Dynamic Data Fusion for Future Sensor Networks

UMAKISHORE RAMACHANDRAN, RAJNISH KUMAR, MATTHEW WOLENETZ, BRIAN COOPER, BIKASH AGARWALLA, JUNSUK SHIN, PHILLIP HUTTO, and ARNAB PAUL

Georgia Institute of Technology

DFuse is an architectural framework for dynamic application-specified data fusion in sensor networks. It bridges an important abstraction gap for developing advanced fusion applications that takes into account the dynamic nature of applications and sensor networks. Elements of the DFuse architecture include a fusion API, a distributed role assignment algorithm that dynamically adapts the placement of the application task graph on the network, and an abstraction migration facility that aids such dynamic role assignment. Experimental evaluations show that the API has low overhead, and simulation results show that the role assignment algorithm significantly increases the network lifetime over static placement.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems, real-time systems and embedded systems*

General Terms: Algorithms, Design, Management, Measurement

Additional Key Words and Phrases: Sensor network, in-network aggregation, data fusion, role assignment, energy awareness, middleware, platform

1. INTRODUCTION

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 the larger footprint to the 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

This work has been funded in part by 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 NSF Research Infrastructure award EIA-99-72872, and Intel Corp.

Authors' addresses: College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, Georgia 30332-0280; email: {rama,rajnish,wolentz,cooperb,bikash,espress,pwb,arnab}@cc.gatech.edu.

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*, that is, applying a synthesis operation on input streams.

This article focuses on the challenges involved in supporting fusion applications in *wireless ad hoc sensor networks* (WASN). Developing a fusion application is challenging in general, for the fusion operation typically requires time-correlation and synchronization of data streams coming from several distributed sources.

Since such applications are inherently distributed, they are typically implemented via distributed threads that perform fusion hierarchically. Thus the application programmer has to deal with thread management, data synchronization, buffer handling, and exceptions (such as timeouts while waiting for input data for a fusion function)—all with the complexity of a loosely coupled system. WASN add another level of complexity to such application development—the need for being power-aware [Cayirci et al. 2002]. 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 a 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 [Zayas 1987].

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 [Madden et al. 2002; Intanagonwiwat et al. 2000; Heidemann et al.

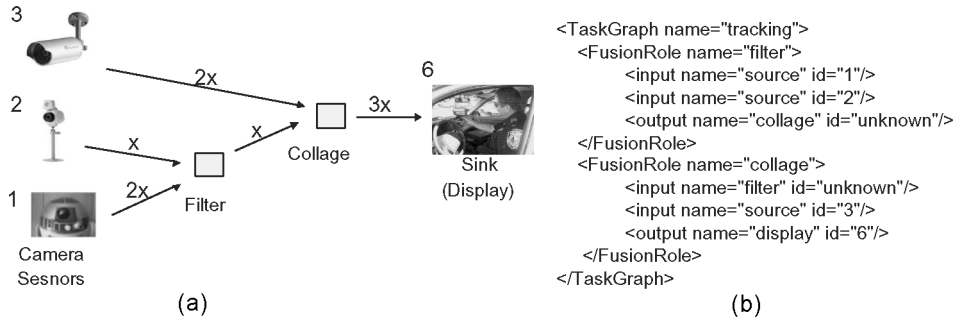

Fig. 1. An example application. (a) Pictorial representation of the task graph (with expected data flow rates on the edges); (b) textual representation of the task graph.

2001] that allow only limited types of aggregation operations as fusion functions. The API includes primitives for on-demand migration of the fusion point.

(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, in a distributed manner, that minimizes communication is difficult. We develop a novel heuristic-based algorithm to find a *good* (with respect to some predefined cost function) mapping of fusion functions to the network nodes.

Also, the placement needs to be reevaluated quite frequently considering the dynamic nature of WASN. The mapping is reevaluated periodically to address dynamic changes in nodes' power levels and network behavior.

We specifically use the term *data fusion* and distinguish it from in-network aggregation. The latter is typically performed on data of the same type to miminize communication, and the operation on the data itself is fairly simple such as addition or min/max. On the other hand, data fusion is performed on data of possibly different types, and the operations may be arbitrarily complex to derive a higher-level decision. Thus fusion point placement is a more involved problem than aggregator placement.

(3) *Experimental evaluation of the DFuse framework.* The evaluation includes microbenchmarks 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. For example, we show that the proposed role assignment algorithm increases the network lifetime by 110% compared to static placement of the fusion functions.

(4) *Simulation results for larger networks.* Using a novel middleware simulator (MSSN) [Wolenetz 2005; Wolenetz et al. 2004, 2005], we simulate the DFuse architecture on a network of hundreds of nodes and for different sizes of the application task graph. We show that the performance of DFuse scales

well for larger networks and application graphs with respect to an optimal placement (approximated using a simulated annealing algorithm).

This journal version of the article is a significant extension of a previous conference publication [Kumar et al. 2003]. The extensions are along the following dimensions: (1) we present a comprehensive list of the APIs available in the fusion module. (2) We introduce the bottom-up and top-down approaches to improving the quality of the initial placement of the role assignment algorithm. Based on the characteristic of the input task graph, one or the other approach may be more desirable to provide a better initial deployment. (3) Using a novel home-grown middleware simulator (MSSN), we present scalability results in terms of increase in network size as well as in the size of the input application task graph. Within the simulator, we have developed approximations to an optimal assignment (using simulated annealing algorithm) that allows comparison of our proposed role assignment algorithm to a theoretical optimal.

The rest of the article 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. APPLICATION CONTEXT AND REQUIREMENTS

A fusion application has the following characteristics: (1) it is *continuous* in nature, (2) it requires efficient transport of data from/to distributed *sources*/*sinks*, and (3) it requires efficient in-network processing of application-specified *fusion* functions. A data source may be a sensor (e.g., camera) or a standalone program; a data sink represents an end consumer and includes a human in the loop, an actuator (e.g., a fire alarm), an application (e.g., a data logger), or an output device such as a display; a fusion function transform the data streams (including aggregation of separate streams into a composite one) en route to the sinks. Thus a fusion application is a directed *task graph*: the vertices are the fusion functions, and the edges represent the data flow (i.e., producer-consumer relationships) among the fusion points (cycles—if any—represent feedback in the task graph).

This formulation of the fusion application has a nice generality. It may be an application in its own right (e.g., video based surveillance). It allows hierarchically composing a bigger application (e.g., an emergency response) wherein each component may itself be a fusion application (e.g., image processing of videos from traffic cameras). It allows query processing by overlaying a specific query (e.g., "show a composite video of all the traffic at the spaghetti junction") on to the task graph.

Consider, for example, a video-based surveillance application. Cameras are deployed in a distributed fashion; the images from the cameras are filtered in some application-specific manner, and fused together in a form that makes

it easy for an end user (human or some program) to monitor the area. The compute-intensive part may analyze multiple camera feeds from a region to extract higher-level information such as "motion," "presence or absence of a human face," or "presence or absence of any kind of suspicious activity." Figure 1 shows the task graph for this example application, in which *filter* and *collage* are *fusion functions* that transform input streams into output streams in an application specified manner. The fusion functions may result in contraction or expansion of data flows in the network. For example, the filter function selects images with some interesting properties (e.g., a 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.

2.1 Technology Trends

Given the pace of technology, it is conceivable to imagine future sensor networks wherein some nodes have the computational capability of today's handhelds (such as an iPAQ), and communication capabilities equivalent to Bluetooth, 802.11a/b/g, 802.15.3 (WPAN), or even UWB (up to 1 Gb/s). While a quest for smaller footprint devices with lower cost continues, we expect that there will have a continuum of capabilities from the Berkeley motes to today's handhelds. Recent advances in low-power microcontrollers, and increased power-conscious radio technologies lend credence to this belief. For example, next-generation iMote prototypes (go online to `http://www.intel.com/research/exploratory/motes.htm`) and Telos motes [Polastre et al. 2004] are available for research now. Although not as computationally powerful as a modern iPAQs, iMotes provide 12-MHz 32-bit ARM7TDMI processors and 64-kB RAM/512-kB FLASH, a significant increase in capability compared to Berkeley mote MICA2 (go online to `http://www.xbow.com/Products/productsdetails.aspx?sid=72`) predecessors that only had 8MHz 8-bit ATmega128L microcontrollers with 4-kB RAM/128-kB FLASH. Furthermore, the wireless bandwidth available with iMotes is Bluetooth based (over 600-Kb/s application-level bandwidth), greatly exceeding Berkeley motes' −38.4-Kb/s data rate. Coupled with this trend, high-bandwidth sensors such as cameras are becoming ubiquitous, cheaper, and lighter (in this case possibly due to the large-scale demands of cell-phone manufacturers for these cameras, where camera phone shipment is expected to reach 903 million in 2010[1]). Thus we envision future wireless sensor networks deployments to consist of high bandwidth and powerful sensor/actuator sources and infrastructures coexisting with more constrained nodes, with energy still being a scarce resource.

[1]*Reiter's Camera Phone Report*, section on Statistics—Camera phones. Go online to `http://www.wirelessmoment.com/statistics_camera_phones/`.

2.2 Architectural Assumptions

We have designed the DFuse architecture to cater to the evolving application needs and emerging technology trends.

We make some basic assumptions about the execution environment in the design of DFuse:

—The application level input to the architecture are
  (1) an application task graph consisting of the data flows and relationship among the fusion functions,
  (2) the code for the fusion functions (currently supported as C program binaries),
  (3) a cost function that formalizes some application quality metric for the sensor network (e.g., "keep the average node energy in the network the same").

—The task graph has to be mapped over a large geographical area. In the ensuing overlay of the task graph on to the real network, some nodes may serve as *relays* while others may perform the application-specified fusion operations.

—The fusion functions may be placed anywhere in the sensor network as long as the cost function is satisfied.

—All source nodes are reachable from the sink nodes.

—Every node has a routing layer that allows each node to determine the route to any other node in the network. This is in sharp contrast to most current day sensor networks that support all-to-sink style routing. However, the size of the routing table in every node is only proportional to the size of the application task graph (to facilitate any network node in the ensuing overlay to communicate with other nodes hosting fusion functions) and not the physical size of the network.

—The routing layer exposes information (such as hop-count to a given destination) that is needed for making mapping decisions in the DFuse architecture.

It should be noted that dynamically generating a task graph to satisfy a given data-centric query plan is in itself an interesting research problem. However, the focus of this article is to provide the system support to meet the application requirements elaborated in the previous subsections honoring the above assumptions.

2.3 DFuse Architecture Components

Figure 2 shows the components of the DFuse architecture. There are two components to this architecture that are the focus of this article: the *fusion module*, and the *placement module*.

