DFuse: A Framework for Distributed Data Fusion *

Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin,
Phillip Hutto, Arnab Paul, and Umakishore Ramachandran

{rajnish, wolenetz, bikash, espress, pwh, arnab, rama}@cc.gatech.edu

College of Computing
Georgia Institute of Technology

ABSTRACT

Simple in-network data aggregation (or fusion) techniques for sensor networks have been the focus of several recent research efforts, but they are insufficient to support advanced fusion applications. We extend these techniques to future sensor networks and ask two related questions: (a) what is the appropriate set of data fusion techniques, and (b) how do we dynamically assign aggregation roles to the nodes of a sensor network ? We have developed an architectural framework, *DFuse*, for answering these two questions. It consists of a data fusion API and a distributed algorithm for energy-aware role assignment. The fusion API enables an application to be specified as a coarse-grained dataflow graph, and eases application development and deployment. The role assignment algorithm maps the graph onto the network, and optimally adapts the mapping at run-time using role migration. Experiments on an iPAQ farm show that the fusion API has low-overhead, and the role assignment algorithm with role migration significantly increases the network lifetime compared to any static assignment.

Categories and Subject Descriptors : D.4.7 [Operating Systems]: Organization and Design – Distributed Systems, and Embedded Systems.

General Terms : Algorithms, Design, Management, Measurement.

Keywords : Sensor Network, In-network aggregation, Data fusion, Role assignment, Energy awareness, Middleware, Platform.

*The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, HP/Compaq Cambridge Research Lab, the Yamacraw project of the State of Georgia, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'03, November 5–7, 2003, Los Angeles, California, USA.
Copyright 2003 ACM 1-58113-707-9/03/0011 ...$5.00.

1. INTRODUCTION

With advances in technology, it is becoming increasingly feasible to put a fast processor on a single small sensor along with a sizable memory and a radio transceiver. There is an ever-evolving continuum of sensing, computing, and communication capabilities from smartdust, to sensors, to mobile devices, to desktops, to clusters. With this evolution, capabilities are moving from larger footprint to smaller footprint devices. For example, tomorrow's *mote* will be comparable in resources to today's mobile devices; and tomorrow's mobile devices will be comparable to current desktops. These developments suggest that future sensor networks may well be capable of supporting applications that require resource-rich support today. Examples of such applications include streaming media, surveillance, image-based tracking and interactive vision. Many of these *fusion applications* share a common requirement, namely, hierarchical *data fusion*, i.e., applying a synthesis operation on input streams.

This paper focuses on challenges involved in supporting fusion applications in *wireless ad hoc sensor networks* (WASN). Developing fusion applications is challenging in general because of the time-sensitive nature of the fusion operation, and the need for synchronization of the data from multiple streams. Since the applications are inherently distributed, they are typically implemented via distributed threads that perform fusion in a hierarchical manner. Thus, the application programmer has to deal with thread management, data synchronization, buffer handling, and exceptions (such as time-outs while waiting for input data for a fusion function) - all in a distributed fashion. WASN add another level of complexity to such application development due to the scarcity of power in the individual nodes [5]. In-network aggregation and power-aware routing are techniques to alleviate the power scarcity of WASN. While the good news about fusion applications is that they inherently need in-network aggregation, a naive placement of the fusion functions on the network nodes will diminish the usefulness of in-network fusion, and reduce the longevity of the network (and hence the application). Thus, managing the placement (and dynamic relocation) of the fusion functions on the network nodes with a view to saving power becomes an additional responsibility of the application programmer. Dynamic relocation may be required either because the remaining power level at the current node is going below threshold, or to save the power consumed in the network as a whole by reducing the total data transmission. Supporting the relocation of fusion functions at run-time has all the traditional challenges of process migration [15].

We have developed *DFuse*, an architecture for programming fusion applications. It supports distributed data fusion with automatic management of fusion point placement and migration to optimize a given cost function (such as network longevity). Using the DFuse framework, application programmers need only implement the fusion functions and provide the dataflow graph (the relationships of fusion functions to one another, as shown in Figure 1). The fusion API in the DFuse architecture subsumes issues such as data synchronization and buffer management that are inherent in distributed programming.

The main contributions of this work are summarized below:

1. Fusion API: We design and implement a rich API that affords programming ease for developing complex sensor fusion applications. The API allows any synthesis operation on stream data to be specified as a fusion function, ranging from simple aggregation (such as min, max, sum, or concatenation) to more complex perception tasks (such as analyzing a sequence of video images). This is in contrast to current in-network aggregation approaches [11, 8, 6] that allow only limited types of aggregation operations as fusion functions.

2. Distributed algorithm for fusion function placement and dynamic relocation: There is a combinatorially large number of options for placing the fusion functions in the network. Hence, finding an optimal placement that minimizes communication is difficult. Also, the placement needs to be re-evaluated quite frequently considering the dynamic nature of WASN. We develop a novel heuristic-based algorithm to find a *good* (according to some predefined cost function) mapping of fusion functions to the network nodes. The mapping is re-evaluated periodically to address dynamic changes in nodes' power levels and network behavior.

3. Quantitative evaluation of the DFuse framework: The evaluation includes micro-benchmarks of the primitives provided by the fusion API as well as measurement of the data transport in a tracker application. Using an implementation of the fusion API on a wireless iPAQ farm coupled with an event-driven engine that simulates the WASN, we quantify the ability of the distributed algorithm to increase the longevity of the network with a given power budget of the nodes.

The rest of the paper is structured as follows. Section 2 analyzes fusion application requirements and presents the DFuse architecture. In Section 3, we describe how DFuse supports distributed data fusion. Section 4 explains a heuristic-based distributed algorithm for placing fusion points in the network. This is followed by implementation details of the framework in Section 5 and its evaluation in Section 6. We then compare our work with existing and other ongoing efforts in Section 7, present some directions for future work in Section 8, and conclude in Section 9.

2. DFUSE ARCHITECTURE

This section presents the DFuse architecture. First, we explore target applications and execution environments to identify the architectural requirements. We then describe the architecture and discuss how it is to be used in developing fusion applications.

2.1 Target Applications and Execution Environment

DFuse is suitable for applications that apply hierarchical fusion functions (input to a fusion function may be the output of another fusion function) on time-sequenced data items. A fusion operation may apply a function to a sequence of stream data from a single source, from multiple sources, or from a set of sources and other fusion functions.

DFuse accepts an application as a task graph, where a vertex in the task graph can be one of *data source*, *data sink*, or *fusion point*. A data source represents any data producer, such as a sensor or a standalone application. DFuse assumes that data sources are known at query time (when the user specifies the application task graph). A data sink is an end consumer, including a human in the loop, an application, an actuator, or an output device such as a display. Intermediate fusion points perform application-specific processing on streaming data. Thus, an application is a directed graph, with the data flow (i.e. producer-consumer relationships) indicated by the directionality of the associated edge between any two vertices.

