Characterizing Middleware Mechanisms for Future Sensor Networks

A Thesis
Presented to
The Academic Faculty

by

Matthew D. Wolenetz

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August 2005

Characterizing Middleware Mechanisms for Future Sensor Networks

Approved by:

Dr. Umakishore Ramachandran,
College of Computing,
Georgia Institute of Technology,
Adviser

Dr. Karsten Schwan,
College of Computing,
Georgia Institute of Technology

Dr. Ramesh Jain,
School of Electrical and
Computer Engineering,
Georgia Institute of Technology

Dr. Gregory Abowd,
College of Computing,
Georgia Institute of Technology

Dr. Mark Smith,
School of Electrical
and Computer Engineering,
Georgia Institute of Technology,
Visiting Professor from
HP Labs, Palo Alto, CA

Date Approved July 17, 2005

ACKNOWLEDGEMENTS

I am deeply appreciative of the committee members who evaluated this dissertation. They provided essential and timely feedback on this thesis as well as other collaborative efforts. I sincerely thank Karsten, Ramesh, Gregory and Mark for their cooperation.

During my long career at Georgia Tech, I have encountered a long list of dedicated, knowledgeable and compassionate faculty and staff. Gus Baird, Jim Greenlee, Mark Guzdial, Ellen Zegura, H. Venkateswaran and Yannis Smaragdakis deserve special mention for having motivated me to pursue excellence in research and academics. Among the many staff members I have relied on, Barbara Binder, Cathy Dunnahoo, and Dani Denton also deserve special mention for helping me negotiate the graduate student path. I will always remember fondly the dedication and friendliness of the support staff, especially Neil Bright, and research scientists Phil Hutto and Matthew Wolf. I am highly indebted to them for their guidance and friendship. I wish to thank Brian Cooper for his advice on some of the evaluations in this thesis.

A huge part of my success at Georgia Tech has been the undying support of fellow students. Rajnish Kumar, Sameer Adhikari, Bikash Agarwalla, Junsuk Shin, Hasnain Mandviwala, Dave Lillethun, Martin Modahl and Arnab Paul have been both friends and mentors to me, helping me to ask the right questions. We have shared in the fun of building cool systems and writing papers. Among the long list of students I am thankful to have met are Kathy Gray and Namrata Bachwani. I would like to thank them for being outstanding friends as well as classmates.

I would especially like to express my deepest thanks to my wife, Amanda, for sacrificing so much and for always being my great motivator. Without her and our

son, I would be quite lost. I would like to thank my extended family in Atlanta for being such an excellent support system and for continually cheering me onwards.

I am at a loss for the right words to describe my thanks to my advisor, Kishore, for being an absolute dream of a coach, fellow researcher, mentor, teacher and boss. Always looking on the positive side, Kishore is the epitome of every character I would look for in an advisor.

Although it would be fun and suitable to dedicate this thesis to my mother, whose thesis I typed twenty years ago, and whose support has been phenomenal, I must dedicate this thesis to my brother, Michael. It was he who introduced me to programming years ago, and it was he who performed miracles enabling me to attend Georgia Tech. I am a far happier and fulfilled person for his support, and I can never thank him enough.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Due to their promise for supporting applications society cares about and their unique blend of distributed systems and networking issues, wireless sensor networks (SN) have become an active research area. Most current SN use an arrangement of nodes with limited capabilities. Given SN device technology trends, we believe future SN nodes will have the computational capability of today's handhelds, and communication capabilities well beyond today's "motes". Applications will demand these increased capabilities in SN for performing computations in-network on higher bit-rate streaming data. We focus on interesting *fusion applications* such as automated surveillance. These applications combine one or more input streams via synthesis, or *fusion*, operations in a hierarchical fashion to produce high-level inference output streams.

For SN to successfully support fusion applications, they will need to be constructed to achieve application throughput and latency requirements while minimizing energy usage to increase application lifetime. This thesis investigates novel middleware mechanisms for improving application lifetime while achieving required latency and throughput, in the context of a variety of SN topologies and scales, models of potential fusion applications, and device radio and CPU capabilities.

We present a novel architecture, *DFuse*, for supporting data fusion applications in SN. Using a DFuse implementation and a novel simulator, *MSSN*, of the DFuse middleware, we investigate several middleware mechanisms for managing energy in SN. We demonstrate reasonable overhead for our prototype DFuse implementation on a small iPAQ SN. We propose and evaluate extensively an elegant distributed, local role-assignment heuristic that dynamically adapts the mapping of a fusion application

to the SN, guided by a cost function. Using several studies with DFuse and MSSN, we show that this heuristic scales well and enables significant lifetime extension. We propose and evaluate with MSSN a predictive CPU scaling mechanism for dynamically optimizing energy usage by processors performing fusion. The scaling heuristic seeks to make the ratio of processing time to communication time for each synthesis operation conform to an input parameter. We show how tuning this parameter trades latency degradation for improved lifetime. These investigations demonstrate MSSN's utility for exposing tradeoffs fundamental to successful SN construction.

CHAPTER I

INTRODUCTION: AN OVERVIEW

Due to their unique blend of distributed systems and networking issues, wireless sensor networks (SN) have become an active research area. Most current SN use an arrangement of nodes with limited capabilities. Given SN device technology trends, we believe future SN nodes will have the computational capability of today's handhelds, and communication capabilities well beyond today's "motes". Applications will demand these increased capabilities in SN for performing computations in-network on higher bit-rate streaming data.

We focus on future SN applications, such as automated surveillance, that combine one or more input streams via synthesis operations in a hierarchical fashion to produce high-level inference output streams. An example of a synthesis, or *data fusion*, operation is annotating input video frames with detected faces. These operations will execute on processors within the network. Higher level inference streams may be used to guide actuation decisions such as varying camera frame capture rates or triggering alarms. Actuations may impact sensed data in the future, forming feedback loops. Such an application that performs stream-based in-network hierarchical computation is a *fusion application*. Energy will continue to be a primary limiting factor for future SN, so performing in-network fusion in an energy-conscious manner is key to application longevity. There exists a need to study tradeoffs in terms of how much productivity an application can achieve during its lifetime, how application latency and throughput requirements affect both lifetime and productivity, and how various available middleware and device capabilities for performing low-power communication and processing impact these performance metrics. This chapter briefly

introduces this problem and design space, and then outlines the structure of the rest of the thesis.

1.1 Problem Statement

For future SN to successfully support stream-based fusion applications, they will need to be constructed to achieve application throughput and latency requirements while minimizing energy usage to increase application lifetime. This thesis investigates some novel middleware mechanisms for improving application lifetime while achieving required latency and throughput, in the context of a variety of SN topologies and scales, models of potential fusion applications, and device radio and CPU capabilities. Our methodology promotes simulation-based performance evaluation under hypothetical configurations. Tradeoffs exposed by this methodology inform construction of SN in terms of node capabilities and tuning parameters for the studied middleware mechanisms.

1.2 Design Space

There is an ever-evolving continuum of sensing, computing, and communication capabilities from smartdust, to sensors, to mobile devices, to desktops, to clusters. With this evolution, capabilities are moving from larger footprint to smaller footprint devices. For example, tomorrow's *mote* will be comparable in resources to today's mobile devices; and tomorrow's mobile devices will be comparable to current desktops.

1.2.1 Future SN Capabilities and Applications

Given the pace of technology, it is conceivable to imagine SN in the near future wherein each node has 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 1Gbps). Recent advances in low-power microcontrollers, and increased power-conscious radio technologies lend credence to this

belief. For example, next generation iMote prototypes [27] and Telos motes [43] are available for research now. Although not as computationally powerful as a modern iPAQs, iMotes provide 12MHz 32-bit ARM7TDMI processors and 64KB RAM/512KB FLASH, a significant increase in capability compared to Berkeley mote MICA2 [10] predecessors that only had 8MHz 8-bit ATmega128L microcontrollers with 640KB FLASH. Furthermore, the wireless bandwidth available with iMotes is Bluetooth based (over 600Kbps application-level bandwidth), greatly exceeding Berkeley motes' 38.4Kbps data rate. Similarly, Telos motes, designed for long lifetime with very low duty cycles, provide increased computation and communication capabilities over previous generation motes via energy-efficient idle modes and faster, energy-efficient microcontrollers and radios. We believe this trend will continue as SN applications demand ever greater capabilities for performing computation on high bit-rate data within the network. It is conceivable that recent hardware capabilities enabling CPU frequency and voltage scaling for power saving, *e.g.* ARM xScale packages, will be integrated into future SN devices. Already, such technology is integrated into Stargate devices [11], providing higher capability backbones for mote-based SN. 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, currently on the order of over 20 million annually for Nokia alone [55]).

Thus, we envision future SN to consist of deployments of high bandwidth sensor/actuator sources coupled with powerful wireless ambient processing hardware. Most current SN assume a homogeneous and dedicated arrangement of nodes with limited capabilities (such as Berkeley motes [43, 27, 10]). Such networks have been successfully deployed for many low bit-rate applications, for example seabird habitat monitoring [34] and grape plant monitoring in vineyards [23]. In contrast, future SN will enable a whole host of high bit-rate, computationally intensive applications

such as distributed surveillance, emergency response, and homeland security. Many of these *fusion applications* share a common requirement, namely, hierarchical *data fusion*, i.e., applying a synthesis operation on input streams. The main characteristic of such applications is a sense-process-actuate *control loop* enabled by in-network processing of streaming data. Latency from sensing to actuation, and throughput are the two obvious figures of merit for such applications. In addition, an important figure of merit for such applications is network *lifetime*.

By definition, SN operate on battery power with minimal supervision. Energy is *the* most critical resource in wireless sensor networks, and it is even more critical when we target high bit-rate fusion applications. Therefore, SN applications have a limited operational time before the network becomes partitioned due to energy consumption. There exist tradeoffs in terms of how much productivity an application can achieve during this lifetime, how application latency and throughput requirements affect both lifetime and productivity, and how well various available device capabilities for performing low-power communication and processing can be leveraged to improve performance. We contend that although communication of one bit may cost 3 orders of magnitude higher than processing one instruction [56], fusion applications will routinely require large amounts of processing occurring in-network on sensor nodes. Therefore, processing cost must be accounted for when managing energy. Similarly, large memory footprints may incur significant cost.

1.2.2 Application Domain

As a concrete motivating example, consider a campus-wide automated surveillance application to provide safety for people and resources on campus. The deployed infrastructure consists of a variety of sensors such as cameras and microphones scattered throughout campus. Nodes of the wireless SN are similarly scattered across the campus to provide redundant connectivity and in-network processing resources.

Actuator nodes may be PDAs carried by security officers, or other SN resources such as pan-tilt-zoom motors attached to cameras. As data from sensors pass through the network, nodes perform application-specific *fusion functions* (such as face detection, image correlation, and higher level inferencing).

Fusion behavior and consequent resource requirements will be dynamic, based on responding to changing inputs from the environment. For example, there may be periods of relatively low processing as the system cheaply scans infrequently captured images for features of interest, such as the sound of breaking glass or movement in a restricted area. Once such a feature is detected, the surveillance application operates in a period of intense activity performing extensive processing, such as face recognition, on images captured as frequently as possible.

This specific application is an instance of the general *control loop* described earlier, where both automated and "human-in-the-loop" actuation decisions result from in-network communication and computation. Energy will continue to be a primary limiting factor for such a deployment, so performing in-network fusion in an energy-conscious manner is key to application longevity. Other fusion application examples include streaming media, image-based tracking, interactive vision, and feature extraction for continuous queries used by applications such as EventWeb [36]. These applications share a common requirement of applying synthesis operations (fusion functions) upon multiple input streams in hierarchical manner. Fusion functions can be used for efficiency (e.g. compressing an input stream), or can be part of the application behavior (e.g. feature extraction from an image).

Fusion applications are typically described as a task graph, where nodes in the graph are of three types: data *source* (data producer node), *sink* (a node where a user presents requests), and *fusion* (a node which applies a fusion function). This graph is deployed as an overlay network using *relay* nodes to interconnect indirectly reachable nodes. Relay nodes act as simple data forwarders. When bound to a network node,

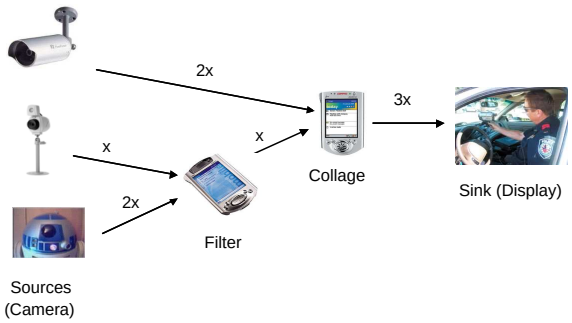

Figure 1: An example surveillance application that uses in-network distributed data fusion. Edge labels indicate relative (expected) transmission rates of data sources and fusion points.

a task graph data fusion node becomes a fusion point.

Figure 1 shows a tiny example task graph of a surveillance application. The filter function selects images with some interesting properties (*e.g.* rapidly changing scene), and sends the compressed image data to the collage function. The collage function decompresses the images coming from possibly different locations, combines the images and sends the composite image to the root (sink) for further processing. We will return to both this tiny task graph and the hypothetical campus surveillance application in more detail later in this thesis.

To support fusion applications, we need specific systems facilities: support for applying synthesis operations at fusion points, support for migration of fusion points from one dying or non-optimal network node to a more suitable node, and support to handle time-stamped data items produced from the data sources. Other middleware requirements include memory and buffer management, programming support, etc.

1.2.3 Network Layers

We focus primarily on the performance of high-level middleware mechanisms such as cost-directed fusion processing in the context of fairly ideal assumptions about the underlying network layers. We assume that any SN node is initially reachable from

any other node, and we assume a routing layer that exposes hop-count information between any two nodes in the network. As energy is drained on nodes due to computation, communication and idling overheads, nodes may "die", eventually causing network partition. Typically, these assumptions can be satisfied by a separate layer that supports a routing protocol for ad hoc networks, like Dynamic Source Routing (DSR) [25], and exposes an interface to query the routing information.

Additional overheads in terms of energy and time used for maintaining routing information and for recovering from lossy radio propagation and unanticipated node failures are assumed to be ideal (negligible) in the context of our studies. While these overheads can significantly degrade SN application performance in general, we focus on evaluation of our novel fusion architecture's mechanisms through a prototype implementation and usage of our novel detailed simulator with these ideal routing, radio propagation and node reliability assumptions. The primary performance management mechanisms we propose and evaluate (cost-aware fusion point placement and predictive CPU scaling) are both meant to increase network lifetime. By using these mechanisms in an informed manner, SN can be constructed to achieve application latency and throughput requirements.

We anticipate that further work outside the scope of this thesis using a combination of our implementation or simulator with more realistic network layers would further confirm the utility of our primary performance management mechanisms for extending SN lifetime and would provide more accurate details of application latency and throughput performance. Recent work [13] suggests that more complex interactions and tradeoffs emerge as less-than-ideal network assumptions are employed in SN studies, motivating usage of actual large scale deployments. Such deployments are out of scope of this thesis, as the SN scales we consider in some experiments exceed hundreds of nodes.

1.2.4 Devices Considered

Where we include device-level bandwidth and resource consumption in our exploration, we use models based on Orinoco 802.11b and Bluetooth ∼721Kbps radio specifications. Our simulator evaluation platform is extendable to address other radio models for use in studies beyond the scope of this thesis. We use these two models as representative, contrasting future SN radio models (802.11b has high bandwidth and cost, while Bluetooth has lower bandwidth and cost). As we cannot predict actual future radio devices exactly, we rely on studying these representative extremes.

Similarly, we have limited the scope of our exploration of CPU capabilities to a linear model of CPU speed and consequent power consumption, based on published experiments of SA-1100 and SA-110 processor power consumption at various frequencies and voltages. We also employ a power model based on Intel's PXA270 xScale datasheet [22] as an alternative CPU model in our predictive CPU scaling mechanism's evaluation.

To limit the scope of the problem space, we have used a simple power model for memory in a SN node. Our simulation results indicate that even our pessimistic, costly memory power model consumes insignificant energy relative to communication and processing for future SN application workloads, so we do not consider a greater variety of memory models in our exploration. Specifics of all of these models are presented later in this thesis.

1.3 Related Work

It is well-recognized that energy is critical in SN, driving a significant amount of recent research into mechanisms for SN energy optimization. Most current SN research focuses on contemporary devices and device models for low-bit rate communication and minimal in-network computation, rather than on mechanisms for supporting high-bit rate communication with significant in-network computation. Approaches for SN

energy optimization range from hardware [43, 27], MAC [58, 51], routing [52, 8], cross-layer approaches [28], and application-specific optimizations such as energy-efficient target tracking [17]. Additionally, there have been middleware approaches to bridge the gap between application and lower layers [19, 31].

Recent research in power-aware routing for mobile ad hoc networks [52, 8] proposes power-aware metrics for determining routes in wireless ad hoc networks. We use similar metrics to formulate different cost functions for guiding our fusion point placement mechanism. While designing a power-aware routing protocol is not the focus of this thesis, routing protocol information may be usable in future work for defining more flexible cost functions or for informing our predictive CPU scaling mechanism.

Similarly, this thesis does not propose a cross-layer algorithm for SN energy optimization, although recent analytical work [28] in this area may assist with characterizing performance bounds. In this particular approach, the low-level scheduling and power control problem that optimizes energy usage for application QoS is shown to be NP-Complete, and the proposed algorithm is centralized, limiting its applicability in distributed SN environments. However, the observation of the intractability of optimal scheduling further motivates our proposed distributed heuristics.

Research into application-specific SN energy optimizations propose evaluation metrics suitable to the applications being studied. An example metric is *QoSv* [17], or "quality of surveillance", determined by how far a target moves before the sensor network detects it. Our research focuses on mechanisms to support more general streaming fusion applications, so we choose application figures of merit applicable and important to these applications, including latency, throughput and lifetime.

Our approach focuses on middleware techniques for SN energy optimization, to bridge the gap between stream-based application requirements and low-level device and network layer capabilities. MiLAN [19] has the most similar goals to our DFuse [31] work presented in Chapter 2, providing a set of middleware mechanisms

for adapting the SN to effect application supplied performance policy. Our example campus surveillance SN fusion application could be accommodated to some degree by MiLAN. However, that middleware does not provide the combination of general streaming data abstractions for in-network computation along with approaches for optimizing the energy usage given application latency and throughput requirements.

Beyond our architecture's initial prototype implementation and evaluation, we have built a simulation-based evaluation framework for our middleware. Prowler [50], TOSSIM [32], and Em∗ [16] simulators and emulator are specialized towards Berkeley mote sensors and communication channels. Our study focuses first on modeling energy usage and performance of a variety of middleware mechanisms for a whole range of futuristic sensor node architectures, requiring a fairly detailed implementation of the middleware inside the simulator and a decoupling from a specific target device.

1.4 Broader Application

Our work is focused on future SN. However, it may be possible to adapt our mechanisms to target lower-bandwidth, lightweight computation capabilities of today's motes. Also, our research may well be applicable outside of SN. Contemporary laptops and handhelds are immediate sibling platforms for applications and supporting middleware mechanisms we study. General application-directed migration of computation may apply in grid computing and distributed media processing, to better achieve latency and throughput requirements, regardless of energy consumption. Furthermore, focused contributions, such as our predictive CPU scaling heuristic, may well apply to more general distributed streaming contexts outside of SN.

1.5 Open Questions

As the design space for future SN devices, applications, and middleware for optimizing energy (lifetime) while meeting application latency and throughput requirements is

vast, we are aware of several open research questions outside the scope of our work:

1. We are not concerned with mechanisms for dynamically adapting the bandwidth, range and signal strength of SN radios, although this route of research may provide additional benefits to applications in terms of latency, throughput and lifetime. It should be possible for later work to reuse our middleware simulator to characterize the potential benefits of such mechanisms, coupled with appropriate models of radio, MAC and routing layers. There is currently much conflicting research on whether multi-hop communication saves energy vs "shouting louder", and varying application domains may have different trends here.

2. Relaxing the ideal MAC and routing layer assumptions in our simulation-based evaluation of middleware mechanisms is future work outside the scope of this thesis. One future approach might be to couple our middleware simulator with an existing wireless network layer simulator. Of the currently available simulator options, GloMoSim [3] appears to be better than ns2-wireless [9], as GloMoSim provides practical support for larger scale wireless deployments than ns2-wireless, critical to successful evaluation of our middleware model. However, significant "wrapping" of GloMoSim's network layer API and lengthy instrumentation of its physical layer to provide runtime feedback to our simulator's power and routing models would be necessary. Furthermore, studies of SN with heterogeneous radio transmission ranges would be difficult to perform with GloMoSim's current restriction to a homogeneous SN.

3. We constrain our study to supporting a single fusion application with a static task graph (in terms of data flow dependencies). In this work, we do not consider relaxations of this assumption including providing support for multiple applications and for applications whose task graphs are dynamic. While our

mechanisms rely on virtualization of local device resources to manage timesharing required when multiple task graph fusion points are mapped to the same device, virtualization support for multiple applications is not our focus.

4. We do not propose new routing layers for power-aware, or more correctly, application-performance aware placement of relay nodes used to connect our overlay network.

5. We do not consider security or privacy issues in our studies presented here.

6. There is a need for coordinated control in SN. Our application models and middleware implementations and models do not focus on control. Rather, they are concerned with keeping up with demand by downstream consumers. For stream based fusion applications we consider, coordinated data streaming from multiple sources is a needed contribution.

7. The design space greatly expands when migration of sources and sinks, and general mobility of SN nodes is introduced. There are opportunities for leveraging such mobility for energy savings through radio power scaling and message ferrying, recharging batteries, and for improving application throughput and latency by dynamically positioning resources more optimally. We do not consider mobility-based approaches for optimization in this work.

8. We do not consider device failures other than for reasons of lack of energy. One potential incremental approach for addressing device failure is to create redundant fusion points in the network, creating an energy vs availability tradeoff. Other approaches in SN domain [5] have considered a similar tradeoff: energy vs accuracy.

1.6 Contributions and Research Outline

To address our problem statement in our design space, this thesis presents the following contributions:

1. *DFuse*, our novel middleware for performing energy aware stream processing in SN is presented in Chapter 2, including its architecture, prototype implementation and evaluation on a 12 node iPAQ SN. We demonstrate reasonable overhead of our implementation through microbenchmarks. We use our implementation to evaluate application performance in terms of network lifetime, number of migrations and residual battery level variance for each of several cost functions used to guide the DFuse dynamic fusion point migration mechanism. Results show that cost function directed migration can significantly extend application lifetime.

2. *MSSN*, our novel simulator of our DFuse middleware, enabling evaluation of SN performance in the context of a variety of potential SN devices, topology scales, middleware capabilities and application workloads is presented in Chapter 3. This simulator is a major contribution of this thesis, enabling evaluation of SN exemplified in three case studies also contributed here. MSSN's non-trivial design and implementation realizes a scalable, believable middleware simulator that enables such low level SN node attributes as CPU speed scaling to be evaluated in the context of application level performance.

3. We use microbenchmarks to quantify fusion function processing and I/O footprints for our campus surveillance application, generating two workload extremes: CPU-intensive and communication-intensive. In the first case study in Chapter 4, we use MSSN to quantify these workloads' performance under differing middleware and SN device configurations to show how MSSN can be used to expose performance tradeoffs in future SN.

4. For our second MSSN case study in Chapter 4, we present scalability analyses of the DFuse fusion point placement mechanism. We show that our distributed *role-assignment* heuristic performs well with respect to our best feasibly calculated optimal performance as application and network topology scales increase. We also demonstrate that MSSN confirms the observed performance of our actual DFuse implementation's role-assignment heuristic at small scales.

5. Our final MSSN case study in Chapter 4 presents a design, implementation and evaluation using MSSN of a novel, tunable CPU-scaling heuristic for further improving SN performance. Using a simple surveillance application model that varies data rate and processing intensity over time, we demonstrate how the CPU-scaling mechanism varies the consequent CPU speed required over time. We quantify this application's performance, and demonstrate how tuning the CPU-scaling heuristic within application tolerances can improve performance.

