MultiSense: Fine-grained Multiplexing for Steerable Camera Sensor Networks

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

ABSTRACT

Steerable sensors, such as pan-tilt-zoom video cameras, expose programmable actuators to applications, which steer them in different directions based on their goals. Despite being expensive to deploy and maintain, existing steerable sensor networks allow only a single application to control them due to the slow speed of their mechanical actuators. To address the problem, we design MultiSense to enable fine-grained multiplexing by (i) exposing a virtual sensor to each application and (ii) optimizing the time to context-switch between virtual sensors and satisfy requests.

We implement MultiSense in Xen and explore how well proportional share scheduling, along with extensions for state restoration and request batching, satisfies the unique requirements of steerable sensors in the form of pan-tilt-zoom video cameras. We present experiments that show MultiSense efficiently isolates the performance of virtual cameras, allowing concurrent applications to satisfy conflicting goals. As one example, we enable a tracking application to photograph an object moving at nearly 3 mph every 23 ft along its trajectory at a distance of 300 ft, while supporting a security application that photographs a fixed point every 3 seconds.

Categories and Subject Descriptors

C.5.0 [**Computer System Implementation**]: General

General Terms

Design, Experimentation, Performance

Keywords

Virtualization, Sensor, Actuator, Pan-Tilt-Zoom

1. INTRODUCTION

Steerable sensor networks allow applications to adjust actuators that control the type, quality, and quantity of data they collect. [1] Steerable pan-tilt-zoom (PTZ) camera networks are an important example of this type of steerable system that are being deployed in a diverse range of settings. For instance, the U.S. Border Patrol is deploying networks of PTZ cameras to continuously monitor the northern border for smugglers [17], and as part of a "virtual fence" on the southern border [8]. Further, networks of traffic cameras that monitor urban environments are now commonplace. Another example of this type of system is steerable radar networks, which are able to improve the accuracy of weather forecasts [20]. While this type of networked cyber-physical system is emerging as a critical piece of society's infrastructure, the deployments are expensive: the hardware cost for the 20-mile prototype of the Border Patrol's "virtual fence" is over $20 million. Further, a key limitation of these systems is that they are not designed for multiplexing. Despite their expense, only a single user, or application, is able to control them.

Supporting concurrent users via fine-grained multiplexing is an important step in providing broader access to these expensive systems. As a simple example, consider using a PTZ camera 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 camera to track suspicious vehicles moving through its field of view. Each application alters the setting of three distinct actuators—pan, tilt, and zoom—to satisfy its goals. Conflicts such as these have been cited as one reason multiple government agencies are unable to coordinate control of border cameras for different purposes, including both smuggling and search-and-rescue operations [17]. While simple multiplexing approaches, which schedule control in a coarse-grained batch fashion, are possible [4], they prevent the fine-grained multitasking required for these examples, and, in the camera example, force a choice between either monitoring the intersection or tracking the suspicious vehicle during each coarse-grained time period.

Although many approaches to time-sharing, including proportional share scheduling, have been well-studied for CPUs and other peripheral devices, such as disks and NICs, steerable sensors, such as video cameras, present new challenges because they differ in both their physical attributes and application requirements.

Physical Attributes. Mechanically steerable sensors are both slow and stateful. Since steering latencies are on the order of seconds, the most contentious resource is control of the sensor, and not the aggregate bandwidth of sensed data or the total number of I/Os. Further, since each actuation changes the sensor's physical state, its current state determines the time to transition to a new state, which results in long, highly variable context-switch times.

Application Requirements. Applications control a sensor's actuators directly to drive data collection—often based on past observations. Since real-world events dictate steering behavior, applications may have timeliness constraints, either to sense data at

[1]This research is supported by NSF grants CNS-0720616, CNS-0720271, EEC-0313747, and CNS-0834243.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MMSys'11, February 23–25, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0517-4/11/02 ...$10.00.

specific locations, e.g., to track a moving object, or to coordinate steering among multiple cameras, e.g., to sense a fixed point from multiple angles.

In general, fine-grained multiplexing benefits any application that values continuous access to data and is willing to tolerate a lower resolution than possible with a dedicated sensor. While the deployment cost of steerable sensors limits their number, it also magnifies the potential benefits of fine-grained sharing. To realize this potential, we design MultiSense, a system for fine-grained multiplexing—at the level of individual actuations—of steerable sensor networks. MultiSense employs a proportional-share scheduler to multiplex multiple virtual sensors on a single physical sensor. While MultiSense is designed to work with a broad range of steerable sensors, we demonstrate its efficacy using pan-tilt-zoom camera sensors that are multiplexed across different applications.

While we could implement sensor multiplexing in numerous ways, MultiSense uses a virtualization-based approach to expose a virtual sensor (*vsensor*) to each application; a vsensor looks no different from the underlying physical sensor in terms of its interface and can be manipulated by an application independently of other vsensors (and applications) that are manipulating the same physical device. Our goal is to extend the benefits of virtual machine performance isolation to include steerable sensor devices. Our hypothesis is that steerable sensors, such as PTZ cameras, are capable of simultaneously tracking multiple real-world events with different sensing modalities, such as a person walking and a building's entry point, and hence, can be shared across concurrent applications. In designing MultiSense, this paper makes the following contributions.

Multiplexing Steerable Sensors. MultiSense employs a finite state machine to track each vsensor's state as it actuates, and uses a request emulation mechanism to buffer actuations until a sense request arrives—similar to a disk that buffers write requests until a read request arrives. We show how MultiSense uses these mechanisms to reduce the significant state restoration overheads incurred from context-switching between vsensors.

Proportional-share Adaptation and Extensions. We introduce Actuator Fair Queuing (AFQ), a proportional share scheduler that can allocate shares of a steerable sensor's time to vsensors, and evaluate a range of extensions and their effect on performance. Our experiments quantify the level of AFQ's isolation and the benefit of each extension.

Implementation and Experimentation. We implement MultiSense in Xen and use it to study multiplexing PTZ video cameras. We present a case study for PTZ video cameras using multiple modalities, including continuous scanning, object tracking, single fixed-point sensing, and multi-sensor fixed-point sensing. Our case studies show that MultiSense is able to satisfy concurrent applications using these sensors. As one example, we enable a tracking application to photograph an object moving at nearly 3 mph every 23 ft along its trajectory at a distance of 300 ft, while supporting a security application that photographs a fixed point every 3 seconds.

In Section 2, we motivate our use of vsensors and present background on multiplexing sensors. Section 3 then discusses MultiSense's basic design, while Section 4 outlines our adaptation of proportional-share and its extensions. Section 5 and Section 6 present MultiSense's implementation and evaluation using pan-tilt-zoom video cameras. Finally, Section 7 puts MultiSense in context with related work, and Section 8 concludes.

2. BACKGROUND

The primary problem addressed in this paper is how to multiplex ("time-share") a steerable sensor, such as a PTZ camera, at a fine time scale across multiple concurrent users with diverse requirements. We chose a virtualization approach for MultiSense to take advantage of the performance isolation capabilities present in modern virtualization platforms. We assume that each sensor node executes a hypervisor (also known as a virtual machine monitor) that hosts multiple virtual machines, one for each user. Each virtual machine exposes a virtual sensor device that appears to be an identical, but slower, version of the physical sensor to the user. A user application can manipulate the virtual sensor independently of other concurrent users; the virtualization layer ensures transparency by hiding the actions of one user from another, thereby providing the appearance of a dedicated sensor to each user. Multiple virtual sensors, one from each virtual machine, are mapped on to the underlying physical sensor and it is the task of the hypervisor to multiplex the virtual sensors onto the physical sensor, akin to time-sharing. Since concurrent requests from multiple users must be serviced by the physical sensor, and since mechanical actuation on steerable sensors is slow, each virtual sensor in MultiSense will appear to be a slower version of the physical sensor. Although MultiSense is capable of supporting a broad range of steerable sensors, in this paper, we focus on Pan-Tilt-Zoom (PTZ) cameras as a representative example of steerable sensors.

