Odessa: Enabling Interactive Perception Applications on Mobile Devices [*]

Moo-Ryong Ra[*], Anmol Sheth[†], Lily Mummert[†], Padmanabhan Pillai[†],
David Wetherall[†‡] and Ramesh Govindan[*]

University of Southern California[*] Intel Labs[†] University of Washington[‡]

ABSTRACT

Resource constrained mobile devices need to leverage computation on nearby servers to run responsive applications that recognize objects, people, or gestures from real-time video. The two key questions that impact performance are what computation to offload, and how to structure the parallelism across the mobile device and server. To answer these questions, we develop and evaluate three interactive perceptual applications. We find that offloading and parallelism choices should be dynamic, even for a given application, as performance depends on scene complexity as well as environmental factors such as the network and device capabilities. To this end we develop *Odessa*, a novel, lightweight, runtime that automatically and adaptively makes offloading and parallelism decisions for mobile interactive perception applications. Our evaluation shows that the incremental greedy strategy of *Odessa* converges to an operating point that is close to an ideal offline partitioning. It provides more than a 3x improvement in application performance over partitioning suggested by domain experts. *Odessa* works well across a variety of execution environments, and is agile to changes in the network, device and application inputs.

Categories and Subject Descriptors

D.4.7 [**Software**]: Operating System—*Organization and Design: Interactive Systems*

General Terms

Design, Experiment, Measurement, Performance

Keywords

Offloading, Parallel Processing, Mobile Perception Application, Video Processing, Incremental Partitioning

1. INTRODUCTION

As the processing, communication and sensing capabilities of mobile devices increase, a new class of mobile *interactive perception* applications is emerging. These applications use cameras and other high-data rate sensors to perform perception tasks, like face or object recognition, and enable natural human-machine interfaces and interactive augmented-reality experiences on mobile devices [16, 21]. For example, face recognition could be used by a social networking application that recognizes people as the user sweeps the camera across a crowded room; a gesture recognition based natural user interface could be used to control a media application running on the mobile device; and object and pose recognition can be used by an augmented reality shopping application that overlays information about an object in the user's hand.

Interactive perception applications have a unique set of requirements that stress the capabilities of mobile devices. First, interactive applications require crisp response. For example, to feel responsive, an augmented reality application would need to display results well under a second. Second, these applications require continuous processing of high data rate sensors such as cameras to maintain accuracy. For example, a low frame rate may miss intermediate object poses or human gestures. Third, the computer vision and machine learning algorithms used to process this data are compute intensive. For example, in one of the applications we study, extracting features from an image can take 7 seconds on a netbook. Finally, the performance of these algorithms is highly variable and depends on the content of the data, which can vary greatly.

These requirements cannot be satisfied on today's mobile devices alone. Even though the computing and communication capabilities of these platforms are improving, interactive perception applications will continue to push platform limits as new, more accurate but more compute-intensive algorithms are developed. However, two techniques can help make mobile interactive perception a reality: offloading one or more of the compute-intensive application components to an Internet-connected server, and using parallelism on multi-core systems to improve responsiveness and accuracy of the applications. Fortunately, interactive perception applications can be structured for offloading, and provide considerable opportunities to exploit parallel processing. In this paper, we describe

[*] This research was sponsored by the USC/CSULB METRANS Transportation Center and by the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the METRANS center, the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. In addition, the first author, Moo-Ryong Ra, was supported by Annenberg Graduate Fellowship.

MobiSys'11, June 28–July 1, 2011, Bethesda, Maryland, USA.

[1] Author contact: mra@usc.edu, anmolsheth@gmail.com,
lily@cs.cmu.edu, padmanabhan.s.pillai@intel.com,
djw@cs.washington.edu, ramesh@usc.edu

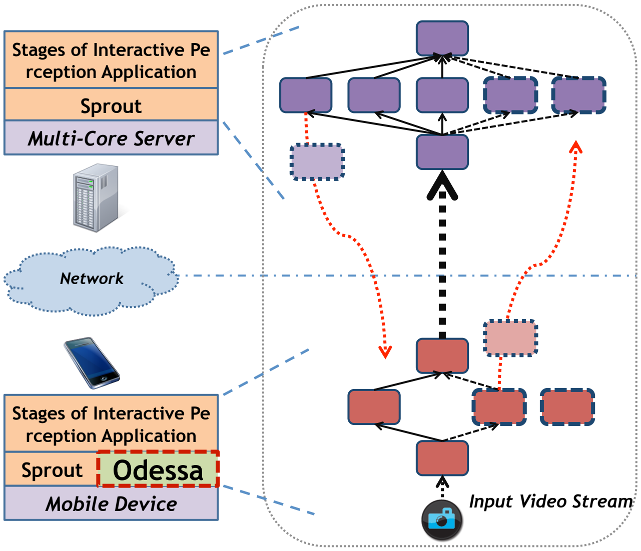

Stages of Interactive Pe rception Application
Sprout
Multi-Core Server
Network
Stages of Interactive Pe rception Application
Sprout
Odessa
Mobile Device
Input Video Stream

Figure 1: Overview of the *Odessa* runtime system.

a runtime called *Odessa* that automatically and adaptively determines how best to use these techniques.

This paper makes three contributions. First, it provides an understanding of the factors which contribute to the offloading and parallelism decisions. We show through extensive experiments (Section 3) on three interactive perception applications that neither offloading decisions nor the level of data or pipeline parallelism can be determined statically and must be adapted at runtime. This is because both responsiveness and accuracy change dramatically with input variability, network bandwidth, and device characteristics at runtime. Our second contribution is the design of *Odessa* (Figure 1, Section 4), a lightweight, adaptive runtime for mobile interactive perception applications. To our knowledge, Odessa is the first work to explore the simultaneous adaptation of offloading and level of parallelism with the goal of jointly improving responsiveness and accuracy. The key insight of our work is that the dynamics and parallelism requirements of interactive perception applications preclude prior approaches that use offline profiling and optimization based partitioning [8, 9]; instead, a simpler greedy and incremental approach delivers good performance. Finally, we provide experimental results (Section 5) on an implementation of *Odessa* that show more than 3x improvement in performance compared to a configuration by a domain expert, and comparable performance to an idealized offline configuration computation that assumes infinite server resources and complete offline performance profiling. Our results also show that *Odessa* works well across a variety of execution environments, and is agile to changes in the network, device and application inputs.

Odessa is qualitatively different from prior work that uses networked computing infrastructure to enhance the capabilities of mobile devices. It is complementary to work on using offloading for conserving energy on the mobile device (e.g., MAUI [9]). Moreover, it does not require prior information on application performance [9, 23] or a set of feasible candidate partitions [3, 8] to make offloading decisions.

2. BACKGROUND

In this section, we describe the metrics and methods related to adaptive offloading and parallelism, we describe the set of interactive perception applications studied, and then discuss Sprout, a distributed programming framework on which our system is based.

Application	# of Stages	Avg. Makespan & Frame Rate
Face Recognition	9	2.09 s, 2.50 fps
Object and Pose Recognition	15	15.8 s, 0.09 fps
Gesture Recognition	17	2.54 s, 0.42 fps

Table 1: Table summarizes the data flow graph of the three computer vision applications along with average makespan and frame rate measured when the application is running locally on netbook platform.

2.1 Metrics and Methods for Adaptation

Two measures of goodness characterize the responsiveness and accuracy requirements of interactive perception applications.

Makespan is the time taken to execute all stages of a data flow graph for a single frame. The makespan is a measure of the responsiveness of the application: a low makespan ensures fast completion of a recognition task and thereby improves user satisfaction. **Throughput** is the rate at which frames are processed and is a measure of the accuracy of the application. A low frame rate may miss intermediate object poses or human gestures. Any runtime system for interactive perception must simultaneously strive to minimize makespan and maximize throughput. In general, the lower the makespan and the higher the throughput the better, but the applications can become unusable at makespans over a second, or throughput under 5 fps.

