SLIPstream: Scalable Low-latency Interactive Perception on Streaming Data

Padmanabhan S. Pillai, Lily B. Mummert, Steven W. Schlosser
Rahul Sukthankar, Casey J. Helfrich
Intel Research Pittsburgh
{padmanabhan.s.pillai, lily.b.mummert, steven.w.schlosser, rahul.sukthankar, casey.j.helfrich}@intel.com

ABSTRACT

A critical problem in implementing interactive perception applications is the considerable computational cost of current computer vision and machine learning algorithms, which typically run one to two orders of magnitude too slowly to be used interactively. Fortunately, many of these algorithms exhibit coarse-grained task and data parallelism that can be exploited across machines. The SLIPstream project focuses on building a highly-parallel runtime system called Sprout that can harness the computing power of a cluster to execute perception applications with low latency. This paper makes the case for using clusters for perception applications, describes the architecture of the Sprout runtime, and presents two compute-intensive yet interactive applications.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; D.2 [**Software**]: Software Engineering

General Terms

Algorithms Design Performance

Keywords

Parallel Computing, Cluster Applications, Multimedia, Sensing, Stream Processing, Computational Perception

1. INTRODUCTION

Interactions between humans and computers have been lopsided at best. Computers today have very rich output capabilities, and can communicate with human users with a variety of video, audio, text, and even physical (e.g., robotic or haptic) means. Although sensing capabilities have vastly improved, communication from humans to computers has been largely limited to a few input modalities — keyboards, buttons, mice, and joysticks. Natural interactions using voice and gestures in real world environments have been largely beyond the capability of today's systems. Most of the successes in this area, such as speech recognition for phone menus, virtual reality systems, and the Wii gaming interface, operate by constraining the problem, such as by using limited context-specific vocabularies, highly controlled environments or requiring the user to use special markers or devices while interacting with the system. Truly natural interactions that approach the richness and complexity of human-to-human visual and aural communications require a new class of interactive perception applications and systems that can process digital video and audio in unconstrained settings at interactive time scales.

Figure 1: Gestris: an example of a gesture-based interactive gaming system

Figure 1 shows our prototype of a camera-based natural interface, with which the user can interact with a game using gestures in an uncontrolled environment. Unlike early efforts in gesture recognition (e.g., [5, 29]), the user need not dominate the camera view and may appear anywhere in the scene, which can contain significant visual clutter and other moving objects. A key problem in realizing such interactive perception applications is that the current best approaches use very compute-intensive computer vision and machine learning techniques. These algorithms often run one or two orders of magnitude too slowly for interactive settings. Compounding this problem, sequential processing speeds have not improved significantly in recent hardware, as the semiconductor industry has shifted towards increasing the number of cores in microprocessors rather than increasing their speed. Making effective use of many-core architectures for computer vision and machine learning remains an open research problem.

The SLIPstream project attempts to address this issue by providing a highly parallel runtime system, called Sprout, that can harness the computing power of both multiple cores and multiple machines in a cluster environment to run perception tasks at in-

NOSSDAV'09, June 3–5, 2009, Williamsburg, Virginia, USA.

teractive speeds. Our system is designed to make developing and executing parallel applications as easy as possible. Sprout helps automate the execution and parallelization of applications on a cluster, and provides an easy-to-use API that hides much of the complexity of dealing with a parallel, distributed system. The application developer only needs to be concerned with the coarse-grained parallelism and structure of the application, which is expressed as a dataflow graph. The developer does not, for example, need to find low-level, fine-grained parallel computations to vectorize, although Sprout does not preclude such complementary optimizations. The system can very rapidly scale the computing resources available to an application by simply adding more machines. Furthermore, our approach attempts to use these parallel resources to not just increase throughput, but also reduce application latency, which is critical in an interactive setting. Our goal is to allow the designer to focus on developing algorithms for interactive perception, rather than distribution aspects or optimizing for available processing resources.

This paper describes our rationale and design for a parallel perception runtime system, and our experiences with an early implementation of our ideas.

