Workload-aware Query Routing Trees in Wireless Sensor Networks

Panayiotis Andreou, Demetrios Zeinalipour-Yazti*, Panos K. Chrysanthis‡, George Samaras

Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus
*Pure and Applied Sciences, Open University of Cyprus, P.O. Box 24801, 1304 Nicosia, Cyprus
‡ Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
p.andreou@cs.ucy.ac.cy, zeinalipour@ouc.ac.cy, panos@cs.pitt.edu, cssamara@cs.ucy.ac.cy

Abstract

Continuous queries in wireless sensor networks are established on the premise of a routing tree that provides each sensor with a path over which answers can be transmitted to the query processor. We found that these structures are sub-optimality constructed in predominant data acquisition systems leading to an enormous waste of energy. In this paper we present MicroPulse[1]*, a workload-aware optimization algorithm for query routing trees in wireless sensor networks. Our algorithm is established on profiling recent data acquisition activity and on identifying the bottlenecks using an in-network execution of the critical path method. A node S utilizes this information in order to locally derive the time instance during which it should wake up, the interval during which it should deliver its workload and the workload increase tolerance of its parent node. We additionally provide an elaborate description of energy-conscious algorithms for disseminating and maintaining the critical path cost in a distributed manner. Our trace-driven experimentation with real sensor traces from Intel Research Berkeley shows that MicroPulse can reduce the data acquisition costs by many orders.*

1 Introduction

Recent advances in embedded computing have made it feasible to produce small scale sensors, actuators and processors that can be used in ad-hoc deployments of environmental monitoring infrastructures [16, 8, 12]. The longevity of a Wireless Sensor Network (WSN) heavily relies on the existence of power-efficient algorithms for the acquisition, aggregation and storage of the sensor readings.

Communicating over the radio in a WSN is the most energy demanding factor among all other functions, such as storage [21] and processing [12]. The energy consumption for transmitting 1 bit of data using the MICA mote [1] is approximately equivalent to processing 1000 CPU instructions [12]. One way to cope with the energy challenge is to power down the radio transceiver during periods of inactivity. In particular, it has been shown that sensors operating at a 2% duty cycle can achieve lifetimes of 6-months using two AA batteries [13].

The continuous interval during which a sensor node S enables its transceiver, collects and aggregates the results from its children, and then forwards them all together to its own parent is defined as the *waking window* τ. Note that τ is continuous because it would be very energy-demanding to suspend the transceiver more than once during the interval of an *epoch*, which specifies the amount of time that sensors have to wait before re-computing a continuous query.

It is important to mention that the exact value of τ is query-specific and can not be determined accurately using current techniques. For instance, a sensor cannot easily estimate how many tuples will be transmitted from its children. Choosing the correct value for τ is a challenging task as any wrong estimate might disrupt the synchrony of the query routing tree. The objective of this work is to automatically tune τ, locally at each sensor without any a priori knowledge or user intervention. Note that in defining τ we are challenged with the following trade-off:

- **Early-Off Transceiver:** Shall S power-off its transceiver too early reduces energy consumption but also increases the number of tuples that are not delivered to the *sink*, the root of the routing tree. As a result the sink will generate an erroneous answer to the query Q; and
- **Late-Off Transceiver:** Shall S keep the transceiver active for too long decreases the number of tuples that are lost due to powering down the transceiver too early but also increases energy consumption. Thus, the network will consume more energy than necessary which is not desirable given the scarce energy budget of each sensor.

[1] A preliminary version of this paper appeared in [20]

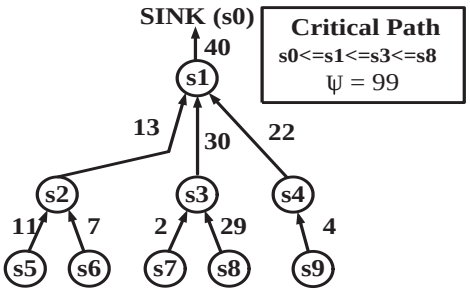

Figure 1. Nine sensing devices and their respective workload (shown as edges) during the execution of a continuous query Q. MicroPulse utilizes this information in order to locally adapt the waking window of each sensor using the Critical Path Method.