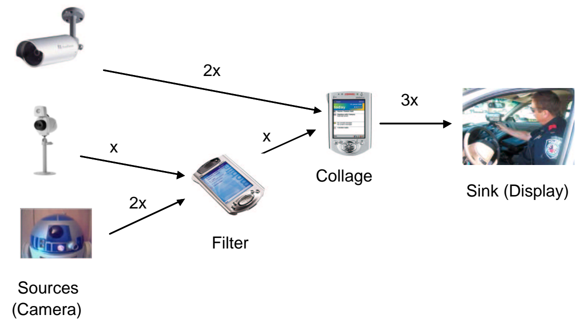


Figure 1: An example tracking application that uses distributed data fusion. Here, *filter* and *collage* are the two fusion points taking inputs from cameras, and the face recognition algorithm is running at the sink. Edge labels indicate relative (expected) transmission rates of data sources and fusion points.

For example, Figure 1 shows a task graph for a tracking application. The filter fusion function selects images with some interesting properties (e.g. rapidly changing scene), and sends the compressed image data to the collage function. Thus, the filter function is an example of a fusion point that does data contraction. The collage function uncompresses the images coming from possibly different locations. It combines these images and sends the composite image to the root (sink) for further processing. Thus, the collage function represents a fusion point that may do data expansion.

DFuse is intended for deployment in a heterogeneous ad hoc sensor network environment. However, DFuse cannot be deployed in current sensor networks given the limited capabilities available in sensor node prototypes such as Berkeley motes [7]. But, as we add devices with more capabilities to the sensor network, or improve the sensor nodes themselves, more demanding applications can be mapped onto such networks and DFuse provides a flexible fusion API for such a deployment. As will become clear in later sections, DFuse handles the dynamic nature of such networks by employing a resource-aware heuristic for placing the fusion points in the network.

DFuse assumes that any node in the network is reachable from any other node. Further, DFuse assumes a routing layer that exposes hop-count information between any two nodes in the network. Typically, such support can be provided by a separate layer that supports a routing protocol for ad hoc networks, like Dynamic Source Routing (DSR) [10], and exposes an interface to query the routing information.

2.2 Architecture Components

Figure 2(A) shows a high-level view of the DFuse architecture that consists of two main runtime components: *fusion module* and *placement module*. The fusion module implements the fusion API used in the development of the application. The fusion module interacts with the placement module to determine a *good* mapping of the fusion functions to the sensor nodes given the dynamic state of the network and the application behavior. These two components constitute the runtime support available in each node of the network.

Figure 2(B) shows the internal structure of the fusion module. Details of the fusion module are discussed in section 3. The modules that implement resource monitoring and routing are external to the DFuse architecture. These modules help in the evaluation of cost functions that is used by the placement module in determining a *good* placement of fusion functions.

Launching an Application and Network Deployment

An application program consists of two entities: a *task graph*, and the code for the *fusion functions* that need to be run on the different nodes of the graph. DFuse automatically generates the glue code for instantiating the task graph on the physical nodes of the network. DFuse also shields the application programmer from deciding the placement of the task graph nodes in the network.

Launching an application is accomplished by presenting the task graph and the fusion codes to DFuse at some designated node, let us call it the *root* node. Upon getting this launch request, the placement module of DFuse at the root node starts a distributed algorithm for determining the best placement (details to be presented in Section 4) of the fusion functions. The algorithm maps the fusion functions of the task graph onto the physical network subject to some cost function. In this resulting overlay network, each node knows the fusion function (if any) it has to run as well as the sources and sinks that are connected to it. The resulting overlay network is a directed graph with source, fusion, and sink nodes (there could be cycles since the application may have feedback control). The application starts up with the sink nodes running their respective codes, resulting in the transitive launching of the codes in the intermediate fusion nodes and eventually the source nodes. Cycles in the overlay network are handled by each node remembering if a launch request has already been sent to the nodes that it is connected to.

The role of each node in the network can change over time due to both the application dynamics as well as health of the nodes. The placement module at each node performs periodic re-evaluation of its health and those of its neighbors to determine if there is a better choice of placement of the fusion functions. The placement module requests the fusion module to affect any needed relocation of fusion functions in the network. Details of the placement module are forthcoming in Section 4.

The fusion module at each node of the network retrieves the fusion function(s) to be launched at this node. It is a space-time trade-off to either retrieve a fusion function on-demand or store the code corresponding to all fusion functions at every node of the network. The latter design choice will enable quick launching of a fusion function at any node while increasing the space need at each node.

3. DISTRIBUTED DATA FUSION SUPPORT

DFuse utilizes a package of high-level abstractions for supporting fusion operations in stream-oriented environments. This package, called *Fusion Channels*, is conceptually language and platform independent.

Data fusion, broadly defined, is the application of an arbitrary transformation to a correlated set of inputs, producing a "fused" output item. In streaming environments, this is a continuous process, producing an output stream of fused items. As mentioned previously, such transformations can result in the expansion, contraction, or *status quo* in the data flow rate after the fusion. Note that a filter function, taking a single input stream and producing a single output stream, is a special case of such a transformation. We assume that fusion outputs can be shared by multiple consumers, allowing "fan-out" from a fusion point, but we disallow a fusion point with two or more distinct output streams. Fusion points with distinct output streams can be easily modeled as two separate fusion points with the same inputs, each producing a single output. Note that the input of a fusion point may be the output of another fusion point, creating fusion pipelines or trees. Fusion computations that implement control loops with feedback create cyclic fusion graphs.

The Fusion Channels package aims to simplify the application of programmer-supplied transformations to correlated sets of input items from sequenced input streams, producing a (possibly shared) output stream of "fused items." It does this by providing a high-level API for creating, modifying, and manipulating fusion points that subsumes certain recurring concerns (failure, latency, buffer management, prefetching, mobility, sharing, concurrency, etc.) common to fusion environments such as sensor networks. Only a subset of the capabilities in the Fusion Channels package are currently used by DFuse.

The fusion API provides capabilities that fall within the following general categories:

Structure management: This category of capabilities primarily handles "plumbing" issues. The fundamental abstraction in DFuse that encapsulates the fusion function 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 may come from the node that hosts the channel or from a remote node. 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 by supplying some wildcard specifiers supported by the API (such as *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. Fusion channels have a fixed capacity specified at creation time. Finally, inputs to a fusion channel can themselves be fusion

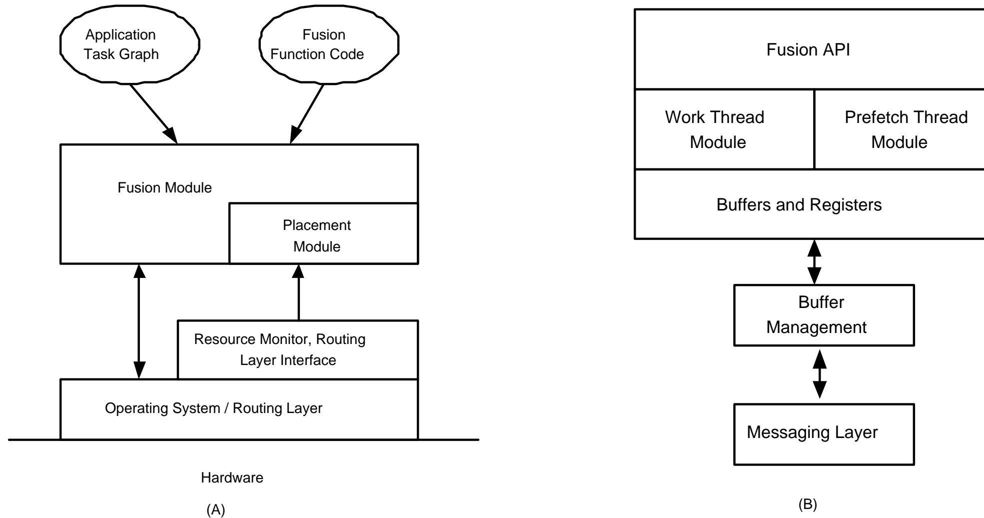


Figure 2: (A) DFuse architecture - a high-level view per node. (B) Fusion module components.

channels, creating fusion networks or pipelines.

