*Technical Report*

UCAM-CL-TR-722
ISSN 1476-2986

Number 722

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

# Energy-efficient sentient computing

Mbou Eyole-Monono

July 2008

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*http://www.cl.cam.ac.uk/*

This technical report is based on a dissertation submitted January 2008 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

*http://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

# Abstract

In a bid to improve the interaction between computers and humans, it is becoming necessary to make increasingly larger deployments of sensor networks. These clusters of small electronic devices can be embedded in our surroundings and can detect and react to physical changes. They will make computers more proactive in general by gathering and interpreting useful information about the physical environment through a combination of measurements. Increasing the performance of these devices will mean more intelligence can be embedded within the sensor network. However, most conventional ways of increasing performance often come with the burden of increased power dissipation which is not an option for energy-constrained sensor networks. This thesis proposes, develops and tests a design methodology for performing greater amounts of processing within a sensor network while satisfying the requirement for low energy consumption. The crux of the thesis is that there is a great deal of concurrency present in sensor networks which when combined with a tightly-coupled group of small, fast, energy-conscious processors can result in a significantly more efficient network. The construction of a multiprocessor system aimed at sensor networks is described in detail. It is shown that a routine critical to sensor networks can be sped up with the addition of a small set of primitives. The need for a very fast inter-processor communication mechanism is highlighted, and the hardware scheduler developed as part of this effort forms the cornerstone of the new sentient computing framework by facilitating thread operations and minimising the time required for context-switching. The experimental results also show that end-to-end latency can be reduced in a flexible way through multiprocessing.

This thesis is dedicated to my loving parents.

# Acknowledgements

I am most grateful to my supervisor Andy Hopper, for his continual inspiration and guidance.

I would like to thank Robert Harle for his encouragement, and patience, especially his astounding ability to cope with my random musings. Thank you for providing me with the impetus I needed.

Many thanks to Andrew Rice and Alastair Beresford for helping me define my objectives with clarity. Thanks to Brian Jones, David Cottingham and Jonathan Davies for sharing their opinions on wide-ranging issues.

I am indebted to Simon Moore, Robert Mullins, and Andrew West for providing me with the hardware development tools and some very sound advice.

Thanks to Andrew Rose and Simon Ford for helping me develop the right set of technical skills while at ARM.

I would also like to thank my brother Lloney for his motivational support.

Special thanks to Hayley Whitfield for her love and concern.

This research was funded by ARM, Trinity College and the Cambridge Commonwealth Trust. I remain forever grateful to them for their immense generosity.

# Contents


**1 Introduction** **10**

1.1 Research Statement . . . . . 11
1.2 Outline of the Thesis . . . . . 12
1.3 Contributions . . . . . 12
1.4 Publications . . . . . 13
1.5 Workflow . . . . . 13

**2 Motivation and Background** **15**

2.1 Sensor Network Organisation . . . . . 18
2.1.1 Towards Heterogeneous Sensor Networks . . . . . 23
2.2 Sensor Network Scalability . . . . . 24
2.2.1 The Role of the DAPH . . . . . 27
2.3 Sensor Processor Power Reduction Options . . . . . 28
2.3.1 Low Power Electronic Devices . . . . . 30
2.3.2 Circuit and System Optimisations . . . . . 31
2.3.3 Efficient Processing within Sensor Networks . . . . . 33

**3 The SpotCore Architecture** **36**

3.1 Handling Loops . . . . . 39
3.2 Instruction Set Design . . . . . 41
3.3 Results . . . . . 45
3.4 Summary . . . . . 47

**4 TopDog Scheduling** **48**

4.1 The Role of the Scheduler in Explicit Parallelism . . . . . 48
4.2 The Case for Balanced Loads . . . . . 51
4.3 Selecting a Concurrency Model . . . . . 51
4.4 Critical Analysis of Concurrency Models . . . . . 53
4.5 Multiprocessing Primitives . . . . . 57
4.6 QoS-Aware Scheduler . . . . . 62

4.7 Scheduling in Energy-Constrained Environments . . . 64
4.8 Towards Estimating and Maximising QoS . . . 65
4.9 The Implementation of TopDog . . . 70
4.10 Results and Discussion . . . 74
4.11 Communicating Events Between Processors . . . 74
4.12 Towards a Robust Multiprocessor Architecture . . . 76
4.12.1 Memory Protection . . . 76
4.12.2 Deadlock Capture . . . 77
4.12.3 Priority Inversion . . . 78
4.13 Summary . . . 80

**5 A Case Study in Scalable Concurrent Software 81**
5.1 Task-Partitioning . . . 82
5.2 Results . . . 86
5.3 Summary . . . 86

**6 A DAPH System 89**
6.1 Reducing Latency in Sensor Networks . . . 89
6.2 Network Latency . . . 92
6.3 Latency Reduction in DANTE . . . 94
6.4 The DANTE Architecture . . . 99
6.5 Time-Critical Tasks in DANTE . . . 102
6.6 DANTE System Results . . . 111
6.7 Summary . . . 117

**7 Conclusions and Future Directions 118**
7.1 Future Work . . . 120

**A DANTE Location System Design 122**

**References 138**

# List of Figures


1.1 Thesis workflow . . . . . 14

2.1 Flow of information in a typical sensor network . . . . . 16
2.2 Conventional sensor network with motes . . . . . 28
2.3 Sensor network with processing hubs . . . . . 29

3.1 SpotCore pipeline . . . . . 38
3.2 SpotCore speeds up loops . . . . . 41
3.3 Comparing SpotCore instructions with other RISC instructions . . . . . 42
3.4 MOVE instruction . . . . . 44
3.5 SpotCore instruction set formats . . . . . 44
3.6 SpotCore processor layout . . . . . 46
3.7 IIR filter results . . . . . 46

4.1 Scheduling order affects speedup . . . . . 52
4.2 Task graph and associated SpotCore code . . . . . 54
4.3 Multiprocessor comprising SpotCores and a TopDog scheduler . . . . . 59
4.4 Thread state diagram . . . . . 63
4.5 TopDog implementation . . . . . 71
4.6 Accelerating critical procedures within the TopDog . . . . . 73
4.7 Memory structures in deadlock detection hardware . . . . . 79

5.1 FFT dataflow . . . . . 82
5.2 Chart showing execution times for different cores . . . . . 87
5.3 Chart showing energy utilisation for different cores . . . . . 87

6.1 Improving latency with more processing flow paths . . . . . 93
6.2 Variation in average latency with number of SpotCores . . . . . 96
6.3 Variation in average latency with uniprocessor operating frequency . . . . 97
6.4 Variation in average throughput with number of SpotCores . . . . . 98
6.5 Variation in average throughput with uniprocessor operating frequency . . 99

6.6 The DANTE location system . . . 103
6.7 Core components of tag-signals-system architecture . . . 104
6.8 Tag collision: spatial view . . . 106
6.9 Tag collision: temporal view . . . 107
6.10 The DANTE protocol . . . 109
6.11 FMAN implementation . . . 110
6.12 DANTE tag . . . 111
6.13 DANTE tag circuit components . . . 112
6.14 DANTE tag and Floor Manager . . . 113
6.15 Wire matrix laid underneath the carpet . . . 114
6.16 DANTE experimental carpet surface . . . 115
6.17 Testing channel overlap . . . 116
6.18 Channel signal strength data . . . 116
6.19 DANTE Floor Viewer . . . 117

A.1 Capacitance parameters . . . 125
A.2 Variation of capacitance with tag distance . . . 126
A.3 Circuit formed by tag and wire embedded in the carpet . . . 127

# Chapter 1

# Introduction

It is becoming increasingly desirable to amplify the utility of the nearly-ubiquitous yet ordinary personal computer by making it more perceptive to our real world environment. Human interaction with the current generation of computing devices can rarely be described as engaging and is sometimes barely tolerable. The root of the problem was first identified in [124] and the solution which has been heavily researched since its publication usually involves a large number of fairly intelligent computing units which are all interconnected and capable of sensing changes in the ambient environment and even within a human being. Sensor-driven computing strives to gather and interpret useful information about the physical environment through a combination of measurements using myriad sensors and data analysis. Although this data analysis might appear trivial in the simple case of motion-controlled lighting or the very familiar thermostat, for instance, the majority of sentient applications require non-trivial amounts of processing, and the desire for improved quality and better refinement of sensor data necessitates complex algorithms.

The sensory information is used to compose the user's context automatically and adapt or augment the computer's actions accordingly, thus enabling a more natural, intuitive and appealing interaction. In addition, combined with progress in artificial intelligence, this increased perception can make computers truly proactive; thereby realizing the dream of ubiquitous computing which is to have very large clusters of computing devices at our service, doing what we want even before we tell them to. These clusters are described succinctly by what can be called the "Four I's" -

- Invisible
- Intelligent
- Interconnected
- Independent

The central tenet of this thesis is that we cannot satisfy all four criteria simultaneously without radical changes to the design of the computational elements embedded in the infrastructure. Sensor networks form a major component in this futuristic vision of computing but ultimately, sentient computing will be limited by power consumption problems due to the scale of the proposed systems and the low computational efficiency of conventional computing devices. The goal is to identify and implement a reliable way of improving computational capability without the excessive power consumption common in desktop and server environments, and demonstrate that this design methodology works in a practical system.

In proposing new dimensions in computing for the future of the planet, Hopper [56] identifies the sensing, aggregation, and interpretation of global-scale sensor data as an essential part of this vision. Detailed monitoring of energy usage and natural resources will become crucial in a bid to optimise their consumption. However, this effort can only be maximally beneficial if the resulting large-scale sensor networks are themselves as energy-efficient as possible.

## 1.1 Research Statement

It is possible to make significant improvements in the energy efficiency of large-scale sensor networks by exposing and exploiting the high-levels of concurrency inherent in such computing environments.

This dissertation tackles the following research questions —

1. Is the addition of supernodes to conventional homogeneous sensor networks beneficial in terms of the overall energy consumption?

2. How can one design an embedded multiprocessor which can be deployed in sensor networks to act as a supernode?

3. Can lightweight hardware be created to improve the integration of multiple cores and provide support for concurrent software? What are the energy benefits of such an addition?

4. How can one scale performance (effectively reduce latency and improve throughput) in an energy-efficient way through multiprocessing?

## 1.2 Outline of the Thesis

This dissertation contains 7 chapters. Chapter 2 examines several approaches to the problem of energy-efficient processing. It considers the current state-of-the-art ways of improving the performance of a single processing node in an energy efficient way, and outlines the difficulty of efficient distributed processing within sensor networks. It analyses current models of distributed communication and computation; and presents an argument for a new sensor network architecture using nodes with enhanced support for parallelism.

Chapter 3 explains the design decisions in creating the power-efficient processor called SpotCore from a tabula rasa. SpotCore forms the basic processing element in the multiprocessing infrastructure which is presented in this thesis, and it has built-in primitives which make multiprocessing easier and faster.

Chapter 4 presents a hardware scheduler with a robust scheduling strategy developed for lightweight management of threads which are independent execution sequences. The hardware scheduler also saves power by minimising the scheduling latency. Other important issues such as deadlocks, priority inversion, and hardware memory protection are discussed.

Chapter 5 evaluates the multiprocessor platform with an algorithm which is very useful in sentient computing. The chapter develops ideas on writing efficient parallel software, and parallel composition. It shows that tasks can be completed as quickly as possible within the network with minimal energy requirements.

Chapter 6 develops a novel sentient system using this multiprocessor, referred to as the Data Analysis and Processing Hub (DAPH), for in-network processing. To provide a real implementation and an experimental test-bed for the DAPH platform a floor-based location system design is presented. The DAPH platform can, however, be used as a generic platform to manage any given cluster of sensors which would benefit from low-power high-performance processing. A location system was chosen for this demonstration because location information is regarded within the research community as one of the most important pieces of context information, and it presents challenging processing demands. The additional computational load due to the scaling of the system can be handled efficiently by the multiprocessing architecture.

Chapter 7 summarizes and concludes the dissertation. It also proposes future research directions.

## 1.3 Contributions

The main contributions of this thesis can be summarized as follows:

1. Survey of sensor network architectures.
2. Development of a small, fast, efficient CPU architecture with multiprocessor extensions.
3. Creation of a novel scheduler implemented in hardware and operating a scheduling algorithm across multiple cores.
4. Debugging and profiling environment for embedded multicore programming.
5. Implementation of an efficient way of scaling processing within a sensor network.
6. Integration and demonstration of the benefits of a multiprocessing hub within a real-time system prototyped as a novel location system design

## 1.4 Publications

Some parts of this work have been published in the following conference papers:

1. Mbou Eyole-Monono, Robert K. Harle, Andrew Rose, SpotCore: A Power-Efficient Embedded Processor for Intelligent Sensor Networks. Proceedings of Second International Conference on Body Area Networks, June 2007.
2. Mbou Eyole-Monono, Robert Harle, Andy Hopper, POISE: An Inexpensive, Low-Power Location Sensor Based On Electrostatics. 3rd Annual International Conference On Mobile and Ubiquitous Systems: Networks And Services (MobiQuitous 2006), July 2006.

## 1.5 Workflow

The workflow shown in Figure 1.1 should guide the reader by summarizing the presentation of the work described in this thesis.

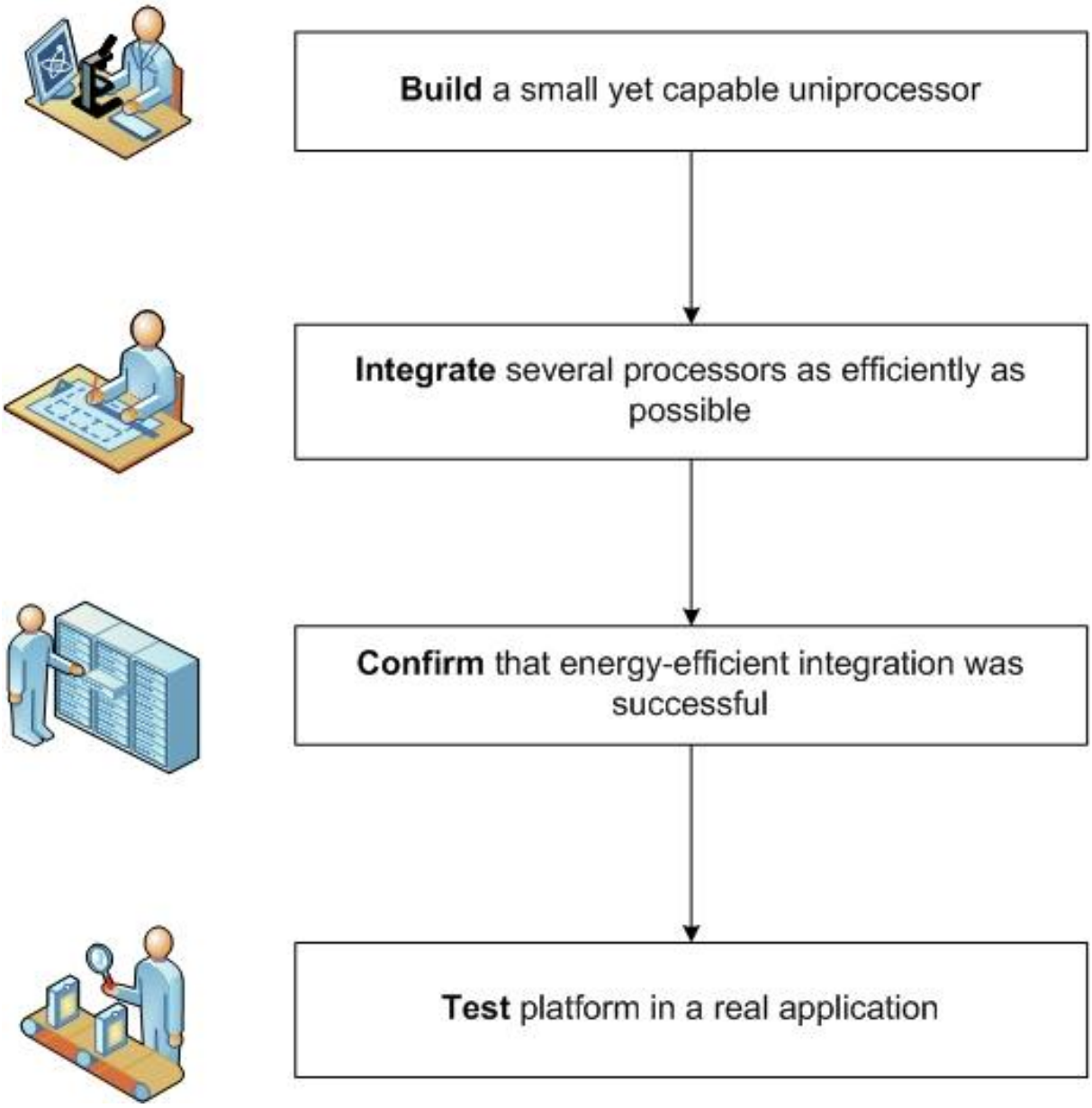


Figure 1.1: Thesis workflow

# Chapter 2

# Motivation and Background

This chapter highlights the importance of improving the efficiency and computational capabilities of the interface between computer networks and the physical domain and proposes a solution based on a multiprocessor known as the Data Analysis and Processing Hub (DAPH). We already encounter many sensors everyday but are not aware of their presence in most instances. A sentient system will almost always contain the following components -

1. Transducer
2. Digitiser or analog-to-digital converter (ADC)
3. Data analyser
4. Event notification, actuator and/or data store

This assembly of parts is not unlike a biological sentient system which fuses information from a very large number of sensory inputs (Figure 2.1). Sensors distributed all over the body are connected to a *central* processing engine, the brain.

Once a variable physical property or quantity has been measured in a sentient system, it is represented as a variable analogue voltage signal which is then sampled and digitised as this facilitates the proceeding processes of communication, interpretation and storage. The digitised (sample) data is usually fed through successive processing steps which may combine data from several other information sources (aggregation) and may also involve tasks such as digital filtering, and frequency transforms in order to extract meaningful information.

The culmination of the sensing and processing carried out in a sentient computing platform is the presentation of information that is purely pertinent to the user (that is at a sufficiently high level of abstraction), automatic actuation to aid the user, or simply efficient storage for future reference or consultation. In certain situations real-time

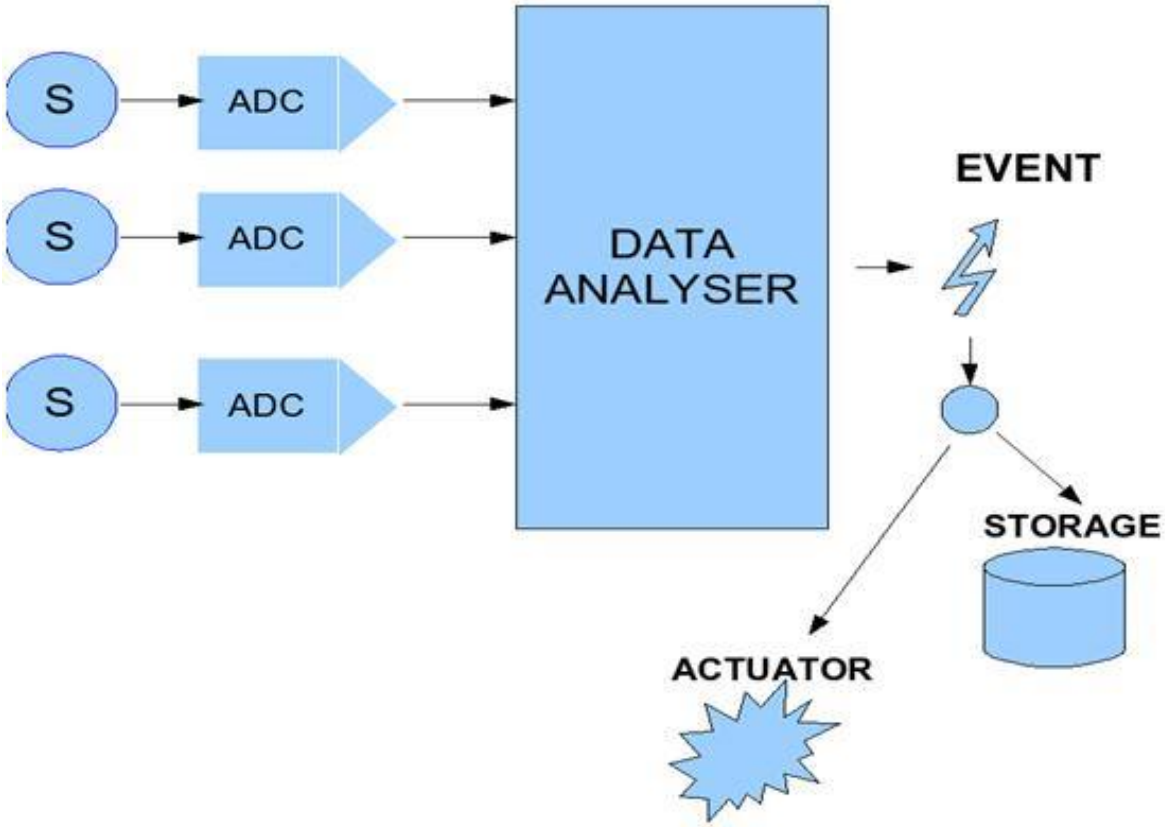


Figure 2.1: Flow of information in a typical sensor network

performance may be expected or even required for satisfactory operation. For instance, a sensor network acquiring position data could be used as a pointing device and would require a small time resolution or an information rate of over 30Hz.

Research in sentient computing has so far been heavily focused on developing and improving the quality of many different types of transducer for the large number of emerging applications. Unfortunately, it seems that these sentient systems will ultimately be crippled by power supply issues or become heavily reliant on existing infrastructure such as desktops and server systems which are inherently inefficient computing platforms.

In addition, systems based on the existing computer infrastructure are not always capable of real-time performance, are noisy, bulky and are usually expensive. This thesis investigates a common requirement so the results can be applicable to any group of sensors for any given application.

To improve the overall power-efficiency consistently, the primary focus is on optimising the data analysis required in most sensor systems in a generic way, given that the choice of transducer and method of event notification and actuation are ultimately application-dependent. There is a great variety of sensors utilising many different sensing techniques. It may well be that the total power required by the transducer, digitiser, event notifier and/or actuator combination is much greater than that required by the data analysis component, in which case the significance of this optimisation is reduced. However, it is more likely that as sentient systems scale, power consumption will become a major stumbling block due to the increased complexity of the algorithms required for data analysis and processing. The immense spatial coverage would cause a great deal of power to be required for data communication and aggregation. It is also likely that sensor research through advances in areas such as MEMS (Micro Electro-Mechanical Systems) will provide very low-power sensors in future. On the other hand, increasing processing capability in a power-constrained environment is a very challenging problem. Other common issues considered when building sentient systems are -

- Accuracy
- Reliability
- Calibration time
- Overall cost (deployment and maintenance)
- Update rate
- Sensitivity
- Lifetime

- Security
- Communication Bandwidth

In addition to optimising power consumption, the proposed architecture will enable greater security and reduce the deployment cost of sensor networks for ubiquitous computing. The increased processing ability will mean the sensor network can take advantage of more robust security algorithms which generally require a great deal of processing. It will also reduce the communication bandwidth required to backhaul information about the environment to a computer network to avoid any bottlenecks. It is hoped that the methodology will allow users to reap the benefits of truly autonomous operation in sensor networks as more capable local processing can find and fix faults in the sentient system quickly.

To summarise, the core motivational points for the architecture propounded by this thesis are the following:

1. In the future, it might become necessary to run compute-intensive applications within the sensor network in order to improve the quality of the data by adapting the sensor parameters in real-time.
2. High-end computing systems may be remote from the sensor network and the bandwidth, hence energy and cost associated with transporting data to and from the cluster will be very large.
3. A connection to a high-performance computer cluster would not always be available for certain deployments of sensor networks.
4. Server clusters often suffer from unpredictable latencies.
5. As sensor systems scale in size and complexity the energy efficiency of a distributed computation will be far from optimal if individual processing elements are not power-efficient.

## 2.1 Sensor Network Organisation

In creating wireless sensor networks, designers strive to reduce deployment cost so that they can obtain sufficient coverage at a reasonable expense, increase network lifetime by optimising energy usage, and improve reliability by incorporating self-organisation or auto-configuration. Wireless sensor network architectures can be split into two major classes - homogeneous or heterogeneous networks. In homogeneous networks, the sensor

nodes are all the same in terms of their sensing, processing and communication capabilities. Heterogeneous networks comprise nodes with different capabilities, that is, a large network of basic nodes might be interspersed with more advanced (capable) nodes which might sense a different (or related) property, process data faster, or serve as a communication hotspot or gateway. One can readily observe that this approach can be very cost-effective because the sensor network can utilize many simple (cheap) nodes to cover a large area and a few more advanced nodes which might not be so cheap to handle the more complicated tasks. In their extensive study of both homogeneous and heterogeneous wireless sensor networks, Mahtre and Rosenburg [81] confirm that heterogeneous networks can indeed be less costly. In a research survey on energy and cost optimizations in wireless sensor networks Mahtre and Rosenburg [80] point out that heterogeneous networks with hierarchical clustering can be more scalable. Let us examine clustering in more detail.

Clustering is a technique that is commonly employed in sensor networks in order to reduce the overall energy expenditure of the network. By clustering we mean that the sensor network is divided into a collection of smaller networks comprising a master or leader node and a few other nodes known as followers. The leader nodes called cluster-heads can then talk to each other and report back to what is known as a sink node which collects or aggregates data for the entire network. The clusters are non-overlapping thus a node belongs to exactly one cluster. Many protocols for electing a leader node have been proposed in the research literature. The most common among these is known as the Low-Energy Adaptive Clustering Hierarchy (LEACH) algorithm [52]. The fundamental idea is that the lifetime of the network is prolonged by changing (rotating) cluster-heads so that no single node is "overworked". Using stochastic techniques, the LEACH protocol ensures that each node will become a cluster-head exactly once in 1/p iterations where p is its probability of becoming a cluster-head in any iteration. Unfortunately, this does not yield the optimal energy efficiency because adjacent nodes could become cluster-heads or the cluster-heads could end up at the edge of the network. As a result, the LEACH protocol is extended in [48] by introducing determinism in the selection process. In this work, the residual energy of each node is used to modify the standard LEACH cluster-head probability threshold during the cluster setup phase. Clustering has energy-saving advantages because it reduces the size and complexity of the routing tables used to pass messages within the sensor network or send sensory data to the sink node. In most networks just one level of clustering is sufficient but multi-level clusters are also possible and indeed relevant for large networks [32].

Most protocols in research literature assume a homogeneous network. The problem of finding the smallest set of cluster-heads so that all nodes in the network belong to exactly one cluster is somewhat similar to the problem of computing the minimum dominating

set in graph theory [28]. Minimising the number of cluster heads is important in order to reduce channel contention which might render the whole clustering scheme counter-productive. The Self-Organising Sensor (SOS) algorithm [103] is proposed as an efficient way of minimising the number of cluster-heads. The Algorithm for Cluster Establishment (ACE) [30] achieves a very high packing density by only allowing new clusters to be created when the overlap between it and existing clusters is small. After creation, clusters will move apart or migrate to reduce overlapping and produce a cluster formation that is nearly ideal. Being a localized algorithm, it has a constant running time and communication overhead irrespective of the size of the network. This is in contrast to an algorithm such as Power Efficient Data gathering and Aggregation Protocol (PEDAP) [110] which is centralised and assumes (unrealistically) that the locations of all the nodes are known by the base station at the start of the clustering process. In this protocol, the base station computes the cluster formation and hence the routing table based on the minimum spanning tree algorithm which scales according to $O(n^2)$ where n is the number of nodes in the network. The base station takes the energy levels of the sensor nodes into account and recomputes and publishes routing information periodically. Upon reflection, one might argue that the computational cost at the base station is amortised by the relatively small set-up cost incurred within the sensor network itself. This sort of analysis leads one naturally towards the domain of heterogeneous sensor networks which will be discussed later.

PEGASIS [74] adopts a radical approach and reduces the number of nodes communicating directly with the base station to one by forming a linear network topology passing through all nodes where each node receives from and transmits to a nearby node. This protocol uses a greedy algorithm to construct a chain of nodes and the leader is chosen at random in each communication round so the nodes take turns in sending data to the base station. PEGASIS has several advantages over other clustering mechanisms, the most notable being the drastic reduction in communication costs between nodes during cluster formation. The volume of data between the leader node and the base station which may be far away is minimised. This is achievable in practice because data can be merged and reduced as it travels from node-to-node. However, some applications will not be tolerant of the long multihop routing delays inherent in PEGASIS and the protocol might even fail completely if data in transit cannot be efficiently fused.

Clustered wireless sensor networks can use either a single hop or a multi-hop mode of communication to send data from followers to cluster-heads. Mhatre and Rosenberg [79] present a thorough cost-based analysis of both these modes, and in addition to providing guidelines to decide which mode should be used for any given parameter set, propose a hybrid communication mode, which is actually more cost-effective than either mode.

There are many clustering algorithms in the research literature which are essentially variants of the fundamental algorithms, for instance the Time-Controlled Clustering Algorithm (TCCA)[100] routinely rotates its cluster-heads like in LEACH but introduces a time-to-live parameter in messages during cluster formation so as to be able to control the sizes of the clusters formed. The Weighted Clustering Algorithm (WCA) [31] uses a larger number of system parameters than other algorithms in electing cluster-heads. This combined weight uses parameters like the ideal node-degree (number of nodes a cluster-head can handle without degrading performance), transmission power, mobility, and remaining energy present at the nodes. The weight factors can then be adjusted to suit different sensor network environments. Since this election procedure is relatively complicated, it is desirable to avoid invoking the algorithm too frequently and this is in fact the case in the absence of mobility, that is, there is no recomputation if the distance between a cluster-head and any of its followers does not change. The Distributed Cluster Algorithm (DCA) [24] also uses weights to select cluster-heads. While the motivation for clustering algorithms comes from the fact that local computation is cheaper in terms of energy than communication it is possible that through over-reliance on system simulations with elaborate assumptions, network designers might inadvertently create solutions in which the energy requirements for the distributed computation of the cluster formation or appropriate network topology actually exceed the energy expended in a more straightforward approach to data dissemination.

Clustering is crucial for energy-efficiency in sensor networks when the key function is data-gathering because aggregation can be performed at the cluster-head as data flows to the base station or sink node. Some clustering algorithms such as LEACH-C[51] ('C' because it uses a *centralised* algorithm) require location information. The importance of understanding the spatial properties of a sensor network cannot be overlooked when clustering. Beaver et al. [25] consider optimising the construction of routing trees for sensor networks using attributes such as location and prove the suitability of this approach to in-network aggregation. They report energy-savings of up to 33% over conventional in-network aggregation algorithms. In general, better clustering using common attributes favours in-network aggregation because the preferred method of data dissemination in sensor networks relies on data generated by the sensors being tagged with attributes. This method known as directed diffusion [59] works as follows — a node requests data by propagating interests for a particular attribute and data corresponding to that attribute are drawn towards that node. Aggregation, hence in-network processing is made possible because the data is self-identifying. A more energy-efficient variant of directed diffusion, known as gradient-based routing[99] was proposed by Schurgers et al. In gradient-based routing, message hop counts are stored at each node and incremented as the message

registering interest progresses through the sensor network. Propagation of data to the interested node then follows the path with the greatest "slope" which is computed at each node by subtracting the hop count of each of its neighbours from its own.

Apart from directed-diffusion, there is another negotiation-based protocol which favours data aggregation by involving data tags. Known as Sensor Protocols for Information via Negotiation (SPIN)[69], it achieves energy-efficiency by adopting a 3-stage message-passing mechanism which relies on transmitting metadata (messages describing new sensor data) instead of just publishing the raw data. The stages in the protocol proceed as follows:

- Sensor node advertises new data using metadata to its single-hop neighbours
- If a neighbour is interested it sends back a request message
- The sensor node then replies with the actual data
- The neighbour then repeats this process

Energy can thus be saved by making the bulk sensor data flow towards interested nodes rather than being sent to every other node and by incorporating some form of computation or aggregation along the network path to these nodes.

In data-gathering applications, sensor network protocols in which nodes sense and subsequently transmit the measured data or its attribute are said to belong to the class of proactive protocols. Energy-savings are also possible and even substantially greater for some applications if a reactive protocol is used. One such protocol is called Threshold-Sensitive Energy Efficient Sensor Network Protocol (TEEN)[78]. In TEEN, two thresholds (a hard threshold and a soft threshold) are sent to the sensor node by the cluster-head. The sensor node does not transmit any data to the cluster-head unless the recorded data is within a specific range (determined by the hard threshold) and has changed by a certain minimum amount (soft threshold). After in-depth analysis on the organisation of wireless sensor networks, Vlajic and Xia [119] assert that the clusters created by any clustering methodology must lie within the isoclusters of the monitored phenomenon for the maximum benefits of clustering to be realised. Isoclusters refer to regions in the sensor network in which the measured data is spatially-correlated. The process of aggregation at the cluster-head in an isocluster is made more efficient because the sensor readings are very close and will usually change together thereby simplifying techniques such as compression. Solis and Obrackza [106] present an aggregation algorithm which uses local information shared by neighbours to group nodes with similar readings. A motivating example from temperature sensing is provided - nodes within a 10 degree temperature range belong to the same isocluster. Energy efficiency is achieved in practice because data will only be

transmitted to the base station or sink node when a line of constant temperature dividing isoclusters and appropriately named an "isoline" is detected. Local Negotiated Clustering Algorithm (LNCA)[125] is another algorithm which achieves superior performance by exploiting the similarity of sensor data in cluster formation. An interesting result of the work on clustering by Singla and Aseri[104] is the fact that a clustered wireless sensor network will perform well even if the formed clusters do not lie within the isoclusters of the observed phenomenon provided the distance between the scenario and the location of the observer is sufficiently great.

### 2.1.1 Towards Heterogeneous Sensor Networks

The major challenges in creating energy-efficient sensor networks were introduced in the last section. Cluster formation was examined as an effective way of reducing energy usage. However, most of the algorithms considered so far were developed under the assumption of a homogeneous sensor network. Most of the researchers struggled with the problem where a non-uniform distribution of the elected cluster-heads resulted in sub-optimal network operation. It is evident that greater determinism in selecting the cluster-head will be a major benefit in most sensor systems. The research work in this dissertation takes this a step further and proposes that we seed the sensor network with more advanced nodes which will act as cluster-heads. These nodes will be capable of faster (high-performance) computation and will also serve as gateways but will be designed such that their energy requirements remain small.

As we shall see later on, the requirement that the energy consumption of these "supernodes" is minimised is very important for this scheme to be scalable. The more abundant, smaller system elements provide dense coverage and close range sensing, while the fewer more capable nodes can perform sophisticated or compute-intensive functions.

Few researchers have stepped outside the general trend to investigate similar heterogeneous processing concepts in building sensor networks. Research on computation hierarchy for in-network processing by Tsiatsis et al. [115] demonstrates that while the digital sampling operation is more efficient on tiny sensor boards (MICA/MICA2 motes from Berkeley/Crossbow [7]), the computationally intensive acoustic beamforming routine used for location estimation is more efficient on a Compaq iPAQ[1] which is a high-performance node. Tsiatsis et al. advocate a hierarchical network level architecture comprising "...a few macro-nodes in a sea of micro-nodes" for system scalability. Wang et al. [120] use a two-tiered network for habitat-monitoring with a few powerful macronodes in the first tier, and many less powerful micronodes in the second tier. Their target application is the recognition and localisation of birdcalls and they discuss task decomposition and collaboration between the macronodes (PC104 boards[4]) which also act as the

cluster-heads and the micronodes which are tiny sensor boards (motes) [7]. In assessing security in energy-efficient sensor networks, Law et al. [72] categorize different networks into different system profiles and in consonance with this thesis envision a large number of sensor nodes interspersed with relatively resource-rich nodes which they refer to as "Rich Uncles".

A research team at Intel [5] has also used heterogeneous sensor networks to improve performance in a scalable way. They experimented with ad-hoc sensor networks with a high bandwidth IEEE802.11 mesh overlay network based on Intel Xscale Technology. While the experiment relied on the high-performance nodes being plugged in, which might be an unrealistic expectation, it proved the usefulness of heterogeneous networks. Another research team at Intel is working on developing the next-generation of motes with enhanced processing capabilities so they can be embedded within the heterogeneous sensor network.

A multi-tier heterogeneous sensor network with up to 3 different levels of processing capabilities is introduced in [113] to handle the problems associated with scaling homogeneous sensor networks. At the lowest tier are Crossbow MPR410 MICA2 motes acting as basic sensor nodes. In the middle tier lies the Crossbow Stargate SPB400[8], and a Hewlett Packard iPAQ (personal digital assistant) forms the top tier. The researchers argue that having greater computational resources close to the source of the sensor data improves the timeliness of responses to sensor events.

As was hinted earlier during the discussion of clustering techniques, heterogeneous sensor network ideas arise naturally during the analysis of cluster formation, out of the tendency to map cluster-heads to more resourceful and well-connected nodes. The Stable Election Protocol (SEP) [105] investigates the impact of heterogeneity in a sensor network with hierarchical clustering. The heterogeneity arises because certain nodes initially have more energy than others. It is important to note that this analysis is applicable to classical homogeneous networks because even these networks will eventually reach a state when nodes will have disparate energy levels unless they are perfectly load-balanced and cluster formation is optimal. Other clustering algorithms which consider heterogeneity and are sufficiently scalable include Adaptive Clustering Protocol (ACP) [93] and Multi-Event Adaptive Clustering Protocol (MEAC) [102].

## 2.2 Sensor Network Scalability

This section argues that some form of tightly-coupled multiprocessing will be required in sensor networks in order to close the performance gap between today's embedded processors and the demands of autonomous sentient computing in a power-efficient way.

To examine the stimulus for the development of a multiprocessor for sensor networks, DAPH, we begin by considering the options for reducing the overall energy requirement of an intelligent network with the ability to sense, compute and communicate.

