Fusion Channels: A Multi-sensor Data Fusion Architecture

Bikash Agarwalla, Phillip Hutto, Arnab Paul, Umakishore Ramachandran
College of Computing
Center for Experimental Research in Computer Systems
Georgia Institute of Technology
801 Atlantic Drive NW
Atlanta, GA 30332-0280 USA

Abstract

Due to the falling price and availability of sensors, information capture and processing at a realtime or soft realtime rate is emerging as a dominating application space. This class includes interactive multimedia, robotics, security and surveillance applications and many more. A common denominator of these applications is fusion of data gathered by various sensors and data aggregators. In this paper we propose a Data Fusion architecture, specifically geared toward such multi-sensor data fusion applications and report on the prototype we have built. Our infrastructure provides a programming abstraction that offers programming ease, at the same time provides built-in optimizations that are quite complicated to implement from scratch. We show the ease of programming through two sample applications and also demonstrate through various experiments that our system has low overhead and offers better performance compared to otherwise naively written fusion routines. We also demonstrate improved scalability.

1 Introduction

We are interested in an emerging class of applications that involve the capture, interpretation and interactive access to continuously streaming data. We believe that such applications, arising from the confluence of mobility, personal media, interactivity, and the ubiquity of small, high-powered computational devices (e.g. sensors), place

interesting and unique demands on system infrastructure and application programmers. Applications in this class include interactive vision applications such as smart kiosks [4], distributed meetings, and telepresence [5], multi-robot coordination [2, 3], aware spaces [1], and safety and security activities such as monitoring and surveillance.

There are two important characteristics of this application space. First, application structure often represents a control loop with data evolving temporally. The loop begins with continuous data capture via sensors, followed by multiple stages of processing, and ends with actuators or effectors. The results of actuation are then sensed by the next round of processing. Second, data from different sources may be combined or *fused* in the pipeline to enable higher-level inferences. Data capture time is often used to correlate items in preparation for fusion.

We seek to provide efficient and flexible system support for *data fusion* applications. Multi-sensor data fusion is a special but important case of data fusion and our efforts have been driven partially by system support for large-scale, heterogeneous sensor networks. Data and sensor fusion applications are increasingly common and enhancing system support for fusion applications will provide a variety of benefits to programmers.

- *Sampling accuracy*: Fusing data from a number of overlapping sensors can improve overall precision and increase inference accuracy.
- *Increasing coverage*: For example, in a distributed meeting, a mixer module may generate a composite of multiple audio and video feeds.
- *Enhancing interaction*: For example, a team of mobile robots might coordinate by exchanging messages with sensor data; each robot gathers input from others and makes its own local decision.
- *Providing flexibility*: In robotics applications, it is often more important to fuse limited available data than to wait indefinitely for some late arriving sensor data. Thus the flexibility to tactically discard data is extremely important in implementing such control loops.
- *Supporting synchronization*: Time is an important attribute in sensor-based applications involving control loops. Sensors sample the world state, which naturally evolves over time.

The key contribution of this paper is the design and implementation of a novel sensor fusion architecture that provides a number of benefits to programmers.

1. It provides an easy-to-use API that abstracts and simplifies common fusion tasks such as the maintenance of timestamp information and pipeline "plumbing" for the construction of fusion networks.

2. It supports access to distributed data sources, such as sensors, and handles typical distributed systems issues such as failure and latency.

3. It offers various data management policies to programmers such as prefetching and caching of data to reduce fusion network latency.

4. It provides efficient thread management to exploit the task parallelism inherent in distributed fusion and sensor networks.

Our fusion architecture is currently implemented as a runtime library over the Stampede programming system [6]. Stampede is a programming infrastructure that provides certain advantages for our implementation such as a global thread model, cross-address-space data structures, and support for time-sequenced data streams, but the architecture is fairly general and could be implemented in a variety of alternative systems. We are using this implementation to develop a variety of applications (robotics, interactive vision, distributed aware spaces, etc.) with collaborators at Georgia Tech. We have evaluated our system qualitatively for ease of use and quantitatively for performance with microbenchmarks and application level metrics.

The rest of the paper is structured as follows. Section 2 describes several requirements that motivate the design of our fusion architecture. Section 3 introduces the architecture and the primary abstraction, fusion channels. In Section 4 we describe the Stampede programming system and the implementation of our architecture as a Stampede library. We describe our experience with several applications implemented using the fusion channel architecture in Section 5. In Section 6 we discuss performance issues and present microbenchmarks and application level metrics. Section 7 surveys related work and we offer conclusions in Section 8.

2 Requirements

Several requirements motivate our fusion architecture. Our over-arching goal has been to provide the programmer with an efficient, scalable API that simplifies the construction of control loop applications involving data fusion. Scalability is important since emerging richly-sensored environments may contain hundreds or thousands of independent sensors, forming sensor networks. Multi-sensor data fusion is simply a special case of the more general process of data fusion. Other forms of data fusion have similar scalability demands.

Data fusion fundamentally involves the collection or *aggregation of correlated data* items and the application of a potentially arbitrary (user-specified) fusion function. Correlation is typically time-based, requiring aggregation

of items generated within the same time interval. The result of fusion is a newly created or *fused* data item.

Our architecture specifically addresses several issues that we believe are characteristic of data fusion applications and that introduce complexity for application programmers. A expedient data fusion architecture should:

1. *Tolerate data arrival indeterminacy*: In loosely-coupled environments, timely data delivery is not guaranteed. Delay can be the result of transient or permanent sensor failure, link failure, or excessive traffic producing unbounded delay. Fusion applications typically require availability of all inputs before the application of the fusion function. Writing code to handle failure and delay in every application is burdensome and delays in one stage compound, producing further delays "down-stream".

2. *Avoid duplicate fusion and support efficient reuse of fused data*: Fused data (e.g. robot position and velocity) is often processed by more than one consumer in non-trivial fusion applications. Overlapping or subsequent requests should not result in redundant application of the fusion function (although flexibility should exist to allow the application of *different* fusion functions to the same input data). Data, once fused, should be cached in anticipation of future requests. Caching must be balanced with the needs of buffer management and reclamation.

3. *Provide efficient and convenient management of memory buffers for fusion inputs and fused data*: The buffer requirements are very dynamic in nature, especially in the presence of failures and partial and indeterminate data availability. When less number of data items are being fused, less buffer space is required. Such vagaries in resource requirements are best dealt with by the data fusion architecture.

4. *Offer low-overhead feedback mechanisms to applications for controlling and monitoring the evolving state of the dynamic fusion network*: Control loop applications with multiple pipeline stages often require dynamism of the data fusion network. For instance, the sensor set of interest for a particular application stage may change over time due to application interest (imagine a surveillance application that brings additional sensors online once a probable threat has been identified) or perhaps in response to transient or permanent sensor failure. Notification of exceptional conditions as well as routine monitoring are required. Such requirements are similar to back-channels found in networking protocols such as TCP out-of-band data. Feedback should also be provided end-to-end in complex networks.