Correlation control: This category of capabilities primarily handles specification and collection of "correlation sets" (related input items supplied to the fusion function). Fusion requires identification of a set of correlated input items. A simple scheme is to collect input items with identical application-specified sequence numbers or virtual timestamps (which may or may not map to real-time depending on the application). Fusion functions may declare whether they accept a variable number of inputs and, if so, indicate bounds on the correlation set size. Correlation may involve collecting several items from each input (for example, a time-series of data items from a given input). Correlation may specify a given number of inputs or correlate all arriving items within a given time interval. Most generally, correlation can be characterized by two programmer-supplied predicates. The first determines if an arriving item should be added to the correlation set. The second determines if the collection phase should terminate, passing the current correlation set to the programmer-supplied fusion function.

Computation management: This category of capabilities primarily handles the specification, application, and migration of fusion functions. The fusion function is a programmer-supplied code block 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, to modify the set of inputs, and to migrate the fusion point. Using a standard or programmer-supplied protocol, a fusion channel may be migrated on demand to another node of the network. This feature is essential for supporting the role assignment functionality of the placement module. Upon request from an application, the state of the fusion channel is packaged and moved to the desired destination node by the fusion module. The fusion module handles request forwarding for channels that have been migrated.

Memory Management: This category of capabilities primarily handles caching, prefetching, and buffer management. Typically, inputs are collected and fused (on-demand) when a fused item is requested. For scalable performance, input items are collected (requested) in parallel. Requests on fusion pipelines or trees initiate a series of recursive requests. To enhance performance, programmers may request items to be prefetched and cached in a *prefetch buffer* once inputs are available. An aggressive policy prefetches (requests) inputs on-demand from input fusion channels. Buffer management deals with sharing generated items with multiple potential consumers and determining when to reclaim cached items' space.

Failure/latency handling: This category of capabilities primarily allows the fusion points to perform partial fusion, i.e. fusion over an incomplete input correlation set. It deals with sensor failure and communication latency that are common, and often indistinguishable, in sensor networks. Fusion functions capable of accepting a variable number of input items may specify a timeout on the interval for correlation set collection. Late arriving items may be automatically discarded or included in subsequent correlation sets. If the correlation set contains fewer items than needed by the fusion function, an error event occurs and a programmer-supplied error handler is activated. Error handlers and fusion functions may produce special *error items* as output to notify downstream consumers of errors. Fused items include meta-data indicating the inputs used to generate an item in the case of partial fusion. Applications may use the structure management API functions to remove the faulty input if necessary.

Status and feedback handling: This category of capabilities primarily allows interaction between fusion functions and data sources such as sensors that supply status information and support a command set (for example, activating a sensor or altering its mode of operation - such devices are often a combination of a sensor and an actuator). We have observed that application-sensor interactions tend to mirror application-device interactions in operating systems. Sources such as sensors and intermediate fusion points report their status via a "status regis-

ter[1]." Intermediate fusion points aggregate and report the status of their inputs along with the status of the fusion point itself via their respective status registers. Fusion points may poll this register or access its status. 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.

4. FUSION POINT PLACEMENT

DFuse uses a distributed role assignment algorithm for placing fusion points in the network. Role assignment is a mapping from a fusion point in an application task graph to a network node. The distributed role assignment algorithm is triggered at the root node. The inputs to the algorithm are an application task graph (assuming the source nodes are known), a cost function, and attributes specific to the cost function. The output is an overlay network that optimizes the role to be performed by each node of the network. The "goodness" of the role assignment is with respect to the input cost function.

A network node can play one of three roles: *end point (source or sink)*, *relay*, or *fusion point* [3]. An end point corresponds to a data source or a sink. The network nodes that correspond to end points and fusion points may not always be directly reachable from one another. In this case, data forwarding relay nodes may be used to route messages among them. The routing layer (Figure 2) is responsible for assigning a relay role to any network node. The role assignment algorithm assigns only the fusion point roles.

4.1 Placement Requirements in WASN

The role assignment algorithm has to be aware of the following aspects of a WASN:

Node Heterogeneity: A given node may take on multiple roles. Some nodes may be resource rich compared to others. For example, a particular node may be connected to a permanent power supply. Clearly, such nodes should be given more priority for taking on transmission-intensive roles compared to others.

Power Constraint: A role assignment algorithm should minimize data communication since data transmission and reception expend more power than computation activities in wireless sensor networks [7]. Intuitively, since the overall communication cost is impacted by the location of data aggregators, the role assignment algorithm should seek to find a suitable placement for the fusion points that minimizes data communication.

Dynamic Behavior: There are two sources of dynamism in a WASN. First, the application may exhibit dynamism due to the physical movement of end points or change in the transmission profile. Second, there could be node failures due to environmental conditions or battery drain. So far as the placement module is concerned, these two conditions are equivalent. In either case, the algorithm needs to find a new mapping of the task graph onto the available network nodes.

[1] A register is a communication abstraction with processor register semantics. Updates overwrite existing values, and reads always return the current status.

4.2 The Role Assignment Heuristic

Our heuristic is based on a simple idea: first perform a naive assignment of roles to the network nodes, and then allow every node to decide locally if it wants to transfer the role to any of its neighbors. Upon completion of the naive assignment phase, a second phase of role transfer begins. A node hosting any fusion point role, checks if one of its neighbor nodes can host that role better using a cost function to determine the "goodness" of hosting a particular role. If a better node is found then a role transfer is initiated. Since all decisions are taken locally, every node needs to know only as much information as is required for determining the goodness of hosting a given role for a given application task graph. For example, if the cost function is based upon the remaining power level at the host, every node needs to know only its own power level.

Naive Tree Building: The procedure of finding a naive role assignment can start at any node. For simplicity, let us say it starts at the root node, a node where an end user interacts with the system. The user presents the application task graph to the root node. The root node decides if it wants to host the root fusion function of the task graph based upon its available resources. If the root node does host the root fusion function, it delegates the task of further building the sub-trees under the root of the task graph to its neighbors, else it delegates the building of complete tree to one of its neighbors. For example, consider the case where the root node decides to host the root fusion function. In this case, if the root fusion function has two inputs from two other fusion points, the root node delegates the two subtrees, one corresponding to each of the input fusion points, to two of its neighbors. For the delegation of building subtrees, the root node selects two of its "richest" neighbors. These neighbors are chosen based upon their reported resources. The chosen delegate nodes build the subtrees following a procedure similar to the one at the root. This recursive tree building ends when the input to the fusion points are data producer nodes (i.e. sources). The completion notification of the tree building phase recursively bubbles up the tree from the sources to the root.