Apart from the power required for computation and communication, some of the pertinent variables which influence the energy consumption of the entire network appear to be the following:

- $N_{links}$: the number of communication links made with other sensors. It refers to the links made from each CPU for data dissemination.
- $c_i$: a reduction factor which represents the fact that as the amount of sensor processing increases the amount of data which needs to be propagated towards the sink node and away from the network is reduced. A single sensor platform has limited processing capability making $c_i$ tend to 1.

Thus, one can reduce energy usage through the following:

1. Reduce the amount of communication needed for distributed computation among many different motes. This means designing the network so $N_{links}$ is small.
2. Reduce the overall time needed to complete a computation. One possibility is to use a platform with its processing energy requirement comparable to current mote platforms but capable of executing data manipulation functions in a much shorter time.
3. Reduce the power consumption of the computing elements without compromising the timely completion of critical tasks.
4. Improve processing capabilities so that $c_i$ is as low as possible.
5. Reduce the power required for communication between computing elements which is some function of the physical distance between them.

As we saw in the last section, some researchers have advocated distributed computation within sensor networks, where motes collaborate in order to meet the processing requirements of sensor applications. This seems like a reasonable option from a reliability standpoint because there is a lot of redundancy in the computation so the algorithm being processed can work around failure of a single node or group of nodes.

It seems likely that as the computational demands of sensor applications increase, unless better algorithms are developed which can hide the communication latencies or are sufficiently localised (spatially), it will become harder to scale this framework as these

latencies will make the efficient and timely distribution of commands, synchronisation messages, and shared data structures between processing elements difficult.

In general, there are two main models of data dissemination in sensor networks (or any network in general). There is the *push* model where data is sent to all collaborating nodes when it becomes available (and when the channel is free). The *pull* model responds to queries from individual sensors. Nodes which are out of the immediate transmission range of one node may be reached in a peer-to-peer fashion. For the purpose of distributed computation, one can see that in sensor networks, the *pull* model is a more appropriate choice, while the *push* model is more suited to simpler data aggregation. The two algorithms discussed in the last section, Directed Diffusion and SPIN, are notable instances of the *push* model and the *pull* model respectively. It is important to note that, contrary to a common misconception, while passing data through several nodes in a peer-to-peer manner might save power, it is not guaranteed to save energy because for each data transaction both transmitter and receiver circuitry on pairs of nodes have to be active simultaneously.

If we apply the *pull* model to the energy equation, due to the requirement to distribute queries, sensor data, (full/partial) results, and synchronisation tokens, $N_{links}$ rapidly approaches the number of processors in the network.

The overall energy of the sensor network increases in noticeably different ways depending on the ratio of the communication rate to the average of the transmit and receive powers. In addition, dynamic computation of optimal information flow paths or partitioning in order to reduce $N_{links}$ as the application progresses will impact performance.

However, if the processing nodes were actually small computing units within a single sensor network device, then the ratio of the communication rate to the average communication power is significantly greater than the case where the computing units are attached to separate sensors. As a result, the effect of increasing $N_{links}$ is mitigated when we have more computational units embedded within a single sensor network device.

The energy required to perform a distributed computation in a wireless sensor network such as that shown in Figure 2.2 increases at a much faster rate as the network is scaled than for one with a monolithic processing platform such as that shown in Figure 2.3.

To put this assertion in perspective, the on-chip data rate between processing elements can exceed 40Mbps while the overall communication power is still in the milliwatt range; but in a wireless sensor network data rates usually do not exceed a few hundred kilobits per second and require transmit/receive powers of about 40mW on average [67]. This implies that the increase in communication efficiency makes a very significant difference in terms of energy-efficiency as sensor networks scale.

### 2.2.1 The Role of the DAPH

The approach presented in this dissertation seeks to minimise energy usage by utilising a multiprocessor platform to speed up execution. For this scheme to be viable, the multiprocessor, called the Data Analysis and Processing Hub (DAPH), must be composed of small, lightweight processors. In chapter 2 we show that through careful design and organisation, these small processor units can be both lightweight and capable of fast execution of common sensor network operations. Although the DAPH approach might increase the communication rate from sensors to the processing hub as they have reduced processing capability in the immediate vicinity; this is offset by the fact that communication pathways between processing elements exchanging data much more frequently is more consolidated in DAPHs. The DAPHs will in general be in close proximity to the sensor groups they serve, which is in accordance with the principles of heterogeneous hierarchical clustering discussed earlier.

In addition, synchronisation is simplified and the overhead of partitioning tasks onto many execution units while taking factors such as their energy state into account is avoided. This framework avoids duplication and facilitates sharing of code, memory and CPU resources. However, the approach is more applicable to applications which require a lot of aggregation of sensor data or non-trivial amounts of processing. If all that the sensors are required to do is to report when a particular sensor reading rises above a specified threshold then the processing capability provided by the DAPH platform will be somewhat unnecessary.

The interrupt latency is also reduced even in the presence of uninterruptible tasks because another processor could easily be made available to handle the incoming interrupt. One can therefore utilise cheap, sensors equipped with low-power communication interfaces which are capable of communicating with the nearest DAPH which in turn is connected to the rest of the infrastructure. This approach also minimises power lost due to the ad-hoc dissemination of crucial control information for synchronisation and load-balancing across the sensor network.

A DAPH is therefore similar to a conventional sink node in terms of data connections but instead of simply acting as a gateway or a router bridging the internet and the sensor network, it is capable of high-performance processing with lower energy demands. In addition, the amount of communication between the DAPH and the rest of the infrastructure is made as low as possible because of the higher level of in-network processing. Since this link might be over a greater distance than the distance between the DAPH and local sensor nodes (and even involve a low bandwidth communication channel) communication energy is saved and external delays are avoided. Figure 2.3 and Figure 2.2 illustrate the main structural difference between the DAPH approach and a more conventional sensor

network. Items labelled P are the processing elements and the sensors are labelled S. Although the processing elements are not shown in Figure 2.3, this is just an idealisation and in reality, they may be present but simply able to *sleep* more hence consume less energy.

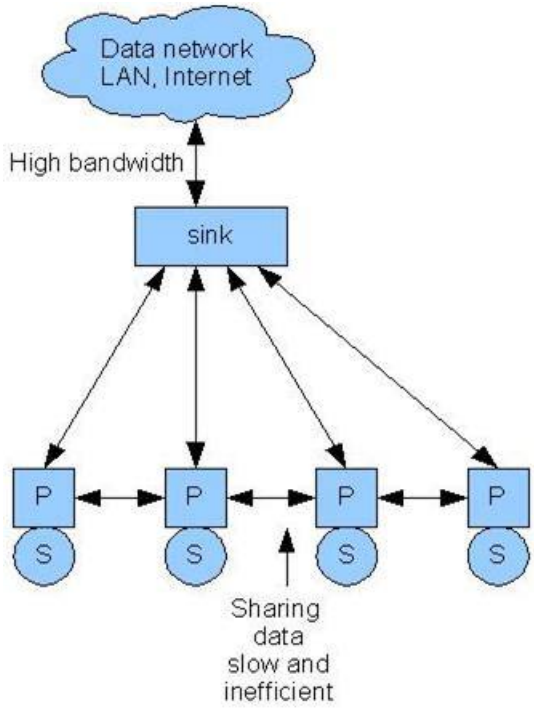


Figure 2.2: Conventional sensor network with motes

## 2.3 Sensor Processor Power Reduction Options

It is common for designers to talk about reducing power when what is usually more significant is a reduction in energy consumption. Power optimisations can indeed yield low energy devices but only if they are accompanied by a reduction in the time needed to complete tasks or at least by keeping the execution time of the processor constant. Another common misconception is to assume that there is a strictly linear dependence between power and frequency. However, increasing the clock frequency leads to the development of more complex circuits to handle the increased timing problems, and uses transistors with smaller dimensions which have more leakage power [65]. Thus arbitrarily increasing the clock frequency is not a viable route for low energy computation because while tasks might appear to complete faster, the power consumption would be increased by a greater factor.

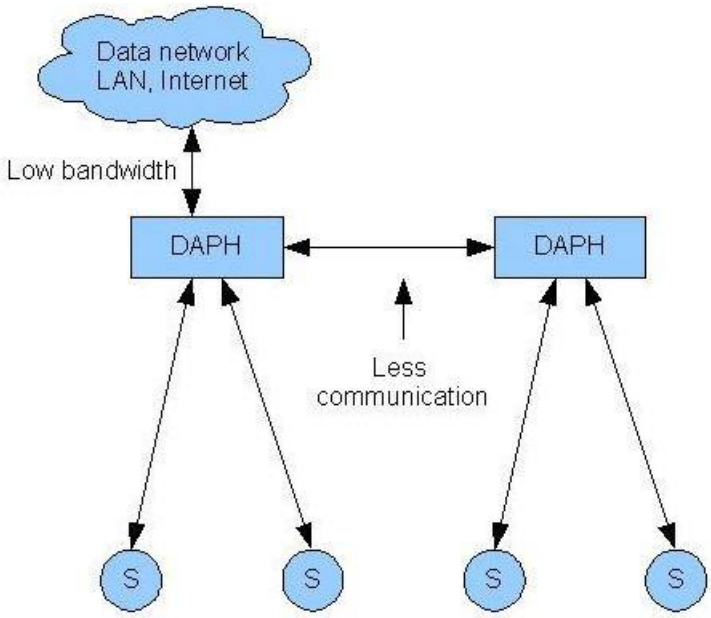


Figure 2.3: Sensor network with processing hubs

One can readily observe that multiprocessing is particularly suited to sensor-driven computing because of the inherent parallelism in data streams and the large set of subroutines which are common between applications making collaboration between multiple CPUs easy. We can reduce energy consumption in sentient environments by executing many tasks in parallel - completing them in less time and "sleeping" more. In the development of the DAPH approach, we choose to take advantage of coarse-grained parallelism rather than fine-grained parallelism. An example of fine-grained parallelism is instruction-level parallelism which gives rise to processor designs incorporating Very Long Instruction Word (VLIW) and Superscalar concepts. There are several reasons for this choice.

Firstly, fine-grained parallelism relies either on very advanced compilers (in the case of VLIW) or substantial hardware (in the case of superscalar architecture with dynamic issuing) to spot parallelism in the instruction and/or data flow.

Secondly, such designs lack the generality, extensibility and flexibility important in sensor networks since the applications can be very diverse.

Thirdly, as we have already noted, it is more intuitive for a designer to separate several tasks into individual threads which can run on separate processors without much interference.

### 2.3.1 Low Power Electronic Devices

Complementary Metal-Oxide Semiconductor (CMOS) is the predominant transistor technology primarily due to the fact that ideally it has no static power dissipation. However, as transistors are scaled down in an effort to increase speed, the power lost due to leakage currents (which is normally negligible) is becoming more significant relative to the power lost due to switching between logic levels, appropriately named dynamic power. This dynamic power can be approximated by $P = C_L * {V_{dd}}^2 * f$ — where $V_{dd}$ is the supply voltage, $f$ is the clock frequency and $C_L$ is the load capacitance.

Thus, in order to reduce dynamic power at the physical device level, one needs to reduce circuit capacitances or the supply voltage. Otherwise, if one is willing to sacrifice raw performance, or if operating the entire circuit at a very high speed is not necessary for a given application, then a slower clock can be supplied to the circuit or clock-gating [65] can be performed at various levels of the design hierarchy. Reducing the supply voltage is usually the most effectual way of reducing the dynamic power consumption due to the quadratic relationship.

However, this often necessitates a reduction of the threshold voltage which in turn causes an exponential increase in sub-threshold current (and hence power dissipation). If the threshold voltage is not reduced as the supply voltage is reduced, the on (drain-source) current would reduce and this will lead to a drop in performance. A small fraction of the total power consumption is lost due to brief short-circuit currents which flow from the supply rail to ground when the circuits are switching.

The International Technology Roadmap For Semiconductors (ITRS) [10] which analyses, summarises and forecasts trends in the semiconductor industry, in order to produce a guide for industry, academia and governments, states that reducing leakage currents is the paramount objective in low-power devices. The ITRS figures demonstrate that while transistor physical gate lengths will be scaled down to about 9nm by 2016, the maximum power consumption will, unfortunately, not reduce but rise by about 25%. This increase can be attributed to increases in chip operating frequencies, interconnect capacitance and resistance, and leakage currents. High-k dielectrics have been employed to reduce gate leakage currents which are on the rise due to ever thinner gate oxide layers.

Given that pure CMOS scaling is fraught with problems and has a limited ability to tackle power issues, researchers are actively looking for new device architectures. Silicon-On-Insulator (SOI) technology is capable of reducing power consumption and improving speed by reducing circuit capacitances. Strained silicon technology [44] is also attractive because it enables both higher clock frequencies and lower power consumption without shrinking device features and at a low cost. It involves changing the inter-atomic spacing of the silicon crystal so that the mobility of charge carriers is improved. Other advances

include double-gate structures such as FinFETs which are transistors built on raised fin-shaped silicon structures and represent a departure from traditional planar design concepts [16]. The quest for speed and low-power consumption has also led researchers to seek to replace the commonly-used polysilicon gates with metal ones. For even lower power dissipation, Kim and Roy propose a double-gate MOSFET design which uses subthreshold currents [66].

Radical approaches are less attractive as they might involve relinquishing investments in an expensive semiconductor manufacturing infrastructure, but the rewards can be great. For instance, experiments with transistors made using carbon nanotubes and nanowires have shown performance improvements by a factor of 3 over current technologies at the same power [64]. Intel and Qinetiq have been able to develop a "quantum well" transistor which consumes only one-tenth of the power of conventional transistors [63].

Spintronics exploits a quantum property of electrons known as spin for processing information rather than their charge. It is already being used to produce Magnetoresistive (or Magnetic) Random Access Memory (MRAM) which is touted as a dense, fast and non-volatile memory. Research is focusing on creating Spin FETs using magnetic semiconductors which exhibit ferromagnetism and are thus capable of producing and manipulating spin-polarized electrons. In terms of energy dissipation, spin-dependent transistors have the great advantage that the energy needed to manipulate electrons' spin is much lower than that used in conventional electronic circuits [19].

### 2.3.2 Circuit and System Optimisations

Low-power circuit designers seek ways to optimise the way information flows through their circuits; and logic solutions that use fewer gates and interconnections. Advanced logic minimisation techniques coupled with intelligent layout tools have been developed. In addition, some tools such as the Power Compiler from Synopsys [109] aggressively reduce power at the Register Transfer Level (RTL) and gate level by automatically performing clock-gating which refers to the ability to stop the clock when there is no useful work being performed.

Since tasks such as producing a netlist from a specification written by the designer in a hardware description language and physical layout are heavily automated due to their increasing complexity, it is generally not always possible for the designer to make significant power-saving changes at the gate level. Careful planning of datapath and control logic is usually the best one can do for low-power design. For instance, ARM microprocessor designers operate primarily at this level and can provide appreciable levels of performance while keeping the power consumption in check (an implementation of the

ARM11 micro-architecture consumes less than 0.6mW/MHz at 1.2V on a 0.13m process technology [3]).

Given that significant amounts of power dissipation occur when driving signals off-chip, due to the large capacitances of the interconnects, more electronic designers are embracing the system-on-chip concept which strives to integrate a large proportion of the functions of a device on a single chip. Designs with cleaner (properly-defined and standardised) interfaces are important in the power argument as they eliminate redundant logic at interfaces which lead to bulky and power-hungry circuits. System-in-package or multichip modules [68] are also attractive from the point of view of better integration.

Asynchronous circuits dispense with the clocking schemes used in synchronous circuits by using the validity of data output from one stage to trigger the next stage. Self-timed logic blocks operate only when needed and so do not drain the power supply unnecessarily. This feature is inherent in the event-driven circuit design and does not require extensive clock-gating to implement. The caveat here is that the handshaking logic required may consume significant amounts of power if not carefully designed, and the low power advantage is more noticeable in designs where only a fraction of the datapath is heavily utilised at any point in time. Examples of successful low-power designs include the AMULET3 microprocessor [43] which is an ARM-compatible processor, and an asynchronous 80C51 microcontroller with a power reduction of 75% [117]. Some asynchronous circuits are delay-insensitive which implies that the design will operate correctly regardless of variations in gate and interconnect delays. This property can be useful in dynamic voltage scaling, used to conserve energy, as the circuit adapts automatically, unlike a synchronous circuit which requires additional circuitry and design effort to scale down the clock frequency on the fly. Despite these advantages, asynchronous design techniques were not pursued in this research work due to the lack of adequate hardware development and prototyping tools.

All computing tasks are not equal — a high-performance hardware system is never fully utilised all the time. An Intelligent Energy Manager [6] performs real-time dynamic voltage (and frequency) scaling which offers the application program an opportunity to tailor the supply voltage and frequency to suit a particular application. This is a smoother and more efficient way of managing power than having full-power operation punctuated by low-power idle or sleep modes.

Other system parameters such as precision and accuracy can be tweaked for low-power operation. Amirtharajah and Chandrakasan present the design of a DSP capable of reducing its computational precision, hence core activity for certain tasks, to reduce power usage [15].

Mycroft et al. were able to reduce the power consumption of an embedded processor system by 20% by reducing switching activity [84]. This was achieved by rearranging internal bit representations at compile-time to minimise logic transitions at run-time.

Instead of running a high-performance task on a fast processor requiring lots of power, it is indeed possible and more efficient to break the task up into several parallel processes, which can then run at a lower speed on low-power, parallel hardware modules. This approach relies on achieving greater concurrency in software programs and having proper synchronisation mechanisms in hardware. This thesis follows this trajectory albeit with a shared memory focus which works well for a small number of processors. These "clusters" of processors may then be interconnected to form a network. In this way, one can extract the maximum performance possible from each processing element as using shared memory is in general a very fast interprocessor communication scheme and outperforms message-passing for up to about 8 cores.

Another possible optimisation route is to develop a more streamlined hardware-software interface or light-weight operating system, and to remove redundant processes which serve to adapt any given hardware to diverse application needs which may or may not be necessary.

### 2.3.3 Efficient Processing within Sensor Networks

Nazhandali [85] notes that designing energy-efficient sensor processors is a fairly recent undertaking and describes how an ultra-low energy processor may be designed by combining optimisations at the microarchitectural and instruction set levels, with subthreshold voltage circuits [86]. These circuits often involve significantly lower clock frequencies and are thus successful in reducing power for applications that do not require a high throughput. However, this limitation is too great in the general case.

Many sensor network platforms have used off-the-shelf components which were not primarily designed for the strict ultra-low power environments they are then embedded within. Virantha et al. [38] present a novel architecture based on an asynchronous 16-bit RISC core. They justify their approach with the fact that in asynchronous designs, not all parts of the circuit are actively changing so power is not wasted. In addition, there is no power-hungry clock-tree. Event-driven execution within sensor networks fits nicely in this paradigm. They observe correctly that verification is often a problem in asynchronous designs and this processor known as SNAP/LE tries to address some of these issues.

Their reported worst-case energy consumption figure is 300pJ/instruction and they note that this stands out favourably when compared to approximately 1500pJ/instruction for an off-the-shelf Atmel microcontroller. The design principles of SNAP/LE differ from those outlined in this thesis since the latter retains a synchronous design methodology

(the most viable route for integrated circuit synthesis) and focuses instead on optimising the instruction set and processor architecture. The researchers in [71] use a low-power compilation methodology to save energy within a wireless sensor network by making optimisations at the microprocessor instruction execution level.

The SNAP/LE project also shows how the execution time of a given task can be reduced relative to an Atmel microcontroller running TinyOS on a Berkeley MICA mote, by using a scheduler implemented in hardware and tightly-coupled to the processor. This technique yields significant power improvements, and is adopted in SpotCore. However, this thesis discusses and evaluates a more scalable hardware-based scheduler which is capable of supporting not only event-driven execution but true multi-threading, and which achieves a better degree of fairness than the simple non-preemptive FIFO-based scheduler used in SNAP/LE.

Mota et al. [83] also take a hardware-oriented approach and improve the information processing capability of sensor network nodes by re-implementing tasks as hardware modules.

Warneke et al. [122] produced a design which improves power-efficiency by having separate hardware subsystems which can be shutdown independently, elaborate clock-gating, and guarded ALU inputs. However, the design uses no datapath pipelining in a bid to avoid the associated hardware overhead. This in turn limits the maximum clock frequency. However, the designers note that the platform known as "Smart Dust" will be used in low data rate scenarios where high clock frequencies are not normally needed. At 500kHz and 1V, the design utilises 12pJ/instruction. With the possibility of collaborative processing between sensor platforms and the high level of interest in in-network processing, much higher levels of performance may be required and hardware limitations on the design speed are inadvisable. The instantaneous power might be reduced at lower frequencies (and voltage) but the overall energy consumption might be worse if the execution time is not also reduced through careful instruction set design.

Ciaran et al. [76] present a good survey of different processor architectures for wireless sensor networks and observe that current microprocessors have limited capabilities for handling complex data-processing tasks. The Texas Instruments MSP430 [58] emerged as the best architecture in the survey, with the smallest power consumption figures compared to the Atmel ATMega128L and the MicrochipPIC18.

The i-Bean [96] uses dual processors clocked at different speeds to improve power efficiency. The Imote2 [33] from Crossbow technology uses a high-performance, low-power 32-bit PXA271 XScale processor and is capable of dynamic voltage and frequency scaling from 13MHz to 416MHz. The platform also incorporates a DSP coprocessor to accelerate multimedia operations by extending the XScale instruction set. Preliminary data suggests

that, with the radio circuitry off, the rest of the chip comprising the CPU and memory consume about 2mW/MHz. It is an interesting fact that on this platform, which is touted as the most power-efficient sensor platform, the processing elements consume as much as 40% of the overall power consumption when the radio circuitry is on; indicating that research into more power-efficient cores is at least as important as research into low-power communication interfaces in the quest to reduce the overall power consumption of sensor platforms.

# Chapter 3

# The SpotCore Architecture

The design of SpotCore is primarily motivated by the desire to integrate as much essential functionality as possible into a single core whilst taking great care in the instruction set and processor design to avoid the introduction of redundant hardware. There are many optimisations which can be applied to the basic RISC pipeline [53] but it is important to identify a set of reliable optimisations which would still yield a reasonable performance from a highly minimalist design philosophy.

While more pipeline stages will enable the design to be clocked at higher frequencies, by putting less work or logic in each stage, this adds complexity, increases hardware size and worsens the branch or exception penalty. One can observe that processing in sensor networks is of a highly concurrent nature as there might be multiple data streams requiring analysis. It is very likely that as these networks scale, this parallelism is going to increase dramatically. This leads to an increased probability of many context-switches so it is desirable to keep as little state internal to the processor as possible (but relevant to any given thread). In addition, SpotCore is being designed for low-power environments where extremely high clock frequencies in the gigahertz range are not feasible due to the substantial increase in power requirements. The critical path length constraint can be relaxed as a result.

Deeply-pipelined processors will typically need a great deal of logic to transfer information contained in instructions still in the pipeline to preceding instructions. These are known as feed-forward paths. Using an instruction width of just 16 bits instead of 32 bits or higher reaps power savings by reducing the bandwidth requirement of instruction memory and may additionally lead to high code density.

The datapath of SpotCore (comprising registers, internal buses and functional units) is configurable but is currently set to 32 bits wide as this is believed to be sufficient for handling many different types of sensor data. The important parts of the internal structure and datapath are shown in Figure 3.1. Due to the streamlined instruction set design, the decoder is particularly lightweight and as a result in the actual implementation

some precomputation is performed in that stage to balance the timing with respect to the execute stage.

The instructions are split into different classes depending on whether they are dyadic, monadic or require no operands. This enables us to attain a highly orthogonal instruction set design which makes very good use of the available encoding space. This encoding scheme is illustrated in detail later. Most of the data processing instructions (Add, Subtract,Multiply, AND etc) on SpotCore are dyadic but due to the restrictions on instruction length only register contents may be used as operands.

This is in contrast to the plurality of addressing modes used on an ARM processor [17] and it greatly simplifies the addressing scheme leading to more compact decode logic. For instance, while a data-processing instruction on an ARM can take one of its operands from the output of the shifter, the same operation on the SpotCore processor would have to be performed by two separate instructions. In addition, register lookup uses indices specified in fixed parts of an instruction in order to further simplify decode.

All SpotCore instructions may be conditionally executed — the predicated instruction scheme has been found to be successful in ARM processors [17] and Intel's IA-64 architecture [61] by reducing the number of branches. However, in order to save encoding space the design mandates that a conditional instruction cannot also be "flag-modifying". This means we can use just one field to specify whether an instruction sets or clears the flags or is itself conditional on some flags set previously within the processor. The field for the condition specification is restricted to 3 bits and the extra bit which would have been necessary to represent a "flag-modifying" instruction is captured by a special 3-bit code placed in that field. The trade-off arose from observing many instruction traces of code compiled for an ARM processor and not finding many instances of conditional instructions which modified flags. The combination of this 3-bit field and a 4-bit primary opcode field leaves only 9 bits for encoding the registers being accessed. This in turn restricts the number of visible and directly addressable registers to eight. Another power-saving measure seeks to reduce the number of register ports - two read ports are adequate if one uses only simple dyadic instructions.

After much deliberation (and some frustration) it was decided three read ports and two write ports would be incorporated in order to be able to support certain very useful instructions:

> STR r0,[r1],r2 Store the value in r0 at the address pointed to by the value in r1 and update the register r1 with the sum of the values in r1 and r2.
>
> MLA r0,r1,r2 Place X + Y*Z in register r0, where X,Y, and Z are the values in r0,r1 and r2 respectively.

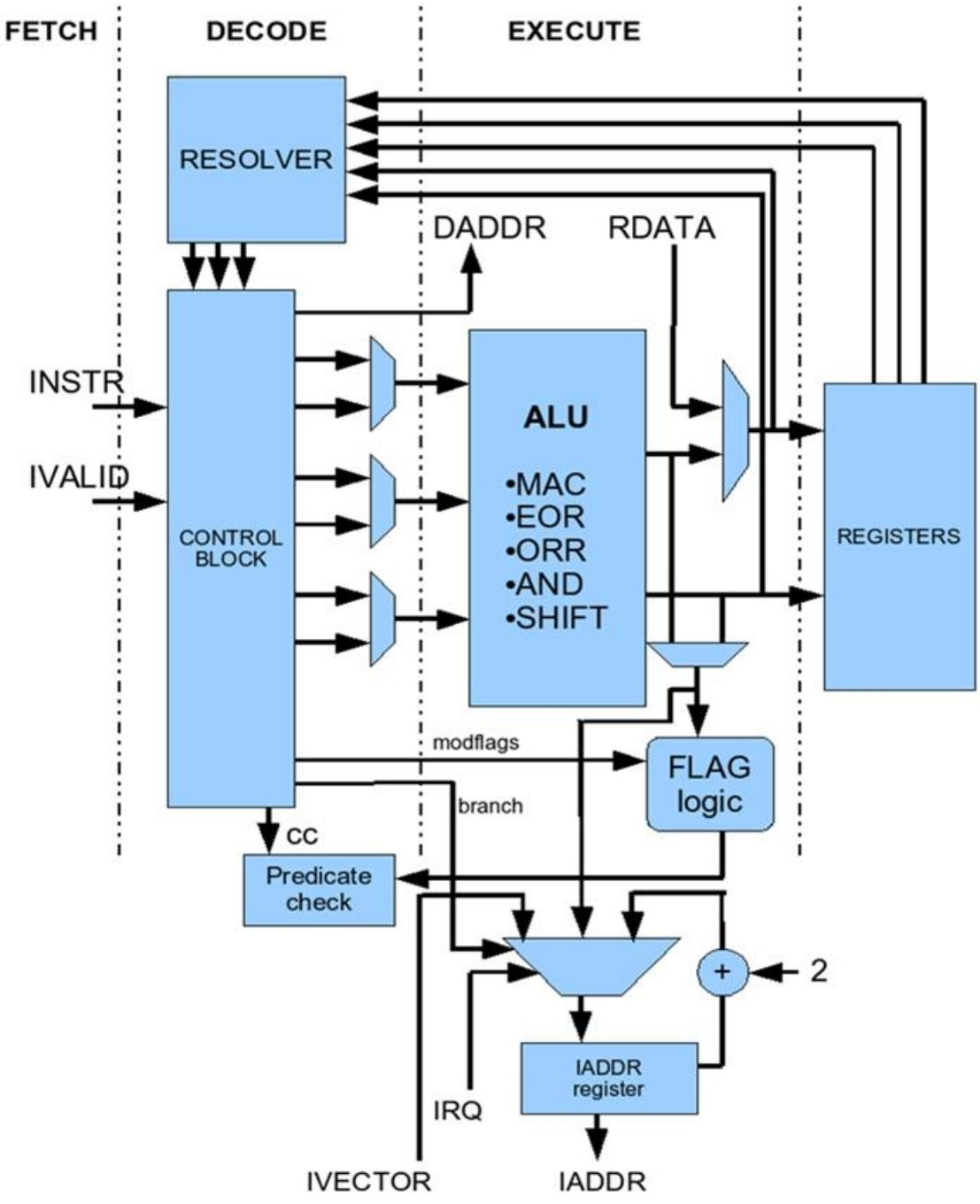


Figure 3.1: SpotCore pipeline

LDR r0,[r2],r3 Load r0 with the value at the address pointed to by the value in r2 and update the register r2 with the sum of the values in r2 and r3.

SORT r0,r1 Swaps data values so that the register with the higher index contains the higher value.

These instructions and a few others necessitate either three simultaneous reads from or two simultaneous writes into the register file. However, the presence of two write ports creates an opportunity to improve the execute stage by arranging the logic so its more critical pathways feed into a less heavily-loaded write port and thereby creating more balanced timing in that pipeline stage.

SpotCore has a smaller register file than most embedded RISC processors and no banked registers. In order to save time on exception entry, register-file stacking is managed by hardware. Register 7 is the program counter and the stack pointer is internal. SpotCore also maintains an internal link register (LR) which is saved automatically when nested subroutines are detected — when a "BL" instruction is issued. Apart from the assumptions surrounding the handling of the LR there are no other assumptions about when a particular register is stacked, that is there is no restriction on the programmer's choice of calling convention. This increases the number of general purpose registers available and also saves time since it can be stacked in parallel with branching. A separate instruction is provided to recover the link register value if necessary. In spite of this feature, the relatively small register file will be a serious limitation if the working register set of an application is large. The options for relieving the register pressure include advanced compilation techniques or making the program counter internal in future implementations. It may also be possible to extend the instruction set by developing a "register prefix" instruction whose function will be to add extra bits to any register indices used in the next instruction.

With respect to security, SpotCore adopts a straightforward model and simply operates in one of two modes - *trusted* or *untrusted*. It avoids any elaborate exception-handling scheme which is normally expensive in terms of area (possibly involving many register banks). For flexibility, the subset of privileged/trusted instructions are specified in a separate decoder independent of the core of the instruction decoding logic. Memory protection is discussed in the next chapter from the perspective of an embedded multiprocessor with the requirement for thread isolation.

## 3.1 Handling Loops

SpotCore has a branch penalty of 2 cycles. Due to the fairly mild impact of branching and exceptions on its short pipeline, it was decided that the performance boost afforded

by branch prediction in this case would not justify the extra hardware needed. However, in order to mitigate the impact of branching in the common scenario involving fixed branches at the end of iterative blocks of code such as in FOR loops, a LOOP instruction was added to the instruction set. Branch prediction logic can automatically infer that multiple iterations will be performed but the uncertainties lead to a few pathological cases. The concept is similar to that used in the Intel x86 architecture [60] but the mechanism presented here is different and the occurrence of nested loops is detected and handled automatically by the SpotCore hardware. The IBM PowerPC [57] has a dedicated loop count register.

The purpose of this instruction is to ensure that the process of checking for the last iteration at the end of the loop body can happen while the loop body itself is being executed so the pipeline can be filled with the correct set of instructions and the effect of the branch is hidden. As a result, an explicit branch at the end of the loop, and the penalty associated with it, are avoided.

To illustrate this point the following typical assembly code sequence A (written in ARM assembly) can be rewritten as code sequence B on SpotCore.

```
;CODE SEQUENCE A
        MOV r0,#10 ;set up loop counter
label
        LDR r1,[r2],r3 ;loop starts here
;rest of loop body
        LDR r4,[r5],r6
        SUBS r0,r0,#1
        BNE label
;other instructions

;CODE SEQUENCE B
;set up loop counter
        MOV r0,#10
;set up loop end address
        MOV r1,#end_address
        LOOP r1,r0
        LDR r1,[r2],r3 ;loop starts here
;rest of loop body
end_address
         LDR r4,[r5],r6
;other instructions
```

In the case of a nested loop, the information held in the loop state machine in the CPU is written out to memory (the stack) automatically as the new "loop state" is created. A separate memory area (structure) holding loop state information at various levels might have the advantage that stack operations do not have to balanced within the loop so that we can break out easily but then this would have to be managed separately which complicates verification as pointers to the loop state must be set up and managed; the loop state will require proper handling when interrupts occur. The LOOP implementation uses a small amount of additional logic but it makes a significant difference to the execution time of an application if the loop body is small and there are many iterations. The result is essentially the same as loop unrolling performed by certain compilers. However, loop unrolling has inefficient memory usage in comparison because the code representing the loop body is replicated, and in some cases not all the overhead can be removed as with the LOOP instruction.

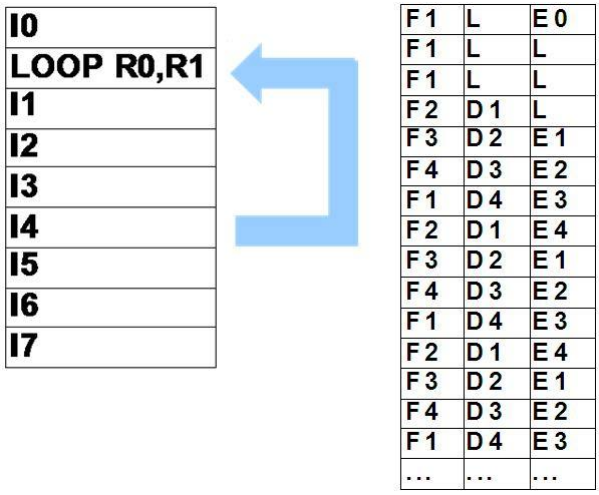


Figure 3.2: SpotCore speeds up loops

For the instruction sequence shown in Figure 3.2, the loop starts at instruction I1 and the last instruction is I4. The pipeline illustration on the right shows how the instructions can be fetched, decoded, and executed seamlessly with no pipeline "bubbles". No general-purpose register is used to maintain the loop — the loop counter is internal. The current loop counter value can be set or retrieved via simple instructions. Setting it to one enables us to get out of the loop after any iteration.

## 3.2 Instruction Set Design

The SpotCore instruction set is largely influenced by the many common RISC instruction sets. SpotCore attempts to include many of the most common instructions without violat-

ing size and power constraints (Figure 3.3), while also including some special instructions for thread management.

| Spot Core | ARM | TI MSP430 | MIPS32 |
|---|---|---|---|
| ADD rX,rY,rZ | ADD rX,rY,rZ | ADD src,dst | ADD rd, rs, rt |
| SUB rX,rY,rZ | SUB rX,rY,rZ | SUB src,dst | SUB rd, rs, rt |
| ORR rX,rY,rZ | ORR rX,rY,rZ | BIS src,dst | OR rd, rs, rt |
| AND rX,rY,rZ | AND rX,rY,rZ | AND src,dst | AND rd, rs, rt |
| EOR rX,rY,rZ | EOR rX,rY,rZ | XOR src,dst | XOR rd,rs,rt |
| BIC rX,rY,rZ | BIC rX,rY,rZ | BIC src,dst | ---------- |
| MUL rX,rY,rZ | MUL rX,rY,rZ | ---------- | MUL rd,rs,rt |
| MLA rX,rY,rZ | MLA rX,rY,rZ | ---------- | MADD rd,rs,rt |
| LDR rX,[rY] | LDR rX,[rY] | ---------- | LW rt,offset(base) |
| STR rX,[rY] | STR rX,[rY] | ---------- | SW rt,offset(base) |
| LDR rX,[rY],rZ | LDR rX,[rY],rZ | ---------- | LWXC1 |
| STR rX,[rY],rZ | STR rX,[rY],rZ | ---------- | SWXC1 |
| MOV rX,#N | MOV rX,#N | ---------- | ORI rX, zero, #N |
| B or BL label | B or BL label | JMP label | B offset |
| BL rX | BLX rX | ---------- | JALR rX |
| LSL or LSR rX,#Sh | MOV r0,r1,LSR #Sh | ---------- | SLL / SLR |
| PUSH {r0-r5} | STM sp!,{r0-r12} | PUSH src | ---------- |
| POP {r0-r5} | LDM sp!,{r0-r12} | POP dst | ---------- |
| RETURN {r0-r5} | LDM sp!,{r0-r12,pc} | RET | ---------- |
| LSL rX,rY | MOV r0,r1,LSR rX | (rotation) | SLLV |
| LSR rX,rY | MOV r0,r1,LSR rX | (rotation) | SLRV |
| INV rX,rY | MVN rX,rY | INV dst | NOR rX, rY, zero |
| INCR rX,rY | ADD rX,rX,#1 | INC dst | ADDI rd, rs, IMM |
| DECR rX,rY | SUB rX,rX,#1 | DEC dst | ---------- |
| ABS rX,rY | ---------- | ---------- | ABS fd,fs |
| SORT rX,rY | ---------- | ---------- | ---------- |
| NEG rX,rY | RSB rX,rY,#0 | ---------- | NEG fd,fs |
| CLZ rX,rY | CLZ rX,rY | ---------- | CLZ rd,rs |
| BITREV rX,rY | ---------- | ---------- | ---------- |
| LOOP rX,rY | ---------- | ---------- | ---------- |
| Thread instructions | ---------- | ---------- | ---------- |

Figure 3.3: Comparing SpotCore instructions with other RISC instructions

Many of these instructions would typically be found in digital signal processors. Apart from the simple register addressing mode shown in the table, the 32-bit ARM has other addressing modes where the second operand could be an immediate value or even specified as the result of a shift operation. Since it was not desirable for the shifter to be in the critical path and the available encoding space is somewhat limited, instructions for getting

immediate values into the processor were separate from those for shifting. Although this might represent a performance problem if these types of instructions are used frequently, it is easy to see that the code size relative to a 32-bit instruction set is unaffected as the two operations simply become two 16-bit instructions.