In this paper we present *MicroPulse*, a novel algorithm for adapting the waking window of a sensing device S based on the data workload incurred by a query Q. Our ideas are established on profiling recent data acquisition activity and on identifying the bottlenecks using an in-network execution of the Critical Path Method.

The *Critical Path Method (CPM)* [6] is a graph-theoretic algorithm for scheduling project activities. It is widely used in project planning (construction, product development, plant maintenance, software development and research projects). The core idea of CPM is to associate each project milestone with a vertex v and then define the dependencies between the given vertices using *activities*. For instance, the activity $v_i \Leftarrow v_j$ denotes that the completion of v_i depends on the completion of v_j. Each activity is associated with a weight (denoted as $\overset{weight}{\Leftarrow}$) which quantifies the amount of time that is required to complete v_i assuming that v_j is completed. The critical path allows us to define the minimum time, or otherwise the maximum path, that is required to complete a project (i.e., milestone v_0). Any delay in the activities of the critical path will cause a delay for the whole project.

In order to adapt the discussion to a sensor network context assume that each sensor s_i is represented by a CPM vertex. More formally, we map each s_i to the elements of the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ using a 1:1 mapping function $f : s_i \rightarrow v_i$, $i \leq n$. Also, let the descendent-ancestor relations of the sensor network be denoted as edges in this graph.

Figure 1 illustrates an example which will be utilized throughout the paper and Table 1 summarizes our main symbols. The weights on the edges of the figure define the workload of each respective node (as the required time to propagate the query results between the respective pairs). It is easy to see that the total time to answer the query at the sink in the given network is at least ψ=99, since the critical path is $s_0 \overset{40}{\Leftarrow} s_1 \overset{30}{\Leftarrow} s_3 \overset{29}{\Leftarrow} s_8$.

Having this information at hand enables the scheduling of transmission between sensors. In particular, sensor s_i can be scheduled to wake-up and transmit at the following deadlines (w_i): $w_1 = \psi - 40 = 59$, $w_2 = w_1 - 13 = 46$, $w_3 = w_1 - 30 = 29$, $w_4 = w_1 - 22 = 37$ while s_0 and s_1 will be listening for these transmissions during the intervals τ_0=[59..99) and τ_1=[29..59) respectively. The same intuition also applies for the leaf nodes, e.g., s_5 starts transmission at $w_5 = w_2 - 11 = 35$ and s_2 listens for this transmission in the range τ_2=[35..46). Additionally, the critical path enables a sensor s_j ($j \leq n$) to estimate the interval during which its parent s_i ($i \leq n$) will have its transceiver enabled. This is very useful because in the subsequent epochs and under a different workload, s_j can find out if it can deliver the new workload without first asking s_i to adjust its waking window.

It should be noted that the edges in Figure 1 have different weights. This is very typical for a sensor network as the link quality can vary across the network [16]. Another reason is that some sensors might have a different workload than other sensors. Note that our scheduling scheme is distributed which makes it fundamentally different from centralized scheduling approaches like DTA [19] and TDDES [4] that generate collision-free query plans at a centralized node. Additionally, our approach is also different from techniques such as [15] which segment the sensor network into sectors in order to minimize collisions during data acquisition.

Our Contributions

In this paper we make the following contributions:

- We devise techniques that intelligently exploit the critical path method in order to prolong the longevity of the network and hence the quality of results. In particular, we optimize the length of the waking window τ using an energy efficient distributed algorithm;
- We propose a distributed maintenance algorithm of the critical path cost which minimizes communication between sensors;
- We provide an extensive experimental evaluation using traces from a real sensor network deployment at Intel Research Berkeley [2].

The remainder of the paper is organized as follows: Section 2 studies the waking window mechanism of popular data acquisition systems. Section 3 presents the underlying algorithms of the MicroPulse Framework. Section 4 presents our experimental study using a trace-driven simulator and Section 5 concludes the paper.

