MultiSense: Fine-grained Multiplexing for Steerable Sensor Networks

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

Abstract

Steerable sensors, such as pan-tilt-zoom cameras and weather radars, expose programmable actuators to applications, which steer them to dictate the type, quality, and quantity of data they collect. Applications with different goals steer these sensors in different directions. 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 designed MultiSense, which enables 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, request batching and merging, and anticipatory scheduling, satisfies the unique requirements of steerable sensors. We present experiments for pan-tilt-zoom cameras and weather radars that show MultiSense efficiently isolates the performance of virtual sensors, 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.

1 Introduction

Steerable sensor networks allow applications to steer mechanical actuators to control the type, quality, and quantity of data they collect. For example, researchers are prototyping the use of steerable weather radars to improve weather prediction and fill coverage gaps in the existing NEXRAD system [15]. The U.S. Border Patrol is also deploying networks of pan-tilt-zoom (PTZ) cameras to continuously monitor the northern border for smugglers [13], and as part of a "virtual fence" on the southern border [5]. 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 steerable radar we consider is nearly $300,000, not including infrastructure, operational, or labor costs, and the cost for the 20-mile prototype of the Border Patrol's "virtual fence" is over $20 million. 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.

Enabling fine-grained multiplexing is an important step in providing broader access to use and experiment with these exclusive 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. While simple multiplexing approaches, which schedule control in a coarse-grained batch fashion, are possible [2], such reservations prevent the fine-grained multitasking required for this example, and force a choice between either monitoring the intersection or tracking the suspicious vehicle during each coarse-grained time period.

Although multiplexing has been well-studied for CPUs and other peripheral devices, such as disks and NICs, steerable sensors present new challenges because they differ in their physical attributes, application requirements, and workload characteristics.

- **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 a 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 specific locations, e.g., to track a moving object, or to coordinate steering among multiple sensors, e.g., to sense a fixed point from multiple angles.
- **Workload Characteristics**. Since sensing applications observe real-world events, we cannot make

assumptions about the spatial or temporal locality of actuation requests—important events may occur anywhere at anytime. However, we can take advantage of locality if it exists. Since one or more applications may track similar events, there are opportunities to merge partial overlaps among requests.

In general, fine-grained multiplexing benefits any application that values continuous access to sensor 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 designed MultiSense, a system for fine-grained multiplexing—at the level of individual actuations—of steerable sensor networks. MultiSense combines a proportional-share scheduler with a range of extensions to multiplex virtual sensors on a single physical sensor.

While sensor multiplexing could be implemented in numerous ways, MultiSense's implementation integrates with a virtualization platform to expose a virtual sensor (vsensor) to each application that has the same interface as the physical sensor. The goal is to extend the boundary of VM performance isolation to include sensors, in addition to other compute resources. We discuss the motivation behind this implementation choice more in Section 2. Our hypothesis is that steerable sensors are capable of sensing multiple real-world events, such as a person walking, a thunderstorm, or a tornado, conforming to different sensing modalities. In designing MultiSense, this paper makes contributions in three areas.

Multiplexing Steerable Sensors. MultiSense employs a finite state machine to track each vsensor's state as it moves, 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 adapt Start-time Fair Queuing [6] to allocate shares of a steerable sensor's time to vsensors, and evaluate a range of extensions and their effect on performance, including request batching, request merging, and anticipatory scheduling. Our experiments quantify the level of SFQ's isolation and the benefit of each extension.

Implementation and Experimentation. We implement MultiSense in Xen and use it to study two different examples of steerable sensors: a PTZ camera and a steerable weather radar. We present a case study for both sensors 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 our example sensors. As one example, we enable a tracking application to photograph an object every 23 ft moving at nearly 3 mph 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 sensor multiplexing. Section 3 dicusses 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 cameras and radars. Finally, Section 7 puts MultiSense in context with related work, and Section 8 concludes.

2 Background

We first motivate the use of vsensors as MultiSense's primary abstraction. We then define a system model that guides our work and highlights the challenges of multiplexing steerable sensors.

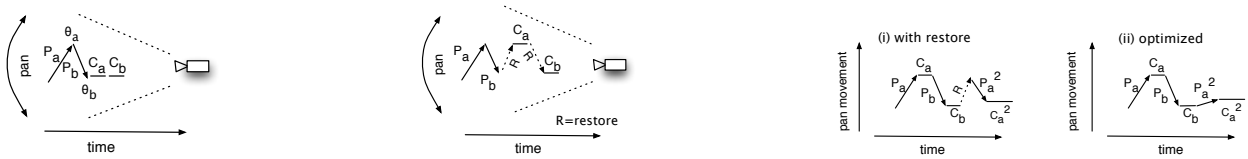
2.1 Approach

We chose a virtualization approach for MultiSense's implementation to take advantage of the performance and fault isolation capabilities present in modern virtualization platforms. Our goal is to lower the barrier for experimenting with expensive steerable sensor networks by using MultiSense in a 4-node testbed we have built, named ViSE [1], that includes PTZ cameras, radars, and weather stations. Virtual machines (VMs) serve to isolate both the testbed's control plane from its users, and its users from each other. Testbed users, or programmatic controllers acting on their behalf, request VMs bound to not only a sliver of the node's CPU, memory, storage, and bandwidth, but also to one or more attached sensors.