It is important to realise that though SpotCore is a standard RISC architecture it incorporates features whose benefits may be common knowledge but which are lacking in very popular RISC designs for reasons which are not always clear. For example, let us consider the MSP430 which is a 16-bit RISC CPU which lies at the heart of many sensor boards. It has 16 registers, 4 of which are treated specially — program counter, status register and constant generator which is particularly important because it provides six frequently used immediate values thereby reducing code size. However, unlike SpotCore (and ARM) not all MSP430 instructions are conditional. Furthermore, unlike SpotCore, MIPS32 [82] has a 32-bit instruction set with presumably sufficient encoding space for predication of any instruction but only a few MIPS32 instructions are actually predicated while all SpotCore instructions (16-bit ISA) can be predicated.

Following the implementation of an "ARM-like" instruction set, some useful but uncommon single-cycle instructions were subsequently added to the SpotCore instruction set without any drastic effect on operational parameters. These extra instructions were ABS (get absolute value), SORT (arrange values in registers based on their numeric size), BITREV (bit-reversal, useful in Fast-Fourier Transform algorithm), the LOOP instruction described previously, and some thread management instructions described later.

The facility to load a relatively large literal from the instruction stream as data was also added to the instruction set. The MOVE instruction (MOV) has a special bit which if set will treat the next instruction as data, and append the last 5 bits of the move instruction to that as seen in Figure 3.4. This means that besides the usual data memory access instructions one can either load a 5-bit value in one instruction or a 21-bit value in two instructions. This provides a very fast and efficient way of loading immediate values using a variable length instruction and though it is not common similar schemes have been mentioned before such as in [91]. Unfortunately, there is currently no way of restarting the instruction so problems may arise if it is split over a page boundary.

In contrast, one cannot load an arbitrary 21-bit value within a single (32-bit) ARM instruction but have to encode the immediate operand as an 8-bit constant and a 4-bit (even-number) rotate which is applied to it.

The four main SpotCore instruction formats are shown in Figure 3.5. These are followed by the four exceptional instruction formats. What this portrays is the fact that the width (number of bits), meaning, and placement of many sections of the instruction are kept as consistent as possible between instructions in a bid to simplify the decoding logic.

| [15:12] | [11:9] | [8:6] | [5] | [4:0] |
|---|---|---|---|---|
| 1010 | <S / Ccode> | <dest reg> | <next bit> | <VAL_5LSBITS> |
| <VAL_16MSBITS> | | | | |

Figure 3.4: MOVE instruction

This also means there is scope for using highly regular instruction patterns resulting in fewer bit transitions between instructions. Having a smaller number of transitions means low energy code sequences can be composed. By splitting the instruction set into different classes depending on their requirements, with respect to the number of registers required for a particular operation, one can achieve a compact hardware layout which favours an efficient design.

| Class 1 | [15:12] | [11:9] | [8:6] | [5:3] | [2:0] |
|---|---|---|---|---|---|
| | opcode1 | Predicate | rX | rY | rZ |

| Class 2 | [15:12] | [11:9] | [8:6] | [5:3] | [2:0] |
|---|---|---|---|---|---|
| | opcode1 | Predicate | opcode2 | rY | rZ |

| Class 3 | [15:12] | [11:9] | [8:6] | [5:3] | [2:0] |
|---|---|---|---|---|---|
| | opcode1 | Predicate | opcode2 | opcode3 | rZ |

| Class 3 | [15:12] | [11:9] | [8:6] | [5:3] | [2:0] |
|---|---|---|---|---|---|
| | opcode1 | Predicate | opcode2 | opcode3 | opcode4 |

| Shift by Immediate | [15:12] | [11:9] | [8:6] | [5] | [4:0] |
|---|---|---|---|---|---|
| | opcode1 | Predicate | rX | direction | ShiftAmount |

| Branch | [15:12] | [11:9] | [8] | [7:0] |
|---|---|---|---|---|
| | opcode1 | Predicate | Link? | Signed Offset |

| PUSH, POP, RETURN | [15:12] | [11:9] | [8:6] | [5:0] |
|---|---|---|---|---|
| | opcode1 | Predicate | opcode2 | r0 – r5 |

| MOVE | [15:12] | [11:9] | [8:6] | [5] | [4:0] |
|---|---|---|---|---|---|
| | opcode1 | Predicate | rX | Next? | value |

Figure 3.5: SpotCore instruction set formats

POP and PUSH instructions read from and write to the stack respectively. The RETURN instruction is similar to the POP instruction with the only difference being the fact that it also loads the PC with the preserved link register value. In the implementation of the RETURN instruction, the reloading from the stack is performed in parallel with branching. In these stack manipulation instructions, the bit field [5:0] is used in a flexible

way to encode an arbitrary set of registers which must be stacked.

## 3.3 Results

We have seen that since the energy usage of sensor applications is a product of power and time, it is important to reduce both the power consumption and the execution time of a set of instructions.

The Verilog design was synthesised using a speed-optimised UMC 130nm technology library ($UMC\_L130E\_HS\_MMRF$). The power estimate obtained was 0.03mW/MHz and the area estimate was $0.08mm^2$. The synthesis procedure involved physical compilation, clock-tree synthesis, routing and post-routing using a tool known as Astro from Synopsys [109]. The critical path length indicated that the design could be clocked at up to 130MHz. Conservative timing constraints — input and output delays, and load capacitances were selected. However, there is still a lot that could be optimised, such as the way the register file was synthesised. It was synthesised directly from the Verilog description as flip-flops. A more efficient method would rely on a memory generator which would be better suited to the cell library used. There was no on-chip debug hardware and scan chain insertion was not performed. The synthesis was performed for "typical" operating conditions — 25 degrees Celsius and 1.2V.

The power figure looks auspicious compared to the TI MSP430 (0.4mW/MHz). The TIMSP430 is a complete System-On-Chip comprising memory and other peripherals (watchdog, timer, UART etc), but the data provided is for TI CPU when it is operating alone with all the peripherals powered down. The more lightweight embedded ARM processors — ARM7TDMI and the ARM Cortex M3 have power figures of 0.06mW/MHz and 0.14mW/MHz (130nm technology and speed-optimised) respectively [3]. The area-optimised ARM Cortex M3 is $0.38mm^2$. The Cortex-M3 [98] implements a new 16-bit variant of the ARM instruction set known as Thumb-2 which is capable of improving code density while maintaining a high level of performance.

If the processor was synthesised without the hardware structures required by the loop instruction, the area estimate dropped by 15% and the power figure saw a reduction of 16%. Thus depending on the application, the inclusion of the loop instruction can be viewed as a great benefit in terms of improving performance or a loss due to the additional hardware complexity.

Figure 3.6 is a snapshot of the full SpotCore processor layout.

Figure 3.7 shows the code size and execution times of different processors running the same digital filter algorithm on 4000 input samples, normalized to an operating clock frequency of 1MHz.

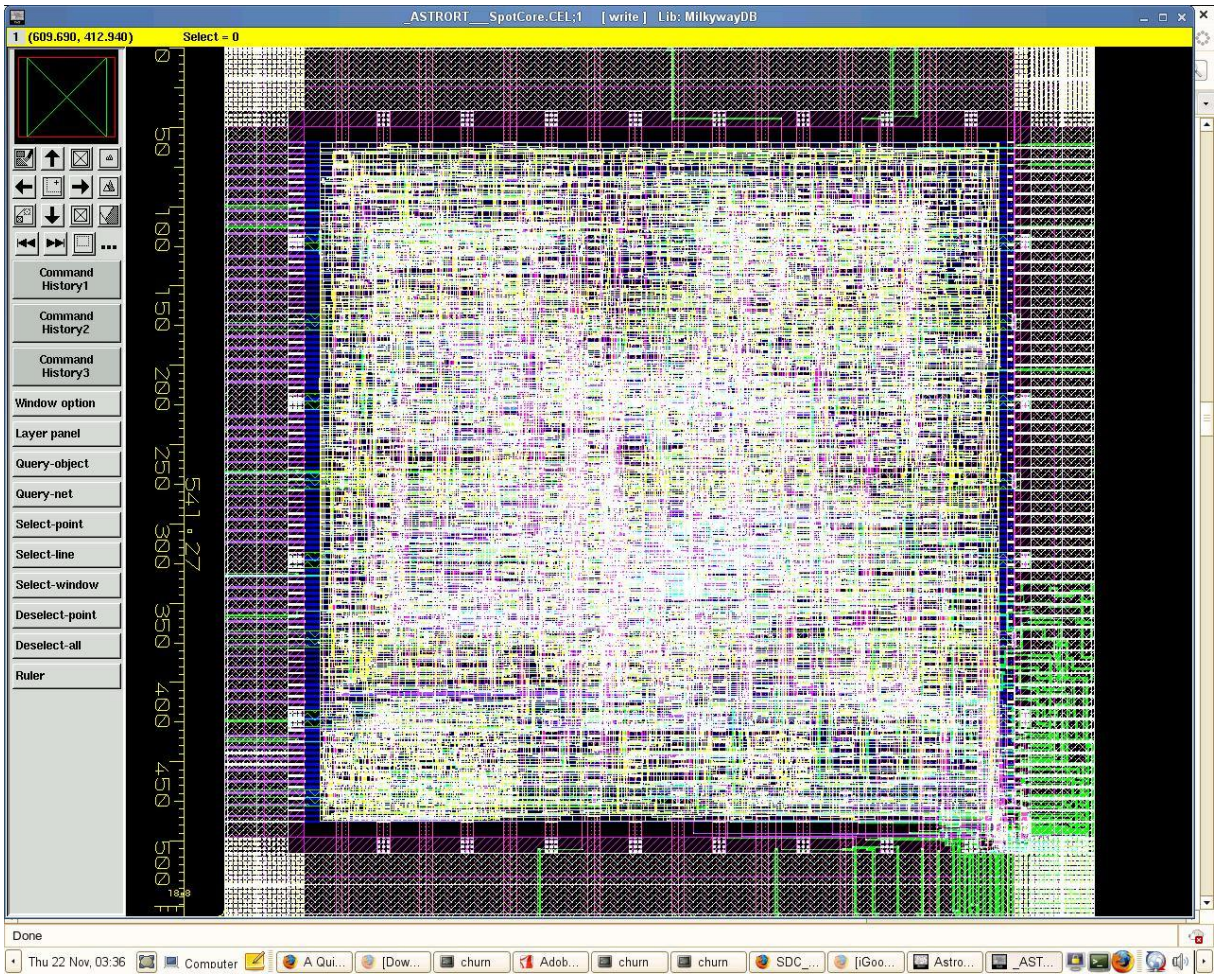


Figure 3.6: SpotCore processor layout

| | Execution time (ms) | Code size (bytes) |
|---|---|---|
| SpotCore | 20.2 | 50 |
| ARM7TDMI | 23.5 | 92 |
| TI MSP430 | 38.5 | 95 |

Figure 3.7: IIR filter results

The performance advantage of SpotCore in this Infinite Impulse Response (IIR) filter experiment is largely due to its ability to know precisely where branches within loops occur; and this improvement is significant for a large number of programs as loops are very common programming constructs. In addition, judging from Figure 3.3, about 70% of the instruction set is ARM compatible which is significant as there exists a wealth of reliable benchmarks for that instruction set. While many of the instructions are also similar to those in the TI MSP430 instruction set, it is clear that SpotCore gained a definite performance advantage because the core supports dyadic instead of monadic data-processing instructions. In addition, the TI MSP430 does not support direct multiplication within the processor datapath but relies instead on a system peripheral which limits performance because a data access is required.

When code performing the matrix multiplication operation was tested on the SpotCore processor and on an ARM7TDMI, the SpotCore processor was more than 18% faster due to its loop-handling capabilities.

## 3.4 Summary

In this chapter, we have applied a selection of low-power CPU design strategies to develop a highly-optimised processor design which can meet performance goals in a power-efficient manner. A 48% improvement in the execution time of an IIR filter routine was observed relative to the TI MSP430 which is widely used on sensor platforms, and a 14% improvement was obtained relative to an ARM7 processor. 16-bit ARM Thumb code normally achieves around 75% of the performance of a standard 32-bit ARM but has much better code density [3]. While this benchmark may seem contrived the structure of the IIR filter is such that it tests the capacity of the CPU's arithmetic, logic, and the effectiveness of its data access mechanism. It is also a very popular DSP algorithm and is found in many standard benchmarks [114]. The benchmarking process is notoriously tricky [53] and is even more so in the case of SpotCore as there is no C compiler for it as yet. A larger example is presented in Chapter 5 and the SpotCore is used in a real-time system in Chapter 6. The synthesis results prove the extremely lightweight nature of the design; and coupled with reduced execution times, it can enable significant energy-savings to be made in the realm of sentient computing. A way of maximising the encoding space by having a very orderly subdivision of the instruction field was presented. One can leverage the small size, performance, and code density of the SpotCore CPU to build multiprocessing hubs which will take advantage of the high degree of data-parallelism inherent in sensor networks. It is clear that such lightweight processing elements will form the cornerstone of scalable sentient computing.

# Chapter 4

# TopDog Scheduling

This chapter describes the implementation and evaluation of several building blocks which are crucial to efficient multiprocessing while maintaining flexibility. It is shown that the solutions scale well with the addition of multiple processors. The key ingredients for effective multiprocessing are:

- Robust programming model to manage concurrency
- Real-time scheduler and synchronization mechanism
- Power-efficient interprocessor communication mechanism
- Enhanced debugging - ability to find race conditions and deadlocks
- Memory protection

## 4.1 The Role of the Scheduler in Explicit Parallelism

There are many different models of concurrency or ways of expressing parallelism in computer programming. In general, parallelism can be extracted from standard sequential programs by identifying a set of different tasks which can operate on largely disjoint blocks of data or a single task which operates on many different blocks of data. Advocates of implicit parallelism argue that it is possible to build compilers which would analyse problems expressed as sequential programs, and automatically decompose them and allocate the resulting subtasks to separate execution units. These are known as *parallelizing* compilers and their main job is to identify dependencies, enforce proper execution ordering and schedule tasks. There are four main kinds of dependencies:

1. Flow dependency - when a variable must be written by one thread before being read by another

2. Antidependency - when a variable must be read by one thread before being written by another

3. Output dependency - when a variable is written by more than one thread

4. Control dependency - when the execution of one thread is dependent on the result of the execution of another

A violation of any of the first three breaks the data dependency model of the program causing race conditions and non-deterministic execution. This analysis is made simpler if all memory references are static. Thus in any given code region, groups of expressions can be scheduled for concurrent execution if they do not read any variable which is written in another group or write any variable which is written or read in another group. In addition, an expression group can proceed concurrently with other groups only if it is not reached by a conditional branch from one of the other groups.

Herein lies the difficulty in automatic parallelization - memory references are not always static; many programs make use of pointer-based memory transfers and the interactive nature of some programs, with interfaces to real world and/or other programs, make demands which force the compiler to produce sub-optimal concurrent programs by establishing fake dependencies due to strict adherence to the aforementioned rules.

Sieve [73] attempts to strike a balance by using declarative concurrency in which the programmer specifies a "sieve" block within which side-effects are delayed. This makes it easy for a compiler to perform dependency analysis and partition the program automatically.

However, the implicit approach can be useful when one wishes to avoid rewriting legacy sequential programs, but it is likely that certain interesting problems will eventually need to be rewritten in order to take advantage of emerging parallel algorithms rather than reverse engineering the existing sequential algorithms. As a result, the approach in this thesis relies on explicit parallelism where the programmer is responsible for the specification of concurrency and generally understands the dependencies within a program and the required synchronization. The threading API presented in this chapter reduces the programmer's involvement in mundane tasks like creating or managing threads in a safe manner, and sharing work efficiently across multiple cores. The main focus of the parallel programmer should be specifying *concurrent sequential blocks of execution* and the multiprocessor hardware should be able to handle the extra details.

Another alternative to explicit parallelism is to use a functional programming language. If a programming language is purely functional, side-effects are not allowed. This means there can be no modification of any globally accessible state. This leads to a great

deal of modularity, implicit control-flow, and an additional feature loved by many programmers - conciseness. The absence of side-effects means the execution order of functions can purely arbitrary and concurrent execution is a natural extension. In practice, functional languages might adopt either a lazy evaluation policy or a strict (greedy) evaluation policy. Lazy evaluation means the program is demand-driven; that is the computation is actually performed only when the result is required and this is the strategy used in Haskell. Strict evaluation is data-driven and an expression is evaluated only when input data are available. While functional languages provide a way of producing parallel programs which are correct-by-construction it is unlikely that they will play a major role in the early stages of parallel computing as designers transition from sequential programming. This is chiefly because it is not possible to reason about, predict and guarantee performance in a straightforward manner in a functional programming environment. In addition, it is possible to replicate some of their desirable features by limiting the amount of shared state in imperative programming languages either by restricting the style of the programmer or by prohibiting certain memory transactions at runtime.

The role of the scheduler within an operating system or kernel is to manage access to the CPU so each thread of execution can run for a certain time period before being replaced by another. This is done primarily to maximise throughput and minimize latency of interrupts (that is, increase responsiveness and availability) rather than purely for performance reasons. The performance of any given thread is reduced because it must share the CPU, and the overall utilisation of the system is limited by the time required to perform context-switches.

As a result any scheduler has two main goals —

1. Give reliable Quality-of-Service (QoS) guarantees (notably latency, and execution time) to applications

2. Minimize context-switching time

In the case of a scheduler within a multiprocessor, it is desirable to perform these actions in a scalable manner across multiple CPUs.

In the multiprocessing architecture presented in this thesis — the thread management and scheduling are centralized and operate in hardware as this is more energy efficient for a small number of cores than a distributed software scheduler. The centralized approach also facilitates debugging as synchronization is performed in one place. To avoid any bottlenecks due to the centralized architecture, the different parts of the scheduler are designed to operate concurrently, which is natural in hardware.

## 4.2 The Case for Balanced Loads

Load-balancing is important in order to maximize utilization. It refers to ensuring that all tasks are scheduled so that the overall work the multiprocessor must perform completes as quickly as possible — that is no processors are unnecessarily idle. In heterogeneous multiprocessing, the processors all have different capabilities and may run at different speeds thus exacerbating the problem of load-balancing. One would have to be careful not to allocate a heavy task to a weak processor and a light task to a more capable processor. Even if there exists a mechanism for moving the heavy task to the more capable processor, determining this fact at runtime is extremely difficult and shuffling tasks accordingly requires a huge overhead. However, in a homogeneous environment in which all cores have the same functionality and operate at the same clock frequency, the problem of load-balancing is simplified as long as the tasks are all of the same size. If one task requires a longer execution time than other tasks, load-imbalance may exist as depicted in Figure 4.1, if the tasks are scheduled incorrectly; leading to an overall execution time which is sub-optimal. In the figure, Scenario 1 represents the uniprocessor case which has the longest completion time. Scenario 2 uses two processors but is poorly balanced while Scenario 3 represents the ideal scheduling arrangement. It is important to note that in all three cases the energy usage is roughly the same assuming the processors are identical and that the context-switch time is minimal. In practice, this is achieved by gating the clock which feeds any core which is not in use. Power-gating which shuts off the supply current to any core which is idle could also be used and is more effective. However, there will be a small delay when the core is powered back on which might hamper performance [65]. In fact, the energy usage in the multiprocessor might be slightly less because there is only one context-switch as opposed to two context-switches in the uniprocessor case.

In the example in Figure 4.1, it is possible to force the scheduler in this architecture to produce the optimal arrangement shown in Scenario 3 by setting the priority of task f3 higher than that of task f2 and task f1. Alternatively, tasks f2 and f1 may be combined into a single task, or a synchronization constraint may be created so that f1 must run only after f2 completes leaving P1 free to run f3. However, one might not know the actual execution time of f3 so it is necessary perform some profiling in order to validate or correct the initial guess.

## 4.3 Selecting a Concurrency Model

The fork-join model is widely considered the most flexible way of specifying logical parallelism in a program. This means it does not place restrictions on the start and endpoints of

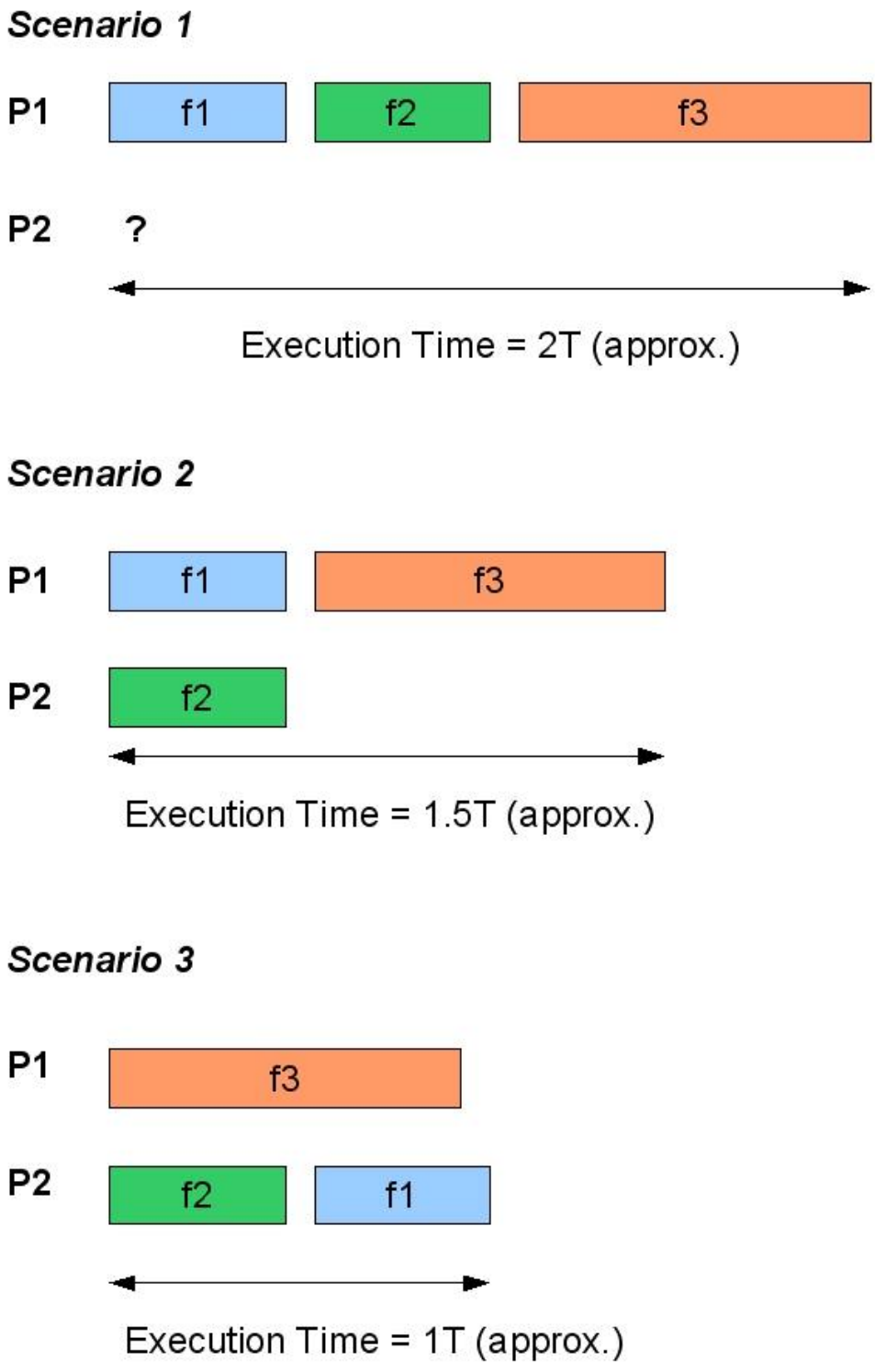


Figure 4.1: Scheduling order affects speedup

parallel tasks within a program. This thesis argues that for this model to be scalable, the implementation must have hardware support for lightweight task creation, switching and synchronization. Such lightweight task creation enables a much finer granularity in the parallelism within the program so that small code sequences can be easily and efficiently composed into threads of execution.

Consider the program with a set of tasks - a,b,c,d,e,f,g,h,i,j. Due to possible data and control dependencies there will be a strict temporal ordering which must be adhered to by the runtime environment. This leads to a representation as a *directed graph* where the edges are separate tasks and the vertices are points in the program at which either a *fork* operation or a *join* operation is performed. The implementation uses the following assembly-level commands which can be summarized as follows—

FORK <threadID>; to start/schedule the execution of a separate thread

SIGNAL <signalID>; decrement counter pointed to by <signalID>

WAIT <signalID>; blocks or suspends a thread until a counter pointed to by <signalID> reaches zero due to an event or signal generated within the system. It then continues execution at the instruction after the WAIT instruction.

CHECK — similar to wait but non-blocking

SET_SIGNAL <signalID>, <value>; reset the counter associated with a signal.

EXIT ; delete all state associated with a thread

This somewhat simplified version of the hardware-level API is used here for clarity. A full description is provided later.

The join operation is performed in practice by using SET_SIGNAL to initialize the counter associated with a given signal, and then using a WAIT command to block a specific thread until all the relevant threads have indicated that they have reached a certain point (or simply completed execution) by issuing a SIGNAL command. Figure 4.2 illustrates this.

## 4.4 Critical Analysis of Concurrency Models

There are other models of concurrency which though less flexible can be sufficiently useful for some programs. Most of these models rely on the concept of a parallel section. Unlike the fork-join model introduced earlier - threads of execution cannot be created and synchronized at arbitrary points but all concurrency must begin and end within a block of code specified by the programmer. Almasi and Gottlieb [14] noted that the lack of

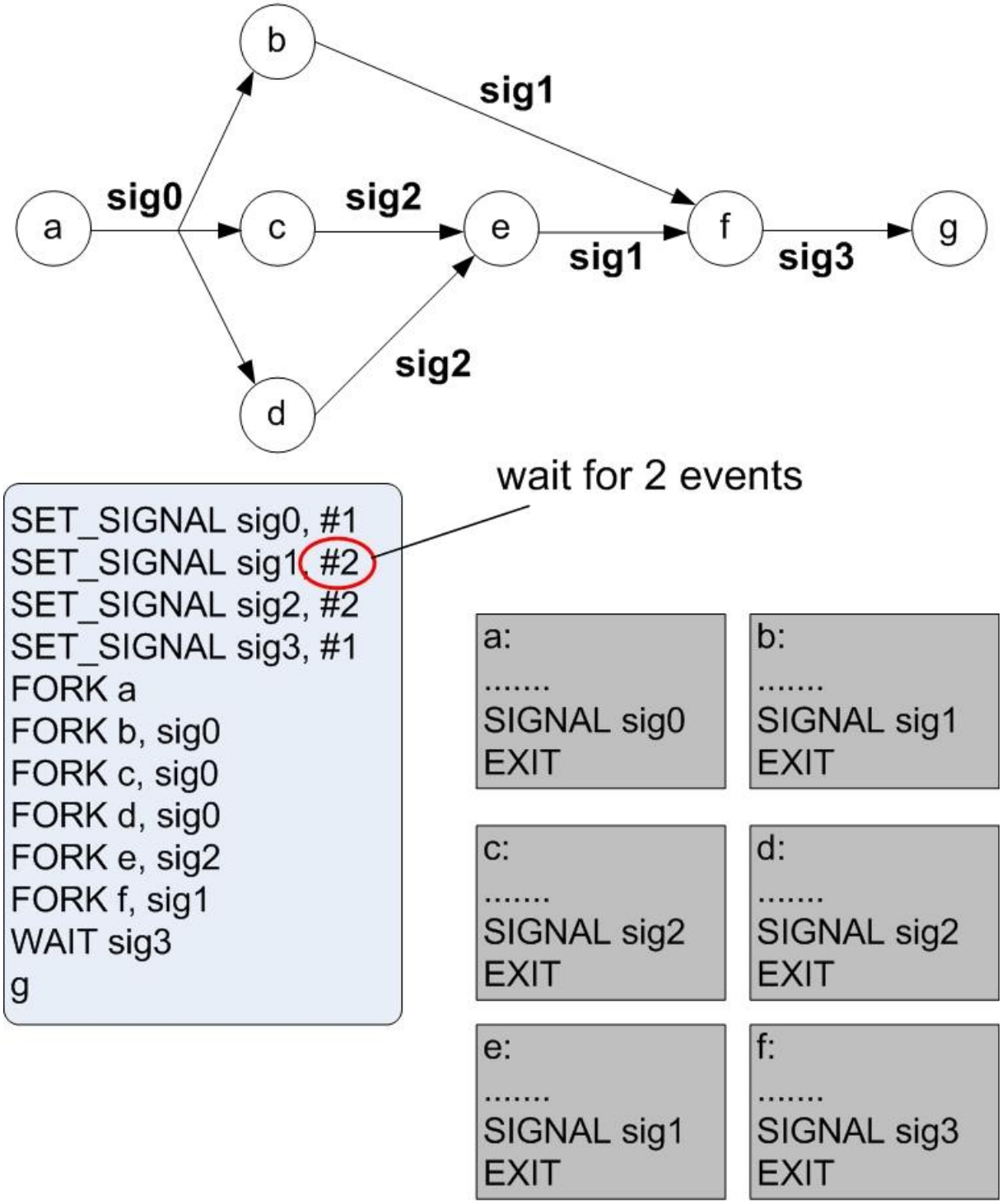


Figure 4.2: Task graph and associated SpotCore code

flexibility is seen in the fact that parallel sections cannot be used to implement the task graph in Figure 4.2. Variants thereof are not uncommon in mathematical processing with many complex dependencies between tasks.

"Communicating Sequential Processes" (CSP) [55] is a theoretical framework or language for describing concurrent systems in an abstract and fundamental way. In CSP, processes exist independently and operate concurrently but may interact or communicate at various points. These concepts which coupled processing with some form of communication led to the development of Occam [101]. The mathematical rigor provided by CSP showed that race conditions could be avoided and this desirable characteristic was implemented in Occam through synchronized communication and disjointness. Occam programs are constructed from a collection of processes which communicate via values passed on unidirectional channels. The Occam language introduces the PAR construct which allows a group of processes to be specified as capable of being run concurrently.

```
PAR
    A
    B
    C
```

In the example above processes A, B, and C execute concurrently and execution or control moves beyond the code block only when all the parallel processes have completed. This is similar to the constructs used by Dijkstra [35] to mark the start and end of concurrent blocks known as "parbegin" and "parend" respectively.

To preserve scheduling invariance, the Occam language dictates that a channel may not be used as output by more than one process in a parallel block.

Occam also has the concept of guarded parallel execution referred to as alternation. The ALT construct is used to combine processes together such that only one process may execute at a time — the one whose guard condition evaluates to TRUE. However, if more than one process is ready the execution is non-deterministic. This could also have been achieved by placing the conditionals within the process implementations of the processes in a PAR block and causing each process to exit immediately if a specific condition was not satisfied. Of course one will then lose the constraint that exactly one process executes even if multiple conditionals are satisfied. In addition, it is possible that allowing the guards to be exposed explicitly facilitates optimizations. Recognizing the importance of this optimization led to the development of a variant of the FORK instruction used in SpotCore which ensures that a thread is not started before the check whether it is ready to run is performed as this can be wasteful. Another benefit of alternation is that the explicit locking, necessary when many independent execution flows need to access shared data, can be avoided.

Another language which uses parallel sections is OpenMP. OpenMP [88] enables the creation of concurrent programs in C, C++ or FORTRAN by providing the programmer with compiler directives (pragmas), library functions, and environment variables. The threads thus created communicate via a shared memory space. OpenMP performs "work-sharing" unlike another notable parallel programming language known as Cilk [47] which performs "work-stealing". In work-sharing, a master thread spawns off several threads (team) which execute in parallel and are synchronized implicitly at a "barrier" or end of the parallel section. The master thread then continues beyond the barrier and the other threads cease to exist. In the Cilk implementation of "work-stealing" when a processor encounters a "spawn" command it suspends its current task, stacks its state on a double-ended queue (deque), and executes this new child task. An idle processor can then remove "steal" and execute the tasks stacked on the deque, which the busy processor might eventually work on but cannot do so at the current instant. Cilk uses a "microscheduler" to manage scheduling across a working group of processors and a "nanoscheduler" for managing procedures within a single processor.

Advocates of work-stealing claim it is more efficient in terms of communication bandwidth than work-sharing algorithms as communication is initiated only when any processor is idle and not between a succession of threads. There is no centralised task pool so contention over a common region of memory is avoided. However, the platform described in this thesis implements work-sharing because work-stealing leads to more context-switches than necessary, that is, at a FORK, a processor P1 does a context-switch to the task being FORKed, and another processor P2 must resume where P1 left off by reloading the abandoned processor state of P1. In the work-sharing scenario, processor P1 continues to execute instructions after the FORK without any context-switch while an IDLE processor may load the FORKed task. The concurrency within the TopDog hardware mitigates the effect of centralized management.

OpenMP has specialised constructs for executing several iterations of a "for loop" in parallel but the programmer has no real control over the mapping of threads to actual execution units, that is, in OpenMP the programmer cannot specify how specific threads may be bound to processors. While this is useful because it enables incremental parallelism, it makes fine-tuning performance more difficult. Both Cilk and OpenMP have mechanisms for locking variables to avoid race conditions. OpenMP has wide industry support and is currently the standard for shared-memory parallel computing as it is relatively straightforward to obtain sufficient speedup for a small number of cores. For a larger number of processors which do not share memory, most parallel programmers use MPI (Message-Passing Interface) [90] which is an API specifying how processes in a parallel

program communicate by explicitly exchanging data using "send" and "receive" library functions.

Note that the parallel programming environments described above operate at a fairly high level and need some underlying operating system support or threading library (for example Cilk relies on POSIX threads). POSIX threads (Pthreads) refer to the standardized threading API for UNIX systems. The programming interface is specified by the IEEE POSIX 1003.1c standard [27]. Creating threads is less expensive than creating UNIX processes as processes have a lot more state (information about program resources) and are isolated from other processes. On the other hand, threads in a single process share the same address space and only retain a little information such as the state of the processor registers and thread priorities. The Pthread library provides the following core multithreading primitives:

- Thread management - creation and termination of threads. Arguments can be passed to threads when created.
- Sequencing threads via joining. The "pthread_join(tid)" function causes the thread invoking it to wait for the thread with the specified ID (tid) to terminate. If one examines this API carefully, one can see that it is not very efficient because if one needs to synchronise a large number of threads there will be have to be many successive calls to "pthread_join(tid)", and the thread invoking it will block, resume, and block again ad nauseam. This inefficiency is avoided in the DAPH by using more powerful constructs for joining.
- Mutual exclusion - provision for the creation of critical regions by using mutex variables (locks).
- Condition variables - these are used when it is desirable to make a thread remain blocked until a specified condition such as a locked variable reaching a certain threshold is satisfied. A signal from one thread causes the waiting thread to be awoken. The implementation relies on mutexes.

## 4.5 Multiprocessing Primitives

From the preceding discussion, it can be established that the essentials of any concurrency model or parallel programming framework are rapid task management, flexible task synchronization and efficient mutual exclusion. These considerations led to the development of a hardware design for managing concurrency within a multithreaded environment which can be extended over many cores. This hardware module is referred to as the "TopDog".

In addition to building low-energy processor cores, optimising the manner in which threads are loaded and removed, and the associated scheduling scheme, maximises processor utilisation and improves energy efficiency. The problem of scalability in a multi-threaded environment is addressed through the development of a novel scheduling algorithm implemented directly in hardware. An effective thread management strategy should also scale with the number of threads and processing elements.

Due to the small physical area of SpotCore it is envisaged that it will be used not only in a multithreaded environment but alongside other cores in a multiprocessor; and in this case it is desirable to have transparent thread migration between cores. The TopDog shares a connection with the processor memory and interrupt interfaces, and can dispatch threads to the processor based on its internal scheduling algorithm. This module elevates the level of performance possible as the processor does not have to keep switching to some kind of supervisory mode in order to check the status of other threads. It also improves scalability in a system comprising multiple cores (Figure 4.3) by providing a common, fast arbitration mechanism. This is applicable in situations where a shared bus is feasible such as symmetric multiprocessors with up to about 16 cores.

In summary, the TopDog carries out the following tasks:

- Fast creation, switching and deletion of threads
- Guarantees atomicity of operations so race conditions associated with thread maintenance are avoided
- Stores thread control blocks (TCB) for different threads and can modify each via simple instructions issued from the processor
- Synchronisation of threads through the provision of a signaling layer
- Implements a scheduling algorithm with QoS considerations from the ground up
- Holds interrupt vectors and priorities, and uses a common CPU access mechanism for interrupts and other threads

The two basic values which must be set before a thread is created are:

1. A vector or pointer to the piece of code which will be executed when the thread starts.
2. A pointer to a region of memory where the thread stack will begin.