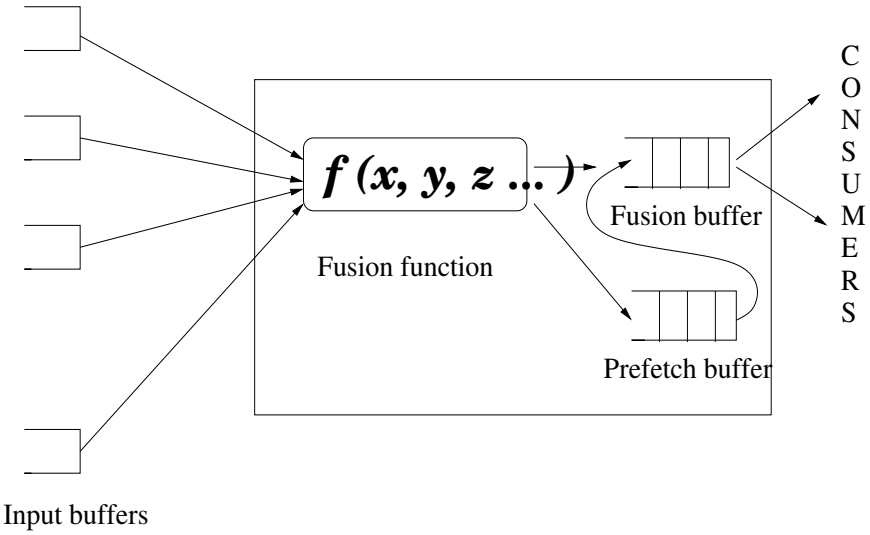

Figure 1. Fusion Channel Architecture

3 Fusion Architecture

The fundamental abstraction in our architecture is called a *fusion channel*. A fusion channel is a named, global entity that abstracts a set of inputs and encapsulates a programmer-supplied fusion function. Inputs to a fusion channel can come from a distinct address space or from a remote host. Item fusion is automatic and is performed according to a programmer-specified policy either on request (demand-driven, lazy, pull model) or when input data is available (data-driven, eager, push model). Items are fused and accessed by timestamp (usually the capture time of the incoming data items). An application can request an item with a particular timestamp or in a more general way (earliest item, latest item). Requests can be blocking or non-blocking. To accommodate failure and late arriving data, requests can include a minimum number of inputs required and a timeout interval. If sufficient inputs are not available by timeout expiry, partial fusion can be performed on the available inputs. Fusion channels have a fixed capacity specified at creation time. Finally, inputs to a fusion channel can themselves be fusion channels, creating fusion networks or pipelines. Figure 1 depicts a fusion channel. It takes a number of inputs, applies an application-specified fusion function to them, and generates a fused output.

3.1 Virtual Timestamps and Correlation

Fusion requires identification of a set of correlated input items. Correlating items based on capture timestamp is the most obvious choice. (Spatial correlation or correlating items captured during a particular interval are other possibilities.) In our architecture, timestamps are virtualized and can be arbitrary sequence values associated

with items by producers (e.g. video frame numbers). To guarantee that data items are synchronized, the virtual timestamp can simply be a real timestamp taken from hosts participating in a synchronization protocol (like NTP). Using virtual timestamp as the attribute for fusion has the added benefit that it is straightforward to map any other attribute of interest in the application to this virtual timestamp (for e.g. iteration space in a grid computation).

3.2 The Fusion Function

The fusion function is a programmer-supplied function that takes as input a set of timestamp correlated items and produces a fused item (with the same timestamp) as output. A fusion function is associated with the channel when created. It is possible to dynamically change the fusion function after channel creation as well.

3.3 Partial Fusion

The architecture provides the flexibility for an application to specify complete or partial fusion. In the latter case, the fusion function is invoked by the system as soon as the minimal set of input data items are available. In addition to or in lieu of specifying a minimal set of input data items to be fused, an application to specify a time interval to wait for input items. The application has the flexibility to specify the desired action upon the expiration of this timer before the minimal input data set is available: invoke the fusion function with available inputs, or report failure. Fused items include meta data indicating the inputs used to generate the item in the case of partial fusion.

3.4 Fusion (Fetch) Policy

Normally, inputs are collected and fused only when a request for the fused item is made. In the case of fusion pipelines this can initiate a back chain of fusion requests. To enhance performance, programmers may specify that items are to be prefetched and fused immediately, once they become available. These eagerly fused items are logically cached in a *prefetch buffer* and moved to the regular fusion buffer for reading when an actual request arrives. (These buffers are logically separate to allow independent capacity management. We assume that the system automatically performs buffer management and reuse. In the Stampede implementation, buffer management is performed using the garbage collection and reference counting mechanisms provided by Stampede.) Scalable implementations (such as ours) will recursively initiate requests of input items in parallel using mechanisms such as bounded thread pools.

3.5 Feedback: Status and Command Registers

Fusion functions need access to status information regarding the incoming fusion network. We have already mentioned that items that result from partial fusion contain meta data describing the constituent inputs. In addition, a failure notification mechanism is required. Special "synthetic" failure items may be placed in the fusion buffer by the runtime system when necessary. We have observed that application-sensor interactions tend to mirror application-device interactions in operating systems. Applications receive data from sensors through a data "register" (the fusion channel). The status of the sensor or fusion pipeline can be interrogated by examining an appropriate "status" register. Similarly, sensors that support a command set (to alter sensor parameters or explicitly activate and deactivate) should be controllable via a "command" register. The specific command set is, of course, device specific but the general device driver analogy seems well-suited to control of sensor networks. Fusion channels provide rudimentary command and status registers.

3.6 Interface Summary

We summarize below the primary calls in our fusion channel interface.

```
fusion_channel    = fusion_create(inputs, fusion_function , buffer_sizes, policy);
fusion_channel_id = fusion_connect (fusion_channel);
item = fusion_get(fusion_channel_id, timestamp, policy, ...);
```

4 Implementation

A distributed implementation of the sensor fusion architecture requires the following capabilities:

- A programming system that associates a virtual timestamp with data items produced by a thread.
- A messaging layer that transports these timestamped data items for access anywhere in the distributed system.
- A mechanism that establishes a correspondence between a virtual time tick and real-time interval.

It turns out that the Stampede programming system [7, 5] has these capabilities and more. Moreover, the overhead of Stampede is also relatively low [8]. Hence we decided to implement the sensor fusion architecture on top of Stampede.