From an application perspective, there are two main concerns:

(1) How do we develop the fusion application? The fusion code itself (e.g., "motion detection" over a number of incoming video streams and producing a digest) is in the purview of the application developer. However, there are a number of systems issues that have to be dealt with before a fusion

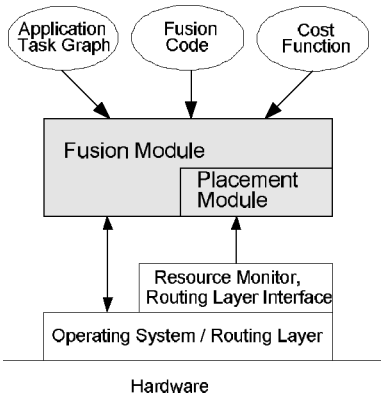

Fig. 2. DFuse architecture.

operation can be carried out at a given node including (a) providing the "plumbing" from the sources to the fusion point; (b) ensuring that all the inputs are available; (c) managing the node resources (CPU and memory) to enhance performance, and (d) error and failure handling when some sources are nonresponsive.

We have designed a fusion module with a rich API that deals with all of the above issues. We describe this module in Section 3.

(2) How do we generate an overlay of the task graph on to the sensor network? As we mentioned earlier, some nodes in the overlay will act as relays and some will act as fusion points. Since the application is dynamic (sources/sinks may join/leave as dictated by the application, new tasks may be created, etc.), and the physical network is dynamic (sources/sink may fail, intermediate nodes may run out of energy, etc.), this mapping is not a one-time deal. After an initial mapping, reevaluation of the mapping (triggered by changes in the application or the physical infrastructure) will lead to new assignment and reassignment of the nodes and the roles they play (relay vs. fusion).

We have designed a placement module that embodies a role assignment algorithm that deals with the above issues. We describe this module in Section 4.

3. FUSION MODULE

The API provided by the fusion module is summarized in Figure 3. The API calls are presented in an abstract form to elide language and platform implementation details.

The fusion API provides capabilities that fall within the general categories described in the following subsections.

3.1 Structure Management

This category of capabilities primarily handles "plumbing" issues. The fundamental abstraction in DFuse is the *fusion channel*, shown in Figure 4. It

—**Structure Management**

```
channel = createFC(inputs, fusion_function)
result = destroyFC(channel)
channel_connection = attachFC(channel)
result = detachFC(channel_connection)
item = get/putFCItem(channel_connection, attributes)
result = consumeFCItem(channel, attributes)
inputs = get/setFCInputs(channel, new_inputs)
inputs = addFCInput(channel, input)
inputs = removeFCInput(channel, input_index)
location = getFCLocation(channel)
result = moveFC(channel, new_location)
result = moveFC(channel, new_location, protocol)
```

—**Correlation Control**

```
params = get/setFCCorrelation(channel, correlation_params)
params include:
    min, max correlation set size
    correlate by timestamp?
    correlation ranges for temporal correlation (per input)
    discard late items?
    correlation predicates:
        boolean addItemToCorrelationSet(channel, item, set)
        boolean activateFusionFunction(channel, set)
```

—**Computation Management**

```
function = get/setFCFunction(channel, fusion_function)
handler = get/setFCEventHandler(channel, event, handler)
source = get/setFCAsynchInput(channel, source)
```

—**Caching, Prefetching, Buffer management**:

```
size = get/setFCPrefusionBufferSize(channel)
policy = get/setFCFusionPolicy(channel, fusion_policy)
policy = get/setFCPrefusionBufferExpiry(channel, expiry_policy)
```

—**Failure and Latency Handling**

```
timeout = get/setFCCorrelationTimeout(channel, timeout)
policy = get/setFCCorrelationTimeoutPolicy(channel, timeout_policy)
item = get/putFCErrorItem(channel, error_item)
```

—**Status and Feedback Handling**

```
status = get/putFCStatus(channel, status, include_inputs?)
command = get/putFCCommand(channel, command, propagate?)
```

Fig. 3. Fusion channel API summary.

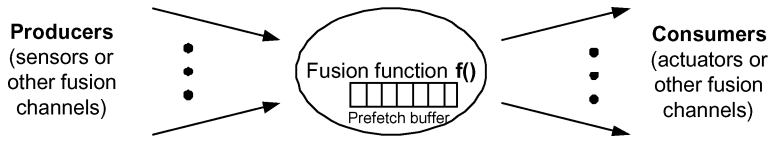

Fig. 4. An example task graph using the fusion channel abstraction.

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 nonblocking. 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 channels, creating fusion networks or pipelines. 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.

3.2 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.

3.3 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 and modify the set of inputs to a fusion channel.

3.4 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.

3.5 Failure and Latency Handling

This category of capabilities primarily allows the fusion points to perform partial fusion, that is, 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 metadata 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.

3.6 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 register."[2] 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.

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

4. PLACEMENT MODULE

This module is responsible for creating an *overlay* of the application task graph on to the physical network that best satisfies an application-specified cost function. A network node can play one of the three roles: *end point (source or sink)*, *relay*, or *fusion point* [Bhardwaj and Chandrakasan 2002]. In our model, the end points are determined by the application. The placement module embodies a distributed *role assignment* algorithm that manages the overlay network, dynamically assigning fusion points to the available nodes in the network.

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

—*Node heterogeneity*. 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.

—*Communication versus computation*. Studies have shown that wireless communication is more energy draining than computation in current day wireless sensor networks [Hill et al. 2000]. It should be noted that, in future wireless sensor networks wherein we expect more significant processing in the nodes (exemplified by the fusion functions), computation is likely to play an increasingly important role in determining node energy [Wolenetz 2005].

—*Dynamic behavior*. As we already mentioned (see Section 2.3), both the application and the network environment are dynamic, thus requiring role assignment decisions to be made fairly frequently during the lifetime of the application. Therefore, it is important that the algorithm have low overhead, and scale well with the size of the application and the network.

The formulation of the role assignment problem, the cost function, and the proposed heuristic are sensitive to the above properties. However, the experimental and simulation results (Section 6) deal only with homogeneous network nodes.

4.1 The Role Assignment Problem

Given $N = (V_n, E_n)$, namely, the network topology, and $T = (V_t, E_t)$, namely, the task graph, and an application-specific cost metric, the goal is to find a mapping $f : V_t \rightarrow V_n$ that minimizes the overall *cost*. Here, V_n represents nodes of the sensor network and E_n represents communication links between them. In the task graph, V_t represents fusion functions (filter, data fusion, etc.) and E_t represents flow of data between the fusion points. A mapping $f : V_t \rightarrow V_n$ generates an overlay network of fusion points to network nodes; implicitly, this generates a mapping $l : E_t \rightarrow \{e | e \in E_n\}$ of data flow to communication links. The focus of the role assignment algorithm is to determine f; determining l is the job of the routing layer and is outside the scope of this article.

We use fusion function and fusion point interchangeably to mean the same thing, namely, the node that runs any fusion function.

4.2 Cost Functions

We describe five sample cost functions below. They are motivated by recent research in power-aware routing in mobile ad hoc networks [Singh et al. 1998; Jae-Hwan and Leandros 2000]. The *health* of a node k to host a fusion role r is expressed as the cost function $c(k, r)$. Note that the lower the value computed by the cost function, the better the node health, and therefore the better equipped the node k is to host the fusion point r.

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

$$c_{MT1}(k, r) = \sum_{i=1}^{m} t(source_i) * HopCount(input_i, k)/Rel(input_i, k) + \sum_{j=1}^{n} t(r) * HopCount(k, output_j)/Rel(k, output_j).$$

Here, *t(x)* represents the amount of data transmission (in bits per unit time) of a data source x, $HopCount(i, k)$ is the distance (in number of hops) between node i and node k, and $Rel(i, j)$ is a value between (0 *and* 1) representing the reliability of the transmission link between nodes i and j.

—*Minimize power variance (MPV)*. This cost function aims to keep the power (energy) levels of the nodes the same. This is a simple cost function that ignores the actual work being done by a node and simply focuses on the remaining power at any given node for role assignment decisions. The cost of doing any work at node k is inversely proportional to its remaining power level $power(k)$, and is formulated as

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

—*Minimize ratio of transmission cost to power (MTP)*. MT1 focuses on the work done by a node. MPV focuses on the remaining power at a given node. MTP aims to give work to a node commensurate with its remaining energy level, and thus represents a combination of the first two cost functions. Intuitively, a role assignment based on MTP is a reflection of how long a given node k can host a given fusion function r with its remaining power level $power(k)$. Thus the cost of placing a fusion function r on node k is formulated as

$$c_{MTP}(k, r) = c_{MT1}(k, r) * c_{MPV}(k) = c_{MT1}(k, r)/power(k).$$

—*Minimize transmission cost—2 (MT2)*. This cost function takes into account node heterogeneity. In particular, it biases the role assignment with a view to protecting low energy nodes. It can be considered a variant of MT1, with the cost function behaving like a step function based on the node's remaining power level. The intuition is that if a node has wall power then its cost function is the same as MT1. For a node that is battery powered, we would protect

it from hosting (*a*ny) fusion point if its energy level goes below a predetermined *threshold*. Thus a role transfer should be initiated from such a node even if it results in increased transmission cost. This is modeled by making the cost of hosting any fusion function at this node *infinity*. This cost function is formulated as

$$c_{MT2}(k,r) = (power(k) > threshold)\ ?\ c_{MT1}(k,r) : INFINITY.$$

—*Minimize computation and communication cost (MCC)*. This cost function accounts for both the computation as well as the communication cost of hosting a fusion function:

$$c_{MCC}(k,r) = c_{MT1}(k,r) * e_{radio}(k) \quad \text{(Communication energy)}$$
$$+\ \ cycleCount(k,r) * frameRate(r) * e_{comp}(k) \quad \text{(Computation energy).}$$

This equation has two parts:

(1) *Communication energy:* $c_{MT1}(k,r)$ represents the transmission cost (bits per unit time); $e_{radio}(k)$ is the energy per bit consumed (Joule/bit) by the radio at node k.

(2) *Computation energy:* $cycleCount(k,r)$ is the computation cost (total number of instructions per input data item) for executing the fusion function r on node k for a standard data input (frame) size; $frameRate(r)$ is the number of items generated per second by r; $e_{comp}(k)$ is the energy per instruction consumed (Joule/instruction) by the processor at node k.

If the network is homogeneous and if we assume that the processing of a given fusion function r is data independent, then $cycleCount(k,r)$ and $e_{comp}(k)$ are the same for any node k. In this case role assignment based on MCC behaves exactly the same as MT1.

In our experimental and simulation results reported in this article (which does relative comparison of our role assignment algorithm to static and optimal), we do not consider MCC any further. However, absolute network lifetime will diminish if the node energy for computation is taken into account. Such a study is outside the scope of this article and is addressed in companion works [Wolenetz 2005; Wolenetz et al. 2004, 2005].

It should be emphasized that the above cost functions are samples. The application programmer may choose to specify the *health* of a node using the figure of merit that best matches the application level requirement. The role assignment algorithm to be discussed shortly simply uses the application provided metric in its assignment decisions.

4.3 In Search of Optimality

The general case of mapping an arbitrary task graph (directed) onto another arbitrary processor graph (undirected) is NP-complete [Garey and Johnson 1979; Papadimitriou and Yannakakis 1988]. Since DFuse treats a fusion function as a black-box, and the application-specified cost function can be arbitrary, the general problem of role assignment is NP-complete.