Table 1. Definition of Symbols

Symbol	Definition
Q	A Query
s_i	Sensor number i (s_0 denotes the sink).
n	Number of Sensors $\{s_1, s_2, ..., s_n\}$
e, d	Epoch duration and Routing tree Depth
ψ	The Critical Path Cost of the network
w_i	Wake-up time instance of s_i
ψ_i	Critical Path cost of s_i
τ_i	Waking window of s_i
λ_i	Workload Increase Tolerance of the parent of s_i

2 Background and Related Work

In this section we study the query routing tree of the two most popular declarative acquisition frameworks: TAG [12, 13] and Cougar [18]. We start out our description by assuming that the query Q has been disseminated to the sensors.

Tiny Aggregation (TAG): In this approach, the epoch e is divided into a number of fixed-length time *intervals* $\{e_1, e_2, \ldots, e_d\}$, where d is the depth of the routing tree, rooted at the sink, that conceptually interconnects the n sensors. The core idea of this framework is summarized as follows: *"when nodes at level i+1 transmit then nodes at level i listen"*. More formally, a sensor s_i enables its transceiver at time instance $w_i = \lfloor e/d \rfloor * (d - depth(s_i))$ and keeps the transceiver active for $\tau_i = \lfloor e/d \rfloor$ time instances. Note that $\sum_{i=d}^{0}(e_i)$ provides a lower-bound on e, thus the answer will always arrive at the sink before the end of the epoch. Setting e as a prime number ensures the following inequality $\sum_{i=d}^{0}(e_i) < e$, which is desirable given that the answer has to reach the sink at time instance e.

For instance, if the epoch is 31 seconds and we have a three-tiered network (i.e., $d = 3$) like Figure 2 (top, left), then the epoch is sliced into three segments $\{10,10,10\}$. During interval [0..10), nodes at level 3 will transmit while nodes at level 2 will listen; during interval [10..20) level 2 nodes transmit while level 1 nodes listen; and finally during [20..30), level 1 nodes transmit and the sink (level 0) listens. Thus, the answer will be ready prior the completion of time instance 31 which is the end of the epoch.

The main drawback of the TAG query routing tree is that the waking window τ is an over-estimation of the expected workload that incurs on the edges of the tree (in the above example 10 seconds!). The rationale behind this over-estimation is to offset the limitations in the quality of the clock synchronization algorithms [12], but in reality it is too coarse. In the experimental Section 4, we found that this over-estimation is two orders of magnitude larger than necessary. Additionally, it is not clear how τ is set under a *variable workload* which occurs under the following circumstances: i) from a *non-balanced topology*, where some nodes have many children and thus require more time to collect the results from their dependents; and ii) from *multi-tuple answers*, which are generated because some nodes return more tuples than other nodes (e.g. due to the query predicate).

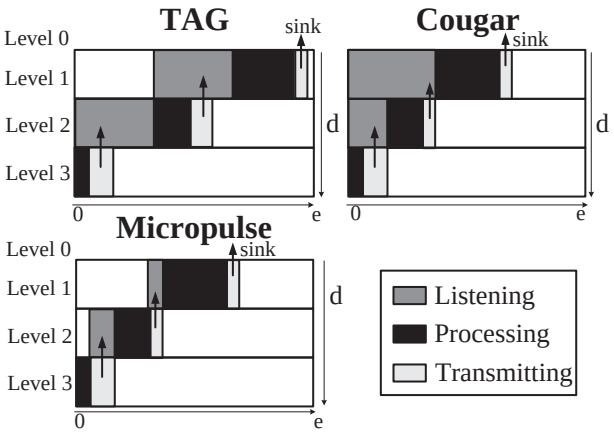

Figure 2. The Waking (Listening) Window (τ) in TAG, Cougar and MicroPulse.

The MicroPulse algorithm presented in this paper gracefully handles both cases of variable workload by utilizing the Critical Path Method. Our algorithm, like TAG, utilizes the TinyOS [7] MAC layer [17] to handle the collisions that will occur if nodes in the same vicinity transmit during the same interval.