4.1 Stampede Programming System

The programming model of Stampede is simple and intuitive. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through *channels*[1]. Threads can be created to run anywhere in the cluster. Channels can be created anywhere in the cluster and have cluster-wide unique names. Threads can *connect* to these channels for doing input/output via *get/put* operations. A timestamp value is used as a *name* for a data item that a thread puts into or gets from a channel. The runtime system of Stampede takes care of the synchronization and communication inherent in these operations, as well as managing managing the storage for items put into or gotten from the channels.

Every item on a channel is uniquely indexed by a *timestamp*. Typically a thread will *get* an item with a particular timestamp from an input connection, perform some processing on the data in the item, and then *put* an item with that same timestamp onto one of its output connections. Items with the same timestamp in different channels represent various stages of processing of the same input. The runtime system calculates lower bounds for timestamp values of interest to any of the application threads. Using these lower bounds, the runtime system can *garbage collect* the storage space for useless data items on channels [9].

The timestamp associated with an item is an indexing system for data items. For pacing a thread relative to realtime, Stampede provides an API borrowed from the Beehive system [10]. Essentially, a thread can declare realtime interval at which it will re-synchronize with realtime, along with a tolerance and an exception handler. As the thread executes, after each "tick", it performs a Stampede call attempting to synchronize with real time. If it is early, the thread is blocked until that synchrony is achieved. It if is late by more than the specified tolerance, Stampede calls the thread's registered exception handler which can attempt recovery in an application specific manner.

Stampede is implemented as a C runtime library on top of several clustered SMP platforms including DEC Alpha-Digital Unix 4.0 (Compaq Tru64 Unix), x86-Linux, x86-Solaris, and x86-NT. It uses a message-passing substrate called CLF, a low level packet transport layer developed originally at Digital Equipment Corporation's Cambridge Research Lab (which has since become Compaq/HP CRL). CLF provides reliable, ordered, point-to-point packet transport between Stampede address spaces, with the illusion of an infinite packet queue. It exploits

[1] Stampede also provides two cluster-wide data abstractions called *queues* and *registers*. Queues also hold timestamped data items and differ in some semantic properties from channels. Registers provide *full/empty* synchronization semantics for inter-thread signaling and event notification.

shared memory within an SMP, and any available cluster interconnect between the SMPs, including Digital Memory Channel [11], Myrinet [12] (using the Myricom GM library), and Gigabit Ethernet (using UDP).

4.2 Implementation of Sensor Fusion Architecture

We have implemented the fusion architecture in C as a layer on top of the Stampede runtime system. All the buffers (input buffers, fusion buffer, and prefetch buffer) shown in the fusion architecture diagram (Figure 1) are implemented as Stampede channels. Since Stampede channels hold timestamped items, it is a straightforward mapping of the fusion attribute to the timestamp associated with a channel item. The Status and Command registers of the fusion architecture are implemented using the Stampede register abstraction. In addition to these Stampede channels and registers that have a direct relationship to the elements of the fusion architecture, the implementation uses additional Stampede channels and threads. For instance, there is a prefetch thread that prefetches items from the input buffers, fuses them, and places them in the prefetch buffer for potential future requests. Similarly, there is a Stampede channel that stores requests that are currently being processed by the architecture to eliminate duplication of fusion work.

The `create` call from an application thread results in the creation of all the above Stampede abstractions in the address space where the creating thread resides. An application can create any number of fusion channels (modulo system limits) in any of the nodes of the distributed system. A *connect* call from an application thread results in the application thread being connected to the specified fusion channel for getting fused data items. For efficient implementation of the `fusion_get` call, a pool of worker threads is created in each node of the distributed system at application startup. These worker threads are used to satisfy any `fusion_get` requests that originate at this node for any fusion channel. Since data may have to be fetched from a number of input buffers to satisfy the `fusion_get` request, one worker thread is assigned to each input buffer to increase the parallelism for fetching the data items. Once the fetch is complete, the worker thread joins the pool of free threads. Once the requisite input items have been fetched, the fusion function is invoked and the fused item is placed in the fusion buffer. This implementation is performance-conscious in three ways: first, requests for fused items coming from different nodes of the distributed system for the same fusion channel get distributed (no central bottleneck); second, there is no duplication of fusion work for the same fused item from multiple requester; third, fusion work itself is parallelized at each node through the worker threads. All of these implementation features lead to a scalable implementation of the fusion architecture.

As we mentioned earlier, Stampede does automatic reclamation of storage space of data items in channels and queues using a global lowerbound for timestamp values of interest to any of the application threads (which is derived from a per-thread state variable called thread virtual time). The fusion architecture leverages this feature for cleaning up the storage space in its internal data structures (which are built using Stampede abstractions). The trick is to advance the per-thread virtual times of the worker threads at a node to be in synchrony with the other Stampede threads on that node periodically.

5 Sample Application

In this section we present two applications drawn from robotics to illustrate the programming advantage achieved due to the fusion architecture.

5.1 Direction Control for Robot Movement

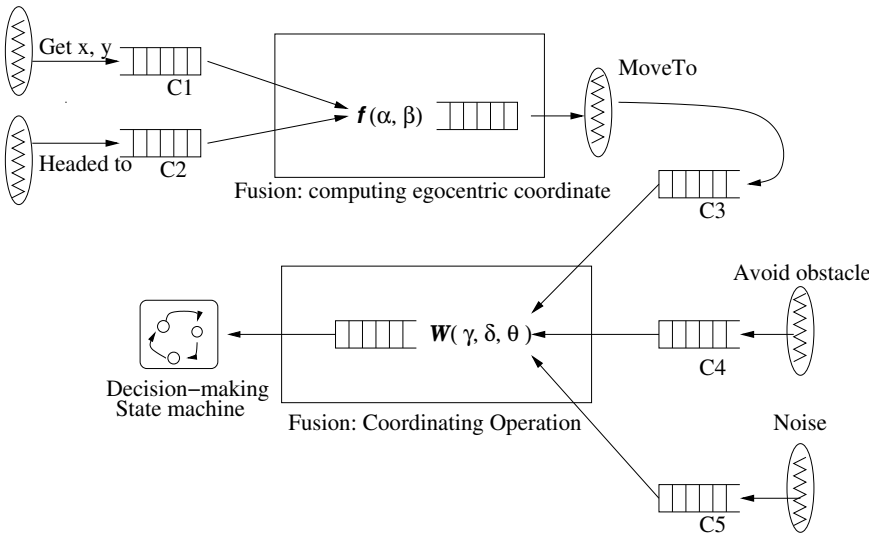

Figure 2. Direction control for robot movement

The first application is a simple one in which a robot has to navigate a terrain to go from a source to destination. Figure 2 shows the pipeline that represents the control loop for this application. The pipeline models how a robot navigates the terrain to move towards its destination. All the stages in the pipeline (shown by threads) are various processing units in a robot. The first stage gathers the robot's global coordinates via one sensor, and the trajectory via another. These two inputs are used to compute the robot's direction of movement in terms of its

ego-centric coordinates[2]. However, the robot has to watch out for obstacles in its way, which are sensed by other sets of sensors. In addition to ego-centric direction and the obstacle information, it is common to introduce a small amount of random noise[3] in the computation. With these three inputs the last stage (move coordination stage) finally determines the direction of motion and forwards the output to a motor controller unit (finite state machine) that initiates the mechanical movement completing the control loop.