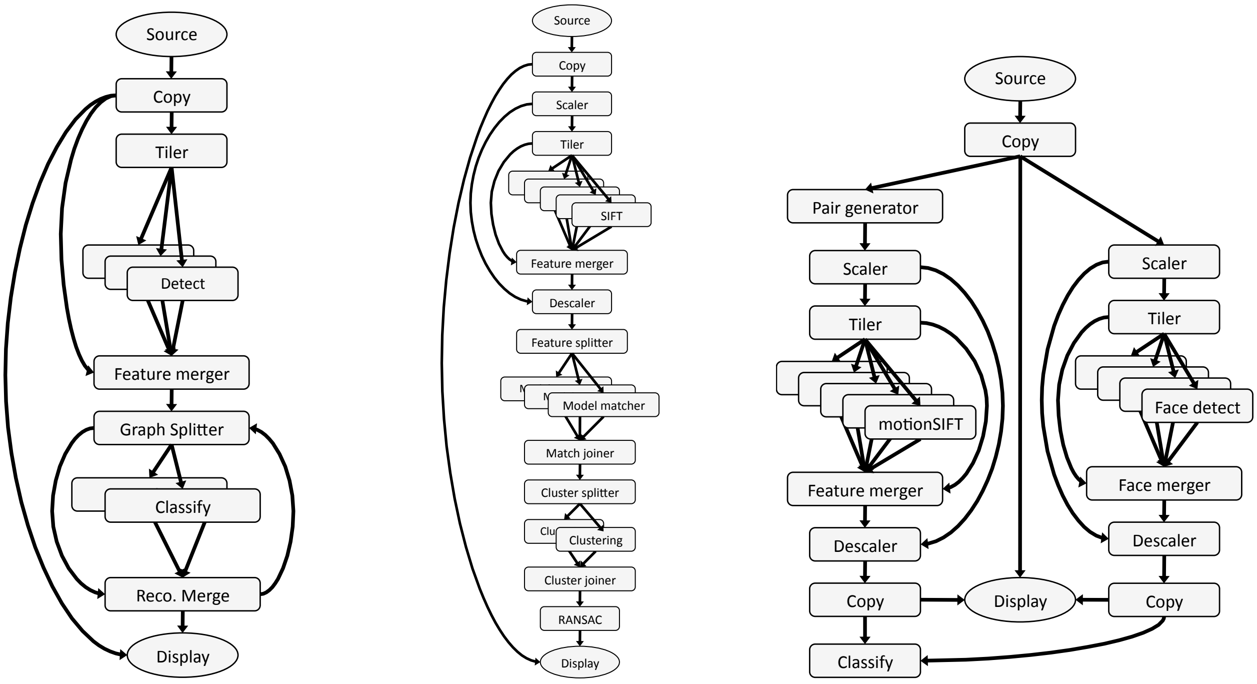
In adapting interactive perception applications on mobile, three techniques can help improve makespan and throughput. *Offloading* moves the most computationally-intensive stages onto the server in order to reduce makespan. *Pipelining* allows different stages of the application (whether running on the mobile device or the server) to process different frames in parallel, thereby increasing throughput. Increasing *data-parallelism*, in which frames are split into multiple sub-frames that are then processed in parallel (either on a multi-core mobile device or a server or cluster), can reduce the makespan by reducing the computation time of a stage. Data and pipeline parallelism provide great flexibility in the degree to which they are used. These techniques are not mutually exclusive: pipelining is possible even when some stages are offloaded, and data-parallel execution is possible on offloaded stages, etc.

The goal of *Odessa* is to decide when and to what degree to apply these techniques to improve both the makespan and throughput of interactive perception applications.

2.2 Interactive Perception Applications

We use three qualitatively different interactive perception applications described below, both to motivate the problem and to evaluate the efficacy of our solutions. Often computer-vision based applications are naturally described using a data-flow model. Figure 2 describes the data-flow graphs for the three applications, as implemented on *Odessa* and Sprout.

Face Recognition. Figure 2(a) shows the application graph for face recognition. The application consists of two main logical blocks consisting of face detector and the classifier. Face detection is done using the default OpenCV [4] Haar Classifier. The face classifier takes as input the detected faces and runs an online semi-supervised



(a) Face Recognition

(b) Object and Pose Recognition

(c) Gesture Recognition

Figure 2: Data flow graph for the three computer vision applications.

learning algorithm [17] to recognize the faces from a data set of 10 people.

Object and Pose Recognition Figure 2(b) shows the data-flow graph for the object instance and pose recognition application [26]. The application consists of four main logical blocks. As shown in the figure, each image first passes through a proportional downscaler. SIFT features [20] are then extracted from the image, and matched against a set of previously constructed 3D models for the objects of interest. The features for each object are then clustered by position to separate distinct instances. A random sample consensus (RANSAC) algorithm with a non-linear optimization is used to recognize each instance and estimate its 6D pose.

Gesture Recognition. Figure 2(c) shows the application graph for a gesture recognition application. Each video frame is sent to two separate tasks, face detection and motion extraction. The latter accumulates frame pairs, and then extracts SIFT-like features that encode optical flow in addition to appearance [6]. These features, filtered by the positions of detected faces, are aggregated over a window of frames using a previously-generated codebook to create a histogram of occurrence frequencies. The histogram is treated as an input vector to a set of support vector machines trained for the control gestures.

2.3 Sprout: A Parallel Processing Framework

Odessa is built on Sprout [25], a distributed stream processing system designed to make developing and executing parallel applications as easy as possible by harnessing the compute power of commodity multi-core machines. Unlike programming frameworks for parallelizing offline analysis of large data sets (MapReduce [10] and Dryad [15]), Sprout is designed to support continuous, online processing of high rate streaming data. Sprout's abstractions and runtime mechanisms, described below, are well-suited to support the *Odessa* runtime system.

Programming Model. Applications in the Sprout framework are structured as a data flow graphs; the data flow model is particularly well suited for media processing applications that perform a series of operations to an input video or audio stream. The vertices of the graph are processing steps called *stages* and the edges are *connectors* which represent the data dependencies between the stages. Stages within an application employ a shared-nothing model: they share no state, and interact only through connectors. This restriction keeps the programming complexity of individual stages comparable to that of sequential programming, and allows concurrency to be managed by the Sprout runtime. This programming model allows programmers to express coarse-grained application parallelism while hiding much of the complexity of parallel and distributed programming from the application developer.

Automated data transfer. Sprout connectors define data dependencies and perform data transfer between processing stages. The underlying implementation of a connector depends on the location of the stage endpoints. If the connected stages are running in the same process, the connector is implemented as an in-memory queue. Otherwise, a TCP connection is used. The Sprout runtime determines the connector type, and handles serialization and data transport through connectors transparently. This allows a processing stage to be written in a way that is agnostic to whether it or related processing steps have been off-loaded.

Parallelism Support. The Sprout runtime supports coarse-grained data parallelism and pipeline parallelism. Data parallelism is supported by having multiple instances of a stage execute in parallel on separate processor cores. Pipeline parallelism is supported by having multiple frames be processed simultaneously by the different processing stages of the application.

The relationship between Sprout and *Odessa*. Sprout provides programmers with *mechanisms* to dynamically adjust running applications, change the degree of parallelism, and migrate processing stages between machines. In *Odessa*, we develop adaptive techniques for determining when and which stages to offload, and deciding how much pipelining and data-parallelism is necessary in order to achieve low makespan and high throughput, and then leverage the Sprout mechanisms to effect the changes.

3. FACTORS AFFECTING APPLICATION PERFORMANCE

In this section we present experimental results that highlight how the performance of interactive perception applications is impacted by multiple factors. We find that:

- Input variability, network bandwidth, and device characteristics can impact offloading decisions significantly, so such decisions must be made adaptively and cannot be statically determined at compile time.

- Once some stages have been offloaded, different choices for data-parallelism can lead to significantly different application performance, and the data-parallelism levels cannot be determined *a priori*.

- Finally, even with adaptive offloading and data-parallelism, a static choice of pipeline-parallelism can lead to suboptimal makespan or leave the pipeline underutilized.

3.1 Experimental Setup

Hardware. Our experimental setup consists of two different mobile devices and a server. The two mobile devices are a netbook with a single-core Atom processor (Intel N270 processor) running at 1.4 GHz with hyper-threading turned off and 0.5 MB of cache, and a dual-core laptop with each core running at 1.8 GHz (Intel T5500 processor) that does not support hyper-threading and 2 MB of cache. The netbook is a surrogate for a future-generation smartphone platform. To compare the two devices, we use the ratio of the sum of frequencies of all available CPU cores, which we call the *relative frequency ratio*. The relative frequency ratio between the two devices is 2.3x. The server is an eight core Intel Xeon processor with each core running at 2.13 GHz with 4 MB of cache. The relative frequency ratio between the server and the two mobile devices is 12.2x and 4.7x for the netbook and laptop respectively.

Input data set. To ensure repeatability across different experimental runs, the input data for each of the three applications is a sequence of frames captured offline in typical indoor lighting conditions at 30 fps at a resolution of 640x480 pixels per frame. The data set used for face recognition consists of 3000 frames in which 10 people each walk up to the camera and make different facial expressions. The input data set for pose detection consists of 500 frames in which the user is holding a book in one hand and manipulating the pose of the book while holding the camera pointed at the book with the other hand. The data for gesture-recognition consists of roughly 500 frames of a person sitting in front of the camera performing the different gestures.

Network configurations. We emulate different network conditions between the mobile device and networked server by varying the delay and bandwidth of the link using Dummynet [5]. The ten different emulated network conditions are summarized in Table 2.

Network	Configuration
LAN	100 Mbps
WAN 20 ms	30 Mbps, 20 ms RTT, Loss rate 0%
WAN 40 ms	30 Mbps, 40 ms RTT, Loss rate 0%
LAN 802.11g	25 Mbps, Loss rate 0%, 1%, 2%
LAN 802.11b	5.5 Mbps, Loss rate 0%, 1%, 2%
3G	0.5 Mbps, 500 ms RTT, Loss rate 0%

Table 2: The ten different network conditions emulated by Dummynet. The WAN bandwidths are symmetric and the RTT for the LAN configurations were under a millisecond.

Application profiler. A lightweight runtime profiler maintains the following application execution metrics: execution time of each stage, the amount of data that flows on the connector links and the network delay time. The profiler also keeps track of the latency (or makespan) and the frame rate of the application. We describe more details about the application profiler in Section 4.1.