Cougar: In this approach, each sensor maintains a *waiting list* that specifies its children. Such a list can be constructed by having each child explicitly acknowledging its parent during the query dissemination phase. Having the list of children enables a sensor to shut down its transceiver as soon as all its children have answered. This yields a set of non-uniform waking windows $\{\tau_1, \tau_2, \ldots\}$[2] as opposed to TAG where we have a single τ which is uniform for all sensors (i.e., $\lfloor e/d \rfloor$).

The main drawback of Cougar is that a parent node has to recursively keep its transceiver active from the beginning of the epoch until all children have answered. In order to cope with children sensor that may not respond, Cougar deploys a timeout h. To understand the drawback of Cougar consider Figure 2 (top, right), where level 2 and level 1 nodes are expected to enable their transceivers at time instance zero and wait until all their children have responded. Given a failure at some arbitrary node s_i ($1 \leq i \leq n$), will require that each node on the path $s_i \rightarrow \ldots \rightarrow s_0$, will keep its transceiver active for h additional seconds.

[2] In particular, if $depth(v_i) < depth(v_j)$ then $\tau_i > \tau_j$ $(\forall i, j \leq n)$.

3 The MicroPulse Algorithm

In this section we describe the underlying algorithms of MicroPulse. We divide our description in the following three conceptual phases:

1. *Construction Phase*, executed once prior the execution of Q, during which the sink constructs the routing tree and becomes aware of the critical path cost ψ.
2. *Pulse Phase*, during which each sensor s_i tunes its wake-up time instance w_i and waking window τ_i according to the value ψ.
3. *Adaptation Phase*, executed when a topology or workload change occurs.

3.1 The Critical Path Construction Phase

This phase starts out by having each node select one node as its parent. This results in a *waiting list* similarly to Cougar [18]. To accomplish this task, the parent is notified through an explicit acknowledgement or becomes aware of the child's decision by snooping the radio. Note that in both TAG [12] and Cougar [18] nodes select as their parent whichever sensor forwarded the query first. Alternatively, nodes could have chosen as their parent the neighbor with the smallest hop count from the sink or the one with the highest signal strength. In more recent frameworks, like GANC [14] and Multi-Criteria Routing [11], sensors select their parents based on query semantics, power consumption, remaining energy and others. In more unstable topologies a node can maintain several parents [5] in order to achieve fault tolerance but this might impose some limitations on the type of supported queries. Nevertheless, all these alternatives are supplementary to this step.

In the next step, we profile the activity of the incoming and outgoing links and then propagate this information towards the sink. In particular, we execute one round of data acquisition where each sensor s_i maintains one counter for its parent connection (denoted as s_i^{out}) and one counter per child connection (denoted as $s_{i,j}^{in}$), where j denotes the identifier of the child. These counters measure the *workload* between the respective sensors and will be utilized to identify the critical path cost in the subsequent epochs. Note that these counters account for more time than what is required had we assumed a collision-free MAC channel. Additionally, it is important to mention that we could have deployed a more complex structure rather than the counters s_i^{out} and $s_{i,j}^{in}$, that would allow a sensor to obtain a better statistical indicator of the link activity. By projecting the time costs obtained for each edge to a virtual spanning tree creates a distributed *Query Routing Tree*, similar to the one depicted in Figure 1.

The final step is to percolate these local edge costs to the sink by recursively executing the following in-network function *f* at each sensor s_i:

$$f(s_i) = \begin{cases} 0 & \text{if } s_i \text{ is a leaf,} \\ max_{\forall j \in children(s_i)}(f(s_j) + s_{i,j}^{in}) & \text{otherwise.} \end{cases}$$

The critical path cost is then $f(s_0)$ (denoted for brevity as ψ). Using our working example of Figure 1, we will end up with the following values : $f(s_{5 \leq i \leq 9}) = 0$, $f(s_4) = 4$, $f(s_3) = 29$, $f(s_2) = 11$, $f(s_1) = 59$ and $\psi = f(s_0) = 99$.