Importantly, the approach allows users to experiment with all aspects of building this type of non-traditional sensor system from the ground up. For instance, ViSE's nodes communicate over long-distance (10 km) 802.11b links that do not have enough bandwidth to transmit all of the data a camera or radar produces, requiring users to either shift some of their computation into the network or prioritize data transmissions (see [9]). ViSE simply allocates the low-level computing, communication, and sensing resources, and leaves users to experiment with how to best use them.

The testbed is part of GENI, so other goals include linking its nodes and sensors with other substrates using one of GENI's control frameworks. Since our goals are broader than just studying the sensor multiplexing problem, we chose not to implement MultiSense at a higher level of the software stack, such as a C library or

[1] See `http://geni.cs.umass.edu/vise`. ViSE stands for **Vi**rtualized **S**ensing **E**nvironment.



(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 for steerable sensors.

network-accessible API, although our low-level implementation does allow testbed users the freedom to layer higher level abstractions on top of it. Ultimately, the problem MultiSense addresses is independent of the specific layer of the implementation.

We also note that while each VM exposes a local interface to its vsensors, our testbed assumes the use of a distributed application or experiment controller that coordinates control of vsensors on different nodes. Thus, the application logic for controlling the sensor may reside within an individual VM—when there is no need for coordination—or within an external controller—when there is a need for coordination. Since the nodes use x86-class processors connected to the power grid, they do not have the energy or computing constraints that discourage placing application logic on the sensor node, itself.

2.2 System Model

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 VMMs. MultiSense multiplexes requests to steer the sensor across multiple applications, each executing in their own VM on each node. We model each application as 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: $[A_1A_2 \ldots A_nS_1S_2 \ldots S_m]^+$, $n \geq 0$, $m > 0$, where A_i and S_i denote an individual actuation and sensing request, respectively. Each actuation A_i takes time t_i to steer the sensor to the specified setting along a single dimension, where t_i is dependent on the actuator's speed and its current setting. We assume an actuator's speed is relatively constant, although there may be some mechanical jitter as we show in Section 7. Note that sense requests S_i may specify scans that also change the setting of one or more actuators

Our model matches low-level sensing device interfaces, where each actuation request, A_i, specifies a movement of the sensor along a single dimension, e.g., tilt, and movement in multiple dimensions, e.g., pan, tilt, and zoom, requires multiple actuation requests. For instance, a monitoring application for a PTZ camera might issue a repeating pattern of pan and tilt requests, followed by one or more capture requests to retrieve images, while a monitoring application for a steerable weather radar might tilt the antenna to the proper elevation and issue a repeating pattern of 360° sector scans. 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 sector scans. To enable fine-grained multiplexing, MultiSense interleaves requests from concurrent applications on the underlying physical sensor.

2.3 Challenges

We highlight challenges to multiplexing steerable sensors using a simple example with two users—Alice and Bob—sharing control of a single PTZ camera. The challenges for steerable weather radars are similar. 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 a naïve schedule 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. 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. 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 sec-

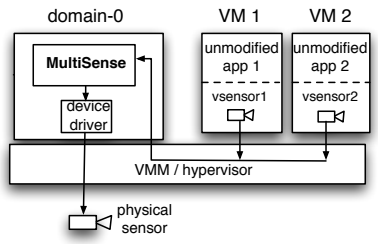


Figure 2: MultiSense Architecture Overview

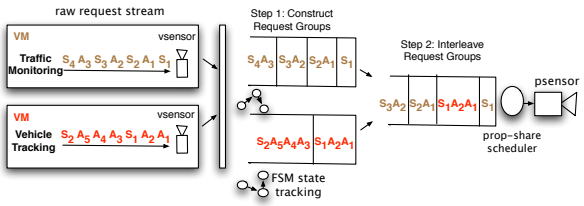


Figure 3: Constructing and interleaving request groups.

onds 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)).

This simple example motivates 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 [8]. This type of scheduling is difficult even on a dedicated sensor since, similar to a disk head, the mechanical steering mechanism has inherent jitter, which we show in Section 6.2.2.