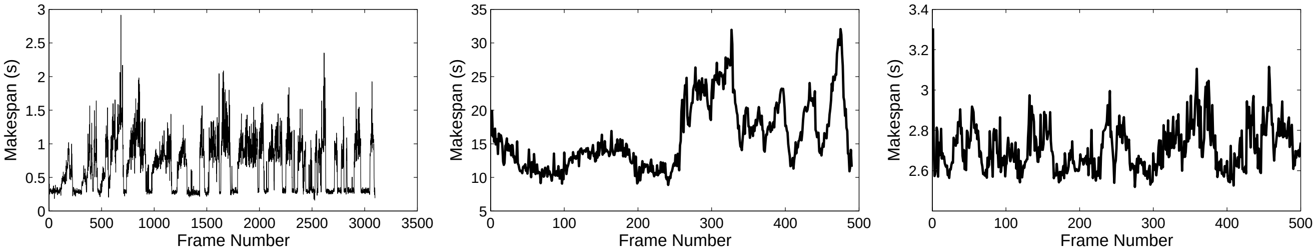
3.2 Input Data Variability

We begin by characterizing the impact of the scene content on the execution characteristics of the application. We run all three applications on the netbook and for each frame we plot the makespan (Figure 3) and the number of features generated (Figure 4). Our results show that input scene content can cause large and abrupt changes in the execution time of different stages, and different applications respond differently to scene variability. This, in turn, can impact the decision of whether to offload the stage or not.

Face Recognition. The sources of input variability are based on the presence of a face in the scene and the similarity between the test face and the other faces in the training data set. Figures 3(a) and 4(a) show the makespan for the application and number of faces detected per frame respectively. We use a single detection and classification stage for this experiment. When there are no faces detected in the scene, the makespan is primarily dominated by the face detection stage (0.29 s with little variability) with the face classifier stage incurring no overhead. However, when faces are detected, the makespan can increase by an order of magnitude, and have high variability with no clear difference between the processing time of frames containing one, two or three faces. For example, between frames 1000 and 1165 in Figure 3(a) the makespan is highly bursty and varies between 0.6 s to 1.4 s even when there is a single person in the scene.

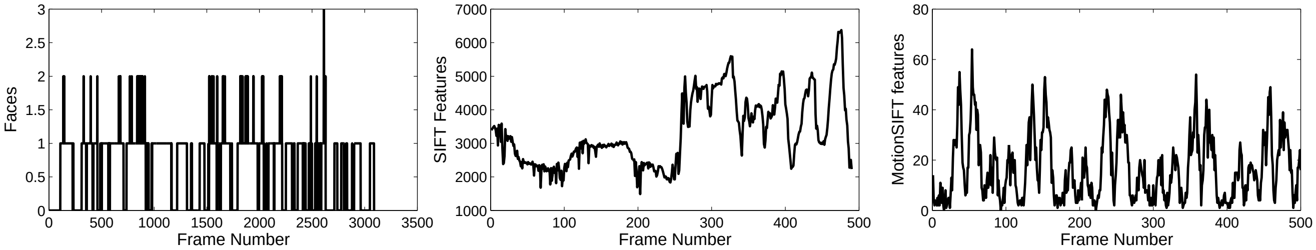
Object and Pose Recognition. The processing time for this application is primarily dominated by the feature extraction from the input scene. A complicated scene containing edges or texture will generate a large number of SIFT features, resulting in increased execution times for both the feature extraction and model matcher stages. Consequently, there is a strong correlation between the number of features detected and the makespan of the input frame as shown in Figures 3(b) and 4(b). Beyond frame number 250 both figures show sharp peaks and valleys. This is caused by the user altering the pose of the notebook that causes a peak when the feature-rich notebook is occupying most of the input scene and a sudden valley when the notebook pose is changed to occupy a small fraction of the input scene.

Gesture Recognition. The primary source of input scene variability is the extent of user motion in the scene. Figure 4(c) shows



(a) Face recognition (b) Object and pose recognition (c) Gesture recognition

Figure 3: Variation in the per frame makespan.



(a) Face recognition (b) Object and pose recognition (c) Gesture recognition

Figure 4: Variation in the number of features extracted per frame.

the number of motionSIFT features extracted per frame. The graph consists of a repeating pattern of two large peaks followed by multiple smaller peaks. The two large peaks are caused as the attention gesture requires the user to raise both hands in front of the camera and the drop them down. This is then followed by a one handed gesture that typically generates a smaller number of features compared to the attention gesture. The sharp drops in the graph are caused when the user is not moving in between gestures. Unlike the previous application, the makespan of the application shown in Figure 3(c) has a weak correlation with the input features. The weak correlation is caused by high variability in the face detection algorithm that is the next compute intensive stage in the application pipeline.

3.3 Variability Across Mobile Platforms

Another factor contributing to execution time variability, and hence to the offloading decision, is the contention between the different application threads for the compute resources (memory and CPU) available on the mobile platform. To explore the extent of impact on performance, we first compare the distribution of execution time of scene-independent stages of the application graph, then benchmark the aggregate performance of the application across both platforms.

Figure 5 shows the distribution in execution time across the two mobile platforms for three scene-independent stages: the image source stage for the face recognition application that reads an image from file to memory (Figure 5(a)), the frame copy stage for the object recognition application that makes copies of the image for multiple downstream stages (Figure 5(b)), and the image scaling stage for gesture recognition application that scales the image in memory (Figure 5(c)).

Ideally, the two devices should demonstrate the same distribution of completion time with a constant speedup factor of at least 2.3x. However, contention between other memory and compute intensive threads of the application graph cause a large variation in execution time on the netbook platform even though the median speedup ranges between 2.5-3.0x. The laptop with a dual-core processor and additional memory exhibits a much smaller variability due to the extra memory and isolated execution of the threads on the multiple processors. This effect, compounded across the different stages, leads to a significant difference in the aggregate performance of the application across the two platforms as shown in Table 3. Instead of the the expected 2.3x speedup, the speedup ranges between 2.9x for the face recognition application and 5.47x for the object and pose recognition application.

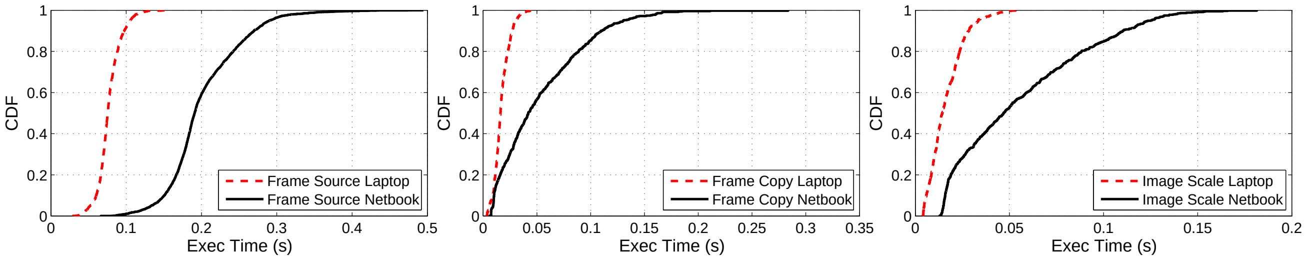
Application	Makespan (s) Laptop	Makespan (s) Netbook	Speedup
Face Recognition	0.078	0.20	2.94
Object and Pose Rec.	1.67	9.17	5.47
Gesture Recognition	0.54	2.34	4.31

Table 3: Median speedup in the overall application performance across the two devices.

3.4 Network Performance

Changes in network bandwidth, delay and the packet loss rate, between the mobile device and the server can each affect interactive performance and trigger stage offloading or parallelization decisions. We characterize the performance of the face recognition application for two different partitions across the different network settings described in Section 3.1. The first application partition runs only the source and display stages locally and offloads all other stages to the server, requiring the mobile device to upload the entire image to the server. The second application partition reduces the network overhead by additionally running the face detection stage locally, requiring the mobile device to upload only the pixels containing the detected faces to the server.

The tradeoff between the two partitions depends on the network



(a) Face recognition (b) Object and pose recognition (c) Gesture recognition

Figure 5: The figures show the variability in the completion time of three example stages running on the laptop and netbook device. These stages perform a fixed operation on the input image that is independent of the scene content.

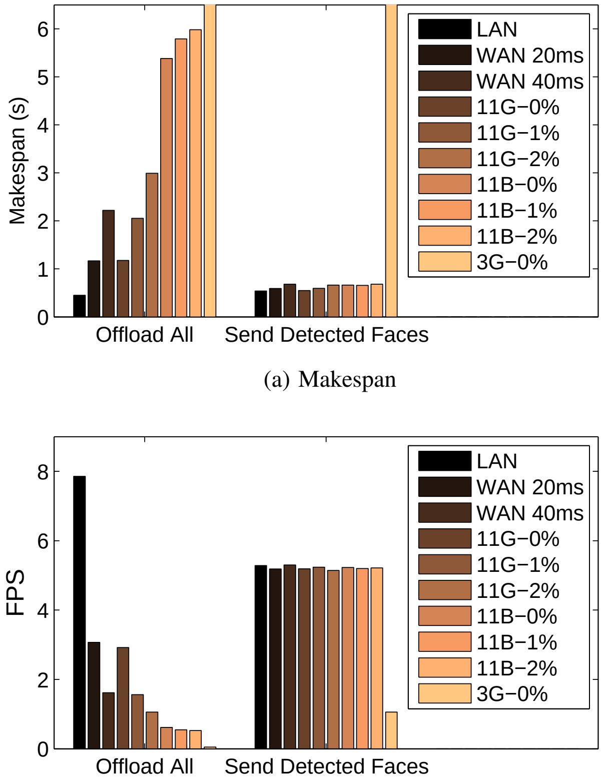
# of Threads	% Frames with faces	Mean exec. time (ms)
1	61.66	149.0
2	24.87	15.6
3	38.11	18.0

Table 4: The accuracy and mean execution time of face detection with increasing number of worker threads.

performance and the time taken to detect faces in the input frame on the mobile device. Transmitting the entire image incurs a fixed overhead of 921.6 KB of data per frame taking 73.7 ms over the LAN network configuration while transmitting only the detected faces requires transmitting between 31 bytes when no faces are detected and from 39.3 KB to 427.1 KB when one or more faces are detected in the frame.

