Multimedia Systems
DOI 10.1007/s00530-012-0292-y

REGULAR PAPER

MultiSense: proportional-share for mechanically steerable sensor networks

Navin Sharma · David Irwin · Michael Zink · Prashant Shenoy

Received: 1 October 2011 / Accepted: 12 July 2012

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. Although 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 (1) exposing a virtual sensor to each application and (2) optimizing the time to context-switch between virtual sensors and satisfy requests. We implement MultiSense in Xen, a widely used virtualization platform, 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 s.

Keywords Virtualization · Sensor · Camera · Radar

Communicated by T. Plagemann.

N. Sharma (✉) · D. Irwin · M. Zink · P. Shenoy
University of Massachusetts, Amherst, USA
e-mail: nksharma@cs.umass.edu

D. Irwin
e-mail: irwin@cs.umass.edu

M. Zink
e-mail: zink@ecs.umass.edu

P. Shenoy
e-mail: shenoy@cs.umass.edu

1 Introduction

Steerable sensor networks allow applications to steer 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 [33]. The US border patrol is also deploying networks of pan-tilt-zoom (PTZ) cameras to continuously monitor the northern border for smugglers [18], and as part of a "virtual fence" on the southern border [10]. 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 $250,000, 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—

Published online: 26 July 2012

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 [18]. Likewise, scientists now control dedicated networks of steerable weather radars in different ways to accomplish different sensing tasks, such as rainfall estimation or tornado tracking [33]. While simple multiplexing approaches, which schedule control in a coarse-grained batch fashion, are possible [6], 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 multiplexing, including proportional-share scheduling, have 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 (spatial and temporal) 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 extends a proportional-share scheduler to multiplex virtual sensors on a single physical sensor.

While we could implement sensor multiplexing 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 virtual machine performance isolation to include sensors, in addition to other compute resources. We discuss the motivation behind this implementation choice further in Sect. 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, with different sensing modalities. 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 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 introduce actuator fair queuing (AFQ) by modifying start-time fair queuing [11] 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 AFQ's isolation and the benefit of each extension.

Implementation and experimentation We implement MultiSense in Xen, a widely used system for creating and managing virtual machines [2], 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 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 s.

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

2 Background

The primary problem addressed in this paper is how to multiplex ("time-share") a steerable sensor, such as a PTZ camera or a weather radar, at a fine time scale across multiple concurrent users with diverse requirements. 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 approach builds on many different areas, including sensor networks, platform virtualization, and proportional-share scheduling, to virtualize stateful sensors with actuators. We discuss prior work in these areas in detail in Sect. 7. The goal is to lower the barrier for Domain scientists to experiment with expensive steerable sensor networks, such as radar systems or the border patrol's virtual fence. This work is in conjunction with a testbed of steerable sensors we have deployed. In this scenario, virtual machines (VMs) serve to isolate both a 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 one or more attached steerable sensors.

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 independent 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 and weather radars as representative examples 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 request 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:

$[A_1 A_2 \ldots A_n S_1 S_2 \ldots S_m]^+, n \geq 0, m > 0$, 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. While we focus on virtualizing the sensor at its lowest level interface, our system model does not preclude higher-level interfaces for interacting and controlling sensors, e.g., combining many actuation and sensing commands to track an object along a path.

We assume a constant actuator speed, although there may be some mechanical jitter as we show in Sect. 6.2.2. 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 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 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. We highlight the challenges in multiplexing stateful steerable sensors using the following examples.

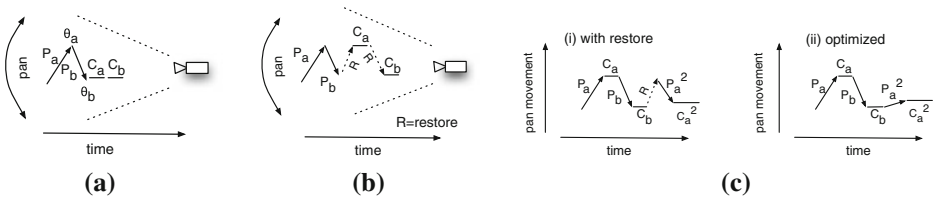


Fig. 1 Examples showing why request interleaving is challenging for steerable sensors. **a** Naive interleaving $P_aP_bC_aC_b$ yields incorrect results, **b** correct interleaving $P_aP_bC_aC_b$ with state restorations and **c** optimizing state restoration costs

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. For a PTZ camera, panning the camera (P_i) is an instance of actuation (A_i) from above, and capturing an image (C_i) is an instance of sensing (S_i) from above. Consider naive 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 Fig. 1a).

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, naive 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 s to pan from 0 to 340°, nearly 4 s to tilt from 0 to 115°, and over 2 s to zoom from 1 to 25x. Naive 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 (Fig. 1b). Better interleavings, such as $P_aC_aP_bC_b$, still pose a problem for a naive 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 (Fig. 1c).

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 per 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, which we show in Sect. 6.2.2.

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.

Note that share-based scheduling controls each vsensor's priority using shares or weights. Thus, MultiSense is capable of changing a vsensor's priority, and its performance, by changing its share of the sensor. For example, in an extreme case, assigning 100 % of the shares to a vsensor, and reducing the shares of other vsensors to zero, would allow a vsensor to have the same performance as a dedicated physical sensor.

Since our goal is to study how well MultiSense could satisfy conflicting demands of concurrent applications using proportional-sharing techniques and various optimizations, the study of an admission controller and the effects of preempting currently executing applications is out of the scope of this paper. In fact, assigning 100 % weight to a critical application and 0 % to others in a proportional-sharing technique could simulate the job of an admission controller.

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—Domain-0 in Xen—and interleaves requests from each vsensor on the underlying physical sensor, as shown in Fig. 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 Sect. 4. We use the term

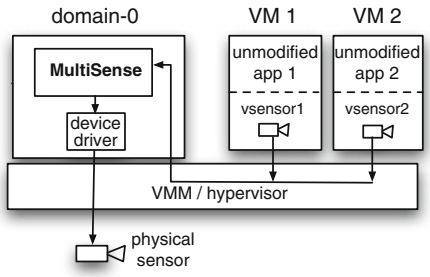


Fig. 2 MultiSense architecture overview

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 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) with 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

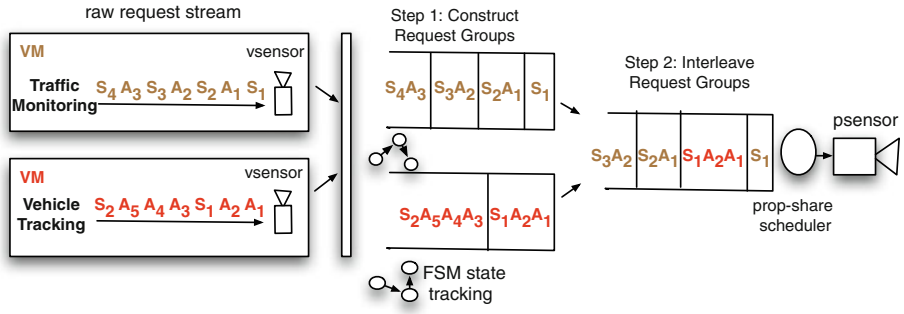


Fig. 3 Constructing and interleaving request groups

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 Sect. 2.2, 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 Fig. 1c, 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 $50° \rightarrow 30° \rightarrow 75° = 65°$, 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 s.

