does not restore the physical camera state to the previous user state. However, since VSense encounters all safe actuation requests during request emulation, it knows whether the actuation requests in the critical section specify absolute or relative values. VSense simply sets a flag if it encounters an actuation request with relative values, and, if the flag is set, performs full eager restoration prior to executing the critical section.

5 Actuation-aware Fair Queuing

In the previous section, we described how VSense uses state machines to group requests from a vsensor into critical sections. We now describe how VSense schedules critical sections from different vsensors on the physical sensor. VSense employs an enhanced proportional-share scheduler that: (i) allocates a certain minimum share of the sensing and actuation resources to each vsensor, and (ii) optimizes mechanical actuation costs incurred while scheduling requests

[1] To see why, suppose $\theta_b = 50^\circ$, $\theta_a = 30^\circ$ and $\theta_a^2 = 75^\circ$. Eager restoration will involve pans $50^\circ \rightarrow 30^\circ \rightarrow 75^\circ = 65^\circ$, while a direct pan from 50° to 75° requires only a 25° movement.

from different vsensors. We first describe how to adapt a proportional-share scheduler to schedule sensor requests and then propose an actuation-aware enhancement that optimizes actuation costs when scheduling requests.

5.1 Proportional-share Sensor Scheduling

The design of proportional-share schedulers has seen significant attention over the past two decades. Several proportional-share scheduling algorithms exist for CPUs [9, 7], NICs [6, 2, 10, 21], and disks [16]. We base our approach on *Start-time Fair Queuing (SFQ)* [9], a proportional-share scheduler originally designed for weighted fair sharing of CPUs and NICs. A proportional-share scheduler such as SFQ assigns a weight w_i to each vsensor and allocates $w_i/\sum_j w_j$ of the physical sensor's time to vsensor i. Controlling the weight assignment alters the share and performance of a vsensor's acuators: a smaller weight results in a smaller share and slower actuation. For example, a weight assignment in a 1:2 ratio for Alice and Bob results in an allocation of $1/3$ and $2/3$ of the physical sensor's time, respectively.

An ideal fair scheduler guarantees that over any time interval $[t_1,t_2]$, the service received by any two vsensors i and j is in proportion to their weights, assuming continuously backlogged requests at each vsensor during the interval. Thus, $\frac{W_i(t_1,t_2)}{W_j(t_1,t_2)} = \frac{w_i}{w_j}$, or equivalently, $\frac{W_i(t_1,t_2)}{w_i} - \frac{W_j(t_1,t_2)}{r_j} = 0$, where W_i and W_j denote the aggregate service each vsensor receives over the interval $[t_1,t_2]$. In our case, the aggregate service denotes the total time the (dedicated) physical sensor consumes scheduling a vsensor's request during the interval.

Ideal fair scheduling is possible only if the physical sensor is able to divide each vsensor actuation into infinitesimally small time units. Since VSense schedules at the granularity of critical sections, enforcing the ideal notion of fairness is not possible. Instead SFQ bounds the resulting unfairness due to this "discrete" granularity scheduling by ensuring that $|\frac{W_i(t_1,t_2)}{w_i} - \frac{W_j(t_1,t_2)}{w_j}| \leq (\frac{l_i^{max}}{w_i} + \frac{l_j^{max}}{w_j})$ for all intervals $[t_1,t_2]$, where l_i^{max} is the maximum length of a critical section from vsensor i. Intuitively, the unfairness bound is a function of the maximum length of time SFQ allocates the physical sensor—the maximum length of a critical section—to each vsensor.

We define the SFQ algorithm for scheduling critical sections in VSense as follows. For ease of exposition, we will use the terms critical sections and requests interchangeably: SFQ maintains a queue of pending requests for each vsensor.

- Upon arrival, the scheduler assigns each request r_i^k with a start tag $S(r_i^k)$, where $S(r_i^k) = max(v(A(r_i^k), F(r_i^{k-1}))$, r_i^k denotes the k^{th} request of vsensor i, $F(r_i^{k-1})$ denotes the finish time of the previous request, $v(t)$ represents virtual time, described below, and $A(t)$ represents the actual arrival time of the request. The start tag of a request is the maximum of the virtual time at arrival or the finish tag of the previous request.

- The finish tag of a request is $F(r_i^k) = S(r_i^k) + \frac{l_i^k}{w_i}$, where l_k^k denotes the length of the k^{th} request and w_i denotes the weight assigned to vsensor i. Intuitively, the finish tag of a request is its start tag incremented by the length of time required to execute the entire critical section, normalized by the vsensor's weight. To enable precise computation of l_k^k, SFQ computes the finish tag *after* the request/critical section completes execution. Once SFQ computes a request's finish tag, it computes the start tag of the next request in its queue.

- The scheduler starts at virtual time 0. During a busy period—when the scheduler is continuously scheduling requests on the physical sensor—SFQ defines the virtual time at time t, $v(t)$, to be the start tag of the request currently executing. At the end of a busy period, SFQ sets the virtual time to the maximum finish tag of any request completed during this busy period. The virtual time does not increment when the physical sensor is idle.

- The scheduler always schedules the request with the minimum start tag next, ensuring that it schedules the vsensor with the minimum weighted service thus far. This is the key property that ensures each vsensor receives its fair share of the psensor over time. Note also that scheduling the request with the minimum start tag implies that the virtual time during a busy period is always equal to the minimum start tag of any request in the system.