2. RELATED WORK

Cluster-based interactive multimedia applications. FlowVR [3] and Stampede [24] both provide support for distributed execution of interactive multimedia applications on compute clusters. An application is structured as a dataflow of processing modules and explicit data dependencies. Modules execute asynchronously on separate threads. The underlying system transports data between modules transparently. FlowVR focuses on integration of disparate modules that execute at different rates or may themselves encompass parallel code, and a hierarchical component model that facilitates composition of large applications [22]. Unlike in Sprout, latency and parallelization are controlled by hand tuning of module code, execution rates, and placement on compute nodes. Stampede emphasizes space-time memory (STM), a distributed data structure for holding time-indexed data, as a key abstraction around which applications are constructed. While modules are placed on compute nodes to minimize latency, the placement algorithm assumes that the number of modules and data-parallel variations is small enough to pre-compute optimal configurations [20]. Sprout assumes a shared-nothing model based on explicit data channels between modules and makes no assumptions about the number of modules or configurations.

Distributed stream processing engines. Systems such as Aurora [8], Borealis [2], and TelegraphCQ [7] provide support for continuous queries over data streams. These systems are used for applications such as financial data analysis, traffic monitoring, and intrusion detection. Data sources supply tuples (at potentially high data rates) which are routed through an acyclic network of windowed relational operators, such as selection, projection, union, aggregation, and join. Operators and data are distributed over compute nodes to achieve a QoS goal, typically a function of performance (e.g., latency), accuracy, and reliability. Compute nodes may be geographically distributed. QoS is managed by dynamically migrating operators, partitioning data, shedding load, and reordering operators or data. Although these systems process streaming data, perform runtime adaptation, and consider real-time constraints, they are limited to relational operators and data types. Other stream-processing systems such as XStream [11] and GigaScope [16] go beyond relational operators and data types, but are narrowly focused on specific domains.

System S [4] provides support for user-defined operators, stream discovery, dynamic application composition, and operator sharing between applications. It has been used to process multimedia streams, and assumes a resource-constrained data center environment in which utilization is high and jobs may be rejected. Compute resources are allocated to applications to maximize an importance function, typically a weighted throughput of output streams [28], unlike Sprout which is primarily concerned with low latency.

Coarse-grain data-parallel systems. MapReduce [9] and Dryad [15] are systems that allow large data sets to be processed in parallel on a compute cluster. MapReduce applications consist of user-specified *map* and *reduce* phases, in which key-value pairs are processed into intermediate key-value pairs, and then values with the same intermediate key are merged. Dryad admits a more general application structure; a job consists of an acyclic dataflow graph of sequential processing modules. Both systems operate from stored data rather than streams, and are employed in off-line rather than interactive applications. Like Sprout, MapReduce and Dryad provide simple programming abstractions and handle many of the messy details of distributed computation.

Language support for stream applications. Streaming languages provide high-level programming constructs to enable efficient use of multiple processors on a single machine. Brook [6] extends C with data-parallel constructs for stream operations on graphics hardware. StreamIt [12], StreamC [17], and SPUR [30] represent programs as dataflows of processing modules, enabling the compiler to extract task, data, and pipeline parallelism. Generally, module execution times and data rates must be known at compile time to construct a steady-state program graph and map it to the underlying hardware. StreamWare [13] relaxes this requirement by providing a platform-independent stream virtual machine abstraction to the compiler and application, and mapping operations to hardware at run time. In contrast, since perception workloads are highly data-dependent, Sprout focuses on runtime adaptation.

3. DESIGN

3.1 Interactive perception workloads

Interactive perception applications, whether processing video or audio, typically consist of the following steps. First, raw data is encoded as a set of low-level features. These *local descriptors* characterize the content in a localized spatial and temporal neighborhood and can either be sampled densely throughout the stream or only at specific interest points [10, 21]. Standard representations for audio exploit the frequency domain (e.g., using short-time Fourier transforms) while common descriptors for video include patches [19], motion estimated using optical flow [18], or spatio-temporal generalizations of SIFT [21,23]. The computed descriptors can be stored as high-dimensional features or quantized into discrete "words" using a clustering algorithm (e.g., k-means [14]). The latter are required for higher-level representations that express the stream using histograms, such as in the popular "bag of features" model.