2.1 System Model

We assume each steerable sensor exposes one or more programmable actuators that applications control to steer it, and attaches to a node with local processing, storage and communication capabilities that is capable of running modern hypervisors. MultiSense multiplexes requests to steer the sensor across multiple applications, each executing in their own VM on each node. We assume that each application issues a stream of *actuation requests* to steer the sensor, followed by one or more *sense requests* to collect data. Thus, an application's request pattern takes the form:

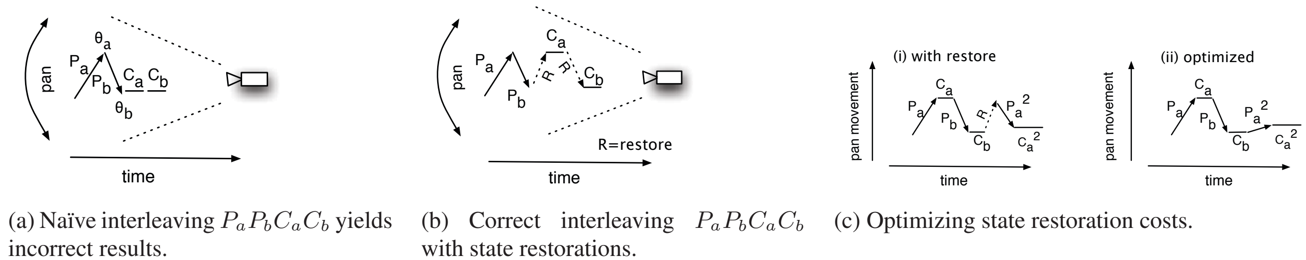
> $[A_1 A_2 \ldots A_n S_1 S_2 \ldots S_m]^+$, $n \geq 0$, $m > 0$, where A_i and S_i denote an individual actuation and sensing request, respectively.

The request pattern matches low-level sensing device interfaces, where each actuation request A_i alters the setting of only a single actuator. Each actuation A_i takes time t_i to steer the sensor to the specified setting, where t_i is dependent on the actuator's speed and its current setting.

We assume a constant actuator speed, although there may be some mechanical jitter. Sense requests S_i either *capture* data by collecting it using the current setting of the actuators, or *scan* data by collecting it while changing the setting of the actuators. For instance, a monitoring application for a PTZ camera might issue a repeating pattern of *pan* and *tilt* requests, followed by one or more *image capture* requests to retrieve images. We assume that actuation and sense requests from different applications are independent of one another, although a scheduler may take advantage of partial overlaps in requests. To enable fine-grained multiplexing, MultiSense interleaves requests from concurrent applications on the underlying physical sensor.

2.2 Design Challenges

A simple approach for multiplexing multiple users onto a physical sensor is to employ time-sharing and allocate a fixed time slice to each concurrent user in round-robin fashion. However, steerable sensors have actuators that are stateful (e.g, the pan and tilt actuators in a PTZ camera determine where the camera is pointing). Since each user can modify the state of these actuators via actuation commands, naive time sharing can be problematic for such stateful sensor devices.



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

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

(c) Optimizing state restoration costs.

Figure 1: Examples showing why request interleaving is challenging for steerable sensors.

We highlight the challenges in multiplexing stateful steerable sensors using a simple example.

Example 1: Consider two users—Alice and Bob—sharing control of a single PTZ camera. Assume that Alice first issues a pan, followed by a capture, denoted by P_aC_a while Bob issues a similar sequence P_bC_b, where the subscripts a and b denote Alice and Bob, respectively. Consider naïve time-sharing that interleaves these requests in the following order on the camera: $P_aP_bC_aC_b$. 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)).

As a result of the ordering, executing Alice's capture request C_a next results in an inconsistent picture, since *the camera's lens is at pan position θ_b when Alice expects the camera's lens to be at pan position θ_a*. Since the camera is stateful, Bob's actuation leaves the camera in a different state than Alice left it. As a result, naïve time-slicing using time quanta is inappropriate, since Alice and Bob would have no guarantee of the camera's state at the beginning of any time-slice.

Example 2: A straightforward solution is to *restore* Alice's state before context-switching back to her, similar to a CPU scheduler that restores the state of a thread's program counter and registers prior to scheduling it for execution. However, unlike CPUs and other peripheral devices, state restoration for mechanically steerable sensors is *slow*, and can be more expensive than the execution time of actuation requests.

For instance, the PTZ camera we use for our experiments takes nearly 9 seconds to pan from 0° to 340°, nearly 4 seconds to tilt from 0° to 115°, and over 2 seconds to zoom from 1x to 25x. Naïve state restoration can also exacerbate a sensor's slowness by executing wasteful actuations. In our example, restoring Alice's state to position θ_a is wasteful, since it requires re-executing the P_a pan request (Figure 1(b)). Better interleavings, such as $P_aC_aP_bC_b$, still pose a problem for a naïve strategy, since it is often more efficient to steer the sensor directly from θ_b to the position of Alice's next request P_a^2, rather than directly restoring her previous state (Figure 1(c)).

These simple examples motivate two basic elements of our approach. First, we maintain the correct vsensor state for each user to ensure their sensing requests are consistent. Second, we automatically group together requests of the form $A_i^*S_i$ to prevent wasteful actuations, since interleaving actuation requests from other vsensors within a group results in unnecessary state restoration. Despite these elements, context-switches between groups inevitably require some state restoration, making them inherently slow. Since MultiSense does not know each user's request pattern in advance, these context-switch times are also unpredictable.

Users will notice unpredictable context-switch times if they have strict timeliness requirements, and will perceive them as changes in vsensor actuation speed. For example, rather than maintaining a stable vsensor speed of v degrees/second, an application may observe a speed of $\frac{v}{2}$ degrees per second for one sensing request, and then a speed of $2v$ for the subsequent one. One option for reducing this variability is to require all applications to reveal their desired request pattern and timeliness requirements at allocation time, and then decide whether to insert the request pattern into a fixed, repeating schedule of actuator movements, similar to Rialto's approach to hard real-time CPU scheduling [13]. This type of scheduling is difficult even on a dedicated sensor since, similar to a disk head, the mechanical steering mechanism has inherent jitter.

Real-time scheduling similar to Rialto also requires strict admission control policies that limit the number of simultaneous users a system supports, and is problematic because sensing applications generally do not know their request patterns or requirements in advance, since real-world events may occur anywhere at anytime. Ultimately, some uncertainty is inherent if we allow each application the freedom to determine what actuation requests to issue and when to issue them. As a result, in our design of MultiSense, we explore how well proportional-share scheduling and its extensions isolate vsensor performance and meet the practical timeliness requirements of representative applications. Share-based scheduling is appropriate for allocating a resource whose supply varies over time. Since the time the physical sensor spends context-switching is dependent on the request patterns of its applications, the time available to control the sensor has the effect of a resource with varying supply.