This thesis concludes with an overview of contributions and lessons learned in Chapter 5, and directions for future work in Chapter 6.

CHAPTER II

DFUSE: AN ARCHITECTURE FOR DISTRIBUTED DATA FUSION IN SENSOR NETWORKS

2.1 Introduction

This chapter focuses on finding a middleware solution to challenges involved in supporting fusion applications in future SN. Developing fusion applications is challenging in general because of the time-sensitive nature of the fusion operation, and the need for synchronization of the data from multiple streams. Since the applications are inherently distributed, they are typically implemented via distributed threads that perform fusion in a hierarchical manner. Thus, the application programmer has to deal with thread management, data synchronization, buffer handling, and exceptions (such as time-outs while waiting for input data for a fusion function) - all in a distributed fashion. SN add another level of complexity to such application development due to the scarcity of power in the individual nodes [7]. In-network aggregation [33, 21, 18] and power-aware routing [52, 8] are techniques to alleviate power scarcity in SN. While the good news about fusion applications is that they inherently need in-network aggregation, a naive placement of the fusion points in the network 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 points in the network 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 [59].

We have developed *DFuse*, an extendable 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 earlier 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 our DFuse architecture 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 [33, 21, 18] 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* (according to some predefined cost function) mapping of fusion functions to the network nodes.

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

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

In Chapter 4, we demonstrate how DFuse can be extended with an additional predictive CPU scaling mechanism to further adapt the SN for improved performance.

The rest of this chapter is structured as follows. Section 2.2 analyzes fusion application requirements and presents the DFuse architecture. In Section 2.3, we describe how DFuse supports distributed data fusion. Section 2.4 explains a heuristic-based distributed algorithm for placing fusion points in the network. This is followed by implementation details of the framework in Section 2.5 and its evaluation in Section 2.6. We then compare our framework with other existing and ongoing efforts in Section 2.7, and conclude our DFuse framework contribution in Section 2.8.

2.2 DFuse Architecture

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

2.2.1 Target Applications and Execution Environment

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

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

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

DFuse assumes that addresses for data sources and sinks in the input task graph are known beforehand. For data-centric queries, e.g. "show a collage of images from the north region", the source addresses are not known at query time. The addresses of such data sources can be obtained by employing date-centric techniques, e.g. sink floods the network with the query, and data sources report their addresses to help

sink node build the task graph. The problem of representing a given data-centric query as a task graph is out of scope of this thesis.

DFuse is intended for deployment in a heterogeneous ad hoc sensor network environment. However, DFuse cannot be deployed in current sensor networks given the limited capabilities available in sensor node prototypes such as Berkeley motes [20]. But, as we add devices with more capabilities to the sensor network, or improve the sensor nodes themselves, more demanding applications can be mapped onto such networks and DFuse provides a flexible fusion API for such a deployment. As will become clear in later sections, DFuse handles the dynamic nature of such networks by employing a resource-aware heuristic for placing the fusion points in the network.

DFuse assumes that any node in the network is reachable from any other node. Further, DFuse assumes a routing layer that exposes hop-count information between any two nodes in the network. Typically, such support can be provided by a separate layer that supports a routing protocol for ad hoc networks, like Dynamic Source Routing (DSR) [25], and exposes an interface to query the routing information.

2.2.2 Architecture Components

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

Figure 2(B) shows the internal structure of the fusion module. Details of the fusion module are discussed in section 2.3. The modules that implement resource monitoring and routing are external to the DFuse architecture. These modules help

in the evaluation of cost functions that is used by the placement module in determining a *good* placement of fusion functions.

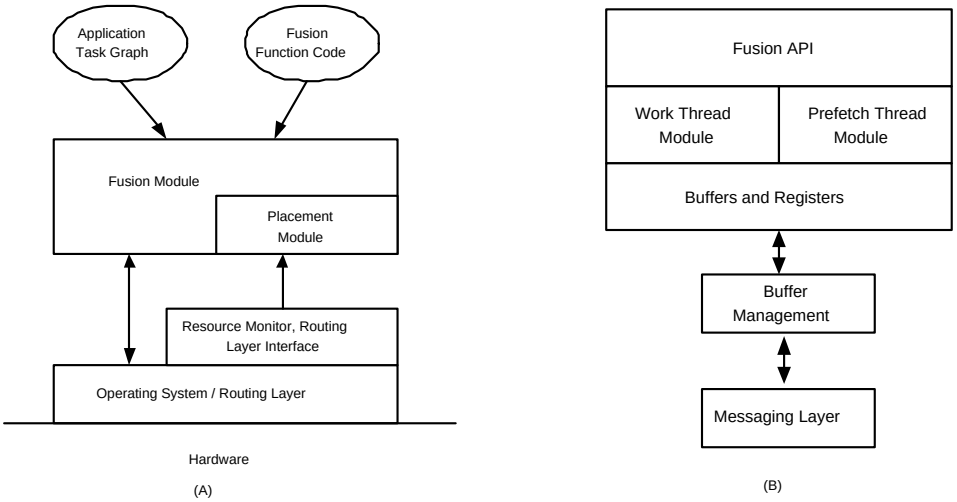

Figure 2: (A) DFuse architecture - a high-level view per node. (B) Fusion module components.

2.2.3 Launching an Application and Network Deployment

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

Launching an application is accomplished by presenting the task graph and the fusion codes to DFuse at some designated node, let us call it the *root* node. Upon getting this launch request, the placement module of DFuse at the root node starts a distributed algorithm for determining the best placement (details to be presented in Section 2.4) of the fusion functions. The algorithm maps the fusion functions of the task graph onto the physical network subject to some cost function. In this resulting overlay network, each node knows the fusion function (if any) it has to run as well as the sources and sinks that are connected to it. The resulting overlay network is

a directed graph with source, fusion, and sink nodes (there could be cycles since the application may have feedback control). The application starts up with the sink nodes running their respective codes, resulting in the transitive launching of the codes in the intermediate fusion nodes and eventually the source nodes. Cycles in the overlay network are handled by each node remembering if a launch request has already been sent to the nodes that it is connected to.

The role of each node in the network can change over time due to both the application dynamics as well as health of the nodes. The placement module at each node performs periodic re-evaluation of its health and those of its neighbors to determine if there is a better choice of placement of the fusion functions. The placement module requests the fusion module to affect any needed relocation of fusion functions in the network. Details of the placement module are forthcoming in Section 2.4.

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

2.3 Distributed Data Fusion Support

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

Data fusion, broadly defined, is the application of an arbitrary transformation to a correlated set of inputs, producing a "fused" output item. In streaming environments, this is a continuous process, producing an output stream of fused items. As mentioned previously, such transformations can result in the expansion, contraction, or *status quo* in the data flow rate after the fusion. Note that a filter function, taking a

single input stream and producing a single output stream, is a special case of such a transformation. We assume that fusion outputs can be shared by multiple consumers, allowing "fan-out" from a fusion point, but we disallow a fusion point with two or more distinct output streams. Fusion points with distinct output streams can be easily modelled as two separate fusion points with the same inputs, each producing a single output. Note that the input of a fusion point may be the output of another fusion point, creating fusion pipelines or trees. Fusion computations that implement control loops with feedback create cyclic fusion graphs.

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

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

2.3.1 Structure management

This category of capabilities primarily handles "plumbing" issues. The fundamental abstraction in DFuse that encapsulates the fusion function is called a *fusion channel*. A fusion channel is a named, global entity that abstracts a set of inputs and encapsulates a programmer-supplied fusion function. Inputs to a fusion channel may come from the node that hosts the channel or from a remote node. Item fusion is automatic and is performed according to a programmer-specified policy either on request (demand-driven, lazy, pull model) or when input data is available (data-driven, eager,

push model). Items are fused and accessed by timestamp (usually the capture time of the incoming data items). An application can request an item with a particular timestamp or by supplying some wildcard specifiers supported by the API (such as *earliest item*, *latest item*). Requests can be blocking or non-blocking. To accommodate failure and late arriving data, requests can include a minimum number of inputs required and a timeout interval. Fusion channels have a fixed capacity specified at creation time. Finally, inputs to a fusion channel can themselves be fusion channels, creating fusion networks or pipelines.

2.3.2 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.

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

2.3.4 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.

2.3.5 Failure/latency handling

This category of capabilities primarily allows the fusion points to perform partial fusion, i.e. fusion over an incomplete input correlation set. It deals with sensor failure and communication latency that are common, and often indistinguishable, in sensor networks. Fusion functions capable of accepting a variable number of input items may specify a timeout on the interval for correlation set collection. Late arriving items may be automatically discarded or included in subsequent correlation sets. If

the correlation set contains fewer items than needed by the fusion function, an error event occurs and a programmer-supplied error handler is activated. Error handlers and fusion functions may produce special *error items* as output to notify downstream consumers of errors. Fused items include meta-data indicating the inputs used to generate an item in the case of partial fusion. Applications may use the structure management API functions to remove the faulty input if necessary.

2.3.6 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[1]." Intermediate fusion points aggregate and report the status of their inputs along with the status of the fusion point itself via their respective status registers. Fusion points may poll this register or access its status. Similarly, sensors that support a command set (to alter sensor parameters or explicitly activate and deactivate) should be controllable via a "command" register. The specific command set is, of course, device specific but the general device driver analogy seems well-suited to control of sensor networks.

2.3.7 Fusion API Summary

We summarize the primary calls in our fusion channel interface in Figure 3. Functions are presented in a very abstract form to elide language and platform implementation details.