Given specific task graphs and specific cost functions, optimal solutions can be found deterministically. For example, consider an input task graph in which

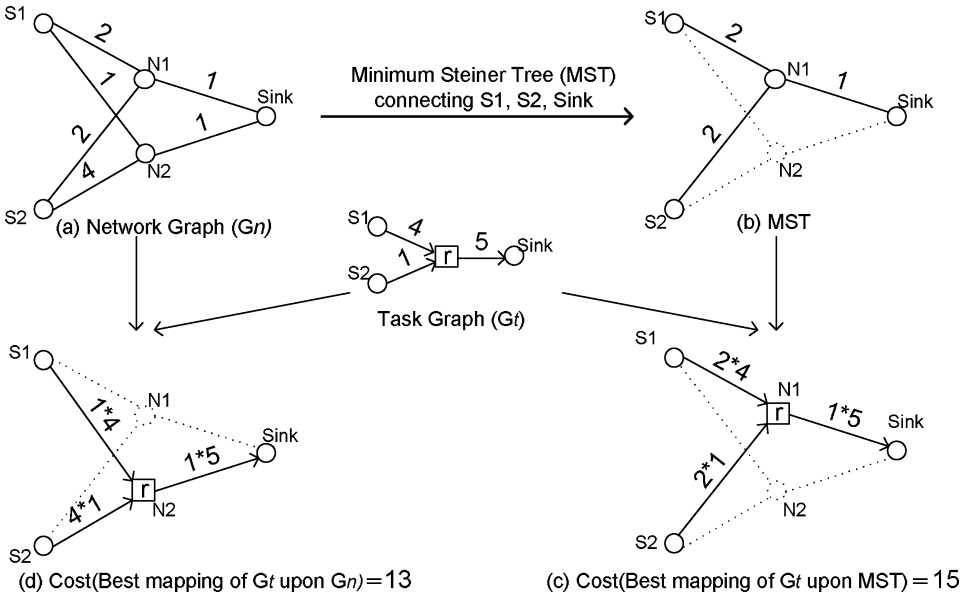

Fig. 5. Mapping a task graph using minimum Steiner tree. Example shows that MST does not lead to an optimal mapping. For G_n, the edge weights can be thought of as hop counts, and for G_t, as transmission volume. Edge weights on the overlay graphs (c) and (d) are obtained by multiplying the edge weights of the task graph with those the corresponding edge weight of the network links.

all the fusion functions are data expanding. For transmission-based cost functions ($MT1, MT2$), a trivial $O(1)$ algorithm would place all the fusion points at the sinks (or as close as possible to the sinks if there is a problem hosting multiple roles at a sink).

At the other extreme, consider a task graph where *all* the fusion functions are data contracting. In this case, for transmission-based cost functions, fusion functions need to be applied to nodes that are as close to the data sources as possible. Therefore, finding a minimum Steiner tree (MST) of the network graph connecting the sources and the sinks, and then mapping individual fusion functions to nodes close to the data sources, *may* lead to a good solution, though optimality is still not guaranteed, as shown in an example mapping in Figure 5. In the example, since an MST is obtained without considering the transmission requirements in the task graph, a mapping based on MST turns out to be more expensive than the optimal. Also, finding MST is APX-complete,[3] with best known approximation bound of 1.55 [Robins and Zelikovsky 2000]. Finally, as is illustrated in this example, MST-based solutions cannot be applied for an arbitrary task graph and cost functions.

Moreover, it is impractical to apply existing approximate solutions to Steiner tree and graph mapping problems to sensor networks. Such solutions assume that (1) the network topology is known, (2) costs on edges in the network are known, and (3) the network is static. None of these assumptions hold in our case:

[3]APX is the class of optimization problems in NP, having polynomial time approximation algorithms.

the network topology is not known and must be discovered; costs on edges are known locally but not globally (and it is too expensive to gather this information at a central planner); and even if we could find the optimal deployment, because of inherent dynamism of WASN, we need to redeploy.

All of these considerations lead us to design a heuristic for role assignment. The fundamental design principle is not to rely on any global knowledge.

4.4 The Role Assignment Heuristic

We have developed a heuristic that adheres to the design principle of using local information and not relying on any global knowledge. The heuristic goes through three phases: *initialization*, *optimization*, and *maintenance*. We describe these three phases in the following subsections.

4.4.1 *Initialization Phase.* In this phase, we make a first-cut naive assignment of fusion points to the network nodes. The application has not started yet, and no application-specified cost function is used in this initial mapping. The only real input to this phase of the algorithm is the resource constraints of the network nodes (for, e.g., "is a node connected to a wall socket?"; "does a node have enough processing power and memory?" etc.). The initial placement, however, has a huge impact on the quality of the steady-state behavior of the application. For example, if the initial mapping places a fusion point k hops away from an optimal placement, at least k role transfers are needed before reaching an optimum. Therefore, we adopt either a *top-down* or *bottom-up* approach based on the *nature* of the application task graph to make the initial placement a good starting point for the subsequent phases. Essentially, if it is known that all the fusion functions in the task graph are data contracting, then placing them as close to the data sources as possible lowers the transmission overhead. This is the bottom-up approach, where we start from the leaves (sources) of the task graph and work toward the root (sink). On the other hand, if the task graph has mostly data-expanding fusion functions, then placing the fusion points close to the sinks makes sense. This is the top-down approach, where the mapping starts from the root of the task graph and progresses toward the leaves. The default (in the absence of any a priori knowledge about the task graph) is to take a bottom-up approach since data contraction is more common in sensor applications.

In either case, this phase is very quick. Needless to say, the mapping of sources and sinks to network nodes is predetermined. Therefore, the job of the initialization phase is to determine the mapping of the fusion points to the network nodes. In principle, if the task graph has a depth k and the network has a depth n, an attempt is made in the initialization phase to place all the fusion points in at most k levels of the tree either near the sources (bottom-up) or the sinks (top-down).

—*Top-down spproach.* This phase starts working from the root (sink) node.[4]

Based on its resource constraints, the root node either decides to host the

[4]Note that there could be multiple root nodes (one for each sink in the application task graph). This phase of the heuristic works in parallel starting from each such root.

root fusion function or not. If it decides to host the root fusion function, then it delegates the mapping of the children subtrees to its neighbors (either in the same level or the next level). This role assumption and delegation of the subtrees progresses until all the fusion functions are placed on network nodes. Relay nodes (as required) are used to connect up the sources to the network nodes that are hosting the fusion points of the task graph closest to the leaves. It is possible that in this initial phase a node may be asked to host multiple fusion functions, and it is also possible that a node may decide not to host any fusion point but simply pass the incoming subtree on to one of its neighbors. In choosing delegates for the subtrees, the "resource richness" of the neighbors is taken into account. This recursive tree building ends at the data source nodes, that is, the leaves of the task graph. The completion notification of the tree building phase recursively bubbles up the tree from the sources to the root.

—*Bottom-up approach.* As with the top-down approach, this phase starts at the root node. However, the intent is to assign fusion points close to the sources due to the data contracting nature of the task graph. To understand how this works, suppose the input task graph is a tree with depth k. The leaf nodes are the data sources. Their parents are the fusion points that are $k-1$ level distant from the root. For each fusion function at layer $(k-1)$ the root node asks an appropriate data source (commensurate with the task graph connectivity and avoiding duplication) to select the network nodes to host the set of fusion functions at level $k-1$ that are its consumers. To select s fusion node from among its neighbors, a data source would prefer a node that is closer to the root node in terms of hop count. The selected nodes at level $k-1$ report their identity to the root node. Once all the fusion functions at level $(k-1)$ have been thus mapped, the root node recursively maps the fusion functions at the next higher levels of the tree in a similar way. As with the top-down approach, relay nodes (as required) bridge the first-level fusion points of the task graph to the root.

For the bottom-up approach, the root node plays an active role in mapping the fusion point. The alternative would be to flood the complete task graph to the leaf nodes and other participating network nodes close to the leaves. Since the data sources can be quite far from the root, such flooding can be quite expensive. Therefore, in the bottom-up approach the root node explicitly contacts a network node that is hosting a fusion point at a given level to map its parents to network nodes. This was not necessary for the top-down approach, where the root node needed to contact only its neighbor nodes for the mapping of the subtrees.

4.4.2 *Optimization Phase.* Upon completion of the initialization phase, the root node starts a recursive wave of "start optimization phase" message to the nodes of the network. This phase is intended to refine the initial mapping *before* the application starts. The input to this phase is the *expected* data flows between the fusion points and the application-specified cost function. During this phase, a node that is hosting a fusion point is responsible for either continuing to play

that role or transferring it to one of its neighbors. The decision for role transfer is taken by a node (that hosts that role) based on local information. With a certain predetermined periodicity, a node hosting a fusion function informs its neighbors of its role and its health (a metric determined by the application-specified cost function). Upon receiving such a message, the recipient computes its own health for hosting that role. If the receiving node determines that it is in better health to play that role, an "intention to host" message is sent to the sender. If the original sender receives one or more intention messages from its neighbors, the role is transferred to the neighbor with the best health. The overall health of the overlay network improves with every such role transfer.

The optimization phase is time bound and is a tunable parameter of the role assignment heuristic. Upon the expiration of the preset time bound, the root node starts a recursive wave of "end optimization phase" messages in the network. Each node is responsible in making sure that it is in a consistent state (for example, it is not in the middle of a role transfer to a neighbor) before propagating the wave down the network. Once this message reaches the sources, this phase is over, and the application starts with data production by the sources.

4.4.3 *Maintenance Phase.* The maintenance phase has the same functionality as the optimization phase. The input to this phase are the *actual* data flows observed between the fusion points and the application-specified cost function. In this phase, nodes periodically exchange role and health information and trigger role transfers if necessary. The application dynamics and/or the network dynamics leads to such role transfers. Any such heuristic that relies entirely on local information is prone to getting caught in local minima (see Section 4.5). To reduce such occurrences, the maintenance phase periodically increases the size of the set of neighbors a node interacts with for potential role transfer opportunities. The overhead of the maintenance phase depends on the *periodicity* with which the neighborhood size is increased and the *broadcast radius* of the neighborhood. These two factors are exposed as tunable parameters to the application.

4.4.4 *Support for Node Failure and Recovery.* Failures can occur at any of the nodes of the sensor network: sources, sinks, or fusion points. A sink failure is directly felt by the application and it is up to the semantics of the application whether it can continue despite such a failure. A source or intermediate fusion point failure is much more subtle. Such a failure affects the execution of the fusion functions hosted by the nodes downstream from the failed node that are expecting input from it. The manifestation of the original node failure is the unsuccessful execution of the fusion functions at these downstream nodes. As we mentioned in Section 3.5, the fusion module has APIs to report fusion function failure to the application. The placement and fusion modules together allow recovery of the application from failure as detailed below.