3. MULTISENSE DESIGN

MultiSense extends traditional hypervisors by adding support to multiplex steerable sensors using a virtual sensor abstraction. A vsensor behaves like a slower version of the physical sensor that has identical functionality: an application designed to interface with the physical sensor should also interface with the corresponding vsensor. MultiSense resides in the hypervisor or a privileged control domain—e.g., Domain-0 in Xen—and interleaves requests from each vsensor on the underlying physical sensor, as shown in Figure 2. We separate MultiSense's functions into three categories described below. The goal of this decomposition is to reduce context-switch overheads while preserving a level of performance isolation.

1. **State Restoration**. MultiSense tracks the state of the physical sensor and each vsensor using finite state machines (FSM), and restores state whenever it detects a state mismatch at context-switch time.

2. **Request Groups**. MultiSense prevents wasteful context-switches by automatically grouping together requests from each vsen-

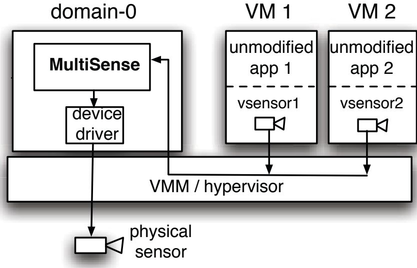


Figure 2: MultiSense Architecture Overview

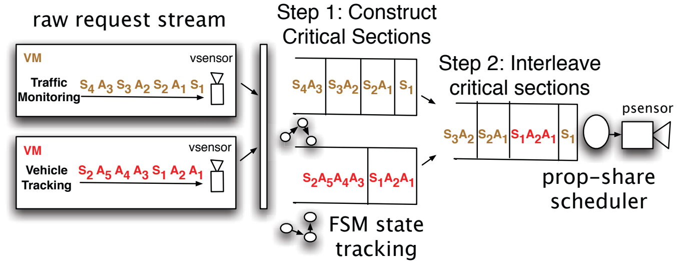


Figure 3: Constructing and interleaving request groups

sor of the form $A_i^* S_i$ and atomically issuing them to the sensor.

3. **Scheduling**. MultiSense employs a proportional-share scheduler and extensions at the granularity of request groups to determine an ordering that balances fair access to the sensor with its efficient use.

We describe MultiSense's FSMs, and their use in restoring state and inferring atomic request groups in this section, and discuss scheduling in Section 4. We use the term actuator broadly to include both mechanical actuators, as well as 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.

3.1 Sensor State Machines

Finite state machines track the state of each physical and virtual sensor, where a state is an n-tuple that represents a setting for each of n actuators. Each state transition has a cost that denotes the time the sensor takes to complete the transition. MultiSense employs a virtual state machine (VSM) to track the current state of each virtual sensor and a physical state machine (PSM) to track the state of the physical sensor. The state of a virtual sensor (and hence the VSM) changes only when the corresponding user actuates its vsensor. 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 MultiSense to track the state expected by each user, as well as the current state of the underlying physical sensor.

3.2 Intelligent State Restoration

Whenever MultiSense context-switches from one vsensor to another, it compares the state of the currently executing vsensor state machine (VSM) and the physical sensor's state machine (PSM). As with a CPU, if there is a state mismatch, MultiSense performs state restoration by automatically issuing requests for each out-of-sync state parameter to synchronize the vsensor's state with the physical sensor's state. As an example, assume that Alice's VSM is in state $pan = \theta_a$ $tilt = \phi_a$ $zoom = Z_a$, and the PSM is in state $pan = \theta_b$ $tilt = \phi_a$ $zoom = Z_b$. The two state machines are out-of-sync along the pan and zoom dimensions but in-sync along the tilt dimension. MultiSense synchronizes Alice's VSM state with the PSM by issuing a pan request to move the camera from θ_b to θ_a and a zoom request to move from Z_b to Z_a. No synchronization action is necessary along the tilt dimension.

We refer to this simple state restoration strategy as the eager strategy, since it eagerly synchronizes states with a past state on every context-switch. For steerable sensors, the eager strategy imposes a higher overhead than necessary, since it ignores actuation requests queued by each vsensor. Recall the example from Section 2.2, where Alice issues $P_a C_a$ followed by $P_a^2 C_a^2$, and the P_a request causes the camera to move to pan position θ_a. Now suppose that Bob's request $P_b C_b$ executes next, and the camera pans to position θ_b. Before executing Alice's next request, the eager 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 Alice's queued pan request P_a^2 intends to pan to position θ_a^2, making it more efficient to move the camera directly from θ_b to θ_a^2. To see why, suppose $\theta_b = 50^\circ$, $\theta_a = 30^\circ$ and $\theta_a^2 = 75^\circ$. Eager restoration pans from $50^\circ \rightarrow 30^\circ \rightarrow 75^\circ = 65^\circ$, while a direct pan from 50° to 75° requires only a 25° movement. For the PTZ camera we use, an additional 40° pan movement wastes more than 1 second.

MultiSense avoids this overhead using a lookahead strategy that does not restore state parameters that queued vsensor acutations will subsequently modify. For example, let VSM_{prev} denote the VSM state prior to a context-switch, and let VSM_{next} denote the VSM state that would result from executing requests queued after the last context-switch. $VSM_{prev} \cap VSM_{next}$ now denotes the set of state parameters not modified by these requests. The lookahead strategy only restores the states in $VSM_{prev} \cap VSM_{next}$. In the Alice and Bob example, $VSM_{prev} \cap VSM_{next}$ includes the parameters zoom and tilt, but not pan, since Alice's queued request will modify the pan parameter.

3.3 Grouping Requests via Request Emulation

To eliminate wasteful state restoration overheads, MultiSense automatically groups requests from each vsensor that the physical sensor should execute atomically. Each group includes a sequence of zero or more actuation requests, followed by a sense request from a single vsensor. Request groups prevent interference from the actuation requests of competing vsensors. However, since sensing and actuation requests are often blocking calls executed synchronously on the underlying physical sensor, vsensors only see a single request at a time, which does not permit grouping. To group requests, MultiSense enables asynchronous execution of blocking requests by emulating the execution of requests on the vsensor and deferring their actual execution on the physical sensor.

Request emulation allows the vsensor to behave as if the request actually executed on the sensor, allowing the blocking call to com-

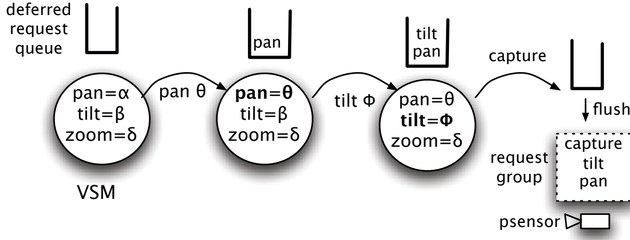


Figure 4: Request emulation and request groups

plete and the vsensor to continue execution. The vsensor's VSM tracks the state changes that result from any emulated requests, and defers their execution until the vsensor context-switches in. To ensure correctness, we only emulate actuation requests, since they do not return data that alters an application's control flow. Since sense requests return real-world data, MultiSense cannot emulate them, but must execute them using the physical sensor in the appropriate state to return a correct result. When a sense request arrives, MultiSense flushes the queue of deferred actuation requests to its scheduler, which then schedules the request group as a single atomic unit. The sense request blocks until the result returns.