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

- **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?)
```

Figure 3: Fusion channel API summary

2.4 Fusion Point Placement

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

A network node can play one of three roles: *end point (source or sink)*, *relay*, or *fusion point* [4]. An end point corresponds to a data source or a sink. The network nodes that correspond to end points and fusion points may not always be directly reachable from one another. In this case, data forwarding relay nodes may be used to route messages among them. The routing layer (Figure 2) is responsible for assigning a relay role to any network node. The role assignment algorithm assigns only the fusion point roles.

2.4.1 Placement Requirements in SN

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

2.4.1.1 Node Heterogeneity

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

2.4.1.2 Power Constraint

A role assignment algorithm should minimize data communication since data transmission and reception expend more power than computation activities in wireless sensor networks [20]. Intuitively, since the overall communication cost is impacted by the location of data aggregators, the role assignment algorithm should seek to find a suitable placement for the fusion points that minimizes data communication.

2.4.1.3 Dynamic Behavior

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

2.4.2 The Role Assignment Heuristic

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

2.4.2.1 Initialization Phase

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

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

2.4.2.2 Optimization Phase

After completion of the naive tree building phase, the root node informs all other nodes in the network about the start of the optimization phase. During this phase, every node hosting a fusion point role is responsible for either continuing to play that role or transferring the role to one of its neighbors. The decision for role transfer is taken solely by the fusion node based upon local information. A fusion node periodically informs its neighbors about its role and its *health* – an indicator of how good the node is in hosting that role. Upon receiving such a message, a neighboring node computes its own health for hosting that role. If the receiving node determines that it can play the role better than the sender, then it informs the sender (fusion node) of its own health and its intent for hosting that role. If the original sender receives one or more intention requests from its neighbors, the role is transferred to the neighbor with the best health. Thus, with every role transfer, the overall health of the overlay network improves. Application data transfer starts only after the optimization phase to avoid possible energy wastage in an unoptimized network.

Once the application is running, DFuse uses a third *maintenance phase* that works similar to the optimization phase (same role transfer semantics). Details are presented in Section 2.5.

2.4.3 Sample Cost Functions

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

We describe four sample cost functions below. They are motivated by recent works on power-aware routing in mobile ad hoc networks [52, 8]. The *health* of a node k to run fusion role f is expressed as the cost function $c(k, f)$. A fusion node compares its own health with the reported health of its neighbors, and it does the role transfer if

there is an expected health improvement that is beyond a threshold. Note that the lower the cost function value, the better the node health.

2.4.3.1 Minimize transmission cost - 1 (MT1)

This cost function aims to decrease the amount of data transmission required for running a fusion function. Input data needs to be transmitted from sources to the fusion point, and the output data needs to be propagated to the consumer nodes (possibly across hops). For a fusion function f with m input data sources (fan-in) and n output data consumers (fan-out), the transmission cost for placing f on node k is formulated as shown in Figure 4. Here, *t(x)* represents the transmission rate of the data source x, and $hopCount(i, k)$ is the distance (in number of hops) between node i and k.

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

Figure 4: Minimize Transmission Cost - 1 (MT1)

2.4.3.2 Minimize power variance (MPV)

This cost function tries to keep the power of network nodes at similar levels. If $power(k)$ is the remaining power at node k, the cost of placing *any* fusion function on that node is as shown in Figure 5.

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

Figure 5: Minimize Power Variance (MPV)

2.4.3.3 Minimize ratio of transmission cost to power (MTP)

This cost function aims to decrease both the transmission cost and lower the difference in the power levels of the nodes. The intuition here is that the cost reflects how long a node can run the fusion function. The cost of placing a fusion function f on node k can be formulated as shown in Figure 6.

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

Figure 6: Minimize Ratio of Transmission Cost to Power (MTP)

2.4.3.4 Minimize transmission cost - 2 (MT2)

This cost function is similar to **MT1**, except that now the cost function behaves like a step function based upon the node's power level. For a powered node, the cost is same as $c_{MT1}(k, f)$, but if the node's power level goes below a threshold, then its cost for hosting any fusion function becomes infinity. Thus, if a fusion point's power level goes down, a role transfer will happen even if the transfer deteriorates the transmission cost. The cost function can be represented as shown in Figure 7.

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

Figure 7: Minimize Transmission Cost - 2 (MT2)

2.4.4 Heuristic Analysis

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

The heuristic could occasionally result in the role assignment getting caught in a suboptimal mapping. However, due to the dynamic nature of SN and the re-evaluation of the health of the nodes at regular intervals, such occurrences will be short lived. For example, if 'minimize transmission cost (MT2)' is chosen as the cost function, and if the network is caught in a suboptimal mapping, that would imply that some node is losing energy faster than an optimal node. Thus, one or more of the suboptimal nodes will die causing the algorithm to adapt the assignment. Note that if 'minimize transmission cost (MT1)' were chosen in this case, the assignment would *only* be adapted if the new assignment has a lower transmission cost, regardless of node energy possibly being below a threshold endangering network partition and application lifetime termination. This behavior is observed in real life as well and we show it in the evaluation section. Later in Chapter 4, we show further evaluation of how close our role assignment heuristic approaches optimal at large application and topology scales.

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

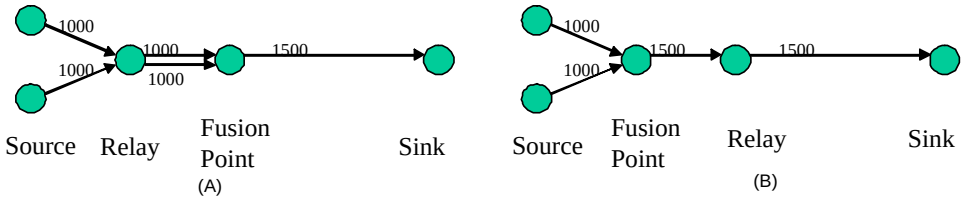

Figure 8: Linear Optimization Example

The heuristic leads mainly to two types of role transfers:

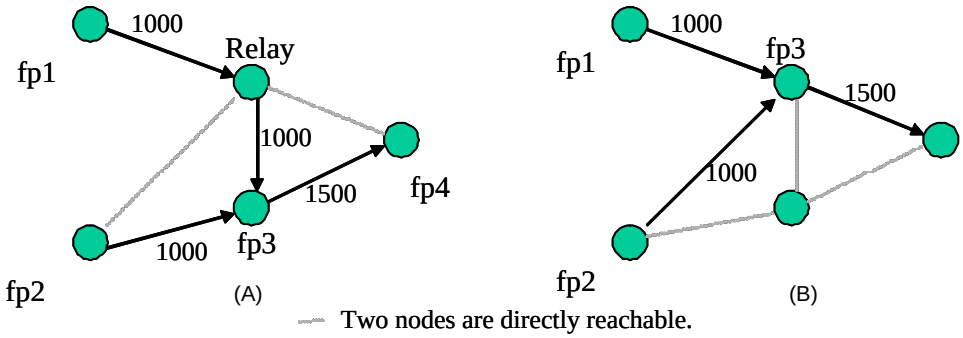

Figure 9: Triangular Optimization Example

2.4.4.1 Linear Optimization

If all the inputs to a fusion node are coming via a relay node (Figure 8A), and there is data contraction at the fusion point, then the relay node will become the new fusion node, and the old fusion node will transfer its responsibility to the new one (Figure 8B.) In this case, the fusion point is moving away from the sink, and coming closer to the data source points. Similarly, if the output of the fusion node is going to a relay node, and there is data expansion, then the relay node will act as the new fusion node. In this case, the fusion point is coming closer to the sink and moving away from the data sources.

2.4.4.2 Triangular Optimization

If there are multiple paths for inputs to reach a fusion point (Figure 9A), and if there is data contraction at the fusion node, then a triangular optimization can be effected (Figure 9B) to bring the fusion point closer to the data source points. The fusion point will move along the input path that maximizes the savings. In the event of data expansion at the fusion point, the next downstream node from the fusion point in the path towards the sinks will become the new fusion node. The original fusion point will simply act as a relay node.

2.5 Implementation

DFuse is implemented as a multi-threaded runtime system, assuming infrastructure support for timestamping data produced from different sensors, and a reliable transport layer for moving data through the network. Multi-threading the runtime system enhances opportunities for parallelism in data collection and fusion function execution for streaming tasks. The infrastructural assumptions can be satisfied in various ways. As we mentioned in Section 2.3, the timestamps associated with the data can be virtual or real. Virtual timestamping has several advantages, the most important of which is the fact that the timestamp can serve as a vehicle for propagating the causality between raw and processed data from a given sensor. Besides, virtual timestamps allows an application to choose the granularity of real-time interval for chunking streaming data. Further, the runtime overhead is minimized since there is no requirement for global clock synchronization, making virtual time synchrony attractive for SN. For transport, given the multi-hop network topology of SN, a messaging layer that supports ad hoc routing is desirable.

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

1. Implementing a multi-threaded architecture for the fusion module that supports the basic fusion API calls (Section 2.3), and the other associated optimizations such as prefetching;

2. Implementing the placement module that supports the role assignment tasks (Section 2.4); and

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

The infrastructural requirements are met by a programming system called Stampede [46, 1]. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through *channels*. Stampede also provides *registers* with *full/empty* synchronization semantics for inter-thread signaling and event notification. The threads, channels, and registers can be launched anywhere in the distributed system, and the runtime system takes care of automatically garbage collecting the space associated with obsolete items from the channels. Though Stampede's messaging layer does not support adaptive multi-hop ad hoc routing, we adopt a novel way of performing the evaluation with limited routing support (Section 2.6). For the ease of evaluation, we have decoupled the fusion and placement module implementations. Their interface is a built-in communication channel and a protocol that facilitates dynamic task graph instantiation and adaptation using the DFuse API. Transmission rates exhibited by the application are collected by this interface and communicated to the placement module.

2.5.1 Data Fusion Module

We have implemented the fusion architecture in C as a layer on top of the Stampede runtime system. All the buffers (input buffers, fusion buffer, and prefetch buffer) are implemented as Stampede channels. Since Stampede channels hold timestamped items, it is a straightforward mapping of the fusion attribute to the timestamp associated with a channel item. The Status and Command registers of the fusion architecture are implemented using the Stampede register abstraction. In addition to these Stampede channels and registers that have a direct relationship to the elements of the fusion architecture, the implementation uses additional Stampede channels and threads. For instance, there are prefetch threads that gather items from the input buffers, fuse them, and place them in the prefetch buffer for potential future requests. This feature allows latency hiding but comes at the cost of potentially wasted network

bandwidth and hence energy (if the fused item is never used). Although this feature can be turned off, we leave it on in our evaluation and ensure that no such wasteful communication occurs. Similarly, there is a Stampede channel that stores requests that are currently being processed by the fusion architecture to eliminate duplication of work.

The `createFC` call from an application thread results in the creation of all the above Stampede abstractions in the address space where the creating thread resides. An application can create any number of fusion channels (modulo system limits) in any of the nodes of the distributed system. An `attachFC` call from an application thread results in the application thread being connected to the specified fusion channel for getting fused data items. For efficient implementation of the `getFCItem` call, a pool of worker threads is created in each node of the distributed system at application startup. These worker threads are used to satisfy `getFCItem` requests for fusion channels created at this node. Since data may have to be fetched from a number of input buffers to satisfy the `getFCItem` request, one worker thread is assigned to each input buffer to increase the parallelism for fetching the data items. Once fetching is complete, the worker thread rejoins the pool of free threads. The worker thread to fetch the last of the requisite input items invokes the fusion function and puts the resulting fused item in the fusion buffer. This implementation is performance-conscious in two ways: first, there is no duplication of fusion work for the same fused item from multiple requesters; second, fusion work itself is parallelized at each node through the worker threads.

The duration to wait on an input buffer for a data item to be available is specified via a policy flag to the `getFCItem`. For example, if *try_for_time_delta* policy is specified, then the worker thread will wait for time *delta* on the input buffer. On the other hand, if *block* policy is specified, the worker thread will wait on the input buffer until the data item is available. The implementation also supports partial fusion in case

some of the worker threads return with an error code during fetch of an item. Taking care of failures through partial fusion is a very crucial component of the module since failures and delays can be common in SN.

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).

2.5.2 Placement Module

The placement module implementation is an event-based simulation of the distributed heuristic for assigning roles (Section 2.4) in the network. It takes an application task graph and the network topology information as inputs, and generates an overlay network, wherein each node in the overlay is assigned a unique role of performing a fusion operation. It currently assumes an ideal routing layer (every node knows a shortest-hop route to every other node) and an ideal MAC layer (no contention). It should be noted that any behavior different from this ideal one can be encoded, if necessary, in an appropriate cost function. Similarly, any enhancement in the infrastructure capabilities such as multicast can also be captured in an appropriate cost function.

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

1. Initialization phase: This phase starts with registering the "build naive tree" event at the root node with the application task graph as the input. If the task graph does not have any rate information (expected data rates of sources and fusion functions) available, root node invokes the bottom-up naive assignment

algorithm, else it decided between bottom-up and top-down assignment based upon the number of data contracting / expanding fusion functions. The task graph is simply mapped to the network starting from the root node or the data sources as described in Section 2.4, disregarding cost function evaluation. At the end of this phase, there will be a valid, though naive, role assignment. Every node will know its role, if any, and it will know the addresses of the producer and consumer nodes corresponding to its role.

2. Optimization phase: In this phase, the heuristic runs with a cost-function based upon the hop-count information and fusion function characteristic (data expansion or contraction) at the fusion points. The application has not yet been launched. Therefore the nodes do not have actual data transmission rates to incorporate into the cost function. The nodes exchange the hop-count and fusion characteristics information frequently to speed up the optimization, and lead to an initial assignment of roles prior to launching the application.

3. Maintenance phase: This phase behaves similarly to the optimization phase, with the difference that the application is actually running. Therefore, the cost function will use real transfer rates from the different nodes in possibly changing the role assignments. In principle, we could have moved directly from the first phase to this phase. The reason for the second (optimization) phase prior to application startup is to avoid excessive energy drain with actual application data transmissions before the network stabilizes to an initial good mapping. The frequency of role transfer request broadcasts in the third phase is a tunable parameter.

During the optimization phase, the cost function uses the fusion function characteristics such as data expansion or contraction. If such information is not available for a role, then data contraction is assumed by the placement module. If there are

multiple consumers for data produced at some fusion point, then it is tricky to judge if there is an effective data expansion or contraction at such nodes. Even if the fusion characteristic indicates that there is data contraction, if the same data is to be transmitted to more than one consumer, effectively there may be data expansion. Also, if two or more consumers are within a broadcast region of the fusion point, then a single transmission may suffice to transport the data to all the consumers, and this will lessen the effect of the data expansion. However, these effects are accountable in the cost function.

2.6 Evaluation

We have performed an evaluation of the fusion and placement modules of the DFuse architecture at two different levels: micro-benchmarks to quantify the overhead of the primitive operations of the fusion API including channel creation, attachments/detachments, migration, and I/O; 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 2.5.1. We first describe the micro-benchmarks and then the results with the placement module, both for a small network of twelve iPAQs.

2.6.1 Fusion API Measurements

Figures 10 and 11 show the cost of the DFuse API. In Figure 10, each API cost has 3 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 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

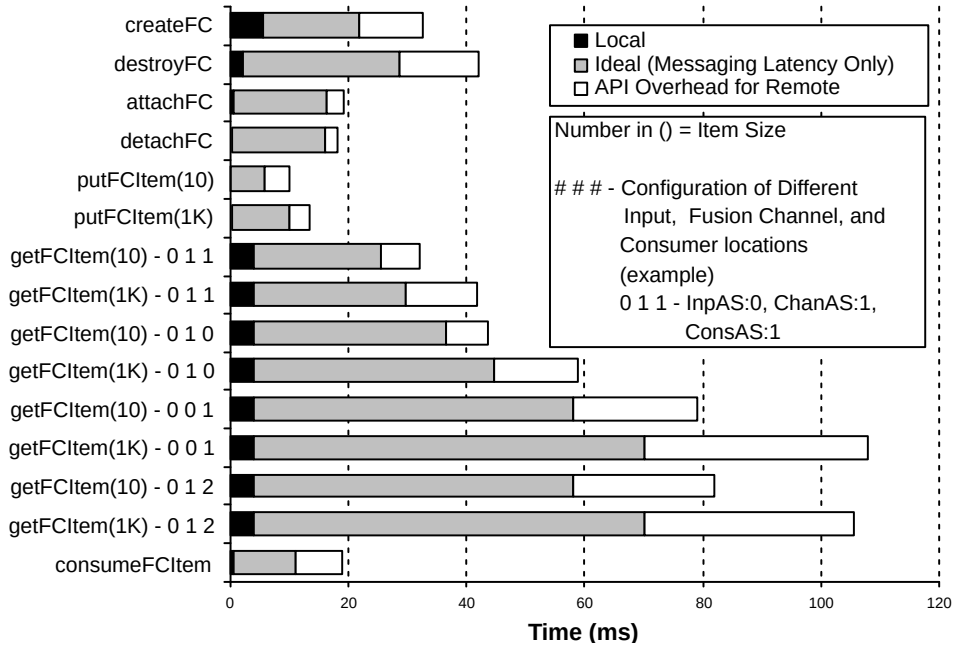

Figure 10: Fusion Channel APIs' cost. See Figure 11 for moveFC cost.

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 results shown in Figure 10, we create fusion channels with capacity of ten items and one primitive Stampede channel as input. Reported latencies are the average of 1000 iterations.

On our iPAQ farm, netperf [38] indicates a minimum UDP roundtrip latency of 4.7ms, and from 2-2.5Mbps maximum unidirectional streaming TCP bandwidth. Table 1 depicts how many round trips are required and how many bytes of overhead exist for DFuse operations on remote nodes. From these measurements, we show messaging latency values in Figure 10 for ideal case costs on the farm. We calculate these ideal costs by adding latency per round trip and the cost of the transmission of total bytes, presuming 2Mbps throughput. Comparing these ideal costs in Figure 10 with the actual total cost illustrates reasonable overhead for our DFuse API implementation. The maximum cost of operations on a local node is 5.3ms. For operations on remote nodes, API overhead is less than 74.5% of the ideal cost. For operations with more

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

Table 1: Number of round trips and message overhead of DFuse. See Figures 10 and 11 for `getFCItem` and `moveFC` configuration legends.

than 20ms observed latency, API overhead is less than 53.8% of the ideal cost. This figure also illustrates that messaging constitutes the majority of observed latency of API operations on remote nodes. Note that ideal costs do not include additional computation and synchronization latencies incurred during message handling.

The placement module may cause a fusion point to migrate across nodes in the sensor fusion network. Migration latency depends upon many factors: the number of inputs and consumers attached to the fusion point, the relative locations of the node where `moveFC` is invoked to the current and resulting fusion channel, and amount of data to be moved. Our analysis in Figure 11 assumes a single primitive stampede channel input to the migrating fusion channel, with only a single consumer. This analysis shares the same ideal cost calculation methodology as for Figure 10. 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 `moveFC`.

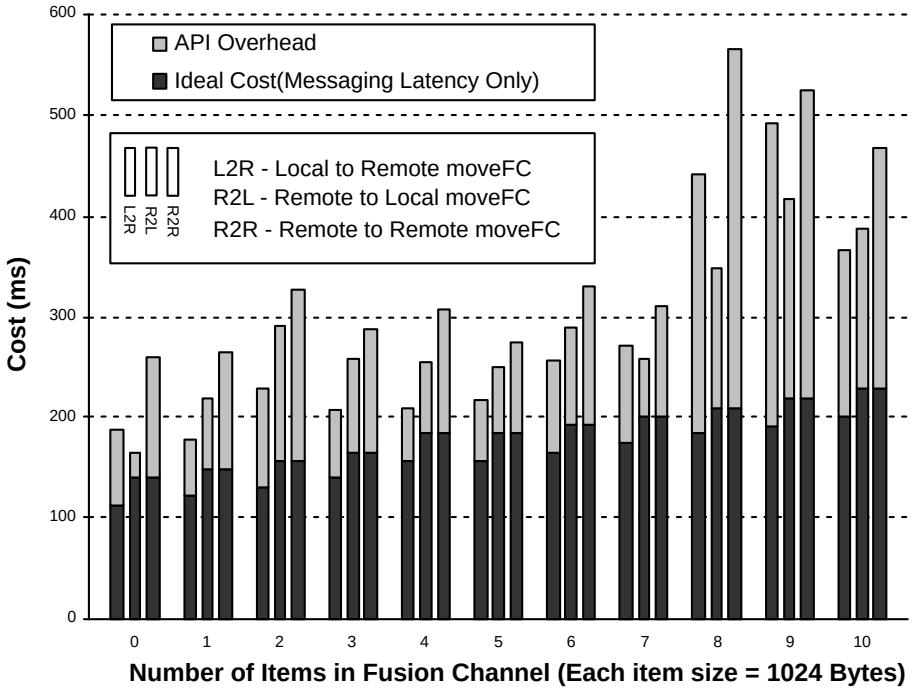

Figure 11: Fusion channel migration (moveFC) cost

2.6.2 Placement Algorithm Measurements

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

Figure 13 shows the topological view of the iPAQ farm used for the tracker application deployment. It consists of twelve iPAQ 3870s configured identically to those in the measurements above. Node 0, where node i is the iPAQ corresponding to ith node of the grid, acts as the sink node. Nodes 9, 10, and 11 are the iPAQs acting as the data sources. The location of *filter* and *collage* fusion points are guided by the placement module.

The placement module simulator runs on a separate desktop in synchrony with the fusion module. At regular intervals, it collects the transmission details (number of bytes exchanged between different nodes) from the farm. It uses a simple power model (discussed later) to account for the communication cost and to monitor the

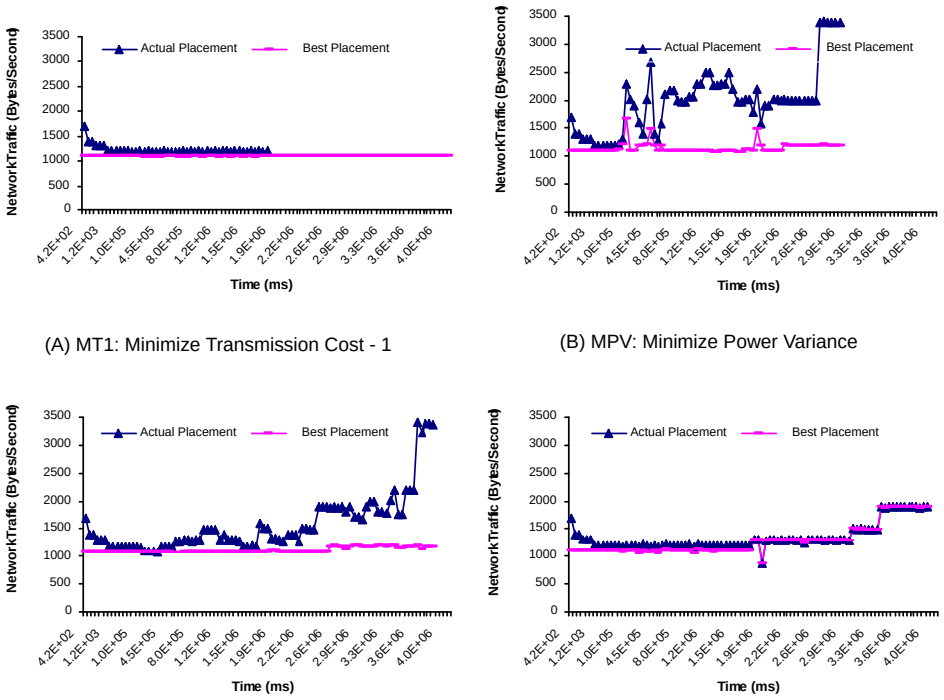

Figure 12: The network traffic timeline for different cost functions. X axis shows the application runtime and Y axis shows the total amount of data transmission per unit time. Optimization runs until 2000ms, and then maintenance phase commences.

power level of different nodes. If the placement module decides to transfer a fusion point to another node, it invokes the `moveFC` API to effect the role transfer.

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

The network is organized as the grid shown in Figure 13. 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 upon the amount of data a node transmits or receives. The power consumption corresponds to ORiNOCO 802.11b PC card specification [45]. Our current power