In the next step, the representation from the incoming stream is matched against training data. In the simplest case, matching could involve a straightforward correlation between known templates or (more typically) employ a discriminative classifier, such as a support vector machine [25, 26] or a cascaded linear combination of decision stumps [18, 27] trained specifically to recognize events of interest. In most cases, matching is performed using a *scanning window* approach, which involves sweeping a region of interest in a brute force manner over the stream both in space and time. The

sweep is often performed at multiple spatial scales because of perspective effects that cause objects closer to the camera to appear larger in the image, while distant objects occupy only a small portion of the frame. Scanning window approaches are computationally intensive but are generally very amenable to parallelization. Despite their expense, such brute force approaches allow the perception algorithm to localize the detected event in space and time, and are also more robust to scene clutter.

The final step aggregates lower-level matching results, first by eliminating matches that fall below a specified detection threshold and then combining multiple events detected in similar locations and scales (non-maxima suppression). The perception algorithm can thus flag events of interest and localize them in space and time, if needed.

3.2 Application model

The application model and interfaces used in our system have been designed for ease of use. Our approach to parallelizing interactive perception tasks is based on identifying and executing coarse-grained parallel components in an application. Hence, the developer only needs to divide his application into a series of processing stages that exhibit a few explicit data dependencies, i.e., one stage produces a particular set of data that is consumed by another. These relations are captured in a dataflow graph. This abstraction is particularly well suited for perception, computer vision, and multimedia processing tasks, as it mirrors the high-level structure of these applications, which typically apply a series of processing steps to a stream of video or audio data. The developer does not need to identify or extract any fine-grained parallelism in his application. In particular, the developer does not need to extract instruction- and block-level parallelism, nor vectorize computations. Although our system permits and encourages the use of multithreaded or vectorized code, the developer can simply write sequential code for the processing stages. In fact, the developer does not even need to worry about thread-safety of his code or the libraries it uses. Dealing with parallelism at this level of detail should be easy for the developer, a key goal of our approach.

In keeping with the ease-of-use goal, the API for writing a stage has been designed to require minimal additional effort from the developer. Our system is entirely written in C++, a standard language familiar to most developers and amenable to high performance applications. An implementation of a stage needs to define just one `exec()` method that takes one or more parameters for input data; any outputs produced are passed back through additional pointer parameters to this function. This is all that is strictly necessary for writing a stage. The developer does not have to deal with communication primitives or buffers, as our system handles inter-stage communications, and provides data in native user-defined structures and classes. Hooks are provided for initialization and shutdown methods, as well as for any special marshaling code that may be needed for the user-defined classes (e.g., deep copying, or special memory allocation). The stage executes the provided `exec()` method when all inputs are ready; an optional firing rule can be specified to change this behavior. Outputs are automatically propagated to downstream stages. Outputs and inputs may connect to multiple parallel instances of stages to realize parallel execution structures.

Finally, our system uses a human-readable configuration file to indicate how the stages are assembled to form the application, essentially defining the dataflow. As our system is intended to automate replication of stages and degrees of parallelism, this file is actually a template of the structure of the application, with hints for extracting parallelism.

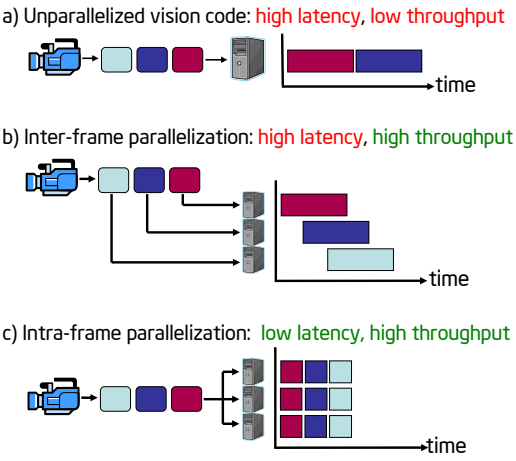

Figure 2: Parallel execution to minimize latency

3.3 Parallel execution

Given an application divided into stages, and a template dataflow graph describing how the components are connected, our system attempts to parallelize the execution of the application by mapping instances of the stages to multiple processors and machines. The actual methods of parallelization employed greatly affect any speedup achieved, and in particular whether latency or throughput is improved. Figure 2 illustrates this idea for an image processing task that performs independent processing on frames of a video stream. The sequential application is slow, in terms of frame latency and throughput. As frame operations are independent, we can make use of inter-frame parallelization by processing subsets of the frames on multiple instances of the application. This allows the throughput to increase, but latency for a given frame remains the same. Ideally, one would exploit intra-frame parallelization — dividing the processing of *each* frame among multiple machines can improve both latency and throughput. However, there is some cost to this approach. Inter-frame parallelization requires little knowledge or modification of the application, while intra-frame parallelization may require intrusive modifications.