MultiSense avoids this overhead using a lookahead strategy that does not restore state parameters that queued vsensor actuations 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. $\text{VSM}_{\text{prev}} \cap \text{VSM}_{\text{next}}$ now denotes the set of state parameters not modified by these requests. The lookahead strategy only restores the states in $\text{VSM}_{\text{prev}} \cap \text{VSM}_{\text{next}}$. In the Alice and Bob example, $\text{VSM}_{\text{prev}} \cap \text{VSM}_{\text{next}}$ includes the parameters zoom and tilt, but not pan, since Alice's queued request will modify the pan parameter.

3.3 Request grouping via request emulation

To eliminate wasteful state restoration overheads, MultiSense automatically groups requests from each vsensor that the physical sensor should execute atomically (Fig. 3). 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 Fig. 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

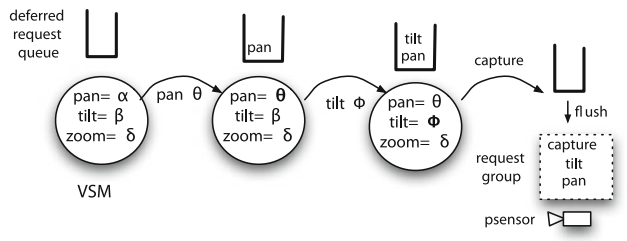


Fig. 4 Request emulation and request groups

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 dictate an application's control flow, the application will observe errors prior to making control flow decisions.

Likewise, since MultiSense adds an additional layer in the software stack that controls the issuing of requests to the physical sensor, it is capable of including logic that prevents cascading errors from unknowing applications that may damage the sensor. For example, an application might damage a PTZ camera or radar by issuing actuation commands too fast. MultiSense's additional layer in the software stack between the application and the raw sensor provides a location to implement filters that assure the sensor is not operated in an unsafe manner.

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 [11].

As background, we provide a brief summary of SFQ. Start-time fair queuing 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 actuators: 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 k'th request of vsensor i; $F(r_i^{k-1})$ denotes the finish time of the previous request; $v(t)$ represents virtual time, described below; and $A(t)$ represents the actual arrival time of the request. The start tag of a request is the maximum of the virtual time at arrival or the finish tag of the previous request.
- The finish tag of a request is $F(r_i^k) = S(r_i^k) + \frac{l_i^k}{w_i}$, where l_i^k denotes the length of the k'th request and w_i denotes the weight assigned to vsensor i. Intuitively, the finish tag of a request is its start tag incremented by the length of time required to execute the entire critical section, normalized by the vsensor's weight. To enable precise computation of l_i^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

Actuator fair queuing 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 three important extensions, discussed in the following sections, 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 work-conserving mode, actuation speeds may also become faster if any vsensor is not using its share by being passive. 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 as our foundation because it bounds the resulting unfairness due to this discrete granularity by ensuring that $|\frac{W_i(t_1,t_2)}{w_i} - \frac{W_j(t_1,t_2)}{w_j}| \leq (\frac{lmax_i}{w_i} + \frac{lmax_j}{w_j})$ 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 s for our camera, one goal of our evaluation is to explore performance in the common, rather than the worst, case for representative applications. Start-time fair queuing also raises other issues when co-opted for steerable sensors. We discuss these issues below and present AFQ's extensions to mitigate them.

4.2 Context-switch costs and batching

Start-time fair queuing 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°. Start-time fair queuing 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 s 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 [7]. 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 Sect. 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.

4.3 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 [12]. 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 [12].

However, mechanically 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 mechanically steerable sensors. We evaluate the impact of anticipatory scheduling and synchronous requests in Sect. 6.2.2. We note that even without assumptions about spatial and temporal locality, anticipatory scheduling may have benefits for new electronically steerable radars [25] that do not have steering latency, because some applications will still need to interpret data from a previous request before issuing a subsequent request to the scheduler. We leave evaluating the impact of anticipatory scheduling on electronically steerable sensors as future work.

4.4 Overlapping requests and request merging

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.

To account for these partially overlapping requests, our AFQ 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. For example, if Alice sends a scan request of 10 to 50° and Bob sends a scan request of 25 to 75°, our AFQ algorithm merges the two scan requests and sends a single scan request of 10 to 75° to the physical radar. 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 Sect. 6.2.1.

5 Implementation

MultiSense integrates with Xen Linux's virtual device framework. Xen is a widely used platform for creating and managing virtual machines on servers [2]. Xen already includes support for sharing a server's CPU, memory, and common I/O devices, such as disks. The sensors we study, which we describe in Sect. 5.1, 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 [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.

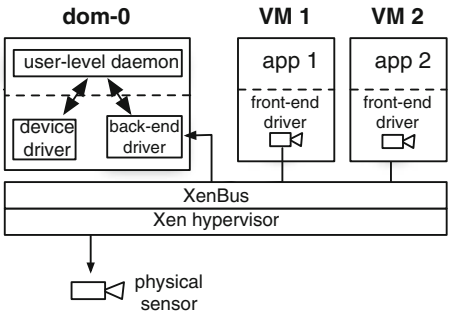


Fig. 5 MultiSense uses Xen's split-driver framework for communication, and a user-level daemon in Domain-0 to maintain vsensor VSMs and execute scheduling policies. Each request passes from application → front-end driver → back-end driver → daemon → device

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 Sect. 5.3, 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 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 Sect. 3.3.

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 (1–25x). 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 (Fig. 6). The camera is capable of panning at 40°/s, tilting at 30°/s, and zooming at 12x/s, 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.

Similar to PTZ cameras, steerable weather radars are also an example of multimedia systems.[1] As shown in Fig. 7, weather radars scan and produce high-resolution images of severe weather conditions, such as tornado, which are used by various government agencies and research communities to issue weather warnings and study weather behaviors, respectively. A network of weather radars, with possibly overlapping regions (Fig. 7), is often used to observe and predict correct weather conditions.

[1] One could compare a weather radar to a more complicated camera that works at a lower frequency.