Figure 6 shows the makespan and throughput of the application for the two application partitions. The large image transfer is very sensitive to even a small amount of delay and loss that significantly degrades application performance. Transmitting the entire image to a server on a network with a RTT of 40 ms degrades the frame rate from about 8 fps to 1.8 fps. Even a loss-less 802.11g link is bandwidth-bottlenecked and cannot support more than 3 fps.

Transmitting only the detected faces over the network makes the face detection stage which runs on the mobile device a bottleneck and requires an average of 189.2 ms to detect faces, limiting the maximum frame rate to 5.31 fps. Moreover, since the network is not heavily used this application partition is robust to delay and bandwidth bottlenecks and packet loss rate on the link. The performance of the application shows negligible degradation across the different network configurations, providing a frame rate of 5.3 fps and makespan of 680 ms. In the case of the 3G network, the large network delay (500 ms RTT) significantly degrades the application performance.



(a) Makespan

(b) Frame Rate

Figure 6: Impact of the network on the performance of the face recognition application.

3.5 Effects of Data-Parallelism

Offloading alone may not be sufficient to reduce makespan; in many mobile perception applications, it is possible to leverage multicore technology to obtain makespan reductions through data parallelism (processing tiled sub-images in parallel). However, as we show in this section, the performance improvement achieved by data-parallelism can be sub-linear because of scene complexity. Additionally, for some computer vision algorithms, extensive use of data-parallelism can degrade application fidelity. This argues for an adaptive determination of the degree of data-parallelism.

Table 4 shows an experiment in which the face detection stage is offloaded, and explores the impact of increasing the number of face detection threads. Increasing the number of detection threads reduces the execution time of the stage at the cost of degrading the fidelity of the face detection algorithm by 36.7% with two threads and 23.5% with three threads. This drop in fidelity is due to faces falling across image tile boundaries, which renders them undetectable by the Haar classifier based face detection algorithm. Furthermore, the reason accuracy increases by about 13% from two threads to three is because the chances that the center tile includes the face is higher for three tiles than splitting the image in the middle into two tiles.

Such degradation in the fidelity of the algorithm could be avoided by using a tiler algorithm that tiles the image for different scales of the input image or using a face detection algorithm that uses

# of Threads	Thread 1 (s)	Thread 2 (s)	Thread 3 (s)
1	1.203 (3323.7)	-	-
2	0.741 (2124.9)	0.465 (1132.6)	-
3	0.443 (1203.6)	0.505 (1543.4)	0.233 (473.0)

Table 5: Average execution time in seconds and number of SIFT features detected by each thread for the object and pose recognition application.

SIFT-like scale invariant features that can be combined across the multiple tiles. However, both of these approaches come at a cost of either increased algorithmic complexity or higher computational overhead.

Table 5 shows the impact of input scene content on the average execution time of the different SIFT feature generator threads along with the average number of features extracted by each thread for the object-recognition application. The reason the feature generator thread execution times vary across image tiles is that SIFT features are not evenly distributed in the image; the slowest thread becomes the bottleneck and causes sub-linear speedup. From the table we observe that the average speedup is limited to 1.6x and 2.3x instead of the expected 2x and 3x speedup respectively.

3.6 Effects of Pipeline Parallelism

A key challenge in exploiting pipeline parallelism is to maintain the balance between under-utilizing the pipeline that delivers low throughput and over-utilizing the pipeline that increases the latency of the application due to excessive waiting time. In this section, we show that, for a fixed off-loading strategy, different data-parallelism decisions can result in different optimal pipelining strategies. This argues that the degree of pipelining must track adaptations in offloading *and* data-parallelism, otherwise the pipeline can be significantly under-utilized or over-utilized.

In Sprout, the degree of pipelining is governed by a bound on the maximum number of *tokens*: each token corresponds to an image in some stage of the pipeline, so the maximum number of tokens governs the pipeline parallelism in Sprout. Sprout users or application developers must specify this maximum number.

To understand the impact of the degree of pipelining, we create two different configurations of the object and pose recognition application and measure the impact of increasing the number of tokens in the pipeline on the throughput (Figure 7) and the makespan (Figure 8) of the application. In both configurations, the SIFT feature generator and the model matcher have been offloaded, but the configurations differ in the degree of data-parallelism for the two stages. The configurations F2-M5 and F5-M2 denote the number of SIFT feature generator threads and number of model matcher threads used by the application running on the server with eight cores.

From Figure 7 we observe that for both configurations the throughput initially increases linearly until the pipeline is full and then levels off beyond four tokens at which point additional tokens generated end up waiting at the head of the pipeline. The throughput of the F2-M5 configuration is higher compared to the F5-M2 configuration as the F2-M5 configuration reduces the execution time of the model matcher stage that is the primary bottleneck of the application.

Figure 8 shows the makespan response to increasing number of tokens in the pipeline. For both the configurations we observe that over-utilizing the pipeline by increasing the number of tokens beyond four increases the time a frame waits at the head of the pipeline. Furthermore, comparing the rate of increase of the wait time across the two configurations, we find that the wait time for the F5-M2 configuration increases faster than the F2-M5 configuration. This is because the F5-M2 configuration has a longer compute time for the bottleneck stage, though the total execution time is comparable. This causes the makespan to rise sooner with fewer tokens and more steeply compared to the other configuration.

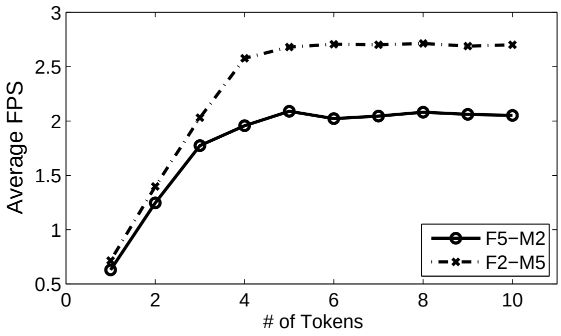


Figure 7: Frame rate with increasing number of tokens.

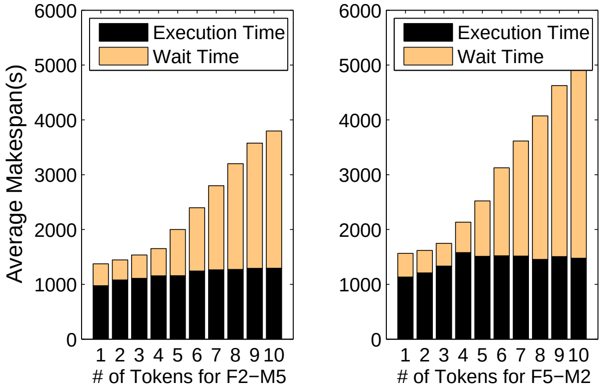


Figure 8: Makespan with increasing number of tokens.

4. DESIGN AND IMPLEMENTATION OF ODESSA

Motivated by the findings in the previous section, *Odessa* adaptively exploits pipelining, data-parallelism and offloading to improve performance and accuracy of these applications. The *Odessa* runtime runs on the mobile device; this enables *Odessa* to transparently improve application performance across different mobile platforms when the mobile is disconnected from the server.

The design the *Odessa* runtime has three goals, in order of decreasing importance:

- It must *simultaneously* achieve low makespan and high throughput in order to meet the needs of mobile interactive perception applications.
- It must *react quickly* to changes in input complexity, device capability, or network conditions. This goal ensures that transient changes in makespan or throughput are minimized or avoided.
- It must have *low computation and communication overhead.*

Prior approaches for offloading frame the problem using a discrete or graph optimization formulation [9, 13, 19]. For this approach to be effective, accurate estimates of stage execution time are required on both the mobile device and the server, which are often obtained by offline profiling. However, the results in Section 3 show that the execution time can vary significantly and cannot easily be modeled offline.

Odessa uses a greedy algorithm that periodically acquires information from a lightweight application profiler to estimate the bottleneck in the current configuration. Then, its decision engine uses simple predictors based on nominal processor frequencies, and a recent history of network measurements, to estimate whether offloading or increasing the level of parallelism of the bottleneck stage would improve performance. This *greedy* and *incremental* approach works very well to improve makespan and throughput, and incurs negligible overhead (as discussed in Section 5.1). Rarely, *Odessa*'s decision may need to be reversed because its estimators may be off, but it has a built-in self-correcting mechanism to maintain stability.

4.1 Lightweight Application Profiler

The primary function of the application profiler is to maintain the complete application performance profile and make this information available to the decision engine running on the mobile device without impacting the application performance. Our profiler does not require any additional programmer input, and accounts for cycles (Figure 2(a)) and parallel sub-tasks (Figure 2(c)) in the application graph. For each frame processed by the application graph, the profiler collects the following information: the execution time of each stage in the graph, the wait time on every connector, the volume of data transferred on each connector, and the transfer time across the network connector edges.