After the first two phases of the role assignment heuristic, any node that is hosting a fusion point knows only the nodes that are consuming data from it. That is, any given node does not know the identity of the network nodes that

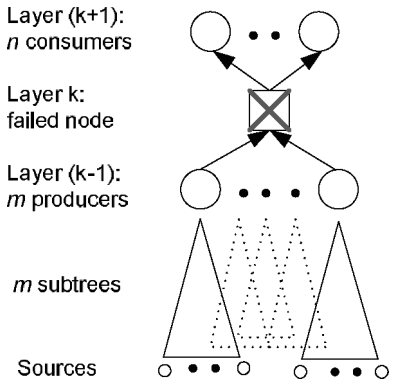

Fig. 6. An example failure scenario showing a task graph overlaid on the network. An edge in this figure may physically comprise of multiple network links. Every fusion point has only local information, namely, identities of its immediate consumers.

are producing data for that node. In fact, due to the localized nature of the role transfer decisions, no single node in the network has complete knowledge of the physical deployment of the task graph nor even the complete task graph. This poses a challenge in terms of recovering from failure. Fortunately, the root node has full knowledge of the task graph that has been deployed. We describe how this knowledge is exploited in dealing with node failure and recovery.

Basically, the root node plays the role of an arbiter to help resurrect a failed fusion point. Note that any data and state of the application that is lost at the failed node has to be reconstructed by the application. The recovery procedure explained below is to simply reestablish the complete task graph on the sensor network subsequent to a failure.

The *recovery procedure* functions as follows: let the failed fusion point be at level k of the task graph, with m input producer nodes at level $(k-1)$ to this fusion point, and n output consumer nodes at level $(k+1)$ awaiting output from this fusion point (see Figure 6). The following three steps are involved in the recovery procedure:

(1) *Identifying the consumers.* The n consumer nodes at level $(k+1)$ generate fusion function failure messages (see Section 3.5). These messages along with the IDs of the consumers are propagated through the network until they reach the root node.

(2) *Identifying the producers.* Since there are m producers for the failed node, there are correspondingly m subtrees under the failed fusion function. The root identifies these subtrees and the data sources at the leaf of these subtrees, respectively, by parsing the application task graph. For each of these m subtrees, the root node selects *one* data source at the leaf level. The m selected data sources generate a *probe* message each (with information about the failed fusion function). These messages are propagated through the network until they reach the m nodes at level $(k-1)$. These m nodes (which are the producers of data for the failed fusion function) report their respective identities to the root node.

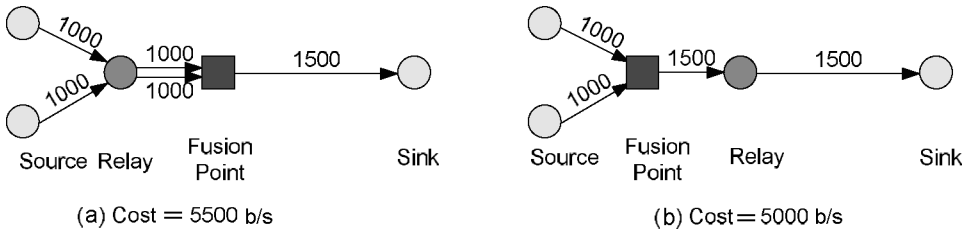

Fig. 7. Linear optimization.

(3) *Replacing the failed fusion point.* At this point, the root node knows the physical identities of the consumers and producers to the failed fusion point. It requests one of them to choose a candidate neighbor node for hosting the failed fusion function. The chosen node informs the producers and consumers of the role it has assumed and starts the fusion function. This completes the recovery procedure.

Needless to say, during this recovery process the consumer and producer nodes (once identified) do not attempt to do any role transfers. Also, this recovery procedure is not resilient to failures of the producers or the consumers (during recovery).

4.5 Analysis of the Role Assignment Heuristic

For the class of applications and the 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 such dynamic adaptations indeed 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 reevaluation of the health of the nodes at regular intervals, such occurrences are 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 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 the 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 facilitates two types of optimizations:

—*Linear optimization*. If all the inputs to a fusion node are coming via a relay node (Figure 7(a)), and there is data contraction at the fusion point, then the relay node becomes the new fusion node, and the old fusion node transfers its responsibility to the new one (Figure 7(b)). In this case, the fusion point is

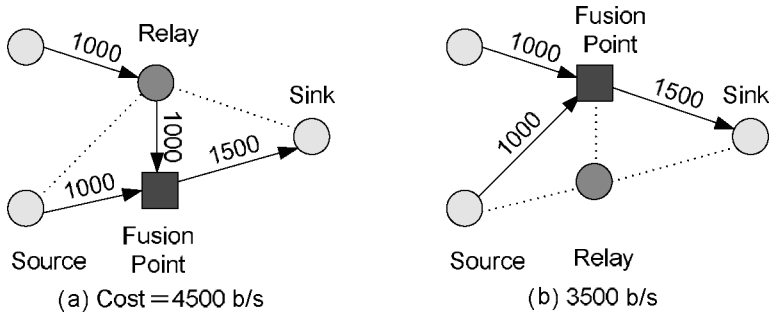

Fig. 8. Triangular optimization.

moving away from the sink, and coming closer to the data sources. Similarly, if the output of the fusion node is going to a relay node, and there is data expansion, once again the relay node acts as the new fusion node. In this case, the fusion point is coming closer to the sink and moving away from the data sources.

—*Triangular optimization*. If there are multiple paths for inputs to reach a fusion point (Figure 8(a)), and if there is data contraction at the fusion node, then a triangular optimization takes place (Figure 8(b)) to bring the fusion point closer to the data sources; in the event of data expansion, the fusion point moves towards the sinks. The original fusion point node becomes a relay node.

5. IMPLEMENTATION

We use iPAQ running Linux as our conceptualization of a "future sensor node" in the implementation. DFuse is implemented as a multithreaded runtime system, assuming infrastructure support for reliable timestamped data transport through the sensor network. The infrastructural requirements are met by a programming system called Stampede [Ramachandran et al. 1999; Adhikari et al. 2002]. 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.

5.1 Fusion Module

The fusion module consists of the shaded components shown in Figure 9. It is implemented 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

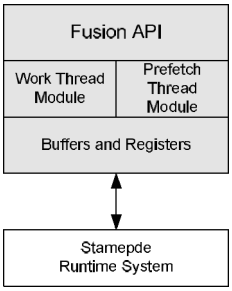

Fig. 9. Fusion module components.

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

Stampede's runtime system sits on top of a reliable UDP layer in Linux. Therefore, there is no support for adaptive multihop ad hoc routing in the current implementation. Further, there is no support for gathering *real* health information of the nodes. For the purposes of evaluation, we have adopted a novel combined implementation/simulation approach. The fusion module is a real implementation on a farm of iPAQs as detailed in the previous subsection. The placement module is an event-driven simulation of the three phases of the role assignment algorithm described in Section 4. 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 models the health of the sensor network nodes. It currently assumes an ideal routing layer (every node knows a route to every other node) and an ideal MAC layer (no contention). Further, path reliability for MT1 and MT2 cost function evaluation is assumed to be 1 in the simulation. However, it should be clear that these assumptions are not inherent in the DFuse architecture; they are made only to simplify the simulator implementation and evaluation.

We have implemented an interface between the the fusion module implementation and the placement module simulation. This interface facilitates (a) collecting the actual data rates of the sensor nodes experienced by the application running on the implementation and reporting them to the placement module simulation, and (b) communicating to the fusion module (and effecting through the DFuse APIs) dynamic task graph instantiation, role changes based on the health of the nodes, and fusion channel migration.

6. EVALUATION

We have performed an evaluation of the fusion and placement modules of the DFuse architecture at two different levels: microbenchmarks to quantify the overhead of the primitive operations of the fusion API including channel creation, attachments/detachments, migration, and I/O; and 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.

We first describe the microbenchmarks and then the results with the placement module, both for a small network of 12 iPAQs. Next, we will describe the

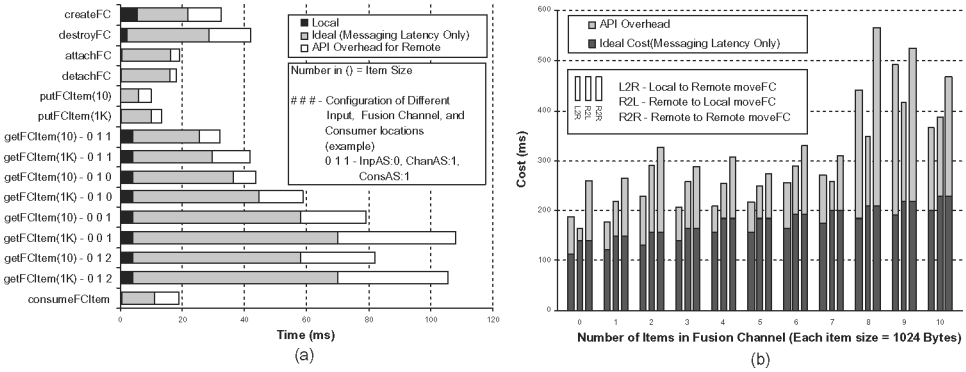

Fig. 10. (a) Fusion channel APIs' cost; (b) fusion channel migration (moveFC) cost.

simulation results for a network of more than a thousand nodes, where we will only look at different aspects of the placement algorithm.

6.1 Micromeasurements of Fusion API

Figure 10 shows the cost of the DFuse API. In part (a), each API cost has three fields—*local*, *ideal*, and *API overhead*. Local 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 the 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 10 items and one primitive Stampede channel as input. Reported latencies are the averages of 1000 iterations.

On our iPAQ farm, netperf [Netperf 2003] indicates a minimum UDP roundtrip latency of 4.7 ms, and from 2- to 2.5-Mb/s maximum unidirectional streaming TCP bandwidth. Table I 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 10(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 2 Mb/s throughput. Comparing these ideal costs in Figure 10(a) with the actual total cost illustrates reasonable overhead for our DFuse API implementation. The maximum cost of operations on a local node is 5.3 ms. For operations on remote nodes, API overhead is less than 74.5% of the ideal cost. For operations with more than 20 ms 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.

Table I. Number of Round Trips and Message Overhead of DFuse (See Figure 10 for `getFCItem` and `moveFC` configuration legends.)

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

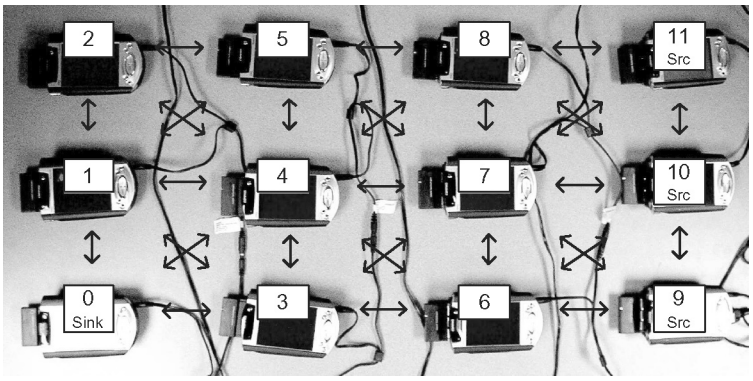

Fig. 11. iPAQ Farm Experiment Setup. An arrow represents that two iPAQs are mutually reachable in one hop.