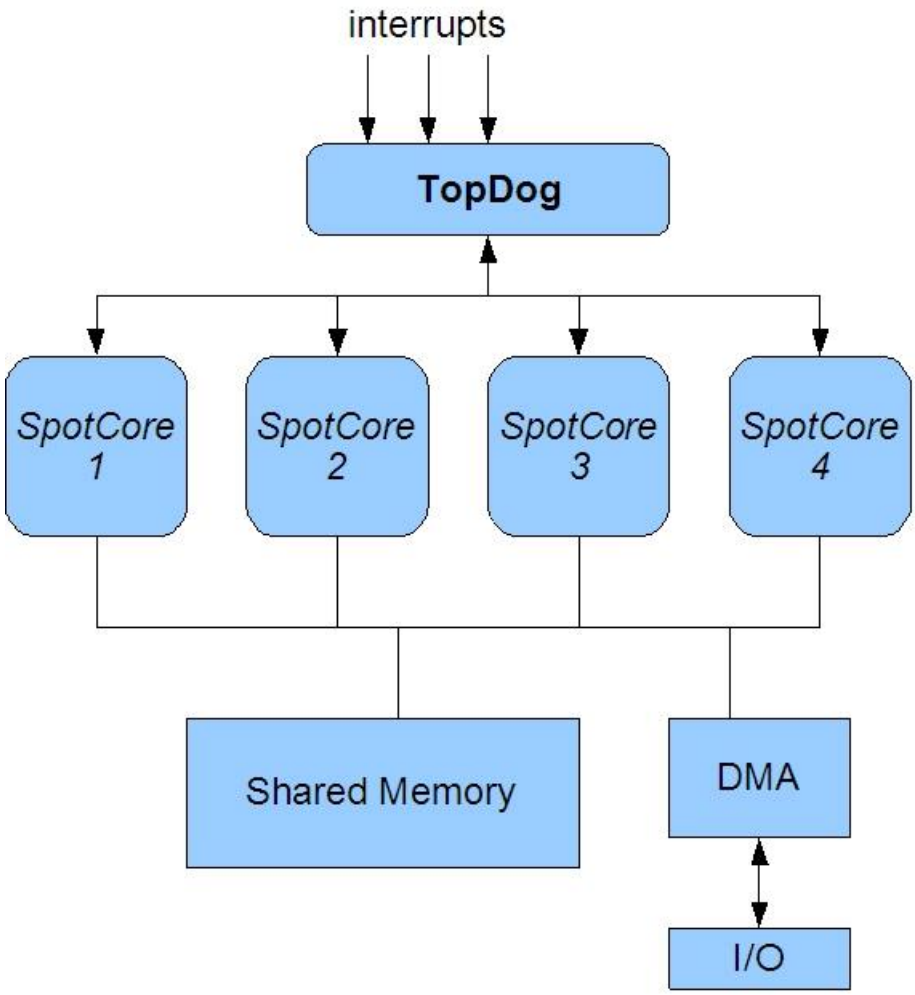


Figure 4.3: Multiprocessor comprising SpotCores and a TopDog scheduler

These are set before FORKing a thread by using the SET_VECTOR and SET_TSP instructions respectively. To load a thread on a SpotCore processor, the Program Counter and Stack Pointer are loaded with these values and they are saved in the TopDog if the thread is usurped. Any state that an interrupted thread contains such as the general-purpose registers, Program Status Register and Loop State is stored on the stack automatically.

After setting the required pointers the FORK instruction activates the thread state machine in the TopDog that puts the thread in a "READY" queue. A variant of the FORK instruction is provided to handle the situation where we do not want a thread to start immediately but to wait for some signal to become valid first. This equivalent to using the ordinary FORK instruction to create a thread whose first instruction causes the thread to block until a condition is satisfied. By specifying the condition while FORKing we can avoid unnecessary context switches.

When a thread has no useful work to do, it can issue a YIELD instruction which causes it to be removed from the CPU. A thread issuing the EXIT instruction removes all active references to itself from the TopDog data structures.

Thread synchronization is achieved by a set of hardware counters within the TopDog. Dedicated memory blocks local to the TopDog are used instead of the system memory for performance and efficiency reasons since they are accessed and updated frequently. In order to perform a JOIN operation on N threads, a counter is initialized to N using the SET_SIGNAL instruction before the threads are FORKed. Each thread then issues a SIGNAL instruction to the TopDog with an associated signal ID. The corresponding counter in the TopDog is decremented whenever a SIGNAL operation is performed. When the counter reaches zero, an event is generated and propagated to any waiting threads. Any thread which needs to start or resume only when a certain number of threads have all reached a specified pivotal point or synchronization barrier must issue an explicit WAIT command or be FORKed into a wait queue.

A non-blocking variant of the WAIT command called CHECK is also provided when the programmer needs to track the value of a signal counter. Apart from being used to synchronise a group of threads in the fork-join model of parallelism, the SIGNAL and WAIT instructions provide a flexible way of enforcing temporal order in the actions of any two threads when that is needed at arbitrary points. The semaphore introduced by Dijkstra [35] is widely recognized as one of the most fundamental primitives for implementing synchronization and mutual exclusion in concurrent programs. According to Dijkstra, there are 2 operations which may be performed on a semaphore or a special integer variable (s) whose initial value is non-negative.

P(s) : decrements s and atomically checks whether s is less than 0, blocking the calling thread if it is.

V(s) : increments s and atomically checks whether s is less than or equal to 0 and resumes a blocked process waiting on that semaphore if it is.

If s can take only 2 values — 0 or 1, then it is known as a binary semaphore which behaves just like a mutex or lock which we encountered earlier, and can be used to create a critical section. The SpotCore instruction set provides two instructions - LOCK and UNLOCK which have the same functionality but can also behave as a true counting semaphore whose importance will be demonstrated shortly. The SIGNAL and WAIT instructions are similar to the semaphore described formerly in that each update is atomic and an event is generated when the count value reaches zero. WAIT is like P(s) without the decrement operation, blocking if count is not zero. SIGNAL is similar to V(s) in that it performs an atomic check and can cause threads to be resumed, but it decrements instead of incrementing.

This modification of the semantics of P(s) and V(s) was necessary in order to produce an efficient implementation for thread synchronisation. This is because the essence of a synchronisation construct is being able to communicate the arrival of one or more threads at a pivotal point to another thread or a group of threads. This requirement does not map easily onto the given semaphore specification because in typical usage, threads may use P(s) to block or suspend themselves if a semaphore is unavailable and V(s) must be issued by threads which previously executed P(s) and were allowed to proceed but no longer require a restricted resource. This is so that another thread may be awoken. However, for the purpose of synchronisation, we can capture the necessary semantics with a primitive based on only a single count direction and a predefined threshold. When this primitive is applied to any scenario involving threads, it is sufficient to perform a "join" operation. Not having to count in both directions and being able to wake all blocked threads leads to a more straightforward and powerful implementation as it combines forking new threads with the last join.

When an event signal is generated, the implementation wakes up threads in a priority-based order rather than the order in which they arrived in the WAIT queue, that is, unlike the First-Come-First-Served (FCFS) policy implemented by what is referred to as a "strong" semaphore. A "weak" semaphore has no ordering.

The original semaphore implementation is especially useful when we wish to restrict the number of threads performing a given operation. The case where we want only a single thread to proceed gives the simplest case of a critical section. However, in some cases one may want a larger number of threads to access a given resource but such that a predefined limit on the number of concurrent threads accessing the resource is not

exceeded. The semaphore is initialized to the maximum number of threads permitted to execute concurrently and each thread then performs P(s) to gain access and V(s) when the resource is no longer required. As a result any thread trying to gain access when the semaphore value is zero or negative will block. When the semaphore becomes positive again as a result of a V(s) by a thread in the "limited concurrency" section, one of the waiting threads will be resumed.

Due to the transformation of the semantics which was introduced earlier for hardware efficiency, this "limited concurrency" section cannot be implemented efficiently with SIGNAL and WAIT instructions despite having some similarity to P(s) and V(s), and the ability to perform explicit synchronisation of multiple threads. Instead the operation of the LOCK and UNLOCK primitives is such that if the lock variable is initialized by issuing a SET_LOCK command, more than one thread can acquire the lock. Using locks in this mode is not recommended as it means the very important concept of single ownership no longer exists. In addition, the Deadlock Detection Engine which is introduced later does not analyse locks with multiple owners for efficiency reasons.

## 4.6 QoS-Aware Scheduler

The central innovation in this chapter is the development of a thread management policy which runs directly in hardware without requiring any CPU time unlike conventional operating systems. Rather than leaving the scheduling decisions entirely up to the operating system, the programmer can specify certain parameters to the TopDog to enable it to achieve the right level of Quality-of-Service (QoS). This development was inspired by Nemesis [94] which is an operating system designed to provide applications (such as multimedia applications) which are very time-sensitive with some form of QoS guarantees required with respect to CPU and I/O resources. The scheduling decisions in TopDog are based on the following three parameters which achieve the appropriate balance between ease-of-use and robustness — PRIORITY, ONTIME, and OFFTIME.

These parameters relate to the thread state diagram in Figure 4.4 and their utility is explained as follows. At the most elementary level, if threads are of the same priority, they gain access to the processor core based on a First-Come-First-Served (FCFS or simply FIFO) scheme. There are 16 priority levels and the TopDog will remove lower priority threads from the processor so a higher priority one can run. Unfortunately, this might very easily lead to starvation of some threads if there are many high priority threads. As a result, the system designed allows the programmer to specify a maximum "ONTIME" and a minimum "OFFTIME" for each thread.

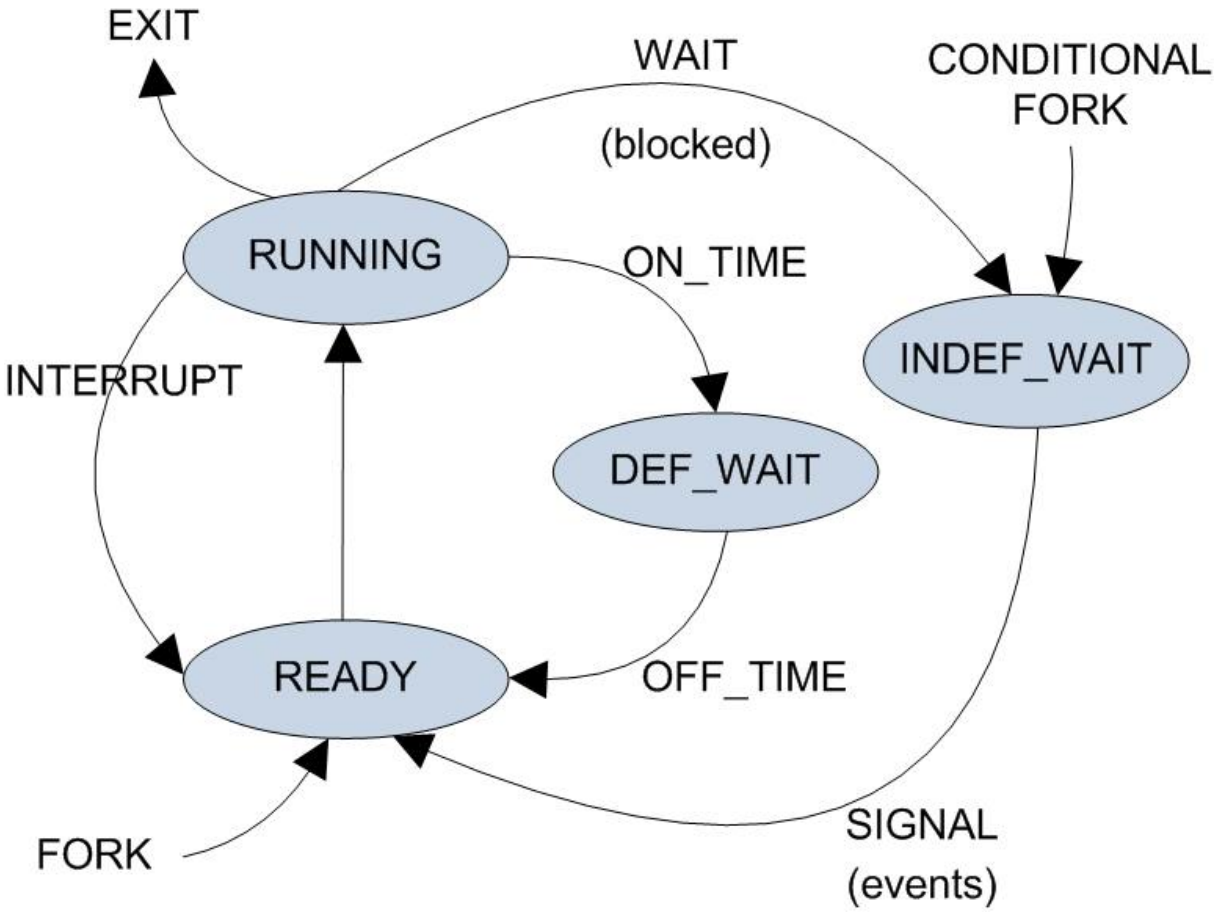


Figure 4.4: Thread state diagram

Together with the priority value, they can be used to fine-tune performance because the priority value controls how quickly the thread gets to execute when it is in the READY state, the ONTIME controls how long it is allowed to spend on a processor, and the OFFTIME determines the delay between getting preempted at the end of its execution time, and returning to the READY state again. The scheme has enough flexibility to support a very broad range of CPU access schemes without resorting to a high-level thread library which is likely to be slower than the hardware implementation because it might not have as much concurrency or have low latency access to requisite data structures.

While the hardware does not guarantee that thread starvation will be avoided, it does provide the sensor system programmer with more scope for controlling thread scheduling than is currently available.

The effect of the ONTIME and OFFTIME parameters is to ensure that the execution probability of any thread does not have a strong dependence on the number of threads of a higher or the same priority because the high priority threads have to back off periodically. One good policy for fairness would be to ensure that low priority threads have more ONTIME and less OFFTIME, while high priority threads have less ONTIME and more OFFTIME.

## 4.7 Scheduling in Energy-Constrained Environments

Let us now compare the TopDog scheduler with those commonly found in resource-constrained environments. MicroC/OS-II is a pre-emptive real-time kernel written in C, which runs on embedded processors such as the Motorola 68k, ARM7, and Altera Nios II [70]. It supports dynamic priorities but no two tasks (threads) may have the same priority. The highest priority thread always runs but may be superseded by an interrupt service routine. Its main drawback is that a low priority thread can wait for an arbitrarily long period of time since the highest priority thread must run to completion or get blocked before it can run. It is believed that this approach does not scale well because fairness is not integral to the operating system and it is harder to give the lower priority thread any QoS as the execution times of many higher priority threads are indeterminate. The TopDog scheduler avoids this undesirable scenario by providing the ability to control the dominance of higher priority threads in a direct way. The uC/OS-II kernel code occupies about 2K bytes and can consume about 5% of CPU time [70]. It can support up to 255 tasks. While the TopDog scheduler has fewer priority levels, it can support multiple threads of the same priority and can manage up to 256 threads.

TinyOS is an open-source embedded operating system developed at University of California Berkeley and is very popular among developers of applications for Wireless Sensor Networks. It operates on many different platforms, speeds development, and is useful for testing research ideas. It is written in nesC which is a C-like structured component-based language. TinyOS can perform the standard functions of task scheduling and interrupt handling, and it has an event-driven architecture which is sufficiently abstract to enable the creation of cross-platform applications while remaining lightweight. Unfortunately, TinyOS provides only an elementary concurrency model with limited operating system support for a large number of threads or a platform with more than one processor. It has no inherent ability to specify multiple priority levels. The two main system threads comprise tasks and hardware event-handlers respectively. Tasks must run to completion and cannot preempt other tasks while they may be preempted by hardware interrupts. The TopDog approach differs from this by allowing threads in the system to pre-empt other threads regularly and by allowing the programmer to specify QoS constraints in an explicit manner.

Another operating system layer for sensor processing known as Contiki builds on the event-driven model by utilizing "protothreads" — lightweight threads which can facilitate multithreading. Since each protothread does not need its own stack, protothreads have been promoted as ideal for memory-constrained systems. The conditional blocking wait abstraction provided by protothreads avoids the complexity involved in dealing with explicit state machines which is common when developing software for event-driven systems. The TopDog gives threads the ability to block until a condition variable becomes true or an external event of interest (interrupt) occurs. Thus, writing software using the TopDog is more scalable as there is no overhead in terms of CPU time or code size and we can also reap the benefits of hardware acceleration of the scheduling algorithm.

## 4.8 Towards Estimating and Maximising QoS

Rather than communicating with the TopDog module as a memory-mapped peripheral on the system bus, special instructions were created to speed-up access and promote flexibility with different memory architectures as no pointers have to be calculated.

The SpotCore instructions which are used for setting QoS requirements are:

- SETPRIORITY — this specifies a 4-bit priority value for the thread being created
- SETONTIME — sets the parameter ONTIME (10-bit value)
- SETOFFTIME — sets the parameter OFFTIME (10-bit value)

The state diagram shown in Figure 4.4 is implemented by multiple memory blocks and a few associated logic controllers. The controllers were simple enough so that the hardware footprint of the TopDog module is small. The key to the fast inter-thread communication and synchronisation mechanism was designing the logic which maps events to thread IDs and that which performs priority analysis on the READY queue, to operate as concurrently as possible.

There are four core memory modules in the TopDog implementation which can all be accessed independently. Their sizes are configurable and depend on the number of threads ($N_{threads}$) and the number of signals ($N_{signals}$).

1. StateCounters: Holds the state of each thread and the count value within that state (for example ONTIME left or OFFTIME left).

   $Size = 16 * N_{threads}$

2. SignalMonitors: Holds the value of each signal. Initialised with $SET\_SIGNAL$ and decremented whenever the $SIGNAL$ instruction is issued.

   $Size = 8 * N_{signals}$

3. SignalMatrix: Holds information about waiting threads.

   $Size = N_{threads} * N_{signals}$

4. ThreadInfo: Holds certain thread parameters - preset ONTIME and OFFTIME, thread execution vector and thread stack pointer.

   $Size = 48 * N_{threads}$

Let $P_x$ be the proportion of CPU resource that a thread $T$ receives.

In the complete absence of other threads or in the presence of only threads running at a lower priority level, the equation describing $P_x$ is:

$$P_x = \frac{t_{on,x}}{t_{on,x} + t_{off,x}}$$

$t_{on,x}$ and $t_{off,x}$ are the thread's ONTIME and OFFTIME parameters, respectively.

A thread will certainly face starvation if the threads of a higher priority cause the processor utilisation to reach 100%. If N is the number of threads whose priorities are higher than the thread being examined, $T$, and $U$ is defined as the total utilisation of the threads above $T$ then:

$$U = \sum_{i=0}^{N-1} \frac{t_{on,i}}{t_{on,i} + t_{off,i}}$$

Starvation of $T$ will occur if the following inequality does not hold

$$U \leq 1$$

This is necessary and sufficient, and can be used to prove starvation.

To alleviate starvation a designer can:

- Raise the priority of the thread, thus reducing N
- Reduce $t_{on}$ of one or more high priority threads
- Increase $t_{off}$ of one or more high priority threads

If $U \leq 1$ but the spare capacity is less than the requested proportion of CPU resources, $P_x$, then the actual allocation $P_{run}$ is $(1 - U)$.

If the total utilisation of the system is less than or equal to 100%, then all threads are guaranteed to get their requested proportion of average CPU resources over an interval of time much longer than the maximum period (ONTIME + OFFTIME). However, with no starvation guarantees, if the system is overprovisioned then only the low priority threads will suffer. Though starvation is clearly not desirable, the scheduling mechanism described herein was the preferred scheduler implementation within the TopDog because of its inherent stability. As we shall see shortly, some other algorithms become unstable or behave in a non-deterministic manner when they are overloaded. In the TopDog scheduler, $U$ of the lowest priority thread can be kept at an acceptable level in a straightforward manner by doing static checks at compile-time. However, in the general case, this static analysis might not be possible because threads might be dynamically loaded or their attributes might change at runtime due to data dependencies for instance. The ability to perform profiling and subsequent optimisation of performance is clearly beneficial.

Let us discuss some of the properties of the TopDog scheduler with reference to common scheduling algorithms and expose the advantages and disadvantages of the TopDog implementation.

The job of the scheduler can be succinctly captured by the question — given a set of runnable threads, which thread is selected to run next, and for how long? How this choice is made has important bearings on several aspects of the system such as quality-of-service, fairness, processor utilisation and most importantly energy consumption. Most conventional scheduling algorithms are dynamic, which means the scheduling decisions are made at run-time but for very simple systems it is possible, and indeed energy-efficient to prepare the schedule beforehand in a static fashion. Though this is of course inflexible, it is useful in certain systems such as safety-critical systems because of its predictability.

There are two kinds of real-time schedulers - soft real-time and hard real-time. Hard real-time schedulers offer guarantees that absolute deadlines will be met while soft real-time schedulers might miss deadlines occasionally and hope the rest of the system tolerates this. Scheduling algorithms are also split between static-priority algorithms which set all task priorities at design time (these priorities remain constant for the lifetime of the task), and dynamic-priority algorithms which set priorities at runtime using heuristics which depend on approaching deadlines for instance. A further categorization of scheduling algorithms separates them into pre-emptive and non-pre-emptive algorithms depending on whether tasks are allowed to run to completion before the scheduler can resume and reallocate resources.

The Rate Monotonic Algorithm (RMA) is a dynamic real-time scheduling algorithm which uses a static priority setting. It is one of the most popular static scheduling algorithms because it is an optimal algorithm, proved by Liu and Layland [75], which gives rise to the following rule which is often quoted in the literature —

> "... If a task set cannot be scheduled using the RMA algorithm, it cannot be scheduled using any static-priority algorithm..."

It operates as follows: the programmer specifies the deadlines or periods of tasks along with their maximum running times, and the tasks which have more frequent deadlines or shorter periods have higher priorities. A related algorithm known as the deadline-monotonic algorithm adds flexibility to the basic algorithm by removing the constraint that deadlines and periods are equal — deadlines can be less than periods. According to Liu and Layland, all deadlines will be met for any arbitrary set of N periodic tasks provided the utilisation U is below a certain value as shown below.

$$U = \left( \sum_{k=1}^{N} \frac{X_k}{T_k} \right) \leq N \left( \sqrt[N]{2} - 1 \right)$$

Where $X_k$ is the computation time of a task and $T_k$ is its period. Unfortunately, for a large number of tasks, this bound is 69.3%. As a general guideline, some experts [108] have suggested that to achieve 100% utilization when using RMA, one should assign periods so that all tasks are harmonic; where periods are exact multiples of shorter periods. This maximises CPU utilisation while retaining schedulability at the expense of flexibility.

The Earliest Deadline First (EDF) [111] algorithm is a dynamic priority scheduling algorithm which uses runtime information to arrange tasks in the order of their approaching deadlines, that is, the task with the closest deadline has the highest priority and is run before any others. EDF is an optimal algorithm. Thus, if any specification of a set of

tasks with arbitrarily chosen periods and maximum execution times, is actually schedulable with no missed deadlines, then EDF is guaranteed to schedule such that all the given tasks complete by their deadlines. It can guarantee this provided the total utilisation is below 100%; a much higher bound on system loading than the RMA algorithm discussed previously. The Earliest Deadline First is used widely and it is applied, for instance, in the real-time scheduler known as *atropos* which was part of the Nemesis kernel [94].

Unfortunately, the EDF algorithm will behave unpredictably if it is overprovisioned and even cause threads normally considered to have a high priority (responsiveness), to be penalised or miss their deadlines as a result. In the worst-case scenario, known as the Domino effect [29], all threads will miss their deadlines. While this might be a pathological case, the TopDog scheduling algorithm stands out in this regard because it is rules out the possibility of any catastrophic failure when the system is overloaded. This fact is important because while one could always run static checks at design-time to ensure that utilisation is well below the maximum, it might be necessary for the system to be actually overprovisioned if it is known a priori that the actual occurrence will be rare. For instance, one can design a purely event-driven system which is actually overprovisioned, and safely rely on the priority mechanism within the TopDog to enforce some control if, for some reason, all the interrupts did arrive at once. In this situation actual utilisation estimates are not very useful but it is important to know that response times of the higher priority threads will not be tampered with. Apart from hardware interrupts other operations common in distributed computing environments such as dynamic loading of external modules might also lead to overprovisioning.

While it is possible in theory to implement the EDF algorithm in hardware, the complexity involved renders it less energy-efficient than the TopDog implementation. This is because extra hardware will be needed to sort tasks according to their deadlines and this will also be a time-consuming process which would not scale as well as the TopDog solution does. As we shall see in the section on the TopDog implementation, the fact that it operates on fixed priorities and has a regular structural framework internally means it can make fast decisions (in constant time) and perform very rapid context-switches.

The EDF scheduler is also less amenable in a controlled and reckonable manner than the TopDog scheduler. For instance, if we want to increase the responsiveness of a thread in the TopDog scheduler, we can simply raise its priority and then recalculate the utilisation of the lowest priority thread in the system. If some threads have been affected then we can proceed to reduce the on-time or increase the off-time of the thread being modified which are two independent ways of keeping the utilisation of that thread constant. We only need to modify the attributes of other threads if this process fails. However, in the Earliest Deadline First scheduler, a similar procedure would involve attempting to make

the task execute more frequently by specifying shorter deadlines, but then the on-time would have to reduce in direct proportion otherwise the utilisation will increase. This risks making the system unstable if it is close to the maximum utilisation; forcing us to do a more extensive adjustment of the runtime parameters of many other threads and not just the one we are interested in. Moreover, specifying shorter periods is not a foolproof method of increasing responsiveness since the task with the nearest deadline varies over time.

Research by Baruah et al. [23] proved that optimally scheduling periodic tasks on multiprocessors is a problem which can be solved in polynomial time (on-line) by using Pfair algorithms. While the research looks promising it uses the concept of proportionate fairness (Pfairness) which relies on dividing the tasks into very small strictly periodic subtasks so that they behave essentially like "fluids" and the fairness constraint can be applied in a tractable manner. The drawbacks are complexity of the optimal algorithm and the fact that such highly regular tiny schedulable quanta are difficult to achieve in practical hardware.

## 4.9 The Implementation of TopDog

Figure 4.5 shows the main components of the TopDog hardware module. For clarity, other parts which are not critical to its operation as a hardware scheduler are not shown here. Its primary input is from the CPU command interface and it handles instructions such as FORK, WAIT, SIGNAL, SET_SIGNAL, CHECK, YIELD and EXIT, as well as other multiprocessing instructions.

Before issuing the FORK command, the SET_PRIORITY instruction should be issued otherwise a thread with the lowest available priority level will be created. The thread ID is generated automatically by concatenating the priority and the lowest free thread number at that priority level. This approach was used since the process of creating a unique ID might be error-prone and complicated in software. Error-prone because it has to be performed atomically by all threads and complicated (time-consuming) because when threads exit, their IDs need to be recycled.

Other thread setup instructions such as SET_ONTIME, SET_OFFTIME, SET_VECTOR and SET_TSP are required which set thread attributes, that is, the maximum run time, inactive time, where the execution should begin and where the stack associated with this thread lies in memory. The FORK instruction puts the new thread ID into one of the READY queues, P0 to P15, as shown in the diagram. These are FIFO buffers which enforce the scheduling policy described earlier.

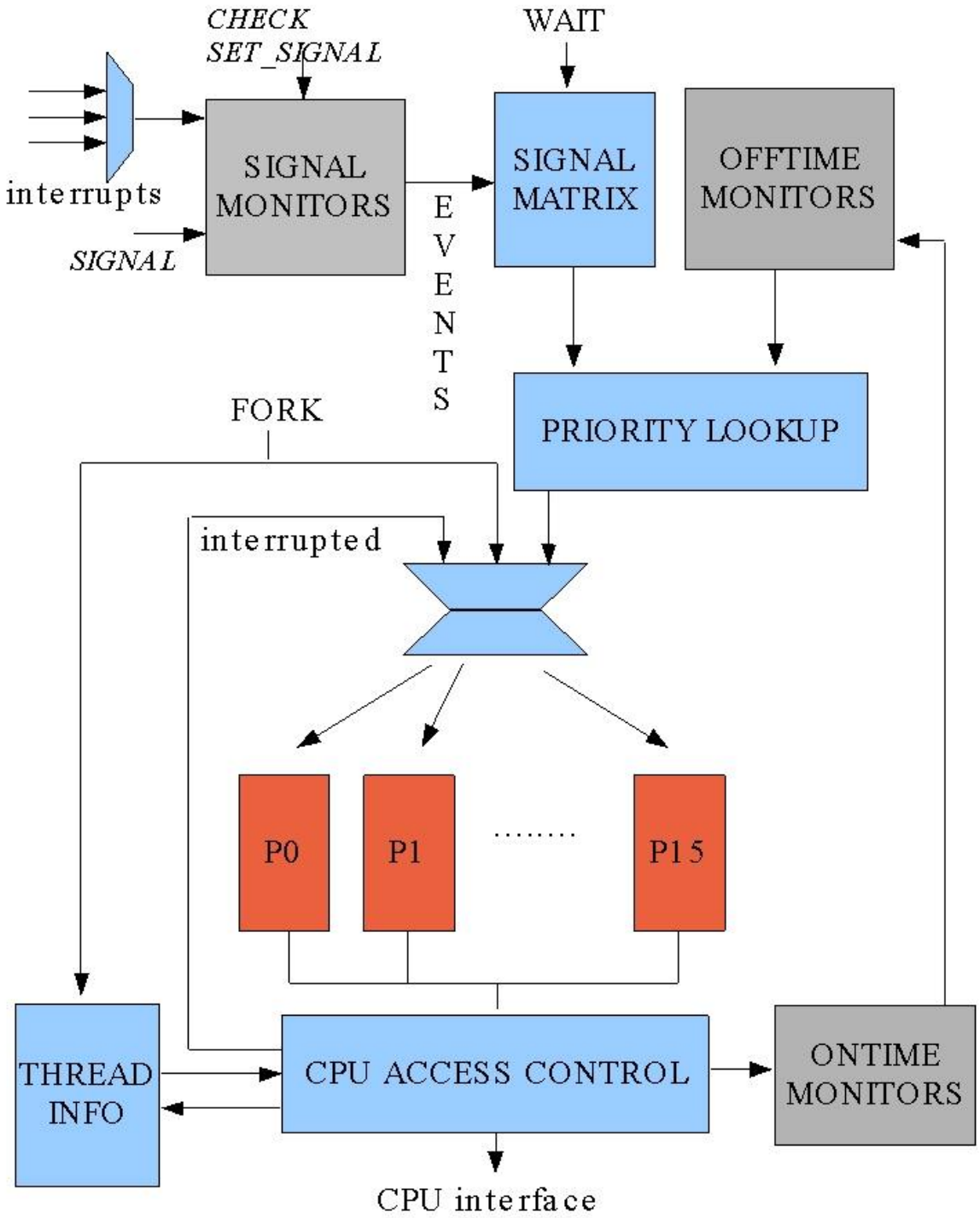


Figure 4.5: TopDog implementation

When a thread is placed in the appropriate buffer, a bit field is also set in a register called PRS which is used to tell the CPU access control block what the current highest active priority level in the entire READY queue is. This enables the CPU access control to quickly determine whether it needs to remove any currently running thread if there are no IDLE CPUs available. It will only interrupt any running thread if it finds that that thread's priority is less than that indicated by the PRS register. To achieve this rapidly enough, the CPU access control has two registers which are bit field representations of pertinent values. These are the PCS and ICS registers shown in Figure 4.6. Polling would be bad for performance as these are extremely time-critical tasks (checks) which need to be performed relatively frequently.

In addition, posting the highest priority thread or set of threads is sped up by the fact that the threads are sorted into separate queues as on entry into the READY state. Although, it may seem like this will create a bottleneck by becoming a single point of synchronisation, in the actual implementation, the various parts of the TopDog are actually loosely-coupled and can proceed somewhat independently. For instance, the SIGNAL MATRIX responds to events from the SIGNAL MONITORS and posts thread IDs to a buffer which is merged with thread IDs posted by the OFFTIME MONITORS also operating independently. The SIGNAL MATRIX holds information about the WAIT QUEUE, that is, it tells us what set of threads are waiting for a given signal to become activated within the SIGNAL MONITORS module. A compact bit field representation is used and the IDs of the waiting threads are recovered in such a way that threads with low ID numbers — hence higher priority — are removed from the queue first. The SIGNAL MATRIX is vastly parallel and comprises 8 banks of 32-bit wide on-chip memory which are each 64 words deep. This is to enable a waiting thread to be found and awoken in a single clock cycle for any of the 64 signal inputs.

The TopDog will issue threads to CPUs in the system or remove existing threads in the following two scenarios. After performing a FORK, receiving a signal event from the SIGNAL MONITORS sub-block or realizing that an OFFTIME MONITOR has expired, the TopDog transfers the thread (or group of threads) thus released into the appropriate priority levels in the READY QUEUE, and then determines whether there is an idle CPU or a thread which can be interrupted if all CPUs are busy. In another scenario, when a thread issues a WAIT command or its ONTIME expires, the TopDog will select the highest priority thread from the READY QUEUE to replace the existing thread.

Since the TopDog and SpotCore subsystems operate independently, are not synchronized, and can both be either a master or a slave in different transactions, a simple protocol involving request and acknowledge packets was devised for effective communication between them. When the TopDog wishes to spawn a new thread or remove or

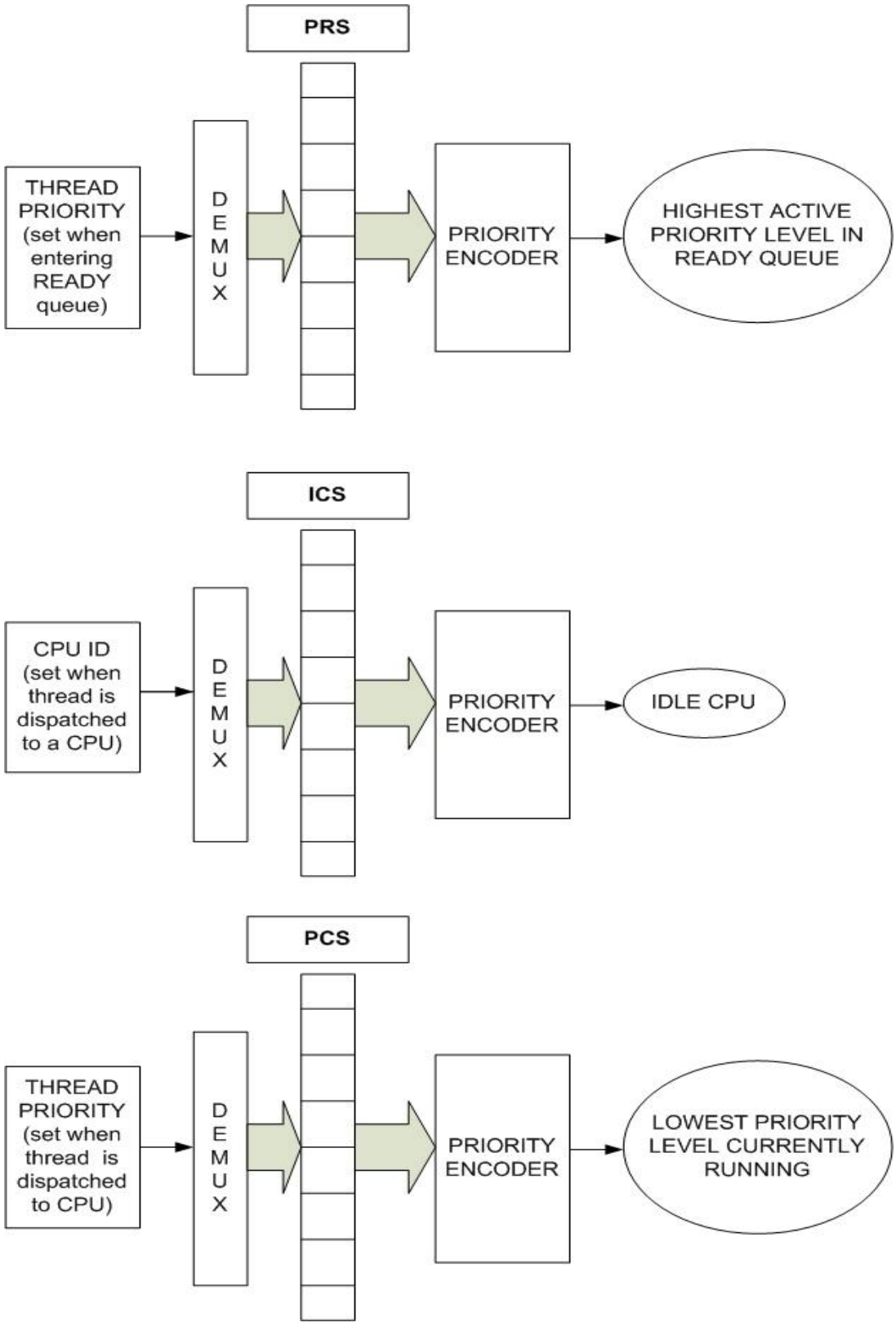


Figure 4.6: Accelerating critical procedures within the TopDog

replace an old one on one of the CPUs attached to it, it sends an interrupt request packet which comprises the Thread Stack Pointer (TSP) and execution VECTOR as well as some extra information which indicates whether we are replacing the current thread or simply offloading it and leaving the CPU idle. These extra bits also indicate whether the replacement thread is a new (fresh) one, or a continuation of a previously interrupted one. This information is pertinent as it informs the CPU in a direct way whether it needs reload its state (registers) from the stack or just start executing immediately. After receiving and processing the interrupt request packet, the CPU returns an acknowledge packet containing the old Thread Stack Pointer (TSP) and the current program counter value (VECTOR).

## 4.10 Results and Discussion

The efficiency of the interprocessor communication scheme the TopDog provides is proved by the fact that it achieves a very low-latency interprocessor communications mechanism with modest hardware resources.

The entire process occurring from the instant a FORK is issued by one thread to when the spawned thread is finally established on another SpotCore processor takes under 10 clock cycles. One the other hand, the interprocessor communication between ARM cores takes upwards of 30 clock cycles if an interrupt-driven mechanism is used and there are no caches or the caches are turned off. The different schemes are explored in the next section but one can note here that if the caches are used then the time-consuming "invalidate" and "clean" cache operations must be performed.

When synthesised, the TopDog hardware uses about 4000 logic gates (NAND) and has a peak memory usage of about 3KB if handling up to 256 threads. The logic and memory usage scale linearly with the number of threads.