Odessa implements this by having each stage piggyback its runtime profile information along with the data and forward it to the downstream stages along each of the output edges. A downstream stage receiving this profile information appends its own profile information only if there are no cycles detected and continues forwarding the data. When a splitter is encountered the profile history to that point is replicated on each output, and is pruned later on the joiner stage. The head of the pipeline that receives the aggregated data forwards it to the decision engine. This piggybacking approach simplifies the design of the decision engine as it receives all the profile data for the most recently concluded frame execution in-order and over a single RPC call. The overhead of looking for cycles and removing redundant data requires a simple linear scan of the profile data and incurs negligible overhead. Since each stage eliminates redundant profile information, the decision engine can easily compute the makespan and throughput of the application.

4.2 Decision Engine

The functionality of the decision engine is split across two threads running on the mobile device. The first thread manages the data parallelism and stage offloading. The second thread manages the pipeline parallelism by dynamically controlling admission into the pipeline. Both of these threads make use of the application profile data to make their decisions. The data received from the profiler is maintained in a heap sorted by the slowest graph element (stage or connector) that facilitates efficient lookup.

4.2.1 Adapting Data Parallelism and Stage Offloading

The algorithm in Figure 9 describes how *Odessa* adapts data parallelism and stage offloading. The algorithm runs periodically (every 1 second in our implementation) and in each iteration *greedily* selects the current bottleneck stage of the application pipeline and decides to make an *incremental* improvement by either changing the placement of a stage, increasing its data parallelism or doing nothing if the performance cannot be further improved. If the bottleneck is a compute stage, the algorithm picks between offloading the stage or increasing the degree of data-parallelism for the stage. If the bottleneck is a network edge, the algorithm estimates the cost of moving either the source or destination stage of the network edge.

```
begin
  $bottleneck$ := pick the first entry from the priority heap.
  if bottleneck is a compute stage
      a. estimate the cost of offloading the stage
      b. estimate the cost of spawning more workers
  elsif bottleneck is a network edge
      a. estimate the cost of offloading the source stage.
      b. estimate the cost of offloading the destination stage.
  fi
   c. take the best choice among a., b., or do-nothing.
   d. sleep ( decision granularity );
end
```

Figure 9: Algorithm for adaptive offloading and data-parallelism

If *Odessa* decides to offload or change data-parallelism, it signals the pipeline admission controller to stop issuing tokens. When the in-flight tokens have been drained, *Odessa* invokes Sprout's stage migration or thread spawning mechanisms (as appropriate), then resumes the pipeline once these have finished.

The effectiveness of the algorithms rests on the ability to estimate the impact of offloading the stage or increasing data parallelism. While it is difficult to accurately estimate the impact of the decision on the performance of the application, *Odessa* uses simple cost estimation techniques that are guided by the runtime application profiler.

Estimating cost of data parallelism. *Odessa* uses a simple linear estimation based cost metric to evaluate the execution time for increasing or decreasing the data parallelism. The linear estimate is based on the following simple equation: $E_{i+1} = \frac{N}{N+1}E_i$, where E_i is current execution time on ith frame and N is a current degree of data parallelism. This assumes linear speedup and uses the runtime profiler data to estimate the gains. To avoid the unbounded increase in the level of data parallelism, *Odessa* dampens the performance improvement estimated by the linear equation after data parallelism has reached a preset threshold. This threshold is set to twice the number of CPU cores for hyper-threaded CPU architectures and the scaling factor is $\frac{N+1}{Thresh}$ for $N > Thresh$. As we have discussed in Section 3.5, data parallelism may not always give linear speedup because of image complexity: if that is the case, *Odessa* will re-examine the decision in the next interval (see below).

Estimating cost of offloading a stage. Moving a compute stage from the mobile device to the server or vice versa should account for change in the execution time of the stage as well as the impact of the change on the network edges. Consider a stage **X** with a single input connector and a single output connector. Suppose **X** and its predecessor and successor stages are running on the mobile. Now to estimate the latency that would result from offloading **X** to the server, we need to compute (a) the reduced processing time of **X** on the server, and (b) the increased latency induced by having its input and output connectors traverse the network between the mobile and the server. *Odessa* estimates the reduced processing time as $T_m(X)\frac{F_m}{F_s}$, where $T_m(X)$ is the execution time for **X** on the mobile, obtained from the application profiler (averaged over the last 10 frames), and F_m and F_s are, respectively, the nominal processor frequencies on the mobile device and the server, respectively. Since *Odessa* also keeps track of the amount of data transmitted on every connector and has an online estimate of the current network bandwidth, it can estimate the increased latency of data transfer on the new network connectors. Combining all of this information, *Odessa* can estimate the effective throughput and makespan that would result from the offloading.

Our current implementation only acts when the resulting estimate would improve *both* makespan and throughput. We are conservative in this regard; a more aggressive choice where an action is taken if either metric is improved may be more susceptible to errors in the estimates. That said, because *Odessa* relies on estimates, an offloading or data-parallelism decision may need to be reversed at the next decision point and can result in a stage bouncing between the mobile and the server. If this occurs (and it has occurred only once in all our experiments), *Odessa* temporarily *pins* the stage in question until its compute time increases by a significant fraction (10%, in our implementation).

4.2.2 Adapting Pipeline Parallelism

A pipeline admission controller dynamically maintains the optimal number of frames in the pipeline to increase frame rate while adapting to the variability in the application performance. The admission controller is implemented as a simple token generator that issues tokens to the frame source stage; upon receiving a token, the frame source stage reads an input image. When an image is completely processed, its token is returned to the admission controller. The admission controller ensures that no more than T tokens are outstanding at any given instant, and T is determined by the following simple equation $T = \lceil (\frac{M}{B}) \rceil$ where M is the makespan and B is the execution time of the slowest stage, both averaged over the 10 most recent frames. Note that the token generator trades off higher throughput for a slightly higher makespan by using the ceiling function that may lead to one extra token in the pipeline.

5. EVALUATION

In this section we present an experimental evaluation of *Odessa*. Our evaluation methodology uses the setup and data described in Section 3.1 and addresses the following questions.

- How does *Odessa* perform on our data sets, and what is the overhead of the runtime profiler and the decision engine?
- How does *Odessa* compare with against other candidate strategies that either use domain knowledge for static stage placement and parallelism, or that use global profiling for optimal placement?
- How does *Odessa* adapt to changes in network bandwidth availability or the availability of computing resources?

5.1 Odessa's Performance and Overhead

We begin by examining the performance of *Odessa*'s decisions for our applications, when run both on the netbook and the laptop. In these experiments (and those described in the next subsection), we used a 100 Mbps network between the mobile device and the server; in a subsequent subsection, we evaluate the impact of constraining the network. For the face recognition application, all stages except the classifier stage begin executing on the mobile (this stage has a large database for matching); for the other applications, all stages begin executing on the mobile device. Figure 10 shows the timeline of the different decisions made by *Odessa* on the netbook (graphs for the laptop are qualitatively similar and are omitted for lack of space), with the circle indicating an offload decision, and a cross indicating the spawning of a thread (increased data parallelism). Table 6 shows the final stage configurations of the applications on both the laptop and the netbook.

Application	Stages Offloaded and Instances	
	Netbook	Laptop
Face Recognition	Face detection(2) - **3.39**	Nothing - **3.99**
Object and Pose Recognition	Object model matching(3), Feature generating(10) - **5.71**	Object model matching(3), Feature generating(10) - **5.14**
Gesture Recognition	Face detection(1), extracting Motion SIFT features(4) - **3.06**	Face detection(1), extracting Motion SIFT features(9) - **5.14**

Table 6: The table shows the stages that were offloaded to the server and the number of instances of each stage offloaded on the server by *Odessa*. The average degree of pipeline parallelism is shown in boldface.

Face Recognition. *Odessa* first chooses to offload the detection thread at frame number 110 and spawns an additional detection thread immediately. After this change in configuration, the frame source stage that performs the JPEG decompression of the stored images is the bottleneck, which *Odessa* cannot offload: it converges to a makespan of around 500ms and a throughput of 5 fps on the netbook. Interestingly, *Odessa* demonstrates the ability to adapt to input scene content by offloading and spawning an additional detection thread only when faces were detected in the input scene. Until frame number 100, there were no faces detected in the input scene and the application sends only 31 bytes of data to the classifier that is running on the server. Beyond frame number 100, as the execution time of the detection thread increases and the network traffic increases, *Odessa* decides to co-locate the detection stage with the classifier stage on the server. Interestingly, on the laptop, *Odessa* does not offload the detection stage: in this configuration, the frame source stage is the more constraining bottleneck than the detection stage. Despite this, the laptop is able to support higher pipeline parallelism and a frame rate of 10 fps because of its faster multi-core processor. Thus, from this application, we see that *Odessa* adapts to input scene complexity and device characteristics.

Object and Pose Recognition. *Odessa* achieves a frame rate of 6.27 fps and 7.35 fps on the netbook and laptop respectively and a makespan of under 900 ms on both platforms. In this application, *Odessa* converges to the same final configuration on the netbook and laptop. Note that *Odessa* is able to differentiate between the impact of the two offloaded stages (feature generator and model matcher); as shown in Figure 10, *Odessa* continues to increase the number of feature generator threads beyond frame number 50 as it increases the frame rate of the application with minor decrease in the makespan of the application. The final configuration results in 10 SIFT feature generator threads but only 3 for model matching. From this example, we see that despite device differences, *Odessa* ends up (correctly) with the same offloaded stages, and is able to assign the requisite level of data parallelism for these stages.