As can be seen from the Figure 2, data fusion is a key property of such a pipeline. We now show the pseudo-code for implementing such a pipeline using the fusion architecture API.

At start up two fusion channels are created.

```
fusion_create( &fusion_channel_1,
  &f /* pointer to fusion function */,
  C1, C2, /* input buffers */ ...)

fusion_create( &fusion_channel_2,
  &W /* pointer to fusion function */,
  C3, C4, C5 /* input buffers */ ...)
```

A consumer of a fused item, such as the MoveTo thread (or the state machine) first connects to the relevant fusion channel. The connection returns a connection index which the thread uses subsequently in all calls relating to this channel. The connection is done as follows:

```
fusion_connect(fusion_channel_1, &connection_index)
```

Once a connection is established, the consumer threads can ask for fused items by timestamp.

```
fusion_get (connection_index,
timestamp_requested,
&buffer_to_hold_fused_item
&size_of_fused_item ... )
```

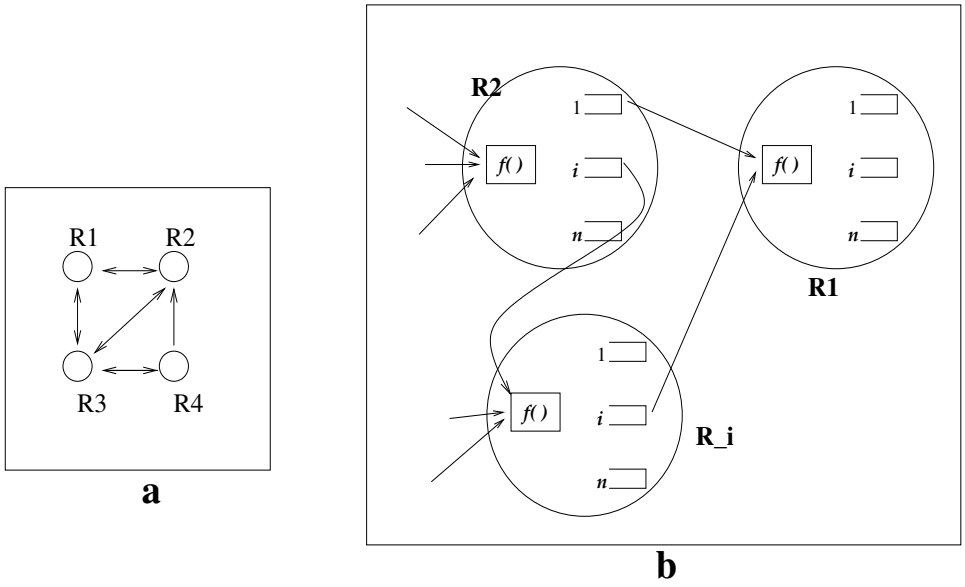

Figure 3. Collaborative robots application

5.2 Collaborative Robot Tasks

The second application is a collaborative robot team. This is another common scenario in many robotics applications; robots collaborate among themselves to accomplish a common task, such as participating in a game or landmark localization. Figure 3 -(a) shows a group of communicating robots. Typically in such applications, a robot R_p tells every other robot R_q about its own estimate on the position of R_q. Let us call this estimate $L_p(q)$. These estimates can be erroneous as the sensors are often inaccurate. R_q tries to estimate the correct location L^q from many possibly inaccurate estimates, by computing a function (often probabilistic) of all the received estimates from other robots: $L^q = \phi(L_1(q), L_2(q) \ldots L_n(q))$. However, links between robots may fail, or introduce indeterminate delays. In such cases a robot may have to compute the estimate on partially available data. Our sensor fusion API provides support for such a scenario. Figure 3 -(b) shows our implementation of such a robot team. Each circle represents a robot that has a set of output buffers (marked 1 to n) and a fusion channel marked by $f()$. The output buffers are fed as input to the fusion channels, the buffer marked 1 goes to R_1 and so on. Depending on the environmental conditions and desired accuracy, the application programmer can choose the number of items desired, and/or a time interval to wait for sensor inputs to be available before fusion. Such parameters can be set at the time of creation of the fusion channel and modified dynamically during the execution as well. The fusion architecture API offers real advantage in such situations, since handling of link indeterminacy is completely

[2]The ego-centric coordinate system is the robot's reference frame.

[3]The random noise prevents the computation from getting stuck in local minima.

removed from the application programmer.

6 Performance

In this section, we report on preliminary performance results of the fusion architecture. The experimental platform for this work is a cluster of SMP nodes running Linux. The hardware consists of 17 Dell 8450 servers each with eight 550MHz Pentium III Xeon CPUs, 2MB of L2 cache per CPU and 4GB of memory per node. The 8450 uses the Intel ProFusion chipset which provides two 64-bit/100MHz system (front-side) busses, one for each bank of four CPUs. The nodes are interconnected with doubled gigabit Ethernet through a dedicated switch. The operating system is Linux with the 2.4.9 kernel. The system scheduler in this kernel is oblivious to the 8450's split system bus. The compiler is GCC version 2.96 with optimization set to -O2.

We carry out the evaluation at two levels. First, we perform a set of micro measurements to (a) quantify the overhead of the fusion_get API (which is built on top of Stampede) by comparing it to an implementation that performs the fusion directly using Stampede channels (i.e. without the fusion library); and (b) quantify the scalability of the fusion architecture. Next, we evaluate the performance with respect to the two applications we discussed in Section 5.

6.1 Micro Measurements

Figure 4 shows the setup for a series of micro experiments. A simple fusion application that spans two nodes is shown. A fusion channel is collocated with a consumer on Node 1. The inputs that need to be fused are produced on Node 0 by producer threads. The consumer calls fusion_get to fuse the data items produced on Node 0. The latency for this operation is measured in this experiment.