Note that, during this phase, different fusion points are assigned to distinct nodes whenever possible. If there are not as many neighbors as needed for delegation of the subtrees, the delegating node assumes multiple roles. Also, even the data producing nodes are treated similar to the non-producing nodes for the role assignment purpose in this phase. During later phases, a cost function decides if multiple fusion points should be assigned to the same sensor node or if data sources should not be allowed to host a fusion point.

Optimization Phase: After completion of the naive tree building phase, the root node informs all other nodes in the network about the start of the optimization phase. During this phase, every node hosting a fusion point role is responsible for either continuing to play that role or transferring the role to one of its neighbors. The decision for role transfer is taken solely by the fusion node based upon local information. A fusion node periodically informs its neighbors about its role and its *health* – an indicator of how good the node is in hosting that role. Upon receiving such a message, a neighboring node computes its own health for hosting that role. If the receiving node determines that it can play the role better than the sender, then it informs the sender (fusion node) of its own health and its intent for hosting that role. If the original sender receives one or more

intention requests from its neighbors, the role is transferred to the neighbor with the best health. Thus, with every role transfer, the overall health of the overlay network improves. Application data transfer starts only after the optimization phase to avoid possible energy wastage in an unoptimized network. DFuse uses a third *maintenance phase* that works similar to the optimization phase (same role transfer semantics). Details are presented in Section 5.

4.3 Sample Cost Functions

Health of a node is quantified by an application-supplied cost function. The choice of the particular set of parameters to use in a cost function depends on the figure of merit that is important for the application at hand.

We describe four sample cost functions below. They are motivated by recent works on power-aware routing in mobile ad hoc networks [14, 9]. The *health* of a node *k* to run fusion role *f* is expressed as the cost function $c(k, f)$. A fusion node compares its own health with the reported health of its neighbors, and it does the role transfer if there is an expected health improvement that is beyond a threshold. Note that the lower the cost function value, the better the node health.

Minimize transmission cost - 1 (MT1): This cost function aims to decrease the amount of data transmission required for running a fusion function. Input data needs to be transmitted from sources to the fusion point, and the output data needs to be propagated to the consumer nodes (possibly across hops). For a fusion function *f* with *m* input data sources (fan-in) and *n* output data consumers (fan-out), the transmission cost for placing *f* on node *k* is formulated as:

$$c_{MT1}(k, f) = \sum_{i=1}^{m} t(source_i) * hopCount(input_i, k)$$

$$+ \sum_{j=1}^{n} t(f) * hopCount(k, output_j)$$

Here, *t(x)* represents the transmission rate of the data source *x*, and $hopCount(i, k)$ is the distance (in number of hops) between node *i* and *k*.

Minimize power variance (MPV): This cost function tries to keep the power of network nodes at similar levels. If $power(k)$ is the remaining power at node *k*, the cost of placing *any* fusion function on that node is:

$$c_{MPV}(k) = 1/power(k)$$

Minimize ratio of transmission cost to power (MTP): This cost function aims to decrease both the transmission cost and lower the difference in the power levels of the nodes. The intuition here is that the cost reflects how long a node can run the fusion function. The cost of placing a fusion function *f* on node *k* can be formulated as:

$$c_{MTP}(k, f) = c_{MT1}(k, f) * c_{MPV}(k)$$

Minimize transmission cost - 2 (MT2): This cost function is similar to **MT1**, except that now the cost function behaves like a step function based upon the node's power level. For a powered node, the cost is same as $c_{MT1}(k, f)$, but if the node's power level goes below a threshold, then its cost for hosting any fusion function becomes infinity. Thus, if a fusion point's power level goes down, a role transfer will happen even if the transfer deteriorates the transmission cost. The cost function can be represented as:

$$c_{MT2}(k, f) \; = \; (\; power(k) \; > \; threshold\;)\;?$$

$$(\; c_{MT1}(k, f) \; : \; INFINITY\;)$$

4.4 Heuristic Analysis

For the class of applications and environments that the role assignment algorithm is targeted, the health of the overall mapping can be thought of as the sum of the health of individual nodes hosting the roles. The heuristic triggers a role transfer only if there is a relative health improvement. Thus, it is safe to say that the dynamic adaptations that take place improve the life of the network with respect to the cost function.

The heuristic could occasionally result in the role assignment getting caught in a local minima. However, due to the dynamic nature of WASN and the re-evaluation of the health of the nodes at regular intervals, such occurrences will be short lived. For example, if 'minimize transmission cost (MT1 or MT2)' is chosen as the cost function, and if the network is caught in a local minima, that would imply that some node is losing energy faster than an optimal node. Thus, one or more of the suboptimal nodes will die causing the algorithm to adapt the assignment. This behavior is observed in real life as well and we show it in the evaluation section.

The choice of cost function has a direct effect on the behavior of the heuristic. We examine the behavior of the heuristic for a cost function that uses two simple metrics: (a) simple hop-count distance, and (b) fusion data expansion or contraction information.

The heuristic leads mainly to two types of role transfers:

Linear Optimization: If all the inputs to a fusion node are coming via a relay node (Figure 3A), and there is data contraction at the fusion point, then the relay node will become the new fusion node, and the old fusion node will transfer its responsibility to the new one (Figure 3B.) In this case, the fusion point is moving away from the sink, and coming closer to the data source points. Similarly, if the output of the fusion node is going to a relay node, and there is data expansion, then again the relay node will act as the new fusion node. In this case, the fusion point is coming closer to the sink and moving away from the data source points.

Triangular Optimization: If there are multiple paths for inputs to reach a fusion point (Figure 4A), and if there is data contraction at the fusion node, then a triangular optimization can be effected (Figure 4B) to bring the fusion point closer to the data source points. The fusion point will move along the input path that maximizes the savings. In the event of data expansion at the fusion point, the next downstream node from the fusion point in the path towards the sinks will become the new fusion node. The original fusion point will simply act as a relay node.

5. IMPLEMENTATION

DFuse is implemented as a multi-threaded runtime system, assuming infrastructure support for timestamping data produced from different sensors, and a reliable transport layer for moving data through the network. Multi-threading the runtime system

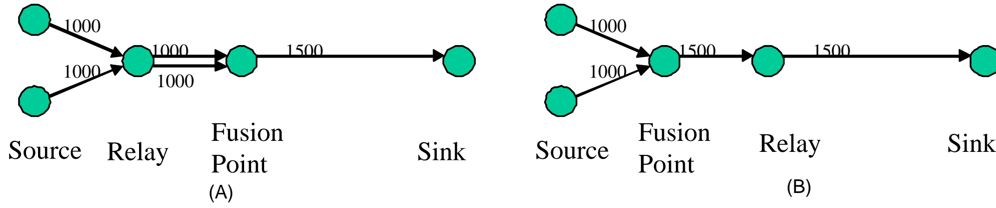


Figure 3: Linear Optimization example.