Gesture Recognition. In this example, differences in device capabilities result in different *Odessa* configurations. On the netbook *Odessa* offloads the single detection thread and spawns only 4 ad-

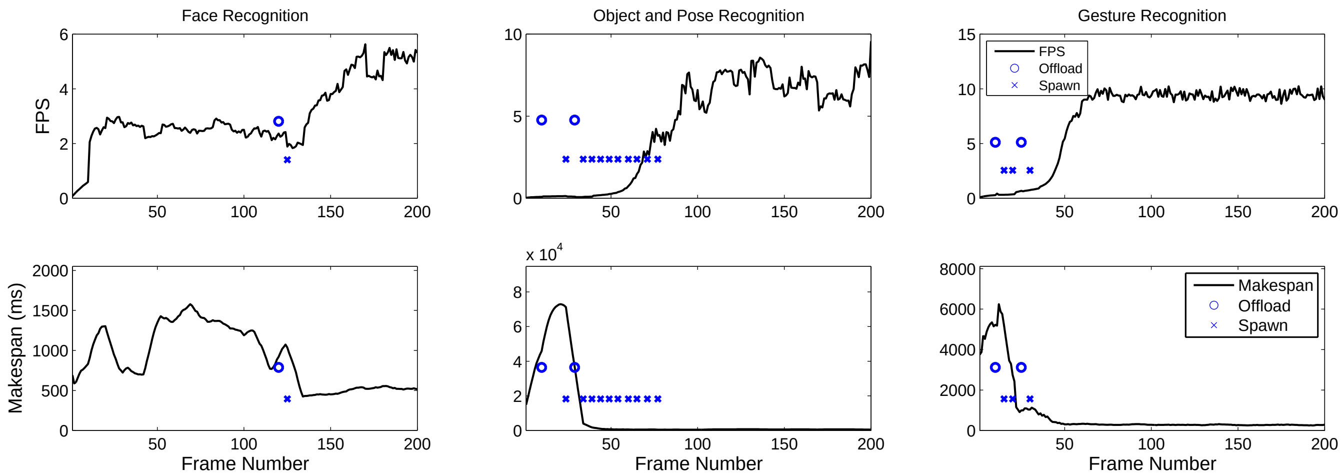


Figure 10: Figure shows the decisions made by *Odessa* across the first 200 frames along with the impact on the makespan and frame rate for the three applications on the netbook.

ditional motionSIFT threads but has a higher average pipeline parallelism of 3.06 tokens. Beyond 4 motionSIFT threads, the frame source stage on the netbook is the bottleneck and *Odessa* cannot further improve the performance of the application. However, on the laptop platform the frame source stage is not the bottleneck due to the dual core architecture and faster processor speed. Consequently, *Odessa* spawns a total of 9 motionSIFT threads and more than doubles the average pipeline parallelism to 5.14. This leads to a 68% increase in frame rate on the laptop compared to the netbook platform.

Overhead. *Odessa*'s overhead is negligible. The size per frame of measurement data collected by the lightweight profiler ranges from 4 KB for face recognition to 13 KB for pose detection, or from 0.1%-0.4% of the total data transferred along the pipeline. Moreover, the profiler's compute cost is less than 1.7 ms on each device, a negligible fraction of the application's makespan. Finally, the decision engine takes less than 1.2 ms for an offload or a spawn decision.

5.2 Comparison With Other Strategies

We now compare *Odessa*'s performance against three competing strategies. The first is the *Offload-All* partition in which only the video source stage and display stage run locally and a single instance of all other stages are offloaded to the server. The second is the *Domain-Specific* partition that makes use of the domain specific knowledge about the application and input from the application developer to identify the compute-intensive stages in the application graph. For this partition only the compute-intensive stages are offloaded to the server and number of CPU cores are *equally* distributed between the different parallelizable compute-intensive stages. These two strategies help us calibrate *Odessa*'s performance: if *Odessa*'s performance is not significantly better than these, an adaptive runtime may not be necessary.

We also consider another idealized strategy whose performance should, in theory, be *better* than *Odessa*'s. This strategy, called the *Offline-Optimizer*, mimics what an optimized offloading algorithm would have achieved if it could perfectly profile execution times both on the server and the mobile device. The inputs to this optimizer are the profiled run times of each stage on both the mobile and the server; the most compute-intensive stages are profiled with maximum data parallelism. Because execution times can vary with input complexity, for this experiment we use an input which consists of a single frame replicated 500 times. The output is an optimal configuration, obtained by exhaustively searching all configurations and picking those whose makespan and throughput dominate (i.e., no other configuration has a lower makespan and a higher throughput). The pipeline parallelism is optimally controlled and the number of tokens are adjusted to fully utilize the pipeline structure. We compare *Odessa* against *Offline-Optimizer* only for the pose detection algorithm: for the other algorithms, recognition is done over sequences of frames, so we cannot use frame replication to ensure consistent input complexity, and any performance comparison would be obscured by interframe variability.

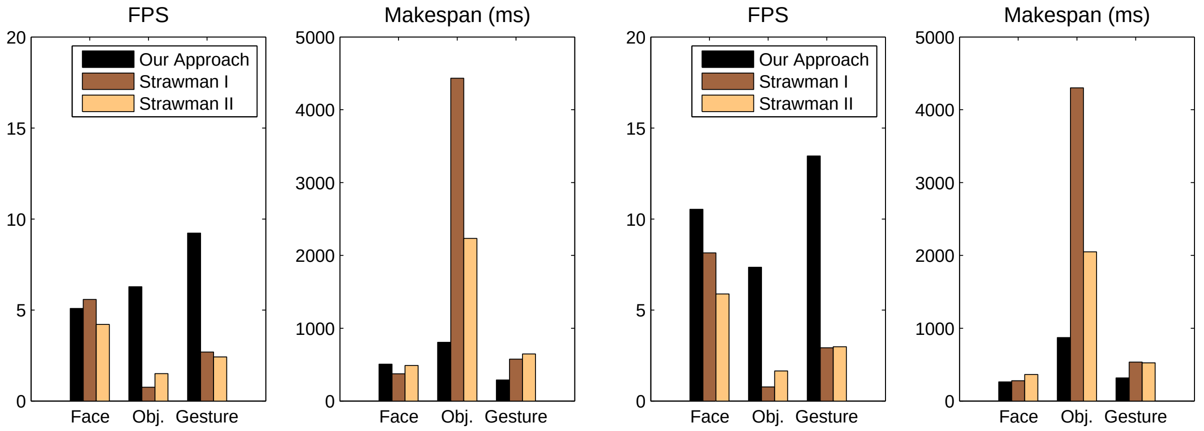
Application	Offloaded stages (instances)
Face Recognition	Face detection (4), Classifier (4), **2**
Object and Pose Recognition	SIFT Feature Generator (3), Object model matching (3), Clustering (2), **3**
Gesture Recognition	Face detection (1), MotionSIFT stage (8), **2**

Table 7: The table shows the instances of the offloaded stages along with the number of tokens (in boldface) in the pipeline for the *Domain Specific* partition of the application graph.

Table 7 shows the stages and their instances along with the number of tokens for the *Domain-Specific* static partition. Pipeline-parallelism for the *Domain-Specific* and *Offline-Optimizer* partitions is set based on the number of compute intensive stages in the application graph: the rationale is that, since most of the time will be spent in the bottleneck stages, adding additional tokens will cause increased wait times. Thus, the face recognition application has two tokens, the object and pose recognition application has 3 and the gesture recognition application has 2 tokens.

Figure 11 shows the aggregate performance of *Odessa* along with the two static partitions.

Face Recognition. *Odessa's* performance is comparable to *Offload-All* or *Domain-Specific* for the netbook; in this case, the frame source stage is a bottleneck, so *Odessa*'s choices approximate that of these two static strategies. However, on the laptop, *Odessa*'s throughput is almost twice that of *Domain-Specific*. This performance difference is entirely due to *Odessa*'s adaptive pipeline ad-



(a) Netbook

(b) Laptop

Figure 11: Figures shows the frame rate and makespan achieved by *Odessa* along with two statically partitioned application configurations across the two client devices.

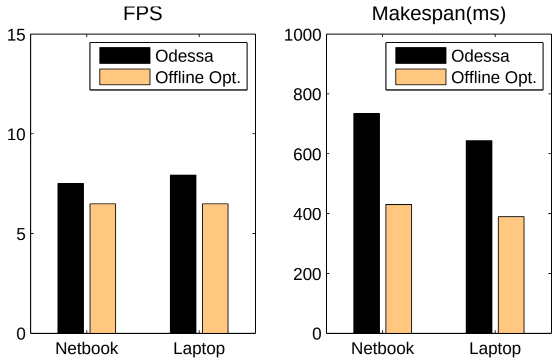


Figure 12: Figures shows the frame rate and makespan achieved by *Odessa* for pose detection, compared to the *Offline-Optimizer*.

mission controller; although, intuitively, 2 tokens seem like a reasonable choice given that this application has two bottleneck stages, much higher pipelining is possible because data parallelism significantly reduces the impact of the bottleneck.