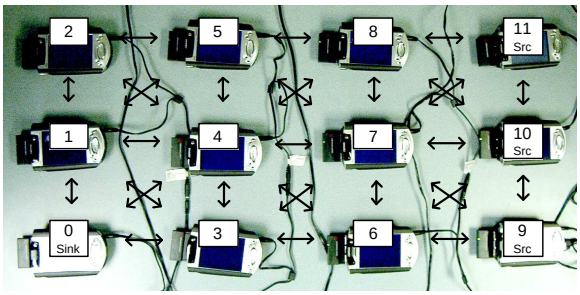

Figure 13: iPAQ Farm Experiment Setup. An arrow represents that two iPAQs are mutually reachable in one hop.

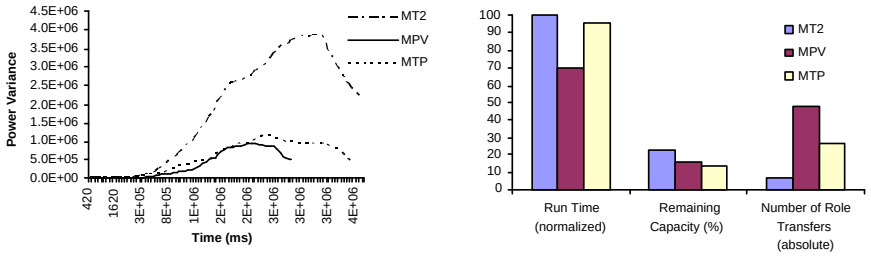

Figure 14: Comparison of different cost functions. Application runtime is normalized to the best case (**MT2**), and total remaining power is presented as the percentage of the initial power.

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 100ms to determine if role transfer is indicated by the cost function. After optimization, the algorithm runs in maintenance phase until the network becomes fragmented (some consumer cannot reach one of its inputs). During the maintenance phase, role transfer decisions are evaluated every 50 seconds. The role transfers are invoked only when the health improves by a threshold of 5%.

Figure 12 shows the network traffic per unit time (sum of the transmission rate of every network node) for the cost functions discussed in Section 2.4.3. It compares the network traffic for the actual placement with respect to the best possible placement

of the fusion points (best possible placement is found by comparing the transmission cost for all possible placements). Note that the application runtime can be increased by simply increasing the initial power level of the network nodes.

In **MT1**, the algorithm finds a locally best placement by the end of optimization phase itself. The optimized placement is only 10% worse than the best placement. The same placement continues to run the application until one of the fusion points (one with the highest transmission rate) dies, i.e. the remaining capacity becomes less than 5% of the initial capacity. If we do not allow role migration, the application will stop at this time. But allowing role migration, as in **MT2**, enables the migrating fusion point to keep utilizing the power of the available network nodes in the locally best possible way. Results show that **MT2** provides maximum application runtime that is more than twice as long as that for **MT1**. The observed network traffic is at most 12% worse than the best possible for the first half of the run, and it is same as the best possible rate for the latter half of the run. **MPV** performs worst, while **MTP** has comparable network lifetime as **MT2**. Figure 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 12B), it provides the best (least) power variance compared to other cost functions (Figure 14A). 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 14B). These results show that the choice of cost function should be dependent upon application context and network condition. If, for an application, role transfer is complex and expensive, but network power variance is not an issue, then **MT2** should be preferred. However, if network power variance needs to be minimized and role transfer is inexpensive, **MTP** should be preferred. Simulation results for other task graph configurations have been found

to provide similar insight into the cost functions' behavior.

2.6.3 Discussion

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

2.7 Related Work

Data fusion, or in-network aggregation, is a well-known technique in sensor networks. Research experiments have shown that it saves considerable amount of power even for simple fusion functions like finding min, max or average reading of sensors [33, 21]. While these experiments and others have motivated the need for a good role assignment approach, they do not use a dynamic heuristic for the role assignment and their static role assignment approach will not be applicable to streaming media applications.

DFuse employs a script based interface for writing applications over the network similar to SensorWare [5]. 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 [33], look at a sensor network as a distributed database and provide a query-based interface for programming the network. TAG uses an SQL-like query language and provides in-network aggregation support for simple classes of fusion functions. But TAG assumes a static mapping of roles to the network, i.e. a routing tree is built based on the network topology and the query in hand.

Recent research in power-aware routing for mobile ad hoc networks [52, 8] proposes power-aware metrics for determining routes in wireless ad hoc networks. We use similar metrics to formulate different cost functions for our DFuse placement module. While designing a power-aware routing protocol is not the focus of this thesis, routing protocol information could be used to define more flexible cost functions.

2.8 DFuse Framework 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 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 increases 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 use models of our DFuse framework as the basis for

studies in the following chapters, and give directions for future framework research in Chapter 6.

CHAPTER III

MSSN: A SIMULATOR FOR EVALUATING DFUSE MIDDLEWARE

3.1 Introduction

This chapter presents the motivation, design and implementation of our simulator for evaluating the DFuse middleware in a much larger context of application workloads, SN devices and scales than afforded by our implementation of DFuse in Chapter 2. Our simulator enables analysis of application performance under a variety of device radio, CPU and memory power models.

Recent SN middleware, such as DFuse (Chapter 2, [31]) and SensorWare [5], provide specific techniques to support our vision of future SN fusion applications, including efficient stream transport, data fusion capabilities, and dynamic placement of processing in the network. Taking from DFuse, we use data fusion to describe the operation of any general purpose application function that transforms, or fuses, one or more input streams into one output stream. Examples of fusion functions are image comparators and MPEG compressors. Fusion functions connect to form the task graph for a fusion application. The location of each fusion function's evaluation within the network directly impacts communication costs for fetching inputs and transmitting outputs. DFuse monitors fetching and transmitting of data to infer communication costs and to model sensor battery levels. Based on application provided policy, in the form of a cost function, DFuse locally decides whether to migrate a fusion function to a neighbor node. Example cost functions could target transmission

cost minimization or battery level equalization. Fusion function migration is essentially dynamic adaptation of the application task graph's mapping to the underlying SN. DFuse can also prefetch and buffer streaming input items on behalf of fusion functions to reduce latency and enable pipelined parallelism of stream processing.

To study how these adaptive migration and prefetching middleware mechanisms perform, we need to include more than DFuse's cost model that is based only on application-level communication [31]. Node components such as CPU and memory impact energy consumption and processing latency. More generally, for any future SN, it is not always clear how to obtain greatest efficiency given the array of SN node hardware and middleware capabilities. For example, consider decreasing CPU clock frequency to save processing energy. While possibly increasing network lifetime, slower processing may increase latency beyond tolerable limits for a compute-intensive application. On the other hand, if an application is communication-intensive and performance is limited by available network bandwidth at a node, then slower processing may save energy without compromising performance. Similar, often non-intuitive, optimizations are possible, motivating detailed studies to guide SN tuning.

Existing SN evaluation frameworks do not provide the combination of flexible node CPU, memory and radio power and latency models, and an extensible set of middleware mechanisms for fusion applications including adaptive migration and prefetching. We develop a SN planning tool, *MSSN* (Middleware Simulator for Sensor Networks), enabling evaluation of a variety of fusion application workloads, middleware features, and node architectures. Using observed latency, throughput, number of processed items, and network lifetime as figures of merit, we evaluate trade-offs among the above parameters. Building on the campus surveillance application from Chapter 1 and the DFuse framework from Chapter 2, this chapter presents MSSN. This event driven simulator enables modeling of futuristic SN with on the order of 1000 nodes, surpassing previous DFuse simulators and implementations [57] capable

of modeling the middleware on only tens of nodes with neither CPU nor memory power models.

The rest of this chapter is structured as follows. Pertinent related work is presented in Section 3.2. Our simulator's design and implementation, including initial application workload, device and middleware models is presented in Section 3.3. Results of using MSSN to quantify SN performance follow in Chapter 4.

3.2 Related Work

Our vision of future SN applications is reinforced by recent middleware initiatives such as IrisNet [14], SensorWare [5] and DFuse [31] to support smart sensors, and current technology trends such as voltage, bandwidth, and frequency scaling [40, 48, 44, 12].

IrisNet builds on the ubiquity of cameras and their role in serving as a source for abundant information. IrisNet uses a database centric approach to publish generated data, and uses XML to query the network. Sensors send their data to a resource-rich proxy that processes the data to find interesting features and update the sensor database distributed across the network. As the processing nodes are always powered, IrisNet does not care about optimizing energy consumption.

SensorWare is a framework for programming sensor networks, but its features are orthogonal to DFuse, core to our MSSN middleware model: 1) SensorWare does not use an integrated strategy for dynamic processing placement to minimize cost, and 2) DFuse provides a fusion point abstraction with prefetching and streaming buffer support. SensorWare "scripts" needing these capabilities would have to encode them in the application.

DFuse middleware supports migration of processing across nodes to save energy or to balance load using application-specified cost functions. However, previous results [31] account only for transmission energy and not for energy consumed by processing or memory at a node. Also, the DFuse study, while based on an actual SN

deployment, is limited to 12 nodes and is bound to their hardware characteristics. Our simulation-based study supports evaluation of varying middleware and architectural characteristics for SN larger than 1000 nodes.

A primary goal of MSSN is to incorporate more representative sources of SN energy usage beyond the radio, such as CPU and memory hardware. Existing wireless simulators do not incorporate fusion point mechanisms we evaluate (GloMoSim [3], NESLsim [47]), are not scalable (ns2-wireless [9]), or are specific to contemporary motes (TOSSIM [32] , Em* [16], and Prowler [50]). MSSN focuses on modeling energy usage and performance of DFuse-like middleware for a whole range of futuristic sensor node architectures, requiring a fairly detailed implementation of the middleware inside the simulator and a decoupling from a specific target device. Coupling MSSN with SN MAC and Routing layer simulators, such as GlomoSim or NESLsim, would speed relaxation of current immobile sensor node and ideal MAC and routing layer assumptions used by MSSN, and is one possible avenue for future research (see Chapter 6).

In pursuit of quality of service support and saving energy, researchers have developed techniques for adapting the application, middleware [41], OS [39], network protocols [58], and hardware. There is a need to look into adaptation in a coordinated, cross-layer manner and our study is an effort in that direction.

3.3 Evaluation Methodology

To quantify SN performance, we build an event driven Middleware Simulator for Sensor Networks (MSSN) encompassing fusion application workloads, devices, and middleware. We use this simulator to show trade-offs among tunable middleware and architecture parameters for each of two large application workloads, and analyze these results to reveal non-intuitive tuning guidelines to assist SN planning. Previous DFuse evaluations are specific to a real iPAQ deployment and a simple power model

that accounts only for stream communication energy cost, and show that **MTP** and **MT2** yield longer lifetime than **MPV** [31]. Furthermore, these previous studies are limited in both topology and application scale.

To constrain the search space, we 1) use a model of DFuse as our middleware to perform fusion, migration, and prefetching of items needed for fusion; 2) assume all SN nodes' initial battery levels and CPU, radio, and memory power models are the same for an experiment; 3) use video streams in place of all kinds of streams; 4) assume fixed node locations, but allow fusion points to relocate using the DFuse migration feature; 5) assume greedy sinks that immediately request their next inputs, maximizing throughput; and 6) assume fusion functions exhibit static behavior (their latency and footprints do not vary with time).

3.3.1 Application Workloads

For our large-scale application workloads, we model the motivating campus surveillance application, presented in Chapter 1, as a general fusion application that performs hierarchical in-network processing on streams produced initially by cameras. We model this application with a representative set of fusion functions shown in Table 2. `Collage` output is a concatenation of two input images, while `Select` outputs the brighter of two input images. `MotionDetect` is based on inter-frame differencing, and `FD/FR` is a CPU-intensive face detection and recognition function.

Parameters in Table 2 directly impact energy consumption. Cycle counts correspond to active CPU energy consumption. Input and output footprints determine streaming communication costs. Persistent fusion function state, such as face detector models, plus buffered inputs and outputs determines the amount of state communicated during fusion point migration. For FD/FR, we use previously published time measurements [30] using a 206MHz SA-1100 iPAQ H3600. We report results from our benchmarks for the remainder, using a similar 206MHz iPAQ 3870 platform running

Fusion Function	Footprint (KB) Communication Input	Output	Persistent State	Measured Time (ms)	Instruction Count	Derived Cycles	CPI
`Collage`	56*2	112	0	3.9	309K	803.4K	2.59
`Select`	56*2	56	0	3.5	327K	721K	2.20
`EdgeDetect`	56	56	0	12.7	1844K	2616.2K	1.42
`MotionDetect`	56	56	94	4.9	N/A	1009K	N/A
`FD/FR`	30	30	3500	9510	N/A	1959**M**	N/A

Table 2: Fusion Function Costs: Communication and persistent state footprints are from code inspection, and required number of processor cycles are derived from microbenchmarks. Required cycles are confirmed by instruction counts from code inspection and reasonable derived CPI where available.

Linux "familiar" distribution version 0.6.1, verifying measured time using CPI derived from instruction counts. Instruction counts are determined by inspection of assembly code generated by the gcc 2.95.2 ARM cross-compiler with `-c -g -Wa,-a,-ad` options. These functions are small and iterative, so SA-1110 with 16K-Icache and 8K-Dcache should obtain frequent cache hits and low cycles per instruction (CPI) as shown in Table 2. We model a *CPU-intensive* workload with a task graph including FD/FR, and a *communication-intensive* workload without using FD/FR.

To construct an arguably simple model for a surveillance application task graph, we compose functions listed in Table 2 together, observing their arity: Collage and Select each fuse two inputs into one output, while EdgeDetect, MotionDetect and FD/FR each transform a single input into one output. For these functions, all inputs and outputs are a frame of video. Figure 15 depicts the general structure we construct: each input of a two-input fusion function comes from the output of a distinct one-input fusion function, and the input of a one-input fusion function comes from the output of a two-input fusion function. Following these simple rules, we construct a tree-like task graph, where the maximum length path from the root (sink) to any camera (source) is constant. This construction methodology enables easy scaling of the number of fusion functions, as the number of fusion operations is a function of the

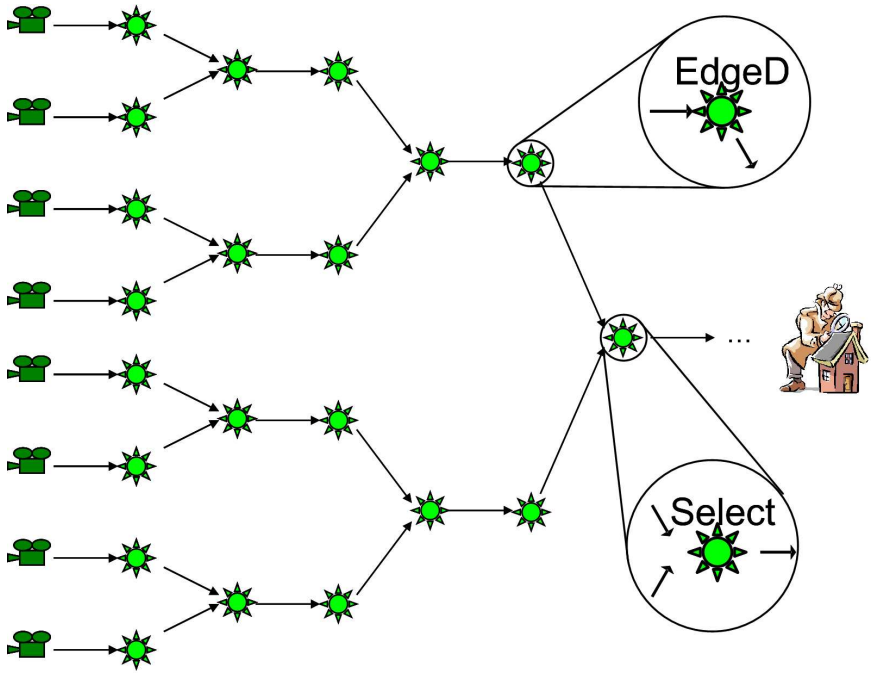

Figure 15: Expanded view of the campus-wide surveillance application model's task graph

number of cameras in this model. We randomly choose either Collage or Select when connecting a two-input function, and likewise randomly choose from EdgeDetect, MotionDetect, and FD/FR when connecting a one-input function.

For these initial studies with MSSN, we constrain our application's behavior to be static in terms of the amount of processing required at each kind of fusion point for every iteration on streaming data. For example, FD/FR will always take 1959 million cycles of a SN node's CPU to process one input image of size 56KB. We relax this static application behavior constraint later in Chapter 4.

Astute readers will observe that function input and output data sizes differ somewhat in Table 2. Microbenchmarking existing functions from a variety of sources leads to these differences. For the purpose of calculating transmission energy and latency between functions where the I/O sizes differ, we assume that each item has the size of the producer's output. For example, if a Select gets one of its inputs from

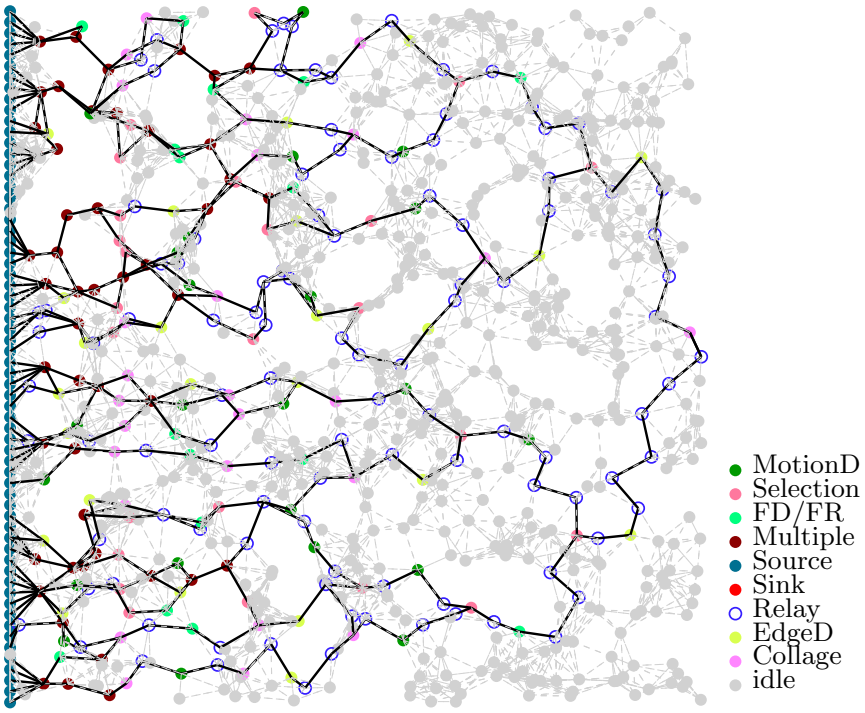

Figure 16: Sample SN topology showing an initial overlay mapping for the campus-wide surveillance application

a FD/FR, then each of the items communicated to Select from FD/FR is assumed to be 30KB.

Given the task graph, we then construct the initial overlay network. First, we randomly scatter SN nodes within a square "campus". Next, we use an algorithm to map the task graph's fusion functions to network nodes. The result of this algorithm approximates superimposing Figure 15 onto a square campus. As mapped fusion functions (now, fusion points) may require multi-hop connections to reach their neighbors in the task graph, we connect fusion points using relay nodes as necessary. Relays act as data forwarders. We place relay nodes along a shortest hop path between fusion points, forming multi-hop relay chains to connect the overlay.

Figure 16 depicts a sample overlay network topology from our experiments, prior to any fusion point migrations and node failures. 64 cameras are located to the left and the sink is located in the middle of the right edge. 800 additional nodes are placed

within the campus. Darker lines indicate actively mapped connections between fusion points, along relay chains where necessary. Some nodes host more than one fusion point simultaneously.

Our previous small application studies in Chapter 2 ignore node CPU and memory costs and indicate that **MT2** and **MTP** achieve greater lifetime extension than **MPV**. For our initial large application tradeoff study using MSSN, we choose the **MPV** cost function to direct migration. We expect that lifetime will be extended with this cost function for large applications similar to our earlier small application study. Based on trends observed in small application studies, we expect that **MT2** and **MTP** would yield even greater lifetime extensions than **MPV** for large applications, but we wish to observe a lower bound on lifetime extension by using **MPV**.

3.3.2 Power Models

To show trade-offs among tunable parameters of middleware and node architecture for the large-scale workloads, we use power models based on contemporary devices.

3.3.2.1 Processor Power Model

Voltage scaling is a popular technique for saving energy in today's CMOS microprocessors. Energy consumption in CMOS circuits can be accurately represented as a simple equation [6] that says clock frequency reduction linearly decreases energy consumption, and voltage reduction results in a quadratic decrease in energy consumption. From the SA-1100 specification, we find that the processor consumes at most 230 mW at 133 MHz, and at most 330 mW at 206 MHz at 1.5 Volts [24]. Power measurement experiments on SA-1100 microprocessor indicate that power requirement increases monotonically with increase in clock frequency [54, 44]. Earlier research on SA-110 confirms the linear relationship [37]. We use a linear model for energy consumption based on these two data points for determining energy usage at clock speeds from 59-206MHz.

Mode	Power (mW) 802.11b @ 11Mbps	Power (mW) Bluetooth @ ∼721 Kbps
`Transmit`	1600	102
`Receive`	950	165
`Listen`	805	66
`Sleep`	60	30

Table 3: Radio power model

3.3.2.2 Memory Power Model

Memory is also a major source of energy consumption, especially for memory-intensive workloads [54]. But, its impact on overall energy consumption is difficult to predict because a change in clock frequency changes the available memory bandwidth in a non-linear fashion, and it also affects the energy consumption for memory access [44]. We use a simplified model for memory access energy breakdown. We assume that memory works in three modes similar to the operation of Direct Rambus DRAM (RDRAM): *active*, *idle*, and *sleep*. Power consumption in these modes is as cited by Fan *et al.* [12] (300 mW *active*, 20 mW *idle*, 3 mW *sleep*). We assume that while the CPU is executing a fusion function, the whole memory is being accessed actively. In a realistic scenario, CPU execution and memory activity will be interleaved, and memory will keep switching between active and standby modes during CPU execution. Our assumption accounts for the worst case energy consumption by memory and it also simplifies simulation.

3.3.2.3 Communication Power Model

Radio is the communication medium in the SN domain we consider, and it is the most power hungry among the components of the SN node architecture. Hence, saving communication energy is critical to increasing application lifetimes. Fusion applications such as campus surveillance will require significant bandwidth relative to the limited communication capabilities of contemporary motes. We therefore choose to

use models of contemporary radios for short-range digital communication with bandwidths over 500Kbps. We choose two common packages with differing bandwidths and power requirements to observe performance impacts of using a relatively energy-efficient, slow radio compared to using a more power-hungry, fast radio. We expect that a similar variety of radio packages will become available in form factors sufficient for SN nodes in the future.