Real-time scheduling similar to Rialto is also 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. This type of 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 VMMs 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 VMM or a privileged control domain—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 vsensor 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. Finite state machines track the state of each physical and virtual sen-

sor, where a state is an n-tuple that represents a specific setting for each of n actuators, and each state transition has a cost that denotes the time the sensor takes to complete the transition. 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. Likewise, steerable radars have both mechanical, e.g. scanning, and non-mechanical, e.g. pulse frequency, actuators. Each actuation modifies the state of one or more of these parameters, causing the sensor to transition from one state to another.

3.1 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 can perform state restoration by automatically issuing requests for each out-of-sync state parameter to synchronize the vsensor's state with that of the physical sensor. 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 change the zoom setting 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 2.3, where Alice issues P_aC_a followed by $P_a^2C_a^2$, and the P_a request causes the camera to move to pan position θ_a. Now suppose that Bob's request P_bC_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 $\theta_b = 50°$, $\theta_a = 30°$ and $\theta_a^2 = 75°$. Eager restoration pans from

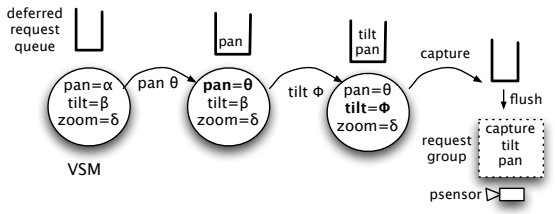


Figure 4: Request emulation and deriving request groups.

$50° \rightarrow 30° \rightarrow 75° = 65°$, 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. Now let $VSM_{prev} \cap VSM_{next}$ denote 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.

3.2 Request Groups

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 complete 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 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 perceive errors from actuations. We report any errors as a result of actuations when its corresponding sense request executes, similar to a file system that defers reporting disk errors until it empties the buffer cache.

4 Proportional-share for Steerable Sensors

MultiSense flushes request groups to a proportional-share scheduler that decides when to execute them. We adapt the standard Start-time Fair Queuing (SFQ) algorithm, originally designed for NICs and CPUs, to schedule steerable sensors by setting the length of a request group equal to the time the group would take to execute on the dedicated sensor [6]. As with other proportional-share schedulers, SFQ 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 work-conserving mode, actuation speeds may also become faster if any vsensor is not using its share by issuing requests. 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 during the interval. Thus, $\frac{W_i(t_1,t_2)}{W_j(t_1,t_2)} = \frac{w_i}{w_j}$, where W_i and W_j denote the total time the dedicated physical sensor consumes executing requests from vsensor i and vsensor j, respectively, during the interval.

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 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 $pan = -170^\circ\ tilt = -90^\circ\ zoom = 1x$ to $pan = 170^\circ\ tilt = 25^\circ\ zoom = 25x$. 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. SFQ also raises other issues when co-opted for steerable sensors. We discuss these issues below and present extensions to mitigate them.

4.1 Context-switch Costs

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 $25^\circ \rightarrow 30^\circ \rightarrow 75^\circ \rightarrow 40^\circ = 85^\circ$. 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 $25^\circ \rightarrow 30^\circ \rightarrow 45^\circ \rightarrow 75^\circ = 50^\circ$. For our PTZ camera, this results in nearly a 1 second reduction in overhead. We address this issue by extending SFQ to select the k pending requests with the smallest start tags, one from each vsensor, instead of selecting only the request with the minimum start tag.

Given a batch of k requests, we reorder them to minimize the physical sensor's total actuation time. In our example, this strategy selects the more efficient Alice $\rightarrow$ Carol $\rightarrow$ 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 [3]. 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.

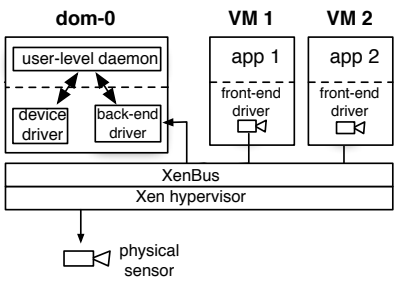


Figure 5: MultiSense's implementation uses Xen's split-driver framework to serve as a communication channel, and a user-level daemon in Domain-0 to maintain vsensor VSMs and execute scheduling policies. Each request passes from application → front-end driver → back-end driver → daemon → device.

4.2 Synchronous I/O

Applications, such as object tracking, must execute sense requests synchronously if they use the result to determine their next actuation. Proportional-share schedulers, such as SFQ, that track progress and make decisions using virtual clocks do not handle synchronous requests well because of deceptive idleness [7]. Synchronous requests prevent an application from queuing up additional requests for the scheduler to consider, which may cause the scheduler to pre-maturely context-switch after a synchronous request completes, but before the application is able to issue additional requests. Anticipatory scheduling addresses the problem by pausing for a period after the execution of a synchronous request, giving the currently executing application a small time window to issue subsequent requests for the scheduler to consider [7].