Object and Pose Recognition. *Odessa* significantly outperforms both static partitions; for example, the frame rate achieved by *Odessa* is 4x higher and makespan 2.2x lower than *Domain-Specific* across both platforms. Unlike the latter, *Odessa* chooses to run the clustering algorithm on the local device for both the netbook and laptop platform which frees up extra CPU resources on the server. *Odessa* uses these extra CPU resources to spawn additional SIFT feature generator stages to reduce the pipeline bottleneck. Another reason for the performance difference is *Odessa*'s adaptive parallelism, as discussed above.

Finally, Figure 12 shows that the *Offline-Optimizer* has a lower makespan (about 200-300 ms lower) than *Odessa*, both on the netbook and the laptop. This is to be expected, since the offline optimizer does not model two important overheads to makespan: thread contention when executing 10 threads for each of the compute-intensive stages, and token waiting time. Encouragingly, *Odessa* achieves comparable or better throughput across the pipeline. These results indicate that, even if it were practical to profile stage execution times offline and optimize offloading decisions using these profiled times, the benefit is not significant.

Gesture Recognition. The frame rate achieved by *Odessa* is 3.4x and 4.6x higher on the netbook and laptop platform respectively compared to the static partitioning approaches and the makespan is under 350 ms for both platforms. Although *Odessa* achieves comparable data-parallelism as *Domain-Specific* on the laptop, it is again able to achieve much higher levels of pipelining, hence the performance difference.

5.3 Adapting to Varying Execution Contexts

The execution context of perception applications is primarily determined by the available computational resources and performance of the network. These resources can vary significantly as the mobile device moves from one execution context to another. For example, the network performance could vary significantly as the mobile device moves from an office to a home environment. Furthermore, the CPU resources available to the application could also vary if the server is shared by other applications. To address such sources of variation in available resources it is important for *Odessa* to be reactive and adapt the application partition quickly.

5.3.1 CPU Resources

We begin by evaluating the ability of *Odessa* to respond to events where additional CPU resources become available on server. We emulate such an event by changing the number of cores available to the *Odessa* decision engine from two to eight during the execution of the application (frame number 250). Figure 13 shows the application throughput and makespan along with the decisions made by *Odessa*. The application starts off running locally on the netbook. By frame number 31, *Odessa* utilizes the two CPU cores available on the server by offloading the SIFT feature generator stage and the model matcher stage and spawning an additional thread for each of the two stages. This increases the throughput from 0.1 fps to 2.5 fps. At frame number 250, 6 additional cores are available on the server. *Odessa* immediately reacts to this change in CPU resources and spawns 8 additional threads for the SIFT feature generator and 1 additional thread for the model matcher stage. Note than even though beyond frame number 260 the makespan does not in-

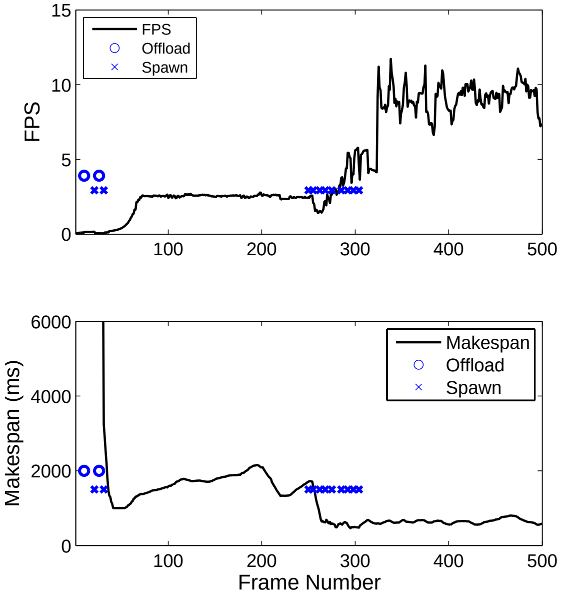


Figure 13: The ability of *Odessa* to adapt to abrupt changes in the number of CPU cores available to the application. The number of cores are increased from 2 to 8 at frame number 250.

crease, *Odessa* continues to spawn additional stages to further improve the throughput. *Odessa* completes adapting the application partition by frame number 305 after which the average throughput is 8.4 fps and the makespan is reduced to 490 ms.

5.3.2 Network Performance

We next evaluate the ability of *Odessa* to adapt to changes in network performance as the mobile device moves from a high bandwidth indoor 802.11n network to a low bandwidth outdoor network. The first two graphs in Figure 14 show the throughput and makespan of the application and the third graph shows the total amount of network data. The application starts off executing all the stages running locally on the netbook that is connected to the eight-core server over a 100 Mbps. At frame 122, *Odessa* decides to offload the detection stage to the server and spawn another instance of the detection stage on the server. After this, the image source stage that performs the jpeg decoding is the bottleneck and the performance cannot be further improved. At frame number 1237 the bandwidth of the network is dropped to 5 Mbps which makes the network edge the bottleneck of the application. The network delay significantly increases the makespan of the application and the throughput reduces significantly. Within 70 frames, *Odessa* decides to pull back the two detection threads from the remote server reducing the amount of data being sent over the network and the throughput increases to about 4 fps.

5.4 Data-Parallelism and Application Fidelity

Odessa increases data-parallelism until makespan and throughput changes are marginal. However, increasing data-parallelism may not always be desirable. For face recognition, or object and pose detection, increasing data parallelism by tiling images may *decrease* application fidelity, since, for example, a face may be split across tiles. Ideally, *Odessa*'s decision engine should take

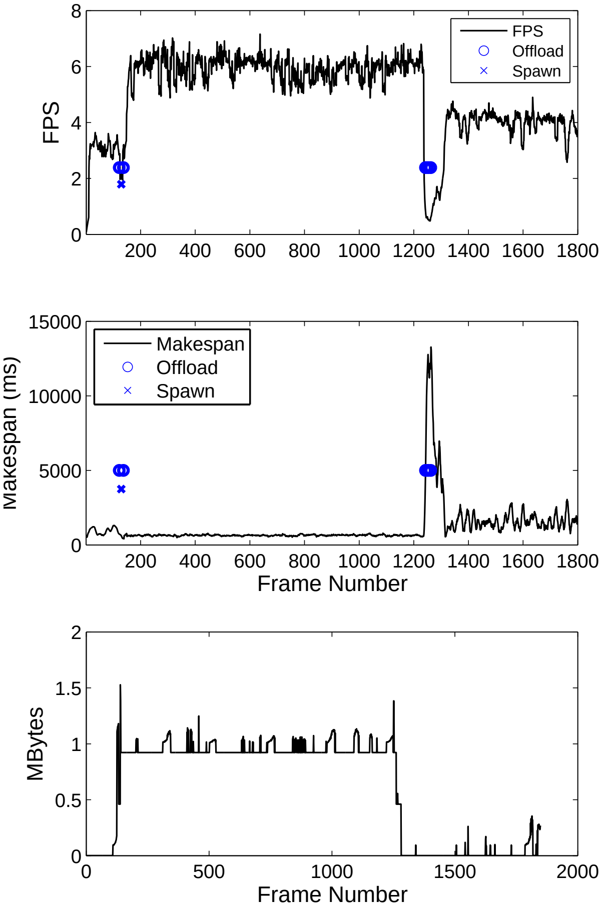


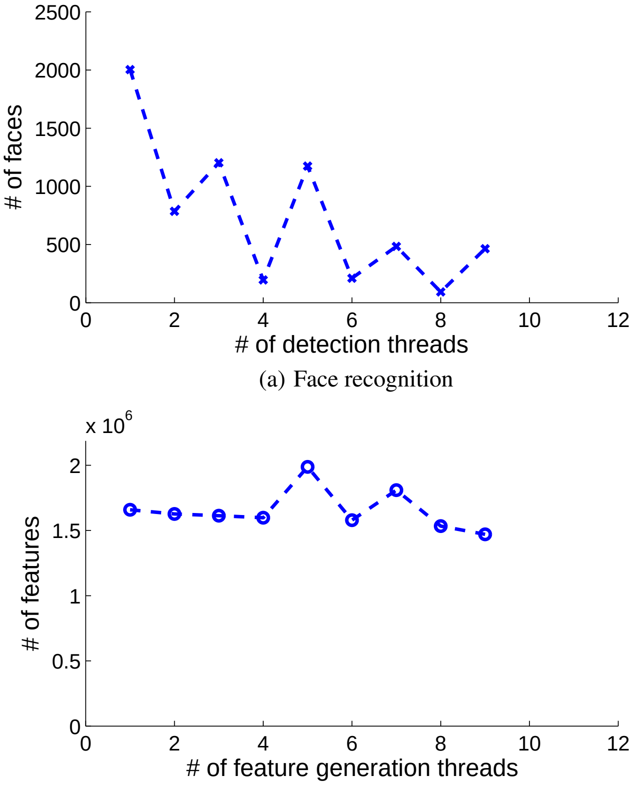
Figure 14: *Odessa* adapting to changes in network performance. The network bandwidth is reduced from 100 Mbps to 5 Mbps at frame number 1237. *Odessa* pulls back the offloaded stages from the server to the local machine to reduce the data transmitted over the network.

application fidelity into account when deciding the level of data-parallelism; we have left this to future work.

However, Figure 15 quantifies the impact of *Odessa*'s current design on application fidelity. On each graph, the x-axis shows the number of threads used (the level of data-parallelism) and the y-axis shows the total number of faces or features detected.