Sprout makes use of several parallelization techniques to achieve low-latency execution. The template dataflow graph is inherently a task-parallel model, and the runtime is free to execute separate stages in parallel when data dependencies permit. When a configuration file indicates that data items are independently processed by a stage, the runtime may instantiate multiple copies of the stage, and distribute processing among these to improve throughput. Data-parallel constructs are also supported. For example, a stage that compares a data item to those in a large database can be executed as multiple instances, each operating on a subset of the database. In this example, some modification of the stage code is needed, as it must export a set of tuning methods that lets the runtime assign a subset of work to each instance. To correctly connect the parallel stage instances, the runtime has built-in adapters to duplicate data streams or split them in a round-robin fashion, as well as collect and merge outputs. More intrusive mechanisms for reducing processing times, such as splitting a frame among parallel stages, are supported, but require more effort from the application writer to devise custom split and merge adapters.

Sprout expands the template dataflow configuration and distributes the requisite set of stages across a cluster of machines. As processing time for many interactive perception applications is highly

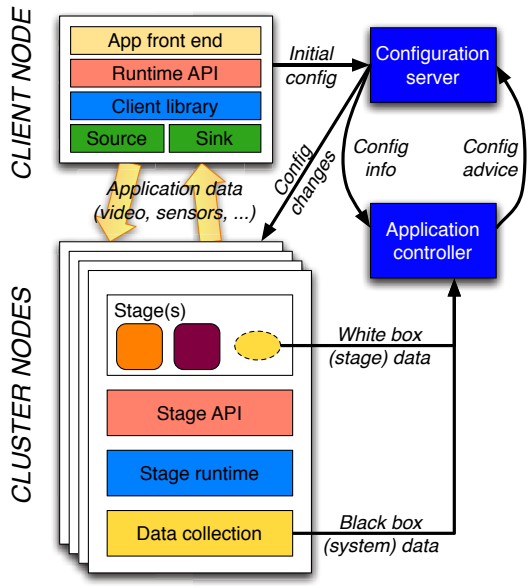

Figure 3: Sprout system diagram

data- and scene-dependent, and may not be known precisely *a priori*, a full implementation of our system will incorporate runtime monitoring and adaptation, varying the degree of parallelism to meet latency and throughput requirements, in addition to automated placement of stages. The system must be able to dynamically create, shutdown and migrate stages to balance loads and latencies. Finally, in particularly resource-constrained situations, the system will resort to load shedding (e.g., dropping frames, or other application-specific mechanism) to gracefully degrade performance while ensuring low latency.

4. IMPLEMENTATION

4.1 Architecture of Sprout runtime

Sprout applications consist of a set of processing stages that run in parallel on a compute cluster. Each machine in the cluster runs a *stage server* process, which executes the stages assigned to that machine. Each stage running in the stage server has a single thread to run its `exec()` method when its input arrives. Auxiliary threads in the stage server manage input and output connections and respond to RPC calls from Sprout clients and external programs to handle stage management and monitoring. Performance dictated our choice of a single process as the container for all stages on a node in order to minimize context switch time.

Programmers implement application stages according to the Stage API and link them against the Sprout stage server library. This produces a single binary which can run any stage in the application, allowing the Sprout runtime to manage stage placement by selectively activating user stages as appropriate. We chose selective activation over dynamic linking to simplify code management.

Sprout *client* programs link with the Sprout client library which provides methods for instantiating applications. A client provides a simple configuration file which specifies the application graph. The client library passes this graph to the Sprout configuration server (described below) which maps the application graph to the cluster nodes, instantiates the stage servers, orchestrates data connections between the stages, and activates them once setup is complete.