Intuitively, SFQ advances start and finish tags in weighted proportion to each request's length; thus, the request with the minimum start tag represents the vsensor that has received the least service in proportion to its weight. In this case, an actuation request's length corresponds to the time it takes to execute the request on the dedicated sensor. SFQ has several salient properties that are useful when scheduling request from vsensors.

First, a vsensor is unable to accumulate "credits" for any unused allocation from the past. The property ensures fairness since a vsensor is unable to remain idle for a long period of time, accumulate credits, and then hoard the sensor once active, thereby starving other well-behaved vsensors. The "use it or lose it" property derives from assigning a start tag that is at least equal to the virtual time in the system; since virtual time advances so long as some vsensor is active, the start tag cannot lag behind that of the least-serviced vsensor.

Second, and a direct consequence of the above property is that SFQ fairly reallocates any unused resources among active vsensors in proportion to their weights. Thus, if a vsensor does not use its fair share allocation, SFQ automatically reallocates the unused sensor capacity among the active vsensors. This *work-conservation* property ensures that SFQ does not waste capacity that is potentially useful to active sensors. Work-conservation is a property of SFQ that requires no additional mechanisms. For example, if three vsensors have weights 1:1:1, each receives a share of $1/3$ if all three are active. However, if a vsensor becomes idle, its start tag stops advancing, and SFQ automatically schedules requests from the others, causing their start tags to advance faster and in proportion 1:1 (resulting in 50% allocation for each). When the idle vsensor becomes active again, the respective shares revert back to 33% each. We demonstrate this behavior experimentally in Section 7.

5.2 Actuation-aware SFQ

While our adaptation of the original SFQ algorithm to schedule sensors is straigtforward, its use is not without problems in our context. The primary difficulty arises due to the high mechanical actuation costs of sensors. SFQ enables fair sharing of the physical sensor while completely ignoring actuation costs when scheduling requests. By not considering actuation costs, SFQ, while fair, yields significant inefficiencies. As an example, consider three users Alice, Bob and Carol who share a PTZ camera. Suppose that Alice issues a pan request to position 30°, while Bob and Carol issue pan requests to positions 75° and 40°. Assume the camera is currently at position 25°, and Alice, Bob and Carol have start tags of 10, 11 and 12. SFQ services these requests in the order of Alice, Bob, and Carol, triggering pans from $25° \rightarrow 30° \rightarrow 75° \rightarrow 40°$ (total of 85° pan movement). However, since Alice and Carol's requests are "close" to each other, it is more efficient to service the requests in the order Alice, Carol, Bob. The reordering incurs a lower total pan overhead of only 50° ($25° \rightarrow 30° \rightarrow 45° \rightarrow 75°$). Since SFQ is actuation-oblivious, it does not account for actuation overheads when scheduling requests. To address this limitation, we present an enhanced actuation-aware scheduler that optimizes these overheads.

By simply considering requests in order of their start tags, SFQ misses opportunities to optimize actuation overheads. In contrast, our actuation-aware fair queuing (AFQ) considers both start tags (for fairness) and actuation costs (for efficiency). Rather than considering the request with the least start tag, AFQ chooses k pending requests with the smallest start tags, one from each vsensor; $k \geq 1$. Given this batch of k requests, AFQ then reorders these requests to minimize the physical sensor's total actuation time, and schedules these requests in that order. In the above example, AFQ would determine that scheduling requests in the order Alice, Carol and Bob yields a lower total pan time and reorder them.

For a sensor with a single actuator, the problem is analogous to a disk controller that strives to optimize seek overheads to increase global I/O throughput. However, sensors consist of multiple independent actuators (e.g., pan, tilt, zoom, etc., for a PTZ camera). As a result, the general problem of minimizing actuation time across multiple actuation dimensions is equivalent to the well-known NP-hard Traveling Salesman Problem (TSP). AFQ uses the greedy nearest neighbor heuristic for TSP that greedily selects the next "closest" request in terms of actuation time. For small values of k, a brute force search that tries all permutations is also feasible.

AFQ's parameter k defines a tradeoff between fairness and efficiency for stateful actuators: the higher the value of k the more efficient, but less fair, the schedule. Thus, if $k = 1$, the approach reduces to pure SFQ by choosing a batch of one request with the least start tag. Larger values of k allow more opportunities for optimizing actuation cost at the expense of greater unfairness. In Section 7.3, we show that a value of k that is close to half the number of vsensors N in the system strikes a good balance between fairness and efficiency.

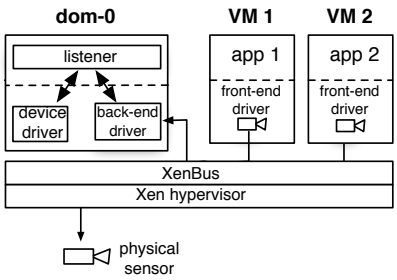


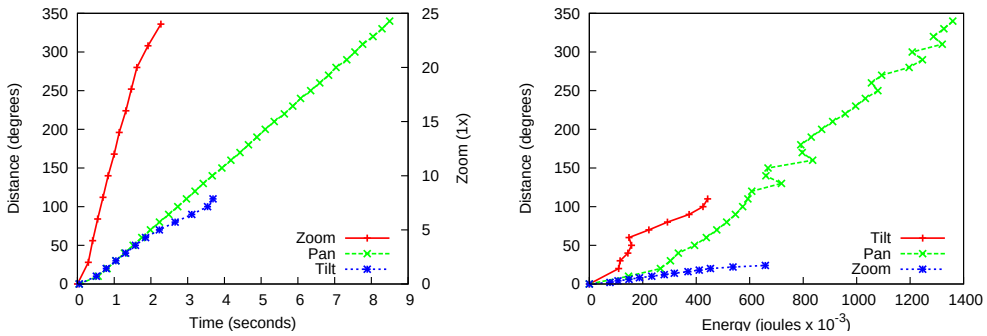
Figure 5: VSense's implementation using Xen's split-driver framework to serve as a communication channel and a user-level listener in `domain-0` to maintain vsensor VSMs and execute scheduling policies. Each request passes from application → front-end driver → back-end driver → listener → device.