## 4.11 Communicating Events Between Processors

The multiprocessor system developed in this thesis relies on having all the processors connected to a shared memory as opposed to having processors with their own local memories. A distributed memory approach is more scalable because there are limitations on the available bandwidth in a shared memory system as the number of processors (hence connections to a single memory infrastructure) is increased [53]. However, using shared memory has the advantage that the software development process can be more evolutionary and the interprocessor communication mechanism is straightforward — memory

load or store. Provided the bus contention is manageable, for a small cluster of processors, sharing data this way can be faster than the case where there are disjoint memories. However, techniques such as caching and buffering of memory writes are usually employed to improve performance. Thus, it is often the case that even in a system described as having shared memory, a write from one processor is not immediately visible to the other processor and a read from memory is not guaranteed to be the most up-to-date data. As a result, for interprocessor communication through shared memory to be effective, there needs to be a way of ensuring memory coherency. In the absence of additional hardware such as a Snoop Control Unit [18], the process of exchanging information between processors can be slow because it may involve a set of cache maintenance operations. It is important to note that while it may be possible to reduce the actual amount of interprocessor communication through careful algorithm design which would overlap computation with communication, certain critical messages would still have to be exchanged relatively frequently between collaborating processors. These messages (relating to thread management operations and synchronisation events) would have to be delivered in a timely manner.

If we consider the case of an elaborate memory hierarchy with a point of coherence which is not adjacent to the processor, then the TopDog hardware may have some benefit because it might be able to reduce the overhead which arises in thread management operations spanning multiple processors. For example, in a multiprocessor with no TopDog, some other type of suitably fast interprocessor communication would be necessary to determine the execution state of the processors in the system. In this situation, one might want to send a newly created thread to an idle processor quickly, or at least realise that there is currently no idle processor and continue executing the established threads without too many time-consuming operations.

However, while the TopDog hardware might be able to speed up the communication of events between processors, it is not a general interprocessor communication solution as it cannot handle arbitrary data. In addition, by relying on a shared bus topology, it can only connect to a small number of processors (say up to 8). The TopDog is similar in principle to a way of exchanging event information, commands, and status updates in multiprocessors using what is called a mailbox. This could just be a region of shared memory (non-cacheable) or as is the case in [112], special registers in the shared address space of the processors. In [112] the sender deposits a message in one register and writes a command to another which triggers an interrupt. This causes another processor to suspend its execution and read the message. Without using interrupts, the receiving processor would have to poll until the command field changes to some prearranged value. While this might be less efficient, it avoids the requirement for a shared interrupt

controller. Note that the system bus must be able to guarantee that reads from or updates to the mailbox are atomic. This mailbox concept has been found to be useful in [36], and the multiprocessor described in [116] uses a distributed interrupt mechanism for interprocessor communication. In [36] mailbox registers are used to implement channel communications in a heterogeneous multiprocessor. An elaborate discussion on how mailboxes with interrupts can be used for "fast and simple" communication between a microcontroller and a DSP can be found in [11]. The TopDog hardware can be viewed as the unification of a mailbox, shared vectored interrupt controller, and a simple scheduler. It stores thread state information written to it by threads running on any processor, and it then uses this information to decide when to load or interrupt (and suspend) threads.

## 4.12 Towards a Robust Multiprocessor Architecture

Let us now examine ways of making a system using the TopDog more robust by providing memory protection hardware, hardware-assisted deadlock detection, and support for priority inversion.

### 4.12.1 Memory Protection

The role of memory protection is to guard certain memory regions from undue interference and to provide strict isolation from spurious reads or writes. Ideally, we would like to have the most flexible variant of this where an arbitrary number of memory regions can be specified, starting at arbitrary locations, with arbitrary sizes or granularity and arbitrary access permissions. However, in order to keep the tables holding the access permissions small and to enable fast look-up, the implementation in the TopDog uses the top N bits of the address bus from the SpotCore CPU to check the validity of each memory access when it is initiated. This results in $2^N$ entries in a memory block which is checked on-the-fly during any memory access. The granularity or minimum block size which can be protected is thus overall *memory size*/$2^N$. The current prototype has a total memory of 32K and can effectively protect 1K blocks from code which is "not trusted". While this scheme is straightforward to build, a finer granularity than 1K may be expensive. Another implementation might choose to specify the base address along with a range(size) for each thread thus providing a more flexible mechanism which is better for small memories. Threads are effectively isolated because a thread is prevented from accessing memory outside its specified scope in read-mode or write-mode.

### 4.12.2 Deadlock Capture

A lock is important in guaranteeing mutually exclusive access to a shared resource. If the resource a thread is attempting to lock is already locked, the thread will be suspended by the TopDog which will immediately remove it from the processor and automatically awaken it when the resource becomes available. This is in contrast to most systems which suffer from the spinlock problem where the thread waits in a loop until the lock is released. Spinlocks are inefficient (waste energy) and appear in many software programs because event-driven mechanisms and runtime environments are poorly supported. However, there is a body of research [41] against concurrency management based on locking. These promote lock-free implementations such as transactional memory instead. Using transactional memory, each thread performs writes optimistically but will roll back its write operations and try again if any conflict is detected. In some instances, the overhead of doing this can be unacceptably high but the scheme has several advantages, the main ones being that deadlocks are avoided by design, and no time is wasted acquiring or managing locks. Unfortunately, this scheme is not sufficiently lightweight for an embedded system, that is, the transaction log wastes memory space, and rewrites due to failure wastes CPU cycles if there are many transaction failures and in complex system this means there can be no realtime guarantees. In addition, I/O devices need to be locked anyway before access, as writes are not easily reversible. In this case, locking seems like a necessary evil. The TopDog ensures that the acquisition of locks remains fast while it also checks for deadlocks automatically.

Bacon et al. [20] provides a good analysis of the problem of deadlocks. In general, deadlocks can be avoided if certain conditions hold, and these are summarized as follows:

- All resources are acquired at once at start of the thread
- No thread can have multiple locks
- Locks are acquired in a strict order
- Locks can be transferred or revoked

It was decided not to use any of these in this design because the first three placed undue restrictions on the programmer's flexibility in managing concurrency while the last represents a departure from pure locking semantics.

The three memory structures shown in Figure 4.7 form the core of the Deadlock Detection hardware. It uses just slightly more than 1 KB and a little logic to handle up to 32 locks between 256 threads. Notice that there is some redundancy as the LOCK REQUEST MATRIX contains essentially the same information as the REQUEST TABLE

but with a different representation. This was arranged thus because the LOCK REQUEST MATRIX speeds up the process of looking for and waking any waiting thread(s) when a lock has been released while the REQUEST TABLE speeds up the operation of the Deadlock Detection logic as described below.

A deadlock occurs when there is what is commonly known as a "circular wait" condition. In the simplest case, thread 1 waits for a lock held by thread 2 which is waiting for another lock held by thread 1. However, this scenario can be extended arbitrarily and one can have lock requests spanning many threads and ultimately ending with a request for a lock held by the thread which started the request in the first place. Therefore, when the Deadlock Detection logic gets a request, it checks both the OWNERSHIP TABLE and REQUEST TABLE alternately until it reaches a thread which is not waiting for any locks. However, if it ends up referring to the thread ID it started with; it raises a deadlock exception which can then be used to recover the system in a timely manner. So for example, in Figure 4.7 a deadlock exception will be raised if thread 5 requests lock number 1.

### 4.12.3 Priority Inversion

Another problem with locking is that it can lead to what is known as priority inversion when used in a system with thread priority levels. In a typical case of priority inversion a high priority thread faces an abnormally long delay because a medium priority thread has interrupted a low priority thread which owns a lock that the high priority thread needs. Since this has the capacity to damage the reliability of the system which is designed from the ground up to provide deterministic performance, it was decided that hardware support for preventing priority inversion had to be incorporated into the TopDog hardware.

One method of preventing priority inversion is to use a priority ceiling [37] such that a task has its priority raised to the highest level possible in the system or to the level of the thread with the highest priority which might need access to the resource being locked. The priority is then dropped when the resource is released. However, giving a low priority thread such a high priority on acquiring a lock can be wasteful and hampers the responsiveness of the system unnecessarily if there is no subsequent contention. High priority threads which do not need that particular lock must not suffer as a result of trying to prevent priority inversion. The other method referred to as priority inheritance, is a more efficient solution as it raises the priority of the thread holding the lock to that of the higher priority thread requesting the lock. Unfortunately, changing a thread's priority necessitates a lot of updates within different parts of the TopDog, affects the initial scheduling or runtime arrangements, and it may have to be performed several times. Thus the concept of a *shadow priority* was developed. The shadow priority table

| Ownership Table | |
|---|---|
| Lock # | Thread # |
| 0 | 5 |
| 1 | 6 |
| 2 | 7 |
| 3 | 8 |
| 4 | ? |
| 5 | ? |
| 6 | ? |

| Request Table | | |
|---|---|---|
| Thread # | Lock # | |
| 0 | ? | |
| 1 | ? | |
| 2 | ? | |
| 3 | ? | |
| 4 | ? | |
| 5 | ? | <=1 |
| 6 | 2 | |
| 7 | 3 | |
| 8 | 0 | |

| Lock Request Matrix | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| | Threads | | | | | | | | |
| Lock # | | | | | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | | | | | | | | | X |
| 1 | | | | | | ○ | | | |
| 2 | | | | | | | X | | |
| 3 | | | | | | | | X | |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |

Figure 4.7: Memory structures in deadlock detection hardware

holds the elevated priority level information of any thread holding a lock and is updated whenever any high priority thread tries to access the locked resource. This scheme was able to prevent priority inversion without changing the existing TopDog hardware described previously which is highly optimised for low latency.

## 4.13 Summary

This chapter has presented the concepts behind the TopDog hardware scheduler and illustrated how the prevailing ideas in multiprocessing influenced the design of hardware operations for managing concurrency. The great importance of having the right tools to extract thread-level timing information for profiling in order to achieve low energy consumption was also illustrated. The TopDog hardware provides a backend which can be used for this purpose. The TopDog enables the entire process of forking to be much less than in conventional systems. The approach elucidated in this chapter is unique because it also incorporates traditional operating system features like QoS and the avoidance of priority inversion. It extends the "energy-efficiency-through-low-latency" concept to other important runtime elements like deadlock checking and memory protection and makes them an integral part of the TopDog. A unified interrupt mechanism also makes it easy to write energy-efficient event-driven code for several processors by setting triggers on external signals or internal events with real-time guarantees.

# Chapter 5

# A Case Study in Scalable Concurrent Software

The previous chapters have examined the creation of an efficient processing unit and detailed the construction of the DAPH platform, particularly the novel hardware which facilitates thread management and reduces overhead. This chapter explores the scalability of software written with a parallel architecture like the DAPH platform in mind. It accomplishes this by looking at an application involving intensive processing.

Many processes in sensor-driven computing rely on being able to obtain and analyse the frequency spectrum of an incoming signal. This might be for the purpose of high-level inference, data compression, or communication; among many others.

The Fast Fourier Transform (FFT) provides a very efficient way of computing the Discrete Fourier Transform (DFT). The DFT is a mathematical procedure which extracts the frequency domain representation of a sampled (discrete-time) signal. For a set of N sample points, the DFT is defined as:

$$X(k) = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$$

Unfortunately, using the above calculation results in a lot redundancy and the FFT manages to exploit properties of the solution such as symmetry, and also reuses previously computed values within the calculation in order to reduce the complexity from $O(N^2)$ to $O(N\log_2 N)$. To achieve this, the most common variant of the FFT algorithm known as the radix-2 FFT divides an N-point DFT into two N/2-point DFTs and then subdivides each N/2-point DFT into two N/4-point DFTs, and so on. This assumes N was a power of 2 to start with. After several stages we will be left with just 2-point DFTs which cannot be subdivided any further. This algorithm is a very good benchmark because its "divide-and-conquer" strategy is applicable to many other applications, and its dataflow patterns are non-trivial and will act as a kind of stress-test for any interprocess-communication mechanism, when written for a multiprocessor system.

## 5.1 Task-Partitioning

For brevity, the details of the derivation will not be described here, [77] is an excellent source for this. However, if the operation of a decimation-in-time FFT is sketched, it results in two alternative implementations depending on whether we supply the FFT with input samples that are in-order or rearranged based on their bit-reversed addresses. Figure 5.1 depicts the scenario where the input samples have been shuffled, and the output data is in order as a result.

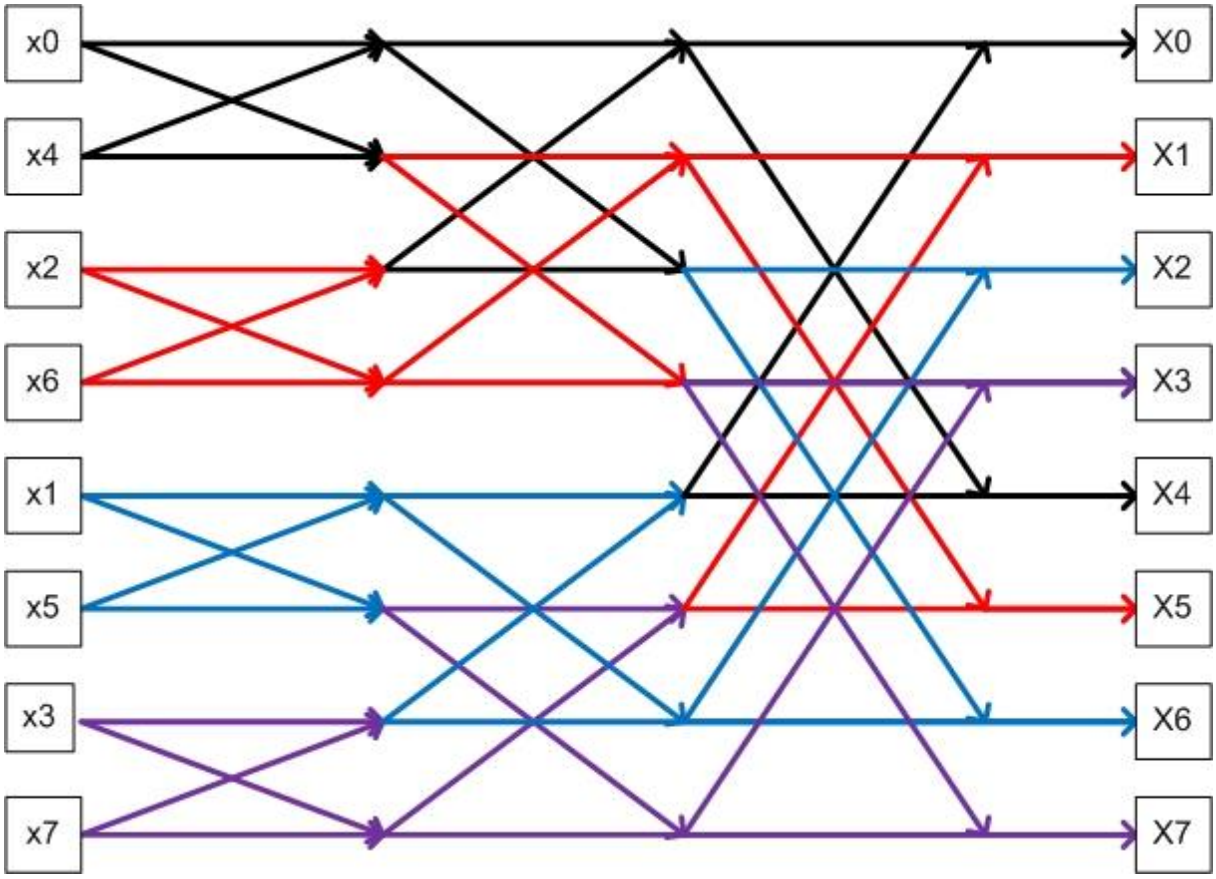


Figure 5.1: FFT dataflow

Memory access is made uniform across the multiprocessor implementation so the difference between the two approaches is minimal because the way the work is partitioned, where the threads are eventually run, and the data elements these threads access is irrelevant. Culler et al. [34] explore a more generic FFT implementation which does not rely on uniform memory access. In practice, uniform memory access is satisfied for 4 cores in the multiprocessor design used here by using memory with two ports and clocking it at twice the operating frequency of the CPUs so the memory module appears to have four

coherent ports. This is not normally possible in larger systems, but small amounts of on-chip memory in an embedded system can be sufficiently fast.

The work is partitioned between the processors in a way that does not require the programmer to explicitly specify the number of CPUs which are going to be used. In general, the number of points in the FFT will be more than the number of CPUs. The basic unit of work is the FFT butterfly which are all coloured differently within each stage in Figure 5.1. Note that the twiddle factors are left out for clarity.

The computations required for each butterfly are:

X1 = X0 + W*Y0

Y1 = X0 – W*Y0

W is the twiddle factor.

X1 and Y1 are stored back to the same memory locations that held X0 and Y0 respectively as the algorithm shown in Figure 5.1 is a memory-conserving in-place algorithm.

Since there are N/2 butterflies at each stage in the calculation of an N-point FFT, we have to allocate a number of butterflies to each thread. Thus at each stage a master thread will pass two numbers (parameters) on the stack of each thread that it forks to indicate the range over which that thread should operate. The first parameter indicates where the thread should start in the data series and the second is the number of butterflies to execute. The SIGNAL and WAIT constructs are used as described previously to synchronise all the threads at the end of each stage before moving to the next stage. The data re-ordering stage which must come first was also multithreaded. In order to avoid race conditions which would make some form of synchronisation necessary thereby adding overhead, this was not performed in-place and a second input data buffer was used. Each thread runs several iterations of the following sequence over the input data elements. This code sequence illustrates the utility of the single-cycle "bit reversal" processor instruction when used to sort the input samples such as those in the 8-point FFT illustrated in Figure 5.1.

```
MOV R3,#InputDataBaseAddress
BITREV R1,R0
LSL R1,#29 ;interested in top 3 bits for 8-point FFT
ADD R1,R3,R1
LDR R2,[R1]
STR R2,[R0]
;add 2 to R0 and check that we have not reached
the end of the thread's allocated data set
```

Efficiently sharing work between threads needs careful thought. On the one hand we could let the master thread which runs at the start of each stage initialise the pointers required by the butterflies, that is, the pointers to the top data element and the twiddle factor. However, we chose to perform this computation for every butterfly within the thread itself and save memory while also improving parallel performance as this precomputation is spread across many threads. Besides the overhead of dispatching and loading threads, this precomputation represents another source of overhead relative to the normal unparallel case because in sequential code, the computation will effectively reuse the values deduced from previous iterations within the same stage. However, since the work is divided such that any butterfly can be executed by any thread, each butterfly computation must be preceded by a quick calculation of the position of its input complex vectors based on a single parameter, its unique butterfly number of which there are (N/2) per stage as we have already seen. This trade-off makes partitioning and sharing work more straightforward and adapts automatically to changes in the number of cores.

Although the FFT computational flow diagram looks complex there is an intrinsic pattern which lends itself to dataflow analysis. We can simplify the generic expressions relating arbitrary butterfly numbers to actual points in the dataset at the input of each stage as follows:

For an N-point FFT, the pointer to the top data element within each butterfly can be calculated as follows:

$$ptr1 = p * 2^{m+1} + q$$

The pointer to the bottom data element is:

$$ptr2 = ptr1 + 2^m$$

m is the stage number (from 0 to $(\log_2 N - 1)$)

If n is the number of the butterfly, which lies in the range 0 to ((N/2) – 1) for each stage, then

$$p = \lfloor n/m \rfloor$$

$$q = n\%m$$

In addition, the relationship between the position n and the pointer to the twiddle factor $W_N^k$ was determined to be:

$$ptrw = q * \frac{N}{2^{m+1}}$$

It is possible to simplify these expressions further and optimise them to minimise their impact on the execution time.

The code fragment below shows how the so-called anchor points or handles were calculated. *GETLOOPCOUNT* is used to get the value n.

```
GETLOOPCOUNT R3
LSR R3,R6          ;R3=p
MOV R0,#0
INV R0,R0          ;all ones
LSL R0,R6          ;obtain bit mask
GETLOOPCOUNT R2
AND R2,R2,R0       ;R2=q
INCR R1,R6         ;R1=mplus1
LSL R3,R1
ADD R3,R3,R2       ;R3=ptr1
ADD R0,R3,R4       ;R0=ptr2
MOV R4,#1024
LSR R4,R1
MUL R2,R2,R4       ;R2=ptrw
```

Computing the handles as shown above adds a 27% overhead to the butterfly computation. Thus a simple serial implementation in which only the core butterfly operations — read data, perform complex multiplication and complex additions, and write back results — are performed will actually be faster than the parallel variant on a single core. Also observe how the structuring of the code avoids register *spillage* even though data is being passed between many temporary variables.

Note that the SpotCore processor has no floating-point unit so the twiddle factor was scaled up and supplied in the testbench. This requires the top input data element in each butterfly to be scaled up as well, which implies both outputs of the butterfly need to be scaled down by the same factor. (As expected, this factor was a power of 2 so that a logical shift right would suffice in scaling down instead of an elaborate division.)

## 5.2 Results

The results obtained from running a 256-point FFT algorithm on the DAPH with a variable number of cores can be compared with those from well-established cores (Figure 5.2 and Figure 5.3). The execution times for the DAPH were obtained by timing an implementation running on an Altera Field Programmable Gate Array (FPGA) board. The execution times for the ARM7TDMI, ARM1136JF-S, MIPS 20Kc and IBM PowerPC were computed from the EEMBC [114] FFT benchmark scores (telecom file). Obviously the very advanced cores ARM11, MIPS 20Kc and IBM PowerPC 750C have very good performance in terms of raw computational speed. Though historically they are RISC CPUs, they have undergone many transformations in order to boost performance and between them they include features such as speculative execution (e.g PowerPC 750C), floating point units, and long pipelines with multiple-issue. The IBM PowerPC has the shortest execution time but the worst energy utilisation relative to the other cores. However, adding more SpotCores to the DAPH reduces the execution time with a relatively small increase in the energy utilisation. That is, though the power consumption of the DAPH platform goes up, the execution time falls in roughly the same proportion; a process which would not work very well without the TopDog scheduler to speed up interprocessor communications. These results are only meant to serve as an approximate demonstration of the embedded parallel programming technique used here because of the inherent inconsistencies in any CPU-to-CPU comparison. For instance, the more advanced CPUs all have L1 instruction and data caches and the power consumption of the ARM11 can vary by as much as 42% depending on whether the caches are included or not. However, in assembling the data for Figure 5.2 care was taken to use only the quoted execution times when the processors are running fixed-point code like SpotCore. The energy values in Figure 5.3 exclude the energy consumption of the memory subsystem. Also, while the results of the synthesis tool can be very close to the actual fabricated device if set up correctly, it must be acknowledged that this is susceptible to myriad variations and an error margin of 10% must be applied.

## 5.3 Summary

This chapter has illustrated how performance can be increased in energy-constrained environments in a more reliable way. While the need for speed will continue to drive many processor architecture developments, it is very clear from the results presented that this approach is orders of magnitude less energy efficient. The scalability of the DAPH like most symmetric multiprocessing architectures is not without its limits and for more than a small number of processors, four in this case, the overhead of a message-passing layer

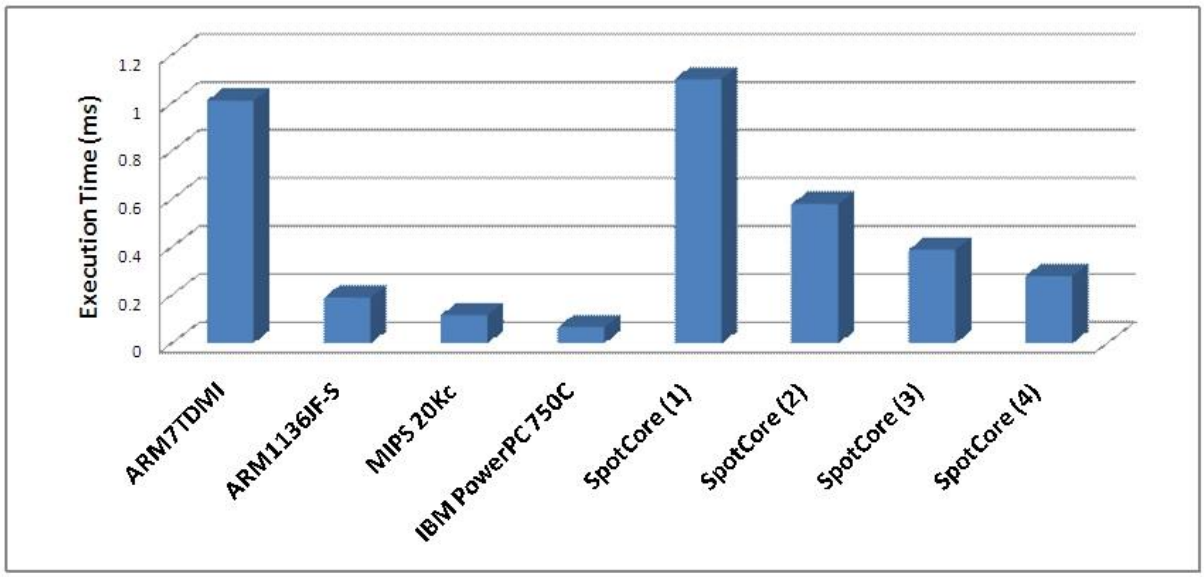


Figure 5.2: Chart showing execution times for different cores

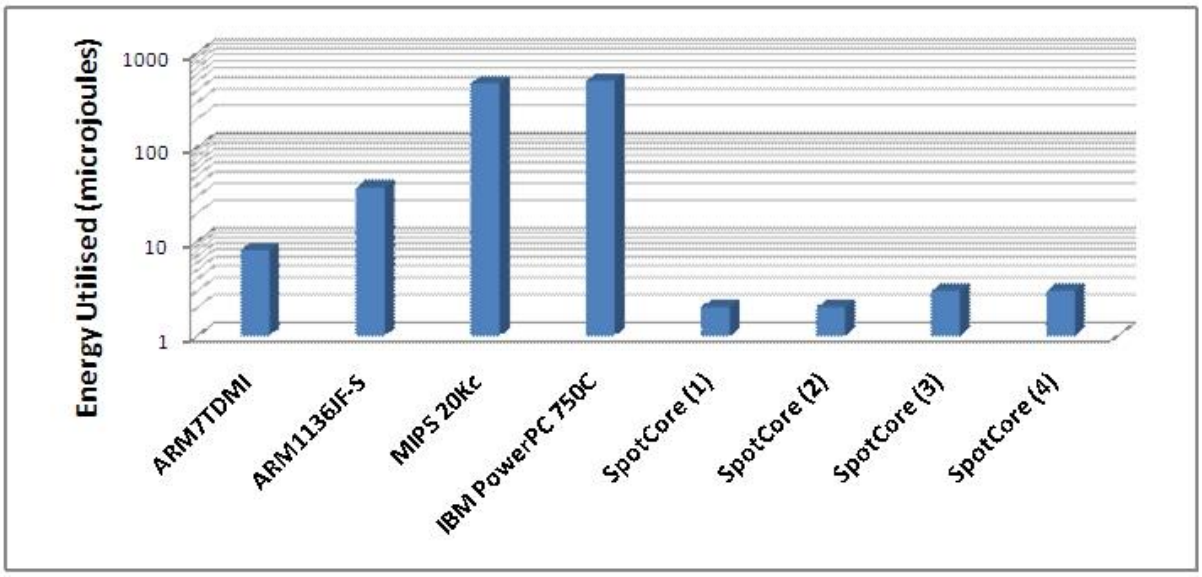


Figure 5.3: Chart showing energy utilisation for different cores

will degrade performance. However, sensor networks will benefit greatly from the ability to make significant improvements in execution times (roughly fourfold in this example) without any noticeable increase in energy requirements by simply taking advantage of parallelism which is readily available within them.

# Chapter 6

# A DAPH System

This chapter uses the multiprocessor platform developed and analysed in the previous chapters to improve end-to-end latency within an example application. The rationale for using a multiprocessing platform to reduce latency, as opposed to just using higher processor clock frequencies for instance, is explained and the system architecture is described so that the time-critical tasks can be identified and understood in the relevant context. Two sets of results are presented:

1. Results showing how latency can be reduced in an energy-efficient way using the DAPH platform.

2. Experimental evidence that the DAPH can be successfully integrated in a working location system described herein.

## 6.1 Reducing Latency in Sensor Networks

Two key factors that determine a computer system's performance are throughput and latency. Traditionally, most designs of computer systems usually consider throughput in the first instance and try to minimise latency as an afterthought. Throughput is defined here as the amount of raw computation that a device can perform per unit time while latency is its responsiveness or how quickly it can handle events in an interactive environment, that is, the lag between generating an event and obtaining an observable response from a sentient system. If one wishes to embed computing in the sensor network itself then the resulting design should still yield performance in terms throughput and latency. This chapter evaluates the effect of multiprocessing on latency in the context of a novel location system design which requires in-network processing.

Latency problems can generally be overcome by developing dedicated hardware solutions so that real-time performance is guaranteed. However, this is not flexible, cannot

be easily maintained, and might suffer from long development times since digital hardware generally requires extensive verification. In addition, validation of complete system behaviour is slow for large hardware designs unless a suitable field programmable gate array (FPGA) or emulator is available. Specialised hardware is also not arbitrarily scalable so system integrators must be careful when reasoning about the expected system performance as the number of input/output channels is increased.

While it is easy to see that closing the loop from sensing to actual feedback in a sensor network requires a low latency infrastructure, other application areas can also benefit from a solution that reduces latency while requiring only software development effort. It is likely that a multicore environment can allow a set of independent hard real-time threads to meet their deadlines in a more flexible manner than a single core one. A case in point is software defined radio which usually involves a significant number of time-critical tasks. A multicore approach can reduce development complexity by ensuring that there are appropriate limits on the latency experienced by many different tasks — that is no thread is effectively starved or denied access to CPU resources for a long period of time. This technique has been proven to reduce time-to-market or provide a realistic development time [22]. By decoupling various threads of execution, a multicore solution preserves flexibility so the design can adapt to emerging standards without much redevelopment. Such tasks are similar to those commonly employed within sensor networks and include:

- Multichannel receiver
- Multichannel transmitter
- Signal scanner
- Echo canceller
- Voice codec
- MAC interface
- PHY controller
- Radio control and system timing

The innovative work is based on the picoChip[1] platform which comprises 250 DSP cores which can be clocked at 160MHz. Each core is a 16-bit processor and multiple cores interact via a message-passing paradigm using an interprocessor routing fabric which is configured at compile-time. In developing the software defined radio, 99 cores were used

[1] www.picochip.com

for the main transceiver, 66 for the scanners, 14 managed the audio subsystem, and 22 handled the MAC and PHY interfaces. 40 cores were used for timing and diagnostics and the remaining cores were simply put to sleep. The modularity meant that scheduling was not unmanageable despite the large number of threads and the fact that they have unpredictable execution times. In addition to this modularity, the resulting device is more power-efficient than many commercial uniprocessors because it is capable of handling up to 64 RF channels and 7 antennas which is significantly more than the single core design.

Watt and May [123] describe how embedded systems can be easily defined and constructed using software. As such, system partitioning is no longer a rigid process which must be performed in the initial phases of the design but threads can be used or defined flexibly to run blocks of sequential code, as concurrent sub-blocks or as a hardware emulation engine enabling fast input/output. These automatically make internal timing independent of the timing of the interfaces while guaranteeing low latency because operations such as thread creation and termination, thread synchronisation and channel communication are implemented in hardware and have a low overhead.

End-to-end latency can be viewed as a Quality-of-Service (QoS) requirement. If sensor measurements generated within the sensor network require an external system for analysis, interpretation and subsequent feedback, then the end-to-end latency can be high because the external network might be remote from the sensor network or there might be extra (unpredictable) processing and communication delays. This is because the external servers while having more throughput will be handling more requests from other sensor networks or users and so might not respond in a deterministic fashion to interactive events. The principal factor governing the latency in a sensor network with DAPHs is therefore the processing delay at each DAPH. This in turn depends on the the number of processing elements, their operating speeds, and the time required to performed context-switches. In Chapter 4, it was shown that the TopDog hardware mitigated the performance cost of context-switches. In this chapter, it will be shown that it is more effective (and hence energy-efficient) to reduce latency by employing more cores than simply clocking a single core at a higher rate.

The case for balanced loads was also presented in Chapter 4 and it was shown that appropriate scheduling was crucial if speedup was to be achieved while utilising approximately the same amount of energy. This analysis can be extended even further to the situation shown in Figure 6.1, where tasks with varying processing requirements $W_i$ are dispatched from a task pool to an array of processing elements. It can be shown that if the tasks are unbalanced then the latency experienced by incoming tasks is much worse in the uniprocessor case and improves greatly as more CPUs are added to the system, assuming steady-state conditions. If the tasks are balanced, the average latency is approximately

the same. This observation comes from the fact that in the unbalanced uniprocessor case, the longer tasks tend to hog the processor leading to an unacceptably high number of context-switches or if uninterruptible will cause significant delays in the execution of other tasks.

This is in contrast to the multiprocessor case where tasks with different processing requirements can be dispatched together and no task has an adverse effect on the latency of other tasks.

Managing latency in an effective manner is relevant because the case where tasks are actually unbalanced is common in practice and can be a problem for sensor network development. In particular, tasks which interface with the physical environment can be made uninterruptible in order to provide a timely service to a critical routine in a transceiver, to avoid loss of synchronisation in data communications, or to avoid corrupting shared memory buffers of an input data stream.

The preceding qualitative analysis of latency improvement through multiprocessing is illustrated in the concrete example described in this chapter.

## 6.2 Network Latency

Besides processing latency, communication delays can also be problematic. In the proposed DAPH sensor network organisation, it is likely that processing latency at the hub is more pertinent because the tiered approach reduces communication latency or the cumulative delays which contribute to the latency or time interval between measuring some physical data and sending a corresponding event to the external system.

With the DAPH approach, unless each DAPH acts as the head of a cluster which itself has multiple levels then the communication delays will be less than in the homogeneous or single-layer ad-hoc network where there might be an arbitrary number of hops between the individual node and the sink node or base station. However, one needs to avoid creating a processing bottleneck at each DAPH so the latency due to processing needs to be constrained. Minimising latency is generally understood to be a difficult problem in sensor networks because nodes are deployed without any formal placement, the communication links are not very reliable and the network topology might be dynamic. Srivathsan and Iyengar [107] identify some of the problems associated with reducing latency at the network and physical layers within the sensor network particularly the difficult trade-off between energy consumption and latency. For instance, a sensor network designer would like nodes to sleep more but then this would exacerbate routing latency. Some MAC protocols such as the Q-MAC proposed by Vasanthi et al. [118] reduce latency by utilising a dynamic sleep schedule.

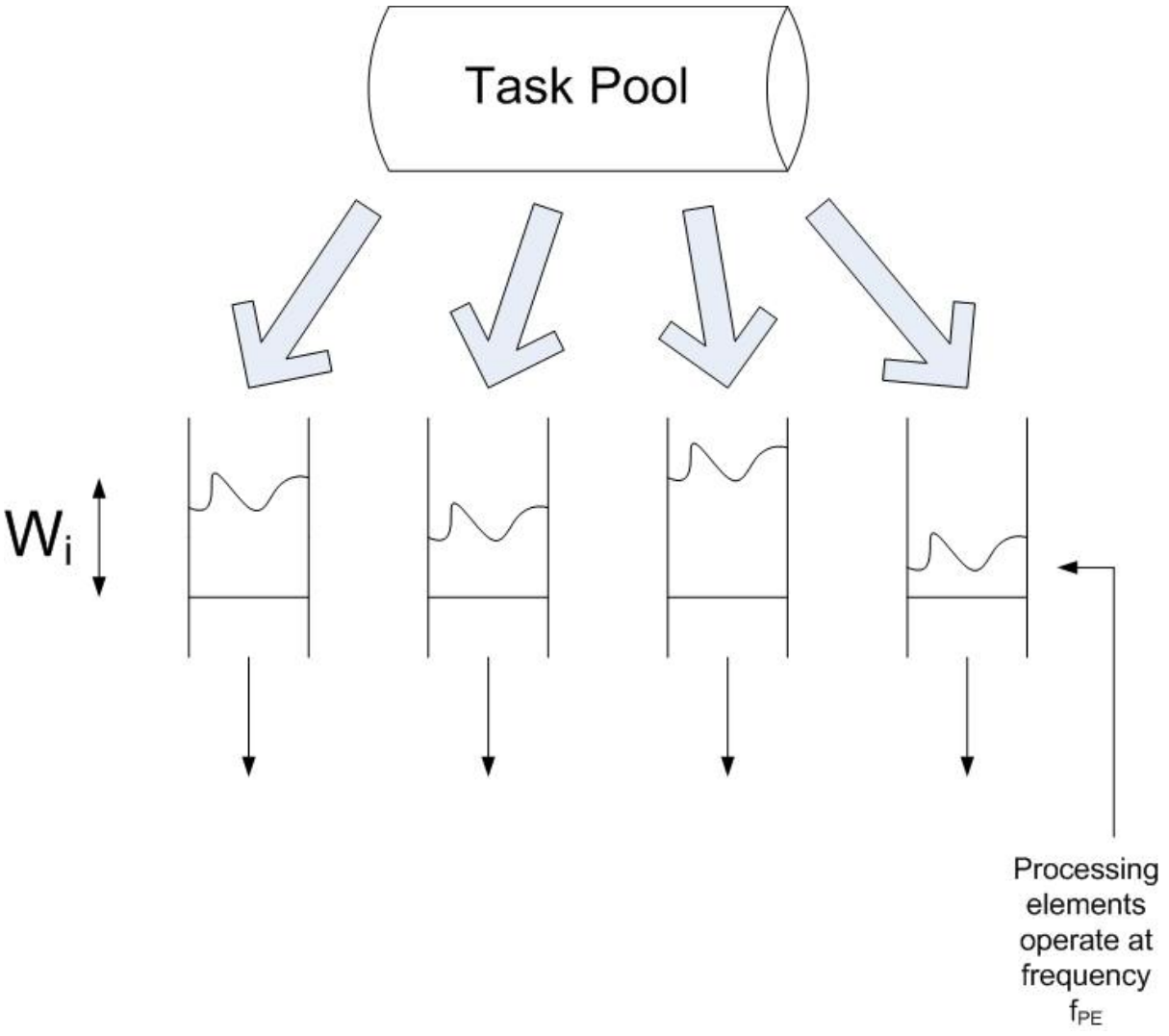


Figure 6.1: Improving latency with more processing flow paths

The next section describes how latency is improved in a prototype system which uses a DAPH for in-network processing. It is based on a novel location system known as DANTE, and the DAPH platform is used to reduce the latency experienced in querying the location system while providing deterministic handling of incoming radio events. The remainder of the chapter then examines the DANTE system in detail in order to justify the signalling protocol and the timing requirements of the pertinent threads of execution. An explanation of how the DANTE system is constructed for optimum energy-efficiency will be presented.