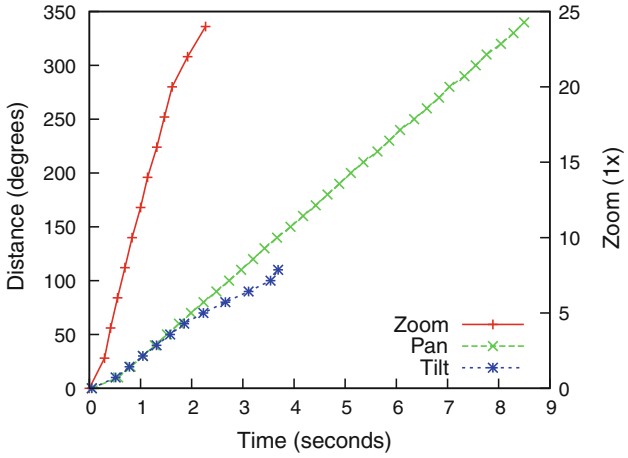


Fig. 6 Time benchmarks for the pan, tilt, and zoom actuators of sony SNC-SRZ30N network camera that show the distance moved by the actuator is roughly linear to the time and energy required to complete the actuation

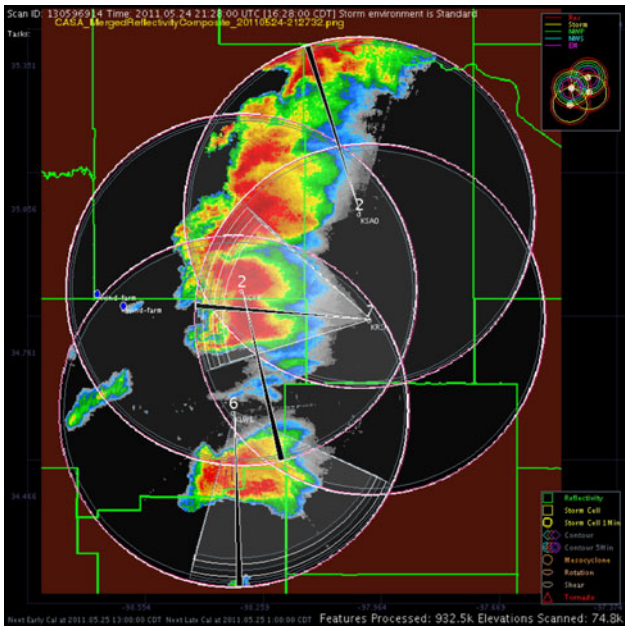


Fig. 7 Image of a strong tornado in Oklahoma region taken by a network of four weather radars

To study steerable weather radars, we developed an emulator, written in Java, modeled after the experimental IP1 radar (Fig. 8). The IP1 uses a direct-drive, high-torque azimuth positioner and linear actuator elevation positioner to reposition its antenna [19]. The positioner is able to scan from 0 to 360° at a maximum speed of 120°/s, and change elevation, e.g., tilt, from −2 to 30° at a maximum speed of 30°/s. The radar performs sector scans and produces data at a maximum of 3 Kb/° when sampling. Each actuation request specifies a start and stop position, which includes the azimuth and elevation angles of the antenna, for each

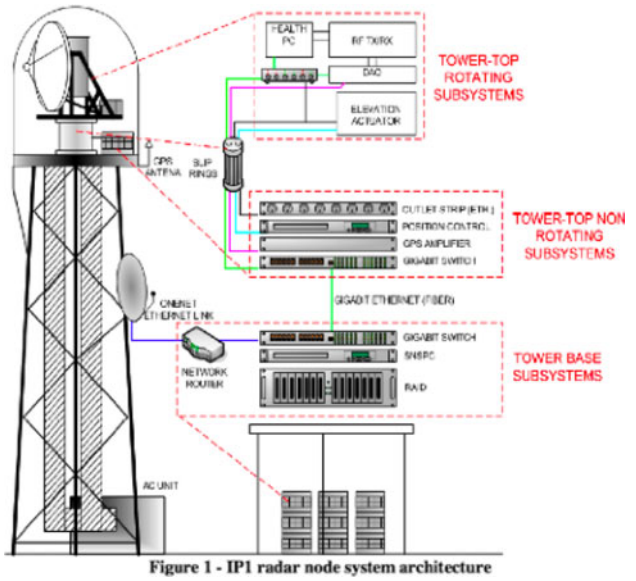


Fig. 8 Architecture of an IP1 radar node. The tower top rotating assembly contains the radar antenna, transceiver, data acquisition system, and elevation actuator—all mounted on a frame atop the azimuth positioner. On the radome floor, the nonrotating subsystems include a gigabit Ethernet switch, Ethernet controlled power outlet strip, GPS amplifier, and position controller computer. A gigabit Ethernet optical fiber links the tower top with the data processing server on the ground. The site is connected to the Internet through a radio link

scan. Our emulator introduces necessary delays and generates right volume of data to emulate scan time and radar data, respectively.

5.2 Radar emulator

The emulator consists of three modules—a receiver, executor and dispatcher—where each module executes in a separate thread. The receiver opens a TCP socket and listens incoming steering requests. After receiving a request, the receiver checks its validity before appending it to circular request queue. Invalid requests return with an appropriate error code. As with the IP1, valid steering requests specify an identifier, as well as azimuth and elevation start angles and stop angles within a specified range. Our emulator keeps other radar parameters, such as the antenna speed or the transmitter waveform, constant. The emulator maintains a constant antenna speed of 120°/s, which represents the maximum the IP1 is able to sustain. The IP1 antenna executes sector scan in multiple elevations, where the antenna sweeps over complete azimuth range, then steps up the antenna's elevation by 1°, and repeats sweeping a complete azimuth range, and so on. The IP1 generates data at a rate of 3 Kb/°.

The executor module, which maintains the physical state of the radar, is the core part of the emulator. The module dequeues each request from the circular request queue, and calculates its positioning latency and execution time. We define positioning latency as the time to move the radar from its current position to the start position of the request. Thus, the execution time represents the time required to complete a sector scan from the request's start position to its end position. The executor also (1) computes the quantity of data generated by the sector scan, (2) waits for the appropriate amount of time to satisfy the steering latency, and (3) creates and enqueues a response into a dispatcher queue. The size of response header and payload is identical to the same request on the IP1. Finally, the executor updates its internal state vector for the radar and services the next request. The dispatcher module serves only to dequeue responses from the dispatcher queue and transmit responses back to the back-end user level daemon.