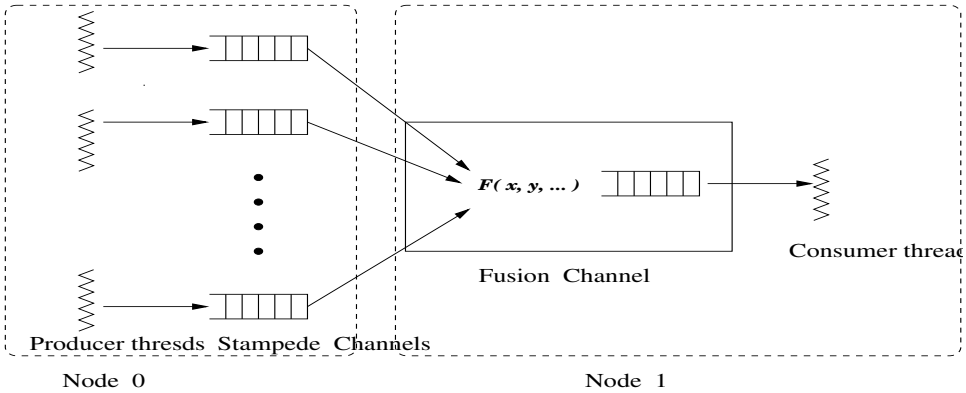

Figure 4. Measuring the latency of fusion API

Figure 5 describes a comparable experimental setup designed using Stampede channels only without the fusion library. This setup performs the exact same fusion application as in Figure 4. The main difference between the two experimental setups is that the fusion library implicitly uses thread parallelism for fetching the inputs for fusion in Figure 4, whereas the straight Stampede implementation in Figure 5 naively uses a single thread that has to sequentially fetch all the inputs before fusing them. We conduct two micro measurements.

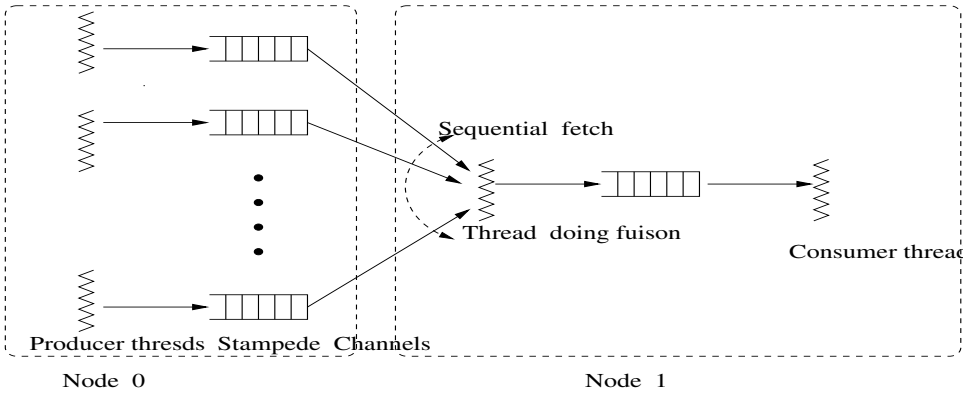

Figure 5. Experiment without fusion library

Experiment 1 : In this experiment we wish to measure the overhead of the fusion architecture. We use only a single input. This helps us quantify the exact overhead of the fusion architecture that has been implemented on top of Stampede. We study the latency of the fusion_get call as a function of increasing data size. This is compared against the scenario in Figure 5 that is implemented without the fusion library. Figure 6 plots the latency in millisecond, against the data size (given in bytes). The latency when using the fusion library is little more than without it. The difference is also plotted in the same graph. For a data size of 1.28MB, the fusion library adds an overhead of 1.7 millisecond which represents a 21% increase in latency over vanilla Stampede.

Experiment 2 In this experiment we wish to measure the performance advantage that the fusion library gives when there are multiple inputs that need to be fused. The main source of expected performance improvement is the thread parallelism used in the fusion library for fetching the inputs in parallel for fusion compared to the naive implementation. In this experiment we keep the data size constant (1 KB) and vary the number of input channels (plotted on the X axis). The latencies (on Y axis) for the two experimental setups are shown in Figure 7. As can be seen as the number of inputs grows, the fusion library version starts performing better. The size of the thread pool was kept at 8 [4]. For 32 input channels, the fusion API has a latency of 14 milliseconds, while the latency without the fusion library is almost 19 milliseconds.

[4]Since all our experiments are carried out on 8-way SMPs, 8 threads give maximum parallelism.

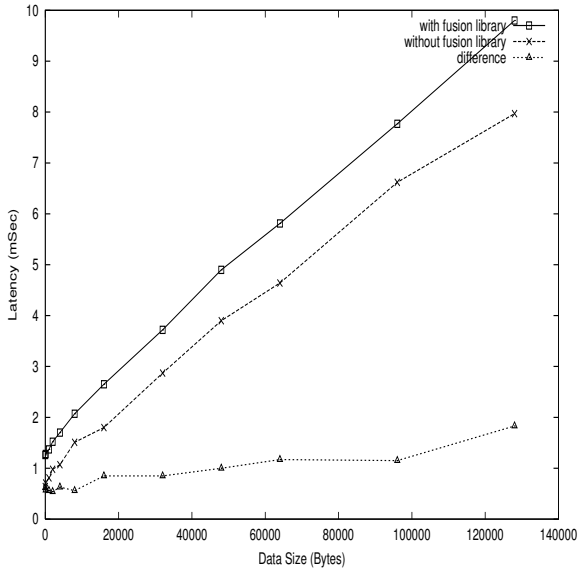

Figure 6. Overhead of fusion library

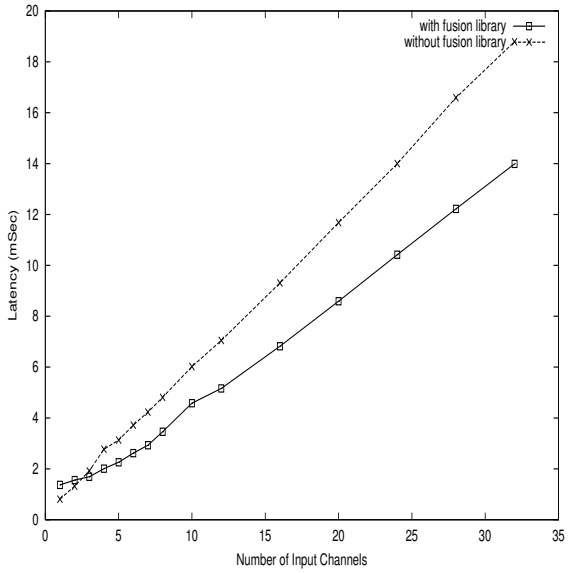

Figure 7. Effect of parallel fetch (Experiment 2)

Experiment 3 This experiment is a slight variation of Experiment 2. The setup is roughly the same except that each producer thread runs on a different node. Figure 8 illustrates the performance figures for this experiment. The X-axis represents number of input channels and the Y-axis represents latency as before. Data size is kept constant at 1KB. In this experiment, we varied number of input channels up to 10 (and not 32, as in Experiment 2) as we were limited by the number of machines. For 10 input channels, the fusion library latency is 4.5 milliseconds,

while the naive implementation latency is 6.5 milliseconds. This experiment confirms that our system gives better performance even when all the producers are in different nodes.