## 6.3 Latency Reduction in DANTE

Location systems provide information concerning the whereabouts of a person or object within a defined region of space to a certain prescribed level of accuracy. This sentient computing application was chosen for demonstration because it usually has real-time demands unlike many other sensor systems such as temperature loggers. It comprises tags attached to the items to be located and an augmented floor space capable of two-dimensional positioning using low-frequency (LF) electromagnetic waves. For energy-efficiency reasons explained later, while position sensing is performed using the low-frequency signals, each tag transmits its identity via UHF radio frequency (RF) signals. The time-critical threads in the design are:

1. RF receiver
2. Matrix position estimator
3. LF driver

The ability to perform fast in-network processing enables one to push applications into the hardware controlling the augmented floor itself. In the DANTE system, while one can simply obtain location data in a streaming fashion, as was necessary to measure accuracy and calibrate the system, it is possible to write simple applications which run independently on the DAPH, and can be called by an external application running on the host computer. The prototype system used a digital hardware development board connected to the augmented floor, the radio frontend, and the host system. A serial port(RS232)-to-USB converter was used for communication with the PC so it was only necessary to create standard RS232 protocol-compliant hardware within the development board. Although the relevant data was transmitted via a reliable connection in this prototype sentient system, many systems might have a low bandwidth connection to the external computer infrastructure or the energy cost of communication might be too high.

It might therefore be immensely advantageous to be able to perform queries from the host computer instead of streaming raw location data to it. Examples of such queries are:

- Which tag is closest to a point P with coordinates (x,y)?
- How many tags have been sighted in the rectangular region with corner points W,X,Y,Z?
- Are "tag A" and "tag B" in close proximity?

Although, this was not the primary goal of this experiment, a privacy-conscious reader will note that downgrading the location information or not providing raw real-time position data has the added advantage that it enhances anonymity. Apart from querying the DANTE system, an application can also register "callbacks" so it can be triggered on events such as when a tag crosses a particular line or enters a pre-defined region.

The DANTE system was rigorously tested with an increasing level of activity, in order to verify the claim that a multiprocessor system is better able to guarantee real-time performance or low latency. Apart from increasing the number of incoming radio events, a large number of queries were also generated by the host computer and sent to the DANTE hardware comprising a number of SpotCore processors working together. The query "Which tag is closest to a point P with coordinates (x,y)?" was repeated at random by the external application while the effect of a large number of tags was represented by increasing the tag update rate and creating pseudo-IDs on-the-fly within the development board (FPGA) connected to the augmented floor. Counters were then placed at strategic points within the hardware design on the FPGA so the latency and throughput could be recorded and sent back to the PC.

Although the actual computation of the query was straightforward, the results show that by adding more cores one can get a better performance, in terms of the latency and throughput, than by simply increasing the clock frequency of a single core. This is chiefly due to the fact that the thread managing the radio frontend and controlling the wire matrix has to be hard real-time (and thus has a high priority) but also requires more time relative to the queries which are received from the remote application. Thus the average latency reduces more significantly in a system with multiple cores running at the base frequency of 12.5MHz than in a system with a processor running at higher frequency(this test was performed at clock frequencies of 25MHz and 50MHz as seen in Figures 6.2 and 6.3). In addition, Figures 6.4 and 6.5 show the average throughput results obtained for the different scaling options.

The results demonstrate some of the benefits of using the Data Analysis and Processing Hub (DAPH) to perform more advanced processing within a sensor network —

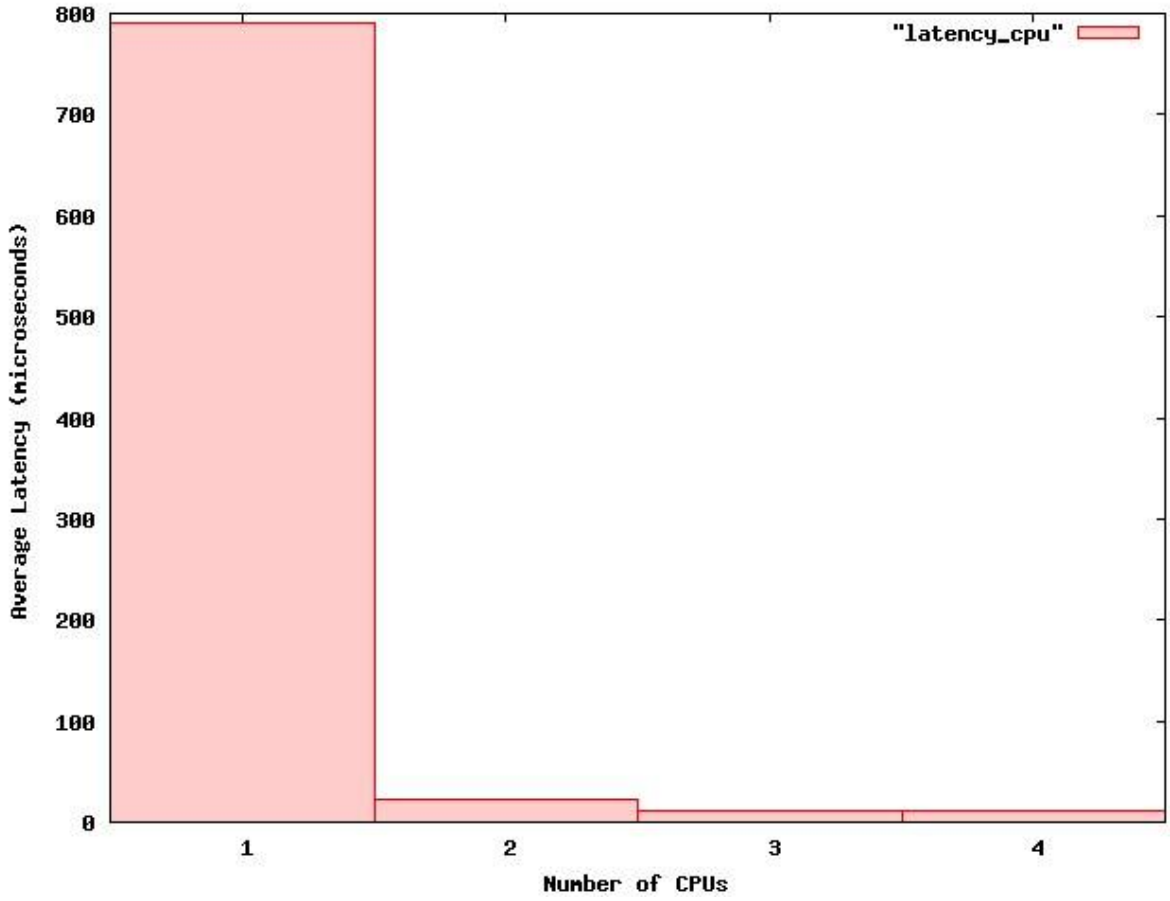


Figure 6.2: Variation in average latency with number of SpotCores

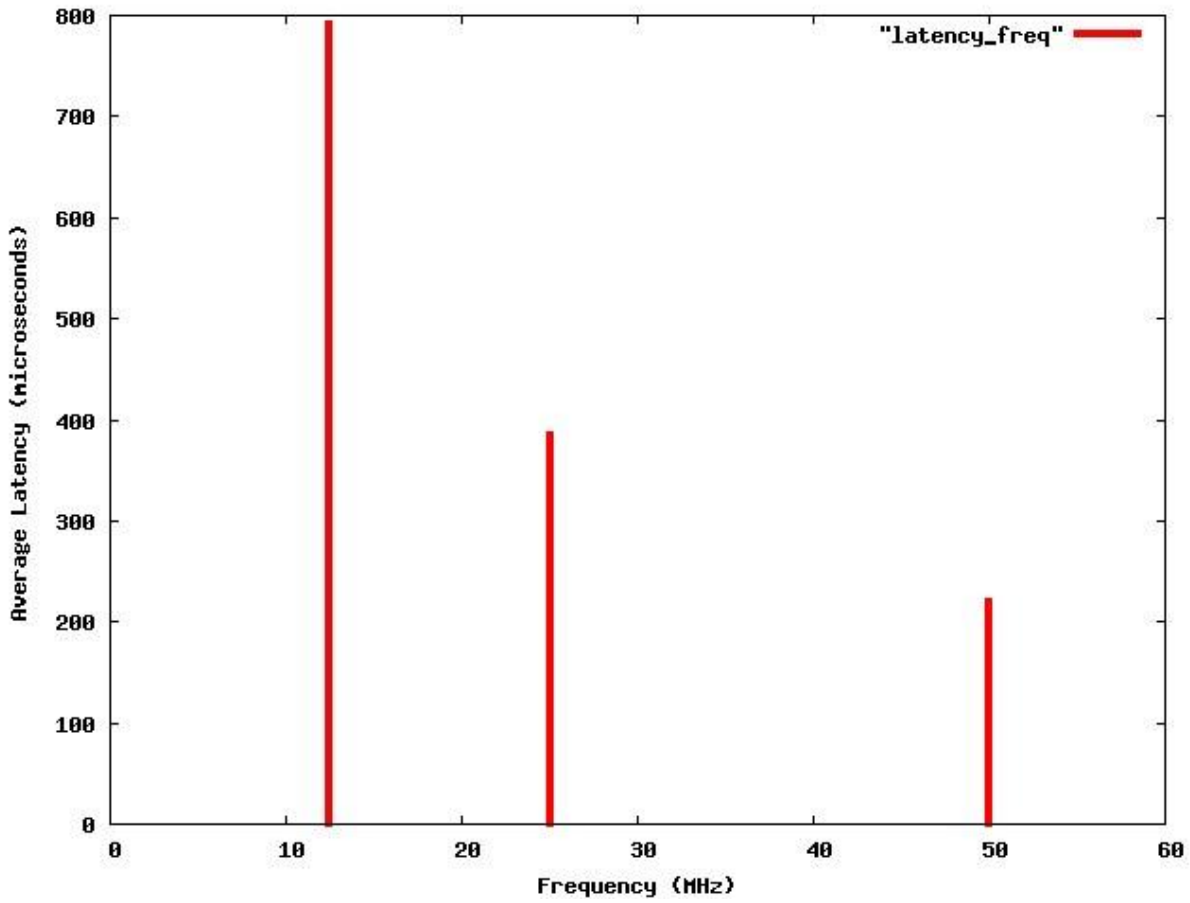


Figure 6.3: Variation in average latency with uniprocessor operating frequency

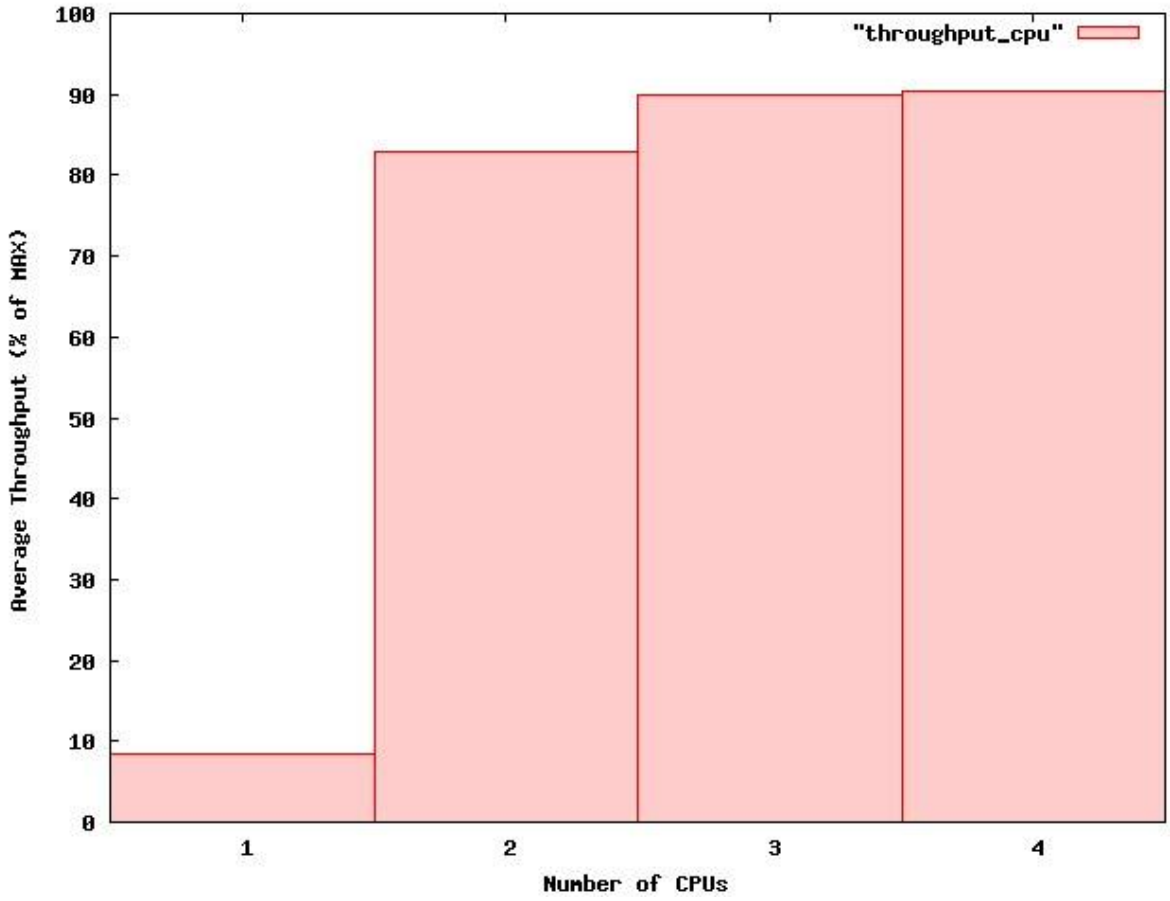


Figure 6.4: Variation in average throughput with number of SpotCores

less computational load on the external computer infrastructure, reduced communication between the sensor network and the external system, and lower latency on both the "sink-facing interface" and "sensor-facing interface" of the DAPH.

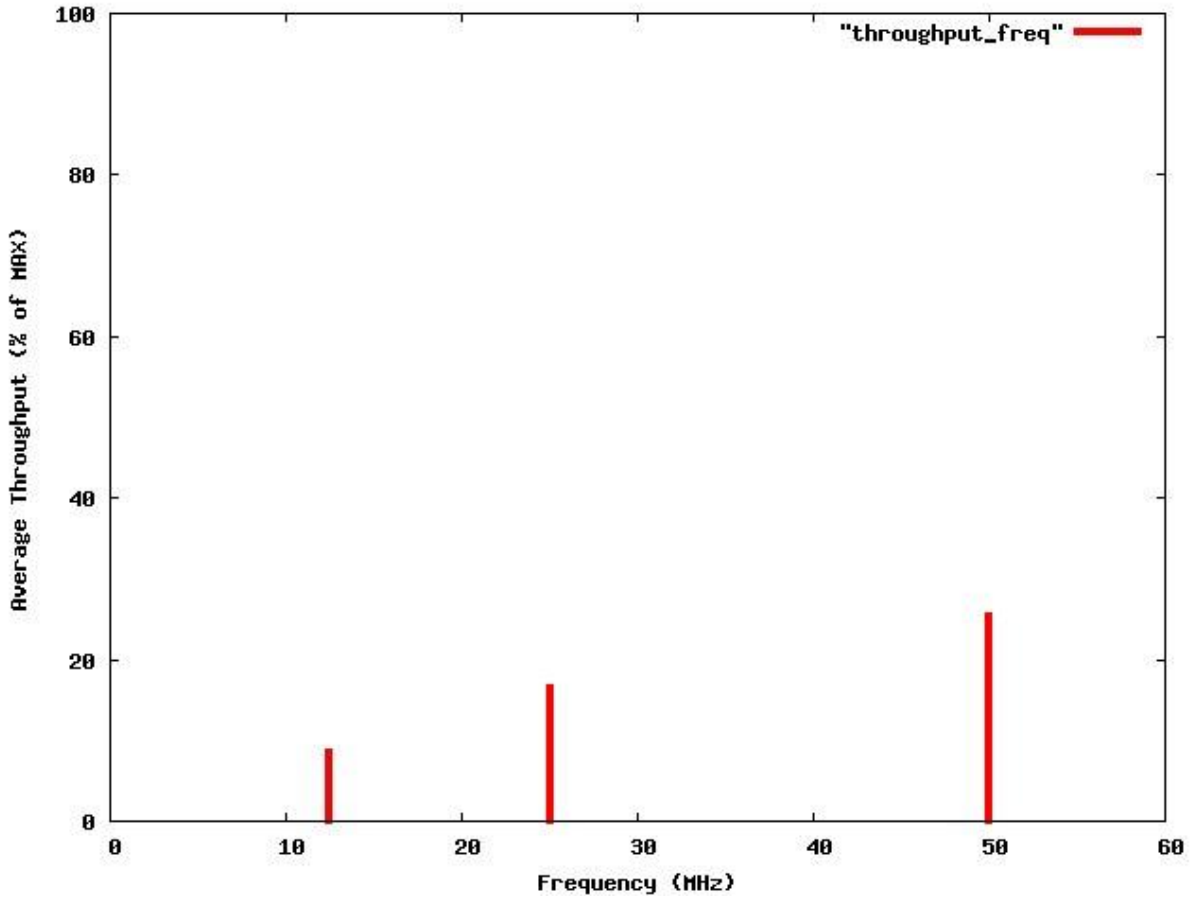


Figure 6.5: Variation in average throughput with uniprocessor operating frequency

## 6.4 The DANTE Architecture

The novel location system known as DANTE uses multiple antennas distributed over the floor surface and identification tags on mobile entities. There have been numerous systems developed for location estimation using a wide range of sensor technologies and transducers from infra-red [121], to radio [9] and ultrasound [50]; and the research area can be described as fairly mature. However, the key problem which currently faces most location systems is the fact that both the resolution and reliability cannot be increased without an expensive infrastructure or very high power requirements. That is, in constructing location systems, there is an implicit trade-off between the resolution of location data or number of entities to be located and the amount of power or number of infrastructure

elements required. Current systems have not fully addressed the problem of scalability which involves minimising the cost per unit area of coverage and the additional cost per user. For instance, while estimating the signal strength at various points in a Wireless Local Area Network (WLAN) can provide us with a relatively cheap way of obtaining an approximate location [21], the location accuracy is an order of magnitude less than a system such as the BAT system [50] which relies on measuring the time-of-flight of ultrasound signals and whose deployment is more expensive. The system described herein is both low-cost and low-power with the accuracy determined by the resolution of the antenna grid used.

Given that our feet are in contact with the floor surface most of the time, a natural way of locating people in two dimensions is to try to determine where they are relative to certain known positions local to the floor. CarpetLAN [42] tries to achieve both communication and positioning using transceivers embedded in the floor. Unfortunately, the setup is infrastructure-heavy as each carpet box is 10cm thick making retrofitting difficult. In addition each floor transceiver has a high power consumption of about 8W. The resolution of the system is 1m which is not particularly high relative to other location systems in the literature which are also capable of high-speed communication [21]. The design described does not deal with scalability issues and has some difficulty with multiple targets in close-proximity.

POISE [39] is a system which detects footfalls and footlifts by measuring the change in voltage due to the sudden rapid movement of static charge near a conductor or an electrode embedded in the floor. This static charge accumulates easily when shoes interact with a carpet. Unfortunately, the POISE system does not consider the case where more than one person is present and it faces serious identification challenges due to the unreliable nature of the tracking data when the sentient surface is deployed only over small regions. In addition, unlike the work presented here it does not use an energy-efficient embedded computing infrastructure but instead transfers the sensor data to a PC for processing.

The idea of sensing presence by using the fact that the human body is partially conducting and thus has the ability to modify the capacitance of an electrical circuit in close-proximity has been explored by several researchers. Zimmerman et al. [126] apply electric field sensing to a smart table and a person-sensing room. SmartSkin [95] applies the principle that the human body can act as a virtual ground and shunt signals to create an interactive table. Floor sensing was not considered in this case.

Other floor-based sensing systems do not rely on measuring current or voltage variations due to changing electric fields but measure pressure instead. These include Active Floor [13], Smart Floor [89], MagicCarpet [92], and ZTiles [97]. Unfortunately these systems tend to use expensive components or require a great deal of infrastructure. The

LiteFoot [46] uses a matrix of optical proximity sensors placed beneath the floor at a density of one sensor per 4cm. The primary application is recording dance steps. The sensors work either by detecting the shadow cast on them in the case where the floor surface is illuminated from above or alternatively by measuring the increase in reflected light.

Some researchers have suggested using RFID tags and associated readers for location determination. LANDMARC [87] uses a set of RFID readers in the environment with fixed reference tags to improve location accuracy by dynamically adapting to the environment. Kaddoura et.al. [62] investigate using RFID readers at certain points within a house to bootstrap the identification service in their pressure-sensitive floor location system.

Bohn and Mattern [26] present a system which uses tags embedded in the floor and mobile readers. The researchers advocate the arrangement with tags embedded in the floor by claiming the inverse increases the cost of the system, complicates deployment and maintenance, and compromises privacy controls if passive mobile tags are used. A lightweight and inconspicuous reader design is developed in this chapter and this can avoid the cost issues associated with large-scale deployment. In addition, a mobile RFID reader will require more power than the simple tags used in DANTE because the reader would need to energise the passive tags in the floor. It is also worth noting that tags which are generally fragile cannot be simply strewn on the floor but must be either encased or embedded in the floor in a way which does not leave them open to damage by being trod on.

The applications of DANTE include monitoring the way a particular region of indoor space is used which can lead to several optimisations in heating, lighting and other facilities. It can be used in entertainment similar to commercial game or dance mats. It can also be useful in healthcare, specifically with regards to the elderly by automatically monitoring their movements or lack thereof in the case of accidental falls.

The antenna matrix in the DANTE location system comprises a grid of small thin wires which can be woven into the carpet, embedded beneath the floor or simply laid out on the surface of the floor (and held in place by adhesive tape). Each wire element in the grid forms a logical channel. The spacing of the channels was chosen to be 10cm which is less than the average foot length of about 24cm (European). A finer mesh would increase the cost of the system without providing any significant additional location information (orientation information may be gathered nonetheless). However, if one is interested in building a location system without tags such a fine mesh would be useful in detecting presence by analysing the changes in capacitance as a human body passes over them. Though experiments were conducted on this feature in the course of the research project, it was not integrated into the DANTE architecture presented below

as it requires a denser hence costly wire mesh to work successfully. In addition, Harle [49] points out that "tagged" systems are preferred over "tagless" systems in general, because users might wish to retain control of the determination and dissemination of their location data. Although "tagless" systems are more acceptable aesthetically as some tags are cumbersome, and scale well as there is no additional cost per additional user, the common perception that they are overly invasive would have to be changed in order for their use to become widespread.

All the wires forming the antenna matrix are connected to a central hardware controller known as the Floor Manager (FMAN). Each FMAN is currently designed to handle up to 64 antennas. The FMAN determines the position and identity of each tagged entity and can communicate with desktop computers and servers present on a local area network and interested in location information (Figure 6.6).

The backbone cable runs along the edge of the augmented room and is used to collect the wiring and connect the grid of antennas to the FMAN. It is shielded from interference using a grounded metallic sheath. The length of wire required to form the antenna matrix is given by,

$L_{wire} = 2LW/d = 2A/d$

Where d is the spacing of the wires in the matrix, A is the area to be augmented and L and W are its length and width respectively. A length of copper wire with a diameter of $0.28mm^2$ costs about 2 pence per metre[1] which means the wire matrix will cost approximately 40 pence per $m^2$ (assuming wire spacing,d = 10cm). This is fairly cheap, and the cost scales linearly with the coverage area. A cost comparison of location systems can be found in [54]. A denser mesh is used in the research by [45] on large surface area electronic textiles and they show that their wire matrix can be woven into a carpet to create a location system.

## 6.5 Time-Critical Tasks in DANTE

There are two main approaches to locating a tag with a matrix of antennas. In the first approach, a tag will transmit its identity periodically and the system will perform a signal strength analysis across multiple receiving antennas to determine its location. The alternative approach is to transmit signals on each of the antennas either at different frequencies, or with different codes or at different times, and perform received signal strength estimation at the tag. In the case where a pilot signal is transmitted at different times on different antennas, the centralised controller can be dispensed with and an analogue delay line used instead to avoid having a high number of connections at the FMAN. This

[1] uk.farnell.com

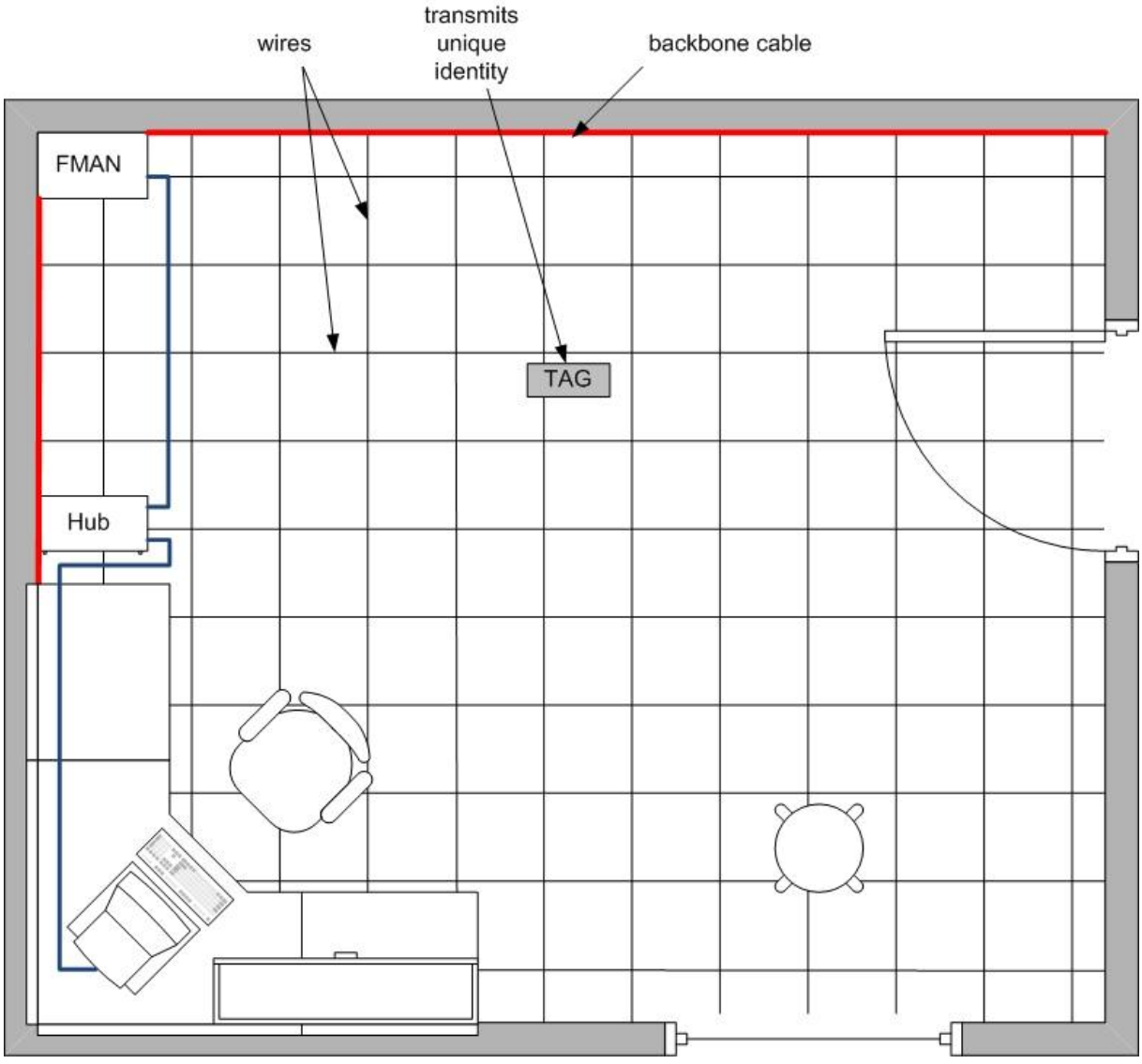


Figure 6.6: The DANTE location system

works by using a circuit which introduces a time delay between successive (daisychained) channels so that the common signal travelling from one channel to the other will be transmitted from different antennas at different times, which is the same effect observed when a centralised controller polls the channels. The sensing system required in the first approach (tag-signals-system) is shown in Figure 6.7. It comprises an analogue multiplexer which connects each channel to the analogue frontend in turn, a bandpass filter followed by or incorporating a high-gain buffer for amplification. The high-gain could simply be achieved by cascading three stages of a transistor amplifier with appropriate buffering between each stage to match the output impedance of one stage with the input impedance of the next with switches between stages so some stages can be bypassed by the controller if the output starts to saturate. The controller interfaces with the analog-to-digital converter (ADC), switches between channels, and selects the appropriate gain which should be applied to the incoming signal.

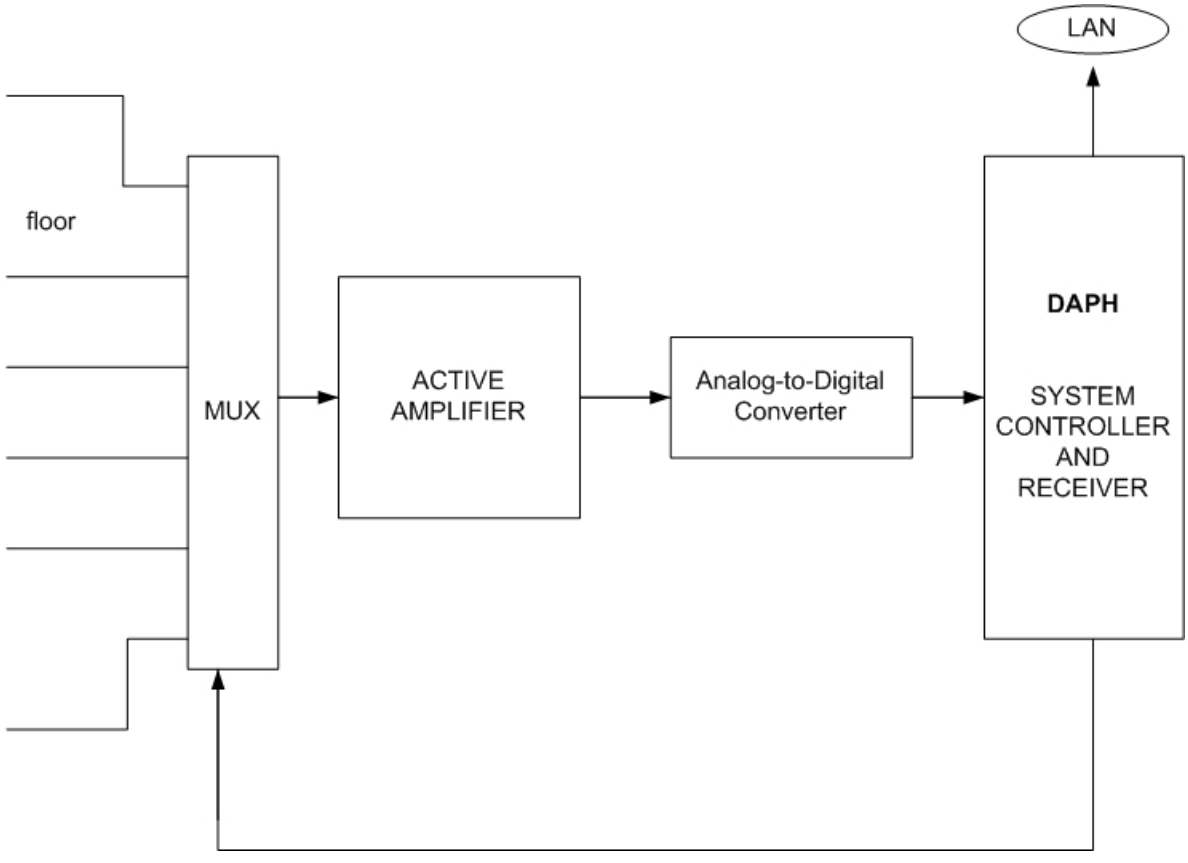


Figure 6.7: Core components of tag-signals-system architecture

This seems attractive because the system-signals-tag approach necessitates a separate radio channel to broadcast the tag identity along with its calculated position. However, for the random transmission multiple access technique which is explained in the next

section to be viable, the transmission time must be kept to a minimum otherwise the system is not scalable. This is problematic in the tag-signals-system scheme because the physical layer (see Appendix A) dictates that a relatively low frequency (LF) signal (around 10kHz) is used for position sensing. Due to the low signalling rate this might not be a scalable way of transmitting the identity code from the tag. However, in addition to this simple anti-collision mechanism the tag-signals-system provides a "spatial" collision detection scheme which works when tags transmitting simultaneously are not also co-located as seen in Figure 6.8a. Ideally, the system should find two active channels - one horizontal and one vertical, and it selects the channel with the stronger signal and listens to the rest of the transmission on that channel. If there are more than 2 active channels as illustrated in Figure 6.8b by the presence of more than one peak in any dimension, then a collision is detected. The controller can then make a note of it for monitoring purposes, wait (back off) for the duration of an entire frame, comprising a preamble and ID code and start scanning again. It is important to note that this only acts as a first defence against collisions as some may still get through — if the tags happen to use the same channels by being at the same position or if one tag starts transmitting towards the end of another tag's transmission, that is, after that tag's preamble, then the transmission received is much longer than normal and this can be easily detected.

Another problem with the tag-signals-system protocol relates to the total transmission time needed for an unsynchronised scanner at the FMAN to detect a signal. This means the tag should transmit continuously for at least as long as it takes the FMAN to poll all the channels. That is there is a requirement for the preamble, which precedes the tag identification in the transmission frame from the tag, to be longer than the scanning duration so no actual data is missed while the receiver locks onto the signal from the tag. This is not an energy-efficient solution and it greatly increases the probability of collisions.

The anti-collision scheme provided in ALOHA [12] relies on ensuring that there is a randomly selected interval between tag identity transmissions. It is shown here that for this to scale, the frame size must be made as small as possible. The scheme is implemented in practice by setting the microcontroller's watchdog timer value to a random value after each transmission. This sleep timeout value is bounded in order to ensure that the location update rate does not fall below 10Hz. The watchdog timer in the microcontroller on the tag has a nominal timeout value of 18ms. The timeout value varies slightly with supply voltage and temperature; a feature which is actually acceptable and indeed useful for our purposes as it adds more randomness. The CPU on the tag then randomly selects what is known as a "prescaler" in the range 1 to 5 and applies this to the watchdog timer. This implies the sleep time will be either 18ms, 36ms, 54ms, 72ms or 90ms.

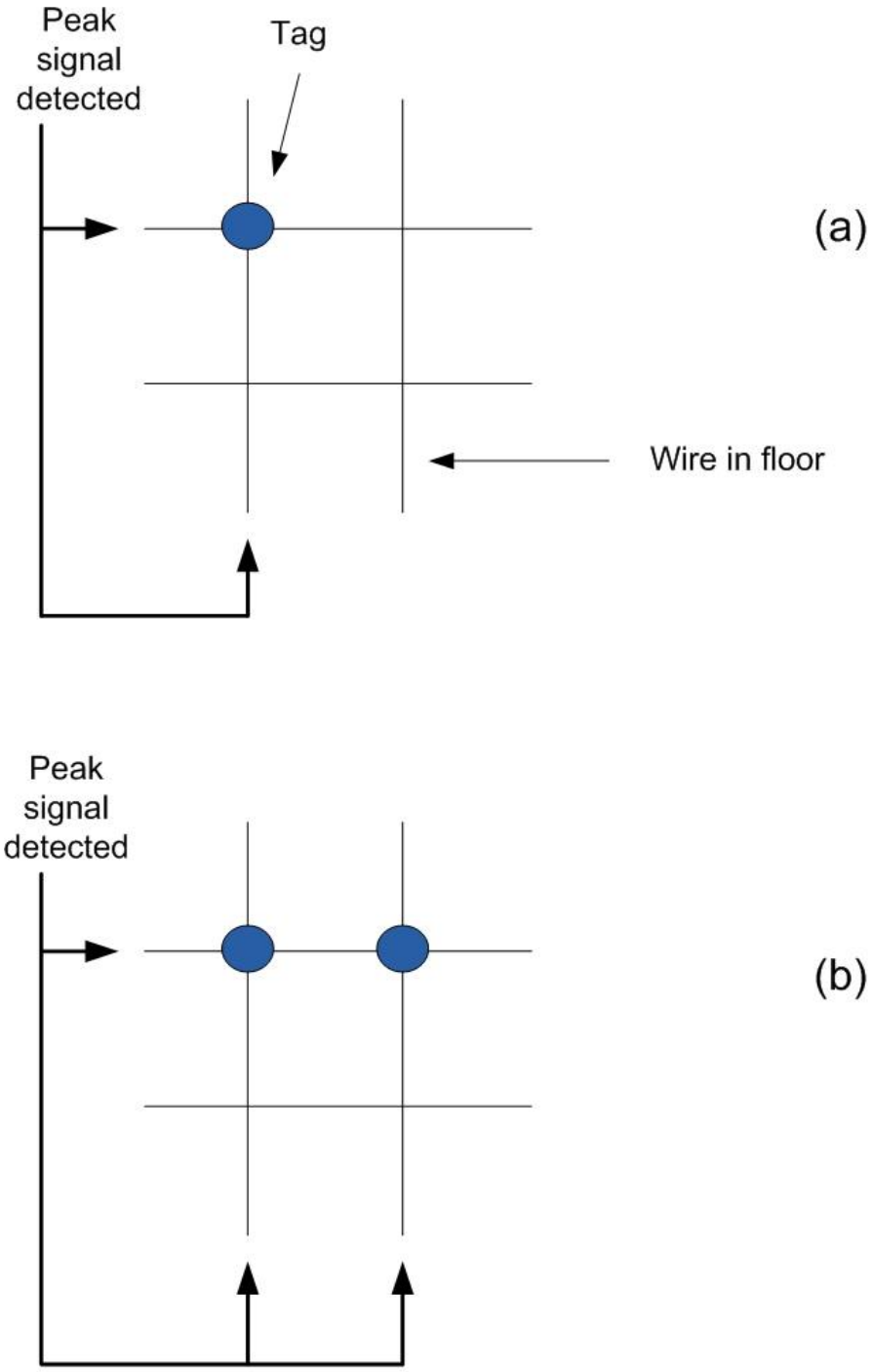


Figure 6.8: Tag collision: spatial view

The pure (unslotted) ALOHA medium-access protocol was selected for tag communications instead of slotted ALOHA because this saves energy as the tags do not have to listen for a synchronisation signal. Slotted ALOHA has better performance for a large number of tags but in practice the implementation of pure ALOHA gives sufficient performance for a small number of tags - limited by the maximum area covered by a single FMAN (about 5m x 5m). The trade-off is thus justified because the number of tags would not reach a level where the benefits of synchronisation in slotted ALOHA or some other more complex medium-access protocol would have been necessary. We shall now try to obtain an analytic expression for the probability that any tag identity transmission will be successful P(success).