As an example, consider how Alice's virtual camera maps onto a physical camera. Assume that Alice issues an actuation request P_a to pan to position θ_a. Request emulation triggers a VSM state transition to a new pan position θ_a, as shown in Figure 4. The figure also shows that MultiSense queues the request for deferred execution. Once the blocking pan completes, Alice's application continues execution and issues an actuation request to tilt to position ϕ_a, causing request emulation to continue by triggering another state transition in the VSM. Finally, Alice issues a capture request C_a, which MultiSense groups with the two pending actuation requests in the vsensor's queue and flushes it to the scheduler for execution on the physical sensor. Alice blocks until the group executes and returns the appropriate image.

One consequence of request emulation is that applications do not immediately perceive errors from actuations. We report any errors as a result of an actuation when its corresponding sense request executes, similar to any write-back cache that will defer reporting hardware errors until after a write executes. Note that this change affects neither application correctness nor device safety. MultiSense delays reporting actuation errors to the time of an application's next sense request. Since sensor data dictates an application's control flow, the application will observe errors prior to making control flow decisions. Likewise, since MultiSense controls the issuing of requests to the physical sensor, it is capable of preventing cascading errors from unknowing applications that may damage the sensor.

4. PROPORTIONAL-SHARE FOR STEERABLE SENSORS

MultiSense flushes request groups to a proportional-share scheduler that decides when to execute them. We design Actuator Fair Queuing (AFQ) by modifying the standard Start-time Fair Queuing (SFQ) algorithm, originally designed for NICs [3] and CPUs, to schedule steerable sensors [9].

As background, we provide a brief summary of SFQ. 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, $\frac{W_i(t_1,t_2)}{w_i} - \frac{W_j(t_1,t_2)}{w_j} = 0$, 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.

We define the SFQ algorithm for scheduling critical sections in MultiSense 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 kth 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 kth 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.

4.1 Actuator-Fair Queuing

AFQ differs from SFQ by setting the length of a request equal to the time it would take to execute on the dedicated sensor, and introducing batch-based reordering, discussed in the following subsection, that address scheduling issues specific to steerable sensors. As with other proportional-share schedulers, AFQ associates a weight w_i with each vsensor and allocates $w_i/\sum_k w_k$ of the physical sensor's time to vsensor i. Lowering a vsensor's weight assignment affects its performance by slowing down its actuation speed. In

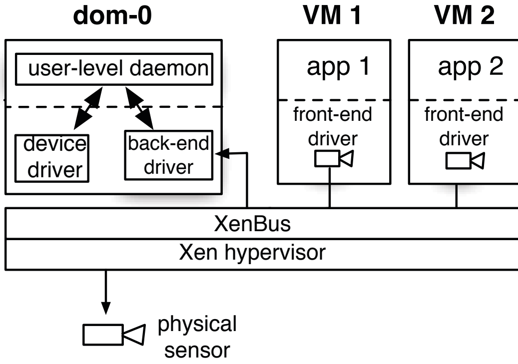


Figure 5: MultiSense uses Xen's split-driver framework for communication, and a user-level daemon in Domain-0 to maintain vsensor VSMs and execute scheduling policies. Each request passes from application → front-end driver → back-end driver → daemon → device.

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 ($< 0.3\%$).

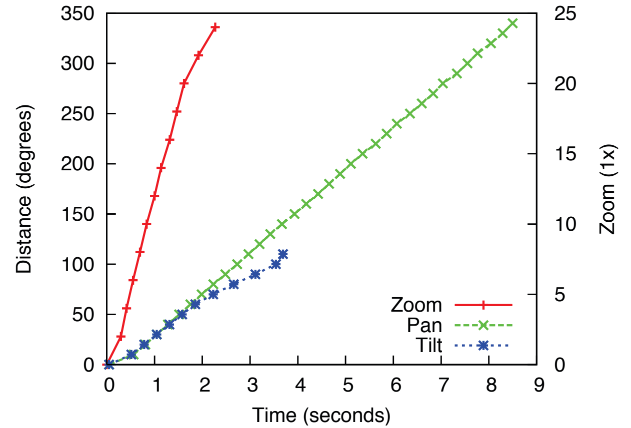


Figure 6: Time benchmarks for the pan, tilt, and zoom actuators of Sony SNC-SRZ30N Network Camera that show the distance moved by the actuator is roughly linear to the time and energy required to complete the actuation.

work-conserving mode, actuation speeds may also become faster if any vsensor is not using its share by being passive.

The ideal is only possible if the physical sensor is able to divide each actuation into infinitesimally small time units. Since actuations are of variable length and MultiSense schedules at the granularity of request groups, enforcing the ideal is not possible. We chose SFQ as our foundation because it bounds the resulting unfairness due to this discrete granularity 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 request group from vsensor i. Intuitively, this bound is a function of the largest possible request group, which for our PTZ camera is an actuation, from $pan = -170°$ $tilt = -90°$ $zoom = 1x$ to $pan = 170°$ $tilt = 25°$ $zoom = 25x$. Since this worst-case scenario takes nearly 16 seconds for our camera, one goal of our evaluation is to explore performance in the common, rather than the worst, case for representative applications.

4.2 Batching

SFQ ignores the actuation costs from context-switching between request groups, causing significant overheads. As an example, consider three users Alice, Bob and Carol sharing a PTZ camera. Assume that the camera is currently at position 25°, and Alice, Bob and Carol have start tags of 10, 11 and 12, respectively, when Alice issues a pan request for position 30° and Bob and Carol issue pan requests for positions 75° and 40°. SFQ services these requests in order of the start tags—Alice, then Bob, and finally Carol—and triggers pans from $25° \rightarrow 30° \rightarrow 75° \rightarrow 40° = 85°$. However, since Alice and Carol's requests are close to each other, servicing the requests in the order Alice, then Carol, and finally Bob lowers the overhead to $25° \rightarrow 30° \rightarrow 45° \rightarrow 75° = 50°$. For our PTZ camera, this results in nearly a 1 second reduction in overhead. We address this issue in AFQ by selecting the k pending request groups with the smallest start tags, one from each vsensor, instead of selecting only the request group with the minimum start tag.

Given a batch of k request groups, we reorder them to minimize the physical sensor's total actuation time. In our example, this strategy selects the more efficient Alice → Carol → Bob ordering. For a single actuator, the batching strategy is similar to proportional-share disk schedulers that use an elevator algorithm to reorder batched requests [6]. Since our sensors have multiple actuators, minimizing actuation time is an instance of the NP-hard Traveling Salesman Problem. We use a greedy heuristic that always executes the next closest request in the batch. For small values of k, a brute force search that tries all permutations is also feasible. Introducing the parameter k defines a new tradeoff: the higher the value of k the more efficient, but less fair, the schedule. In Section 6.2, 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 for our examples.

5. MULTISENSE IMPLEMENTATION

Although MultiSense is designed to support a range of steerable sensor, here we focus on supporting PTZ camera sensors. Our MultiSense prototype is implemented in the Xen virtual machine and integrates with Xen's virtual device framework. Our PTZ camera sensors are implemented as character devices that transfer streams of data serially to applications. In Linux, applications typically interface with sensors through character device files using the open, close, read, write, and ioctl system calls. To support devices, Xen uses a split-driver approach that divides conventional driver functionality into two halves: a front-end driver that runs in each VM

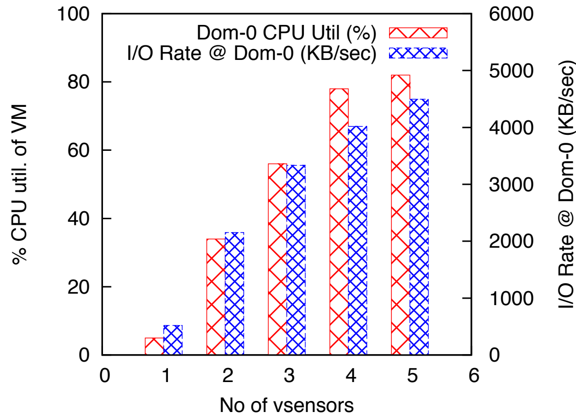


Figure 7: Utilization of Domain-0's 40% CPU share and I/O rate for streaming data. The number of vsensors vary from 1 to 5, where the first vsensor is a PTZ camera and the remaining vsensors are streaming data.