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 the amount of data to be moved. Our analysis in Figure 10(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).

6.2 Application Level Measurements of DFuse

We have implemented the fusion application (Figure 1) using the fusion API and deployed it on the iPAQ farm. Figure 11 shows the topological view of the iPAQ farm used for the fusion application deployment. It consists of 12 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. We have tuned the fusion application to generate data at consistent rates as shown in Figure 1, with x equal to 6 kB/min. This is equivalent to a scenario where cameras scan the environment once every minute, and produce images ranging in size from 6 to 12 kB after compression.

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 (see Section 6.2.1) 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.

6.2.1 *Power Model.* The network is organized as the grid shown in Figure 11. For any two nodes, the routing module returns the path with a minimum 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 on the amount of data a node transmits or receives. The power consumption corresponds to ORiNOCO 802.11b PC card specification [ori 2003]. Our current power model only includes network communication costs. After finding an initial mapping (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 100 ms to determine if role transfer is warranted by the cost function. After optimization, the algorithm runs in maintenance phase until the network becomes partitioned (connectivity can no longer be supported among all the fusion points of the task graph). During the maintenance phase, role transfer decisions are evaluated every 50 s. The role transfers are invoked only when the health improves by a threshold of 5%.

6.2.2 *Experimental Results.* Figure 12 shows the network traffic per unit time (sum of the transmission rate of every network node) for the cost functions discussed in Section 4.2. 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 the 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, that is, 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

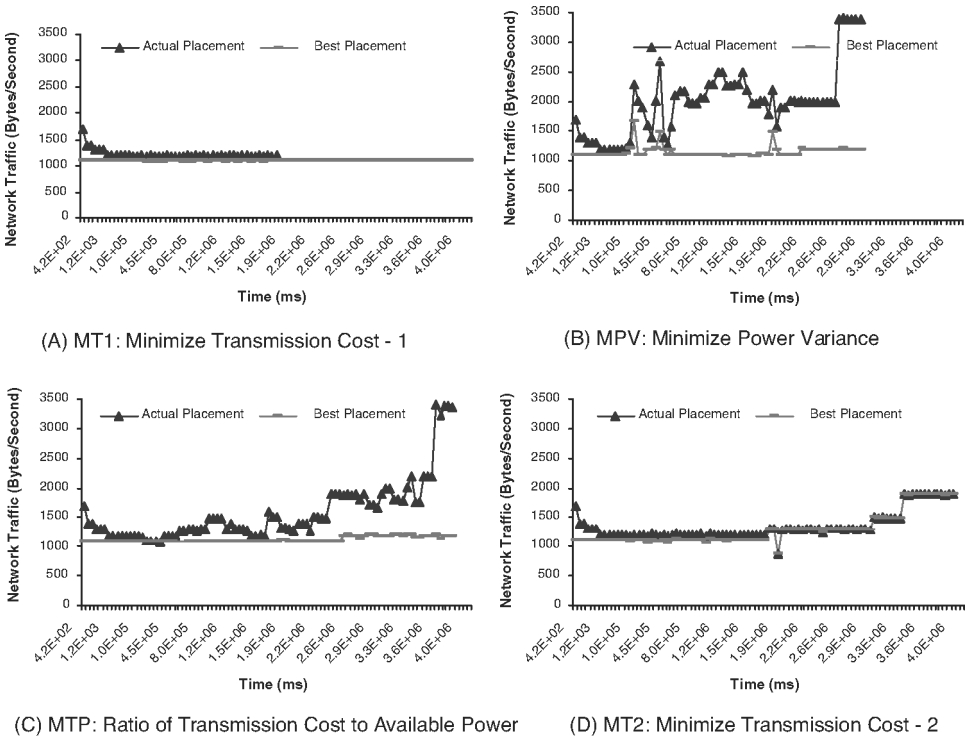

Fig. 12. The network traffic timeline for different cost functions. The x axis shows the application runtime and the y axis shows the total amount of data transmission per unit time.

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, with an 110% increase compared to 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 the same as the best possible rate for the latter half of the run. MPV performs the worst while MTP has comparable network lifetime as MT2. Figure 12 also shows that running the optimization phase before instantiating the application improves the total transmission rate by 34% compared to the initial naive placement.

Though MPV does not provide comparably good network lifetime (Figure 12(b)), it provides the best (least) power variance compared to other cost functions (Figure 13(a)). 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 13(b)). These results show that the choice of the cost function depends on the application context and the network condition. If, for an application, role transfer is complex and expensive but network power variance is not an issue, then MT2 is preferred. However, if network power variance needs to be minimized and role transfer is inexpensive, MTP is preferred.

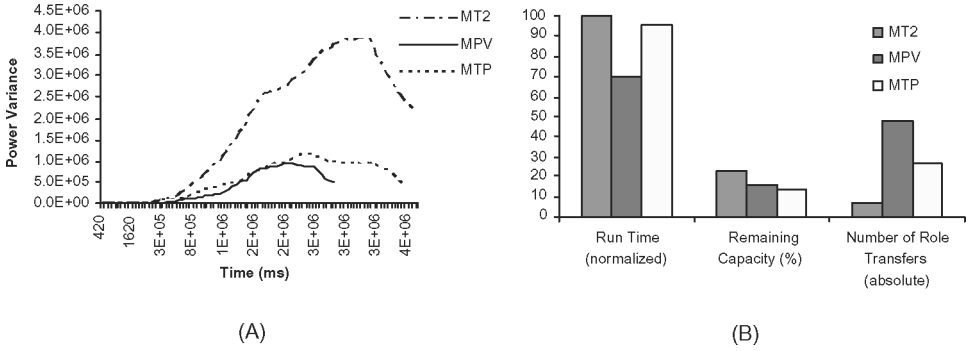

Fig. 13. Comparison of different cost functions. Application runtime is normalized to the best case (MT2), and total remaining power is presented as the percentage of the initial power.

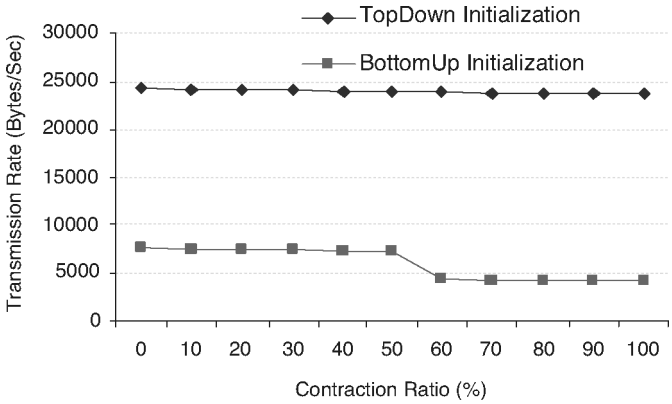

Fig. 14. Effect of data contraction on initial mapping quality. Network grid size is 32×32.

6.3 Simulation-Based Study of Large Networks and Applications

To evaluate DFuse's scalability, we employ *MSSN*, an event driven simulator of the DFuse fusion channel and placement module. We report on three studies performed with this simulator: (a) control overhead of initialization phase algorithms, (b) performance of the role assignment algorithm for a small application on a large network, and (c) scalability of the role assignment heuristic for a large application.

6.3.1 *Initialization Phase Algorithms.* We have simulated the two initialization algorithms to study their control overhead properties and initialization quality. The control overhead relates directly with the time and energy consumed during the initialization phase. The variables are the network graph size, the input task graph size, and the average distance from sink to the sources. The results show that the control overhead, in terms of the total number of control messages, depends mainly on the distance between the sink and the data sources, and it varies only a little with the size of the task graph. Also, Figures 14 and 15 show that, compared to the top-down algorithm, the bottom-up approach leads to much better initial mapping in terms of total transmission

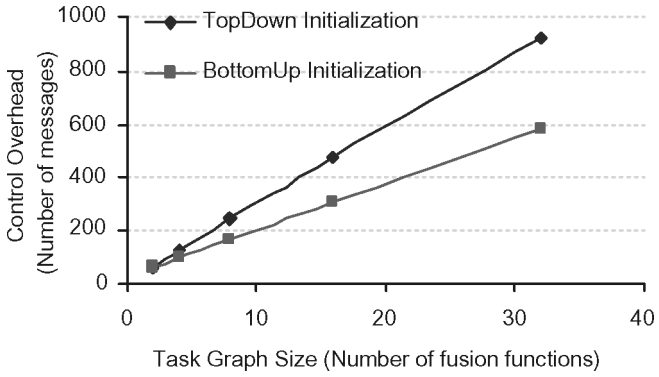

Fig. 15. Effect of input task graph size upon total transmission cost of initial mapping. Network grid size is 32 × 32.

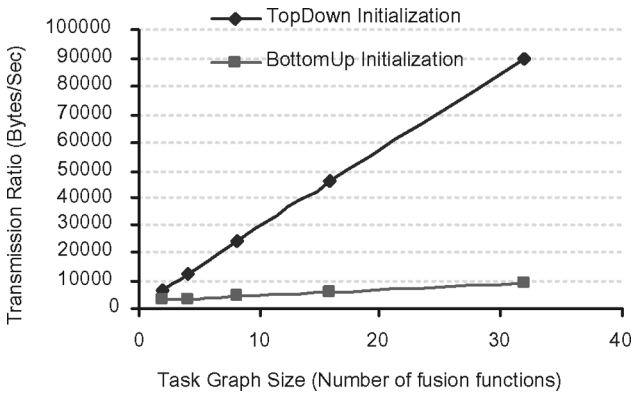

Fig. 16. Effect of input task graph size upon control overhead of initialization algorithms. Network grid size is 32 × 32.

cost. Even when the task graph has a considerable number of data expansion fusion points (i.e., the contraction ratio less than 50% in Figure 14), the bottom-up algorithm gives better initial mapping than the top-down algorithm. Figures 16 and 17 show that the control overhead (number of control messages) for the two initialization algorithms increase linearly with the increase in task graph size or with the increase in sink to data source distance. Among the two algorithms, the bottom-up algorithm has the lower message overhead.