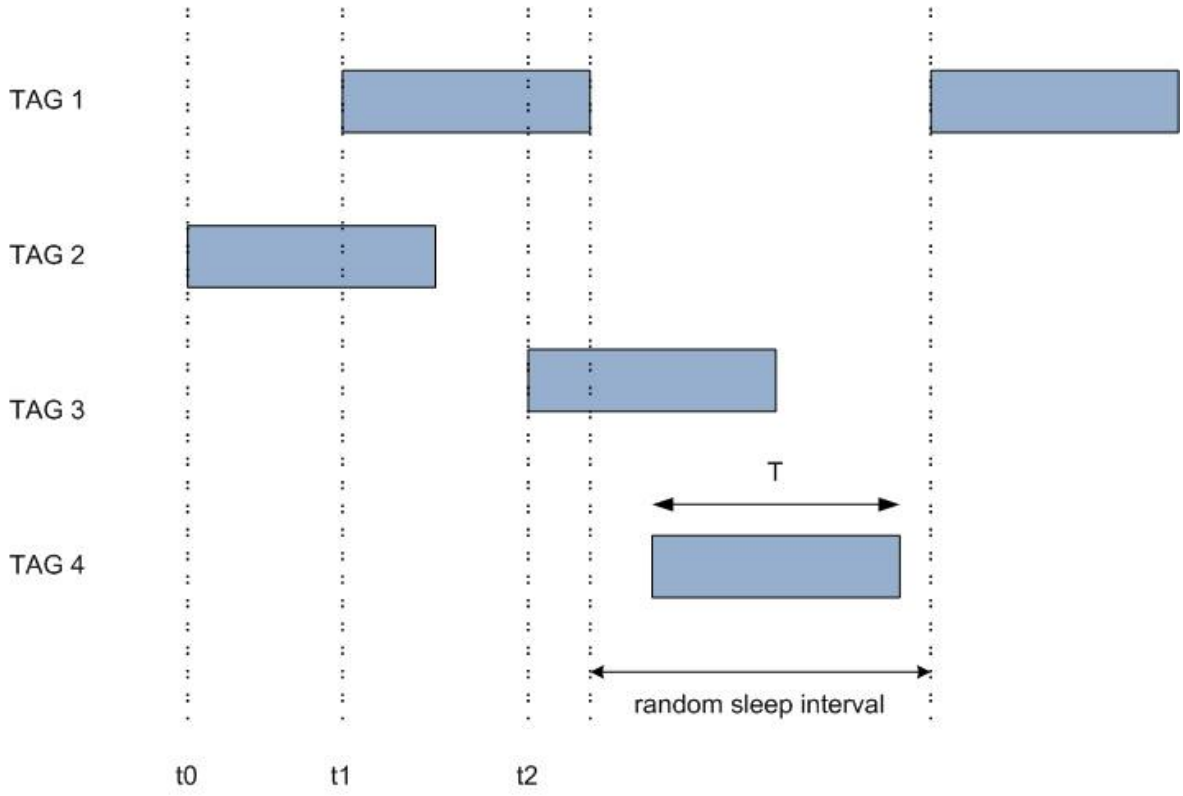


Figure 6.9: Tag collision: temporal view

As Figure 6.9 shows, each frame comprising a preamble and a 16-bit tag ID can overlap with one or more frames.

From the figure we can deduce that a collision will occur if one tag transmits at time t1 and any other tag starts transmitting at t where

$t > (t1 - T)$ or $t < (t1 + T)$

We can see that the transmissions of tag 2 and tag 3 collide with that of tag 1 because in this general case $t0 > (t1 - T)$ and $t2 < (t1 + T)$. If the probability density function

describing the tag transmissions is P(t) and the probability that a tag causes a collision is P then we can write.

$$P = \int_{t-T}^{t+T} P(t)dt$$

For a set of N tags, the probability of success is therefore

$$P(success) = (1-P)^{(N-1)} = \left(1 - \int_{t-T}^{t+T} P(t)dt\right)^{(N-1)}$$

It was experimentally determined that for a frame size of 10ms and maximum number of slots in the random selection of sleep durations, P(t) is normally distributed with a value of approximately 22 ($s^{-1}$). This is inversely proportional to both the frame size and the number of slots but hardware limitations prevented further optimisation of these values. For instance, the frame size is fixed by the rate at which the pilot low frequency signal switches between channels. In addition, no trade-off could be made with respect to the number of random slots because the update rate should be at least 10Hz but cannot be too high as it is necessary for the tag to sleep as much as possible in order to prolong battery life.

If P(t) = $\alpha$, then,

$$P(success) = (1 - 2T\alpha)^{(N-1)}$$

This is in good agreement with the results of a C++ simulation program written to test the scalability of this system.

Some research literature on the ALOHA protocol use a Poisson distribution to model the arrival rate, that is,

$$f(k;\lambda) = \frac{e^{-\lambda}\lambda^k}{k!}$$

(where k is the number of occurrences)

However, the normal distribution used in this calculation is shown to be a good approximation since the parameter $\lambda$ or rate, representing the average number of occurrences per unit time is sufficiently large.

As a result of the preceding analysis the protocol shown in Figure 6.10 was chosen for implementation.

The protocol in Figure 6.10 saves power at the FMAN by being event-driven, that is unlike the previous case the FMAN does not scan continuously but waits until prompted by the radio receiver circuitry before starting its transmissions which are then demultiplexed temporally and spatially. This improved implementation is shown in Figure 6.11.

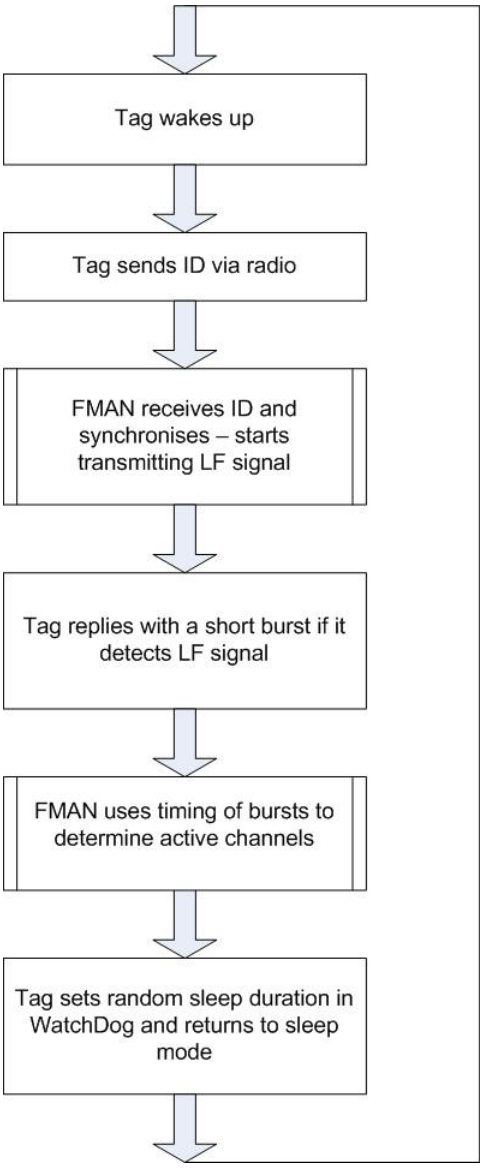


Figure 6.10: The DANTE protocol

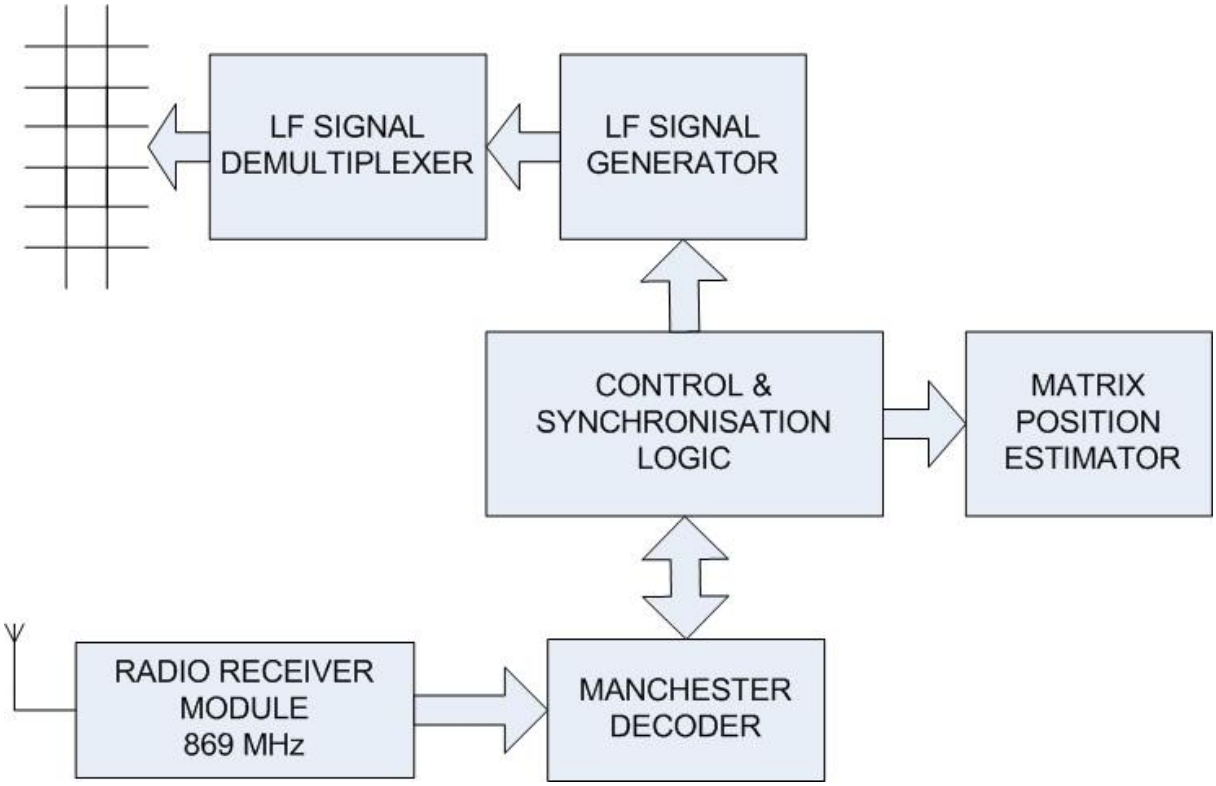


Figure 6.11: FMAN implementation

However, for a very large number of tags it would make sense for the FMAN to transmit continuously and have the tag record the signal strength values for different channels, compute the most active channel, and then wait for a random but precise interval before transmitting. The tag will then transmit a single frame consisting of its ID and the time that has elapsed since it detected a signal peak for both the horizontal and vertical channels in close proximity.

The data transmitted from the tag is encoded using Manchester encoding. It is very robust as it is a self-clocking code. That is, there will always be synchronisation or it will possible to recover the clock signal regardless of the data being transmitted. Although this increases the signal bandwidth relative to codes which are not inherently synchronised such as Non-Return-to-Zero-Inverted (NRZI), such codes will require a higher-level protocol incorporating features such as bit-stuffing. This would add overhead at the transmitter and at the receiver, and the fact that the frame size might no longer be fixed at a minimum value further complicates the design.

## 6.6 DANTE System Results

The prototype tag used for location determination is shown in Figure 6.12. The components of the tag are shown in Figure 6.13. It uses commercial-off-the-shelf circuit components and has a total cost of about £10. The packaged tag is clipped to a shoe as

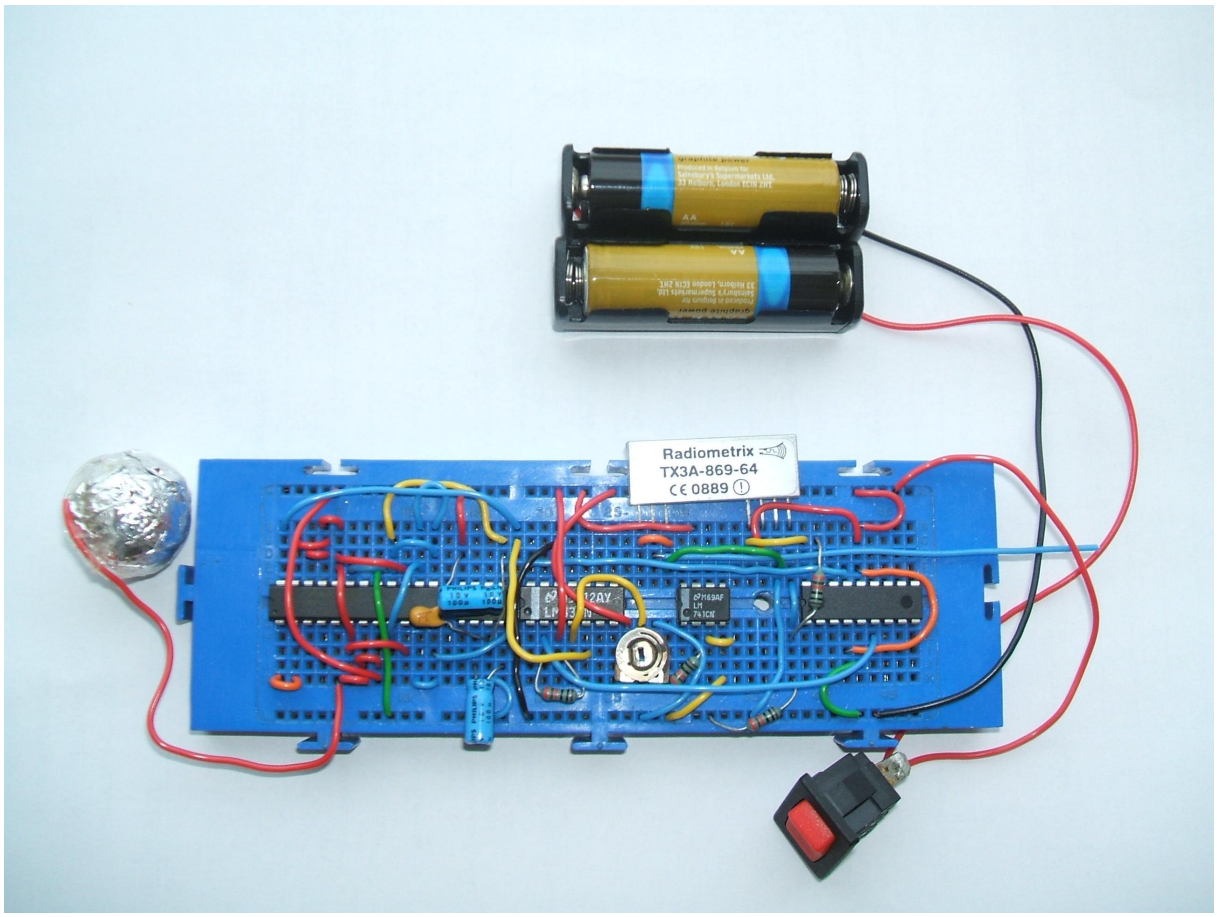


Figure 6.12: DANTE tag

shown in Figure 6.14. The tag receiver operates at the low (audio) frequency of 11.3kHz for reasons explained in the detailed physical layer description (Appendix A). To filter out noisy transmissions around this frequency a bandpass circuit with a high Q (Quality factor — defined as the ratio of centre frequency to the absolute difference of the corner 3dB frequencies) of 64 was utilised. This was implemented as two cascaded second-order analog filter sections. Ideally, one would place an analog-to-digital (ADC) converter after the band-pass filter in the tag circuit to obtain some estimate of the signal strength but in order to save power, such an ADC which typically consumes between 10 and 50 milliwatts, was not used in this design. The tag effectively used a fixed threshold and modulated the length of the reply so it was proportional to the strength of the signal.

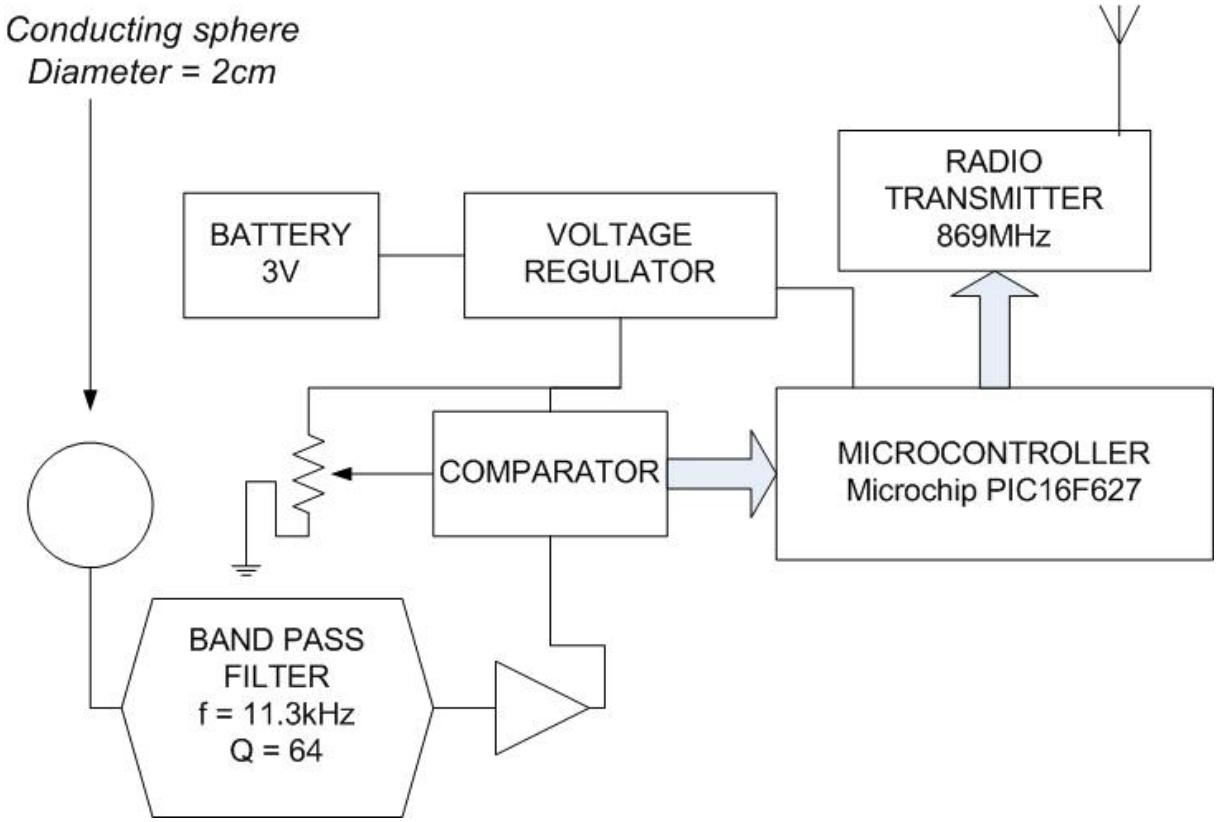


Figure 6.13: DANTE tag circuit components

The DANTE tag operates on a 3V supply. The current consumption of the major components is shown in the following table.

| *Component* | *Current (mA)* |
|---|---|
| Microcontroller | 1 |
| Radio transmitter | 7 |
| Band-Pass Filter | 13 |

The current during standby or sleep mode was less than $100\mu A$. Since the tag operated for only 10ms at a time, at an update rate of 10Hz the average power is only 6.3mW. A *jitter* switch can be used to trigger the circuit only when there is some movement further reducing the power consumption.

Figure 6.14: DANTE tag and Floor Manager

The DANTE system was tested as follows. First, the underside of a conventional roll of carpet was lined with cheap thin wires as shown in Figure 6.15. In order to make

testing easier and consistent, the top surface was then marked with tape so that the matrix formed coincided with the wire matrix underneath the carpet as shown in Figure 6.16.

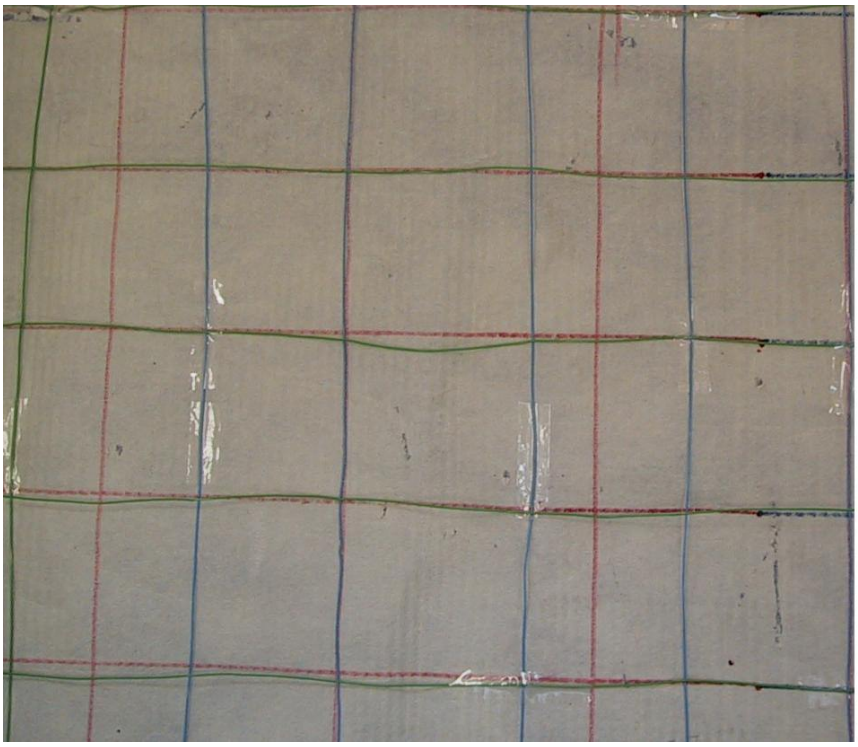

Figure 6.15: Wire matrix laid underneath the carpet

In order to verify that the channel regions were properly segmented, that is, the low frequency signal transmitted by the FMAN was received by the tag at a distance equal to half the channel spacing and dropped significantly at a greater distance, the tag was positioned as shown in Figure 6.17. The results of this preliminary experiment are plotted in Figure 6.18 in terms of the average duration of the reply from the tag.

The augmented carpet occupied a $4m^2$ region of the lab floor space. Sixteen channels were used in this experiment but up to 64 channels are possible with a single FMAN before the connections become unwieldy. The DANTE protocol discussed earlier was implemented as threads running on a system with up to 4 SpotCore CPUs with some complementary logic. The Verilog hardware design was then synthesised and downloaded to an Altera Field Programmable Gate Array (FPGA) board which was connected to the analog frontend comprising demultiplexers, signal buffers, and an 869MHz radio receiver. The 16 channels were split into 12 rows and 4 columns giving a matrix with a total of 48 crosspoints. Each point was visited in turn and the DANTE hardware reported the calculated position via a Universal Serial Bus (USB) connection to a computer. Each position

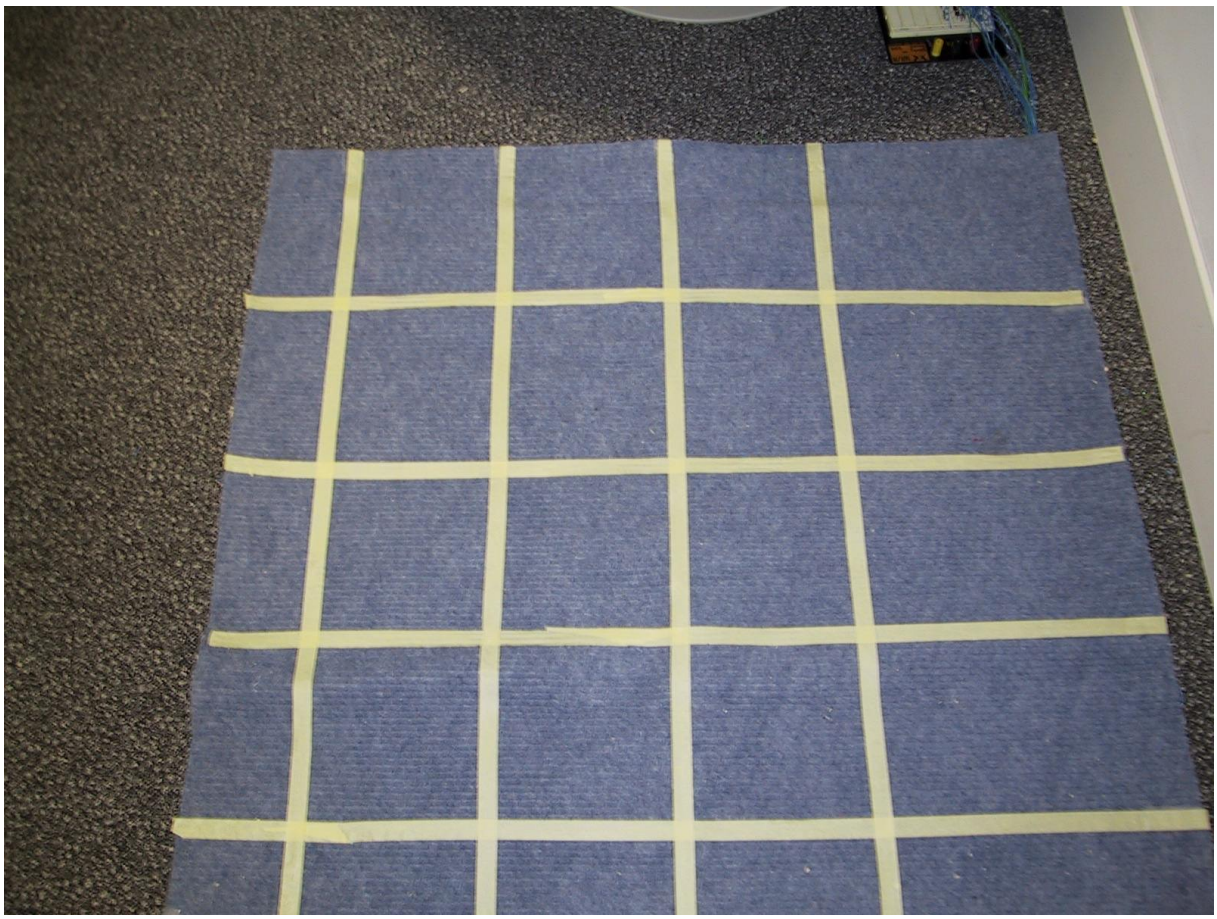

Figure 6.16: DANTE experimental carpet surface

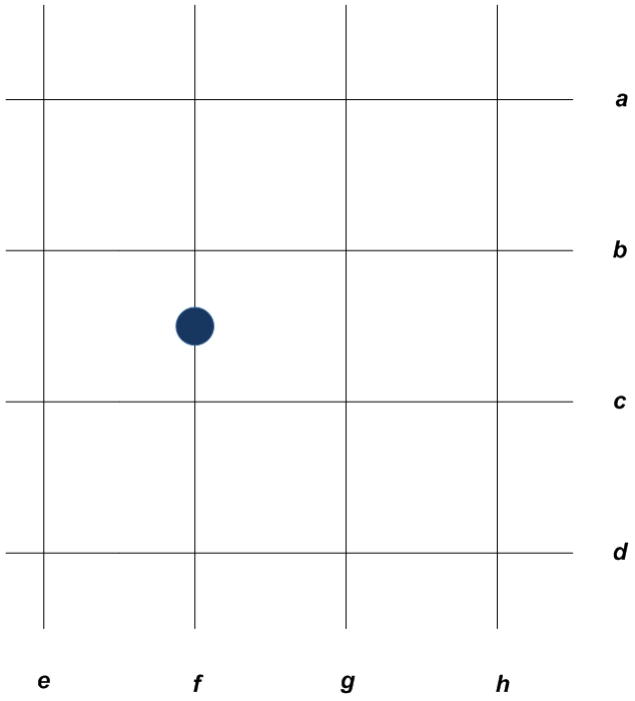


Figure 6.17: Testing channel overlap

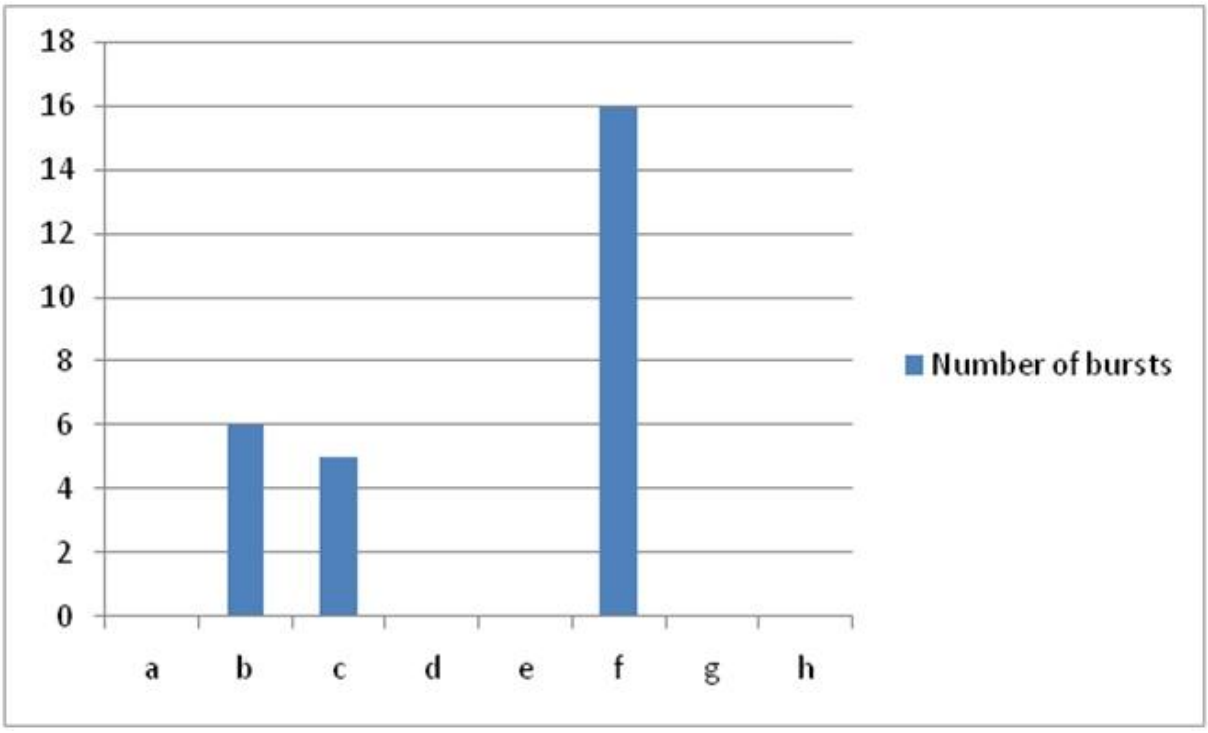


Figure 6.18: Channel signal strength data

was then displayed along with a history of points as shown in the example in Figure 6.19. The system always correctly identified when the tag was above each crosspoint. If the tag was placed between two channels then the reported position was always one or the other but this was non-deterministic giving the system a positioning accuracy of 10cm.

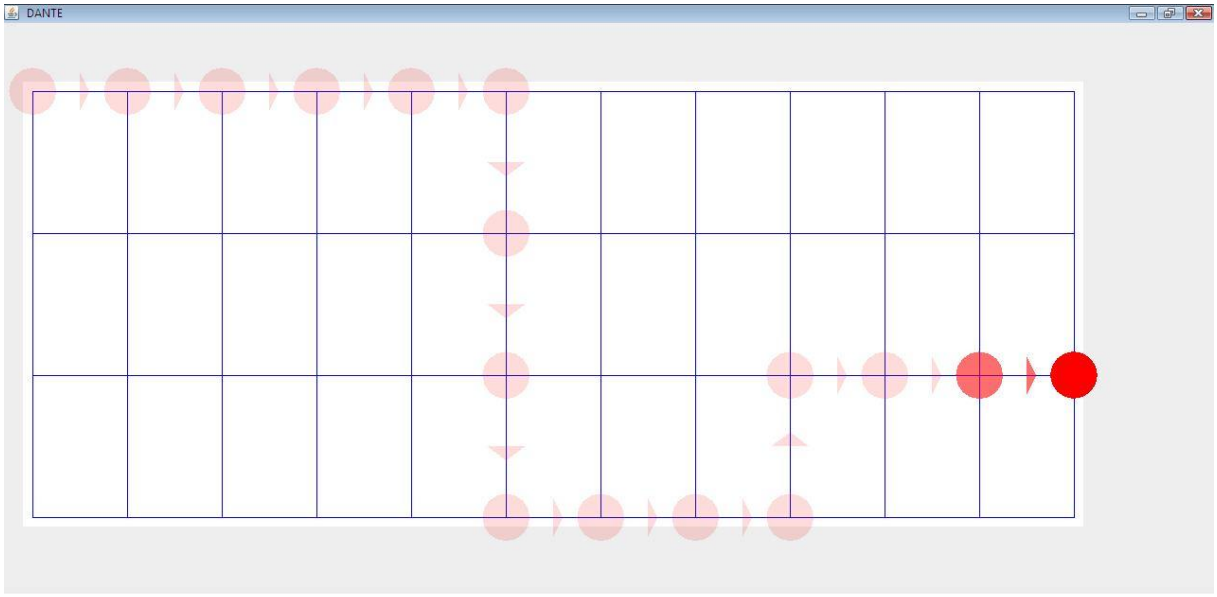


Figure 6.19: DANTE Floor Viewer

## 6.7 Summary

This chapter has presented a sentient computing system using the DAPH platform to reduce latency. It shows that in-network processing is possible with the DAPH architecture and can be more energy-efficient than an approach which simply relies on utilising faster single core platforms. The characteristics of a novel location system based on capacitive-coupling are described. The experimental data obtained was in good agreement with the expectations of the preliminary analysis in Appendix A. This chapter also elucidated some important location system design trade-offs involving cost, resolution and power consumption; and provided an in-depth exploration of the design space of the network protocol in the DANTE system with respect to time-critical tasks. It seems very likely that future progress in minimising end-to-end latency in sensor networks will involve some form of multiprocessing.

# Chapter 7

# Conclusions and Future Directions

This thesis has described the construction of a multiprocessor system aimed at improving sensor networks. It has evaluated this design's performance with a classic algorithm in sensor-driven computing and discussed its utility in an energy-efficient indoor location system design.

Chapter 3 established that the CPU used was lightweight and small but still capable of computationally intensive tasks such as digital filtering which is very important in sensor networks. An efficient and compact (16-bit) instruction set was defined, and performance improvements through instructions such as the LOOP instruction, which are unavailable in popular instruction sets, were discussed. The synthesis results confirmed that the CPU design was small enough so that instantiating it multiple times on a single chip would not be too costly in terms of the silicon area requirement.

The need for fast interprocessor communication hardware was explained in Chapter 4 and the innovative design which reduced the latency involved in managing threads was elucidated. Since multiprocessing is an advanced research area driven mainly by the supercomputing community and with many implementation routes depending on the criteria being optimised, the design decisions in this thesis were justified as extensively as possible. This was because sensor networks have a different set of paradigms from traditional high-performance supercomputing. This chapter discussed ways of maximising Quality-of-Service (QoS) or at least guaranteeing that a QoS contract between the programmer and the hardware is always upheld. Other aspects of the multiprocessor system design such as a unified approach to interrupts, prevention of priority inversion, memory protection and deadlock detection were also elucidated. These were unified in the sense that they were designed to work across a number of cores from the ground-up unlike many existing solutions which are designed primarily for multithreading on a single core and hence prone to efficiency problems.

Chapter 5 shows that a routine critical to sensor networks can be sped up with some careful consideration and the addition of a small set of primitives. Some of the diffi-

culties in parallel composition particularly in making tasks sufficiently independent are highlighted. The process of creating independent "work units" will invariably add some overhead to the original sequential program. However, this is acceptable if enough parallelism can be extracted and this limit of parallelization is captured by Amdahl's law which can be summarised as — the amount of speedup possible is limited by the section of the program which cannot be parallelized. The scalability of the DAPH is compared with the approach taken in conventional processor systems and the standard ways of elevating performance are found to be less energy-efficient. While the energy benefits may seem marginal in situations where performance is the ultimate attraction, or where developers are not willing to consider the move to parallel software, such deployments are inappropriate for sustainability and in sensor networks which are energy-constrained by design, there might be no alternative.