6 VSense Implementation

We have implemented VSense in the Xen VMM, a widely-used open-source virtualization platform, to virtualize two representative examples of sensors with actuators: the Sony SNC-RZ30N PTZ Network Camera and the DavisPro Vantage2 Weather Station. In this paper, we concentrate on the PTZ camera, since it exposes multiple actuators with a range of settings, whereas the weather station only exposes a single coarse actuator—sensing rate settings of 5, 10, and 15 minutes. VSense enables applications in each VM to actuate both devices using their standard protocols. Below we detail how VSense integrates into XenLinux's virtual device framework and combines the elements—inferring critical sections, intelligent state restoration, and AFQ—from the previous sections.

High-power sensors are typically character devices that transfer streams of data to applications. Our example sensors use serial connections—RS-232 and USB— for communication. In Linux, user-level applications use character device files to interact with character devices. Each character device file supports 5 basic functions: `open`, `close`, `read`, `write`, and `ioctl`. Drivers typically use open and close to track the user-level applications using the device, read and write to transfer data to and from the device, and ioctl to actuate the device. To virtualize device drivers, Xen uses a "split-driver" approach that divides conventional driver functionality into two halves: a *front-end driver* that runs in each VM and a *back-end driver* that typically runs in `domain-0`, a privileged management domain. Details of the split-driver approach can be found in [1].

Figure 6 depicts VSense's Xen implementation. To build VSense, we implemented a generic front-end character driver for Xen that passes the front-end's open, close, read, write, and ioctl requests to the back-end driver, which executes them and returns the response. As with other character drivers, the front-end/back-end communication channel supports multiple threads to permit asynchronous device interactions. In VSense's current implementation the back-end driver passes requests to a user-level listener using the back-end's read and write system calls: the listener receives new requests us-



(a) Time Benchmark (b) Energy Benchmark

Figure 6: Time (a) and energy (b) benchmarks for the pan, tilt, and zoom actuators of Sony SNC-SRZ30N Network Camera. For both benchmarks, the distance moved by the actuator is roughly linear to the time and energy required to complete the actuation.

ing read and issues request responses using write. The listener includes the logic to infer critical sections, perform state restoration, and schedule actuations, and uses the sensor's conventional device driver as the mechanism to actuate the physical device.

Using a user-level listener in lieu of a kernel-mode listener has two main advantages. First, some sensor manufacturers release binary-only device drivers for Linux that are only accessible from user-level. Second, implementing the FSM and scheduler at user-level simplifies debugging. Since the dominant latency for our example sensors is actuation and not data transfer, as we show in Section 7.1, the overhead of moving data between kernel-space and user-space is negligible. For sensors where data transfer is the dominant cost, VSense could integrate the functions of the user-level listener as a separate kernel module that interacts with the back-end driver.

VSense's front-end and back-end drivers are reusable with different types of sensors since they act only as a communication channel for character device requests; we use the same pair for both the PTZ camera and the weather station. Sensor-specific state machine and scheduling functions reside in the user-level listener, which we customize for each device. The listener maintains a vector and queue for each vsensor that stores the current setting of its actuators and its backlog of deferred safe requests, respectively. When a safe request arrives, the listener associates a start tag with it, places it an the end of its vsensor's queue, and changes the actuator's vector entry. When an unsafe request arrives, the listener flushes the deferred vsensor requests in order of minimum start tags to a common queue used by AFQ. As soon as k requests arrive or time t passes since the last scheduling opportunity, the AFQ scheduler reorders the requests in the common queue using the greedy nearest neighbor heuristic, issues them to the device driver of the physical sensor, and returns the response, as described in Section 5.2.

One consequence of request emulation using critical sections is that applications do not perceive errors from actuations. Since our implementation defers critical sections until an unsafe request arrives, we also defer any errors as a result of actuations to the execution of an unsafe request.

From	**→ To**	**Latency**	**Percentage**
application	→ front-end	0.24 μsecs	$7.1x10^{-8}$
front-end	→ back-end	6.35 μsecs	$1.9x10^{-4}$
back-end	→ listener	286 μsecs	$8.51x10^{-3}$
listener	→ camera	274 μsecs	$8.15x10^{-3}$
camera	→ listener	3.35 secs	99.7
listener	→ back-end	17 μsecs	$5.1x10^{-4}$
back-end	→ front-end	27 μsecs	$8.0x10^{-4}$
front-end	→ application	229 μsecs	$6.8x10^{-3}$
total		**3.36** secs	**100**