However, steerable sensors violate the assumption of anticipatory schedulers that requests from a single application always have similar degrees of spatial and temporal locality. Unlike disks, which control the layout of their own data, we cannot always assume that real-world events will be close to each other in space or time. As a result, while anticipatory scheduling does achieve better fairness properties, by preventing the virtual clocks of applications continuously issuing small requests from lagging behind, in many cases it actually decreases, rather than increases, performance for steerable sensors. We evaluate the impact of anticipatory scheduling and synchronous requests in Section 6.2.2.

4.3 Overlapping Requests

If concurrent applications are interested in similar events, our scheduler takes advantage of the spatial and temporal locality between the applications. This phenomenon is most prevalent for steerable weather radars that sense by performing sector scans of the atmosphere at specific elevations. During a severe weather event, it is likely that scanning algorithms run by different agencies or scientists will observe similar regions of the atmosphere. The concept also applies to PTZ cameras that scan continuous areas or capture bursts of activity.

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**

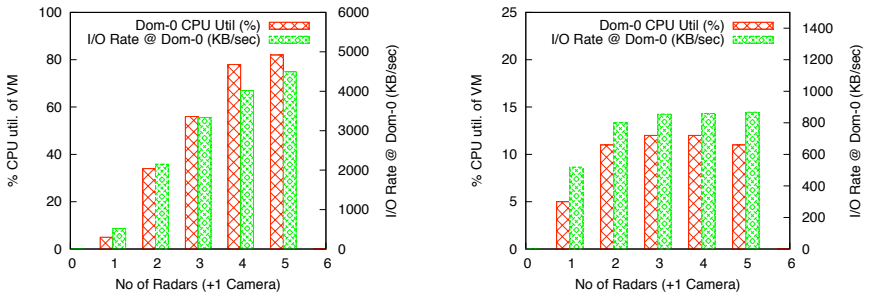
Table 1: Latency breakdown for a sample vsensor actuation of the Sony PTZ camera in our Xen implementation. The dominant factor in the request latency ($> 99.7\%$) is the time to actuate the camera. Our implementation imposes comparatively little overhead ($< 0.3\%$).

To account for these partially overlapping requests, our SFQ implementation merges multiple requests within a batch of k according to a simple policy: if any portion of the scans from two requests overlap the scheduler merges them and only executes the single merged request. MultiSense uses the data collected from the merged request to form the correct result for each individual request and return it to the respective application. If workloads exhibit a high level of overlap, the performance gains from merging are significant, as we show in Section 6.2.1.

5 Implementation

MultiSense integrates with XenLinux's virtual device framework. The sensors we study, which we describe in next section, are 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 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 4.2 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-



(a) No Actuation Costs (b) Includes Actuation Costs

Figure 6: Utilization of Domain-0's 40% CPU share and I/O rate for radars with and without actuation costs. The number of vsensors vary from 1 to 5, where the first vsensor is a PTZ camera and the remaining vsensors are radars.

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 is negligible. 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. We use the same pair for both the PTZ camera and the weather radar. 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/merging and anticipatory scheduling. 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 SFQ 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 3.2.

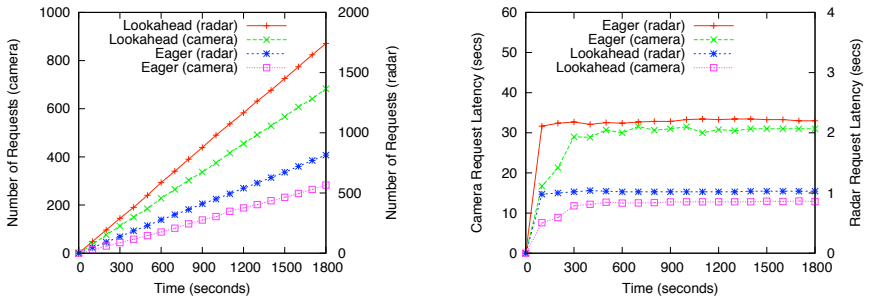
5.1 Example Sensors

We evaluate MultiSense for both PTZ cameras and steerable weather radars. We use 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 $1/3°$. We benchmarked the speed of each of the camera's actuators independently. 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 of the motor.