3.2 The Pulse Phase

In this phase each sensor s_i ($i \leq n$) locally defines three parameters using the critical path cost ψ. These parameters enable s_i to derive: i) the time instance during which it should wake up (i.e., w_i), ii) the interval during which it should transmit (i.e., τ_i), and iii) the workload increase tolerance of the parent of s_i (i.e., λ_i) which signifies when the synchrony of the query routing tree might be disrupted.

Algorithm 1 presents the main steps of this procedure which propagates ψ top-down, from the sink to the leaf sensors, with a message complexity of $O(n)$. The first step aborts the impossible case where the critical path is larger than the epoch. The second step calculates the wake up time instance w_i, such that s_i has enough time to collect the tuples from all its children s_j ($\forall j \in children(s_i)$). In practice, this is defined by the child of s_i with the largest workload (i.e., $s_{i,maxchild}^{in}$). The second step also defines the waking window of τ_i, which is the complete window during which s_i will enable its transceiver. In the third step, the children of s_i are notified with the adjusted critical path cost (i.e., $\psi - s_j^{out}$). Concurrently with step three, s_i also notifies its children s_j with the workload increase tolerance of s_i (i.e., λ_i) and a flag which signifies whether these nodes belong to the critical path. Thus, s_j can intelligently schedule its transmissions in cases of local workload deviations.

To facilitate our presentation we will now simulate the execution of Algorithm 1 on the example of Figure 1. To simplify the discussion, assume that the costs a, b and c (which account for *processing, the inaccurate clock* and the *collisions at the MAC layer*) are all equal to zero. Additionally, assume that the critical path cost is small enough to fit within the epoch (i.e., $\psi << e$). In particular, with $\psi = 99$ we get the following quadruples $(s_i, w_i, \tau_i, \lambda_i)$ at each sensor: $\{ (s_0, 59, [59..99), 0), (s_1, 29, [29..59), 0), (s_2, 46, [46..59), 17), (s_3, 29, [29..59), 0), (s_4, 37, [37..59), 8), (s_5, 35, [35..46), 0), (s_6, 39, [39..46), 4), (s_7, 27, [27..29), 27), (s_8, 0, [0..29), 0), (s_9, 33, [33..37), 0) \}$

To understand the benefits of λ_i, consider the scenario where node s_7 increases its workload by 15 time instances. Since $\lambda_7 = 29 - 2 = 27$, s_7 knows that the transceiver of

Algorithm 1 : **MicroPulse Pulse Phase**

Input: n sensing devices $\{s_1, s_2, \ldots, s_n\}$ and the sink s_0, the Critical Path cost ψ, the epoch e.
Output: A set of n waking windows τ_i ($\forall i \leq n$), wake-up time instances w_i ($\forall i \leq n$) and workload increase tolerance thresholds λ_i ($\forall i \leq n$)
Execute these steps beginning from s_0 (top-down):

1. If $\psi > e$ then abort *"The Critical Path is larger than the Epoch"*.
2. Find the maximum $s^{in}_{i,j}$ in s_i's children and denote the identifier of this sensor as $maxchild$. Now calculate the wake time w_i as follows:
$$w_i = \psi - s^{in}_{i,maxchild} - a - b - c, \quad (1)$$
where a, b and c are three variables which offset the costs of *processing, the inaccurate clock* and *collisions at the MAC layer*. The waking window is the interval:
$$\tau_i = [w_i..(w_i + s^{in}_{i,maxchild})) \quad (2)$$
3. Now disseminate ψ_i to s_i's children s_j ($\forall j \in children(s_i)$). Upon receiving ψ_i, each s_j decreases ψ_i locally, as follows:
$$\psi_j = \psi_i - s^{out}_j \quad (3)$$
4. At the same time with step 3, disseminate $s^{in}_{i,maxchild}$ to s_i's children s_j ($\forall j \in children(s_i)$). s_j will utilize this information in order to define the *workload increase tolerance* (λ_i) of s_i, as follows:
$$\lambda_j = s^{in}_{i,maxchild} - s^{out}_j \quad (4)$$
5. Repeat steps 2-5, recursively until all sensors in the network have set w_i, τ_i and λ_i respectively ($i \leq n$).