Data is delivered to a Sprout application by data *sources*. We have implemented sources which provide images from cameras and files, as well as data from distributed file systems. As specialized stages, sources generally run within stage servers, but can also be instantiated within Sprout clients if needed. Data is consumed by data *sinks*, which are specialized stages that accept input but provide no further output to the Sprout graph. Rather, their output is displayed to the user, stored in an archive, or routed to other external systems. A Sprout application can have any number of sources and sinks connected at any point in the application graph. Data connections between stages, sources, and sinks are managed by the Sprout runtime, not application writers. Connections are either over TCP sockets between machines or via local memory references when two stages run on the same machine.

For each Sprout cluster, a *configuration server* manages the placement, startup, and shutdown of stages. The configuration server is centralized so as to have a single view of the cluster and applications it manages. The configuration server's interactions with applications are occasional rather than in-band, so it does not need to be extremely scalable.

The process for application setup is initiated by a Sprout client and orchestrated by the configuration server. The configuration server generates an initial placement of stages to stage servers, invokes stage servers on the cluster machines if they are not already running, and directs those stage servers to instantiate the appropriate application stages. Once the stages are instantiated, the stage servers create input and output connections for each stage, either over TCP or through local memory, as appropriate. The configuration server then directs each stage to connect to the stages immediately downstream. Once those connections have been made the configuration server directs the stages to start processing.

The last component in the Sprout runtime is the application controller, which is responsible for runtime adaptation. The application controller gathers application-specific or *white-box* observations about the status of the application from the stages themselves. These can be any manner of data including frame rates, processing time per frame, number of extracted features, etc. As well, the application controller gathers application-agnostic or *black-box* observation about the status of the systems themselves, such as the utilizations of CPU, network, and disk. These observations drive the decision-making process for adaptation, which will suggest *advice* to the configuration server for changes to be made in the application. These changes can include adjusting the level of parallelism, co-location or migration of stages, or other application-specific adaptations. As structural changes are made by the configuration server, they are communicated back to the application controller.

Runtime adaptation is a very rich area of future work for SLIPstream, and we have only begun to scratch the surface. Our initial implementation of runtime adaptation in Sprout includes the data collection architecture for both white- and black-box data, and some simple decision trees for detecting and mitigating CPU bottlenecks by adjusting parallelism.

4.2 Example application: Gestris

Gestris is an interactive two-player game system in which players use hand gestures to move and rotate blocks in a Tetris-style game (Figure 1). The system requires no special props, clothing, or markers. Instead, gestures are detected from two video streams (one for each player) using a volumetric event detection algorithm [18] that is robust to background clutter. The gestures are translated to keystrokes that control the actual game, which has not been modified.

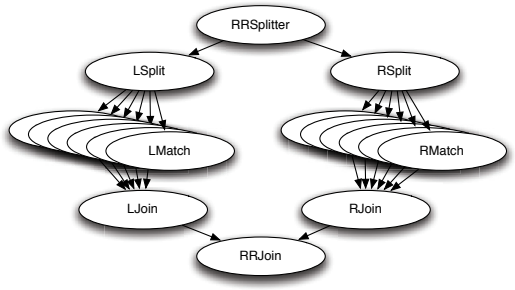

Figure 4: Gestris application graph

The Gestris application has just one processing stage that matches a set of gesture templates to a region of a sequence of video frames. The perception system receives an interleaved stream of frames from both cameras. To parallelize this application, we separate these video streams, and process them concurrently. The matcher stage is replicated, and each instance assigned a disjoint subregion in which to check for gestures. The sequence of matching gestures is merged, and returned to the keystroke generator. The complete graph of the Gestris perception system shown in Figure 4 runs on two machines equipped with 3.0 GHz quad-core Intel® Core™ 2 Extreme processors, and handles 15 frames per second from each camera with latencies under 250 ms. A third machine handles video acquisition, keystroke generation, and execution of the game itself.