Table 1 Latency breakdown for a sample vsensor actuation of the Sony PTZ camera in our Xen implementation

From	→ To	Latency	Percentage
Application	→ Front-end	0.24 µs	7.1×10^{-8}
Front-end	→ Back-end	6.35 µs	1.9×10^{-4}
Back-end	→ Listener	286 µs	8.51×10^{-3}
Listener	→ Camera	274 µs	8.15×10^{-3}
Camera	→ Listener	3.35 s	99.7
Listener	→ Back-end	17 µs	5.1×10^{-4}
Back-end	→ Front-end	27 µs	8.0×10^{-4}
Front-end	→ Application	229 µs	6.8×10^{-3}
Total		3.36 s	100

The dominant factor in the request latency (>99.7 %) is the time to actuate the camera

Our implementation imposes comparatively little overhead (<0.3 %)

5.3 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 80 GB 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. 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 using a user-level daemon in Domain-0. As Table 1 shows, the overhead of these additional layers is minimal compared (order of µs) 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

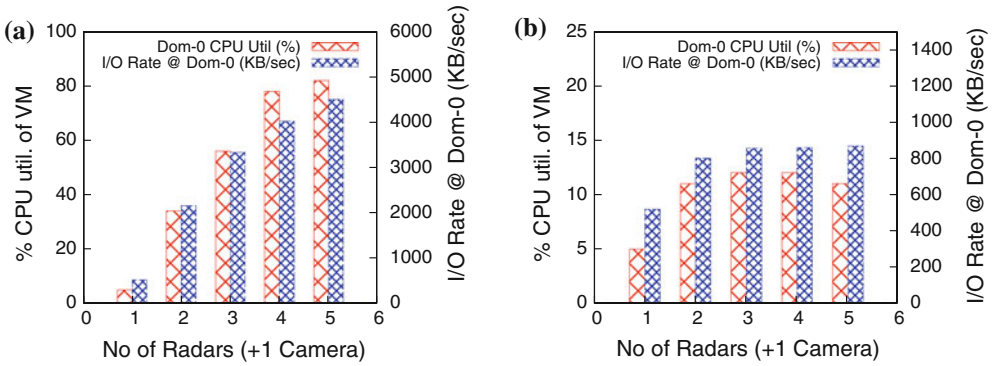


Fig. 9 Utilization of Domain-0's 40 % CPU share and I/O rate for radars with and without actuation costs. The number of vsensors vary from 1 to 5, where the first vsensor is a PTZ camera and the remaining vsensors are radars. **a** No actuation costs and **b** includes actuation costs

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 9a 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 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]. This maximum I/O rate is 5x more than the maximum possible sensing rate including actuation overheads as shown by Fig. 9b, 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

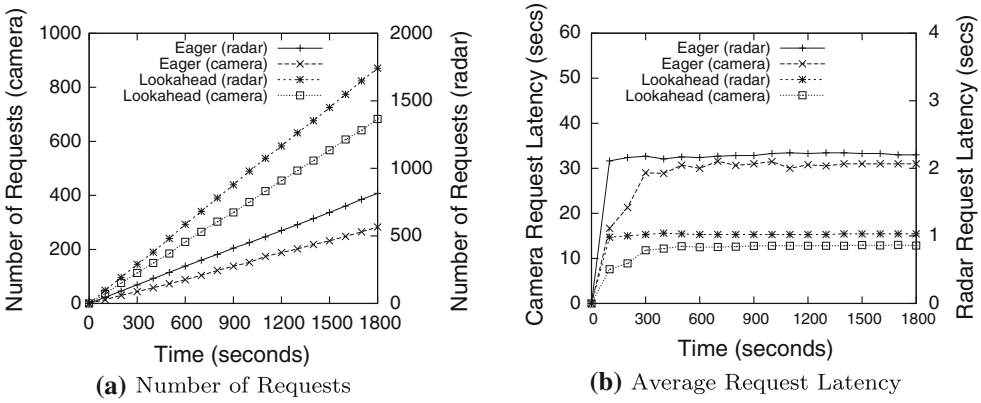
We first evaluate the impact of MultiSense's strategies for state restoration, request groups, and scheduling individually using synthetic workloads. The experiments demonstrate the extent to which these optimizations improve request throughput and latency. MultiSense's primary metric for success is whether or not it accommodates real concurrent applications. We present a case study for 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 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. For the radar, the deterministic workload issues continuous scans between two extreme points 180° apart 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 Sect. 6.3.

6.1 State restoration and request groups

We demonstrate the impact of state restoration and request grouping, independent of our scheduling policy, on throughput—the number of requests a sensor is able to satisfy per time interval. We first compare the eager approach to state restoration, described in Sect. 3.2, with our lookahead approach. Figure 10 shows results from an experiment using five vsensors, with batch size of three, executing the random workloads described above, with both the radar and the camera. Figure 10a shows the progress of completed requests on the physical sensor for both approaches, while Fig. 10b 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 min 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 11 shows the results. Using



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

Fig. 10 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

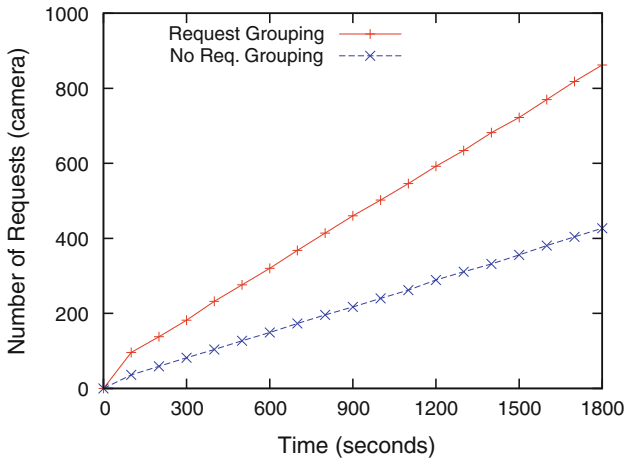


Fig. 11 Request grouping improves performance 2x

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 (Fig. 20), a 2x increase in request latency would mean capturing an image every 6 s, versus capturing it every 3 s.

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. 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.

Actuator fair queuing 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 12 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 Fig. 13, AFQ proportionally distributes share of the passive vsensor (vsensor-5's share during time interval 500–1,000 and 1,500–2,000 s) among active vsensors (vsensor-1 and 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, Fig. 14 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 (one–four) and one running the random workload (five). The result demonstrates that over short time periods SFQ is not always fair: during the period 0–100 s 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 receives similar performance by time 200 and vsensor-3/vsensor-2 receives 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 s.