To study steerable weather radars, we developed an emulator, written in Java, modeled after the experimental IP1 radar deployed on the UMass-Amherst campus. The IP1 uses a direct-drive, high-torque azimuth positioner and linear actuator elevation positioner to reposition its antenna. The positioner is able to scan from $0°$ to $360°$ at a maximum speed of $120°/sec$, and change elevation, e.g., tilt, from $-2°$ to $30°$ at a maximum speed of $30°/sec$. The radar performs spiral scans and produces data at a maximum of 3 kilobytes/degree when sampling. Each actuation request specifies a start and stop position, which includes the azimuth and elevation angles of the antenna, for each scan. Our emulator imposes the necessary delays and outputs the appropriate volume of data for each request.

5.2 Benchmarks

Before evaluating MultiSense, we benchmark its implementation overhead. Our experiments run on Mac-Minis with 1.83 Ghz Intel T5600 CPUs, 1GB RAM, and 80GB SCSI disks, running version 3.2 of the Xen hypervisor with Ubuntu Linux using kernel version 2.6.18.8-xen in



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

Figure 7: The lookahead state restoration strategy outperforms the eager approach in our sample workloads. The number of requests completed (a) is 3x more and the average latency to satisfy each request (b) is 3x less using the lookahead approach.

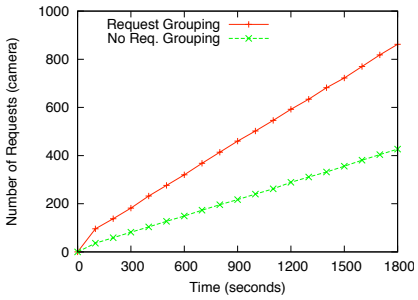


Figure 8: Request grouping improves performance by a factor of 2.

both Domain-0 and each guest VM. Each guest uses a file-backed virtual block device to store its root file system image. 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. We vary the number of VMs from 1 to 5, where the first VM controls the PTZ camera and the other VMs control the radar. Figure 6(a) shows the maximum achievable I/O rate that MultiSense is able to deliver to each vsensor by allowing our radar emulator to produce data as fast as possible with no delays from actuation overhead. The result demonstrates that MultiSense is able to handle an I/O rate of 4.6 MBps. This maximum I/O rate is 5x more than the maximum possible sensing rate including actuation overheads as shown by Figure 6(b), which uses a workload of random actuations. The experiment also demonstrates that MultiSense uses only 12% of Domain-0's 40% CPU share, or 4.8% of the total CPU, in this extreme case.

6 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 both the camera and radar 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 in a single actuator plane, e.g., pan, tilt, or zoom, in a single direction interspersed with sense requests, while the random workload repeatedly issues requests for random settings of the actuators followed by a sense request. Each scan issues a sense request every 10° starting at one extreme of the plane and moving to its other extreme. For the radar, the deterministic workload issues continuous scans between two extreme points at a specific elevation, while the random workload repeatedly issues scans between a random start and stop position. 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 a sensor's throughput—the number of requests it is able to satisfy per time interval. We first compare the

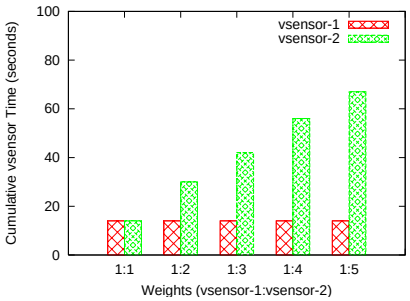


Figure 9: SFQ enforces performance isolation over large numbers of requests. The ratio of the total vsensor time for the two continuous scan workloads is in proportion to the assigned weights.

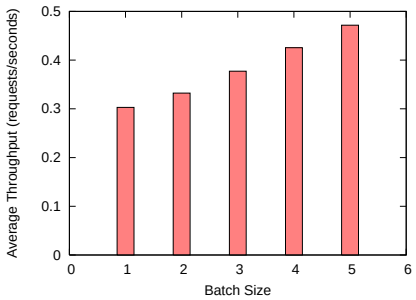


Figure 10: SFQ shows better global performance in terms of average throughput in requests/secondas as batch size increases. For this experiment, each increment in the batch size results in roughly a 10% improvement.

eager approach to state restoration, described in Section 3.1, with MultiSense's lookahead approach. Figure 7 shows results from an experiment using five vsensors, with batch size of 3, executing the random workloads described above, with both the radar and the camera. Figure 7(a) shows the progress of completed requests on the physical sensor for both approaches, while Figure 7(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. 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 Scheduling

The goal of SFQ is to enforce performance isolation between vsensors—each vsensor should receive performance in proportion to its weight. While SFQ bounds the maximum unfairness within any time interval, our extensions relax this bound to increase efficiency. We first demonstrate SFQ'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.