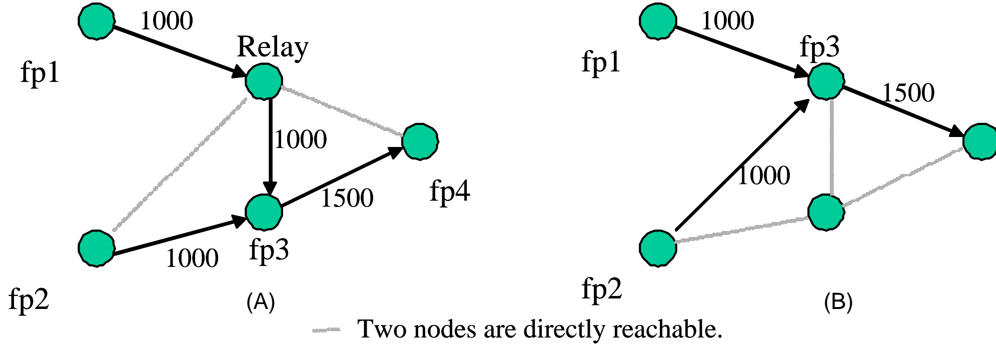


Figure 4: Triangular Optimization example .

enhances opportunities for parallelism in data collection and fusion function execution for streaming tasks. The infrastructural assumptions can be satisfied in various ways. As we mentioned in Section 3, the timestamps associated with the data can be virtual or real. Virtual timestamping has several advantages, the most important of which is the fact that the timestamp can serve as a vehicle for propagating the causality between raw and processed data from a given sensor. Besides, virtual timestamps allows an application to choose the granularity of real-time interval for chunking streaming data. Further, the runtime overhead is minimized since there is no requirement for global clock synchronization, making virtual time synchrony attractive for WASN. For transport, given the multi-hop network topology of WASN, a messaging layer that supports ad hoc routing is desirable.

Assuming above infrastructure support, implementing DFuse consists of the following steps:

1. Implementing a multi-threaded architecture for the fusion module that supports the basic fusion API calls (Section 3), and the other associated optimizations such as prefetching;

2. Implementing the placement module that supports the role assignment tasks (Section 4); and

3. Interfacing the two modules for both instantiating the application task graph and invoking changes in the overlay network during execution.

The infrastructural requirements are met by a programming system called Stampede [13, 2]. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through *channels*. Stampede also provides *registers* with *full/empty* synchronization semantics for inter-thread signaling and event notification. The threads, channels, and registers can be launched anywhere in the distributed system, and the runtime system takes care of automatically garbage collecting the space associated with obsolete items from the channels. Though Stampede's messaging layer does not support adaptive multi-hop ad hoc routing, we adopt a novel way of performing the evaluation with limited routing support (Section 6). For the ease of evaluation, we have decoupled the fusion and placement module implementations. Their interface is a built-in communication channel and a protocol that facilitates dynamic task graph instantiation and adaptation using the DFuse API. Transmission rates exhibited by the application are collected by this interface and communicated to the placement module.

5.1 Data Fusion Module

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) 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 are prefetch threads that gather items from the input buffers, fuse them, and place them in the prefetch buffer for potential future requests. This feature allows latency hiding but comes at the cost of potentially wasted network bandwidth and hence energy (if the fused item is never used). Although this feature can be turned off, we leave it on in our evaluation and ensure that no such wasteful communication occurs. Similarly, there is a Stampede channel that stores requests that are currently being processed by the fusion architecture to eliminate duplication of work.

The `createFC` 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. An `attachFC` 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 `getFCItem` 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 `getFCItem` requests for fusion channels created at this node. Since data may have to be fetched from a number of input buffers to satisfy the `getFCItem` request, one worker thread is assigned to each input buffer to increase the parallelism for fetching the data items. Once fetching is complete, the worker thread rejoins the pool of free threads. The worker thread to fetch the last of the requisite input items invokes the fusion function and puts the resulting fused item in the fusion buffer. This implementation is performance-conscious in two ways: first, there is no duplication of fusion work for the same fused item from multiple requesters; second, fusion work itself is parallelized at each node through the worker threads.

The duration to wait on an input buffer for a data item to be available is specified via a policy flag to the `getFCItem`. For example, if *try_for_time_delta* policy is specified, then the worker thread will wait for time *delta* on the input buffer. On the other hand, if *block* policy is specified, the worker thread will wait on the input buffer until the data item is available. The implementation also supports partial fusion in case some of the worker threads return with an error code during fetch of an item. Taking care of failures through partial fusion is a very crucial component of the module since failures and delays can be common in WASN.

As we mentioned earlier, Stampede does automatic reclamation of storage space of data items in channels. Stampede garbage collection uses a global lower bound for timestamp values of interest to any of the application threads (which is derived from a per-thread state variable called thread virtual time). Our fusion architecture implementation leverages this feature for cleaning up the storage space in its internal data structures (which are built using Stampede abstractions).

5.2 Placement Module

The placement module implementation is an event-based simulation of the distributed heuristic for assigning roles (Section 4) in the network. It takes an application task graph and the network topology information as inputs, and generates an overlay network, wherein each node in the overlay is assigned a unique role of performing a fusion operation. It currently assumes an ideal routing layer (every node knows a route to every other node) and an ideal MAC layer (no contention). It should be noted that any behavior different from this ideal one can be encoded, if necessary, in an appropriate cost function. Similarly, any enhancement in the infrastructure capabilities such as multicast can also be captured in an appropriate cost function.

The placement module runs in three phases, each for a predefined duration. The application is instantiated only at the end of the second phase. The three phases are:

1. Naive tree building phase: This phase starts with registering the "build naive tree" event at the root node with the application task graph as the input. In this phase, the task graph is simply mapped to the network starting from the root node (or the set of sinks in the task graph) as described in Section 4, disregarding cost function evaluation. At the end of this phase, there will be a valid, though naive, role assignment. Every node will know its role, if any, and it will know the addresses of the producer and consumer nodes corresponding to its role.
2. Optimization phase: In this phase, the heuristic runs with a cost-function based upon the hop-count information and fusion function characteristic (data expansion or contraction) at the fusion points. The application has not yet been launched. Therefore the nodes do not have actual data transmission rates to incorporate into the cost function. The nodes exchange the hop-count and fusion characteristics information frequently to speed up the optimization, and lead to an initial assignment of roles prior to launching the application.
3. Maintenance phase: This phase behaves similarly to the optimization phase, with the difference that the application is actually running. Therefore, the cost function will use real transfer rates from the different nodes in possibly changing the role assignments. In principle, we could have moved directly from the first phase to this phase. The reason for the second (optimization) phase prior to application startup is to avoid excessive energy drain with actual application data transmissions before the network stabilizes to an initial good mapping. The frequency of role transfer request broadcasts in the third phase is a tunable parameter.