6.2.1 Request batching and merging

Figure 15 demonstrates the performance improvement from batching for the camera. The experiment uses random workloads from five vsensors to stress actuation, and shows that the average throughput increases as the batch size

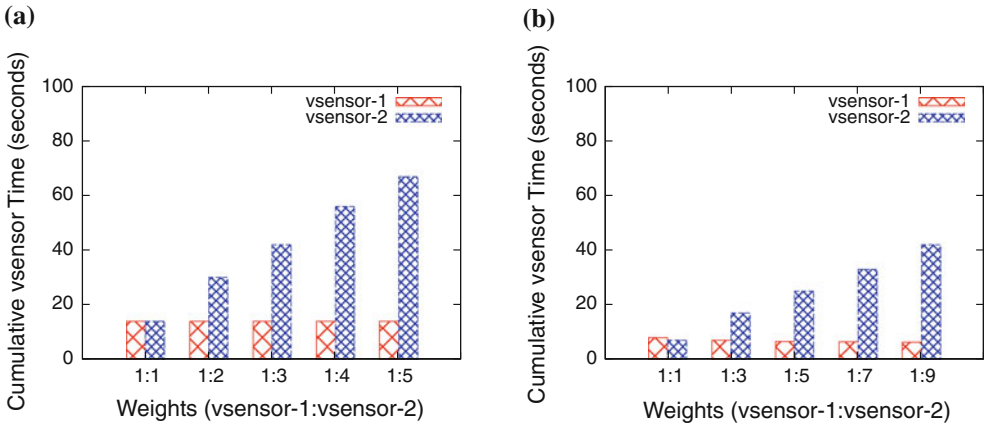


Fig. 12 AFQ enforces performance isolation over large numbers of requests. The ratio of the total vsensor time for the two continuous scan workloads is in proportion to the assigned weights. **a** Camera and **b** radar

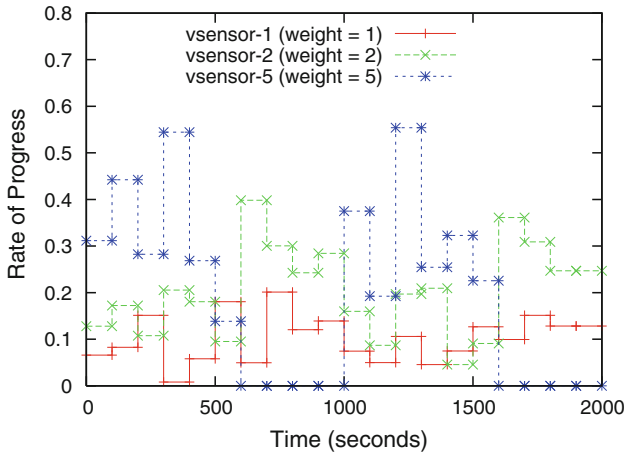


Fig. 13 AFQ maintains the work-conserving property when applied to actuators

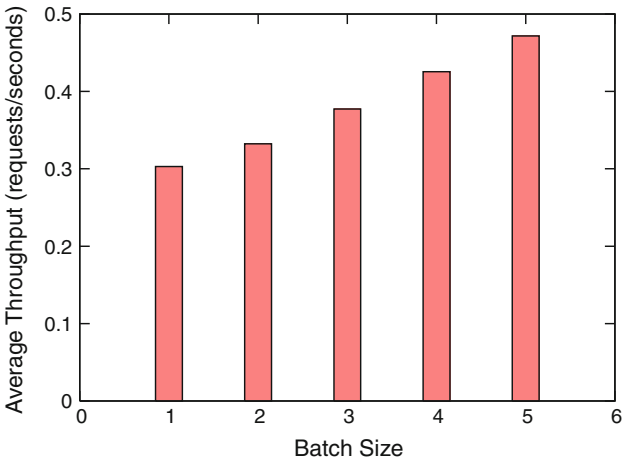


Fig. 15 AFQ shows better global performance in terms of average throughput in requests/s as batch size increases. For this experiment, each increment in the batch size results in roughly a 10 % improvement

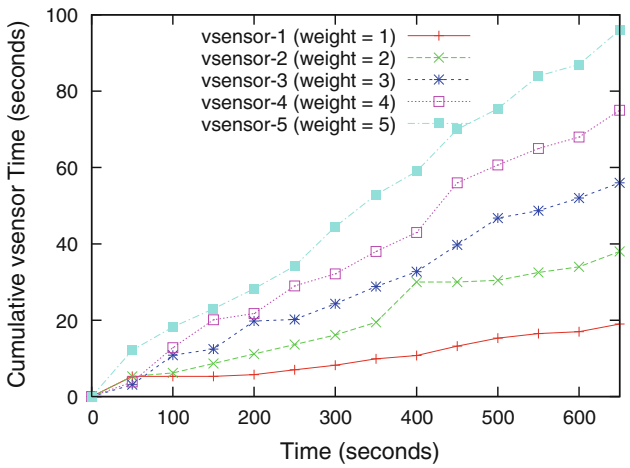


Fig. 14 While AFQ enforces performance isolation over many requests, it may diverge slightly, due to high state restoration costs, over short intervals of time

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 16 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 Fig. 15. 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 16b plots the cumulative vsensor time over the course of the experiment for a batch size of four. Comparing the result with Fig. 14 in the previous section emphasizes the decrease in performance isolation. As expected, SFQ, which corresponds to a batch size of one, 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

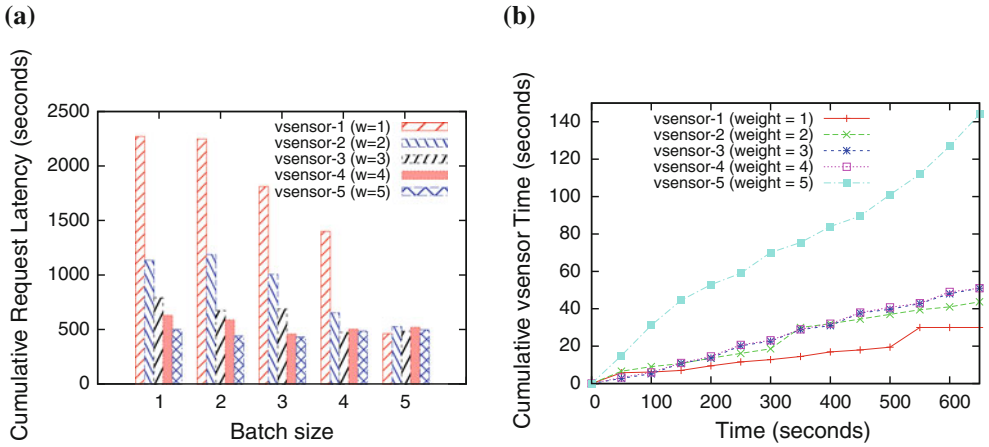


Fig. 16 AFQ exhibits less fairness as the batch size increases (a) in terms of the average latency per request. For this experiment, batch sizes of four and greater are unfair, and exhibit much less performance isolation (b) than SFQ. **a** Fairness versus batch size and **b** unfairness with batch size equals to four