Algorithm 2 : **MicroPulse Adaptation Phase**

Input: A sensor s_i, the critical path value ψ_i, the wake-up time w_i, the waking window τ_i, a flag which indicates if s_i lies on the critical path, an error threshold δ.
Output: An updated set of w_i, τ_i and λ_i values.

```
1:  procedure Adapt(s_i)
2:      ▷ Step 1: Calculate Workload Indicators
3:      workload'_i = ψ_i − w_i;        ▷ Workload of previous epoch
4:      for j = 1 to children(s_i) do
5:          add(tuples(s_j), workload_i); ▷ Build new workload
6:      end for
7:      add(tuples(s_i), workload_i);        ▷ Append local tuples
8:      x = |workload_i − workload'_i|       ▷ Workload Deviation
9:      if (x < δ) then
10:         signal(finished);        ▷ Negligible Workload Change
11:     end if
12:     ▷ Step 2: Important Workload Change on the CP
13:     if (cp_i) then
14:         send("Request Critical Path Reconstruction", s_j);
15:         signal(finished);
16:     end if
17:     ▷ Step 3: Important Workload Change NOT on the CP
18:     if (workload_i decreased by x) then
19:         w_i = w_i + x; ▷ Adjust local wakeup time
20:     else▷ Workload was Increased by x
21:         if (x ≤ λ_i) then    ▷ x is less than the available slack
22:             w_i = w_i − x; ▷ Adjust local wakeup time
23:         else
24:             send("Request Critical Path Reconstruction", s_j);
25:         end if
26:     end if
27:     signal(finished);
28: end procedure
```

its parent s_3 is enabled for 27 additional time instances, thus s_7 can start delivering the workload earlier (i.e., $w_7 = 12$ instead of $w_7 = 27$) succeeding in completing the transmission on-time.

3.3 Adaptation Phase

In this section we describe an efficient distributed algorithm for adapting the MicroPulse query routing tree in cases of workload changes.

First notice that the naive approach to cope with workload changes is to reconstruct the MicroPulse tree in every epoch. The message cost of such an approach is analyzed as follows: the MicroPulse construction phase has a message complexity of $O(1)$ as it can be executed in parallel with the acquisition of data tuples from sensors (i.e., the critical path cost can be piggybacked with data tuples). The Pulse phase on the other hand, has a message complexity of $O(n)$ as it requires the dissemination of the critical path cost to all n nodes in the network. The algorithm we propose in this section can circumvent the $O(n)$ cost incurred by the pulse phase in every epoch by deploying a set of rules we describe next.

Algorithm 2, presents the MicroPulse Adaptation algorithm which proceeds in three steps. The first step of the algorithm (lines 2-11) calculates the workload indicators of the current epoch (i.e., $workload_i$) and the previous epoch (i.e., $workload'_i$). If the workload has changed by more than a user defined user threshold δ in line 9, we consider this change as significant and proceed with the adaptation of the routing tree in line 12. Otherwise, we disregard this deviation and abort the algorithm. Assuming a significant deviation, step 2 handles the case where the change occurs on the critical path. In such a case, s_i has to request the reconstruction of the routing tree using the construction and pulse phases. For instance, if the workload of s_3 changes from 30 time instances to 35 time instances (see Figure 1) then this will trigger the reconstruction of the MicroPulse tree and this change should be propagated to all nodes in the network. Although this case is possible, our experimen-

tal study in section 4.1 has shown that it is not frequent.

Finally, step 3 of Algorithm 2 (lines 17-26) handles the more common case where the change does not occur on the critical path. In such a case, if the workload is *decreased* by x (line 18) then a sensor locally delays its wake up variable by x (i.e., to $w_i + x$). For instance, if the workload of s_2 drops from 13 to 11 (thus, $x = 2$), then $w_2^{new} = w_2 + x = 46 + 2 = 48$. Similarly if the workload is *increased* by x (line 20) then there are two cases: i) the increase is less or equal to the slack λ_i and ii) the increase is greater than the slack λ_i. For the first case (i) consider a workload increase at s_2 from 13 to 18 (thus, $x = 5$ that is smaller than $\lambda_2 = 17$). This yields the following adaptation of the wake up time $w_2^{new} = w_i - x = 46 - 5 = 41$. For the second case (ii) consider a workload increase at s_2 from 13 to 32 (thus, $x = 19$ that is larger than $\lambda_2 = 17$). This yields the reconstruction of the tree as such an increase might potentially create a new critical path.