During the optimization phase, the cost function uses the fusion function characteristics such as data expansion or contraction. If such information is not available for a role, then data contraction is assumed by the placement module. If there are multiple consumers for data produced at some fusion point, then it is tricky to judge if there is an effective data expansion or contraction at such nodes. Even if the fusion characteristic indicates that there is data contraction, if the same data is to be transmitted to more than one consumer, effectively there may be data expansion. Also, if two or more consumers are within a broadcast region of the fusion point, then a single transmission may suffice to transport the data to all the consumers, and this will lessen the effect of the data expansion. However, these effects are accountable in the cost function.

6. EVALUATION

We have performed an evaluation of the fusion and placement modules of the DFuse architecture at two different levels: micro-benchmarks to quantify the overhead of the primitive operations of the fusion API including channel creation, attachments/detachments, migration, and I/O; ability of the placement module to optimize the network given a cost function. The experimental setup uses a set of wireless iPAQ 3870s running Linux "familiar" distribution version 0.6.1 together with a prototype implementation of the fusion module discussed in section 5.1.

6.1 Fusion API Measurements

Figure 5 shows the cost of the DFuse API. In part (a), each API cost has 3 fields - *local*, *ideal*, and *API overhead*. Local

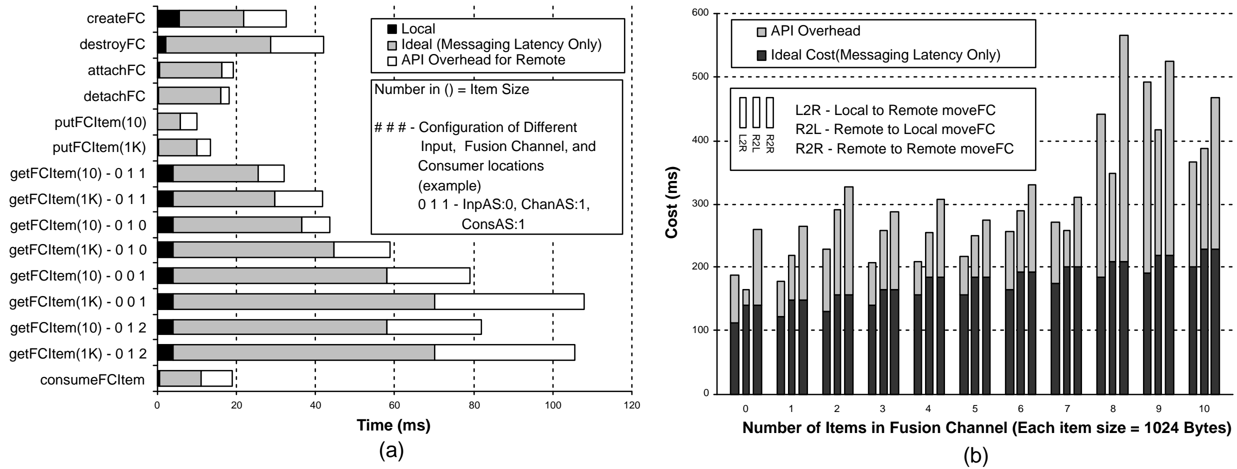


Figure 5: (a) Fusion Channel APIs' cost (b) Fusion channel migration (moveFC) cost

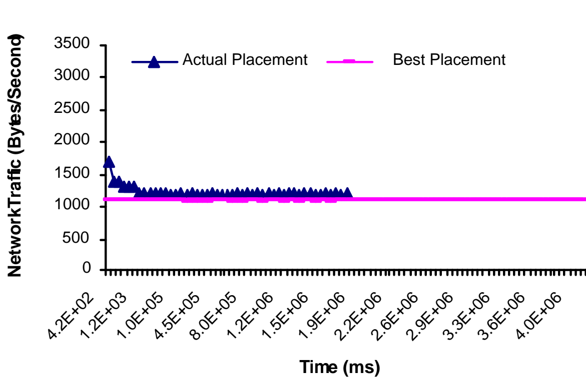
API	Round Trips	Msg overhead (bytes)	API	Round Trips	Msg overhead (bytes)
createFC	3	596	getFCItem(1K) - 0 1 0	6	3112
destroyFC	5	760	getFCItem(10) - 0 0 1	10	1738
attachFC	3	444	getFCItem(1K) - 0 0 1	10	4780
detachFC	3	462	getFCItem(10) - 0 1 2	10	1738
putFCItem(10)	1	202	getFCItem(1K) - 0 1 2	10	4780
putFCItem(1K)	1	1216	consumeFCItem	2	328
getFCItem(10) - 0 1 1	4	662	moveFC(L2R)	20	4600
getFCItem(1K) - 0 1 1	4	1676	moveFC(R2L)	25	5360
getFCItem(10) - 0 1 0	6	1084	moveFC(R2R)	25	5360

Table 1: Number of round trips and message overhead of DFuse. See Figure 5 for getFCItem and moveFC configuration legends.

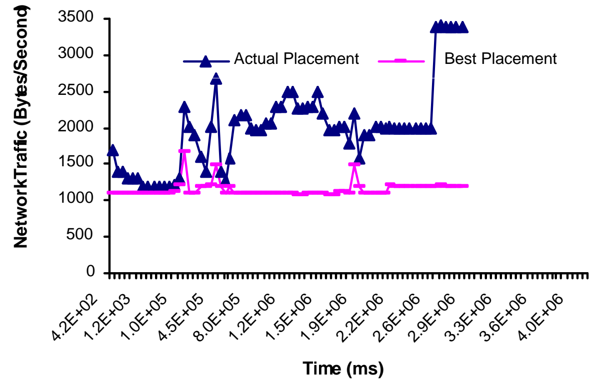
cost indicates the latency of operation execution without any network transmission involved, ideal cost includes messaging latency only, and API overhead is the subtraction of local and ideal costs from actual cost on the iPAQ farm. Ideally, the remote call is the sum of messaging latency and local cost. Fusion channels can be located anywhere in the sensor network. Depending on the location of the fusion channel's input(s), fusion channel, and consumer(s), the minimum cost varies because it can involve network communications. getFCItem is the most complex case, having four different configurations and costs independent of the item sizes being retrieved. For part (a), we create fusion channels with capacity of ten items and one primitive Stampede channel as input. Reported latencies are the average of 1000 iterations.

On our iPAQ farm, netperf [12] indicates a minimum UDP roundtrip latency of 4.7ms, and from 2-2.5Mbps maximum unidirectional streaming TCP bandwidth. Table 1 depicts how many round trips are required and how many bytes of overhead exist for DFuse operations on remote nodes. From these measurements, we show messaging latency values in Figure 5(a) for ideal case costs on the farm. We calculate these ideal costs by adding latency per round trip and the cost of the transmission of total bytes, presuming 2Mbps throughput. Comparing these ideal costs in Figure 5(a) with the actual total cost illustrates reasonable overhead for our DFuse API implementation. The maximum cost of operations on a local node is 5.3ms. For operations on remote nodes, API overhead is less than 74.5% of the ideal cost. For operations with more than 20ms observed latency, API overhead is less than 53.8% of the ideal cost. This figure also illustrates that messaging constitutes the majority of observed latency of API operations on remote nodes. Note that ideal costs do not include additional computation and synchronization latencies incurred during message handling.