4.3 Example application: Object recognition

pMocha is a parallelized version of an object instance recognition application [1], which consists of three major components that execute in Sprout stages: SIFT feature generation [23], similarity calculation against a database of training images, and classification. The full application graph appears in Figure 5.

pMocha exploits several opportunities for parallelism. Incoming frames are sent round-robin to subtrees of feature generator stages. Each subtree splits an incoming image into five subimages (four quadrants plus an overlapping center subimage) and then generates SIFT features from each. The features for a whole image are merged by the ImageMerger stage, and those from alternating frames from the left and right subgraphs are ordered by the InputJoiner. Each set of features is compared against a database of training images, generating a similarity vector which is used downstream for classification. The database is partitioned among the workers, which each receive a copy of the features. Finally, the similarities are gathered and classified, resulting in the object's identification. The three major components run concurrently in a pipeline fashion.

Re-factoring the original Mocha application to run on Sprout was a straightforward task, requiring a few days for a programmer who had never worked with Mocha before. We have scaled pMocha to process live video at a resolution of 640x480 pixels per frame, running at 25 frames per second, with a latency of between 0.08 and 0.5 seconds (2–10 frames outstanding). To maintain that data rate, pMocha requires 14 8-core servers, each with two four-core 2.83 GHz Intel® Xeon™ E5440 processors and 8 GB of memory. The majority of the machines (10 out of 14) are devoted to SIFT feature generation, two are devoted to similarity calculation, and the remainder for splitting and joining. The original non-parallelized implementation of Mocha on one 8-core machine can only sustain two frames per second.

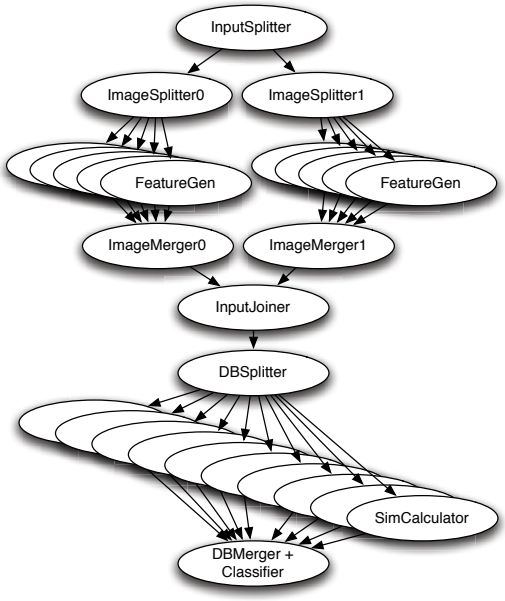

Figure 5: pMocha application graph

Optimizing pMocha, even if only by hand, has taught us about some of the tasks that lay ahead for runtime adaptation. First, throughput bottlenecks quickly became evident at specific stages (in particular, the SIFT feature generator and the similarity calculator), and were addressed by increasing the level of parallelism, when possible. Second, while the increased throughput from parallel stages was able to keep up with the frame rate, initially latency was unacceptable due to processing time for feature extraction. As a solution, we introduced subimage feature extraction, a form of intra-frame parallelization, which reduced the latency by roughly a factor of five. Third, we encountered load imbalances in both the parallel feature generators and the similarity calculators because processing time is strongly data dependent. Complex images tend to have more features and therefore take longer to process. Because objects often appear centered in the frame, the central subimage tends to contain more features and requires more time to process than the others. To prevent load imbalances, we randomly assigned work among the parallel stages. Lastly, the SIFT feature generator transparently uses the Intel Performance Primitives (IPP) library to parallelize SIFT at a fine granularity, independent of the coarse-grained parallelism of Sprout. To avoid interference with IPP, we dedicated entire machines to feature generation, mapping other pMocha functions to their own machines.

5. CONCLUSIONS AND FUTURE WORK

The SLIPstream project is pursuing natural modalities of interaction between humans and computers. A key problem is that computer vision and machine learning algorithms used in perception tasks have very high processing requirements and unacceptably high latencies. We believe that harnessing the scalable processing capacity of computer clusters will be a key enabler for these applications.