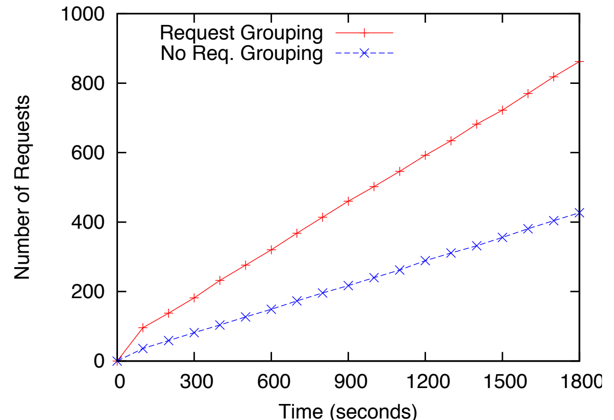


Figure 8: Request grouping improves performance 2x.

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 [2]. Figure 5 depicts MultiSense's Xen implementation using a generic front-end character driver 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 interactions. In our current implementation the back-end driver passes requests to a user-level daemon running in Domain-0 using the back-end's read and write system calls. This daemon includes the logic to maintain and restore state, group requests, and schedule groups using a sensor's conventional application-level interface. Implementing MultiSense at user-level has two advantages beyond simplifying debugging. First, manufacturers often release binary-only drivers for Linux that are only accessible from user-level, necessitating user-level integration. Second, the user-level daemon decouples our implementation from a specific virtualization platform, allowing us to switch to alternatives, e.g., Linux VServers, if necessary. Since the dominant performance cost for steerable sensors is actuation time and not data transfer, as we show in Section 5.2, the overhead of moving data between kernel-space and user-space in our case is insignificant. For sensors where data transfer is the dominant cost, we could integrate the functions of this daemon into the back-end driver.

MultiSense's front-end/back-end drivers are reusable with different types of sensors since they only serve as a communication channel for requests. The user-level daemon maintains a vector and queue for each vsensor that stores the current setting of its actuators and its backlog of deferred actuation requests, respectively. The daemon also manages VSMs and state restoration as well as our extensions, such as request batching. When an actuation request arrives, the daemon associates a start tag with it, places it at the end of its vsensor's queue, sends back a response, and changes the actuator's vector entry. When a sense request arrives, the daemon batches it with any deferred requests in order of their minimum start tag, assigns the start tag of batch as the start tag of the sense request, and flushes the batch to the common queue used by the AFQ scheduler. As soon as k request batches arrive or time t passes from the last scheduling opportunity, the scheduler reorders the request batches in the common queue using our greedy heuristic and issues them to physical sensor, as described in Section 3.3.

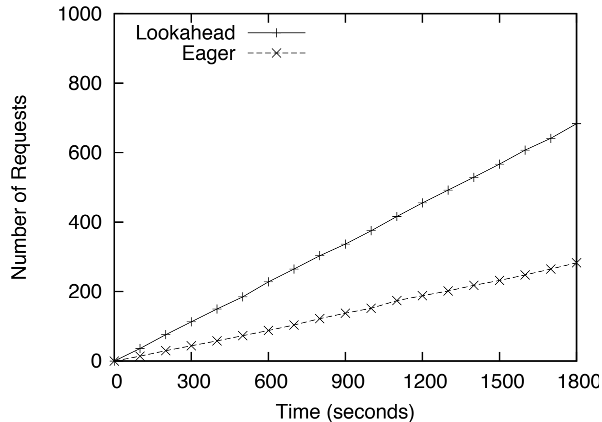
5.1 PTZ Camera

We evaluate MultiSense for PTZ cameras using the Sony SNC-RZ50N PTZ Network Camera. Beyond the three actuators we focus on, the camera has many non-obvious actuators, including resolution setting, shutter speed, backlight compensation, night vision, and electronic stabilization, that influence an image's fidelity. The 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 control of pan and tilt increments as small as $1/3^\circ$. We benchmarked the speed of each of the camera's actuators independently (Figure 6). The camera is capable of panning at 40°/sec, tilting at 30°/sec, and zooming at 12x/sec, although shorter movements are slower due to the acceleration/deceleration of the motor, which accounts for a major fraction of overall actuation time in case of shorter movements.

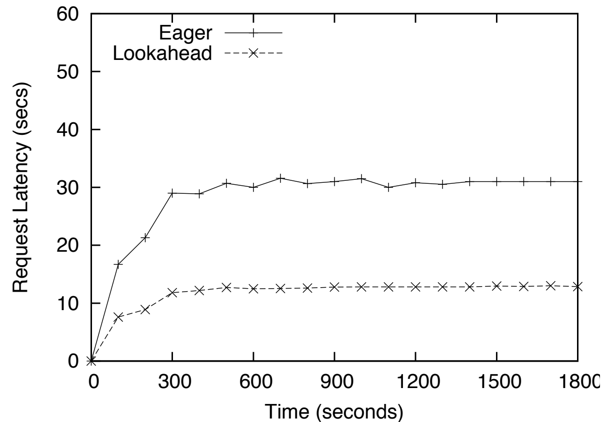
5.2 Benchmarks

Before evaluating MultiSense, we benchmark its implementation overhead. Our experiments run on our testbed nodes which each use a 2.00 Ghz Intel Celeron CPU, 1GB RAM, and an 80GB SCSI disk running version 3.2 of the Xen hypervisor with Ubuntu Linux using kernel version 2.6.18.8-xen in both Domain-0 and each guest VM. Each guest uses a file-backed virtual block device to store its root file system image. Each node consists of a Sony RZ50N PTZ camera sensor. Using the camera, Table 1 reports the overhead MultiSense imposes on a single vsensor actuation request and its response as it flows from the application to the device and then back to the application. Xen adds two additional layers in the flow—the front-end and back-end device driver—while MultiSense adds one layer by using a user-level daemon in Domain-0. As Table 1 shows, the overhead of these additional layers is minimal compared (order of μseconds) to the actuation times (order of seconds). We also benchmark the maximum aggregate I/O that MultiSense is able to support, and its CPU overhead.

For these experiments, we use Xen's proportional-share credit scheduler to allocate Domain-0 40% of the CPU and each VM 10% of the CPU. Domain-0 requires some CPU to process vsensor I/O requests and execute MultiSense's scheduler. We vary the number



(a) Number of Requests



(b) Average Request Latency

Figure 9: The lookahead state restoration strategy outperforms the eager approach in our sample workloads. The number of requests completed (a) is 2x more and the average latency to satisfy each request (b) is 2x less using the lookahead approach.

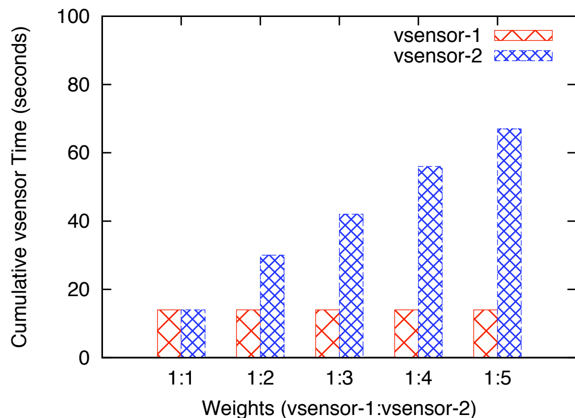


Figure 10: AFQ enforces performance isolation over large numbers of requests. The ratio of the total vsensor time for the two continuous scan workloads is in proportion to the assigned weights.