For our simulations, power consumption for different radio modes is shown in Table 3. We use numbers corresponding to two different bandwidths: one with 802.11b (Orinoco card) [45], and another for Bluetooth [49]. Though 802.11b can operate at multiple data rates, corresponding power specifications are unavailable for Orinoco. We use only one transmission rate for each of the two radios. Bluetooth numbers are valid only for shorter transmission range ($\sim$66 ft for Class 2 devices) compared to the range of 802.11b ($\sim$500 ft in open and $\sim$125ft in closed space). We scale network size with respect to radio range to have the same initial topology across our experiments.

We observe that energy drain by idle nodes waiting in *listen* mode for long periods of time may dominate overall network energy use. One way to reduce this cost is to impose a duty cycle on the network nodes, enabling them to incur lower *sleep* radio costs for much of the time they would have been in *listen* mode otherwise. This is a common practice among today's motes, designed for sleeping over 99% of the time. We therefore include a variant of the radio power model that assumes an optimal sleep duty cycle such that a radio never uses *listen* mode, but uses *sleep* mode instead. Having such a duty cycle incurs scheduling overhead. Rather than imposing an arbitrary overhead onto our general SN model, we choose to explore the lower bound of radio cost in *listen* mode by including this optimal sleep mode as an optional radio power model. Previous research shows that such a lower bound assumption is reasonable by using an efficient radio to wake the main communication radio when necessary [2].

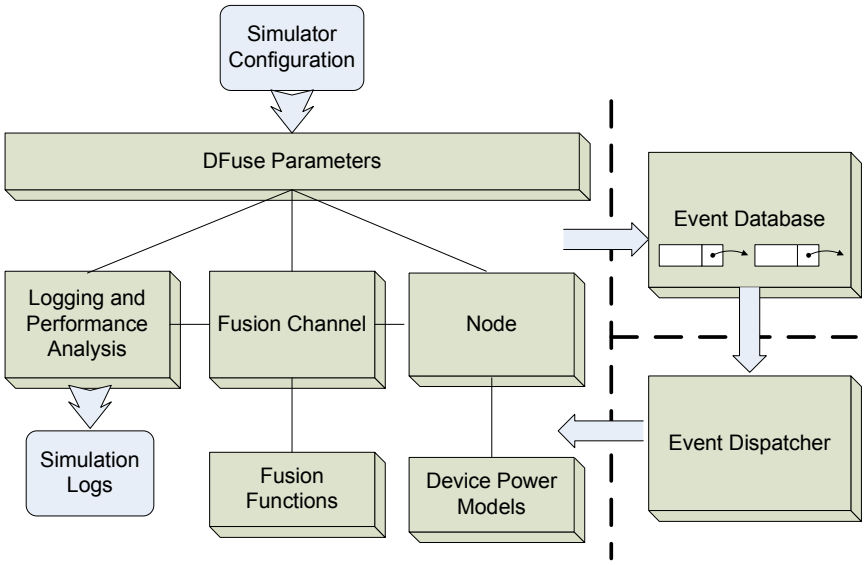

Figure 17: MSSN Architecture Diagram

3.4 Architecture of the Simulator

We present here the event-driven simulator, MSSN, we have built to evaluate future SN under varying architectural, middleware, and workload characteristics. It consists of approximately 9000 lines of C++ code, and is available for download at http://www.cc.gatech.edu/∼wolenetz/files/MSSN-snapshot-06-22-2005.tar.gz. The simulator includes a rich set of configuration options and is extensible to support additional simulated middleware features.

Figure 17 presents the high-level architecture for MSSN. As is common for discrete event simulators, MSSN is comprised of three categories of components: event database, event dispatch, and logic specific to the system being modeled. The responsibilities for each component are as follows:

- The **DFuse Parameters** component contains logic to retrieve and handle the configuration of the simulator. Important configuration options include topology management (random or grid, and number of nodes), initial task graph

deployment corresponding to the desired application workload, and parameterization of node capabilities including device power models and initial energy. This component also contains logic to model the ideal routing layer, including the Floyd-Warshall algorithm and service routines used to destroy and rebuild input and output relay chains connecting the overlay network.

- The **Logging and Performance Analysis** component contains event handlers for periodically recording the performance of the simulated SN, including extensive code to handle the simulated annealing oracles discussed later in Chapter 4.

- The **Fusion Channel** component models instances of DFuse fusion channels, including the buffering of input items and caching of output items. Fusion channels invoke application-specific **Fusion Functions** upon receipt of all of the next inputs from each upstream producer. Sources and sinks are specialized fusion channels with different logic to handle input, output and computation dispatch. The Fusion Channel component includes logic necessary to handle the DFuse role-assignment heuristic with significant complexity associated with correctly handling fusion channel migration. Cost functions for the placement heuristic are in this component. It also includes logic for the predictive CPU scaling heuristic discussed later in Chapter 4.

- The **Node** component handles the modeling of node instance's resources, including energy and neighbor connectivity. In cooperation with the **Device Power Models** component, nodes are also responsible for scheduling the radio and processor devices. Scheduling on the processor involves potentially time-sharing the CPU across multiple fusion channels mapped to the same node. Radio scheduling is done on a first-come, first-serve basis. When a node wishes to transmit a message to a neighbor, the transmission window is scheduled to occur as early as possible without overlapping previously allocated radio time

Event	Target Component	Description
`InitEvent`	DFuse Parameters	Triggers simulator startup
`ChanStart`	Fusion Channel	Triggers startup of a fusion channel
`GetNext`	Fusion Channel	Models receipt of a request from a downstream consumer for its next input from this channel
`GotNext`	Fusion Channel	Models receipt of a response from an upstream producer with the next item
`FusionComputed`	Fusion Channel	Models the completion of a processing iteration of the associated fusion function on the node's CPU
`CheckCosts`	Fusion Channel	Models periodic triggering of the local placement heuristic for this channel
`MigrateChanState`	Fusion Channel	Models the completion of migration of this fusion channel to a neighbor node
`StatGather`	DFuse Parameters	Triggers periodic statistics logging for performance analysis

Table 4: Events handled by MSSN's middleware logic

on the transmitting and receiving nodes. Relaxing this pair-wise ideal MAC is potential future work.

- The **Event Database** and **Event Dispatcher** components control the execution of middleware logic in simulated time. Events are tagged with a dispatch time, beginning at time zero. Initially, a simulator start event is inserted into the event database, and the event dispatcher is given control. The dispatcher retrieves an earliest time event from the database and executes its logic. Event logic transfers control to one of the components in the modeled SN middleware. As the event is handled by the simulated middleware, further events may be inserted into the event database. Eventually, control returns to the dispatcher, which iterates until either the event database is empty or the simulated middleware triggers simulator shutdown.

The bulk of the simulator is concerned with accurately modeling the middleware

with events ranging from message delivery to migration completion. Table 4 highlights the major events handled by middleware logic in MSSN. Significant complexity is involved in the handling of these events. For example, if a node on one of a fusion point's input relay chains is nearing energy exhaustion, the simulator needs to correctly destroy and rebuild that input relay chain, rebuilding the routing tables during the process. Items in-transit on the relay chain need to be accounted for, and the state of both the producer and consumer ends of the relay chain needs to be updated to account for the change. Migration uses this basic relay chain rebuild mechanism to implement the remapping of a fusion point to a neighbor node. However, to prevent the need for the old fusion point host to forward later communications to the new host, we use "weak" migration, delaying until there are no items in-transit along any of the migrating point's input and output relay chains, and then transferring buffers and runtime state associated with the fusion point to the target node. Prefetching is implemented by giving each fusion point a buffer to store fused results into, and by attaching a sink directly to every fusion point. These special sinks incur no energy or delay costs, but they drive the fusion points to request and fuse as fast as possible while they have room in their local output buffers.

We use an ideal MAC model that incurs neither energy nor latency overhead due to packet loss. The simulator serializes, in simulated time, all access to radio channels between nodes on a pair-wise basis, modeling simple lossless and collision-free MAC. Future work may relax this ideal assumption, but we are looking for basic trends here. We assume that the routing layer provides notification of pending node battery failure piggy-backed on top of regular traffic, enabling route maintenance. We currently impose no modeled overhead for local calculation of the cost function, as this occurs relatively infrequently and only incurs minimal communication with immediate neighbor nodes (we do account for migration costs, though). We do not model the cost of initial application deployment currently, as this is highly dependent on many

factors, primarily sensor node OS and bootstrapping characteristics. We assume a simplified fusion point model, wherein fusion points only request the immediate next set of input items, performing "optimistic" prefetching by trying to keep their output buffer full (limited to a capacity of 5 fused items in these initial experiments).

3.5 Modularity of the Simulator

We have designed MSSN to model fusion applications supported by DFuse, and to be flexible with respect to different device power models, workloads and integration of additional local resource schedulers at fusion points. Currently, MSSN models a SN as a collection of nodes and communication links, much as in Figure 16. It allows simulation of in-network data fusion on application generated items using application specified fusion functions. It also enables fusion point migration across nodes driven by an application specified cost function (we use **MPV** in these initial experiments). MSSN supports more than 1000 simulated sensor network nodes, beyond our capability to actually deploy for real-world experiments. The limiting factor is recalculation complexity of routing tables using the $O(n^3)$ Floyd/Warshall *All Pairs Shortest Path* algorithm, which happens every time a node dies due to low energy. MSSN models shared scheduling of CPU and radio resources by multiple concurrent resource requests. For example, if a node hosts two fusion points that simultaneously begin fusion function execution, the simulator serializes their access to the CPU in simulated time.

Each of the assumptions listed in Section 3.3 to constrain the search space are embodied in the various MSSN components. An important issue addressed here is how flexible MSSN's design is to changes in these assumptions to simulate other domains (e.g. a set of middleware mechanisms other than DFuse, a different MAC, different power models, etc.) We address this modularity and flexibility question by indicating which component(s) of MSSN are responsible for each assumption, and

give examples of the expected amount of changes necessary for MSSN to be adapted for hypothetical assumption changes:

- The model of DFuse is highly integrated into MSSN. For example, DFuse's Fusion Channel abstraction forms the core of the application workloads simulated with MSSN. Simulating the DFuse role-assignment heuristic is also part of the responsibility of the Fusion Channel component of MSSN. This high degree of integration is reasonable, as we expect fusion applications to require an abstraction like Fusion Channels to manage computation and communication between fusion points. The current model of Fusion Channels in MSSN's implementation allows for a subset of the DFuse Fusion Channel package presented in Chapter 2. For instance, partial fusion is not allowed: dispatching of fusion function execution only occurs once all inputs have been received, regardless of delay in fetching inputs. Also, each task graph vertex is restricted to fetching, processing and producing items in timestamp sequence. Deviation from these implementation assumptions would require significant extra modeling of buffer management within the Fusion Channel component. As relays are specialized Fusion Channels, the relay chain teardown and construction service routines in the DFuse Parameters component would also need adaptation to enable correct fusion point migration and re-routing of the overlay network if these assumptions change.

- MSSN is quite flexible with respect to power models for each of CPU, radio and memory. So long as the pair-wise ideal MAC, ideal routing, and pessimistic memory access models are not changed, adapting MSSN to different power models corresponding to different radio transmission, reception, listen, and sleep costs; different CPU active and idle costs; and different global-memory active access and idle costs is as simple as changing a few short functions within

the Power Model component. However, modeling additional modes for each of these devices may involve significant complexity. Currently, the Node component tracks the mode of each node's CPU, radio and memory devices. Fusion processing and message transmission events trigger mode switching and latency and energy cost accounting. Changing the profile of available modes would require significant modification of accounting logic. MSSN's implementation structure distributes this logic, making correctness of modifications a significant issue.

- Currently, all nodes begin with homogeneous battery levels and CPU, radio and memory power models. Battery levels are independently maintained within each node instance, so changing this resource to be heterogeneous at startup is trivial during node construction. However, power models are currently contained within a global set of service routines. Enabling heterogeneous power models across nodes would require refactoring the Node and Power Model components such that each Node instance is associated with a Power Model instance. Also, the topology construction routines within the DFuse Parameters component would need changing to properly construct heterogeneous nodes, directed by new options from the input configuration file.

- MSSN workloads currently use video streams in place of all kinds of streams. Actually, modeling of fusion functions within MSSN is quite general to any kind of streaming data. So long as the workload's fusion functions conform to the parameters presented in Table 2, adding new types of fusion functions is trivial. The workloads used in this thesis are based on microbenchmarks of video processing functions, but other modalities of streams could be modeled easily. Making fusion functions perform actual application behavior such as actually processing images would be trivial. Modifications would involve making

the Fusion Function component execute actual application code, assuming the application code is portable to the simulator's execution environment. However, SN device-specific behavior, such as actual image capture at sources and accounting properly for energy used during fusion processing would take significant effort. For example, the simulator's execution environment may have optimized floating point capabilities relative to the SN node CPU being modeled. Determining the actual number of CPU cycles necessary on a SN node CPU for each iteration of a real application's fusion function run on the optimized CPU during simulator execution may require significant effort.

- MSSN is able to model SN scales up to about 1000 nodes during simulations with reasonable execution time by having simple MAC, routing and transport layer models. Observing effort taken by other wireless network simulators such as GloMoSim [3] and ns2-wireless [9], simulating collision domains efficiently is required to model topology scales that MSSN targets. Currently, the simple, ideal pair-wise MAC model used by MSSN enables this scaling. If the MAC layer were changed to a more general collision domain model, then significant effort in making the new implementation efficient at runtime would be necessary. Also, if communication failures for reasons other than node power failure were modeled, then the current ideal routing and reliable transport assumptions embodied in MSSN would need changing. One possible avenue for alleviating this effort might be to interface MSSN with an existing wireless simulator to enable reuse of more general MAC, routing and transport layers. However, there may be significant effort necessary in implementing this interface, including modifying the wireless simulator to trigger "upcalls" to MSSN for energy accounting. Also, MSSN is run as a single-threaded process, while other simulators may be distributed across multiple threads and machines, making any interface significantly more complex.

- The location, or rather, the connectivity of nodes modeled with MSSN is assumed to be static, except for node failures due to low battery levels. Introducing node mobility would require additional events and handlers to adapt node locations and routing tables. The DFuse Parameters component is responsible for topology construction and routing table management. The level of effort required to model mobility is primarily dependent on assumptions about the routing layer. If the ideal routing layer assumption that every node always knows its 1-hop neighbors and the correct shortest path to any other node is used, introducing mobility would be trivial. Changing this assumption would require significant effort. One solution would be to delegate wireless network transport, routing and transmission modeling to an external wireless simulator.

- Extending MSSN's model of the DFuse role-assignment heuristic to evaluate additional cost functions would be quite trivial, so long as the inputs to the cost function are simply derived from SN state. For example, making migration biased towards nodes with faster radios for communication-intensive fusion points and towards nodes with faster processors for computation-intensive fusion points would be simple to encode in MSSN, assuming heterogeneous SN node modeling modifications from above.

- Adding new local resource schedulers to MSSN can be done, as we show in our predictive CPU scaling heuristic case study in Chapter 4. Also, changing the production rate and varying the the number of cycles each fusion function takes dynamically during simulation is demonstrated in this case study. However, changing the actual task graph dynamically would require significant effort. In this case, the fusion channel input and output buffer management routines in the Fusion Channel component, and the "weak" migration scheme encoded in both the Fusion Channel and DFuse Parameters components would require

significant modification.

- Extending MSSN to handle additional on-line statistics gathering is definitely possible. For our DFuse migration scalability case study presented in Chapter 4, we implement detailed simulated annealing analysis routines that are triggered periodically by the `StatGather` event handler in the Logging and Performance Analysis component. However, care should be taken that these analysis routines operate feasibly at the intended workload and topology scales. For example, our **BF** (brute-force) placement oracle in Chapter 4 should only be used for application workload models with very few fusion points.

3.6 Summary

The next chapter gives results using MSSN to quantify SN performance for the initial workload, device and middleware models presented here. Chapter 4 also includes case studies using MSSN to evaluate DFuse's scalability and the performance of a novel dynamic CPU scaling mechanism. These case studies also demonstrate how flexible MSSN is for evaluating performance of a variety of application workload models on SN with a variety of node architectures, SN topologies, local resource management mechanisms (CPU scaling), and on-line performance analysis routines.

CHAPTER IV

CASE STUDIES USING MSSN TO EVALUATE SENSOR NETWORK MIDDLEWARE

With our MSSN simulator of DFuse-like middleware, we have the means to evaluate the performance of fusion applications in a variety of possible configurations of future SN hardware, middleware and application workloads. In this chapter, we present three focused studies that demonstrate the utility of MSSN for answering questions critical to tuning future SN:

- First, we experiment with the workload, device and middleware models detailed in our presentation of MSSN in Chapter 3. Beyond demonstrating basic usage of MSSN, we give quantitative results showing trade-offs among tunable parameters of middleware and node architecture for each of two large scale application workloads. Analysis of these results reveals non-intuitive guidelines to help tune future SN. We quantify figures of merit for a baseline configuration of middleware with only fusion capability as a function of processor speed and radio characteristics. We then quantify figures of merit with prefetching and with both prefetching and adaptive computation migration.

- Next, we perform a scalability analysis of the DFuse fusion point placement mechanism introduced earlier in Chapter 2. While we have already shown that this mechanism can deliver significant lifetime extension depending on the cost function used, we must also determine whether or not it delivers performance improvements at realistic SN topology and application scales.

- Finally, we propose, design, implement in MSSN and evaluate an additional

energy management mechanism that locally, dynamically *scales* a fusion point's CPU speed to conserve energy. As workloads studied with MSSN so far have exhibited static behavior in terms of capture rate and processing intensity, we construct a new surveillance application model that captures the dynamic behavior of the surveillance application presented earlier in Chapter 1. We analyze how our *Predictive CPU Scaling* heuristic can be tuned to stay within application end-to-end latency and productivity performance tolerances while conserving energy (maintaining good SN lifetime).

4.1 Basic Middleware Simulation Results for DFuse

Using MSSN, we have performed experiments with our large-scale application workloads to shed light on the impact on application figures of merit of using combinations of middleware fusion point migration and optimistic prefetching features with varying device CPU speeds and radio characteristics: Bluetooth (*B/T*) vs 802.11b Orinoco (*802*), and normal *Listen* cost vs ideal *Sleep* listen cost, for both our compute-intensive (*CPU*) and communication-intensive (*Comm*) application models. We quantify performance in terms of latency of response to the sink's requests, instantaneous throughput, network lifetime, and number of video frames delivered to the sink per lifetime (productivity). Figure 18 shows performance of the two workloads for a baseline SN configuration without prefetching and migration, Figure 19 shows performance of the two workloads with prefetching of fusion function inputs enabled, and Figure 20 shows performance of the two workloads in the presence of migration using **MPV** along with prefetching. Analysis reveals:

1. In the presence of optimistic prefetching, increasing the radio bandwidth may not improve latency nor throughput for compute-intensive workloads (Figure 19, *B/T-*-CPU* vs *802-*-CPU*). Also, productivity may actually decrease if the change induces extra cost for idle nodes. For example, delivered items per

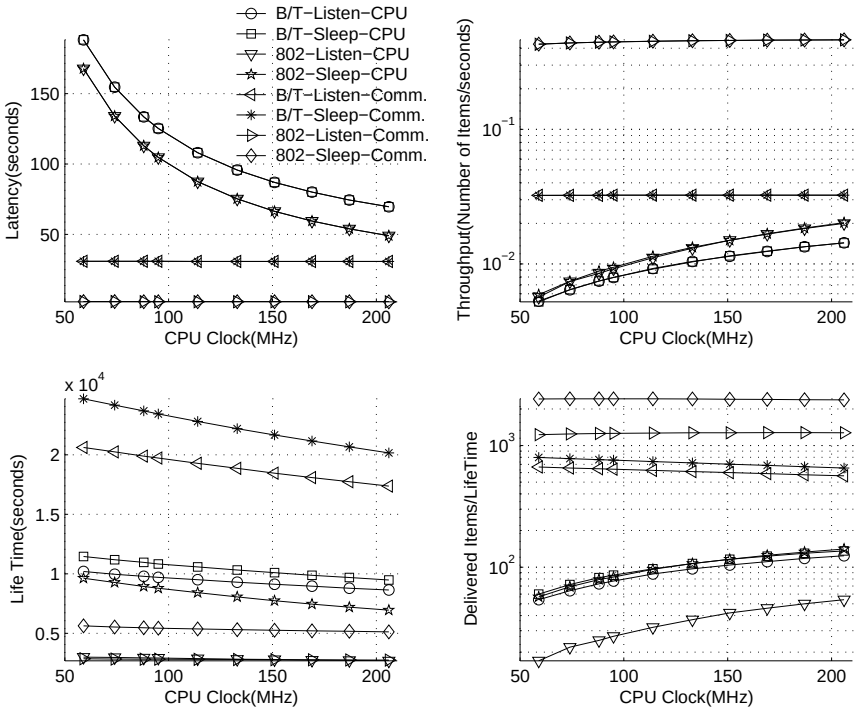

Figure 18: Baseline results: migration and prefetching disabled

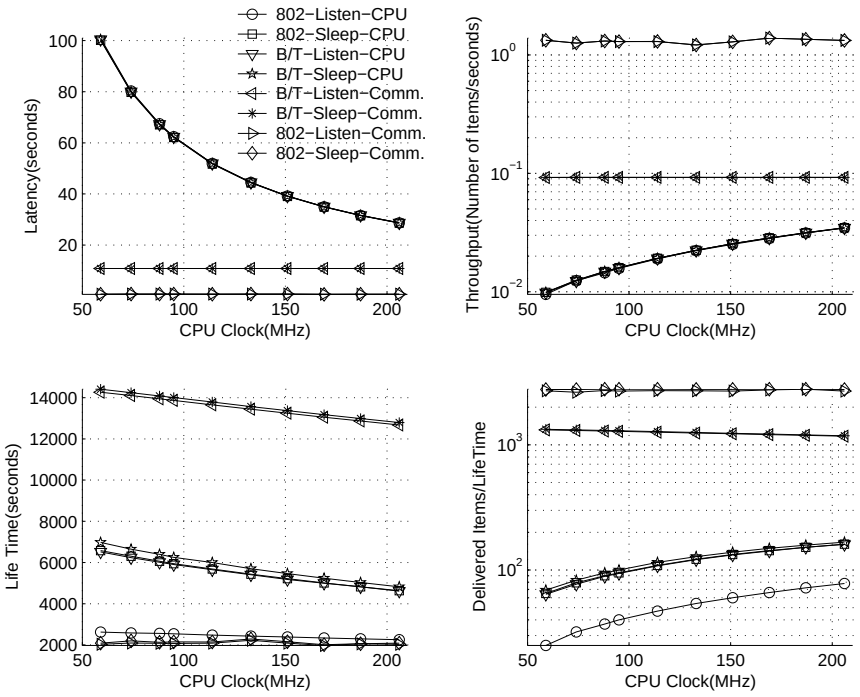

Figure 19: Results with prefetching enabled

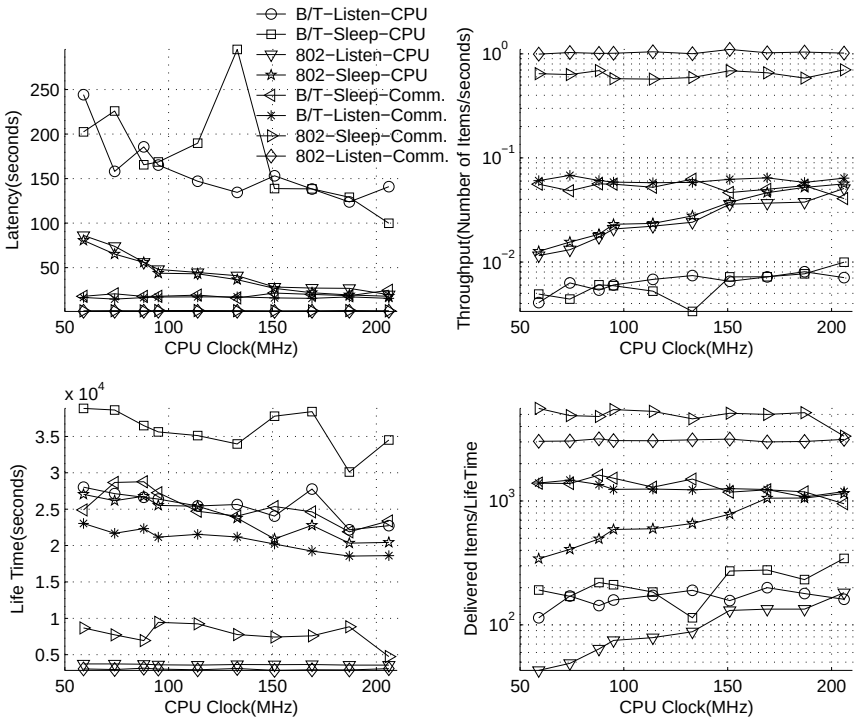

Figure 20: Results with prefetching and migration enabled

lifetime decreases when not using the ideal *Sleep* model and changing to a more expensive radio model in terms of listen cost (Figure 19, *B/T-Listen-CPU* vs *802-Listen-CPU*).

2. Cost function directed migration can significantly extend application lifetime in sensor networks with topologies and task graphs two orders of magnitude larger than previous studies: comparing Figure 20 to both Figures 18 and 19, lifetime is increased in all cases studied.

3. Compared to experiments with only prefetching enabled, turning on dynamic fusion point migration yields only slightly lower latency and throughput in most cases we study, while extending lifetime and increasing delivered items per lifetime (Figures 20 and 19). The exception, *B/T-*-CPU*, is encountered when frequently migrating larger state across a lower bandwidth connection. Although application lifetime is still extended, average latency and throughput

may suffer, potentially leading to a drop in the total number of delivered items per lifetime. Suggested potential solutions to this specific problem would incorporate the latency cost of migration within the cost function being evaluated or in the determination of cost function evaluation frequency.

4. Although an optimal radio *Sleep* duty cycle is expected to improve application lifetime by not wasting energy in listen mode for idle nodes, it does not result in a significant change in lifetime in the presence of optimistic prefetching, except when using expensive Orinoco listen (Figure 19, **-Sleep-** vs **-Listen-**).

5. More intuitively, prefetching results in increased throughput compared to the baseline, while network lifetime with prefetching is lower than the baseline since more work is being done per unit time. (Compare Figure 19 to Figure 18.)

4.1.1 Summary of Initial MSSN Studies

Results confirm intuition that latency and throughput are very closely dependent upon the architectural configuration options. From this result, we can see benefits of multiple clock frequency and variable bandwidth support. Also, migration has to be done judiciously with a carefully chosen cost function. Indiscriminate use of migration may waste energy and reduce throughput. Even directing migration with a simple cost function, **MPV**, based only on available energy in a node enables application lifetime extension for large scale applications, confirming viability of migration middleware for SN management. The design space covered here is only part of what is possible. SN applications may exhibit bursty behavior, motivating further energy savings by exploring opportunities for coordinated, local CPU frequency scaling in further studies. MSSN is also an ideal platform for performing scalability analyses of the DFuse placement heuristic.

4.2 Scalability of DFuse Placement Heuristic

We have performed experiments to determine how well each DFuse cost function for directing fusion point migration scales with respect to network topology and application size. Scalability is key to utility in real, large scale SN deployments. We use MSSN, extended to include optimal cost "oracles" for each cost function, to analyze this scalability under simplifying API, MAC and routing layer assumptions and ignoring CPU and memory energy and delay costs as in our original DFuse evaluation in Chapter 2. Both the migration and optimistic prefetching features of MSSN are enabled in these experiments.

We simulate DFuse behavior under three cost functions (**MT2**, **MTP** and **MPV** from Chapter 2) upon networks as large as 1024 nodes and for different sizes of input application task graphs. For our scalability studies here, we extend MSSN to include several *oracles*, or algorithms that output an "optimal" mapping of the application to the network and associated cost given the cost function used by the simulated placement module. By running these oracles multiple times during a simulation, evaluation of how DFuse's local placement heuristic performs relative to a globally good algorithm becomes possible. We disable the simulator's extra modeling of CPU and Memory energy usage to focus on the transmission cost characteristics as DFuse is applied to large scale network topologies and applications. We perform two scalability analyses: a small application on varying scales of network topology, and varying scale applications on a large network topology.

Our hypothesis is that observed performance trends in terms of transmission cost performance relative to optimal will confirm trends observed in our earlier small scale iPAQ farm experiments. For larger applications, we also expect that the mappings traversed by the DFuse placement module result in costs close to optimal for each cost function, and that good scalability will be demonstrated by a fairly constant,

low ratio of DFuse cost to optimal cost as the application scale increases. Simulation experiments with larger networks and applications confirm our hypotheses of placement heuristic scalability, revealing interesting properties of the fusion point placement algorithm:

- 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 SN can further extend lifetimes when using migration middleware.

- For large topologies and 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 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.

The remainder of this case study is structured as follows. In section 4.2.1, we present DFuse fusion point migration scalability evaluations for a small application task graph as the number of nodes in the SN topology increases as a base case. In section 4.2.2, we present more generally applicable experiments with large applications and topologies to quantify scalability of the DFuse fusion point migration mechanism relative to our best feasibly computed "optimal" performance. We conclude with a brief summary of scalability results in section 4.2.3.

4.2.1 Single Fusion Point Scalability Study

To determine the performance with respect to optimal transmission cost of the placement module as the number of network nodes increases, we use a simple, single fusion point (image collage) application model with two camera sources and a single sink. We scale the network topology from a 4x4 to a 32x32 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 begins in the maintenance phase, with the collage fusion channel mapped to a random node not chosen for a camera or sink. We tailor the cameras each to produce 2KB images 5 times per simulated second, and tune the collage fusion function to simply concatenate the two input images, yielding 20KB 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 802.11b 11Mbps bandwidth, barring excessive collisions. For a 4x4 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 is optimally placed, is 60KB*hops/second, assuming no collisions or other latencies. Similarly, for a 32x32 grid, the initial optimal transmission cost at maximum possible throughput is 620KB*hops/second, for any of our random camera placements.

As in our real farm deployment, we enable simulated prefetching of input items for the collage fusion function, letting the collage immediately request the next inputs from the cameras before receiving a request for their fused output from the sink, essentially enabling pipelined processing. The sink simply repeatedly requests the next fused item from the collage. Processing at both the collage and sink are instantaneous (1 CPU cycle), making camera production the slowest processing step in the pipeline. Simulated migration of fusion channels is also enabled in all of these experiments, to enable evaluation of the placement module for this application workload for large scale sensor network topologies. The placement module reevaluates the placement of the collage fusion channel every simulated 10 minutes, including an MSSN extension to perform global analysis to find a minimum transmission cost mapping for comparison, given the current application throughput (fairly static) and network state (degrading as nodes run out of energy). We include calculation of "proximity to optimal transmission cost mapping" in this global analysis to indicate the number of hops the collage channel would need to migrate to reach the current optimal transmission cost mapping.

4.2.1.1 Single Fusion Point Scalability Results

We first examine lifetime trends across these cost functions for our single fusion point application as the number of network nodes is varied. We expect the previously observed trend that **MT2** and **MTP** outperform **MPV** to be applicable to larger SN topologies. We also expect that increasing the number of nodes will yield longer application lifetimes. Lifetime results for grid topologies ranging from 4x4 nodes to beyond 8x8 nodes, with 5 random trials per experiment are shown in Figure 21, revealing:

- In general, as more nodes are in the SN topology, the application lifetime increases, indicating that all of the cost functions studied here are able to leverage

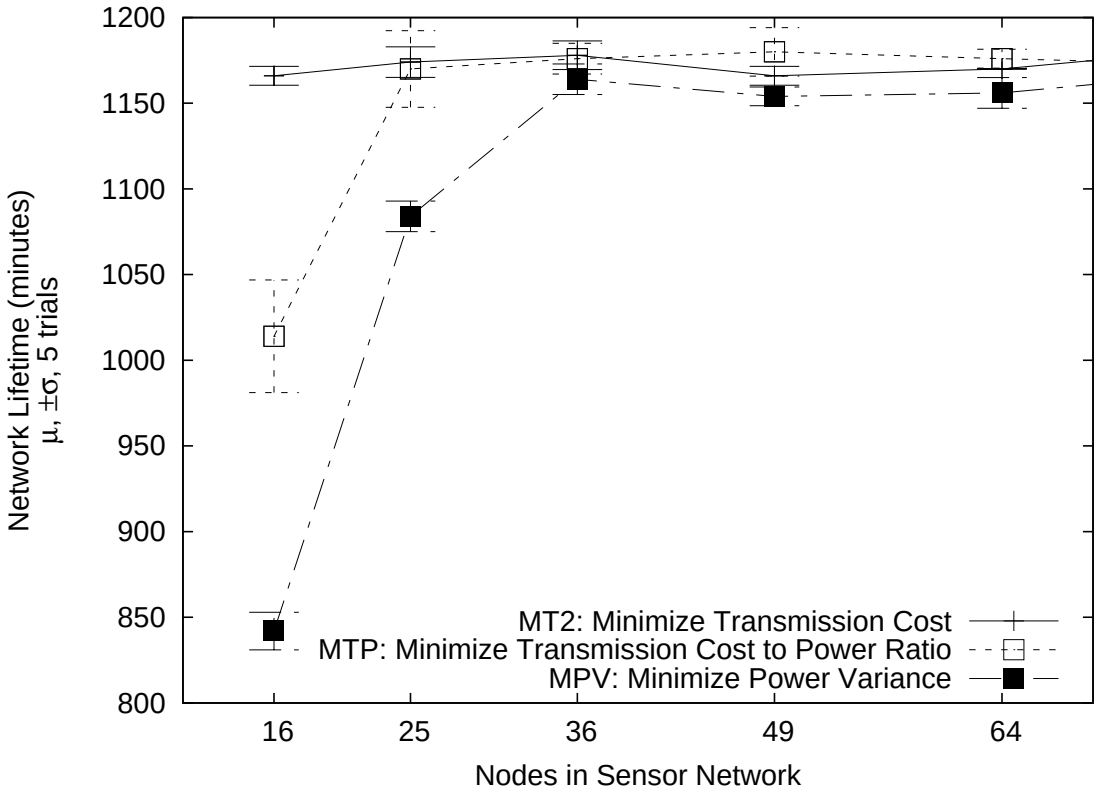

Figure 21: Lifetime for Single Fusion Point Application for Varying Network Sizes and Cost Functions

redundant energy resources to extend lifetime. Application lifetime is upper-bounded by the energy capacity of nodes critical to any possible overlay mapping. In our small application topology, the two sources and one sink do not migrate, only the collage migrates. Although we assume that sources and sinks have infinite energy, their neighbors do not. Consequently, for larger grid topologies, the neighbor nodes of these fixed endpoints will run out of energy first, causing partition. This explains why the three curves converge to a similar maximum lifetime of about 1150 minutes for grid topologies larger than 5x5 nodes.

- **MT2** achieves significantly longer lifetime than **MTP** for small topologies. **MTP** is a compromise between **MT2** and **MPV**, optimizing more for transmission cost minimization when node energies are similar, but optimizing more

for node energy equalization when node energies differ greatly. For small scale topologies, node energy levels diverge more rapidly, making **MTP** behave more like **MPV**, resulting in significantly lower lifetime than **MT2**.

- Observed lifetimes confirm the general trend observed in earlier experiments with a prototype DFuse deployment on an iPAQ farm (Chapter 2) where **MT2** achieves greater lifetime than **MTP**, and **MPV** achieves the least lifetime because its policy is uncorrelated with minimizing runtime energy costs.

Next, we analyze runtime transmission cost performance of **MT2**, **MPV** and **MTP** for our single fusion point application. Figures 22, 23 and 24 show, for each of the **MT2**, **MPV** and **MTP** cost functions, and for each of two topology scales: 4x4 and 32x32 nodes, the application lifetime and runtime transmission cost over time. These figures show transmission costs 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 except **MT2**, we use 3 trials with varying initial random selections of camera and collage placements. For **MT2**, performance across 3 trials is very similar for the same topology scale, so we show only one trial for each topology scale in Figure 22. As a baseline verification, the initial optimal transmission costs depicted in these figures agree with the predicted 60KB*hops/second for the 4x4 topology and 620KB*hops/second for the 32x32 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 22), actual placement corresponds closely to the optimal placements for the 4x4 topology, and rapidly converges to optimal mapping followed by slow degradation for the 32x32 topology. For **MPV** (Figure 23), 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

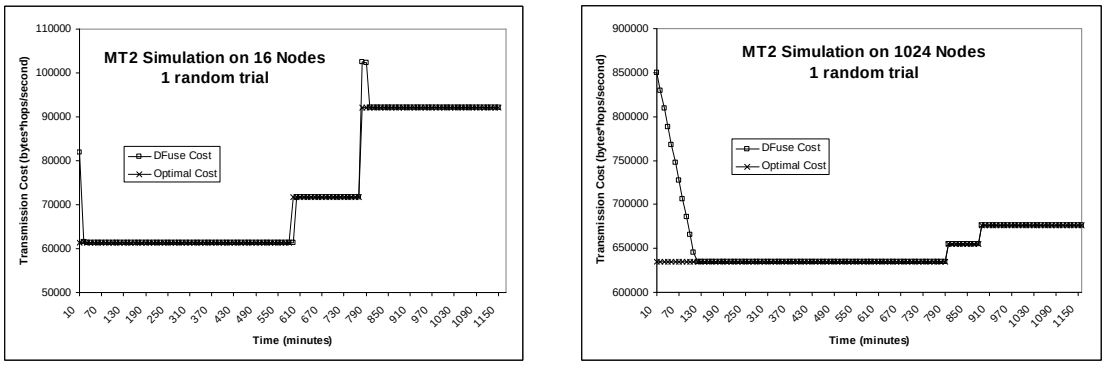

Figure 22: **MT2** for single fusion point application on 4x4 and 32x32 grids, showing DFuse and optimal transmission costs over time for 1 trial

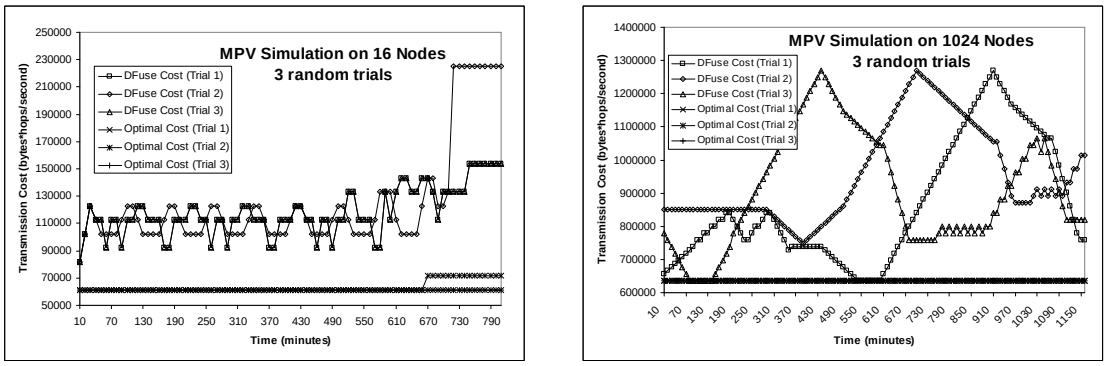

Figure 23: **MPV** for single fusion point application on 4x4 and 32x32 grids, showing DFuse and optimal transmission costs over time for 3 trials

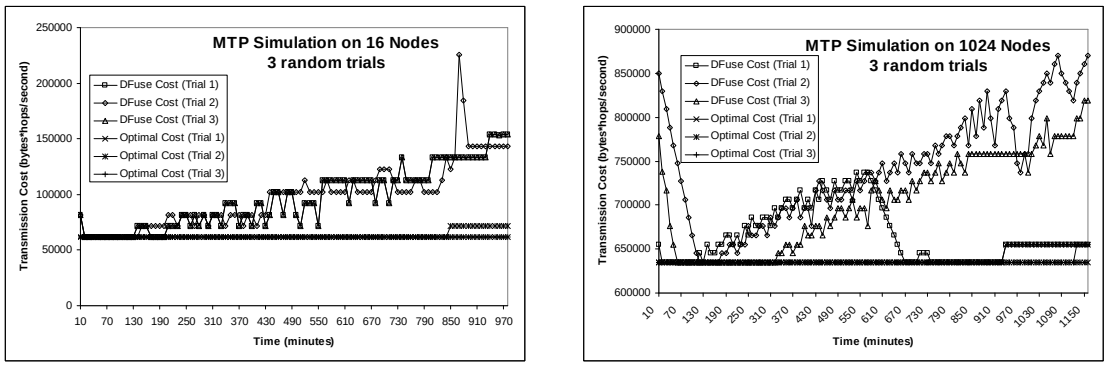

Figure 24: **MTP** for single fusion point application on 4x4 and 32x32 grids, showing DFuse and optimal transmission costs over time for 3 trials

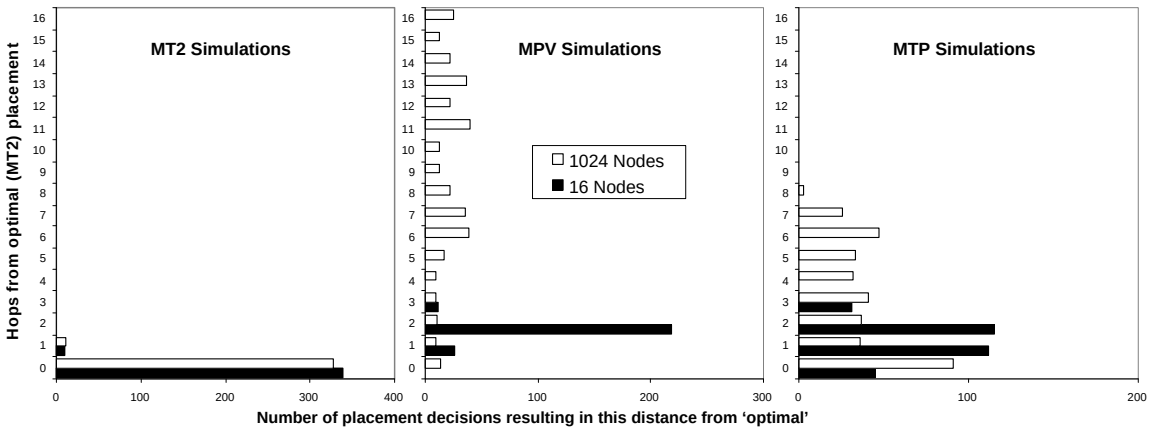

Figure 25: Proximity to optimal (transmission cost) mapping, shown as a histogram of number of hops an instant migration would need to take from current mapping, evaluated at every placement heuristic execution.

cost-agnostic behavior, hence the high variance in transmission cost over the life of the network and across trials. For **MTP** (Figure 24), 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 remains the same as the network scale increases from the iPAQ-farm results from Chapter 2 to the 16 and 1024 node simulations here: **MT2** achieves very close to optimal transmission cost, even in the 32x32 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, 32x32 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 mappings result from using **MPV**, there are enough redundant resources in the network to let the application survive until the sink's neighbors die. One possible solution to further extending the lifetime of this application would be to enable node mobility or migration of (at least) the sink and cameras, but such mobility and endpoint migration is out of scope for this study.

Figure 25 gives overall summaries of the proximity of the locally decided collage placement to the globally optimal transmission cost placement, for each of the 3 cost functions and 2 network scales studied. Zero hops dominate for **MT2** 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 section 4.2.2 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 the development and evaluation of a multihop placement heuristic.

4.2.2 General, Large Application Scalability Study

For the DFuse placement heuristic to be scalable as application and network size increases, it must perform well with respect to optimal scheduling. During a DFuse application's lifetime, the cost-function directed migration of fusion points through the network attempts to minimize the cost function locally at each fusion point. The global sum of the cost function at all fusion points gives a snapshot of the cost of

the current mapping for any cost function. By comparing this sum with an optimal global scheduler's cost at each snapshot, we evaluate the DFuse placement heuristic.

The difficulty with determining DFuse scalability here is that it is infeasible to perform "brute force" determination of optimal cost for realistic applications on large networks. Applications with f fusion points mappable to n sensor nodes have n^f possible mappings, making determination of optimality for realistic applications having on the order of more than 10 fusion points on more than 10 nodes impractical, especially as application scale increases. We approach the problem of determining optimality by using approximation algorithms with feasible complexity.

4.2.2.1 Steiner Tree Bounded Approximation Attempt

One possibility we attempt for obtaining a bounded approximation of optimality is to employ approximation algorithms to obtain a minimum Steiner tree [15]. The inputs to these algorithms are a graph with weighted edges $G=\{V,E\}$ and a set of distinguished vertices $S \subset \{V\}$. The algorithms output a connected subgraph of G in polynomial time with minimum total cost and containing all S vertices. Such algorithms are practical for obtaining minimum length wires needed to join points in circuits [60]. One possible simple bounded approximation algorithm we attempt for use in determining an "optimal" cost is a modified minimum spanning tree (**MST**) algorithm. It has been proven that this **MST** algorithm outputs an approximate solution for the Steiner problem in graphs with a bound of 2.0 [53].

However, use of this algorithm to provide a bounded approximation of optimal DFuse cost has significant faults. To use **MST**, we provide the entire network topology as the input graph G, and the set of nodes corresponding to sources and sinks as the subset of required vertices S. While the resulting graph will have an approximated minimum sum of all edge weights, **MST** does not account for the possibility of multiple flows along the same edge that are definitely possible in DFuse mappings.

Therefore, the resulting cost and its optimality bound cannot be trusted. Furthermore, while **MST** may have some utility for determining minimum transmission cost mappings for **MT2**, it is not directly applicable for use in evaluation of **MPV** and **MTP**. These faults apply generally to any of the Steiner tree approximation in graph algorithms. We therefore choose other heuristics for determination of "optimal" cost.

4.2.2.2 Simulated Annealing Approximation Approach

We extend MSSN to include three general-purpose *oracles* to determine "optimal" cost in the context of **MT2**, **MPV** and **MTP** cost functions. For small applications, we implement a general brute force oracle, **BF**, that evaluates all n^f possible mappings. This oracle is similar to the MSSN extension used for the single point scalability study above, except that **BF** supports any possible application, topology and cost function. We also employ two different tunings, **SA1** and **SA2** of a simulated annealing algorithm in the form of two approximating oracles. Instead of comparing performance to an optimal minimum transmission cost in all cases, we compare DFuse performance with the optimal cost with respect to the cost function used by DFuse in this section. This enables observation of **MTP** and **MPV** performance in terms of the metric they are each attempting to optimize.

Simulated annealing [26, 35] is a framework for calculating an approximation of a globally optimal solution to a multivariate optimization problem. It is based on a statistical mechanics analogy to the *annealing* process that occurs naturally in fluids as they slowly cool to form low energy state crystalline structures. Simulated annealing has been shown effective in providing good solutions for a variety of NP-Complete problems such as the Traveling Salesman Problem and circuit design, placement and wiring [26].

The simulated annealing process [26] begins with the initialization of the system to a random state. The process takes as input an objective function, or cost function, to

for each $c \in channels$
  do $c.host \leftarrow random\ node \in nodes$
$oldCost \leftarrow currentCost \leftarrow \sum_{c \in channels} costFunction(c)$
$T \leftarrow (-average(\Delta cost\ for\ each\ neighbor\ with\ higher\ cost)/ln(0.80))$
if $type =$ **SA1**
  then $iterationsPerTemperature \leftarrow |channels|^2$
  else $iterationsPerTemperature \leftarrow |channels| * |nodes|$//**SA2**
$temperatureCount \leftarrow 0$
repeat
 for $i \leftarrow 1$ **to** $iterationsPerTemperature$
  do $\begin{cases} candidate \leftarrow random\ neighbor\ of\ current\ state \\ \textbf{if } (\Delta cost(current\ to\ candidate) < 0) \\ \quad \textbf{then } current\ state \leftarrow candidate//Accept \\ \quad \textbf{else if } random\ number \in [0,1] < exp(-\Delta cost/T) \\ \quad \textbf{then } current\ state \leftarrow candidate//Accept \\ \quad \textbf{else } //Reject \end{cases}$
 $T \leftarrow T * 0.95;\ temperatureCount++;\ oldCost \leftarrow currentCost$
until ($type =$ **SA1 and**
  ($T < T_{cold}$ **or** $temperatureCount > |nodes|^2$ **or** $currentCost = oldCost$))
  or ($type =$ **SA2 and** $T < T_{cold}$)

Table 5: Algorithm $SA - Oracle(type, channels, nodes, costFunction(), T_{cold})$

evaluate a state. During the process, the systems' state evolves in response to random perturbations that are either accepted or rejected. A simplistic approach would be to accept only cost-decreasing perturbations, however it is likely that the resulting cost would be only a local minima. Simulated annealing involves the concept of a *temperature* T, in the same units as the cost function, that guides the probability of accepting a cost-increasing perturbation. At initially high temperatures, more of these increases are accepted, enabling escape from local minima. As the system cools, fewer increases are accepted and the state eventually freezes. The probability of accepting a cost-increasing perturbation is given by the Boltzmann factor $exp(-(\Delta cost)/T)$. At each temperature level, multiple perturbations are applied to sample the search space. Then the temperature is decreased according to a specified *annealing schedule*. Eventually, the annealing schedule specifies when the process terminates (when the system is deemed frozen). By choosing a slow, long cooling schedule and a large

number of perturbations evaluated at each temperature level, more of the state space is searched. Rapid cooling increases the possibility of arriving at a poor local minima, analogous to a metastable non-crystalline structure in natural annealing. Therefore, tuning the annealing schedule is necessary to achieve good results in practical running time.

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 not change as T decreases for more than 4 successive temperature levels, we define T_{cold} for each of the DFuse cost functions. Pseudocode for our **SA1** and **SA2** oracles is in Table 5.

Fusion Function	Description	Footprint (KB) Communication Input	Output	Persistent State
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

Table 6: Scalable Application Model: Fusion Functions (adapted from Table 2)

There exists the possibility that some states encountered during simulated annealing may be invalid in the context of DFuse. Specifically, the overlay network described by the mapping of fusion channels to nodes may be disconnected due to nodes' energy depletion. We address two issues with the algorithm presented by disconnected states: initial temperature determination and cost evaluation. If the initial random state is disconnected, we keep this initial state but use the current DFuse mapping's cost as input to the formula used to calculate the initial temperature. While the simulator is running and the oracle is executing, DFuse's mapping is guaranteed to be connected (and to have a finite cost). To enable the algorithm to escape local minima by way of temporarily traversing disconnected states, we let $disconnectedstatecost = initialcost * 3$. As there is a possibility that a disconnected state may be output from the algorithm, we flag these in our results.