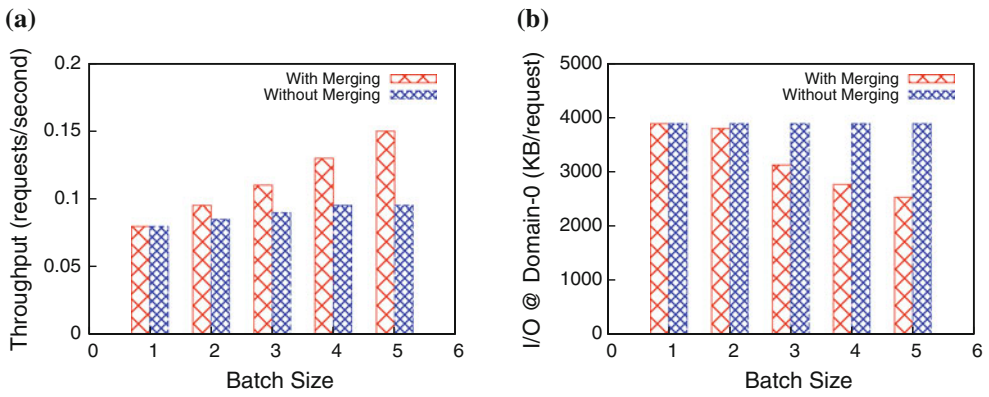


Fig. 17 Request merging **a** results in a 75 % improvement in throughput and a 35 % decrease in I/O rate **b** for our example workloads compared to no request merging. **a** Average throughput and **b** I/O rate

workloads, a batch size of three 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.

Using the same setup as the experiments above, we 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 17a shows that merging results in a 75 % improvement over batching without merging for multiple batch sizes. Figure 17b also shows that merging decreases aggregate I/O (data per request) by nearly 35 %. Finally, we explore how the degree of overlap present in a workload affects performance. Figure 18 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 19 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 requests by either issuing them asynchronously or issuing them on separate threads. 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 five actuation requests. In this experiment, we only use the pan and tilt actuators so we can quantify speed in terms of degrees/second.

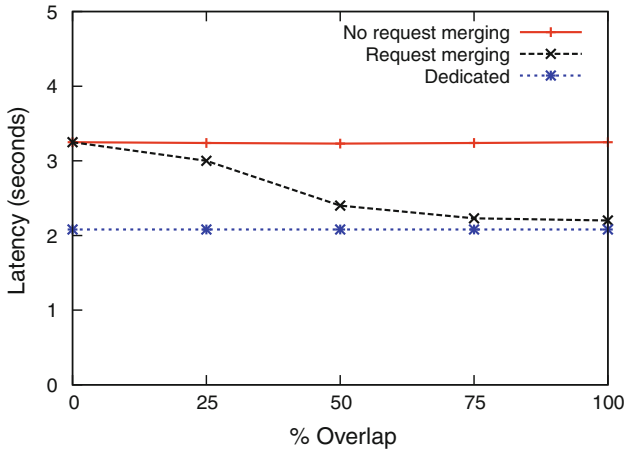


Fig. 18 Request merging takes advantage of overlapping requests to increase the aggregate performance

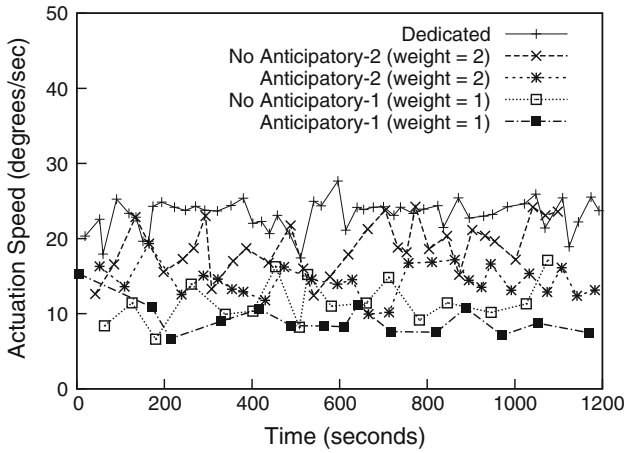


Fig. 19 Anticipatory scheduling decreases performance, in terms of actuator speed, for steerable sensors when there is little spatial and temporal locality

As Fig. 19 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°/s, while the average speed for the vsensor with weight = 1 is 12°/s and with weight = 2 is 19°/s. The average speed for the dedicated sensor is less than the maximum speed in Sect. 5.1 due to the random workload, which includes numerous short requests. Both 12 and 19 are roughly 50 and 80 % of the 23°/s possible with the physical sensor. In this example, the speeds are higher than the vsensors' relative weights because of the proximity of requests. 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. We use the lookahead state restoration approach, request groups, and AFQ.

Continuous monitoring for the camera, continuously pan in increments of 65°—nearly one-fifth of the possible pan range—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.

Object tracking for the camera, periodically track a predefined 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 a fixed point and repeatedly capture images at a regular interval, while for two radars, scan a 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/s with an average inter-image interval of 0.09 s. 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/s with an average inter-image interval of 3.35 s. Similarly, two fixed-point sensing applications—at a distance of 180°—are both able to capture 0.2 images/s with an average inter-image interval of 4.65 s. With the radar, fixed-point sensing with a dedicated sensor is able to scan the same 30° sector every 0.5 s, but executing a random workload of 30° scans reduces the rate to every 1.5 s. We use these sensing rates for comparison in our case study below.

We first execute both continuous monitoring (Figs. 20a, 21a) and object tracking (Figs. 20b, 21b) concurrently with the fixed-point sensing application for both the camera and the radar. In both cases, we maintain a weight of one for

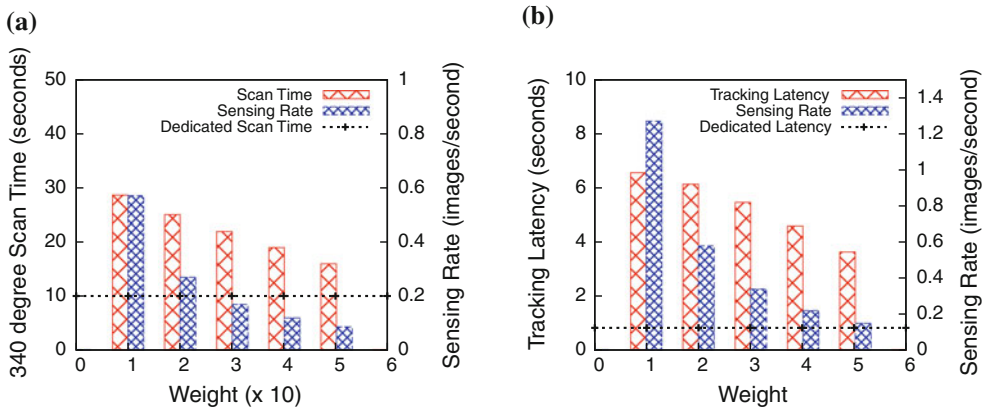


Fig. 20 For the camera, MultiSense is able to serve concurrent sensing applications. A continuous monitoring application (a) and an object tracking application (b) both maintain tolerable performance for varying weight assignments, while competing with a fixed-point sensing application with weight one. **a** Continuous monitoring and **b** object tracking