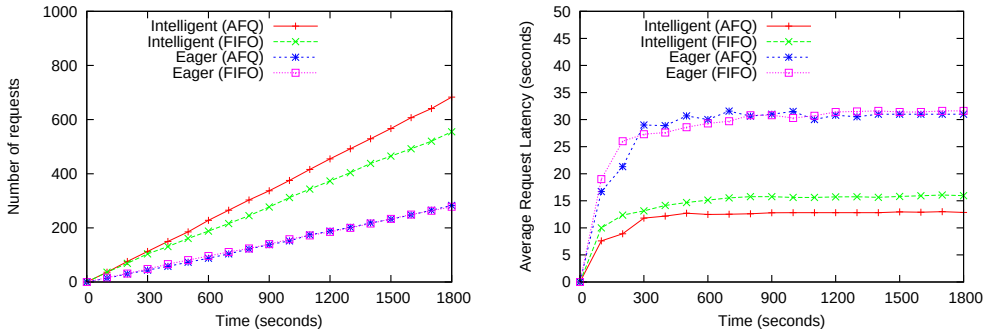
Table 1: Latency breakdown for a sample vsensor actuation of the Sony PTZ camera in our Xen implementation. The dominant factor in the request latency ($> 99.7\%$) is the time to actuate the camera. Our implementation imposes comparatively little overhead ($< 0.3\%$).

Thus, applications only perceive errors as a result of unsafe (sense) requests, and not the actuation causing the request. We are currently exploring how the approach potentially affects application-level failure diagnosis strategies.

7 Experimental Evaluation

We evaluate VSense using the Sony SNC-RZ30N PTZ Network Camera. PTZ cameras are a common example of of a sensor with actuators; the SNC-RZ30N exposes the three obvious actuators—pan, tilt, and zoom—that we concentrate on, along with many non-obvious actuators, including resolution setting, shutter speed, backlight compensation, night vision, and electronic stabilization, that influence an image's fidelity. We first benchmark the capabilities of the PTZ camera to determine both the time and energy to actuate it, and the overheads incurred by VSense's virtualization layer (Section 7.1). Our evaluation then addresses three questions:

- **How do different approaches to state restoration affect performance?** We compare the efficiency—as measured by the time to complete each actuation request—of eager state restoration and intelligent state restoration using critical sections. We show for our workloads that inferring critical sections and using intelligent state restoration performs 3x better than an ea-



(a) Number of Requests (b) Average Request Latency

Figure 7: Intelligent state restoration that infers critical sections and defers their execution significantly outperforms a simple eager approach in our sample workloads for two different scheduling policies—FIFO and AFQ. The number of requests completed (a) is 3x more and the average latency to satisfy each request (b) is 3x less using the intelligent approach.

ger approach that restores state on each context-switch (Section 7.2).

- **How strong is performance isolation and what are its limits?** We evaluate SFQ and AFQ's ability to enforce performance isolation between vsensors. We show that SFQ enforces performance isolation, but is inefficient. We then explore how AFQ's batch size parameter defines a tradeoff between performance isolation and efficiency. We find that in practice a batch size equal to half the number of active vsensors strikes a good balance between fairness and efficiency (Section 7.3).
- **Can concurrent applications satisfy conflicting goals using VSense?** The ultimate judge of VSense's utility is whether concurrent applications are able to satisfy conflicting goals. We use case studies of three different applications—fixed-point sensing, continuous monitoring, and object tracking—to demonstrate VSense's ability to satisfy concurrent goals (Section 7.4). We show that these applications receive tolerable, but degraded performance (ranging from 1.5x less to 8x less), when running concurrently for varying weight assignments.

Each experiment runs on a Mac-Mini with an Intel T5600 CPU running at 1.83Ghz, 1 gigabtye of RAM, and a single 80 gigabyte SCSI disk. The node runs the Xen hypervisor (version 3.2) with an Ubuntu Linux distribution using kernel version 2.6.18.8-xen in both Xen's privileged `domain-0` VM and in each guest VM. Each VM uses a file-backed virtual block device to store its root file system image. The PTZ camera is capable of panning between -170° and 170° and tilting between -90° and 25° of center, while supporting 25 different optical zoom settings (1x to 25x). The camera's direct drive motor allows precise control of pan and tilt increments as small as $1/3^\circ$.

In addition to our application case studies, our evaluation uses both deterministic and random workloads. The deterministic workloads perform *continuous scans* in a single plane—either pan or tilt—of the camera's lens in a single direction, while the random workload repeatedly requests a random setting of the pan, tilt, and zoom actuators and captures an image. Each sequential scan captures an image every 10° starting at one extreme of the plane (*e.g.*, (-170° for pan and -90° for tilt) and moving to its other extreme (*e.g.*, 170° for pan and 25° for tilt). The workloads stress VSense by forcing the camera's lens to move to its extreme points in every direction, while also satisfying random requests.

7.1 Benchmarks

Figure 6(a) shows the time to alter the settings of the pan, tilt, and zoom actuators as a function of distance. Each point in the graph reports an average of 10 independent trials. The benchmark validates two underlying assumptions of our work. First, the camera's mechanical actuators are slow: panning all 340° takes 9 seconds, tilting all 115° takes 3 seconds, and zooming from 1x to 25x takes 4 seconds. Second, the time to set an actuator is dependent on its current state. For pan, tilt, and zoom the time is roughly linear to the distance the actuator must move. We also report energy consumption in Figure 6(b), which follows the same linear trend, to emphasize the point that actuation consumes not only a sensor's time, but also its energy. While both are critical resources, this paper focuses on multiplexing in time and leaves energy as future work.