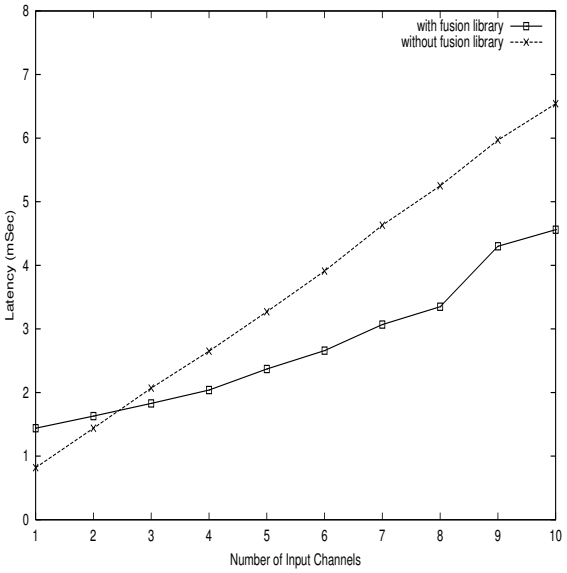

Figure 8. Experiment 3, Data size=1KB

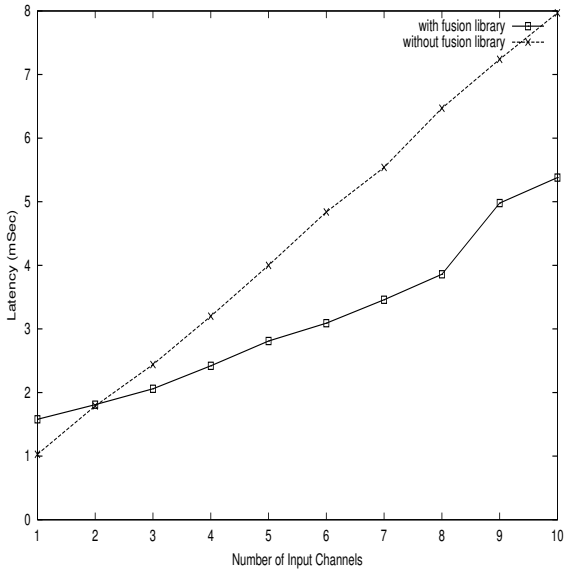

Figure 9. Experiment 3, Data Size = 2KB

Next, we estimate the expected improvement in scalability as follows. The total latency of obtaining a fused item is the sum of time units required to fetch and apply the fusion function. Fusion function time includes the computational time of the function, local copy into the fusion buffer and then final hand off to the consumer thread. Therefore, $T_{latency} = T_{fetch} + T_{fusion}$. The fusion library improves over the naive implementation by reducing T_{fetch} through parallelism. We can quantify how T_{fetch} scales up in both cases by observing the rate at which the latency increases as the number of input channels increases. It is reasonable to assume that T_{fusion} is the same for both cases. The differential $\Delta T_{latency}/\Delta n = \Delta T_{fetch}/\Delta n + \Delta T_{fusion}/\Delta n$ quantifies the rate at which the latency increases with increasing number of input channels. For n input channels, the fetch time taken by the naive implementation is $T_{fetch}^{(naive)} = n.t_{get}$ where t_{get} is the time for a thread to get an item from a Stampede channel. The differential $\Delta T_{fetch}^{(naive)}/\Delta n = t_{get}$ is the average time taken to get one item. For the parallelized implementation of the fusion library, $T_{fetch}^{(par)} = t_{get} + n.t_{ov}$, where t_{ov} denotes a unit of synchronization overhead that is incurred due to worker threads acting in parallel [5]. Hence the differential in this case, $\Delta T_{fetch}^{(par)}/\Delta n = t_{ov}$

[5]It is reasonable to assume that the total synchronization overhead increases linearly with the number of parallel threads.

is proportional to the unit of synchronization overhead, which for a given hardware and systems set up remains independent of the data size (in fact a constant). So, coming back to the differential of latency, we get

$$\frac{\Delta T_{latency}^{(naive)}}{\Delta n} = t_{get} + \frac{\Delta T_{fusion}}{\Delta n}$$

$$\frac{\Delta T_{latency}^{(par)}}{\Delta n} = t_{ov} + \frac{\Delta T_{fusion}}{\Delta n}$$

Although dependent upon the data size, in this particular example, the differential of fusion is small, since fusion is completely local to one machine and involves only in memory copy. t_{get} on the other hand is the time delay for data transport across machines. While t_{ov} is independent of the data size, t_{get} is not. To sum up, both the components in the differential of the latency in the naive implementation are increasing functions of data size. In parallel implementation only one component depends on the data size. Hence, the parallel implementation should scale up better than the naive implementation. This is illustrated by a comparative observation of the latency variation for data size 1KB and 2 KB. Figure 9 plots the latency for data size of 2KB. The slope of the line representing the fusion library remains approximately the same in both Figure 8 and Figure 9. To make this observation clear, in Figure 8, the latency (with fusion library) varies from 1.4 milliseconds to 3.4 milliseconds as the number of input channels varies from 1 to 8, thus giving an average slope of 0.29 [6]. In Figure 9, in a similar way the slope is approximately 0.31. Without the fusion library, the slope for Figure 8 is 0.63, while in Figure 9, i.e., data size 2KB, the slope is 0.78. This comparison shows that the fusion library scales better than a naive implementation.

6.2 Application Performance

In this section we quantify some of the application characteristics that benefits from our infrastructure. In particular, we consider the simulation of collaborative robots discussed in 5.2. The q-th robot (R_q) computes a function $\phi(L_1(q), L_2(q) \ldots L_n(q))$ to estimate L^q. We estimate the average time t_{avg} taken at every step for this computation, i.e., $t_{avg} = \frac{1}{n}\Sigma_{q=1}^{n} t_q$, where t_q is the time taken by R_q to compute $\phi()$. We simulate two different situations: (i) Each robot waits in a blocking mode for the estimates from all other robots and then computes ϕ, and (ii) Wait a definite amount of time, and then compute the estimate, even under partial availability of the data.

[6] There is a kink in the graph at number of input channels = 8. This is because the thread-pool size is kept at 8 on an 8-way SMP. For more than 8 input channels, there is some serialization and a resulting increase in the latency.