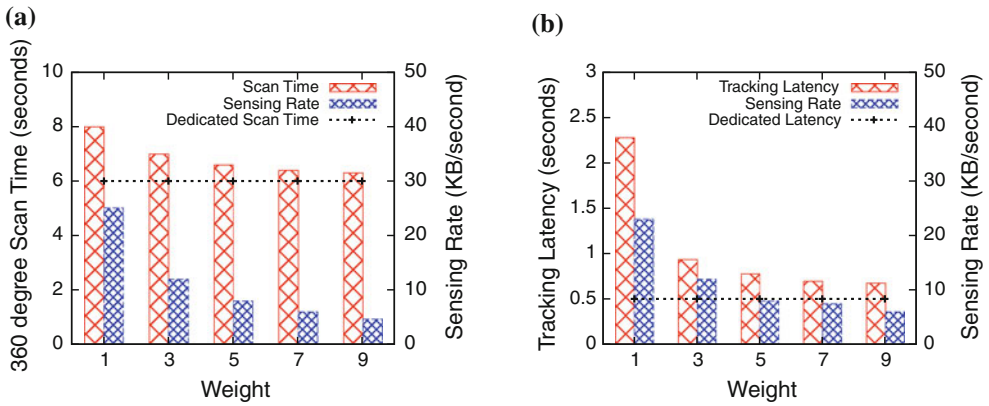


Fig. 21 For the radar, MultiSense is able to serve concurrent sensing applications. A continuous monitoring application (a) and an object tracking application (b) both maintain tolerable performance for varying weight assignments, while competing with a fixed-point sensing application with weight one. **a** Continuous monitoring and **b** object tracking

fixed-point sensing, while varying the weights assigned to continuous monitoring and object tracking. Figure 20 shows the results for the camera and Fig. 21 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 1.5 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 s. Thus, in the real-world, the monitoring application is able to capture five distinct points 113 ft apart, e.g., four doorways, at distance of 100 ft from the camera every 4 s.[2] Simultaneously, fixed-point sensing maintains an average sensing rate of nearly 0.2 images/s, 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 s, which is suitable for tracking a moving object at a distance of 300 ft moving at 2.66 mph, e.g., a person walking, for up to 1,779 ft (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.[3] During tracking, the fixed-point sensing application maintains a sensing rate of 0.3 images/s.

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

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

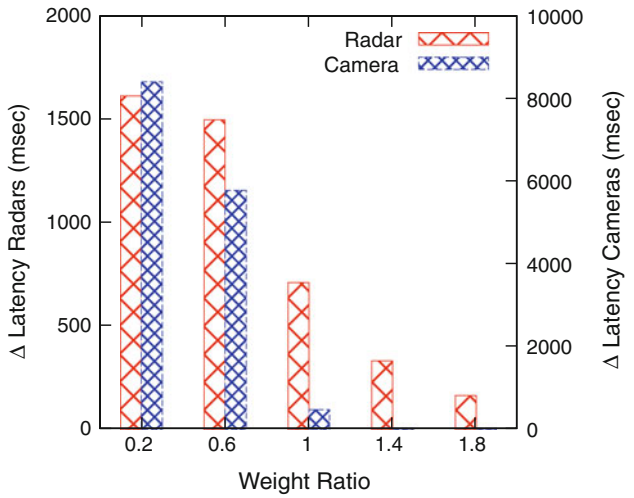


Fig. 22 The difference in latency when coordinating multiple vsensors on different nodes to sense the same point

Now consider continuous monitoring for the radar. With a 1:5 ratio, the radar is able to complete a 360° scan in about 7 s, while simultaneously scanning the same 30° sector every 11 s. 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 s. Since the radar we emulate has a range of 25 miles, we are able to capture the storm's movement every 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.

Since MultiSense borrows techniques from proportional-share scheduling, it was not designed for synchronizing actuation and sensing within tight time-scales, e.g., millisecond-level, between multiple vsensors at different physical sensors. We ran experiments to quantify appropriate time-scales for synchronizing multiple applications that use vsensors on different physical sensors. The experiment uses 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's results demonstrate the extent to which MultiSense satisfies timeliness requirements. Figure 22 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 s) simultaneous sensing. Even with a low relative weight assignment the sensors sense the same point within 2 s of each other, which is suitable for a range of scenarios, such as estimating three-dimensional wind direction for radars [30] or 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 [15], since motes are generally not powerful enough to execute multiple applications concurrently. While some recent mote-class OSes incorporate threads and time-sharing [8], the energy constraints of motes prevent them from using high-power sensors with rich programmable actuators, such as PTZ cameras or steerable weather radars. PixieOS [17] 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 [14]; 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 [31]. Other frameworks, including VMedia [24], 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 [32]. 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 [29], 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 [21].

MultiSense applies the proportional-share paradigm [11], which has been well studied in other contexts, to multiplex control of steerable sensors. Start-time fair queuing was originally prototyped for multiplexing packet streams and later extended to CPUs [11]. More recently, there has been work on proportional-share scheduling for energy—another non-traditional resource—using virtual batteries [9]. 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 [7] 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 [27], 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].

Finally, control strategies for both single PTZ cameras [20, 22, 23, 28] and collections of PTZ cameras [4, 5] have been well studied in the computer vision community. However, prior work focuses primarily on controlling cameras, or networks of cameras, for a single task, e.g., acquiring head imagery [5], rather than multiplexing cameras across multiple tasks, as in MultiSense. Additionally, for MultiSense, we evaluate a PTZ camera's performance when multiplexing, e.g., how fast it can track multiple objects, and do not address the important computer vision problem of recognizing objects in video streams, which is necessary for some tasks, e.g., tracking a person. We view multiplexing a camera, and evaluating its performance for multiple types of common tasks as a orthogonal, but complementary, to the recognition problem. While many of the problems studied in the computer vision community for PTZ cameras may also apply to steerable radars, we are not aware of any work in the area.