This paper presents our design for Sprout, a core systems component of the SLIPstream vision, which provides the APIs and runtime support to implement parallel, interactive perception applications. Initial results from two applications implemented on

the Sprout prototype indicate that our approach is effective for developing parallel vision algorithms, tuning them for latency, and enabling interactive-speed perception applications that operate in unconstrained environments. We hope that Sprout will prove to be easy-to-use and readily applicable to a broad range of vision applications, and that it can serve as a form of rapid-prototyping system for interactive perception applications. In particular, Sprout allows one to focus on creating algorithms rather than tuning for performance, yet achieves interactive speeds by exploiting the available hardware resources. Later, focused tuning and optimization efforts can be applied to achieve the performance goals more efficiently.

Sprout is currently a work in progress. In particular, runtime adaptation, automatic placement of stages, and system optimization for latency are under active development. We are also investigating systematic ways to incorporate domain-specific techniques for managing fidelity, such as load shedding and dynamically adjusting classification accuracy. A complete implementation of Sprout will be an effective tool for developing and executing interactive perception applications.

6. REFERENCES

[1] Visual Object Instance Recognition. `http://people.csail.mit.edu/rahimi/projects/objrec/`.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. Innovative Data Systems Research*, 2005.

[3] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR: a middleware for large scale virtual reality applications. In *Proc. Euro-Par*, 2004.

[4] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A Distributed, Scalable Platform for Data Mining. In *Proceedings of the Workshop on Data Mining Standards, Services, and Platforms*, 2006.

[5] A. F. Bobick and J. W. Davis. The recognition of human movement using temporal templates. *PAMI*, 23(3), 2001.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the Conference on Innovative Data Systems Research*, 2003.

[8] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the Conference on Innovative Data Systems Research*, 2003.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1), 2008.

[10] P. Dollar, V. Rabaud, G. Cottrell, and S. Belongie. Behavior recognition via sparse spatio-temporal features. In *IEEE Workshop on PETS*, 2005.

[11] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. XStream: a Signal-Oriented Data Stream Management System. In *Proc. International Conference on Data Engineering*, 2008.

[12] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[13] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proc. Architectural Support for Programming Languages and Operating Systems*, 2008.

[14] J. Hartigan and M. Wang. A k-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979.

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

[16] T. Johnson, M. S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. Query-aware partitioning for monitoring massive network data streams. In *Proc. SIGMOD*, 2008.

[17] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable Stream Processors. *IEEE Computer*, pages 54–62, August 2003.

[18] Y. Ke, R. Sukthankar, and M. Hebert. Efficient visual event detection using volumetric features. In *Proceedings of International Conference on Computer Vision*, 2005.

[19] Y. Ke, R. Sukthankar, and M. Hebert. Event detection in crowded videos. In *Proceedings of International Conference on Computer Vision*, 2007.

[20] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proc. Supercomputing*, 1999.

[21] I. Laptev and T. Lindeberg. Space-time interest points. In *Proc. International Conference on Computer Vision*, 2003.

[22] J.-D. Lesage and B. Raffin. A Hierarchical Component Model for Large Parallel Interactive Applications. *The Journal of Supercomputing*, July 2008.

[23] D. Lowe. Distinctive image features form scale-invariant keypoints. *IJCV*, 60(2), 2004.

[24] U. Ramachandran, R. Nikhil, J. M. Rehg, Y. Angelov, A. Paul, S. Adhikari, K. Mackenzie, N. Harel, and K. Knobe. Stampede: a cluster programming middleware for interactive stream-oriented applications. *IEEE Trans. Parallel and Distributed Systems*, 14(11), 2003.

[25] C. Schuldt, I. Laptev, and B. Caputo. Recognizing human actions: A local SVM approach. In *Proc. International Conference on Pattern Recognition*, 2004.

[26] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[27] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proc. Computer Vision and Pattern Recognition*, 2001.

[28] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Proc. ACM/IFIP/USENIX Middleware*, 2008.

[29] C. Wren, F. Sparacino, A. Azarbayejani, T. Darrell, T. Starner, A. Kotani, C. Chao, M. Hlavac, K. Russell, and A. Pentland. Perceptive spaces for performance and entertainment: Untethered interaction using computer vision and audition. *Applied AI*, 11(4), 1996.

[30] D. Zhang, Z.-Z. Li, H. Song, and L. Liu. A Programming Model for an Embedded Media Processing Architecture. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2005.