4 Experimental Evaluation

In this section we describe the trace-driven experimental methodology we adopt and the results of our evaluation.

4.1 Experimental Methodology

Datasets: We utilize a real trace of sensor readings that is collected from 58 sensors deployed at the premises of the Intel Research in Berkeley [2] between February 28th and April 5th, 2004. The sensors utilized in the deployment were equipped with weather boards and collected time-stamped topology information along with humidity, temperature, light and voltage values once every 31 seconds (i.e., the epoch). The dataset includes 2.3 million readings collected from these sensors. Using this dataset we derive the following two datasets:

i. **Intel54**: This is a subset of readings from the 54 sensors that had the largest amount of local readings. The rest four sensors were excluded from our experiments because they had many missing values.

ii. **Intel540**: This is a set of 540 sensors that is randomly derived from the of Intel54 dataset. In particular, we randomly replicate nodes from the Intel54 dataset until we obtain a set of 540 nodes.

Query Sets: We utilize three representative queries from two predominant classes of queries in wireless sensor networks.

The first class of such queries are *aggregate selection queries* [18, 12] (i.e., `SELECT agg() FROM sensors`). Roughly, these queries can be distinguished in: i) *distributive aggregates*, where records can be aggregated in-network without compromising correctness (e.g., `Max, Min, Sum, Count`) and ii) *holistic aggregates*, where in-network aggregation might compromise the result correctness (e.g., `Median`), thus all tuples have to be transmitted to the sink before the query can be executed. The separation between the above cases is important as each individual case defines a different workload per edge (i.e., distributive aggregates have a *fixed workload* of one tuple per edge while holistic aggregates a *variable workload*).

The second class of representative queries are *non-aggregate selection queries* (e.g., `SELECT moteid FROM sensors`). Assuming a static topology such queries generate a *fixed workload* per edge, unless we apply a predicate on the query (e.g., `temperature > X`) and generate a *variable workload* per edge in this manner.

In our experiments we utilize the following query-sets which encapsulate all the above cases:

- *Single-Tuple queries (***ST***)*: where a sensor transmits exactly one tuple per epoch. Distributive aggregates belong to this category. We utilize the following non-aggregate selection query in our study:

```
SELECT moteid, temp
FROM sensors
WHERE temp = (SELECT MAX(temp) FROM
sensors)
```

- *Multi-Tuple queries with Fixed size (***MTF***)*: where a sensor transmits a set on f tuples per epoch with f being a constant. Holistic aggregates and non-aggregate selection queries with a fixed workload belong to this category. We utilize the following representative query in our study:

```
SELECT moteid, temp
FROM sensors
```

- *Multi-Tuple results with Arbitrary size (***MTA***)*: where a sensor transmits a set of f' tuples per epoch with f' being a variable that might change across different epochs. Non-aggregate selection queries with a variable workload belong to this category. We utilize the following representative query in our study:

```
SELECT moteid, temp
FROM sensors
WHERE temp>39
```

Sensing Device & Communication: We use the energy model of Crossbow's new generation TelosB [1] sensor device to validate our ideas. TelosB is an ultra-low power wireless sensor that consumes 23mA in receive mode (Rx), 19.5mA in transmit mode (Tx), $7.8mA$ in active mode (MCU active) with the radio off and $5.1\mu A$ in sleep mode. Our performance measure is *Energy*, in *Joules*, that is required at each discrete time instance to resolve the query.