For this large application scalability study, we construct a scalable application workload model based on workloads used in previous studies in Chapter 3. The task graph consists of a tree composed from subgraphs comprised of fusion channels that combine 2 inputs into 1 output stream and fusion channels that transform 1 input into 1 output stream. We only consider the time and energy used to transmit inputs, outputs and channel buffer and state migration when determining resource consumption. Table 6 (adapted from Table 2) shows these data sizes for each function. For example, the *Select* function takes two 56KB inputs and outputs 56KB each

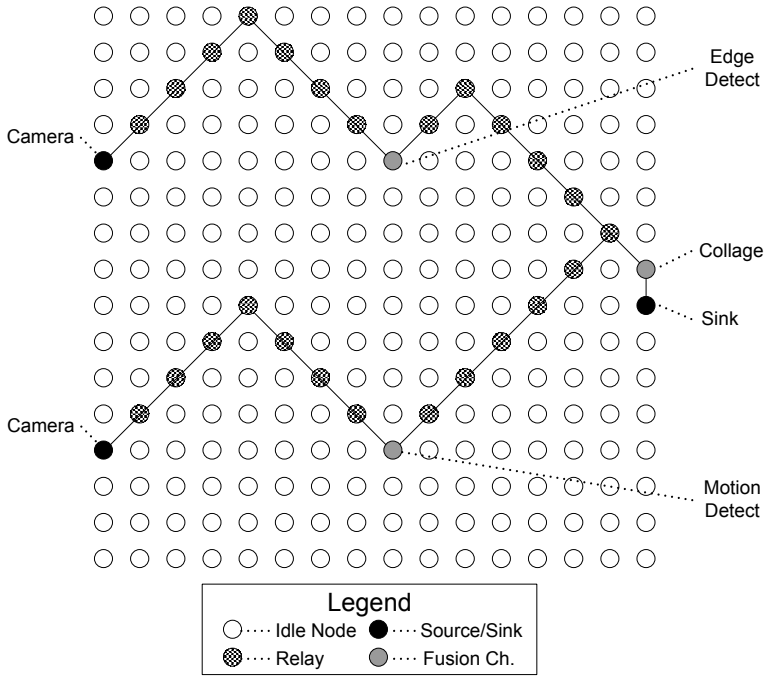

Figure 26: Scalable Application Model: Sample 3 Fusion Point Application on 16x16 Grid

iteration, and the *MotionDetect* channel migrates 94KB in addition to any buffer state. We configure the sensor network to be a 256 node 16x16 grid with rectilinear and diagonal nearest-neighbor connectivity and initially map the application onto it in a form much like a tree. The number of fusion points is determined by the number of camera sources used, thereby easily scaling the application. Figure 26 presents a sample 2 camera, 3 fusion point application's initial mapping (on a much smaller grid here). Note that there are many equivalent-length shortest hop count paths possible between two nodes with our rectilinear and diagonally connected grid topology, explaining the "crooked" paths in Figure 26. Larger application scales are formed by combining subgraphs like these together into a larger tree in a fashion similar to our previous workloads in Chapter 3.

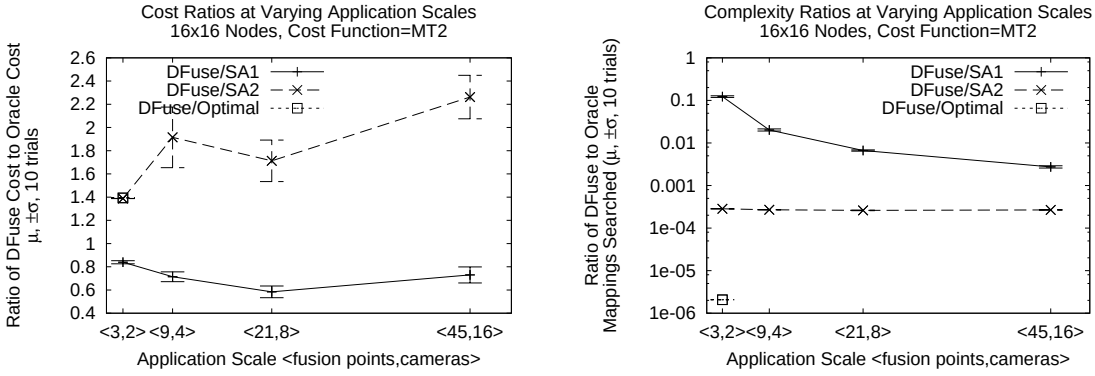

Figure 27: Large Application Scalability Results For **MT2**

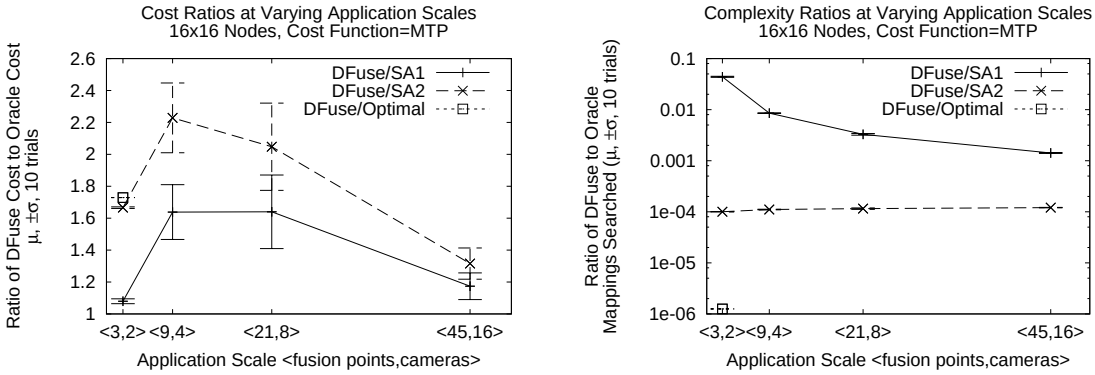

Figure 28: Large Application Scalability Results For **MTP**

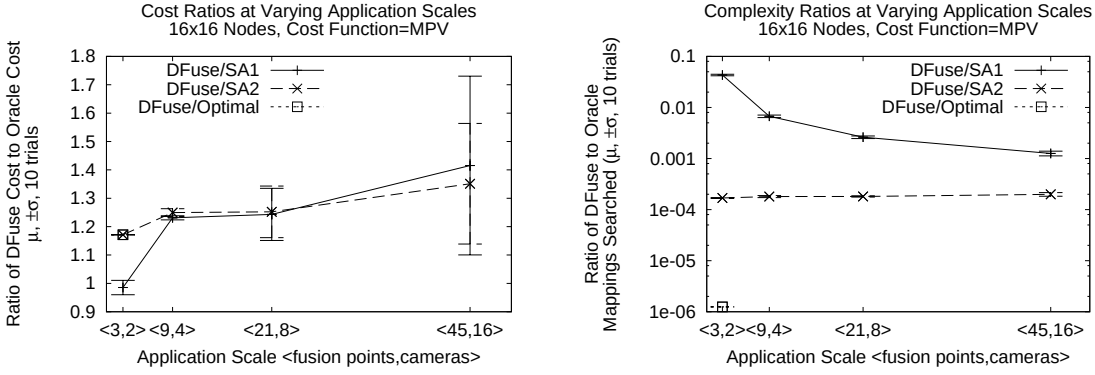

Figure 29: Large Application Scalability Results For **MPV**

4.2.2.3 Large Application Scalability Results

Figures 27, 28 and 29 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 perform 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 +/- one standard deviation across trials. For example, Figure 28 shows that DFuse achieves an average cost 2.2 times higher than **SA2** for **MTP** with a 9 camera, 4 fusion point application, but only searches about 1/10,000 of the mappings that **SA2** searches. It is computationally infeasible to run **BF** for large application scales, but we include it for the 3 fusion point, 2 camera configuration as an optimal baseline.

For **MT2**, Figure 27 shows that DFuse *outperforms* **SA1** at all application scales studied. This result is likely due to the relatively small portion of the state space that contains globally minimum transmission cost mappings. **SA1** searches f^2 mappings per temperature, and can terminate temperature reductions much sooner than **SA2**, resulting in searching a very small portion of the search space (about 1/100,000 for the smallest application scale studied). **SA2** performs optimally at this small application scale. For **MT2**, DFuse performs about the same relative to **SA1** and **SA2** as scale increases. 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 good as DFuse at the cost of much larger runtime complexity.

For **MTP**, Figure 28 shows that DFuse approaches both **SA1** and **SA2** and the standard deviation decreases as scale increases. Inspection of simulation traces also indicates that DFuse performs about the same amount of searching and **SA2** performs about twice the amount of searching for **MTP** vs **MT2**. While it is infeasible to

perform a brute force search at larger scales, it is quite interesting to see that DFuse approaches our best feasibly computed optimal global oracle at larger scales for **MTP**.

For **MPV**, Figure 29 shows that **SA1** and **SA2** perform similarly as scale increases (similar means, similar increasing standard deviations). The existence of similar means between **SA1** and **SA2** confirms 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 application scale increases, 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 application scale increases because **SA1** and **SA2** perform about 1.1 to 1.3 times as good as DFuse on average across all scales studied.

4.2.3 Placement Heuristic Scalability Conclusion

Overall, these large scale application scalability experiments show that DFuse performs similarly with respect to our best feasibly computed optimal global oracle as application scale increases, demonstrating good scalability. For **MTP**, DFuse performance actually improves as application scale increases. Our small application scalability results with MSSN confirm that performance rankings between **MT2**, **MPV** and **MTP** observed in Chapter 2 on our prototype DFuse implementation using an iPAQ farm remain the same as the number of SN nodes increases. Our small application results also 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 SN can further extend lifetimes when using migration.

4.3 Predictive CPU Scaling Heuristic for Future SN

Our evaluations of DFuse include the assumptions that the SN is homogeneous, and the device capabilities remain constant for an application lifetime. As SN devices become more computationally capable and SN applications perform greater amounts of computation to process high bit-rate data, there emerges a significant increase in energy usage for computation relative to communication. For example, expensive computations such as face detection and recognition can now be done on sensor nodes. For such computations, our iPAQ-based microbenchmarks and power models from Chapter 3 indicate that about 100ms of iPAQ processing is necessary on a data size of 56KB. Single hop communication to fetch inputs would cost roughly 106 mJ using Orinoco 11Mbps, while performing one execution of the FD/FR fusion function would cost roughly 31 mJ on a 206MHz SA-1100 package. If the data streams are compressed, then the proportional amount of energy used for processing increases. There is a significant opportunity for reducing energy consumption by reducing processing costs. In this case study, we explore dynamic *CPU scaling* as a potential mechanism for improving processing efficiency while maintaining required application performance.

SN will exhibit dynamism due to network interactions such as dynamic overlay adaptation by fusion point migration and network layer induced latencies due to lossy channels. SN applications will also exhibit application-specific dynamism. For example, to save energy and increase lifetime, a campus surveillance SN fusion application may perform minimal, infrequent anomaly "detection" operations, in a slow *periodic* mode. Once a situation needing attention is detected, information gathering and processing activities will increase to *full speed* mode, to analyze the situation in more depth and to achieve improved "liveness" of data (equivalent to end-to-end latency

from capture at sources to delivery to sinks) until the situation is resolved. We describe an application model exhibiting dynamic *periodic* and *full speed* surveillance modes for our heuristic evaluation later in Section 4.3.2.2.

Since power consumption of modern processors decreases as processor frequency and voltage decrease (see MSSN's power model), fusion point processing should ideally be done no faster than I/O to keep the steps in pipelined processing equally long. If the application can tolerate increased end-to-end latency, processing speed can be dropped further.

Therefore, a middleware like DFuse for supporting dynamic fusion applications should also include the ability to dynamically *scale* the CPU speed of individual SN nodes to save processing energy. We anticipate current technology trends enabling voltage and frequency scaling [44, 12, 40, 48] to be available in future SN devices.

There are many related approaches in distributed systems research for performing energy-adaptive communication management. Examples include:

- Energy can be saved in mobile communications by queuing data for future delivery in an application-driven manner [29].

- Due to significant delays when transitioning between sleep and active modes in some radio packages such as 802.11b, cooperation between both CPU and communication schedulers may be necessary to yield energy savings when radio doze time maximization is critical [42]. While bursty communications can reduce radio mode transitions and maximize radio doze time for 802.11b, it is not clear that mode transition delays will be common across all future SN radio packages.

We are not aware of any CPU scaling technologies incorporated into middleware for supporting fusion applications in SN.

We therefore design, implement in our MSSN simulator, and evaluate a novel dynamic, local *predictive CPU scaling* mechanism for fusion points that uses processing and I/O behavior history to adapt CPU speed. Given performance results for our tunable scaling heuristic, we then show how information about SN application performance requirements and tolerances to degradation can be used to tune the heuristic. Future extensions to the heuristic may include the ability to improve doze time under 802.11b by performing bursty packet scheduling.

The remainder of this case study is structured as follows. We present the design and implementation of our heuristic in section 4.3.1, followed by its implementation, characterization of a more recent CPU power model than SA-1100, and dynamic workload modeling in section 4.3.2. This case study concludes with an evaluation of our scaling heuristic in section 4.3.3.

4.3.1 Heuristic Design

Our predictive CPU scaling heuristic uses recent processing and I/O intensity to predict future intensity. The primary insight we use in our heuristic is that time taken to perform an iteration of fusion function processing should be approximately equal to the time taken to fetch the inputs and cache the fused output. By keeping these times similar to each other, stages in the distributed fusion application pipeline should have similar durations, reducing buffering, decreasing end-to-end latency, and reducing time spent by processors in idle mode (when they could have worked slower and more cheaply).

Our heuristic runs in a distributed fashion, using only information gathered from the local fusion point and a single tuning factor supplied by the application. Inputs include:

- *oldIOtime* is the summation of I/O time taken to gather inputs and store outputs into finite local buffers for the previous processing iteration. Any application or

network induced latencies (for example, slow sinks or cameras, or lossy propagation channels) are included in this input. If the finite local "prefused" buffer was full when the previous iteration completed fusion, then the time spent blocking on writing to the buffer is included. Basically, this input encompasses all time spent during the previous processing iteration except for actual fusion processing, with the exception of CPU scheduling latencies that occur when multiple fusion points mapped to the same node timeshare a SN node's CPU.

- *pendingOutputs* is the number of items currently in the local prefused buffer awaiting retrieval by downstream consumers. This figure is used to put additional "back-pressure" on a fusion point before the prefused buffer reaches capacity. Large values for *pendingOutputs* imply large end-to-end latency, because large numbers of intermediate items are being buffered in the fusion pipeline. Also, larger amounts of buffered outputs increases the latency and transmission cost for fusion point migration. Therefore, our heuristic biases against large values of *pendingOutputs*. As *pendingOutputs* grows, the heuristic slows processing so that I/O can drain the buffer.

- *oldFtime* is the summation of actual processing time taken during the previous iteration, including any CPU scheduling latency due to timesharing the CPU.

- *oldSpeed* is the frequency setting for the CPU for the previous iteration.

- *lastCycles* is an input derived from multiplying *oldSpeed* by *oldFtime*, and represents the actual number of processor cycles used to perform the previous iteration's fusion.

- *FCTR*, short for Fusion to Communication Time Ratio, is the QoS tuning factor provided by the application, discussed below.

All of these input parameters, except *FCTR*, are the actual values obtained during

simulator execution. Supplied by the application, *FCTR* is the only control parameter for this heuristic. Our heuristic outputs *newSpeed*, the setting of the node's CPU for the current fusion point processing iteration. The heuristic assumes that the current iteration's communication time and number of processor cycles taken during fusion will be constant at *oldIOtime* and *lastCycles*, respectively. *FCTR* describes the application's desired ratio of fusion processing time to I/O processing time (adjusted by current buffer usage). All of these parameters are used to calculate *newSpeed*:

$$\begin{aligned} FCTR &= \frac{desired\ Ftime}{oldIOTime * (1 + pendingOutputs)} \\ &= \frac{\frac{lastCycles}{newSpeed}}{oldIOTime * (1 + pendingOutputs)} \\ &= \frac{\frac{oldSpeed * oldFtime}{newSpeed}}{oldIOTime * (1 + pendingOutputs)} \end{aligned}$$

$$newSpeed = \frac{oldSpeed * oldFtime}{FCTR * oldIOtime * (1 + pendingOutputs)}$$

To prevent extreme processor speed thrashing, we adjust *newSpeed* to be the average of *oldSpeed* and *newSpeed*, and then bound *newSpeed* within the minimum and maximum SN device CPU speeds. On the first iteration of this heuristic, we simply gather inputs for the next iteration and set *newSpeed* to be an application-defined initial speed. Later iterations dynamically adapt the CPU speed based on runtime behavior as shown above.

4.3.2 Heuristic Implementation

We implement our predictive CPU scaling heuristic as an extension to MSSN, our simulator of DFuse middleware presented earlier in Chapter 3. In the following, we present details of the CPU power models and our model of a dynamic application workload used in this case study.

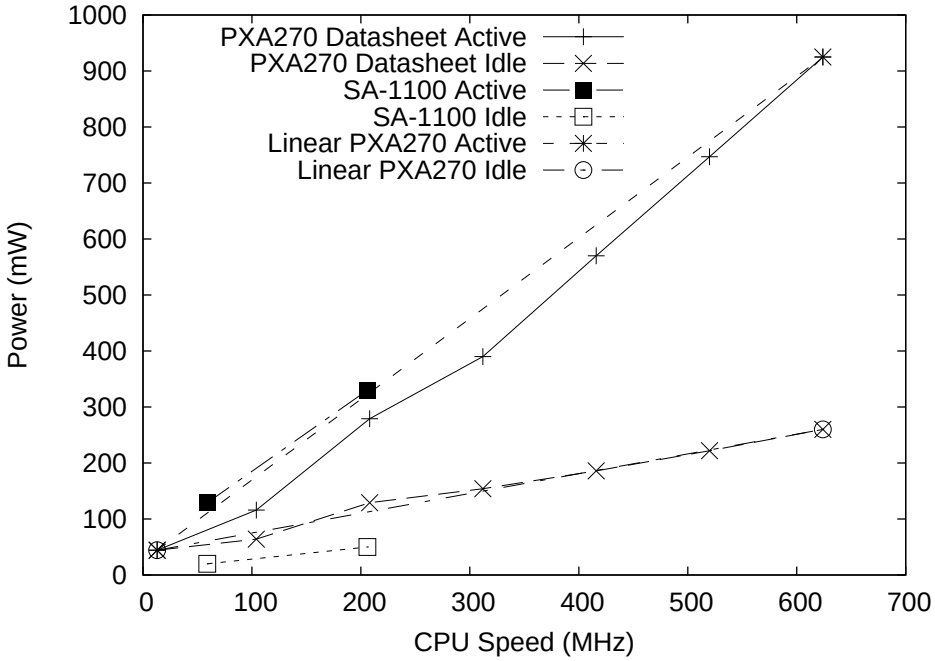

Figure 30: Power models used for CPU scaling studies based on Intel PXA270 and SA-1100

4.3.2.1 Power Models

For CPU scaling to save energy and have a hope for increasing SN lifetime and productivity, the processors used in SN devices must be scalable, and they must consume less energy for the same amount of processing as their speed decreases. We also build a new power model based on Intel's more recent PXA270 xScale datasheet [22], and check if it has a hope for saving energy. Figure 30 shows the active and idle energy profiles for both our SA-1100 model from Chapter 3 and our new PXA270 model. We use a linear approximation of the PXA270 datasheet's values in our model. We assume that changing the CPU speed of a node is free, and that the available CPU speeds are continuous between a minimum and maximum speed. Both of these assumptions simplify our evaluation. If changing speeds is costly, this could be encoded as a threshold and by biasing output *newSpeed* closer to the *oldSpeed*. Discrete CPU speed options could be selected as a further filter on our output *newSpeed*.

From our PXA270 power model, we observe that $\sim$ 3.4mJ are used in active mode

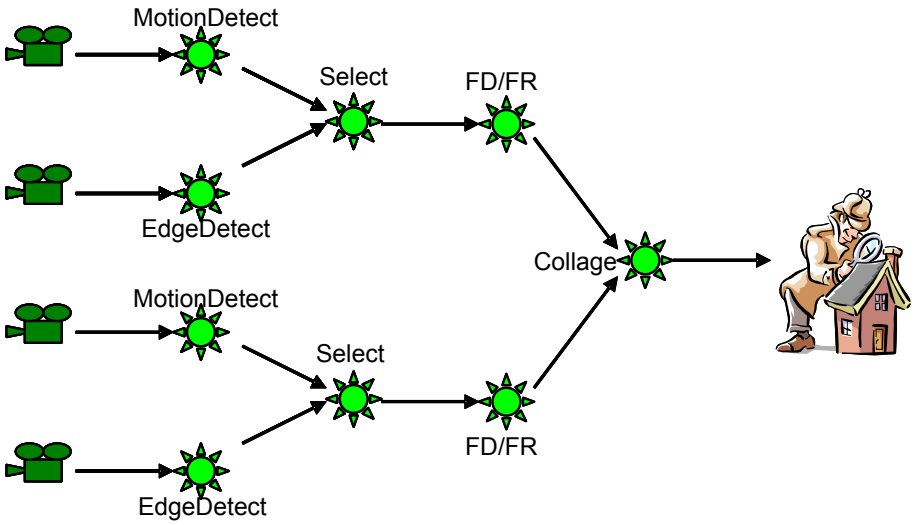

Figure 31: Task graph for our dynamic surveillance application workload

at 13MHz to process 1 million cycles. For this same amount of cycles, only $\sim$ 1.48mJ are used in active mode at 624MHz to process 1 million cycles. These observations indicate that for CPU-intensive applications, speed should counter-intuitively be *increased* to save energy per unit of work. However, if we assume that the units of work arrive at a fixed rate such that the processor would be kept busy at 13MHz, then significant idle energy costs (3.3mJ) would be incurred between processing bursts at 624MHz if the CPU speed were maximized. Therefore, to save energy, PXA270 may need to be slowed for communication-bound applications and sped up for CPU-intensive applications. When compared with our SA-1100 model, our PXA270 power model shows the greatest promise in terms of energy-saving and latency-improving tradeoffs based on dynamic application behavior, so we use PXA270 for the remainder of our CPU-scaling experiments.

4.3.2.2 Dynamic Application Workload

To study our CPU scaling heuristic under realistic application dynamism, we build a new dynamic application workload model. Figure 31 shows the task graph we use,

consisting of 4 camera sources, 7 fusion points and a sink. We introduce three new forms of dynamism (beyond fusion point migration) in this workload to approach behavior close to a realistic surveillance system:

- The application operates in two modes, alternating every 5 minutes of simulated time. In *periodic* mode, the cameras produce images every 30 seconds (or even slower if the remainder of the task graph cannot keep the pace), while in *full speed* mode, the cameras produce images as fast as they are consumed by the rest of the task graph. Inspection of simulation traces for *FCTR* of 1.0 confirms that *periodic* mode yields an actual periodicity of about 29 seconds per frame and *full speed* mode yields an actual periodicity of about 17 seconds per frame. We keep a greedy sink similar to our previous experiments. By alternating modes, we model surveillance applications being directed by out-of-band feedback to change source capture rates in response to presence or lack of events of interest. The rate of capture will be a large factor in the rate of network energy consumption, so applications would prefer to operate slower when reasonable.

- To model this two-mode application better, a FD/FR channel performs no processing on input items captured when the system is in *periodic* mode, and just passes them directly to the collage fusion point. This enables the surveillance application to cheaply detect the beginning of events of interest and then intensively process data during events of interest.

- We vary the number of cycles a FD/FR fusion function takes when processing events of interest in *full speed* mode. We vary FD/FR cycles between 50 and 100% of the benchmarked 1959M cycles, changing no more than 10% between consecutive iterations. This models the variation in processing complexity depending on dynamic properties of the data. The remainder of the fusion functions have static properties from benchmarks in Chapter 3.

We use **MTP** to guide fusion point placement of this task graph at runtime on a random topology of 800 in-network nodes connected by our Bluetooth, ideal Listen cost radio model presented earlier. We choose **MTP** because we have shown that it does a good job of extending lifetime and scales well. We hope to show improvement beyond baseline performance with **MTP** by using our CPU scaling heuristic.

4.3.3 Heuristic Evaluation

We measure delivered application performance along several dimensions. We compare application lifetime while using CPU scaling for our dynamic application workload to application lifetime when the CPU speed is fixed at either the minimum or maximum speed for the entire simulation. We measure end-to-end latency (from capture by camera to delivery to sink) averaged over all items, over only *periodic* mode items, and over only *full speed* mode items. Similarly, we measure productivity per lifetime (items received at sink) overall and separately for each of the two application modes. Finally, to observe dynamic CPU scaling over a single trial, we record the chosen *newSpeed* output of the heuristic for one of the two FD/FR fusion points. We annotate this speed trace with the application mode the item is being processed in.

We hypothesize that *newSpeed* will increase during *full speed* application phases and decrease when I/O (including fetch time from rate-throttling sources) dominates during *periodic* phases. We expect end-to-end latency to increase and lifetime to increase when using CPU scaling compared to when the CPU speed is fixed at the maximum. There are too many variables for us to hypothesize productivity performance under CPU scaling, but an increase would be good. Finally, we hypothesize that tuning the heuristic's *FCTR* will enable trading off latency degradation for lifetime improvements and vice versa, and that a ratio below 1 will generally exhibit better latency and lower lifetime, as the heuristic will try to spend less time processing than performing I/O in this case.

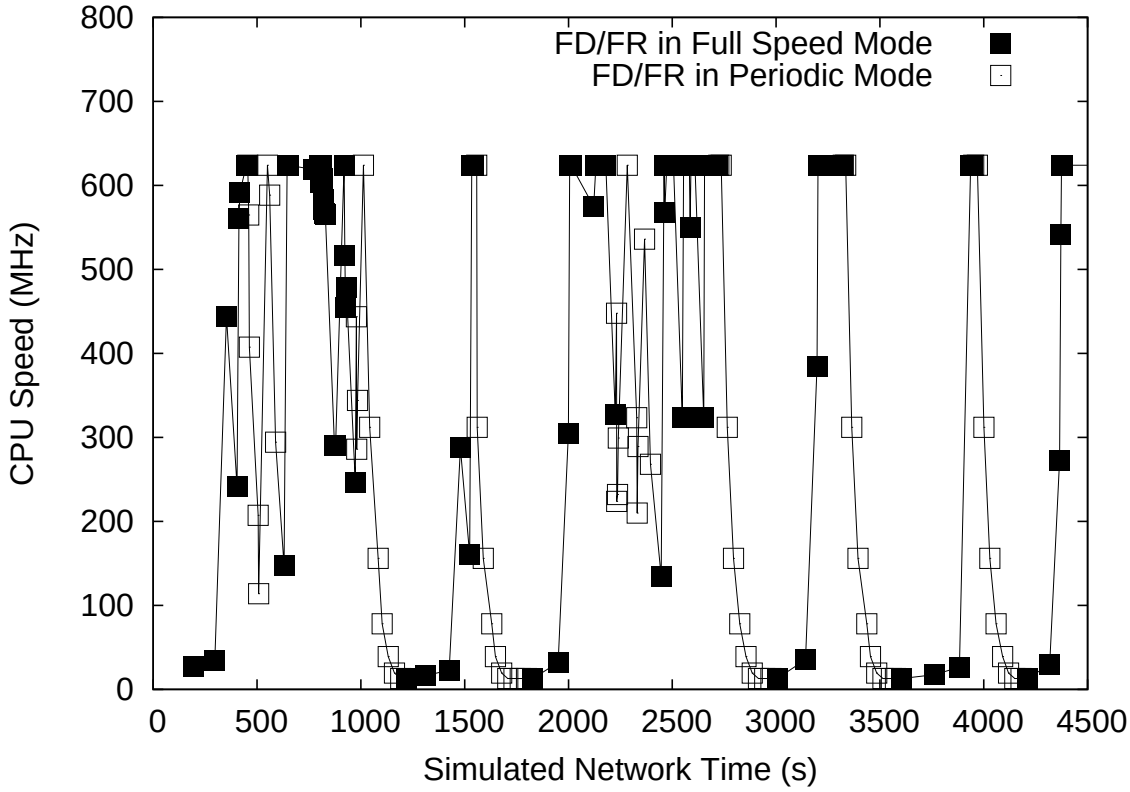

Figure 32: CPU scaling behavior for 1 trial, showing chosen CPU speed at one FD/FR fusion point over time

We present a sample speed trace first, followed by observations of how changing *FCTR* to tune the heuristic impacts delivered performance. We conclude our CPU scaling evaluation with a demonstration of how *FCTR* can be used to tune the SN to deliver required application performance when there is tolerance for some performance degradation.

4.3.3.1 Dynamic CPU Scaling Results

Figure 32 is a sample speed trace, showing the output *newSpeed* of the predictive CPU scaling heuristic for one of the two FD/FR fusion points. For this single trial, we use the PXA270 power model and a *FCTR* of 1.0. Only the first 4500 seconds of lifetime are shown, enabling visual separation of the *periodic* and *full speed* phases. We observe that *full speed* mode items drive the CPU speed higher and *periodic* mode

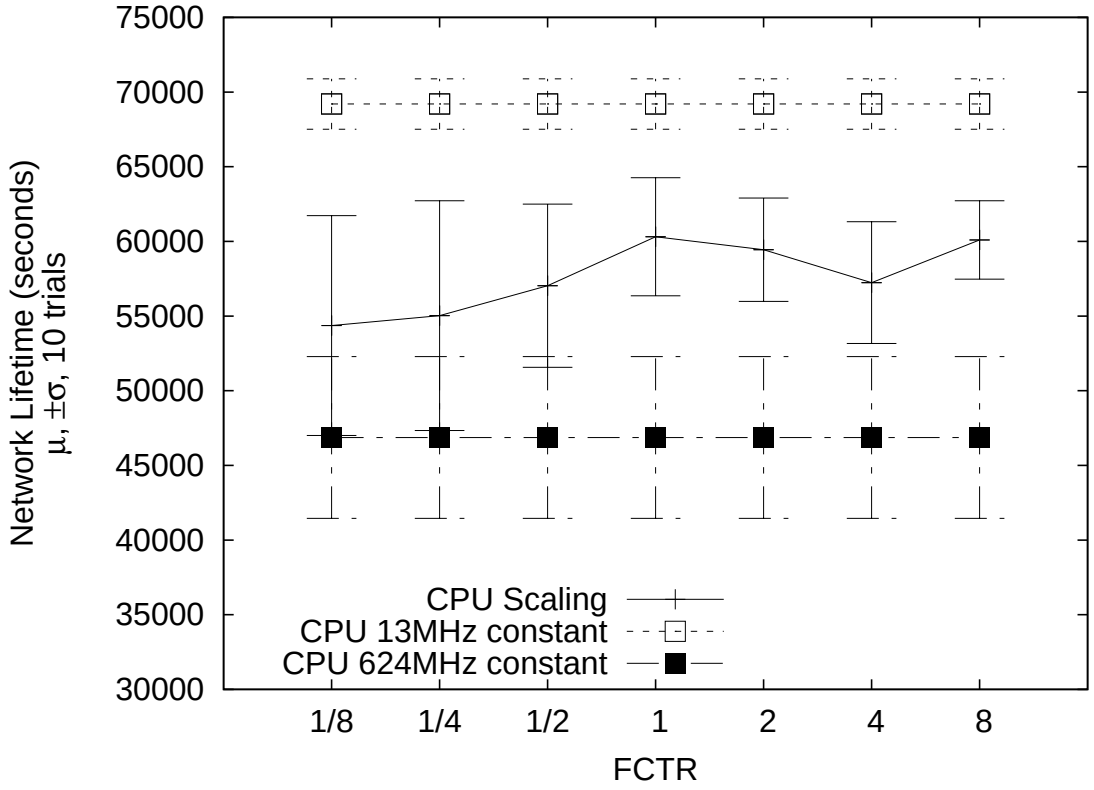

Figure 33: Effect of *FCTR* on network lifetime relative to lifetime at min and max CPU speeds

items drive the CPU speed lower, confirming our hypothesis. The noise apparent in this speed trace is due to pipeline stalls caused by weak migration of fusion points, oscillations in the "back-pressure" encoded as a multiplier of I/O time in our heuristic, and by the general dynamism occurring in our FD/FR model and in the network.

Figure 33 shows the average (and +/- 1 standard deviation) lifetime delivered across 10 trials for each of 7 *FCTR*. We confirm our hypothesis that using CPU scaling can significantly improve lifetime compared to using the maximum CPU speed. As *FCTR* goes from $\frac{1}{8}$ to 1, lifetime increases because the heuristic is more evenly matching fusion processing time with I/O time by reducing the CPU speed to increase fusion time. As *FCTR* exceeds 1, lifetime performance deviates from this trend.

Figures 34, 35 and 36 show the average (and +/- 1 standard deviation) end-to-end

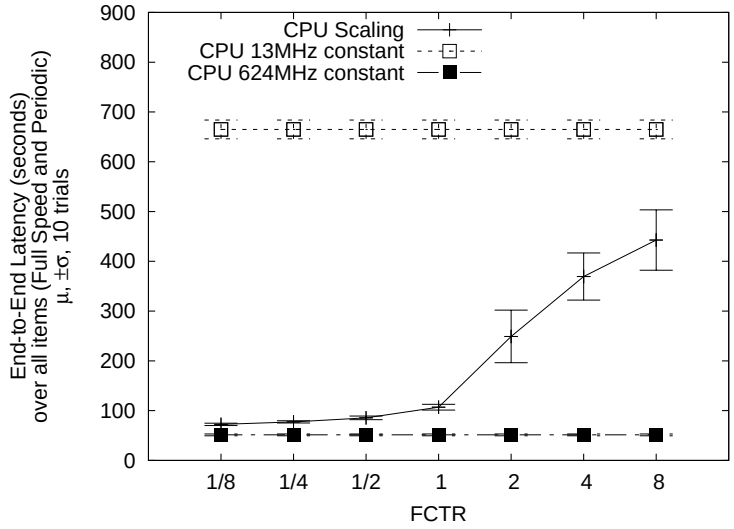