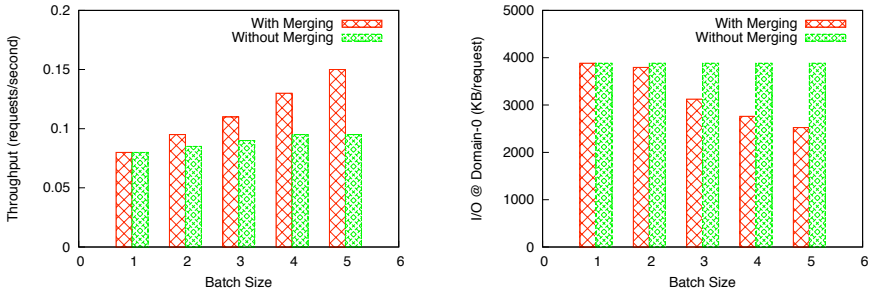
Our adaptation of SFQ advances virtual time in relation to the time each actuation consumes on the dedicated sensor, which we denote as vsensor time. The more vsensor time each actuation consumes the slower the actuator. Figure 9 shows for the camera the total vsensor time of two vsensors with different weight assignments using our variant of SFQ, where 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. The isolation of radar vsensors is similar.

6.2.1 Request Batching and Merging

Figure 10 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 12 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 10. 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.

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, a batch size of 3 exhibits an appropriate balance by increasing performance by 20% while achieving enforcing similar fairness properties. In practice, we have found that a batch size of roughly half the number of active vsensors strikes the appropriate balance. We also evaluate the effect of batching with and without request merging for the radar, since its sensing requests are sector scans that may cause overlap among concurrent requests from different vsensors.

Figure 11(a) shows that merging results in a 75% improvement over batching without merging for multiple batch sizes. Figure 11(b) also shows that merging de-



(a) Average Throughput (b) I/O Rate

Figure 11: Request merging (a) results in a 75% improvement in throughput and a 35% decrease in I/O rate (b) for our example workloads compared to no request merging.

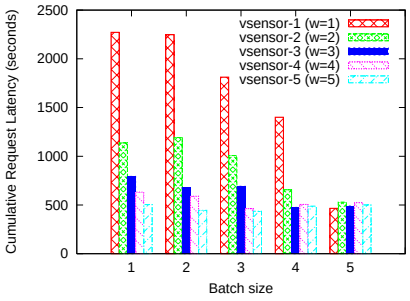


Figure 12: SFQ exhibits less fairness as the batch size increases in terms of the average latency per request.

creases aggregate I/O (data per request) by nearly 35%. Finally, we explore how the degree of overlap present in a workload affects performance. Figure 13 shows the average request latency from two vsensors executing workloads with different degrees of overlap. We set each vsensor to issue 180° sector scans, and vary the starting point of one vsensor to control the size of the overlap. The experiment shows that the average request latency approaches that of a dedicated sensor as the degree of overlap approaches 100%, and that without request merging the average request latency is 1.5x higher.

6.2.2 Anticipatory Scheduling

Figure 14 shows the performance impact of anticipatory scheduling using the camera. Anticipatory scheduling has similar results for the radar. If MultiSense does not use anticipatory scheduling, vsensors must fill the scheduler's queue with multiple sense requests by either issuing them asynchronously or issuing them on separate threads, which is problematic if an application needs the result of a sense request to determine its next position. This experiment charts the actuation speed of two vsensors over time executing random workloads with and without anticipatory scheduling, where each point is an average of 5 actuation requests. In this experiment we only use the pan and tilt actuators so we can quantify speed in terms of degrees/second.

As Figure 14 demonstrates, using anticipatory scheduling with request patterns that have low spatial and temporal locality results in a vsensor that is roughly 25% slower. The experiment also demonstrates how weight translates to the absolute speed of the actuator. Without anticipatory scheduling, the average speed for the dedicated sensor is 23 degrees/second, while the average speed for the vsensor with weight=1 is 12 degrees/sec and with weight=2 is 19 degrees/sec. The average speed for the dedicated sensor is less than the maximum speed in Section 5.2 due to the random workload, which includes numerous short requests. Both 12 and 19 are roughly 50% and 80% of the 23 degrees/second possible with the physical sensor. In this example, the speeds are higher than the vsensors' relative weights because of the proximity of requests and the efficiency increase of batching. The variance in speed for the vsensors is greater than that of the dedicated sensor, which highlights the loose relationship between weight and absolute speed for steerable sensors.

6.3 Case Studies

Our case study explores MultiSense's use with four example applications with specific performance metrics that are applicable to both the camera and radar. In the experiments, we use the lookahead state restoration approach, request groups, and proportional-share scheduling.