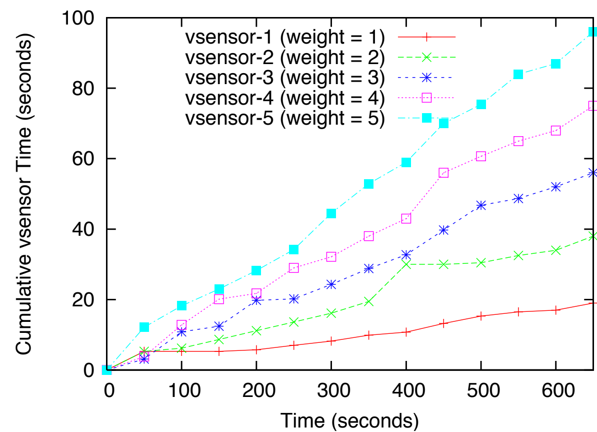


Figure 11: While AFQ enforces performance isolation over many requests, it may diverge slightly, due to high state restoration costs, over short intervals of time.

of VMs from 1 to 5, where the first VM controls the PTZ camera and the other VMs are streaming data continuously to stress I/O request processing. Figure 7 shows the maximum achievable I/O rate that MultiSense is able to deliver to each vsensor when cameras produce data as fast as possible. The result demonstrates that MultiSense is able to handle an I/O rate of 4.6 MBps of streaming data in this extreme case without overloading the CPU allocated to Domain-0, as shown by the Domain-0 CPU utilization in the figure. For reference, Netflix's watch instantly feature has a bit rate 5 MBps using the VC-1 codec [1].

6. EXPERIMENTAL EVALUATION

We first evaluate the impact of MultiSense's strategies for state restoration, request groups, and scheduling individually using synthetic workloads. The experiments demonstrate the extent to which these optimizations improve request throughput and latency. MultiSense's primary metric for success is whether or not it accommodates real concurrent applications. We present a case study for the camera that demonstrates the application-level performance and timeliness requirements MultiSense can achieve using our example sensors. We use both deterministic and random synthetic workloads to benchmark MultiSense's functions.

For the camera, the deterministic workload performs continuous scans using a single actuator in a single direction interspersed with sense requests, while the random workload repeatedly issues requests for a random setting of the actuators followed by a sense request. Each scan issues a sense request every 10° starting at one extreme and moving to the other. We intend these synthetic workloads to be conservative, since they force MultiSense to steer to extreme points in a sensor's state space, while also satisfying randomly generated requests. We describe the workloads for the applications in our case study in Section 6.3.

6.1 State Restoration and Request Groups

We demonstrate the impact of state restoration and request grouping, independently of our scheduling policy, on throughput—the

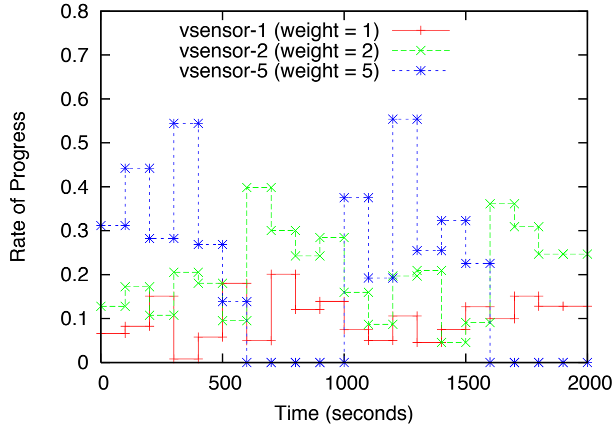


Figure 12: AFQ maintains the work-conserving property when applied to actuators.

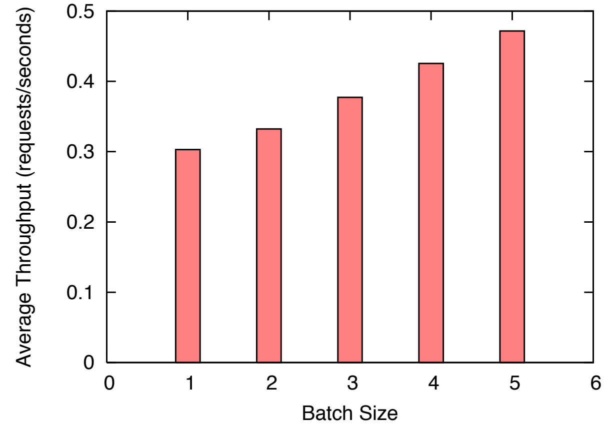


Figure 13: AFQ shows better global performance in terms of average throughput in requests/seconds as batch size increases. For this experiment, each increment in the batch size results in roughly a 10% improvement.

number of requests a sensor is able to satisfy per time interval. We first compare the eager approach to state restoration, described in Section 3.2, with our lookahead approach. Figure 9 shows results from an experiment using five vsensors, with batch size of 3, executing the random workloads described above. Figure 9(a) shows the progress of completed requests on the physical camera for both approaches, while Figure 9(b) plots the average latency to satisfy each request.

The lookahead approach is significantly more efficient: it is able to satisfy nearly 2x as many requests during the same 30 minute time period with 2x less latency on average per request. We also demonstrate the impact of request grouping by running the same experiments above with and without grouping. In this case, we use a group size of 5. Figure 8 shows the results. Using request groups, the camera is able to satisfy 2x more requests than without request groups. Our result highlights the importance of optimizing state restoration and grouping requests for efficiency, since a poor strategy may cancel any benefits from better scheduling. The consequences for an application are significant. For our camera case study (Figure 16), a 2x increase in request latency would mean capturing an image every 6 seconds, versus capturing it every 3 seconds.

6.2 AFQ Scheduling

The goal of AFQ is to enforce performance isolation between vsensors—each vsensor should receive performance in proportion to its weight (Figure 11). While AFQ bounds the maximum unfairness within any time interval, our extensions relax this bound to increase efficiency. We first demonstrate AFQ's strengths and limitations when scheduling steerable sensors, and then present results that show the performance gains, as well as the impact on fairness, for each of our extensions.

AFQ advances virtual time in relation to the time each actuation consumes on the dedicated sensor, which we denote as vsensor time. The more vsensor time each actuation consumes the slower the actuator. Figure 10 shows the total vsensor time of two vsensors with different weight assignments using AFQ, where each vsensor executes the continuous scan workload. The figure demonstrates that a straightforward adaptation of SFQ for actuators isolates vsensor performance: the cumulative vsensor time it allocates is in proportion to the assigned weights. As shown in Figure 12, AFQ proportionally distributes shares of the passive vsensor (vsensor-5's share during time interval 500-1000 and 1500-2000 seconds) among active vsensors (vsensor-1 and vsensor-2). However, while SFQ enforces performance isolation over large numbers of requests, high context-switch costs cause it to perform unfairly over short intervals.

To demonstrate the point, Figure 11 shows how the cumulative vsensor time progresses 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 MultiSense services larger numbers of requests, performance converges to the assigned weights by 550 seconds.