Figure 34: Effect of *FCTR* on end-to-end latency over both *periodic* and *full speed* items

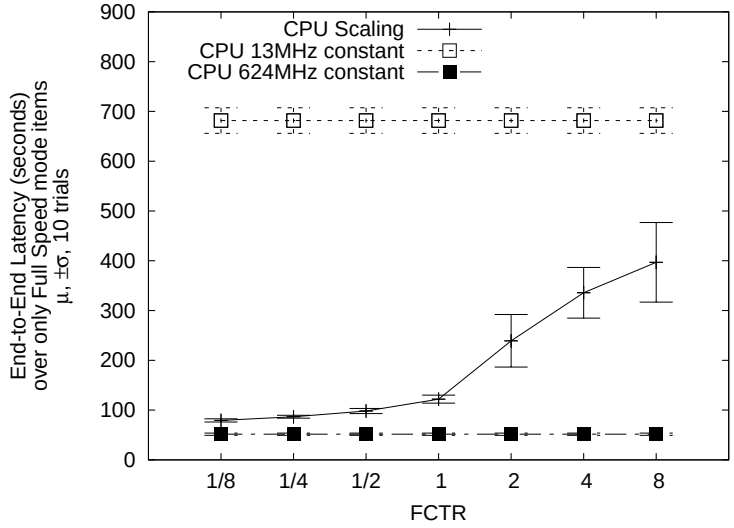

Figure 35: Effect of *FCTR* on end-to-end latency over only *full speed* items

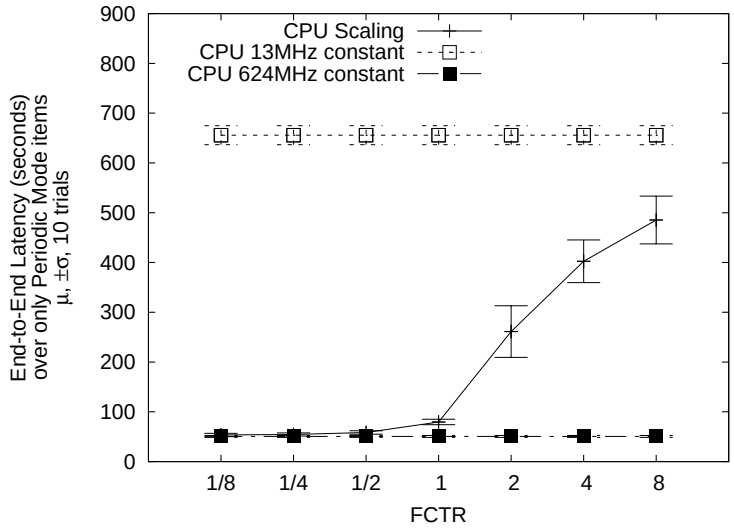

Figure 36: Effect of *FCTR* on end-to-end latency over only *periodic* items

latency delivered across the 10 trials for each of 7 *FCTR*. We confirm our hypothesis that using the maximum possible CPU speed will deliver lowest latency. As *FCTR* increases to 1.0, latency slowly increases because the heuristic will tend to output lower CPU speeds to increase the fusion time to reach a higher fusion time to communication time ratio. Given the noise from application dynamism that the heuristic cannot predict, the heuristic may sometimes output a CPU speed that is too low. These errors have increasing latency penalty as *FCTR* increases and the application becomes less communication-bound. When *FCTR* exceeds 1.0, the application becomes increasingly computation-bound and the latency rapidly increases. Our predictive CPU scaling heuristic achieves the minimum possible latency when *FCTR* is below $\frac{1}{4}$ for items processed in *periodic* mode (Figure 36). These items incur little processing, making it difficult to become computation-bound when *FCTR* is below 1.

Figures 37, 38 and 39 show the average (and +/- 1 standard deviation) productivity across 10 trials and 7 *FCTR*. Productivity, in this context, is a measure of the number of items delivered to the sink during the application's lifetime. We count the number of *periodic* mode items separately from *full speed* mode items delivered to

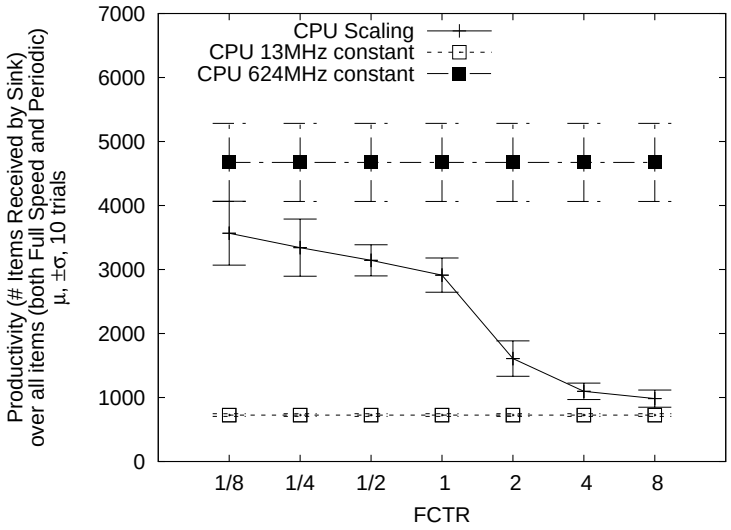

Figure 37: Effect of *FCTR* on productivity over both *periodic* and *full speed* items

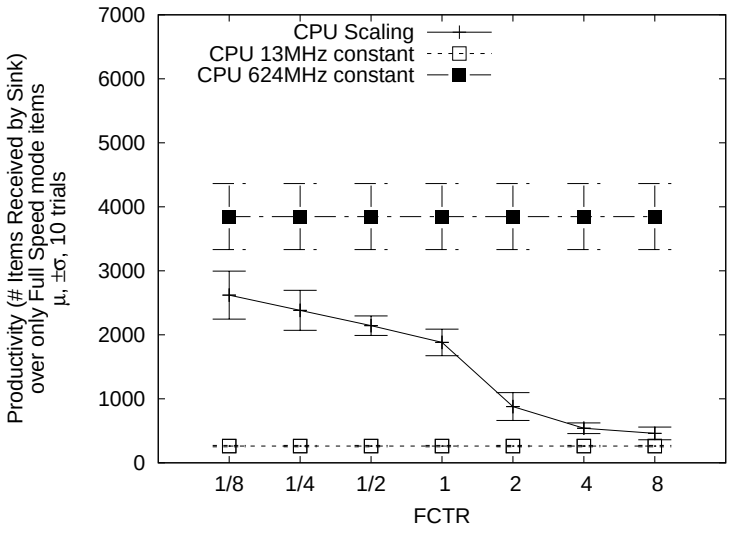

Figure 38: Effect of *FCTR* on productivity over only *full speed* items

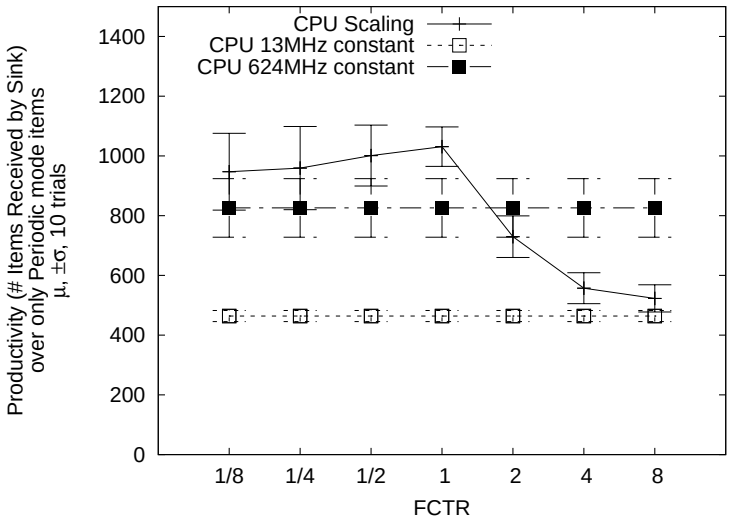

Figure 39: Effect of *FCTR* on productivity over only *periodic* items

the sink, and report their counts and their combined sum in these figures. We also indicate the productivity of the network when the processor speed is consntantly at the maximum and constantly at the minimum in these figures.

Figures 37 and 38 show that using the maximum CPU speed achieves the highest productivity. However, when considering only the *periodic* mode items, Figure 39 indicates that it is possible to achieve greater productivity than when using either a constant maximum or constant minimum speed. For this application's *periodic* mode and the power models being used, it is more energy efficient to actively compute at a slow CPU speed than to compute rapidly and then idle for the remainder of a period. The predictive CPU scaling heuristic is therefore able to decrease energy consumption and increase productivity of *periodic* mode workloads. The ability to increase productivity by performing dynamic CPU scaling is a very positive result. However, by intuition, productivity increase with CPU scaling is only possible for workloads that require only low active CPU duty cycles and CPU power models that allow for energy savings. Energy saving CPU power models must have lower cost when processing speed and idle time are decreased for performing the same amount of active processing in the same unit of time. Our PXA270 power model exhibits this

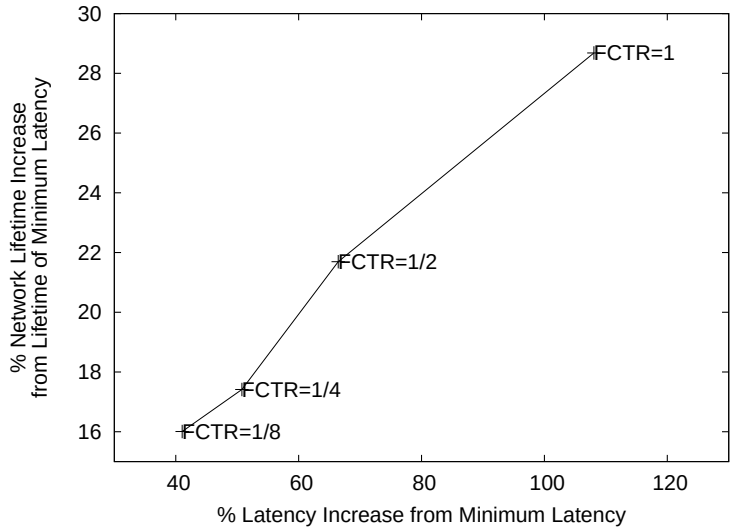

Figure 40: Percentage lifetime increase and associated *FCTR* for corresponding tolerances to end-to-end latency degradation

characteristic, and the *periodic* workload allows for significant processor idle time. Productivity does not exceed that of processing with maximum CPU speed for the *full speed* workload with CPU scaling because the workload is greedy. If the radio and memory power models cost no energy for idling, then we would expect that CPU scaling would achieve at least the productivity observed at maximum CPU speed for PXA270. However, we account for radio and memory idle costs in this study. Due to *newSpeed* not always being the maximum CPU speed and due to radio and memory idling energy overheads that increase as processing slows, productivity for the *full speed* workload is lower with CPU scaling than with using the maximum CPU speed.

4.3.3.2 Performance Tradeoff Analysis and Case Study Conclusion

Results from our predictive CPU scaling evaluation indicate that it is impossible to improve latency beyond that yielded by simply using the maximum possible CPU speed, for the workload and power models we consider. However, many applications may not require the best possible latency and may value other performance improvement, such as increased lifetime and/or productivity. For example, given a tolerance

in requirement for latency, results show that it is possible to select *FCTR* values to feed our predictive CPU scaler that will likely improve lifetime (and improve productivity for workloads similar to *periodic* mode). Figure 40 shows the relationship between tolerance to latency degradation from minimum end-to-end latency and the corresponding expectation of lifetime increase. For example, if the application allows up to 70% worse end-to-end latency than optimal, then a *FCTR* of 1/2 could be used for this workload to achieve a 22% longer lifetime than the lifetime afforded by the minimum latency configuration. Similar tradeoff analyses can be done for other fusion application workloads and SN device models using MSSN, enabling performance characterization and tuning.

CHAPTER V

CONCLUSION

Future SN will be called on to support high bit-rate stream-based fusion applications. To succeed, SN will need to be constructed to achieve application performance requirements while minimizing energy usage to increase application lifetime. This thesis investigates several novel middleware mechanisms for improving application lifetime while achieving required latency and throughput, in the context of a variety of SN topologies and scales, models of potential fusion applications and device radio and CPU capabilities.

First, we discussed *DFuse*, our novel middleware for performing energy aware stream processing in SN. DFuse includes a detailed API for supporting fusion application processing in SN. We showed that the DFuse placement heuristic can significantly extend application lifetime with a reasonable-overhead prototype implementation on a 12 node iPAQ SN.

Next, we discussed *MSSN*, a major contribution of this thesis. The core of this novel middleware simulator is a set of detailed, extensible models of future SN devices, middleware and application workloads. MSSN's non-trivial design and implementation realizes a scalable, believable middleware simulator that enables such low level SN node attributes as CPU speed scaling to be evaluated in the context of application level performance. The remaining contributions of this thesis use MSSN to perform such evaluations.

In our first case study, we used MSSN to quantify the performance of large scale fusion applications to show how MSSN can be used to expose performance tradeoffs in future SN.

We then presented a scalability evaluation of the DFuse placement heuristic using MSSN as our second case study, showing that our distributed *role-assignment* heuristic performs well with respect to our best feasibly calculated optimal performance as application and network topology scales increase.

In a final case study, we propose a new CPU scaling middleware mechanism for future SN to help optimize energy efficiency under dynamic application workloads. We analyze this *predictive CPU scaling* heuristic and show how it can be tuned through MSSN-based analysis to trade off tolerable latency degradation in favor of lifetime extension and vice versa.

Through our use of MSSN to evaluate potential middleware mechanisms, application workloads and SN capabilities, we demonstrate MSSN's utility for exposing tradeoffs fundamental to successful SN construction. By adapting and extending workload, device and middleware mechanism models in MSSN, the simulator can be used in future work to quantify performance for a larger scope of SN applications.

CHAPTER VI

FUTURE WORK

Several enticing directions for further research have become apparent during the course of the work presented in this thesis. This chapter gives an overview of potential future work.

- This thesis does not concern itself with proposing or evaluating mechanisms for *radio scaling*, e.g. dynamically adapting the bandwidth, range and signal strength of SN radios. This route of research may provide additional benefits to applications in terms of latency, throughput and lifetime. It should be possible to extend MSSN to characterize the potential benefits of such mechanisms. Proper evaluation of *radio scaling* mechanisms may need significant extensions to MSSN to include appropriate models of reliable transport, network and MAC layers to account for dynamically changing collision domains. There is currently much conflicting research on whether multi-hop communication saves energy vs "shouting louder", and varying application domains may have different trends here.

- We constrain our study to supporting a single fusion application with a static task graph. While we use dynamic capture rate and processing complexity models (*periodic* vs *full speed*) to help evaluate our predictive CPU scaling mechanism, we do not consider several important sources of additional dynamism. Techniques to extend network lifetime by letting sources and sinks migrate, or by physically moving SN nodes to obtain a cheaper mapping may provide significant benefit. Also, leveraging existing SN node mobility patterns may provide

better application performance by connecting sparsely populated SN and by reducing radio power requirements.

- We currently use shortest path "relay chains" in MSSN to connect fusion points. Investigating power aware routing techniques may provide further insight into how to best connect the overlay with respect to the current cost function, rather than blindly minimizing hops along the chains.

- DFuse assumes that the addresses of the data sources are known at query time. This assumption may be a limiting assumption for many applications where data sources are unknown at query time. Future work may explore different ways of extending DFuse to handle such data-centric queries. One possible approach is to have an interest-dissemination phase before the naive tree building phase of the role assignment algorithm. During this phase, the interest set of individual nodes (for specific data) is disseminated as is done in directed diffusion [21]. When the exploratory source packets reach the sink (root node of the application task graph), the source addresses are extracted and recorded for later use in other phases of the role assignment algorithm.

- During our evaluation with MSSN of the DFuse placement heuristic's lifetime extension performance for large applications in Chapter 3, we examine only **MPV**. While we observe significant lifetime extension when migration with **MPV** is enabled, future work could test our hypothesis that **MT2** and **MTP** provide even greater lifetime extension. We observe this trend in small application studies and expect it would apply to larger scale SN and applications.

- A more ambitious direction for future work would be to interface MSSN with our initial iPAQ DFuse implementation. This combination could enable seamless reuse of application workload models between simulations in MSSN and actual SN running DFuse. Similar frameworks in different domains have demonstrated

the benefits of seamlessly switching between simulations and actual deployments. For example, Em∗ [16] lets developers test their "mote"-based applications in pure simulation, in emulation mode combining simulated processing with physical wireless channels, and in actual SN. Enabling similar capabilities with MSSN and DFuse would minimally require development of interface layers between application code and MSSN, and between application code and DFuse. We expect that such a development effort would pay off by enabling rapid prototyping of fusion applications within MSSN followed by field testing with DFuse. Such work is in line with the motivation of this thesis: future fusion applications are needed now, so we need to create development and evaluation tools to speed their arrival.

REFERENCES

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

[2] Agarwal, Y. and Gupta, R. K., "On Demand Paging Using Bluetooth Radios on 802.11 Based Networks," Tech. Rep. 03-22, Center for Embedded Computer Systems, UC Irvine, UC San Diego, July 2003.

[3] Bajaj, L., Takai, M., Ahuja, R., Tang, K., Bagrodia, R., and Gerla, M., "GloMoSim: a scalable network simulation environment." UCLA Computer Science Department Technical Report 990027, May 1999.

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

[5] Boulis, A., Han, C.-C., and Srivastava, M. B., "Design and implementation of a framework for programmable and efficient sensor networks," in *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, (San Francisco, CA), May 2003.

[6] Burd, T. D. and Brodersen, R. W., "Processor design for portable systems," *Journal of VLSI Signal Processing*, vol. 13, pp. 203–222, August 1996.

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

[8] Chang, J.-H. and Tassiulas, L., "Energy conserving routing in wireless adhoc networks," in *IEEE INFOCOM*, pp. 22–31, 2000.

[9] CMU Monarch Project, "Wireless and mobility extensions to ns-2." Available November 2004 at *http://www.monarch.cs.cmu.edu/cmu-ns.html*, 1999.

[10] Crossbow Technology Inc., "MICA2 datasheet." Available November 2004 at *http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-06_A_MICA2.pdf*.

[11] Crossbow Technology Inc., "Stargate gateway (SPB400) datasheet." Available November 2004 at *http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0049-01_C_Stargate.pdf*.

[12] Fan, X., Ellis, C., and Lebeck, A., "Memory controller policies for dram power management," in *Proceedings of the 2001 international symposium on Low power electronics and design*, (Huntington Beach, California, United States), pp. 129–134, ACM Press, 2001.

[13] Ganesan, D., Krishnamachari, B., Woo, A., Culler, D., Estrin, D., and Wicker, S., "Complex behavior at scale: An experimental study of low-power wireless sensor networks." Technical Report CSD-TR 02-0013, UCLA, February 2002. Available November 2004 at *http://www.cs.umass.edu/˜dganesan/PAPERS/empirical.pdf*.

[14] Gibbons, P. B., Karp, B., Ke, Y., Nath, S., and Seshan, S., "IrisNet: an architecture for a worldwide sensor web," *IEEE Pervasive Computing*, vol. 2, no. 4, 2003.

[15] Gilbert, E. N. and Pollak, H. O., "Steiner minimal trees," *SIAM Journal on Applied Mathematics*, vol. 16, pp. 1–29, January 1968.

[16] Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., and Estrin, D., "Em*: a software environment for developing and deploying wireless sensor networks," in *Proceedings of USENIX 04*, (Los Angeles, California, USA), 2004.

[17] Gui, C. and Mohapatra, P., "Sensor networks: Power conservation and quality of surveillance in target tracking sensor networks," in *Proceedings of the 10th annual international conference on Mobile computing and networking*, September 2004.

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

[19] Heinzelman, W. B., Murphy, A. L., Carvalho, H. S., and Perillo, M. A., "Middleware to support sensor network applications," *IEEE Network Mag.*, vol. 18, no. 1, pp. 6–14, 2004.

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

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

[22] Intel Corp., "Intel PXA270 Processor Electrical, Mechanical, and Thermal Specification." Available June 2005 at *http://www.intel.com/design/pca/applicationsprocessors/datashts/28000205.pdf*.

[23] Intel Corp., "New computing frontiers - the wireless vineyard." Available November 2004 at *http://www.intel.com/labs/features/rs01031.htm.*

[24] Intel Corp., "Intel StrongARM SA-1100 Developer's Manual," *Document no. 278088-04*, 1999.

[25] Johnson, D. B. and Maltz, D. A., "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing* (Imielinski and Korth, eds.), vol. 353, Kluwer Academic Publishers, 1996.

[26] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, May 1983.

[27] Kling, R. M., "Intel mote: An enhanced sensor network node," in *Proceedings of the International Workshop on Advanced Sensors, Structural Health Monitoring, and Smart Structures*, 2003.

[28] Kozat, U. C., Koutsopoulos, I., and Tassiulas, L., "A framework for cross-layer design of energy-efficient communication with qos provisioning in multi-hop wireless networks," in *Proceedings of IEEE/Infocom*, 2004.

[29] Kravets, R. and Krishnan, P., "Application-driven power management for mobile communication," *Wireless Networks*, vol. 6, no. 4, pp. 263–277, 2000.

[30] Kremer, U., Hicks, J., and Rehg, J. M., "A compilation framework for power and energy management on mobile computers," in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[31] Kumar, R., Wolenetz, M., Agarwalla, B., Shin, J., Hutto, P., Paul, A., and Ramachandran, U., "DFuse: a framework for distributed data fusion," in *Proceedings of the first international conference on embedded networked sensor systems*, (Los Angeles, California, USA), pp. 114–125, ACM Press, 2003.

[32] Levis, P., Lee, N., Welsh, M., and Culler, D., "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[33] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W., "TAG: a tiny agregation service for ad-hoc sensor networks," in *Operating System Design and Implementation (OSDI)*, (Boston, MA, USA), Dec 2002.

[34] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., and Anderson, J., "Wireless sensor networks for habitat monitoring," in *ACM International Workshop on Wireless Sensor Networks and Applications*, 2002. Also Intel Research, IRB-TR-02-006, June 2002.

[35] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, M., and Teller, E., "Equations of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, 1953.

[36] Modahl, M., Bagrak, I., Wolenetz, M., Jain, R., and Ramachandran, U., "EventWeb: Distributed media correlation, analysis and distribution framework," in *Proceedings of 10th IEEE Workshop on the Future Trends of Distributed Computing Systems (FTDCS-04)*, (Suzhou, China), May 2004.

[37] Montanaro, J., Witek, R. T., Anne, K., Black, A. J., Cooper, E. M., Dobberpuhl, D. W., Donahue, P. M., Eno, J., Hoeppner, G. W., Kruckemyer, D., Lee, T. H., Lin, P. C. M., Madden, L., Murray, D., Pearce, M. H., Santhanam, S., Snyder, K. J., Stephany, R., and Thierauf, S. C., "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *Digital Tech. J.*, vol. 9, no. 1, pp. 49–62, 1997.

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

[39] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R., "Agile application-aware adaptation for mobility," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, (Saint Malo, France), pp. 276–287, ACM Press, 1997.

[40] Pillai, P. and Shin, K. G., "Real-time dynamic voltage scaling for low-power embedded operating systems," in *ACM Symposium on Operating Systems Principles*, pp. 89–102, 2001.

[41] Poellabauer, C., Abbasi, H., and Schwan, K., "Cooperative run-time management of adaptive applications and distributed resources," in *Proceedings of the tenth ACM international conference on Multimedia*, (Juan-les-Pins, France), pp. 402–411, ACM Press, 2002.

[42] Poellabauer, C. and Schwan, K., "Energy-aware traffic shaping for wireless real-time applications," in *Proceedings of the 10th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2004.

[43] Polastre, J., Szewczyk, R., Sharp, C., and Culler, D., "The mote revolution: Low power wireless sensor network devices," in *Proceedings of Hot Chips 16: A Symposium on High Performance Chips*, 2004. Presentation available November 2004 at *http://webs.cs.berkeley.edu/papers/hotchips-2004-motes.ppt*.

[44] Pouwelse, J., Langendoen, K., and Sips, H., "Dynamic voltage scaling on a low-power microprocessor," in *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, (Rome, Italy), pp. 251–259, July 2001.

[45] Proxim Corp., "ORiNOCO PC Card Specification." 2003 available at *http://www.hyperlinktech.com/web/orinoco/orinoco_pc_card_spec.html*, similar spec available November 2004 at *http://www.proxim.com/learn/library/datasheets/11bpccard.pdf*.

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

[47] Saurabh Ganeriwal, Vlassios Tsiatsis, C. S., "NESLsim: a parsec based simulation platform for sensor networks." Available June 2005 at *http://www.allowbreak ee.ucla.allowbreak .edu/ saurabh/NESLsim*, 2002.

[48] Semeraro, G., Albonesi, D. H., Dropsho, S. G., Magklis, G., Dwarkadas, S., and Scott, M. L., "Dynamic frequency and voltage control for a multiple clock domain microarchitecture," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, (Istanbul, Turkey), pp. 356–367, IEEE Computer Society Press, 2002.

[49] Semiconductor, O., "ML7050LA Specification." Available November 2004 at *http://www.oki.com/semi/english/t-blue.htm*, June 2001.

[50] Simon, G., Volgyesi, P., Maroti, M., and Ledeczi, A., "Simulation-based optimization of communication protocols for large-scale wireless sensor networks," in *Proceedings of IEEE Aerospace Conference*, (Nashville, Tennessee, USA), March 2003.

[51] Singh, S. and Raghavendra, C. S., "PAMAS: power aware multi-access protocol with signalling for ad hoc networks," *ACM SIGCOMM Computer Communication Review*, vol. 28, pp. 5–26, July 1998.

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

[53] Takahashi, H. and Matsuyama, A., "An approximate solution for the steiner problem in graphs," *Math. Japonica*, vol. 24, no. 6, pp. 573–577, 1980.

[54] Viredaz, M. A. and Wallach, D. A., "Power evaluation of a handheld computer," *IEEE Micro*, 2003.

[55] Wang, M., "Nokia sees strong demand for smartphones and camera phones in 2005." Available November 2004 at *http://www.digitimes.com/news/a20041104A6035.html*.

[56] Warneke, B., Last, M., Liebowitz, B., and Pister, K. S. J., "Smart dust: Communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, 2001.

[57] Wolenetz, M., Kumar, R., Shin, J., and Ramachandran, U., "Middleware Guidelines for Future Sensor Networks," in *Proceedings of the First Workshop on Broadband Advanced Sensor Networks*, (San Jose, California, USA), October 2004.

[58] Ye, W., Heidemann, J., and Estrin, D., "An Energy-Efficient MAC protocol for Wireless Sensor Networks," in *Proceedings of INFOCOM 2002*, (New York, New York), June 2002.

[59] Zayas, E., "Attacking the process migration bottleneck," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pp. 13–24, ACM Press, 1987.

[60] Zhou, H., "Efficient steiner tree construction based on spanning graphs," in *International Symposium on Physical Design*, pp. 152–157, 2003.