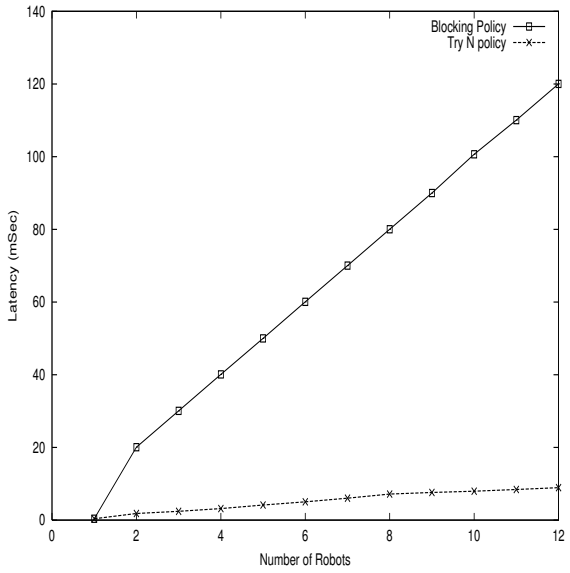

Figure 10. Collaborative robot application

Figure 10 plots the average time for this estimation in both the cases. The X-axis represents number of robots and the Y-axis plots t_{avg}. The first case is indicated by "Blocking Policy" and the second by "TryN Policy". Clearly the second case performs much better. A naive implementation of this application will usually block for all the inputs. However, given a suitable algorithm to make a good estimate even with partial data, the modifications necessary to enable this naive implementation to work with partial availability of data (which is indeterminate) is quite complicated. The intent of this experiment is to bring out the fact that simply using the fusion API leads to performance improvement, the programmer does not need to handle the vagaries of partial availability of data.

7 Related Work

Our work focuses on two aspects of sensor data fusion. First, we provide a clean abstraction for applying arbitrary fusion functions to fused data streams. Second, we offer an efficient, scalable implementation of data correlation and aggregation. While the published literature relating to sensor data fusion is vast, we have found no other works focusing on these particular architectural issues.

Perhaps the most closely related work outside of the fusion literature addresses the implementation of scalable network I/O in UNIX platforms. Early work by Banga and Mogul [35] observed that servers using the UNIX select (poll) system call scaled very poorly when the number of managed connections increased to hundreds or thousands. This work was extended by others [36], [37], [38], [18], [19] optimizations and improvements were developed including BSD kqueues [21], Sun's /dev/poll [20], and POSIX real-time signal-based implementations

[23] of select. Sensor fusion can be viewed as an "AND-based select", with data required from all or most of the inputs. The traditional select is "OR-based" and returns as soon as data is available on any input. One of the improvements in the /dev/poll implementation allows the specification and modification of the file descriptor set independently of the actual blocking select call. This reduces the overhead required to parse and verify the file descriptor set on each select. Fusion channels provide a similar benefit by specifying the inputs at channel creation time. (It might be useful to extend our API to allow the dynamic, incremental modification of the set of input channels.)

At its broadest, sensor fusion involves gathering observations of the world and drawing inferences from these observations [17]. A great many computational and perceptual processes fit into this framework. Traditionally, sensor fusion was developed in military applications involving target identification and acquisition. Today, military data fusion [17], [16] is a highly sophisticated field of effort. In 1985, the U.S. Joint Directors of Laboratories Data Fusion Group developed the so-called JDL Data Fusion model [34] to facilitate communication among researchers. The model was revised and generalized in 1998. This functional model describes four assessment levels involved in fusion systems including: sub-object, object, situation, and impact assessment. A fifth level is a feedback refinement process. A unified process model simplifies these levels into phases: observe, orient, decide, and act. Our work is but a small component of the first level or phase in both models. Subsequent levels and phases involve sophisticated estimation and refinement models, feature extraction, and rule-based decision-making processes.

Outside of military applications, sensor fusion is increasingly deployed in commercial and academic settings including robotics, factory control, fault-monitoring, medical diagnosis, environmental monitoring, and wireless networks. Computer scientists have begun exploring a variety of specific topics within the broad sensor fusion canvas. Some of these works describe system architectures for fusion but they tend to be ad hoc in nature. A sampling of recent works follows.

Marzullo [22] gave a formal model showing how overlapping sensors can be fused to form a single fault-tolerant abstract sensor and showed a relationship between agreement in sensor networks and distributed consensus. Location tracking is a popular application of sensor networks (people, objects, robots, etc.) and a variety of techniques and infrastructures for this problem have been proposed [26], [25]. Power-management is critical in wireless environments and synchronization is critical for sensor fusion. Researchers have begun looking at power-efficient synchronization protocols [15]. Data management facilities are required to manage both sensor attributes as well as the data they produce [33], [24]. Environmental sensing is a well-established field including weather [30] and

habitat monitoring [27]. Directed diffusion has been proposed as a content-based form of data routing for sensor networks in which nodes broadcast interests that effect subsequent routing decisions [32]. Researchers from USC and UCLA [13] have developed a clever naming mechanism for attribute-based naming in dense sensor networks that utilizes directed diffusion. Park, Savvides and Srivastava [14] have developed models and techniques based on the ns-2 simulator for modeling MEMS-based sensor networks. Researchers at UC Berkeley in conjunction with the SmartDust program have begun exploring the special security requirements of sensor networks [31]. Novel recent applications include smart spaces for children [28] and biomedical sensor implants [29].

8 Conclusions

Fusion Channels provide the appropriate abstractions for the emerging class of multi-sensor data fusion applications. Our genric API makes programming easier and our implementation provides useful optimizations such as parallellism and prefetching. We have implemented a prototype on top of Stampede, a runtime library that supports a distributed programming model, offering facilities such as temporal correlation of data and automatic garbage collection. Through micro measurements we have established that our infrastructures incurs low overhead. In fact, compared to naively written fusion routines, it offers performance improvement and scales better.

Fusion Channels are being used at Georgia Tech to develop a variety of applications. We are also adding more capabilities to the programming interface. These include providing QoS mechanisms, event notification to the consumer and handling failures.

References

[1] I. Essa. *Ubiquitous Sensing for Smart and Aware Environments: Technologies towards the building of an Aware Home* DARPA/NSF/NIST Workshop on Smart Environments, July 1999 http://www.cc.gatech.edu/fce/ahri/publications/index.html

[2] D. Mackenzie, R.Arkin, J. Cameron. *Multiagent Mission Specification and Execution.* Autonomous Robots, 4(1): 29-57, 1997

[3] Ashley W. Stroupe, Kevin Sikorski, and Tucker Balch. *Constraint-Based Landmark Localization.* Proceedings of 2002 RoboCup Symposium (to appear), 2002.

[4] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, U. Ramachandran. *Scheduling Constrained Dynamic Applications on Clusters.* Supercomputing '99, December 1999.