- **Continuous Monitoring**. For the camera, continuously pan in increments of 65° and capture an image, while for the radar, continuously execute 360° sector scans at a specific elevation. The performance metric is the time to cover the sensor's entire range.
- **Fixed-point Sensing**. For the camera, pan, tilt, and zoom the lens to a fixed point and repeatedly capture images at a regular interval, while for the radar, execute the same 30° sector scan at a specific elevation. The performance metric is the sensing rate.

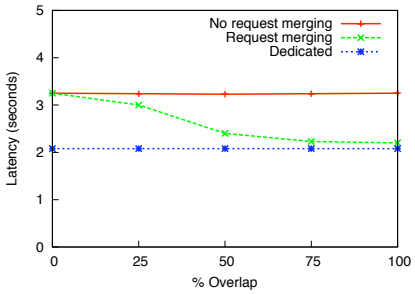


Figure 13: Request merging takes advantage of overlapping requests to increase the aggregate performance.

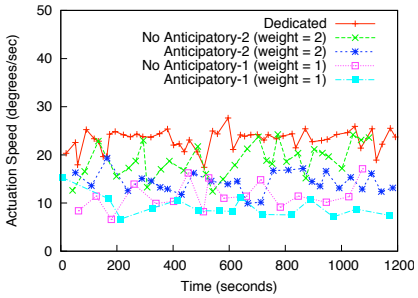


Figure 14: Anticipatory scheduling decreases performance, in terms of actuator speed, for steerable sensors when there is little spatial and temporal locality.

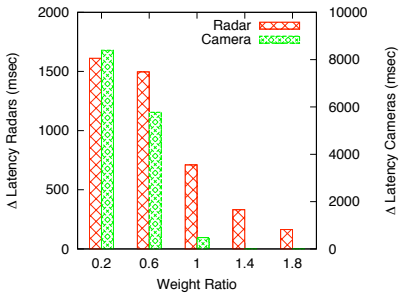


Figure 15: The difference in latency when coordinating multiple vsensors on different nodes to sense the same point.

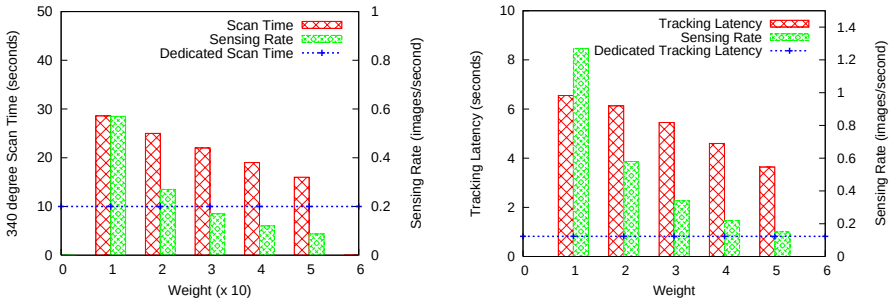
- **Object Tracking** For the camera, periodically track a pre-defined path along both the pan and tilt axes and capture images every 10°, while for the radar execute small sector scans every 30°. The performance metrics are both the latency between sensing requests, and the minimum overall latency necessary to keep up with the moving object.
- **Multi-sensor Fixed Point Sensing**. For two cameras, pan, tilt, and zoom the lens to the same fixed point and repeatedly capture images at a regular interval, while for two radars, scan the same 30° sector at the same elevation. In both cases, both sensors 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. With the radar, fixed-point sensing with a dedicated sensor is able to scan the same 30° sector every 0.5 seconds, but executing a random workload of 30° scans reduces the rate to every 1.5 seconds. We use these sensing rates for comparison in our case study below.

We first execute both continuous monitoring (Figure 16(a) and Figure 17(a)) and object tracking (Figure 16(a) and Figure 17 (b)) concurrently with the fixed-point sensing application for both the camera and the radar. In both cases, 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 and Figure 17 shows the results for the radar, 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 in this experiment. Since weight dictates performance, some applications may need a minimum weight to satisfy their requirements.

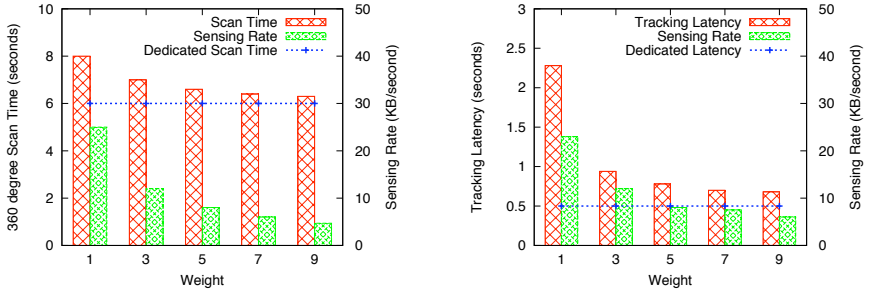
Consider continuous monitoring for the camera with a 1:30 weight ratio, the application is able to pan all 340° in 20 seconds. Thus, in the real-world, the monitoring application is able to capture 4 distinct points 113 feet apart, e.g. four doorways, at distance of 100 feet from the camera every 5 seconds [2]. 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.