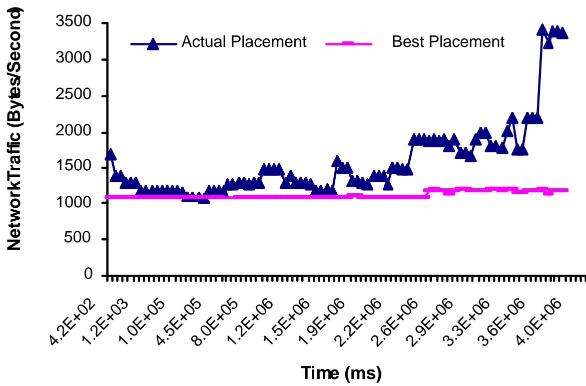
The placement module may cause a fusion point to migrate across nodes in the sensor fusion network. Migration latency depends upon many factors: the number of inputs and consumers attached to the fusion point, the relative locations of the node where moveFC is invoked to the current and resulting fusion channel, and amount of data to be moved. Our analysis in Figure 5(b) assumes a single primitive stampede channel input to the migrating fusion channel, with only a single consumer. Part (b) shares the same ideal cost calculation methodology as part (a). Our observations show that migration cost increases with number of input items and that migration from a remote to a remote node is more costly than local to remote or remote to local migration for a fixed number of items. Reported latencies are averages over 300 iterations for part (b).



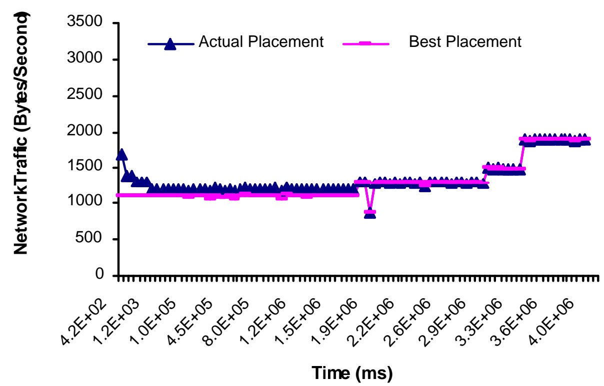
(A) MT1: Minimize Transmission Cost - 1



(B) MPV: Minimize Power Variance



(C) MTP: Ratio of Transmission Cost to Available Power



(D) MT2: Minimize Transmission Cost - 2

Figure 6: The network traffic timeline for different cost functions. X axis shows the application runtime and Y axis shows the total amount of data transmission per unit time.

6.2 Placement Algorithm Measurements

To verify the design of the fusion module and placement algorithm, we have implemented the tracker application (Figure 1) using the fusion API and deployed it on the iPAQ farm.

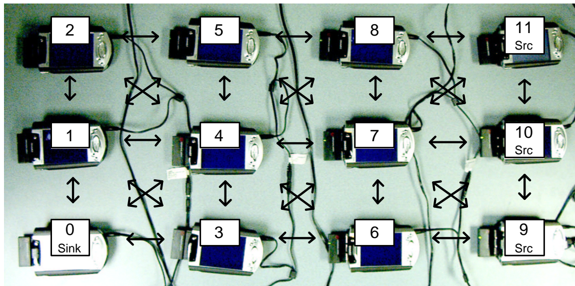


Figure 7: iPAQ Farm Experiment Setup. An arrow represents that two iPAQs are mutually reachable in one hop.

Figure 7 shows the topological view of the iPAQ farm used for the tracker application deployment. It consists of twelve iPAQ 3870s configured identically to those in the measurements above. Node 0, where node i is the iPAQ corresponding to ith node of the grid, acts as the sink node. Nodes 9, 10, and 11 are the iPAQs acting as the data sources. The location of *filter* and *collage* fusion points are guided by the placement module.

The placement module simulator runs on a separate desktop in synchrony with the fusion module. At regular intervals, it collects the transmission details (number of bytes exchanged between different nodes) from the farm. It uses a simple power model (discussed later) to account for the communication cost and to monitor the power level of different nodes. If the placement module decides to transfer a fusion point to another node, it invokes the `moveFC` API to effect the role transfer.

For transmission rates, we have tuned the tracker application to generate data at consistent rates as shown in Figure 1, with x equal to 6KBytes per minute. This is equivalent to a scenario where cameras scan the environment once every minute, and produce images ranging in size from 6 to 12KBytes after compression.

The network is organized as the grid shown in Figure 7. For any two nodes, the routing module returns the path with a min-

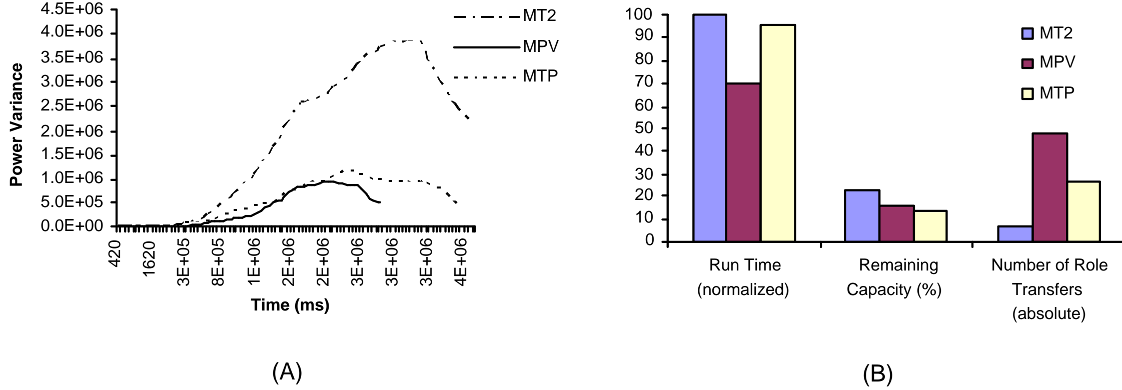


Figure 8: Comparison of different cost functions. Application runtime is normalized to the best case (*MT2*), and total remaining power is presented as the percentage of the initial power.

imum number of hops across powered nodes. To account for power usage at different nodes, the placement module uses a simple approach. It models the power level at every node, adjusting them based upon the amount of data a node transmits or receives. The power consumption corresponds to ORiNOCO 802.11b PC card specification [1]. Our current power model only includes network communication costs. After finding a naive tree, the placement algorithm runs in optimization phase for two seconds. The length of this period is tunable and it influences the quality of mapping at the end of the optimization phase. During this phase, fusion nodes wake up every 100ms to determine if role transfer is indicated by the cost function. After optimization, the algorithm runs in maintenance phase until the network becomes fragmented (some consumer cannot reach one of its inputs). During the maintenance phase, role transfer decisions are evaluated every 50 seconds. The role transfers are invoked only when the health improves by a threshold of 5%.