Table 1 reports the overhead VSense imposes on a single vsensor actuation request and its response as it flows from the application to the camera and then back to the application. Xen adds two additional layers in the flow—the front-end and back-end device driver—while VSense adds an additional layer by using a user-level listener in `domain-0`. As Table 1 shows, the overhead of these additional layers are minimal compared (order of μseconds) to the overhead of actuating the camera (order of seconds). Thus, the speed of mechanical actuators does not preclude the increased overhead of virtualization.

7.2 State Restoration

As discussed in Section 4, different approaches to state restoration are independent of the scheduling policy that determines the interleaving of critical sections. However, the state restoration approach does have a significant impact on the camera's throughput—the number of requests it is able to satisfy in a given time period. Figure 7 compares the simple eager state restoration approach with our intelligent

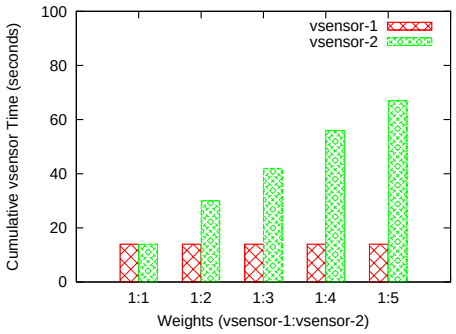


Figure 8: SFQ enforces performance isolation over large numbers of requests. The ratio of the total vsensor time for the two continuous scan workloads is in proportion to the assigned weights.

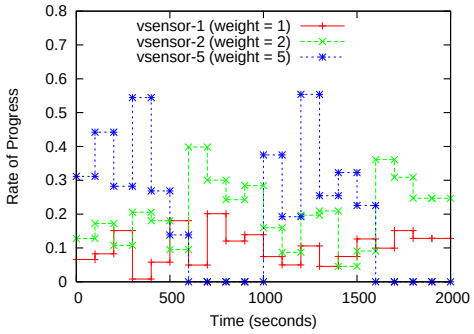


Figure 9: SFQ maintains its work-conserving properties when applied to actuators.

approach that infers critical sections and defers their execution until an unsafe request occurs. We examine the effects of state restoration using two different scheduling policies—FIFO and AFQ—with five vsensors executing the random workloads described earlier.

Figure 7(a) shows the progress of completed requests on the physical sensor for each state restoration approach (Eager or Intelligent) and scheduling algorithm (FIFO or AFQ) pair, while Figure 7(b) plots the average latency to satisfy each request. As expected, for both scheduling algorithms, the intelligent approach is significantly more efficient: it is able to complete nearly 3x as many requests during the same 30 minute time period with 3x less latency on average per request. Also as expected, AFQ is more efficient than FIFO for intelligent state restoration, completing more requests with a lower average latency.

Interestingly, for eager state restoration both FIFO and AFQ complete about the same number of requests with a similar average per-request latency. The reason is that the overhead of the eager state restoration strategy completely dwarfs any efficiencies derived from the scheduling algorithm. The result highlights the importance of deferring state restoration to gain efficiency: a poor state restoration strategy may cancel any benefits from a better scheduling algorithm.

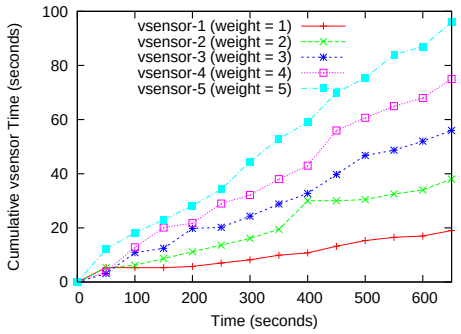


Figure 10: While SFQ enforces performance isolation over many requests, it may diverge slightly, due to high state restoration costs, over short intervals of time.

7.3 Performance Isolation: SFQ and AFQ

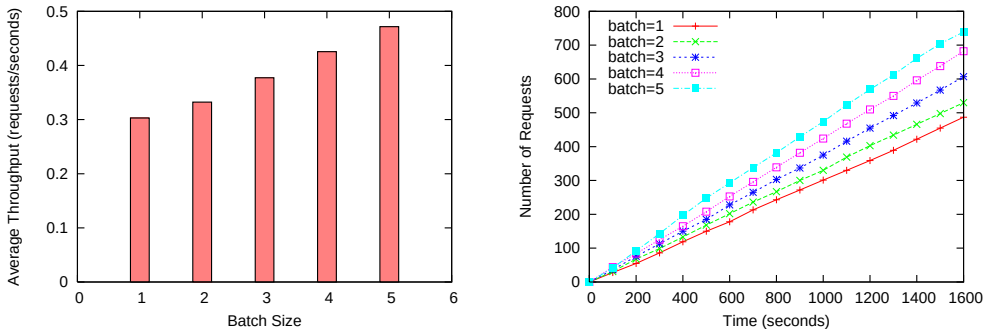
SFQ enforces performance isolation between vsensors—each vsensor should receive actuator performance in proportion to its weight or share—while AFQ relaxes SFQ's fairness bounds to ensure efficient use of each actuator. We first demonstrate SFQ's performance and limitations when used for actuators and then present results from our AFQ extension.

7.3.1 SFQ

The primary attribute that determines vsensor performance is the relative *speed* of its actuators. Our adaptation of SFQ advances virtual time in relation to the *time* each actuation takes on the dedicated sensor, which we denote as vsensor time. Thus, the more vsensor time each actuation takes the slower the actuator. Figure 8 shows the total vsensor time of two vsensors with different weight assignments using our variant of the SFQ scheduling algorithm; each vsensor executes the continuous scan workload. The figure demonstrates that a straightforward use of SFQ for actuators isolates vsensor performance: the cumulative vsensor time SFQ allocates is in proportion to the assigned weights.