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 that their deployment costs are high. 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 two sensors.

Acknowledgments We would like to thank the anonymous reviewers for their insightful comments that improved this paper. This work was supported in part by NSF grants CNS-1117221, OCI-1032765, CNS-0916577, and CNS-0855128.

References

1. Netflix Watch Instantly. http://www.netflix.com/ (2012). Accessed Jan 2012
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T.L., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the 9th ACM symposium on operating systems principles, Bolton Landing, 19–22 Oct 2003
3. Bennett, J.C.R., Zhang, H.: Wf2q: worst-case weighted fair queuing. In: Proceedings of the IEEE international conference on computer communications, Shanghai, June 2002
4. Bi, S., Chong, D., Kamal, A.T., Farrell, J.A., Roy-chowdhury, A.K.: Distributed camera networks. IEEE Signal Process. Mag. **28**(3), 20–31 (2011)
5. Bimbo, A.D., Dini, F., Grifoni, A., Pernici, F.: Uncalibrated framework for on-line camera cooperation to acquire human head imagery in wide areas. In: Proceeding of the fifth IEEE international conference on advanced video and signal based surveillance, Santa Fe, Sep 2008
6. Binsted, K., Bradley, N., Buie, M., Ibara, S., Kadooka, M., Shirae, D.: 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, Aug 2005
7. Bruno, J., Brustoloni, J., Gabber, E., Ozden, B., Silberschatz, A.: Disk scheduling with quality of service guarantees. In: Proceedings of the international conference on multimedia computing and systems, vol. 2, p. 400, Florence, July 1999

8. Cao, Q., Abdelzaher, T., Stankovic, J., He, T.: The LiteOS operating system: towards unix-like abstractions for wireless sensor networks. In: Proceedings of the 7th international conference on information processing in sensor networks, pp. 233–244, St. Louis, Apr 2008
9. Cao, Q., Fesehaye, D., Pham, N., Sarwar, Y., Abdelzaher, T.: Virtual battery: an energy reserve abstraction for embedded sensor networks. In: Proceedings of the real-time systems symposium, San Diego, Nov 2008
10. Francoeur, A.: Border patrol goes high tech. http://www.photonics.com (2009). Accessed on 24 Aug 2009
11. Goyal, P., Vin, H., Cheng, H.: Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In: Proceedings of ACM SIGCOMM conference, Stanford, Aug 1996
12. Iyer, S., Druschel, P.: Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In: Proceedings of the 18th ACM symposium on operating systems principles, Banff, Oct 2001
13. Jones, M., Rosu, D., Rosu, M.: CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In: Proceedings of the symposium on operating systems principles, Saint-Malo, Oct 1997
14. Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D. and Philip Levis.: Integrating concurrency control and energy management in device drivers. In: Proceedings of the symposium on operating systems principles, Stevenson, Oct 2007
15. Levis, P., Maté, D. Culler.: A tiny virtual machine for sensor networks. In: Proceedings of the international conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
16. Li, M., Yan, T., Ganesan, D., Lyons, E., Shenoy, P., Venkataramani, A., Zink, M.: Multi-user data sharing In radar sensor networks. In: Proceedings of the ACM conference on embedded networked sensor systems, Sydney, Nov 2007
17. Lorincz, K., Chen, B., Waterman, J., Werner-Allen, G., Welsh, M.: Resource aware programming in the Pixie operating system. In: Proceedings of the ACM conference on embedded networked sensor systems, Raleigh, Nov 2008
18. Magnuson, S.: New northern border camera system to avoid past pitfalls. In: National defense magazine, Sep 2009
19. McLaughlin, D., Pepyne, D., Chandrasekar, V., Philips, B., Kurose, J., Zink, M.: Short-wavelength technology and the potential for distributed networks of small radar systems. Bull. Am. Meteorol. Soc. **90**(12), 1797–1817 (2009)
20. Micheloni, C., Rinner, B., Foresti, G.L.: Video analysis in pan-tilt-zoom camera networks. IEEE Signal Process. Mag. **27**(5), 78–90 (2010)
21. Nelson, T.: The device driver as state machine. C. Users J. **10**(3), 41–60 (1992)
22. Qureshi, F.Z., Terzopoulos, D.: Surveillance camera acheduling: a virtual vision approach. ACM Multimed. Syst. J. **12**(3), 269–283 (2006)
23. Qureshi, F.Z., Terzopoulos, D.: Planning ahead for PTZ camera assignment and control. In: Proceedings of third ACM/IEEE international conference on distributed smart cameras, Como, Aug 2009
24. Raj, H., Seshasayee, B., Schwan, K.: VMedia: enhanced multimedia services in virtualized systems. In: Proceedings of the multimedia computing and networks conference, San Jose, Jan 2008
25. Salazar, J.L., Knapp, E.J., McLaughlin, D.J.: Dual-polarization performance of the phase-tilt antenna array in a CASA dense network radar. In: Proceedings of the 2010 IEEE international geoscience and remote sensing symposium, Honolulu, July 2010
26. Sharma, N., Irwin, D., Shenoy, P., Zink, M.: MultiSense: fine-grained multiplexing for steerable camera sensor networks. In: Proceedings of the 2011 ACM multimedia systems, San Jose, Feb 2011
27. Shenoy, P., Vin, H.: Cello: a disk scheduling framework for next generation operating systems. In: Proceedings of the international conference on measurement and modeling of computer systems, Madison, June 1998
28. Starzyk, W., Qureshi, F.Z.: Learning proactive control strategies for PTZ cameras. In: Proceedings of the 2011 fifth ACM/IEEE international conference on distributed smart cameras, Ghent, Aug 2011
29. Swift, M.M., Annamalai, M., Bershad, B.N., Levy, H.M.: Recovering device drivers. In: Proceedings of the sixth symposium on operating system design and implementation, San Francisco, Dec 2004
30. Wang, Y., Chandrasekar, V., Dolan, B.: Development of scan strategy for dual doppler retrieval in a networked radar system. In: Proceeding of the IEEE international geoscience and remote sensing symposium, Boston, July 2008
31. Warfield, A., Hand, S., Fraser, K., Deegan, T.: Facilitating the development of soft devices. In: Proceedings of the USENIX annual technical conference, Anaheim, 10–15 Apr 2005
32. Xia, L., Lange, J.: Towards virtual passthrough I/O on commodity devices. In: Proceedings of the workshop on I/O virtualization, USENIX Association, San Diego, Dec 2008
33. Zink, M., Lyons, E., Westbrook, D., Kurose, J., Pepyne, D.: Closed-loop architecture for distributed collaborative adaptive sensing of the atmosphere: meteorological command and control. Int. J. Sens. Netw. **7**(1/2), 4–18 (2010)