6.2.1 Request Batching

Figure 13 demonstrates the performance improvement from batching for the camera. The experiment uses random workloads from 5 vsensors to stress actuation, and shows that the average throughput increases as the batch size increases—each increment in batch size results in roughly a 10% improvement. However, the improvement comes at a cost: the scheduler diverges from strict fairness. Figure 14(a) shows the cumulative request latency for each of the five vsensors as a function of batch size, using the same five vsensors and workloads as Figure 13. The cumulative request latency is the sum of the latencies to satisfy all requests at each vsensor, which is equivalent to each vsensor's makespan.

Figure 14(b) plots the cumulative vsensor time over the course of the experiment for a batch size of 4. Comparing the result with Figure 11 in the previous section emphasizes the decrease in performance isolation. As expected, SFQ, which corresponds to a batch size of 1, exhibits strong performance isolation. As the batch size increases, though, performance isolation decreases, causing the height of the bars to approach each other. For these workloads,

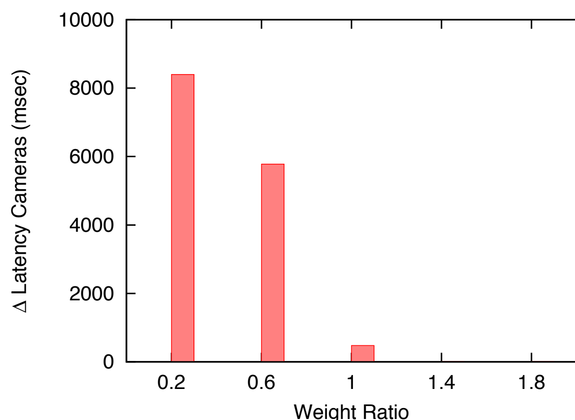


Figure 15: The difference in latency when coordinating multiple vsensors on different nodes to sense the same point.

a batch size of 3 exhibits an appropriate balance by increasing performance by 20% while achieving similar fairness properties. In practice, we have found that a batch size of roughly half the number of active vsensors strikes the appropriate balance.

6.3 Case Studies for PTZ Camera Sensors

Our case study explores MultiSense's use for the camera with four example applications with specific performance metrics. We use the lookahead state restoration approach, request groups, and AFQ.

- **Continuous Monitoring**. Continuously pan in increments of 65°—nearly one-fifth of the possible pan range—and capture an image. The performance metric is the time to cover the sensor's entire range.
- **Fixed-point Sensing**. Pan, tilt, and zoom the lens to a fixed point and repeatedly capture images at a regular interval. The performance metric is the sensing rate.
- **Object Tracking**. Periodically track a pre-defined path along both the pan and tilt axes and capture images every 10°. The performance metrics are both the latency between sensing requests, and the minimum overall latency necessary to keep up with the moving object.
- **Multi-sensor Fixed Point Sensing**. For two cameras, pan, tilt, and zoom the lens to a fixed point and repeatedly capture images at a regular interval. The sensor must also satisfy competing applications. The performance metric is the rate at which both sensors capture the fixed-point, which is equivalent to the minimum sensing rate of the two sensors.

With a dedicated camera, fixed-point sensing has near video quality. The sensing rate is 11 images/second with an average inter-image interval of 0.09 seconds. However, even on a dedicated sensor, actuation does have a significant effect on performance. Executing our random workload, reduces the rate to 0.3 images/second with an average inter-image interval of 3.35 seconds. Similarly, two fixed-point sensing applications—at a distance of 180°—are both able to capture 0.2 images/second with an average inter-image interval of 4.65 seconds.

We first execute both continuous monitoring (Figure 16(a)) and object tracking (Figure 16(b)) concurrently with the fixed-point sensing application for the camera. We maintain a weight of 1 for fixed-point sensing, while varying the weights assigned to continuous monitoring and object tracking. Figure 16 shows the results for the camera, where the left y-axis plots the application's performance metric, the right y-axis plots sensing rate for fixed-point sensing, and the dotted line depicts performance on a dedicated sensor. The results show that MultiSense is able to satisfy the conflicting demands of concurrent applications. Of course, the applications must be able to tolerate less performance than possible with the dedicated sensor, which in these examples ranges from 1.5x to 8x less performance for the different weight assignments. Since weight dictates performance, some applications may need a minimum weight to satisfy their requirements.

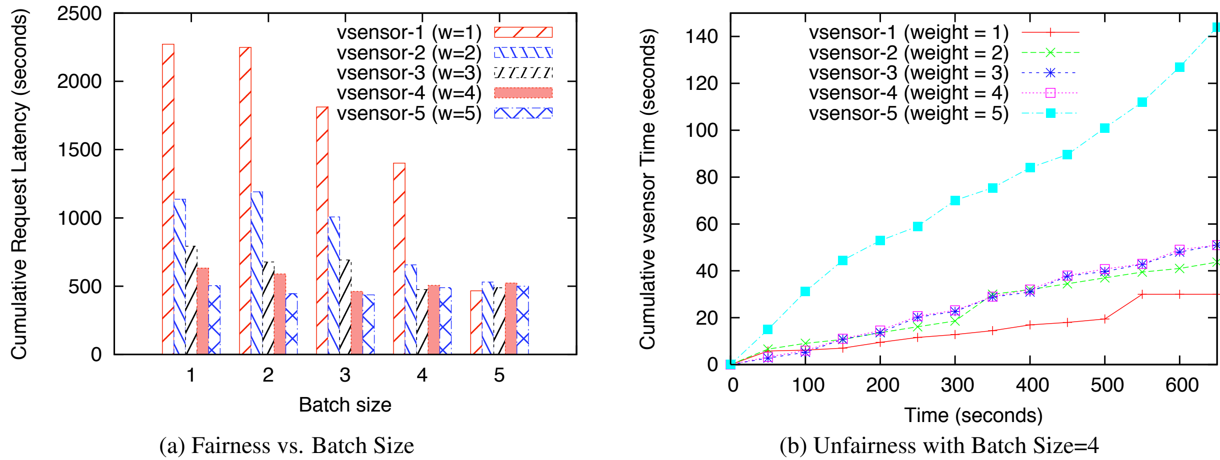
Consider continuous monitoring for the camera with a 1:30 weight ratio, the application is able to pan all 340° in 20 seconds. Thus, in the real-world, the monitoring application is able to capture 5 distinct points 113 feet apart, e.g. four doorways, at distance of 100 feet from the camera every 4 seconds. The example assumes the points are along a circle with radius 100 feet with camera's lens as its center. Simultaneously, fixed-point sensing maintains an average sensing rate of nearly 0.2 images/second, allowing it to continuously capture a single point, such as a nearby intersection. Likewise, for a 1:3 weight ratio, the object tracking application is able to scan a pre-defined path every 10° and capture images at least 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. 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. Our example assumes that the object's trajectory is along a circle of radius 300 feet with the camera's lens as its center. During tracking, the fixed-point sensing application maintains a sensing rate of 0.3 images/second.

We also ran an experiment for a networked multi-sensor scenario where the application coordinates multiple sensors to sense a fixed point, while competing with continuous monitoring on one sensor and fixed-point sensing on the other. The experiment demonstrates the extent to which MultiSense satisfies timeliness requirements. Figure 15 shows the results. The x-axis shows experiments with different weight ratios assigned to the competing applications on each sensor, while the y-axis plots the average difference in latency between two requests. The magnitude of this difference determines how close in time the two sensors are able to capture data for the same point. As the graph shows, higher weight assignments decrease the difference, and provide near (< 1 second) simultaneous sensing. Even with a low relative weight assignment the sensors sense the same point within 2 seconds of each other, which is suitable for a range of scenarios, such as estimating the pedestrian entry/exit points for cameras. We are exploring other challenges that arise in distributed multi-sensor scheduling as part of future work, including applications with tighter time constraints.