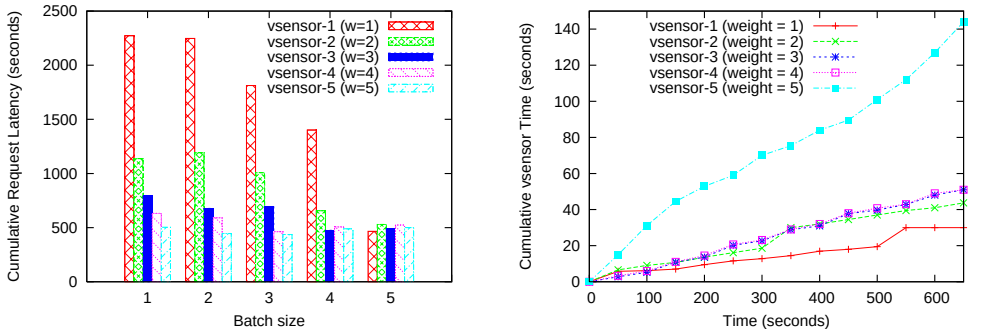
Another important characteristic of SFQ is work-conservation: it automatically reallocates idle resources to vsensors in proportion to their weights. Figure 9 demonstrates VSense's responsiveness to fluctuating workloads. We use three vsensors in the experiment—vsensor-1, vsensor-2, and vsensor-5—where vsensor-1 and vsensor-2 run the continuous scan workload and vsensor-5 runs the random workload; vsensor-5's workload sends actuation requests for 500 seconds and halts for 500 seconds in a repeating loop. As expected, when vsensor-5 becomes inactive, SFQ reallocates the idle actuation time to vsensor-1 and vsensor-2 in proportion to their weights: the result is that each vsensor's rate of progress, measured as the rate its vsensor time increases every 50 seconds, immediately increases for vsensor-1 and vsensor-2 during vsensor-5's idle periods. Thus, SFQ maintains its work-conserving properties in our new context.

While SFQ enforces performance isolation over large numbers of requests, high state restoration costs may cause it to perform unfairly over short intervals. To demonstrate the point, Figure 10 shows how the cumulative vsensor time pro-



(a) Average Throughput (b) Number of Requests

Figure 11: AFQ shows better global performance—in terms of both the average throughput in requests/second (a) and the total number of requests the camera completes (b)—as its batch size increases. For this experiment, each increment in the batch size results in roughly a 10% improvement for both metrics.



(a) Fairness vs. Batch Size (b) Unfairness with Batch Size=4

Figure 12: AFQ exhibits less fairness as the batch size increases (a) in terms of the average latency per request. For this experiment, batch sizes of 4 and greater are unfair, and exhibit much less performance isolation (b) than SFQ.

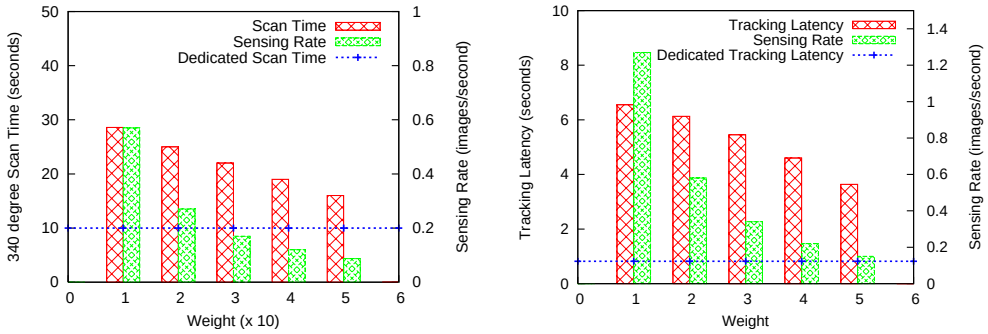
gresses over the course of an experiment. Since each workload includes 100 requests, at any point in time the cumulative vsensor time for each vsensor should be in proportion to the assigned weights. The experiment uses five vsensors—four running the continuous scan workload (1-4) and one running the random workload (5).

The result demonstrates that over short time periods SFQ is not always fair: during the period 0-100 seconds both vsensor-3/vsensor-4 and vsensor-1/vsensor-2 receive similar performance that is not in proportion to their weights. Further, vsensor-1/vsensor-2 receive similar performance by time 200 and vsensor-3/vsensor-2 receive similar performance up to time 400, which diverges from the weight assignments. However, as before, as VSense services larger numbers of requests, performance converges to the assigned weights (by time 550 seconds).

7.3.2 AFQ

Since SFQ does not take into account stateful actuators it may be inefficient, since it schedules requests based on strict fairness. Figure 11 demonstrates that increasing AFQ's batch size parameter increases the efficiency of the actuators. The experiment uses random workloads from 5 vsensors to stress the actuation of the system. The experiment shows that both the average throughput to complete requests (a) and the total number of requests completed (b) increases, as the batch size increases; for this experiment, each increment in the batch size results in roughly a 10% improvement for both metrics. However, AFQ's efficiency gain causes the scheduler to diverge from strict fairness, as we show next.