Figure 6 shows the network traffic per unit time (sum of the transmission rate of every network node) for the cost functions discussed in Section 4.3. It compares the network traffic for the actual placement with respect to the best possible placement of the fusion points (best possible placement is found by comparing the transmission cost for all possible placements). Note that the application runtime can be increased by simply increasing the initial power level of the network nodes.

In **MT1**, the algorithm finds a locally best placement by the end of optimization phase itself. The optimized placement is only 10% worse than the best placement. The same placement continues to run the application until one of the fusion points (one with the highest transmission rate) dies, i.e. the remaining capacity becomes less than 5% of the initial capacity. If we do not allow role migration, the application will stop at this time. But allowing role migration, as in **MT2**, enables the migrating fusion point to keep utilizing the power of the available network nodes in the locally best possible way. Results show that **MT2** provides maximum application runtime that is more than twice as long as that for **MT1**. The observed network traffic is at most 12% worse than the best possible for the first half of the run, and it is same as the best possible rate for the latter half of the run. **MPV** performs worst, while **MTP** has comparable network lifetime as **MT2**. Figure 6 also shows that running the *optimization phase* before instantiating the application improves the total transmission rate by 34% compared to the naive placement.

Though **MPV** does not provide comparably good network lifetime (Figure 6B), it provides the best (least) power variance compared to other cost functions (Figure 8A). Since **MT1** and **MT2** drain the power of fusion nodes completely before role migration, they show worst power variance. Also, the number of role migrations is minimum compared to other cost functions (Figure 8B). These results show that the choice of cost function should be dependent upon application context and network condition. If, for an application, role transfer is complex and expensive, but network power variance is not an issue, then **MT2** should be preferred. However, if network power variance needs to be minimized and role transfer is inexpensive, **MTP** should be preferred. Simulation results for other task graph configurations have been found to provide similar insight into the cost functions' behavior.

6.3 Discussion

By running the application and role assignment modules separately, we have simplified our evaluation approach. This approach has some disadvantages such as limited ability to communicate complex resource monitoring results. Transferring every detail of the running state from the fusion module to the placement module is prohibitive in our decoupled setup due to the resulting network perturbation. Such perturbation, even when minimal state is being communicated between the modules, prevents accurate network delay metric usage in a cost function. However, our simplified evaluation design has allowed us to rapidly build prototypes of the fusion and placement modules.

7. RELATED WORK

Data fusion, or in-network aggregation, is a well-known technique in sensor networks. Research experiments have shown that it saves considerable amount of power even for simple fusion functions like finding min, max or average reading of sensors [11, 8]. While these experiments and others have motivated the need for a good role assignment approach, they do not use a dynamic heuristic for the role assignment and their static role

assignment approach will not be applicable to streaming media applications.

DFuse employs a script based interface for writing applications over the network similar to SensorWare [4]. SensorWare is a framework for programming sensor networks, but its features are orthogonal to what DFuse provides. Specifically, 1) SensorWare does not employ any strategy for assigning roles to minimize the transmission cost, or dynamically adapt the role assignment based on available resources. It leaves the onus to the applications. 2) Since DFuse focuses on providing support for fusion in the network, the interface to write fusion-based applications using DFuse is quite simple compared to writing such applications in SensorWare. 3) DFuse provides optimizations like prefetching and support for buffer management which are not yet supported by other frameworks. Other approaches, like TAG [11], look at a sensor network as a distributed database and provide a query-based interface for programming the network. TAG uses an SQL-like query language and provides in-network aggregation support for simple classes of fusion functions. But TAG assumes a static mapping of roles to the network, i.e. a routing tree is built based on the network topology and the query in hand.

Recent research in power-aware routing for mobile ad hoc networks [14, 9] proposes power-aware metrics for determining routes in wireless ad hoc networks. We use similar metrics to formulate different cost functions for our DFuse placement module. While designing a power-aware routing protocol is not the focus of this paper, we are looking into how the routing protocol information can be used to define more flexible cost functions.

8. FUTURE WORK

We plan to extend our work in two directions: extending the fusion API to accommodate more applications, and further exploring the role assignment algorithm behavior and capabilities. DFuse assumes that the addresses of the data sources are known at query time. This assumption may be a limiting assumption for many applications where data sources are unknown at query time. We are exploring different ways of extending DFuse to handle such data-centric queries. One possible approach is to have an interest-dissemination phase before the naive tree building phase of the role assignment algorithm. During this phase, the interest set of individual nodes (for specific data) is disseminated as is done in directed diffusion. When the exploratory source packets reach the sink (root node of the application task graph), the source addresses are extracted and recorded for later use in other phases of the role assignment algorithm. We plan to study the role assignment behavior in the presence of node mobility and/or failure. We expect future iterations of this algorithm to gracefully adapt fusion point placements in the presence of mobility and failures as is done currently to conserve power. The cost function may need to include parameters pertaining to mobility and node failures. We would also like to investigate cost functions extensions to include cross-layer information (such as placement of relay nodes) to further improve fusion point placement.

9. CONCLUSION

As the form factor of computing and communicating devices shrinks and the capabilities of such devices continue to grow, it has become reasonable to imagine applications that require rich computing resources today becoming viable candidates for future sensor networks. With this futuristic assumption, we have embarked on designing APIs for mapping fusion applications such as distributed surveillance on wireless ad hoc sensor networks. We argue that the proposed framework will ease the development of complex fusion applications for future sensor networks. Our framework uses a novel distributed role assignment algorithm that will increase the application runtime by doing power-aware, dynamic role assignment. We validate our arguments by designing a sample application using our framework and evaluating the application on an iPAQ based sensor network testbed.

10. REFERENCES

[1] ORiNOCO PC Card (Silver/Gold) Specification: *http://www.hyperlinktech.com/web/orinoco/-orinoco_pc_card_spec.html*, 2003.

[2] S. Adhikari, A. Paul, and U. Ramachandran. D-stampede: Distributed programming system for ubiquitous computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems(ICDCS)*, Vienna, July 2002.

[3] M. Bhardwaj and A. Chandrakasan. Bounding the lifetime of sensor networks via optimal role assignments. In *IEEE INFOCOM*, 2002.

[4] A. Boulis, C. C. Han, and M. B. Srivastava. Design and implementation of a framework for programmable and efficient sensor networks. In *The First International Conference on Mobile Systems, Applications, and Services(MobiSys)*, San Francisco,CA, 2003.

[5] E. Cayirci, W. Su, and Y. Sankarasubramanian. Wireless sensor networks: A survey. *Computer Networks (Elsevier)*, 38(4):393–422, March 2002.

[6] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Symposium on Operating Systems Principles*, pages 146–159, 2001.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[9] Jae-Hwan Chang and Leandros Tassiulas. Energy conserving routing in wireless ad-hoc networks. In *IEEE INFOCOM*, pages 22–31, 2000.

[10] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Operating System Design and Implementation(OSDI)*, Boston,MA, Dec 2002.

[12] Netperf. The Public Netperf Homepage: *http://www.netperf.org/*, 2003.

[13] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-time memory: A parallel programming abstraction for interactive multimedia applications. In *Principles Practice of Parallel Programming*, pages 183–192, 1999.

[14] S. Singh, M. Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 181–190, 1998.

[15] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.