6.3.2 *Single Fusion Point Scalability Study.* To determine the performance with respect to the optimal transmission cost of the placement module as the number of network nodes increases, we used a simple, single fusion point (image collage) application model with two camera sources and a single sink. We scaled the network topology from a 4 × 4 to a 32 × 32 grid, with the initial camera placements mapped randomly along one edge (to distinct nodes), and the sink placed in one of the two opposite corners of the grid. Simulation began in the maintenance phase, with the collage fusion channel mapped to a random node not chosen for a camera or sink. We tailored the cameras

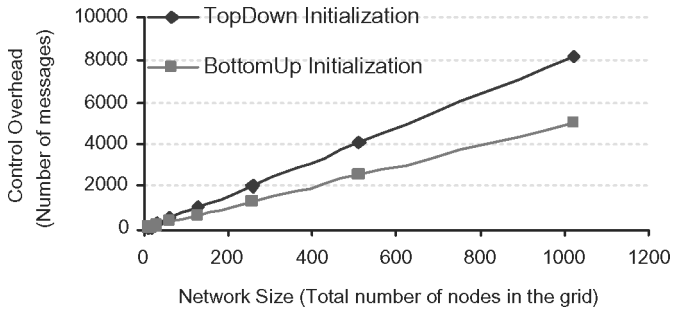

Fig. 17. Effect of network size upon control overhead of the initialization algorithms. Average distance of the sink from the sources is kept in proportion to the grid width.

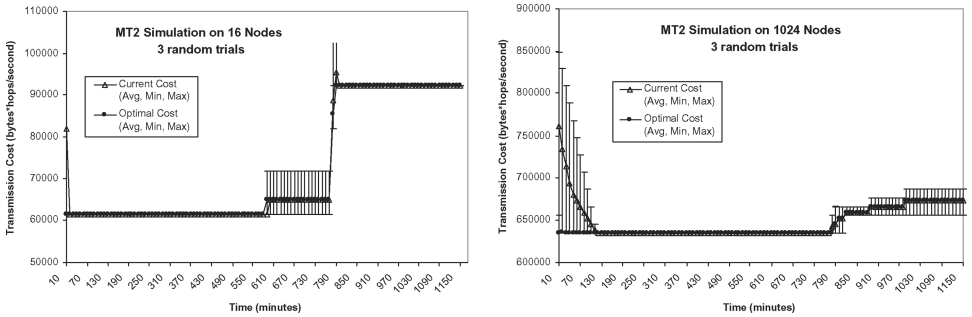

Fig. 18. Results of three simulations using MT2 on 4 × 4 and 32 × 32 grids (16 and 1024 nodes).

each to produce 2-kB images five times per simulated second, and tuned the collage fusion function to simply concatenate the two input images, yielding 20 kB per simulated second throughput to the sink, barring any migration or communication-induced latencies. These rates are 100 times faster than used in our iPAQ farm, but still well within OriNoCo 11Mbps bandwidth, barring excessive collisions. For the 4 × 4 grid (initially fully powered, with a node having a 1-hop distance to nearest neighbors only, including diagonal neighbors), the transmission cost for any of our random camera placements, once the collage was optimally placed, was 60 kB*hops/s, assuming no collisions or other latencies. Similarly, for the 32 × 32 grid, the initial optimal transmission cost at maximum possible throughput was 620 kB*hops/s, for any of our random camera placements.

6.3.2.1 *Single Fusion Point Scalability Results.* Figures 18, 19, and 20 show, for each of the MT2, MPV, and MTP cost functions, and for each of two topology scales, 4 × 4 and 32 × 32 nodes, the application lifetime and runtime transmission cost over time produced by the local placement heuristic, relative to the current transmission cost of an optimal, globally decided mapping. For each cost function and topology scale, we used three trials with varying initial random selections of camera and collage placements. The graphs indicate the average, minimum, and maximum, costs across these trials, for each of the actual and optimal mappings. As a baseline verification, the initial optimal

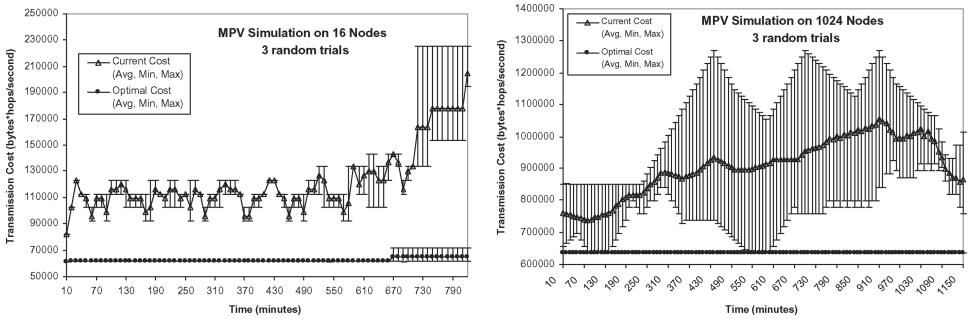

Fig. 19. Results of three simulations using MPV on 4 × 4 and 32 × 32 grids (16 and 1024 nodes).

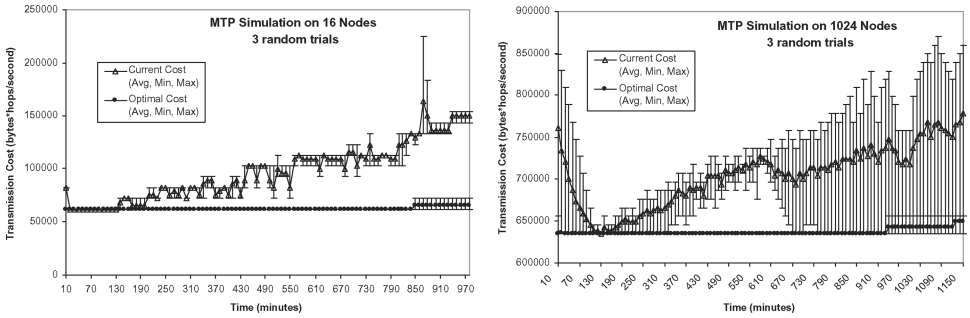

Fig. 20. Results of three simulations using MTP on 4 × 4 and 32 × 32 grids (16 and 1024 nodes).

transmission costs depicted in these figures agree with the predicted 60 kB*hops/s for the 4 × 4 topology and 620 kB*hops/s for the 32 × 32 topology. Furthermore, as nodes die, even the optimal costs increase as redundant, but more expensive resources are leveraged by the placement heuristic.

For MT2 (Figure 18), the actual placement corresponds closely to the optimal placements for the 4 × 4 topology, and rapidly converges to optimal mapping followed by slow degradation for 32 × 32 topology. For MPV (Figure 19), the transmission cost for the majority of network lifetime is approximately 50–100% more than an optimal mapping. Minimizing power variance leads to frequent migrations and transmission cost-agnostic behavior, and hence the high variance in transmission cost over the life of the network. For MTP (Figure 20), actual placement rapidly converges to an optimal mapping, followed by moderate degradation and increasing variance as transmission cost is traded off for greater mapping stability as battery variances increase. The performance rankings of each of these cost functions remain the same as the network scale increases from the iPAQ-farm to the 1024 node simulation: MT2 achieves very close to optimal transmission cost, even in the 32 × 32 node simulation, followed by MTP and then MPV. MT2 and MTP consistently achieve the greatest application lifetime, although MPV achieves the same lifetime in one case: When the simple application is simulated on a large, 32 × 32 sensor network topology, the lifetime of the network is limited by the energy in "hot spot" critical nodes adjacent to the sink. Even though costly migrations and high-transmission cost

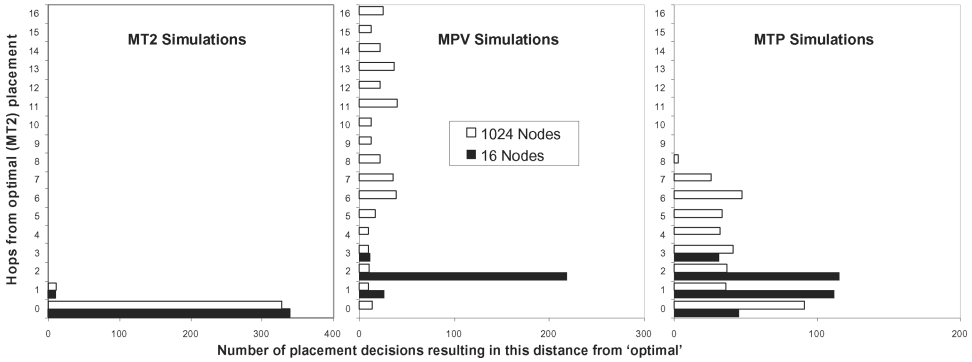

Fig. 21. Proximity to optimal (transmission cost) mapping in terms of number of hops an instant migration would need to take from current mapping, evaluated at every placement heuristic execution.

mappings result from using MPV, there are enough redundant resources in the network to let the application survive until the sink's neighbors die.

Figure 21 gives overall summaries of the proximity of the locally decided collage placement to the globally optimal transmission cost placement, for each of the cost functions and network scales studied. For MT2, 0 hops dominate because it performs close to optimal. This figure also indicates that examining neighbors further away than 1 hop during placement decisions may not be necessary for MT2, as an optimal mapping is never further than a single hop away from the current mapping in our experiments. MPV usage results in poor (large) numbers of hops to get to the best transmission cost mapping because MPV does not attempt to optimize transmission cost. Although a multihop placement heuristic may assist in minimizing battery variance for large networks, it makes no practical sense to do this because the transmission cost will remain unoptimized for MPV while the migration cost would increase, further reducing lifetime performance for MPV. In essence, battery variance might be minimized, but no real work beyond migration would be done. MTP has generally close proximity to the best transmission cost mapping, closer than MPV but not as close as MT2. As this cost function partially optimizes for transmission cost, a multihop placement heuristic may assist in reducing transmission cost and increasing application lifetime. However, in the next section we show that MT2, MTP, and MPV perform well in terms of cost relative to our best "optimal" cost as scale increases, countering the need for a multihop placement heuristic.

6.3.3 *Large Application Scalability Study.* We extended MSSN to include three general-purpose *oracles* to determine "optimal"[5] cost in the context of MT2, MPV, or MTP cost function. For small applications, we implemented a general brute force oracle, BF, that evaluates all n^f possible mappings. We

[5]Applications with f fusion points mappable to n sensor nodes have n^f possible mappings, making determination of optimality for realistic applications having more than 10 fusion points impractical, especially as the application scale increases. We approach the problem of determining optimality by using approximation algorithms with feasible complexity.

Algorithm 6.1: SA-ORACLE($type, channels, nodes, costFunction(), T_{cold}$)

```
for each c ∈ channels
  do c.host ← random node ∈ nodes
currentCost ← Σ_{c∈channels} costFunction(c)
T ← (−average(Δcost for each neighbor with higher cost)/ln(0.80))
if type = SA1
  then iterationsPerTemperature ← |channels|^2
  else iterationsPerTemperature ← |channels| * |nodes|//SA2
temperatureCount ← 0
repeat
 oldCost ← currentCost
 for i ← 1 to iterationsPerTemperature
   do { candidate ← random neighbor of current state
        if (Δcost(current to candidate) < 0)
          then current state ← candidate//Accept
        else if random number ∈ [0, 1] < exp(−Δcost/T)
          then current state ← candidate//Accept
        else //Reject
 T ← T * 0.95
 temperatureCount ← temperatureCount + 1
until (type = SA1 and
       (T < T_cold or temperatureCount > |nodes|^2 or currentCost = oldCost))
      or (type = SA2 and T < T_cold)
```

also employed two different tunings, SA1 and SA2 of a simulated annealing algorithm in the form of two approximating oracles.