7. RELATED WORK

MultiSense 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.

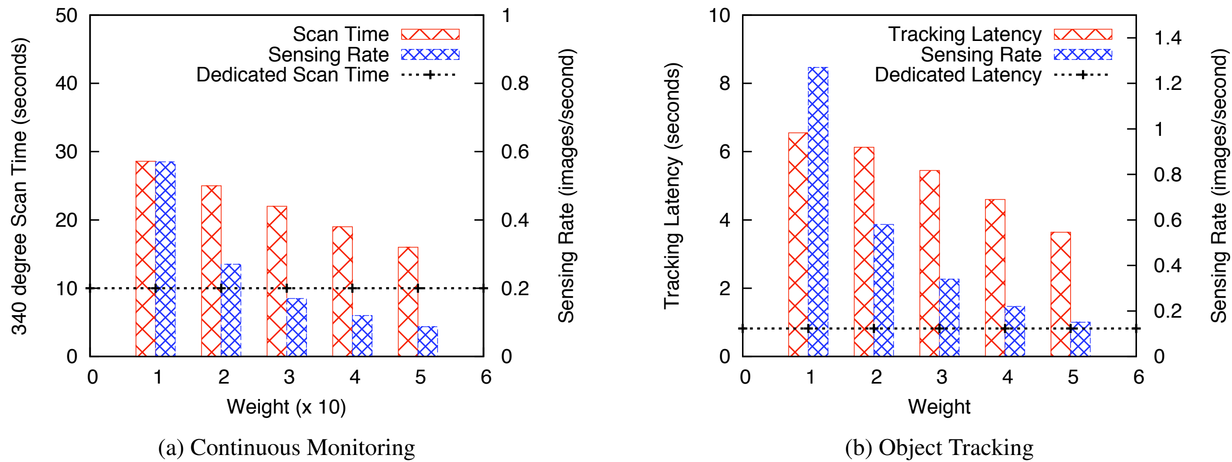
Mote-class sensor networks primarily use virtualization as a mechanism for safe execution and reprogramming, as demonstrated by Maté [11], since motes are generally not powerful enough to execute multiple applications concurrently. While some recent mote-class OSes incorporate threads and time-sharing [5], the energy constraints of motes prevent them from using high-power sensors



(a) Fairness vs. Batch Size

(b) Unfairness with Batch Size=4

Figure 14: 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.



(a) Continuous Monitoring

(b) Object Tracking

Figure 16: For the camera, MultiSense is able to serve 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.

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 [10]; ICEM solves the problem for mote power management by exposing concurrency to drivers through power locks. In contrast, MultiSense 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.

MultiSense uses Xen's [2] basic abstractions for multiplexing I/O devices [19]. Other frameworks, including VMedia [22], use Xen for coordinating shared access to peripheral devices. As with other device virtualization frameworks, VMedia focuses on stationary devices, e.g., web-based cameras and microphones, but does not extend the paradigm to steerable devices. 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 [23]. We virtualize at the protocol layer—the character device file interface—so MultiSense can interpret each vsensor request and control their submission to the physical sensor. Our choice to implement sensor multiplexing and proportional-share scheduling in Xen is a result of our broader goal of lowering the barrier to experimenting with these systems from the ground up. Xen and other virtualization platforms offer the low-level fault, resource, and configuration isolation that we require. MultiSense's FSM that tracks the state of each vsensor is similar to shadow drivers [18], 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].

MultiSense applies the proportional-share paradigm, which has been well-studied in other contexts, to multiplex control of steerable sensors. SFQ was originally prototyped for multiplexing packet

streams and later extended to CPUs [9]. More recently, there has been work on proportional-share scheduling for energy—another non-traditional resource—using virtual batteries [7]. We extend the paradigm to the actuation resources of steerable sensors. Perhaps most related to MultiSense is past work on proportional-share scheduling for disks. Disk schedulers incorporate a similar batching technique [6] and often group together write requests and flush them to disk after a read request occurs. However, there are fundamental differences in the relative speed of actuators and their use, as well as workload characteristics, that present different trade-offs for steerable sensors. Rather than modeling the shared resource as I/O bandwidth or aggregate number of I/Os, which is often the case for disks [16], we use total time controlling the sensor, since this determines when and what applications are able to sense. We also introduce and evaluate new extensions for scheduling steerable sensors and evaluate their impact on applications. As with disk scheduling, other optimizations, such as Anticipatory Scheduling, may further improve performance [12].

8. CONCLUSION

MultiSense extends proportional-share scheduling to multiplex the resource of controlling a sensor's actuators. For steerable sensors, control of the actuators is the most important resource since it determines the type of data the sensor collects. This is the first work, to the best of our knowledge, to multiplex this important, but often overlooked, class of sensors. One reason multiplexing is critical for steerable sensor networks is their high deployment costs. In this paper, we demonstrate techniques for enabling multiplexing and proportional-share scheduling, and evaluate our techniques on synthetic workloads, as well as four real applications, that demonstrate their effectiveness for PTZ cameras.

9. REFERENCES

[1] Netflix Watch Instantly. http://www.netflix.com/WiMessage?msg=59. Retrieved September, 2010.

[2] P. 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. In *Proceedings of the Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

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

[4] 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. In *Proceedings of the Internet and Multimedia Systems and Applications Conference*, Grindelwald, Switzerland, August 2005.

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

[6] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling With Quality Of Service Guarantees. In *Proceedings of the International Conference on Multimedia Computing and Systems*, Florence, Italy, July 1999.

[7] Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. Abdelzaher. Virtual Battery: An Energy Reserve Abstraction For Embedded Sensor Networks. In *Proceedings of the Real-time Systems Symposium*, San Diego, CA, November 2008.

[8] A. Francoeur. Border Patrol Goes High Tech. photonics.com, August 2009.

[9] P. Goyal, H. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm For Integrated Services Packet Switching Networks. In *Proceedings of SIGCOMM*, Stanford, CA, August 1996.

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

[11] P. Levis and D. Culler. Maté: A Tiny Virtual Machine For Sensor Networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[12] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework To Overcome Deceptive Idleness In Synchronous I/O. In *Proceedings of the Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[13] M. Jones, D. Rosu, and M. Rosu. CPU Reservations And Time Constraints: Efficient, Predictable Scheduling Of Independent Activities. In *Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

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

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

[16] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework For Next Generation Operating Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[17] S. Magnuson. New Northern Border Camera System To Avoid Past Pitfalls. National Defense Magazine, September 2009.

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

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

[20] M. Zink, E. Lyons, D. Westbrook, J. Kurose, and D. Pepyne. Meteorological Command & Control: Closed-loop Architecture for Distributed Collaborative Adaptive Sensing of the Atmosphere. *International Journal for Sensor Networks*, 7(1).

[21] D. McLaughlin, D. Pepyne, V.Chandrasekar, B. Philips, J. Kurose, M. Zink. Short-Wavelength Technology and the Potential for Distributed Networks of Small Radar Systems. *Bulletin of the American Meteorological Society*, April 2009.

[22] H. Raj, B. Seshasayee and K. Schwan. VMedia: Enhanced Multimedia Services in Virtualized Systems. In *Multimedia Computing and Networks Conference*, San Jose, CA, January 2008.

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