Face Recognition. In our current implementation, the Face detector uses a Haar classifier, which is unlikely to detect a face when the face is fragmented. Hence, the more an input image is divided, the fewer faces are detected (Figure 15(a)). In some cases, our image splitter adds redundant overlapping image tiles so that the loss of fidelity by tesselation is mitigated. More generally, however, a robustly parallelizable face detection algorithm might avoid this degradation.

Object and Pose Recognition. This application demonstrates such robustness. Its parallelizable stage, SIFT feature extraction, is scale-invariant [20], so the total number of detected features is relatively unchanged as a function of the number of threads (Figure 15(b)). The small fluctuations in fidelity can be explained by the loss of features at the edges of tiles. It may be possible to mitigate these fluctuations by overlapping tiles, which we have left to future work.



(a) Face recognition

(b) Object and Pose Detection

Figure 15: The number of features detected across different number of detection worker threads.

6. RELATED WORK

To our knowledge, no prior work has explored the joint automatic adaptation of offloading, pipelining and data parallelism.

The idea of offloading computation to networked computing infrastructure to overcome the limited capabilities of a wireless mobile device was proposed a decade ago [1, 2, 7, 27]. Since then, a variety of approaches for offloading computation to improve application performance or reduce resource usage have been proposed. Prior work makes use of three primary techniques to offload computation from the mobile device. The application partitioning is done either statically at compile time by Wishbone [23] and Coign [14], dynamically based on programmer specified partitions as in Spectra [11] and Tactics [3], or dynamically based on a run-time optimizer that uses integer-linear programming (as in MAUI [9] and CloneCloud [8]), or graph-based partitioners [13, 19, 24]. These pieces of work occupy different points in the design space relative to *Odessa* in terms of their approach. Some have significantly different goals (like conserving energy on the mobile device) from ours. Crucially, many of these pieces of work use a history of performance measurements, collected before execution, to predict stage execution times for the offloading decision. Narayanan et al. [22] show that history-based online learning of resource utilization outperforms other static methods, and more recently CloneCloud [8] shows the effectiveness of static analysis of Java code to dynamically partition applications. In contrast, *Odessa* uses a greedy and incremental approach guided by the application profiler and simple predictors that works very well to improve makespan and throughput.

Odessa's partitioning for makespan has a similarity with multiprocessor scheduling with precedence constraints, which has been studied extensively (see [18] for a survey). The problem is NP-complete even in the case of two processors and non-uniform execution times [12], so heuristics are typically used. Through an offline analysis we show that the performance achieved by *Odessa*'s greedy heuristic is comparable to an offline optimal decision obtained with complete profiling information. Yigitbasi et al. [31] demonstrate placement techniques to minimize makespans of interactive perception applications on general sets of heterogeneous machines. Like *Odessa*, they use fast heuristics and online profiling, but do not consider tuning of data parallelism or pipeline depth.

The System S distributed stream processing system provides operators capable of dynamically adjusting the level of data parallelism [29]. These *elastic* operators vary based on changes in workload and computational resources. Unlike in Sprout, the level of data parallelism does not extend beyond the boundaries of a single machine. Zhu et al. [32] describe an automatic tuner that operates on developer specified application parameters including the level of operator parallelism. The tuner learns application characteristics and effects of tunable parameters online to maximize application fidelity for a given latency constraint.

Also related to our work are parallel processing frameworks like MapReduce [10] and Dryad [15] for offline analysis of large datasets. While their runtimes schedule data-parallel tasks to optimize throughput or fairness to users, the setting is very different from ours (datacenters vs. mobile, large stored datasets vs. streams) that the details of the solutions vary. Finally, other work has looked at more general VM-based offloading mechanisms [28, 30], while ours relies on the mechanisms provided by Sprout framework.

7. CONCLUSION

In this paper, we have explored the design of a runtime, called *Odessa*, that enables interactive perception applications on mobile devices. The unique characteristics of the applications drive many of the design decisions in *Odessa*, whose lightweight online profiler and simple execution time predictors help make robust and efficient offloading and parallelization decisions. Our evaluation of *Odessa* shows that it can provide more than 3x improvement in performance compared to application configurations by domain experts. Additionally, *Odessa* can adapt quickly to changes in scene complexity, compute resource availability, and network bandwidth. Much work remains, including exploring the performance of *Odessa* under a broader range of applications, extending it to take advantage of the public cloud, and exploring easy deployability on mobile devices by leveraging modern browser architectures.

Acknowledgements

We would like to thank our shepherd, Rajesh Balan, and the anonymous referees, for their insightful suggestions for improving the technical content and presentation of the paper.

8. REFERENCES

[1] R. K. Balan. *"Simplifying Cyber Foraging"*. PhD thesis, 2006. (In CMU-CS-06-120).

[2] R. K. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. "The case for cyber foraging". In *ACM SIGOPS European Workshop*, 2002.

[3] R. K. Balan, M. Satyanarayanan, S.-Y. Park, and T. Okoshi. "Tactics-Based Remote Execution for Mobile Computing".

In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[4] G. Bradski and A. Kaehler. *"Learning OpenCV: Computer Vision with the OpenCV Library"*. O'Reilly Media, 2008.

[5] M. Carbone and L. Rizzo. "Dummynet revisited". *SIGCOMM Computer Communincation Review*, 40(2):12–20, 2010.

[6] M. Chen and A. Hauptmann. "MoSIFT: Recognizing Human Actions in Surveillance Videos". In *CMU-CS-09-161*, Carnegie Mellon University, 2009.

[7] J. Cheng, R. K. Balan, and M. Satyanarayanan. "Exploiting Rich Mobile Environment". Technical Report CMU-CS-05-199, Carnegie Mellon University, 2005.

[8] B.-G. Chun and P. Maniatis. "CloneCloud: Elastic Execution between Mobile Device and Cloud". In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, 2011.

[9] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. "MAUI: Making Smartphones Last Longer with Code Offload". In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[10] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". *Communications of the ACM (CACM)*, 51(1):107–113, 2008.

[11] J. Flinn, S. Park, and M. Satyanarayanan. "Balancing Performance, Energy, and Quality in Pervasive Computing". In *International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[12] M. R. Garey and D. S. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. W. H. Freeman and Company, New York, 1979.

[13] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt. "Adaptive Offloading for Pervasive Computing". *IEEE Pervasive Computing*, 3(3):66 – 73, 2004.

[14] G. C. Hunt and M. L. Scott. "The Coign automatic distributed partitioning system". In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In *European Conference on Computer Systems*, 2007.

[16] M. Kolsch. *"Vision based hand gesture interfaces for wearable computing and virtual environments"*. PhD thesis, 2004. (In 0-496-01704-7).

[17] B. Kveton, M. Valko, M. Philipose, and L. Huang. "Online Semi-Supervised Perception: Real-Time Learning without Explicit Feedback". In *IEEE Online Learning for Computer Vision Workshop*, 2010.

[18] Y.-K. Kwok and I. Ahmad. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors". *ACM Computing Surveys*, 31(4):406–471, 1999.

[19] Z. Li, C. Wang, and R. Xu. "Task Allocation for Distributed Multimedia Processing on Wirelessly Networked Handheld Devices". In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[20] D. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". *International Journal on Computer Vision (IJCV)*, 60(2):91–110, 2004.

[21] E. Miluzzo, T. Wang, and A. T. Campbell. "EyePhone: Activating Mobile Phones With Your Eyes". In *Workshop on Networking, Systems, Applications on Mobile Handhelds (MobiHeld)*. ACM, 2010.

[22] D. Narayanan and M. Satyanarayanan. "Predictive Resource Management for Wearable Computing". In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[23] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. "Wishbone: Profile-based Partitioning for Sensornet Applications". In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[24] S. Ou, K. Yang, and J. Zhang. "An effective offloading middleware for pervasive services on mobile devices". *Pervasive and Mobile Computing*, 3(4):362–385, 2007.

[25] P. S. Pillai, L. B. Mummert, S. W. Schlosser, R. Sukthankar, and C. J. Helfrich. "SLIPstream: Scalable Low-latency Interactive Perception on Streaming Data". In *ACM International Workshop on Network and Operating System Support for Digital Audio and Video*, 2009.

[26] A. C. Romea, D. Berenson, S. Srinivasa, and D. Ferguson. "Object Recognition and Full Pose Registration from a Single Image for Robotic Manipulation". In *IEEE International Conference on Robotics and Automation*, 2009.

[27] M. Satyanarayanan. "Pervasive Computing: Vision and Challenges". *IEEE Personal Communications*, 8(4):10–17, 2001.

[28] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. "The Case for VM-Based Cloudlets in Mobile Computing". *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[29] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. "Elastic Scaling of Data Parallel Operators in Stream Processing". In *IEEE International Parallel and Distributed Processing Symposium*, 2009.

[30] Y.-Y. Su and J. Flinn. "Slingshot: Deploying Stateful Services in Wireless Hotspots". In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2005.

[31] N. Yigitbasi, L. Mummert, P. Pillai, and D. Epema. "Incremental Placement of Interactive Perception Applications". In *ACM Symposium on High Performance Parallel and Distributed Computing (HPDC)*, 2011.

[32] Q. Zhu, B. Kveton, L. Mummert, and P. Pillai. "Automatic Tuning of Interactive Perception Applications". In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.