[5] S. Adhikari, A. Paul, U. Ramachandran. *D-Stampede: Distributed Programming System for Ubiquitous Computing*. In Proceedings of the 22nd International Conf. on Distributed Computing Systems, Vienna, July-2002

[6] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg and L. Kontothanassis. *Stampede: A Programming System for Emerging Scalable Interactive Multimedia Applications*. The 11th International Workshop on Languages and Compilers for Parallel Computing, 98.

[7] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg and K. Knobe. *Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications*. The Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 99.

[8] A. Paul, N. Harel, S. Adhikari, B. Agarwalla, U. Ramachandran, Ken Mackenzie *Interaction between Stampede Runtime and Operating Systems.* College of Computing, Georgia Tech, Technical Report Tech GIT-CC-02-47.

[9] R. S. Nikhil, and U. Ramachandran. *Garbage Collection of Timestamped Data in Stampede*. Principles of Distributed Computing, Jul 00 .

[10] A. Singla, U. Ramachandran and J. Hodgins, *Temporal Notions of Synchronization and Consistency in Beehive*, The Nineth Annual ACM Symposium on Parallel Algorithms and Architectures, Jun 97.

[11] R. Gillett, M. Collins, and D. Pimm. *Overview of Network Memory Channel for PCI*. The IEEE Spring COMPCON '96, Feb. 96.

[12] N. J. Boden, D. Cohen, R. E. Felderman, A.E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. *Myrinet: A Gigabit-per-Second Local Area Network*. IEEE Micro, February 95 p: 29–36.

[13] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin and D. Ganeshan. *Building Efficient Wireless Sensor Networks with Low-level Naming*. Proceedings of the 18th Symposium on Operating Systems Principles, p: 186-201, 2001.

[14] S. Park, A. Savvides and M.B. Srivastava. *Simulating Networks of Wireless Sensors*. Proceedings of the 33rd Conference on Winter Simulation, p: 1330-1338, 2001.

[15] K. Romer *Time Synchronization in Ad Hoc Networks*. The ACM Symposium on Mobile Ad Hoc Networking Computing (MobiHOC '01), Long Beach, CA, Oct 2001.

[16] R. Brooks and S. Iyengar. *Multi-sensor Fusion: Fundamentals and Applications with Software*. Prentice-Hall, 1998.

[17] D. Hall and J. Llinas. *Handbook of Multisensor Data Fusion*. CRC Press, 2001.

[18] M. Welsh, D. Culler and E. Brewer *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*. Proceedings of the 18th Symposium on Operating Systems Principles, Oct 2001.

[19] C. Poellabauer, K. Schwan and R. West *Coordinated CPU and Event Scheduling for Distributed Multimedia Applications*. ACM Multimedia, Ottawa, Ontario, Canada Oct 2001.

[20] B. Chapman. *Polling Made Efficient*. http://soldc.sun.com/articles/polling_efficient.html.

[21] J. Lemon *Kqueue: A Generic and Scalable Event Notification Facility* Proceedings of the FREENIX Track: USENIX Annual Technical Conference, Boston, MA, June 2001

[22] K. Marzullo *Tolerating Failures of Continuous-Valued Sensors* ACM Transactions on Computer Systems, Vol. 8, No. 4, p: 284-304, Nov 1990.

[23] N. Provos and C. Lever *Scalable Network I/O in Linux* Proceedings of the FREENIX Track: USENIX Annual Technical Conference, San Diego, CA, June 2000

[24] S. Madden, M. Shah, J.M. Hellerstein and V. Raman. *Continuously Adaptive Continuous Queries over Streams*. In ACM SIGMOD, June 2002

[25] A.M. Ladd, K. E. Bekris, A. Rudys, G. Marceau, L.E. Kavaraki, D.S. Wallach. *Robotics-Based Location Sensing using Wireless Ethernet*. Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking (MOBICOMM'02), Atlanta, GA, Sep 2002.

[26] B. Horling, R. Vincet, R. Mailler, J. Shen, R. Becker, K. Rawlins and V. Lesser *Distributed Sensor Network for Real Time Tracking*. AGENTS, May 2001.

[27] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler and J. Anderson. *Wireless Sensor Networks for Habitat Monitoring*. ACM International Workshop on Wireless Sensor Networks and Applications, Atlanta, GA, Sep 2002.

[28] M. Srivastava, R. Muntz and M. Potkonjak. *Smart Kindergarten: Sensor-based Wireless Networks for Smart Developmental Problem-solving Environments*. Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (SIGMOBILE'01), p: 132-138 Rome, Italy, July 2001

[29] L. Schwiebert, S.K.S. Gupta and J. Weinmann. *Research Challenges in Wireless Networks of Biomedical Sensors* Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (SIGMOBILE'01), p: 151-165, Rome, Italy, July 2001

[30] D.C. Steere, A. Baptista, D. NcNamee, C. Pu and J. Walpole *Research Challenges in Environmental Observation and Forecasting Systems* Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MOBICOMM'00), p: 292-299, Aug 2000.

[31] A. Perrig, R. Szewczyk, V. Wen, D. Cullar and J.D. Tygar. *SPINS: Security Protocol for Sensor networks*. Proceedings of Mobile Computing and Networking (MOBICOM), p: 189-199, 2001.

[32] C. Intanagonwiwat, R. Govindan and D. Estrin. *Directed Diffusion: a Scalable and Roboust Communication Paradigm for Sensor Networks*. Proceedings of the Sixth Annual Conference on Mobile Computing and Networking, p: 56-67, Boston, MA, 2000.

[33] J. Eisenstein, S. Ghandeharizadeh, C. Shahabi, G. Shanbhag and R. Zimmermann. *Alternative Representations and Abstractions for Moving Sensors Databases* ACM Conference on Information and Knowledge Management (CIKM), Atlanta, GA, Nov 2001.

[34] A.N. Steinberg and C.L. Bowman. *Revision to the JDL Data Fusion Model*. In Handbook of Multisensor Data Fusion, Hall and Llinas, CRC Press, 2001.

[35] G. Banka and J. Mogul. *Scalable Kernel Performance for Internet Servers under Realistic Loads*. In USENIX Technical Conference, June 1998.

[36] G. Banka, J. Mogul and P. Druschel. *A Scalable and Explicit Event Delivery Mechanism for UNIX*. In USENIX Technical Conference, June 1999

[37] A. Chandra and D. Mosberger. *Scalability of Linux Event-dispatch Mechanism*. Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2001

[38] T.Brecht and M. Ostrowski. *Exploring the Performance of Select-based Internet Servers* Technical Report, HPL-2001-314, Internet Systems and Storage Laboratory, HP Laboratories, Palo Alto, November 28th 2001