In Chapter 6, the integration of the DAPH in the DANTE location system which uses capacitive-coupling for sensing position, facilitates the creation of a number of threads for processing the incoming real-time data from the radio front-end while maintaining the responsiveness of the system to other external events. It is shown that a multicore system clocked at a slower rate achieves better minimum latency than a single faster processor. Reducing the operating frequency has power-saving advantages but being able to reduce the supply voltage as well is even better because it actually reduces energy consumption. Unfortunately, semiconductor device physics tells us that reducing the operating voltage of a circuit makes it slower. This is where the strength in the approach presented in this thesis lies — as long as we have partitioned our problem properly — running slowly is not an issue. The additional cores can simply be clock-gated or even powered down when not needed. The TopDog holds this information (which threads are active or are about to wakeup) and can easily transfer it to a system power management module or controller.

In summary, this thesis has contributed some solutions to important questions in sensor network research. It has described how energy benefits may be derived from the addition of supernodes to conventional homogeneous sensor networks, shown how an embedded multiprocessor can be designed for deployment in an energy-constrained environment, developed lightweight hardware to improve the energy-efficient integration of multiple cores, and scaled the performance of an example sentient computing application; with the added benefit of reducing the communication between sensor network and the external network.

## 7.1 Future Work

Future work needs to move towards using the DAPH (cluster of small processors) as a single node in a system comprising a larger network of processors on a chip. In this way, the multiprocessor architecture can continue to be scaled while optimising the performance-energy ratio at each node. One application area of interest is neural networks which have a highly parallel algorithmic structure.

Future work will also involve identifying other key applications for the DAPH platform and evaluating its scalability in these domains. For this to be successful, it will be necessary to add some hardware support for floating point operations and a C compiler for the SpotCore processor architecture. In addition, an elaborate programming environment such as that provided with the sensor node platform called Sun Spot [2] from Sun Microsystems would be advantageous. The Sun Spot platform features an ARM9 processor running an optimised Java Virtual machine without any operating system. Given that obtaining better performance requires some visibility, the TopDog can also be easily instrumented so that the system events such as "forks" are logged for analysing performance.

Looking further ahead, continued scaling of semiconductor device geometries to designs with a minimum feature size of 45nm or even less will enable an astronomical number of transistors to be created on a single integrated circuit, possibly over 100 billion according to current estimates [10]. While using such technology for sentient computing will represent a departure from the design principles promulgated in this thesis because the power dissipation may become uncontrollably high due to leakage currents, the implications are certainly worth considering. For instance, given the small size of the SpotCore processor design, one might be able to incorporate thousands of them in a single chip. Unfortunately, manufacturing difficulties will mean that not all of these processors will function correctly and many failures will occur during actual usage of the device. It is likely that a possible solution to this condition might be linked to the approach used in this thesis, that is, a small amount of auxiliary hardware functions like the TopDog, and can be used to examine the processor state, monitor its operation through assertions or by regularly injecting specially crafted codes into the instruction stream, and confirm the health of each CPU or a small cluster of CPUs. Like the TopDog, these monitoring engines will also facilitate (speed up) the transfer of threads from ailing cores to healthy ones so that no observable loss of performance occurs. The goal of the instruction set used in this thesis was to eliminate as much redundancy as possible, but if used in such an inherently unreliable execution environment it will become necessary to reintroduce some redundancy in the encoding for fault-tolerance.

Shared memory is clearly more advantageous than message-passing in applications such as the FFT where there is a large number of data dependencies, but as multiprocessor systems get larger it becomes more beneficial to adopt a message-passing interprocessor communication methodology. However, the actual point at which the shared memory model breaks down is still hotly debated. More hardware implementations and benchmarking will be needed to test this.

The chapter on scheduling also opened an exciting research area in hardware-based concurrency management that has not been thoroughly explored before from an energy-efficiency perspective. More work will be needed to examine the problem of fairness and answer the important question raised by this work — does the necessity for system *stability* and *determinism* preclude best-effort *fairness* policies? It will also be important to prove some of the properties of the scheduler formally. For instance, are all thread management operations truly atomic? While the TopDog hardware did not show any unexpected behaviour or has thus far been debugged to the best of the designer's abilities, it would be prudent to run the design through a theorem-prover. The SpotCore processor design might also need such formal analysis.

# Appendix A

# DANTE Location System Design

The core of the design of the location system is based on the ability to perform accurate *relative* signal strength estimation across many antennas (channels). Each tag possesses a small conducting sphere to create a uniform electric field and form a capacitor with the wires in the floor. The predominant frequencies in the signal transmitted are below 100 kHz, so the communication occurs in the near-field region. This is because the antenna dimensions are much less than the near-field radius R, which is given by $R = \lambda/2\pi$ , where $\lambda$ is the wavelength. The near-field region is the region close to an antenna where either the resulting electric or magnetic fields dominate. In the far-field region there is a fixed relationship between the magnitudes of the electric and the magnetic fields and energy propagates as radiating electromagnetic waves. Using the near-field region has several benefits in this application. Since, in general, the field strength decays very rapidly in the near-field, near-field communication has the advantage of inherent security, and it causes less electromagnetic interference. There are also relatively fewer restrictions in the electromagnetic spectrum usage regulation for short-range low power transmissions at that frequency. At the low frequency used in the tag design, the wavelength is of the order of a few kilometres so signals can propagate better with less reflection, diffraction and scattering which are common at higher frequencies and lead to multipath propagation and the problems associated with it.

Capacitive-coupling was chosen instead of inductive-coupling which is used in most RFID systems [40], because it satisfies the design constraints more efficiently. That is, wires are simple, cheap and relatively inconspicuous when placed as a grid on the floor compared to magnetic field antennas which by necessity have to be in the form of a coil or wire loop. In addition, with careful consideration one can design the tag antenna so that the field strength decays more slowly than the general case for a magnetic antenna. The magnetic field in the vicinity of a current loop falls according to $1/r^3$ where r is the distance from the centre of the coil. Besides distance and transmission frequency, the power transfer in the magnetic scenario is affected by the orientation of the tag relative

to the reader in the floor, and the number of turns of wire used, in both the reader and the tag. In the case of the electric field, using a spherical antenna arrangement, the transmitter's field is roughly proportional to $1/r^2$ irrespective of orientation and the capacitance of the system can be improved by increasing the radius of the conducting sphere which is ultimately constrained by the tag's small size. One disadvantage of using electric field transmission is the fact that the system is slightly more susceptible to ambient noise than in a magnetic field transmission. This is mitigated by the analogue frontend of the receiver, as well as robust encoding. This is an acceptable trade-off as it keeps the design of the tag simple and maximises the power transfer from the tag which optimises battery life.

The capacitance between a tag and an arbitrary point on a wire embedded in the floor can be calculated as follows from first principles starting with Maxwell's equation of electromagnetism which describes a relationship between electric flux density D and charge density $\rho$.

$$\nabla \cdot D = \rho$$

Transforming this with Stokes theorem gives

$$\oint D \cdot dS = Q$$

which is essentially Gauss's law specifying that the surface integral of the flux density is equal to the enclosed charge Q. This expression can be easily reduced when dealing with a point charge but the same treatment also holds for a spherical conducting surface of even thickness; as the field is uniform in all directions. That is,

$$\oint D \cdot dS = D\left(4\pi r^2\right) = \epsilon E\left(4\pi r^2\right) = Q$$

thus,

$$E = \frac{Q}{4\pi\epsilon r^2}$$

where $\epsilon$ is the permittivity of free space, E is the electric field strength and r is the distance from the centre of the sphere. However, the electric field can be linked to the potential V using the fundamental relation:

$$E = -\nabla V$$

We can then solve for capacitance as follows:

$$E = \frac{Q}{4\pi\epsilon r^2} = -\nabla V$$

The potential difference between a point on the surface and a point at a distance r from the centre of the sphere of radius r0 is

$$V = -\int_{r_0}^{r} (E)\, dr = -\int_{r_0}^{r} \left(\frac{Q}{4\pi\epsilon r^2}\right) dr$$

$$V = \frac{Q}{4\pi\epsilon}\left(\frac{1}{r_0} - \frac{1}{r}\right)$$

From the well known relationship,

$$Q = CV$$

we get

$$C = \frac{4\pi\epsilon}{\left(\frac{1}{r_0} - \frac{1}{r}\right)}$$

However, we need to extend this for a set of N points along each wire by recognising the fact that capacitances in parallel add up. Doing this we get

$$C = \frac{2L}{N}\left(\sum_{k=-\frac{N}{2}}^{\frac{N}{2}} \left(\frac{4\pi\epsilon}{\left(\frac{1}{r_0} - \frac{1}{r_k}\right)}\right)\right)$$

and

$$C = \frac{2L}{N}\left(\sum_{k=-\frac{N}{2}}^{\frac{N}{2}} \left(\frac{4\pi\epsilon}{\left(\frac{1}{r_0} - \frac{1}{\sqrt{(k*\delta l)^2+h^2+c^2}}\right)}\right)\right)\delta l$$

where L, h and c are defined in Figure A.1, and $\delta l = \frac{2L}{N}$

The expression for capacitance has been worked out and represented in discrete form which is amenable to numerical analysis and could be used as a rapid way of verifying the theoretical limits of the system.

If we number the different antennas or channels sequentially starting with channel 0 at a linear position x = 0, then the capacitance of any channel Ci can be expressed in terms of the linear position x as follows:

$$C = \frac{2L}{N}\left(\sum_{k=-\frac{N}{2}}^{\frac{N}{2}} \left(\frac{4\pi\epsilon}{\left(\frac{1}{r_0} - \frac{1}{\sqrt{(k*\delta l)^2+h^2+(x-i*d)^2}}\right)}\right)\right)\delta l$$

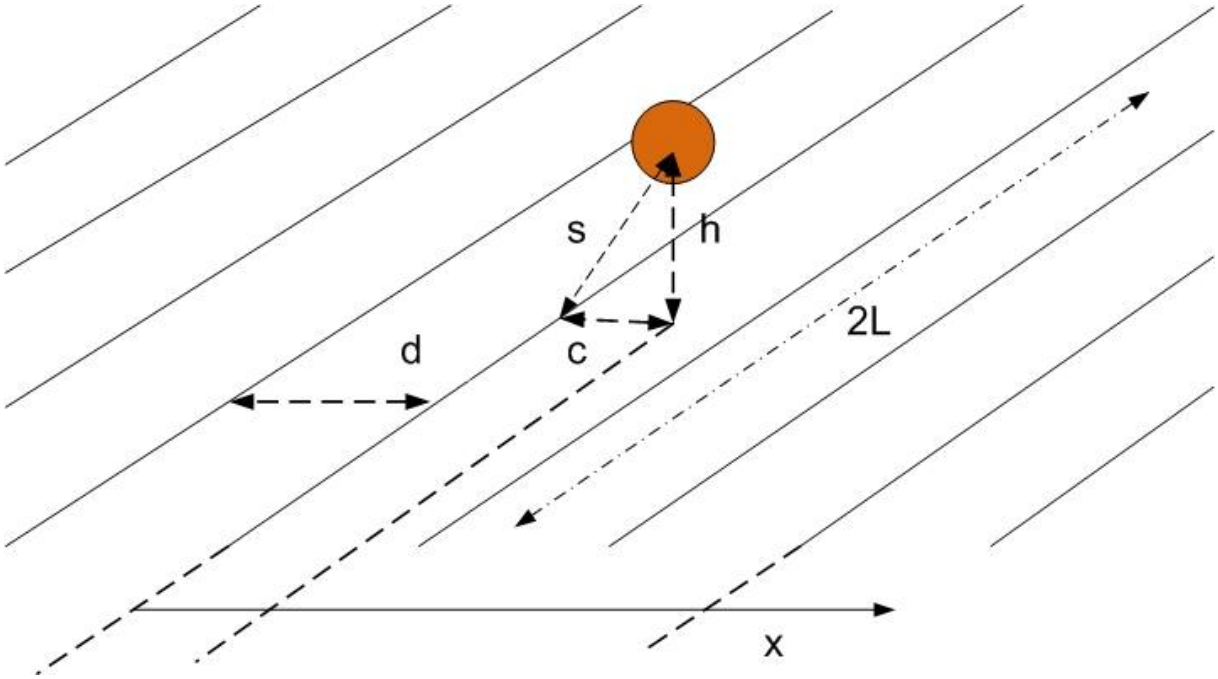


Figure A.1: Capacitance parameters

If we place the tag directly above, say, channel 4, then a simulation in C++ results in the following plot (Figure A.2) for 8 channels, L = 2.5m, r0 = 1 cm, h = 5cm, d = 20 cm.

Let us now examine another important motivation for adopting the system-signals-tag protocol presented in the thesis. Figure A.3 shows the circuit components, notably the capacitances, which arise between the spherical conductor on the tag and the signal generator in the FMAN. Let the voltage on wire be $V_{wire}$ and the voltage on the tag's front-end be $V_{tag}$ then the relationship between $V_{wire}$ and $V_{tag}$ can be expressed in terms of the capacitances as follows:

$\mathrm{V}_{tag} = (C_t/(C_e + C_t))V_{wire}$

Where $C_t$ is the capacitance between the tag and a single wire element, $C_e$ is the capacitance of the sphere relative to earth and $C_x$ that of the wire relative to earth. However, if the scenario is reversed such that we have the tag transmitting a signal which is picked up on the wires in the floor then we get the following expression for the voltage seen by the FMAN.

$\mathrm{V}_{wire} = (C_t/(C_x + C_t))V_{tag}$

Since the length of the wire spans a distance several orders of magnitude longer than the diameter of the conducting sphere, $C_x$ is much greater than $C_e$ which proves that, the energy transfer between the tag and the FMAN is more efficient in the situation where the FMAN transmits a pilot signal.

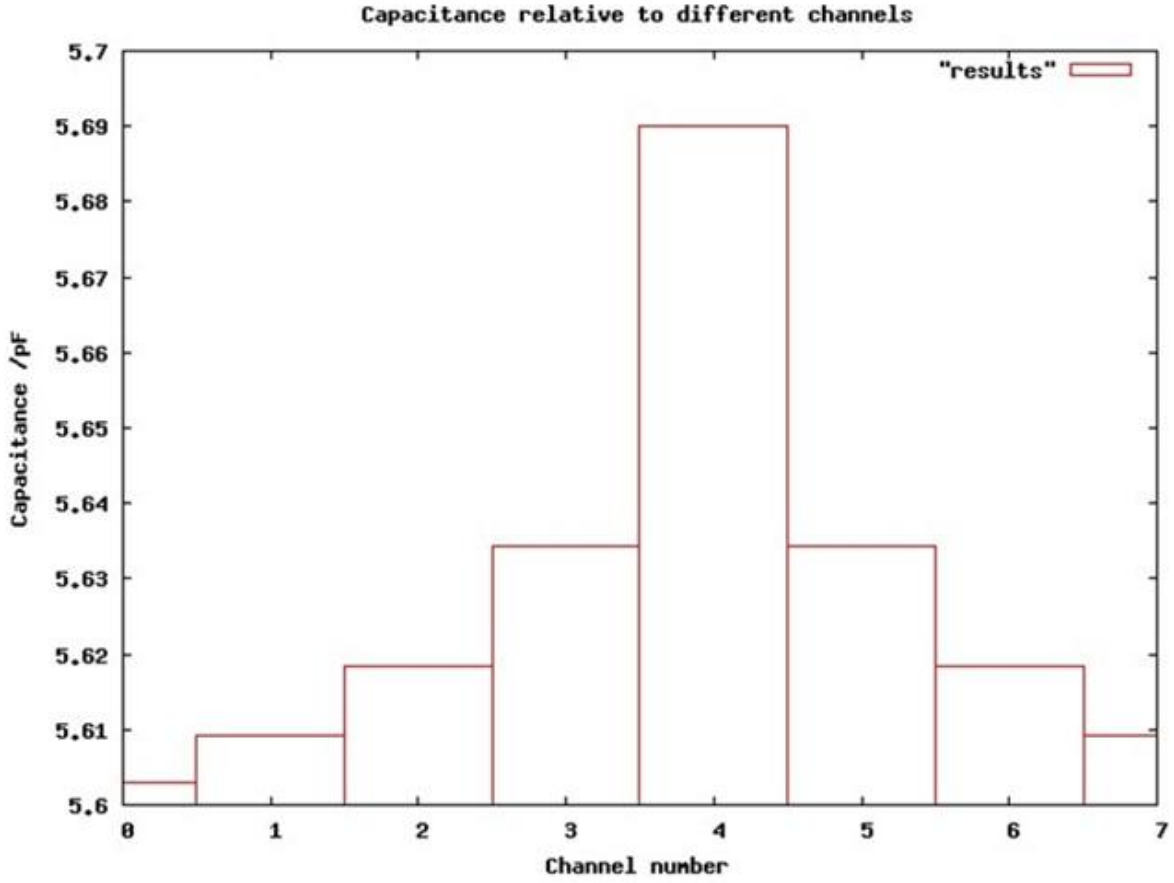


Figure A.2: Variation of capacitance with tag distance

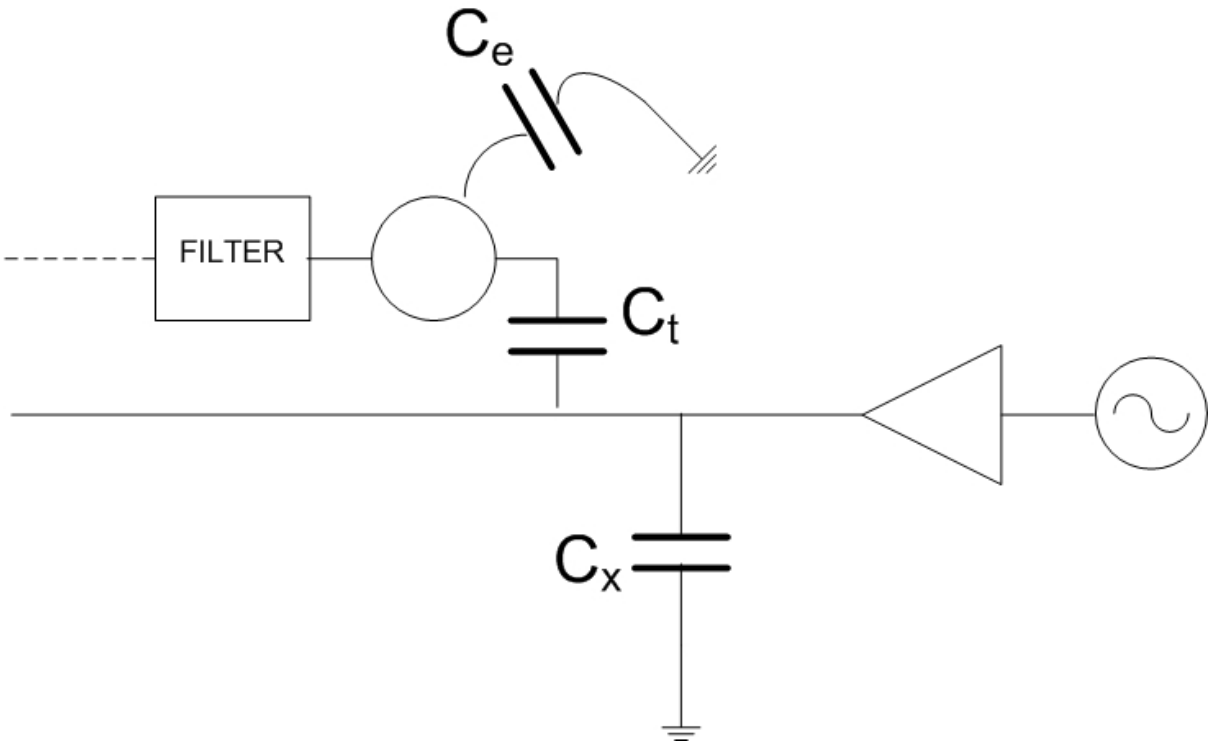


Figure A.3: Circuit formed by tag and wire embedded in the carpet

# References


[1] Compaq ipaq, http://www.compaq.com/products/ipaq. 23

[2] Experimental Technology from Sun Microsystems Laboratories. http://www.sunspotworld.com/. 120

[3] http://www.arm.com/products/cpus/. 32, 45, 47

[4] http://www.pc104.org/consortium/. 23

[5] Intel Wireless Sensor Network Research: http://www.intel.com/research/exploratory. 24

[6] Intelligent Power Management Powering Next-Generation Mobile Devices. Technical report, ARM Ltd. 32

[7] Mica/mica2 sensor node, http://www.xbow.com. 23, 24

[8] Stargate gateway node. http://www.xbow.com/products/xscale.htm. 24

[9] The ubisense smart space platform. http://www.ubisense.net. 99

[10] INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS. Technical report, 2001. Executive Summary. 30, 120

[11] 3DSP. *Application Note: Inter-processor Communication.* 76

[12] N. Abramson. The Aloha System - another alternative for computer communications. In *Proceedings of Fall Joint Computer Conference, AFIPS Conference*, 1970. 105

[13] M. Addlesee, A. H. Jones, F. Livesy, and F. Samaria. The ORL Active Floor. *IEEE Personal Communications*, **4**(5), October 1997. 100

[14] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings. 53

[15] Rajeevan Amirtharajah and Anantha P. Chandrakasan. A micropower programmable DSP using approximate signal processing based on distributed arithmetic. *IEEE Journal of Solid-State Circuits*, **39**(2):337–346, February 2004. 32

[16] Hari Ananthan. FinFET - Current Research Issues. Technical report, Purdue University. 31

[17] ARM Ltd. *ARM Architecture Reference Manual*. 37

[18] ARM Ltd. ARM11 MPCore Processor - Technical Reference Manual. 75

[19] David D. Awschalom, Michael E. Flatte, and Nitin Samarth. SPINTRONICS. *Scientific American*, June 2002. 31

[20] Jean Bacon and Tim Harris. *Operating Systems*. Addison-Wesley, 2003. 77

[21] Paramvir Bahl and Venkata Padmanabhan. RADAR: An in-building rf-based user location and tracking system. In *IEEE INFOCOM*, 2000. 100

[22] Monty Barlow. Many cores make radio work. Cambridge Consultants, IET Cambridge Network Seminar, December 2007. 90

[23] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, **15**:600–625, 1996. 70

[24] S. Basagni. Distributed clustering for ad hoc networks. In *International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN 99)*, pages 310–315, 1999. 21

[25] Jonathan Beaver, Mohamed A. Sharaf, Alexandros Labrinidis, and Panos K. Chrysanthis. Location-aware routing for data aggregation in sensor networks. In *GeoSensor Networks Workshop*, 2003. 21

[26] Jurgen Bohn and Friedemann Mattern. Super-Distributed RFID Tag Infrastructures. 101

[27] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley. 57

[28] Sergiy Butenko. A new heuristic for the minimum connected dominating set problem on ad hoc wireless networks. 20

[29] Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science).* Plenum Publishing Co., 2005. 69

[30] H. Chan and A. Perrig. ACE: An Emergent Algorithm for Highly Uniform Cluster Formation. In *European Workshop on Sensor Networks*, pages 154–171, 2004. 20

[31] M. Chatterjee, S.K. Das, and D. Turgut. WCA: A weighted clustering algorithm for mobile ad hoc networks. *Kluwer Journal of Cluster Computing, Special issue on Mobile Ad hoc Networking*, **5**:193–204, 2002. 21

[32] F. Comeau, S.C. Sivakumar, W. Robertson, and W.J. Phillips. Energy conserving architectures and algorithms for wireless sensor networks. In *39th Annual Hawaii International Conference on System Sciences (HICSS)*, **9**, pages 236c – 236c, 2006. 19

[33] Crossbow Technology, Inc. *Imote2 High-perfromance Wireless Sensor Network Node.* 34

[34] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles and Practice of Parallel Programming*, pages 1–12, 1993. 82

[35] E. Dijkstra. The structure of the the multiprogramming system. *Commun. ACM 11, 5.*, 1968. 55, 60

[36] Damian J. DIMMICH, Christian L. JACOBSEN, and Matthew C. JADUD. A Cell Transterpreter. *Communicating Process Architectures*, 2006. 76

[37] Bruno Dutertre. The Priority Ceiling Protocol: Formalization and Analysis using PVS. Technical report, System Design Laboratory, SRI International, 1999. 78

[38] Virantha Ekanayake, IV Clinton Kelly, and Rajit Manohar. An Ultra Low-Power Processor for Sensor Networks. In *Architectural Support for Programming Languages and Operating Systems*, 2004. 33

[39] Mbou Eyole-Monono, Robert Harle, and Andy Hopper. Poise: An inexpensive, low-power location sensor based on electrostatics. In *3rd Annual International Conference On Mobile and Ubiquitous Systems: Networks And Services (MobiQuitous 2006).* 100

[40] Klaus Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. 2nd edition, 2003. 122

[41] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, **25 (2)**, May 2007. 77

[42] Masaaki Fukumoto and Mitsuru Shinagawa. CarpetLan: A Novel Indoor Wireless(-like) Networking and Positioning System. In *Ubicomp*, pages 1–18, 2005. 100

[43] S.B. Furber, J.D. Garside, and D.A. Gilbert. AMULET3: a high-performance self-timed ARM microprocessor. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors, 1998. ICCD '98.*, pages 247–252, 1998. 32

[44] T. Ghani. A 90nm High Volume Manufacturing Logic Technology Featuring Novel 45nm Gate Length Strained Silicon CMOS Transistors. Technical report, 2003. 30

[45] David Graumann, Meghan Quirk, Braden Sawyer, and Justin Chong. Large surface area electronic textiles for ubiquitous computing: A system approach. In *The 4th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2007. 102

[46] Niall Griffith and Mikael Fernstrom. Litefoot a floor space for recording dance and controlling media. 101

[47] Supercomputing Technologies Group. *CILK 5.3.2 Reference Manual*. MIT Laboratory for Computer Science. 56

[48] M. J. Handy, M. Haase, and D. Timmermann. Low energy adaptive clustering hierarchy with deterministic cluster-head selection. In *IEEE International Conference on Mobile and Wireless Communication Networks*, page 368 372, Sept. 2002. 19

[49] Robert K. Harle. *Maintaining World Models in Context-Aware Environments*. PhD thesis, University of Cambridge, 2004. 102

[50] Andy Harter, Andy Hopper, Pete Steggles, Any Ward, and Paul Webster. The anatomy of a context-aware application. In *5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 1999)*, 1999. 99, 100

[51] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. An application specific protocol architecture for wireless microsensor networks. *IEEE Transaction on Wireless Networking*. 21

[52] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Hawaii International Conf. on System Sciences*, Jan. 2000, pp. 1-10. 19

[53] John L. Hennessey and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003. 36, 47, 74

[54] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *IEEE Computer*, **34**(8):57–66, 2001. 102

[55] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall. 55

[56] Andy Hopper. Computing for the future of the planet. http://www.cl.cam.ac.uk/research/dtg/ ah12/. 11

[57] IBM. POWER ISA Version 2.04, April 2007. 40

[58] Texas Instruments. *MSP430 Ultra-Low-Power Microcontrollers*. 34

[59] C. Intanagonwiwat. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MOBICOM*, August 2000. 21

[60] Intel. *Intel Architecture Software Developer's Manual: Instruction Set Reference*. 40

[61] Intel. *Intel Itanium Architecture Software Developer's Manual*, 2006. 37

[62] Youssef Kaddourah, Jeff King, and Abdelsalam (Sumi) Helal. Cost-Precision Tradeoffs in Unencumbered Floor-Based Indoor Location Tracking. In *Proceedings of the third International Conference On Smart homes and health Telematic (ICOST)*, 2005. 101

[63] Michael Kanellos. Intel helps build low-power transistor. *CNET News.com*, February 2005. 31

[64] Michael Kanellos. Intel sketches out nanotechnology roadmap. *CNET News.com*, February 2005. 31

[65] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual For System-On-Chip Design*. 28, 30, 51

[66] J. Kim and K. Roy. Double Gate MOSFET Subthreshold Circuit for Ultra-low Power Applications. *IEEE Trans. Electronic Devices*, **51**(9):1468–1474, 2004. 31

[67] Ralph M. Kling. Intel mote: An Enhanced Sensor Network Node. 26

[68] E S Kuh. *MULTICHIP MODULES*. 32

[69] J. Kulik, W. R. Heinzelman, and H. Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, **8**:16985, 2002. 22

[70] Jean J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books. 64

[71] Nicholas Lane and Andrew Campbell. The influence of Microprocessor Instructions on the energy consumption of wireless sensor networks. In *Third Workshop on Embedded Networked Sensors (EmNets 2006)*, 2006. 34

[72] Yee Wei Law, Sandro Etalle, and Pieter H. Hartel. Assessing security in energy-efficient sensor networks. 24

[73] Sam Lindley. Implementing declarative deterministic concurrency using sieves. *Declarative Aspects of Multicore Programming*, 2007. 49

[74] Stephanie Lindsey and Cauligi S. Raghavendra. Pegasis: Power-efficient gathering in sensor information systems. In *IEEE Aerospace Conference*, March 2002. 20

[75] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, **20**(1):46–61, 1973. 68

[76] Ciaran Lynch and Fergus O'Reilly. Processor Choice For Wireless Sensor Networks. In *Workshop on Real-World Wireless Sensor Networks*, 2005. 34

[77] Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice Hall PTR. 82

[78] A. Manjeshwar and D. P. Agarwal. Teen: a routing protocol for enhanced efficiency in wireless sensor networks. 1st International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing, April 2001. 22

[79] V. Mhatre and C. Rosenberg. Design guidelines for wireless sensor networks: communication, clustering and aggregation. *Ad Hoc Networks*, **2**(1):45–63, January 2004. 20

[80] Vivek Mhatre and Catherine Rosenberg. *Performance Evaluation and Planning Methods for the Next Generation Internet*, chapter Energy and Cost Optimizations in Wireless Sensor Networks: A Survey, pages 227–248. Springer US, 2005. 19

[81] Vivek Mhatre and Catherine Rosenberg. Homogeneous vs. heterogeneous clustered sensor networks: A comparative study. In *IEEE International Conference on Communications, volume 6, pages 3646-3651*, June 2004. 19

[82] MIPS Technologies. Mips32 architecture for programmers volume ii: The mips32 instruction set. 43

[83] Andre Mota, Leonardo B. Oliveira, Felipe F. Rocha, Ramon Riserio, Antonio A. F. Loureiro, Claudionor J.N. Coelho Jr., Hao Chi Wong, and Eduardo Nakamura. WISENEP: A Network Processor for Wireless Sensor Networks. *ISCC*, **0**:8–14, 2006. 34

[84] Alan Mycroft, Paul Webster, and Phil Endecott. Improved power efficiency in microprocessors. International Patent Publication Number WO 02/095574 A1, November 2002. 33

[85] Leyla Nazhandali. *Architectural Optimisation for Performance- and Energy-Constrained Sensor Processors*. PhD thesis, University of Michigan, 2006. 33

[86] Leyla Nazhandali, Michael Minuth, Bo Zhai, Javin Olson, Todd Austin, and David Blaauw. A Second-Generation Sensor Network Processor with Application-Driven Memory Optimizations and Out-of-Order Execution. In *ACM/IEEE International Conference on Compilers, Architecture, and Sythesis for Embedded Systems*, September 2005. 33

[87] Lionel M. Ni, Yunhao Liu, Yiu C. Lau, and Abhishek P. Patil. Landmarc: Indoor location sensing using active rfid: Pervasive computing and communications (guest editors: Mohan kumar, diane cook and anand tripathi). *Wireless Networks*, **10**(6):701+. 101

[88] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, May 2005. 56

[89] R.J. Orr and G.A Abowd. The Smart Floor: A Mechanism for Natural User Identification and Tracking. In *Proceedings of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, 2000. 100

[90] Peter S. Pacheco. *A User's Guide To MPI*. University of San Francisco, March 1998. 56

[91] Heidi Pan and Krste Asanovic. Heads and tails: a variable-length instruction format supporting parallel fetch and decode, 2001. 43

[92] J. Paradiso, C. Abler, KY. Hsiao, and M. Reynolds. The Magic Carpet: Physical Sensing for Immersive Environments. In *Proc. of the CHI '97 Conference on Human Factors in Computing Systems, Extended Abstracts*, 1997. 100

[93] Vamsi Paruchuri. *ADAPTIVE SCALABLE PROTOCOLS FOR HETEROGENEOUS WIRELESS NETWORKS*. PhD thesis, Louisiana State University and Agricultural and Mechanical College. 24

[94] Dickon Reed and Robin Fairbairns. Nemesis Kernel Overview, May 1997. 62, 69

[95] Jun Rekimoto. Smartskin: An infrastructure for freehand manipulation on interactive surfaces. In *CHI2002*. 100

[96] Sokwoo Rhee, Deva Seetharam, Sheng Liu, Ningya Wang, and Jason Xiao. i-Bean Network: An Ultra-Low Power Wireless Sensor Network. In *Ubicomp*, 2003. 34

[97] Bruce Richardson, Krispin Leydon, Mikael Fernstrom, and Joseph A. Paradiso. Z-tiles: Building blocks for modular, pressure-sensing floorspaces. In *CHI 2004*. 100

[98] Shyam Sadasivan. An Introduction to the ARM Cortex-M3 Processor. Technical report, ARM Ltd., 2006. 45

[99] C. Schurgers and M.B. Srivastava. Energy efficient routing in wireless sensor networks. In *Comm. for Network-Centric Ops.: Creating the Info.*, 2001. 21

[100] Selvadurai Selvakennedy and Sukunesan Sinnappan. An energy-efficient clustering algorithm for multihop data gathering in wireless sensor networks. *JOURNAL OF COMPUTERS*, **1**(1), 2006. 21

[101] SGS-THOMSON Microelectronics Ltd. Occam 2.1 reference manual. 55

[102] Ghalib A. Shah, Muslim Bozyigit, and Ozgur B. Akan. Multi-event adaptive clustering (meac) protocol for heterogeneous wireless sensor networks. 24

[103] Kwangcheol Shin, Ajith Abraham, and Sang Yong Han. Self organizing sensor networks using intelligent clustering. In *ICCSA*, 2006. 20

[104] Mukesh Kumar Singla and Trilok Chand Aseri. Impacts of correlation and observer location on clustered wireless sensor networks. In *IFIP International Conference on Wireless and Optical Communications Networks WOCN '07*, 2007. 23

[105] Georgios Smaragdakis, Ibrahim Matta, and Azer Bestavros. SEP: A stable election protocol for clustered heterogeneous wireless sensor networks. In *Second International Workshop on Sensor and Actor Network Protocols and Applications*. 24

[106] Ignacio Solis and Katia Obraczka. Energy-efficient mapping in sensor networks. 22

[107] S. Srivathsan and S.S. Iyengar. Minimizing latency in wireless sensor networks - a survey. In *Proceedings of the Third IASTED International Conference Advances in Computer Science and Technology*, April 2007. 92

[108] David Stewart and Michael Barr. Rate monotonic scheduling. Embedded Systems Programming, March 2002. pp. 79-80. 68

[109] SYNOPSYS. Power Compiler: Automatic Power Management within Galaxy Design Platform. Technical report, 2004. 31, 45

[110] Huseyin Ozgur Tan and Ibrahim Korpeoglu. Power efficient data gathering and aggregation in wireless sensor networks. 20

[111] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 1997. 68

[112] Texas Instruments. *OMAP5910 Dual-Core Processor Inter-Processor Communication Reference Guide*, 2005. 75

[113] Chen-Khong Tham. *Sensor-Grid Computing and SensorGrid architecture for Event Detection, Classification and Decision-Making*, chapter Sensor Network and Configuration: Fundamentals, Techniques, Platforms, and Experiments. Springer-Verlag, 2006. 24

[114] The Embedded Microprocessor Benchmark Consortium. http://www.eembc.org. 47, 86

[115] Vlasios Tsiatsis, Ram Kumar, and Mani B. Srivastava. Computation hierarchy for in-network processing. *Mobile Networks and Applications*, **10**:505–518, 2005. 23

[116] A.J. van der Wal and G.J.W. van Dijk. Efficient interprocessor communication in a tightly-coupled homogeneous multiprocessor system. In *Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 362–368, 1990. 76

[117] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. pages 96–107. International Symposium on Advanced Research in Asycnhronous Circuits and Systems, 1998. 32

[118] N.A. Vasanthi and S. Annadurai. Energy efficient sleep schedule for achieving minimum latency in query-based sensor networks. In *SUTC*, 2006. 92

[119] N. Vlajic and D. Xia. Wireless sensor networks: To cluster or not to cluster? In *2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks(WoWMoM'06)*, pages 258–268, 2006. 22

[120] Hanbiao Wang, Deborah Estrin, and Lewis Girod. Preprocessing in a tiered sensor network for habitat monitoring. *EURASIP JASP special issue of Sensor Networks*, **4**:392401, 2003. 23

[121] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems*, **10**(1):91102, 1992. 99

[122] B.A Warneke and K.S.J. Pister. An Ultra-Low Energy Microcontroller for Smart Dust Wireless Sensor Networks. In *International Solid-State Circuits Conference*, 2004. 34

[123] Douglas Watt and David May. A Language and Processor for Unifying System-on-Chip Design. Technical Report CSTR-06-010, University of Bristol, April 2006. 91

[124] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, pages 94–10, 1991. 10

[125] D. Xia and N. Vlajic. Near-optimal node clustering in wireless sensor networks for environment monitoring. In *21st International Conference on Advanced Networking and Applications (AINA '07)*, pages 632–641, 2007. 23

[126] Thomas G. Zimmerman, Joshua R. Smith, Joseph A. Paradiso, David Allport, and Neil Gershenfeld. Applying electric field sensing to human-computer interfaces. In *CHI*, pages 280–287, 1995. 100