Figure 12(a) shows the cumulative request latency for the first 50 requests for each of five vsensors as a function of batch size, using the same five vsensors and workloads as Figure 10. The cumulative request latency is the sum of the latencies (equivalent to each vsensor's makespan) to satisfy all requests at each vsensor; as also demonstrated above, efficiency, in this case cumulative request latency decreases with batch size. As expected, SFQ, which corresponds to a batch size of 1, exhibits the strong performance isolation seen in Section 7.3.1. However, as the batch size increases, strict performance isolation decreases and causes the height of the bars to approach each other. Figure 12(b) plots the cumulative vsensor time over the course of the experiment for a batch size of 4; comparing the result with Figure 10 in the previous section demonstrates that AFQ exhibits less fairness with larger batch sizes. For these workloads, a batch size of 3 exhibits a nice balance—20% performance improvement over



(a) Continuous Monitoring (b) Object Tracking

Figure 13: VSense is able to satisfy conflicting goals from multiple concurrent sensing applications. A continuous monitoring application (a) and an object tracking application (b) both maintain tolerable performance for varying weight assignments, while competing with a fixed-point sensing application with weight 1.

SFQ with similar fairness properties—between performance and fairness; in practice, we have found that a batch size of roughly half the number of active vsensors generally strikes the appropriate balance.

7.4 Case Study

Our final set of experiments consider how three example camera applications with distinct performance metrics perform on VSense. The applications include:

- **Fixed-point Sensing**. Pan and tilt the camera's lens to a point and repeatedly capture images at a regular interval. The performance metric for this simple application is the sensing rate of image captures.
- **Continuous Monitoring**. Continuously pan in increments of 65° and capture an image. The application-level performance metric is the time to complete each 340° scan. This application mimics conventional security applications.
- **Object Tracking** Periodically scan a pre-defined path along both the pan and tilt axes and capture images every 10°. The application-level performance metric is the latency between each image capture. A latency too large and the camera will lose the object its tracking.

The fixed-point sensing application using a dedicated sensor has near video quality: its sensing rate is 11 images/second with a steady inter-image interval of 0.09 seconds. However, even on a dedicated sensor, actuation does have a significant effect on performance. Our random workload, which moves to a random location and captures an image, reduces the average sensing rate to 0.3 images/second with an average inter-image interval of 3.35 seconds (the interval is not steady since the camera moves a different distance between each image). Similarly, two fixed-point sensing applications—at a distance of 180°—are both able to capture 0.2 images/second with a steady inter-image interval of 4.65 seconds (since their distance apart does not change). We use these sensing rates for comparison in our case study below.

We run both the continuous monitoring (Figure 13(a)) and object tracking (Figure 13(b)) application against the fixed-point sensing application. In both experiments, the fixed-point sensing application maintains a weight of 1, while we vary the weights assigned to continuous monitoring and object tracking. Figure 13 shows the results, where the leftmost *y*-axis plots the application's performance metric, the rightmost *y*-axis plots the fixed-point sensing application's performance metric (sensing rate), and the dotted line depicts the performance of the application on a dedicated sensor. The results demonstrate that VSense is able to satisfy conflicting demands of concurrent applications, as long as the applications tolerate less performance than possible with the dedicated sensor, which ranges from 1.5x to 8x less performance for the different weight assignments in this experiment. As we describe below, the reduced performance is still capable of satisfying real-world application-level goals.

As one example, with a 1:30 weight ratio, the continuous monitoring application is able to pan all 340° in 20 seconds. Thus, in the real-world, the monitoring application is able to capture 4 distinct points 113 feet apart (e.g., four doorways) at distance of 100 feet from the camera every 5 seconds [2]. The fixed-point sensing application is able to simultaneously maintain a sensing rate of nearly 0.2 images/second, allowing it to continuously capture a single point (e.g., a nearby intersection) in spite of the monitoring application. Likewise, for a 1:3 weight ratio, the object tracking application is able to scan a pre-defined path every 10° and capture images every 6 seconds, which is suitable for tracking a moving object at a distance of 300 feet moving at 2.66 miles/hour (e.g., a person walking) for up to 1779 feet (over 1/3 mile) of the object's motion with 25x zoom. Of course, both the specific speed and the total distance tracked are dependent on the object's trajectory, its distance from the camera, and the camera's optical zoom and resolution settings [3]. During the tracking, the fixed-point sensing application maintains a sensing rate of 0.3 images/second. These case study results, combined with our evaluation of state restoration, SFQ, and AFQ, demonstrate

[2]The example assumes the points are along a circle with radius 100 feet with camera's lens as its center.

[3]Our example assumes that the object's trajectory is along a circle of radius 300 feet with the camera's lens as its center.

the potential of multiplexing sensors at the level of individual actuations.

8 Related Work

Our work adapts existing techniques from many different areas, including sensor networks, platform virtualization, and proportional-share scheduling, to virtualize stateful sensors with actuators. We briefly review important topics in each of these areas.

Mote-class sensor networks primarily use virtualization as a mechanism for safe execution and reprogramming, as demonstrated by Maté [12], since motes are generally not powerful enough to execute multiple applications concurrently. While some recent mote-class OSes incorporate threads and time-sharing [4], the energy constraints of motes prevent them from using high-power sensors with rich programmable actuators, such as PTZ cameras or steerable weather radars. PixieOS [14] uses proportional-share scheduling techniques (in the form of *tickets*) to enable explicit conventional resource control (CPU, memory, bandwidth, energy) by individual mote applications; we extend similar proportional-share scheduling techniques to the equally important actuation "resources" of high-power sensors. Finally, ICEM also encounters a problem with blocking calls to peripheral devices when abstracting devices [11]; ICEM solves the problem for mote power management by exposing concurrency to drivers through power locks. In contrast, VSense does not change the application/device interface to support unmodified applications, and, instead, characterizes actuations as either safe or unsafe and uses request emulation to "complete" blocking calls asynchronously.