In our SA1 and SA2 oracles, the state of our system is described by the current mapping of fusion channels onto nodes, where a node may host multiple fusion channels simultaneously. The oracles take the application-specified DFuse cost function summed over all f fusion channels as the objective function. We use two basic types of randomized permutations to reach a *neighbor* state: picking a new host at random for a fusion channel chosen at random, or swapping the hosts of two randomly chosen fusion channels. We choose a sufficiently high initial temperature in the units of the global cost function by solving for T, given an input parameter for the probability of accepting an average cost increase initially (80%), and by averaging any cost increases across all neighbors of the initial state. Each successive temperature is a fraction (95%) of the previous temperature. SA1 and SA2 differ in the number of perturbations considered at each temperature level, and in the definition of the frozen state. SA1 is tuned to perform a significantly smaller search than SA2 for large state spaces. SA1 evaluates f^2 perturbations at each temperature level and terminates annealing when $T < T_{cold}$, the number of temperature reductions has exceeded n^2 or when the cost at the end of a temperature level is the same as at the beginning. SA2 evaluates $f * n$ perturbations at each temperature level and terminates annealing only when $T < T_{cold}$. By empirically observing when SA2 costs do

Table II. Scalable Application Model: Fusion Functions

Fusion Function	Description	Footprint (kb)		
		Input	Output	Runtime
Collage	Combines two inputs into one output	56*2	112	0
Select	Selects one of two inputs	56*2	56	0
EdgeDetect	Annotates one input	56	56	0
MotionDetect	Compares one input to previous	56	56	94

not change as T decreases for more than four successive temperature levels, we define T_{cold} for each of the DFuse cost functions. Pseudocode for our SA1 and SA2 oracles is given in Algorithm 6.1.

For this large application scalability study, we constructed a scalable application workload model based on workloads used in previous MSSN studies [Wolenetz et al. 2004]. The task graph consisted of a tree composed from subgraphs comprised of fusion channels that combine two inputs into one output stream and fusion channels that transform one input into one output stream. We only considered the time and energy used to transmit inputs, outputs, and channel buffer and state migration when determining resource consumption. Table II shows these data sizes for each function. For example, the *Select* function takes two 56-kb inputs and outputs 56 kb each iteration, and the *MotionDetect* channel migrates 94 kb in addition to any buffer state. We configured the sensor network to be a 256-node 16 $\times$ 16 grid with rectilinear and diagonal nearest-neighbor connectivity and initially mapped the application onto it in a form much like a tree. The number of fusion points was determined by the number of camera sources used, thereby easily scaling the application. Figure 22 presents a sample two-camera, three-fusion-point application's initial mapping (on a much smaller grid here). Larger application scales are formed by combining subgraphs like these together into a larger tree.

6.3.3.1 *Large Application Scalability Results.* Figures 23, 24, and 25 show, for each of the MT2, MTP, and MPV cost functions, and for several larger-scale applications, the performance and runtime complexity of the DFuse local placement heuristic relative to BF, SA1, and SA2 oracles. For each cost function, oracle, and application scale, we performed 10 trials using different random seeds for the simulated annealing oracles and for the random choice among similar arity fusion functions when generating the application task graphs. The graphs indicate the average and ± 1 standard deviation across trials. For example, Figure 24 shows that DFuse achieved an average cost 2.2 times higher than SA2 for MTP with a nine-camera, four-fusion-point application, but only searched about 1/10,000 of the mappings that SA2 searched. It is computationally infeasible to run BF for large application scales, but we include it for the three-fusion-point, two-camera configuration as an optimal baseline.

For MT2, Figure 23 shows that DFuse *outperformed* SA1 at all application scales studied. This was likely due to the relatively small portion of the state space that contained globally minimum transmission cost mappings. SA1 searched f^2 mappings per temperature, and could terminate temperature reductions much sooner than SA2, resulting in searching a very small portion of

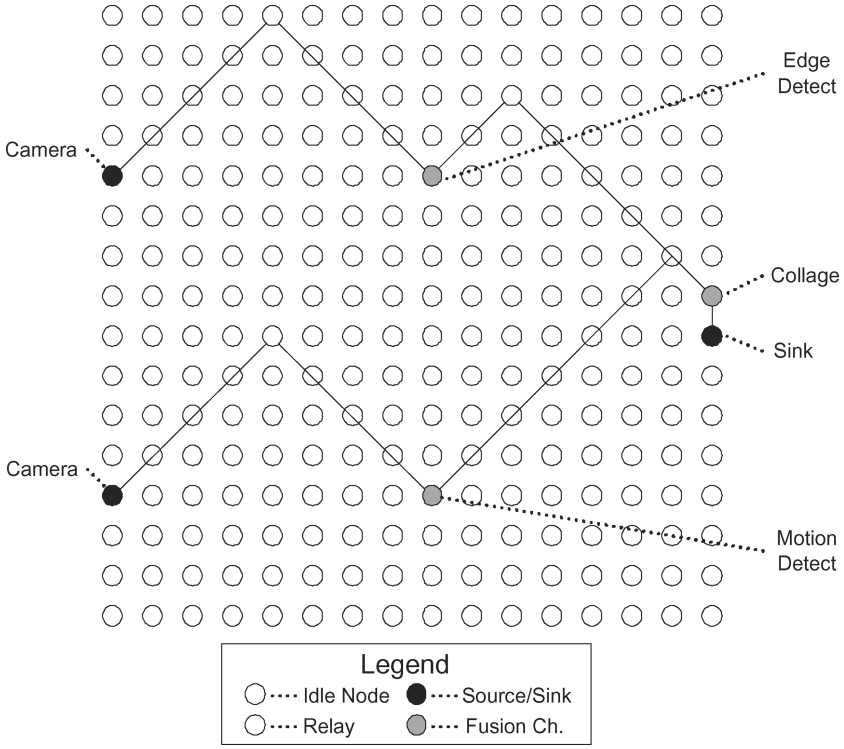

Fig. 22. Scalable application model: sample three-fusion-point application on 16 × 16 grid.

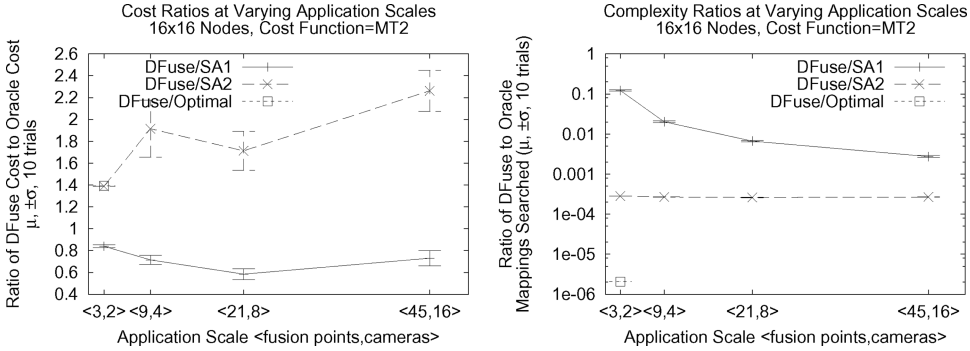

Fig. 23. Large application scalability results for MT2.

the search space (about 1/100,000 for the smallest application scale studied). SA2 performed optimally at this small application scale. For MT2, DFuse performed about the same relative to SA1 and SA2 as the scale increased. While it is impractical to implement a global algorithm such as SA2 in our sensor networks, we see that it would perform only about twice as well as DFuse at the cost of much larger runtime complexity.

For MTP, Figure 24 shows that DFuse approached both SA1 and SA2 and the standard deviation decreased as the scale increased. Inspection of simulation

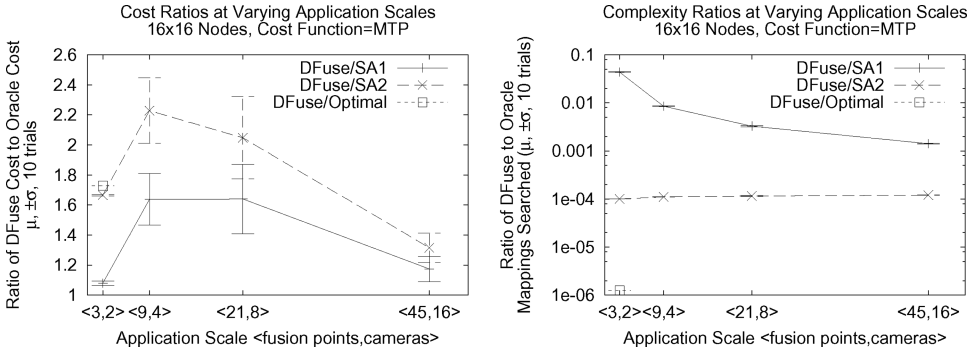

Fig. 24. Large application scalability results for MTP.

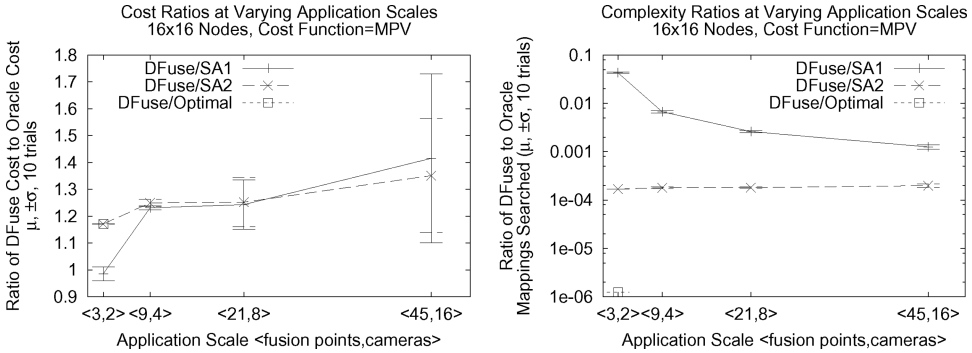

Fig. 25. Large application scalability results for MPV.

traces also indicated that DFuse performed about the same amount of searching and SA2 performed about twice the amount of searching for MTP versus MT2. While it is infeasible to perform a brute force search at larger scales, it is quite interesting to see that DFuse approached our best feasibly computed optimal global oracle at larger scales for MTP.

For MPV, Figure 25 shows that SA1 and SA2 performed similarly as the scale increased (similar means, similar increasing standard deviations). The existence of similar means between SA1 and SA2 confirms the intuition that using MPV results in a large number of global mappings close to the optimum mapping. For smaller applications, many of the nodes will remain idle and therefore have equivalent MPV costs. As the application scale increases, the standard deviation increases, reflecting the larger number of battery level equivalence classes across the nodes. These results indicate that we could expect good MPV performance on average as the application scale increases because SA1 and SA2 performed about 1.1 to 1.3 times as well as DFuse on average across all the scales studied.