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



(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.



(a) Continuous Monitoring (b) Object Tracking

Figure 17: For the radar, 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.

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 tracking, the fixed-point sensing application maintains a sensing rate of 0.3 images/second.

Now consider continuous monitoring for the radar. With a 1:5 ratio, the radar is able to complete a 360° scan in about 7 seconds, while simultaneously scanning the same 30° sector every 11 seconds. Even if we assume that a thunderstorm travels 60 mph, which is relatively fast, the continuous monitoring application is able to capture the storm's movement every mile. If we track the storm the performance is even better. In this case, for a 1:3 ratio, the radar is able to scan a 30° sector every second, while sensing a fixed-point every 11 seconds. Since the radar we emulate has a range of 25 miles, we are able to capture the storm's movement every $\frac{1}{60}$ miles or 88 ft. These situations translate into other real-world events as well. For instance, fixed-point sensing is useful for monitoring the core of a slow moving storm, while object tracking is also useful for tracking a tornado.

[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.

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 for both the camera and the radar. 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 assigment the sensors sense the same point within 2 seconds of each other, which is suitable for a range of scenarios, such as estimating three-dimensional wind direction for radars or pedestrian entry/exit points for cameras.

7 Related Work

MultiSense applies the proportional-share paradigm, which has been well-studied in other contexts, to mul-

tiplex control of steerable sensors. SFQ was originally prototyped for multiplexing packet streams and later extended to CPUs [6]. More recently, there has been work on proportional-share scheduling for energy—another non-traditional resource—using virtual batteries [4]. PixieOS also uses proportional-share scheduling techniques to enable explicit application-level control of CPU, memory, bandwidth, and energy for motes [10]. We extend the paradigm to include the actuation resources of steerable sensors. However, our work focuses on a type of sensor network that does not have the same energy or computing constraints as mote-class sensor networks, and, consequently, does not face the same problems.

Perhaps most related to MultiSense is past work on proportional-share scheduling for disks. Disk schedulers incorporate a similar batching technique [3] and often group together write requests and flush them to disk on after a read request occurs. However, there are fundamental differences in the relative speed of the actuators and their use that present different trade-offs for steerable sensors. Rather than modeling the shared resource as I/O bandwidth or number of I/Os, which is often the case for disks [12], we use aggregate time controlling the sensor, or equivalently its speed, since the responsiveness of the sensor determines when and what applications are able to sense. We also evaluate the effect of optimizations such as merging and anticipatory scheduling for steerable sensors, which have different workload characteristics than disks.

MultiSense uses Xen's [1] basic abstractions for multiplexing I/O devices [14]. However, MultiSense does not implement conventional device virtualization that delegates control of an entire peripheral device or bus to a VM by passing device requests through the hypervisor. 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. Thus, 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. As with prior work on device drivers, we structure devices as state machines, which is a natural choice for stateful devices [11].

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 an application's 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 that demonstrate their effectiveness. Finally, we use our techniques in case studies for two sensors that show their behavior for four real applications.

References

[1] 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. *SOSP*, October 2003.

[2] 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.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling With Quality Of Service Guarantees. *International Conference on Multimedia Computing and Systems*, July 1999.

[4] Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. Abdelzaher. Virtual Battery: An Energy Reserve Abstraction For Embedded Sensor Networks. *RTSS*, November 2008.

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

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

[7] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework To Overcome Deceptive Idleness In Synchronous I/O. *SOSP*, October 2001.

[8] M. Jones, D. Rosu, and M. Rosu. CPU Reservations And Time Constraints: Efficient, Predictable Scheduling Of Independent Activities. *SOSP*, October 1997.

[9] M. Li, T. Yan, D. Ganesan, E. Lyons, P. Shenoy, A. Venkataramani, and M. Zink. Multi-user Data Sharing In Radar Sensor Networks. *SenSys*, November 2007.

[10] K. Lorincz, B. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource Aware Programming In The Pixie Operating System. *SenSys*, November 2008.

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

[12] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework For Next Generation Operating Systems. *SIGMETRICS*, June 1998.

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

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

[15] M. Zink et al. Meteorological Command And Control: An End-to-end Architecture For A Hazardous Weather Detection Sensor Network. *Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services*, June 2005.