We build on Xen's [1] basic abstractions for virtualizing I/O devices in VSense [8, 19]. Modern VMMs, including Xen and VMware, focus on virtualizing the hardware at the lowest layer possible (*e.g.*, the PCI bus, the USB controller, etc.) to support unmodified device drivers. However, virtualizing at this layer requires the physical device to attach to a single VM and "pass-through" device requests to the physical bus [20]. We virtualize at the protocol layer—the character device file interface—so VSense can interpret each vsensor request and control their submission to the physical sensor. VSense's FSM that tracks the state of each vsensor is similar to shadow drivers [17], but we use them to ensure correct operation and enforce performance isolation and do not focus on reliability. Many prior approaches structure device drivers as state machines; the technique is natural for stateful devices [15].

Our adaptation of SFQ [10] is complementary to variants of proportional-share schedulers in other domains (CPU, disk, NIC), including recent work on energy [5]. To the best of our knowledge, this work is the first to apply the concept to stateful devices, such as sensors, where the "resource" is the time incurred to transition the device to a desired setting. Similar tradeoffs between efficiency and fairness also present themselves in prior work on proportional-share disk scheduling [16]. We apply the concept to sensors, but nearly every device exposes similar, though often obscure, actuator settings (e.g., wireless NICs expose a myriad of settings). We leave for future work how these concepts may apply to other types of stateful devices with actuators.

9 Conclusion

VSense adapts the virtualization paradigm to multiplex the "resource" of controlling a sensor's actuators. For high-power sensors with rich actuation capabilities, control of the actuators is a sensing application's most important resource since it determines the type of data the sensor collects. Even though these high-power sensors represent scarce resources that are costly to deploy, applications are currently unable to share them at fine time scales. We know of no prior work addressing fine-grained multiplexing of sensors. VSense elevates control of sensor actuators to a first-class resource in high-power sensor networks, along with CPU, memory, disk, bandwidth, and energy, to enable fine-grained sharing in these settings, and demonstrates its utility for real applications in a detailed case study.

10 References

[1] P.T. Barham, B. Dragovic, K. Fraser, S. Hand, T.L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen And The Art Of Virtualization. *Symposium on Operating Systems Principles*, October 2003.

[2] J.C.R. Bennett and H. Zhang. Wf2q: Worst-case Weighted Fair Queuing. *IEEE International Conference on Computer Communications*, June 2002.

[3] K. Binsted, N. Bradley, M. Buie, S. Ibara, M. Kadooka, and D. Shirae. The Lowell Telescope Scheduler: A System To Provide Non-professional Access To Large Automatic Telescopes. *Internet and Multimedia Systems and Applications*, August 2005.

[4] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS Operating System: Towards Unix-like Abstractions For Wireless Sensor Networks. *International Conference on Information Processing in Sensor Networks*, April 2008.

[5] Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. Abdelzaher. Virtual Battery: An Energy Reserve Abstraction For Embedded Sensor Networks. *Real-Time Systems Symposium*, November 2008.

[6] A.J. Demers, S. Keshav, and S. Shenker. Analysis And Simulation Of A Fair Queueing Algorithm. *SIGCOMM Conference*, September 1989.

[7] K.J. Duda and D.R. Cheriton. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-sensitive Threads In A General-purpose Schedular. *Symposium on Operating Systems Principles*, December 1999.

[8] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access With The Xen Virtual Machine Monitor. *Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, October 2004.

[9] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical Cpu Scheduler For Multimedia Operating Systems. *Symposium on Operating System Design and Implementation*, October 1996.

[10] P. Goyal, H.M. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm For Integrated Services Packet Switching Networks. *SIGCOMM Conference*, August 1996.

[11] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and Philip Levis Integrating Concurrency Control and Energy Management in Device Drivers. *Symposium on Operating Systems Principles*, October 2007.

[12] P. Levis and D. Culler. Maté: A Tiny Virtual Machine For Sensor Networks. *Architectural Support for Programming Languages and Operating Systems*, October 2002.

[13] M. Li, T. Yan, D. Ganesan, E. Lyons, P. Shenoy, A. Venkataramani, and M. Zink. Multi-user Data Sharing In Radar Sensor Networks. *Conference on Embedded Networked Sensor Systems*, November 2007.

[14] K. Lorincz, B. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource Aware Programming In The Pixie Operating System. *Conference on Embedded Networked Sensor Systems*, November 2008.

[15] T. Nelson. The Device Driver As State Machine. *In The C Users Journal*, 10(3):41-60, March 1992.

[16] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework For Next Generation Operating Systems. *ACM Sigmetrics Conference*, June 1998.

[17] M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy. Recovering Device Drivers. *Symposium on Operating System Design and Implementation*, December 2004.

[18] C.A. Waldspurger. Memory Resource Management In Vmware Esx Server. *Symposium on Operating System Design and Implementation*, December 2002.

[19] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating The Development Of Soft Devices. *USENIX Annual Technical Conference*, April 2005.

[20] L. Xia and J. Lange. Towards Virtual Passthrough I/O On Commodity Devices. *Workshop on I/O Virtualization*, December 2008.

[21] H. Zhang and S. Keshav. Comparison Of Rate-based Service Disciplines. *SIGCOMM Conference*, September 1991.