6.3.4 *Placement Heuristic Results Summary.* Our simulation experiments with larger networks and applications confirm our intuitions about the

scalability of the role assignment heuristic:

—As the network is scaled up to 1024 nodes for a single-fusion-point application, all three cost functions behave similarly with respect to each other in terms of transmission cost relative to the current optimal transmission cost: MT2 performs close to optimal, followed by MTP; MPV performs the worst.

—Scalability results, in terms of application lifetime, confirm that more complex cost functions that incorporate transmission energy costs (MT2 and MTP) are able to extend lifetimes better than MPV, and that increasing redundant energy resources (nodes) in WASN can further extend lifetimes when using the migration facility of DFuse.

—For large topologies and the small applications studied, we find that the energy of the neighbors of the fusion application's powered sources and sinks typically determines the lifetime of the application. In this case, there are so many redundant in-network nodes that the lifetime is limited by the fixed location of application endpoints (sources and sinks are assumed to not migrate).

—Examining neighbors further than 1 hop away from a fusion point during a local placement decision to achieve good transmission cost performance is unnecessary for MT2 and MTP, and nonsensical for MPV.

—Most importantly, DFuse's placement heuristic performs well with respect to our best feasibly computed globally optimal performance as the application scale increases for each of the cost functions. We show that heuristic performance actually improves relative to this optimal for MTP as application scale increases.

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 the min, max, or average reading of sensors [Madden et al. 2002; Intanagonwiwat et al. 2000]. While these experiments and others have motivated the need for a good role assignment approach, they did not use a dynamic heuristic for the role assignment and their static role assignment approach is not amenable to streaming media applications.

Using a dataflow (task) graph to model distributed applications is a common technique in streaming databases, computer vision, and robotics [Cherniack et al. 2003; Viola and Jones 2001; Rehg et al. 1997]. DFuse employs a script-based interface to express task graph applications over the network similar to SensorWare [Boulis et al. 2003]. 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 [Madden et al. 2002], 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. TAG assumes a static mapping of roles to the network, that is, a routing tree is built based on the network topology and the query on hand.

Database researchers have recently examined techniques for optimizing continuous queries over streaming data. In many cases, the systems are centralized, and all of the processing occurs on one machine [Chen et al. 2000; Motwani et al. 2003; Chandrasekaran et al. 2003]. The need to schedule operators over distributed, heterogeneous, resource-constrained devices presents fundamentally different challenges. More recently, researchers have examined distributed stream processing, where computations are pushed into the network much as in DFuse [Ahamad and Cetintemel 2004; Pietzuch et al. 2006]. Unlike DFuse, these systems usually assume powerful in-network computation nodes, and attempt to minimize individual metrics like network usage and maximize throughput. For example, SBON [Pietzuch et al. 2006] expects every node to run the Vivaldi algorithm and generate a latency cost estimate of the whole network to facilitate operator placement. DFuse allows a variety of optimization functions so that the processing can be carried out on lower-capability, energy-constrained devices.

Moreover, a common technique in data stream systems is to reorder filter and join (e.g., fusion) operations, according to the rules of relational algebra, in order to optimize network placement [Srivastava et al. 2005]. Similarly, systems such as Aurora* [Cherniack et al. 2003] allow "box sliding" (e.g., moving operators) and "box splitting" (e.g., splitting computation between two copies of an operator), but assume that the semantics of the operator are known in order to determine which reconfigurations are allowable. Our system is meant to deal with task graphs where the computations are complex black boxes, and where operations often cannot be reordered.

Some work has looked at in-network processing of streams using resource-constrained devices [Madden et al. 2002] but have focused primarily on data aggregation. Our work also addresses data fusion, which utilizes more complex computations on specific streams and requires novel role assignment and migration techniques. Java object migration in MagnetOS [Liu et al. 2005] is similar in spirit to role migration in DFuse, but MagnetOS does not allow an application to specify its own cost function to influence the migration behavior.

Other areas where distributed tree optimization has been looked at include distributed asynchronous multicast [Bhattacharya et al. 2003], and hierarchical cache tree optimization on the Internet [Tewari et al. 1999]. However, these optimizations are done only with respect to the physical network properties, and they do not need to consider any input task graph characteristics (fusion function dependencies, black-box treatment, etc.) and sensor network requirements (dynamism, energy). One recent work in composing streaming applications over peer-to-peer environments [Gu and Nahrstedt 2006] has

looked at an optimization problem of our flavor, but the proposed algorithm adapts a mapping only in lieu of node failures, and thus it is not suitable for WASN environments that need much more dynamic adaptations.

Recent research in power-aware routing for mobile ad hoc networks [Singh et al. 1998; Jae-Hwan and Leandros 2000] has proposed 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 article, we are looking into how the routing protocol information can be used to define more flexible cost functions.

8. FUTURE WORK

There are several avenues for future research:

—While the applications targeted by the DFuse architecture are general (see Section 2), the role assignment algorithm assumes knowledge of the data sources for deploying a task graph on the sensor network. This assumption is limiting for addressing data-centric queries where such knowledge is not available a priori. We are exploring different ways to handle this additional source of dynamism in task graph deployment. One possible approach is to have an interest-dissemination phase before the initialization 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 [Intanagonwiwat et al. 2000]. When the response packets from the sources 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 are also investigating techniques for simultaneously deploying multiple application task graphs, including such issues as algorithms for dynamically detecting such overlap, quantifying the energy saving afforded by merging the overlapping subgraphs, and techniques for merging subgraphs.

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. We further show the scalability of role assignment through extensive simulations.

REFERENCES

ADHIKARI, S., PAUL, A., AND RAMACHANDRAN, U. 2002. D-stampede: Distributed programming system for ubiquitous computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems* (ICDCS, Vienna).

AHAMAD, Y. AND CETINTEMEL, U. 2004. Network-aware query processing for stream-based applications. In *Proceedings of the International Conference on Very Large Databases* (VLDB).

BHARDWAJ, M. AND CHANDRAKASAN, A. 2002. Bounding the lifetime of sensor networks via optimal role assignments. In *Proceedings of IEEE INFOCOM*.

BHATTACHARYA, S., KIM, H., PRABH, S., AND ABDELZAHER, T. 2003. Energy-conserving data placement and asynchronous multicast in wireless sensor networks. In *MobiSys '03: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. ACM Press, New York, NY, 173–185.

BOULIS, A., HAN, C. C., AND SRIVASTAVA, M. B. 2003. Design and implementation of a framework for programmable and efficient sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services* (MobiSys, San Francisco, CA).

CAYIRCI, E., SU, W., AND SANKARASUBRAMANIAN, Y. 2002. Wireless sensor networks: A survey. *Comput. Netw. 38*, 4 (Mar.), 393–422.

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research* (CIDR).

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

CHERNIACK, M., BALAKRISHNAN, H., CARNEY, D., CETINTEMEL, U., XING, Y., AND ZDONIK, S. 2003. Scalable distributed stream processing. In *Proceedings of the Conference on Innovative Database Research* (CIDR).

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractablility: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA.

GU, X. AND NAHRSTEDT, K. 2006. On composing stream applications in peer-to-peer environments. *IEEE Trans. Parall. Distrib. Syst. 17*, 8(Aug.), 824–837.

HEIDEMANN, J. S., SILVA, F., INTANAGONWIWAT, C., GOVINDAN, R., ESTRIN, D., AND GANESAN, D. 2001. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*. 146–159.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. 2000. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. 93–104.

INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the MobiCom '00*. 56–67.

JAE-HWAN, C. AND LEANDROS, T. 2000. Energy conserving routing in wireless ad-hoc networks. In *Proceedings of IEEE INFOCOM*. 22–31.

KUMAR, R., WOLENETZ, M., AGARWALLA, B., SHIN, J., HUTTO, P. W., PAUL, A., AND RAMACHANDRAN, U. 2003. DFuse: Framework for distributed data fusion. In *Proceedings of ACM SenSys 2003*.

LIU, H., ROEDER, T., WALSH, K., BARR, R., AND SIRER, E. G. 2005. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*. ACM Press, New York, NY, 149–162.

MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2002. Tag: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the Conference on Operating System Design and Implementation* (OSDI, Boston, MA).

MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. 2003. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research* (CIDR).

NETPERF. 2003. The public Netperf homepage: `http://www.netperf.org/`.

ORI. 2003. ORiNOCO PC Card (Silver/Gold). For the specification, go online to `http://www.hyperlinktech.com/web/orinoco/-orinoco_pc_card_spec.html`.

PAPADIMITRIOU, C. AND YANNAKAKIS, M. 1988. Towards an architecture-independent analysis of parallel algorithms. In *STOC '88: Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, 510–513.

PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., WELSH, M., SELTZER, M., AND ROUSSOPOULOS, M. 2006. Network-aware operator placement for stream-processing systems. In *Proceedings of the International Conference on Data Engineering* (ICDE).

POLASTRE, J., TOLLE, G., AND HUI, J. 2004. Low power mesh networking with telos and IEEE 802.15.4. In *Proceedings of SenSys*. 319.

RAMACHANDRAN, U., NIKHIL, R. S., HAREL, N., REHG, J. M., AND KNOBE, K. 1999. Space-time memory: A parallel programming abstraction for interactive multimedia applications. In *Proceedings of the ACM Conference on Principles and Practices of Parallel Programming*. 183–192.

REHG, J. M., LOUGHLIN, M., AND WATERS, K. 1997. Vison for a smart kiosk. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*. 690–696.

ROBINS, G. AND ZELIKOVSKY, A. 2000. Improved Steiner tree approximation in graphs. In *SODA '00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 770–779.

SINGH, S., WOO, M., AND RAGHAVENDRA, C. S. 1998. Power-aware routing in mobile ad hoc networks. In *Proceedings of MobiCom '98*. 181–190.

SRIVASTAVA, U., MUNAGALA, K., AND WIDOM, J. 2005. Operator placement for in-network stream query processing. In *Proceedings of the ACM Symposium on Principles of Database Systems* (PODS).

TEWARI, R., DAHLIN, M., VIN, H. M., AND KAY, J. S. 1999. Design considerations for distributed caching on the internet. In *Proceedings of ICDCS*. 273–284.

VIOLA, P. AND JONES, M. 2001. Rapid object detection using a boosted cascade of simple features. In *Proceedings of CVPR*. 511–518.

WOLENETZ, M. 2005. Characterizing middleware mechanisms for future sensor networks. Ph.D. dissertation. College of Computing, Georgia Institute of Technology, Atlanta, GA.

WOLENETZ, M., KUMAR, R., SHIN, J., AND RAMACHANDRAN, U. 2004. Middleware guidelines for future sensor networks. In *Proceedings of the 1st Workshop on Broadband Advanced Sensor Networks*.

WOLENETZ, M., KUMAR, R., SHIN, J., AND RAMACHANDRAN, U. 2005. A simulation-based study of wireless sensor network middleware. *Int. J. Netw. Manage.* (Special Issue on Sensor Networks) *15*, 4 (Jul.), 255–267.

ZAYAS, E. 1987. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 13–24.

Received September 2005; revised March 2006; accepted June 2006