The energy formula is as following: $Energy(Joules) = Volts \times Amperes \times Seconds$. Our communication protocol is based on the ubiquitous for sensor networks IEEE standard 802.15.4 (i.e., the basis for ZigBee [3] which is used by many sensor devices including TelosB). Our data frames are structured as following [10]: Each message is associated with a 5B (*Byte*) TinyOS header [9]. This is augmented with an additional 6B application layer header that includes: (i) the sensor identifier (1B), (ii) the message size (4B) and the depth of a cell from the querying node (1B). In each message we allocate 2B for environmental readings (e.g., temperature, humidity, etc.), 4B for aggregate values (`Max`, `Min` and `Sum`) and 8B for timestamps. ZigBee's MAC layer [3] dictates a maximum data payload of 104 bytes thus we segment our data packets whenever this is required.

4.2 Energy for Single-Tuple Answers

In the first experimental series we assess the energy efficiency of the MicroPulse algorithm compared to its competitors TAG and Cougar. We choose to conduct this evaluation in isolation from the rest components (flash, weather board, etc.) in order to identify the distinct properties of the three algorithms we compare.

Figure 3 shows the energy consumption for the Intel54 dataset using the single-tuple query ST. We observe that TAG requires 11,227±2mJ, which is two orders of magnitude more energy than the energy required by MicroPulse (i.e., only 53±35mJ). This is attributed to the fact that the transceiver of a sensor in TAG is enabled for ≈2.14 seconds in each epoch (i.e., $\lfloor e/d \rfloor = 30/14$), while in MicroPulse it is only enabled for $\approx 146ms$ on average. Enabling the transceiver for over two seconds in TAG is clearly the driving force behind its inefficiency. Figure 3 also shows that the MicroPulse energy curve quickly drops to the mean value of $53mJ$ within the first epoch (i.e., the sudden drop at the beginning of the curve). Notice that MicroPulse runs very much like Cougar during the first epoch but our algorithm then intelligently exploits the waking window cost to preserve energy.

Finally, Figure 3 shows that the Cougar algorithm requires on average 882±250mJ, which is one order of magnitude more than the energy required by MicroPulse. The disadvantage of the Cougar algorithm originates from the fact that the parents keep their transceivers enabled until all the children have answered or until the local timer h [3] has expired (in cases of failures). Thus, any failure is automatically translated into a chain of delayed waking windows all of which consume more energy than necessary. One final observation regarding the Cougar algorithm is that it features a large standard deviation (i.e., $250mJ$), which signifies that certain nodes consume more energy than others. This is attributed to the fact that the cost of failures in Cougar is proportional to the depth of the node that caused the failure. In particular, failures at a large depth (i.e., closer to the leaf nodes) will generate a larger chain of waking windows, thus will be more energy demanding than failures that occur at a small depth (i.e., closer to the sink). We have repeated the experiment for the MTF and MTA queries and summarize the results in Table 2.

[3] We configured the child waiting timer h to 200ms.

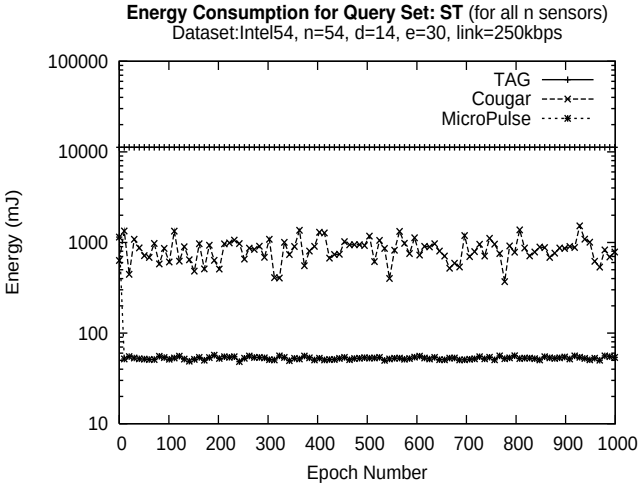

Figure 3. Energy Consumption for the Intel54 dataset with the query ST.

Table 2. Energy (in mJ) for the Intel54 Dataset

	ST	MTF	MTA
TAG	11,227±2	11,228±2	11,225±1
Cougar	882±250	893±239	877±239
MicroPulse	53±35	56±37	50±21

4.3 MicroPulse in a Large-Scale Network

In the second experimental series we evaluate the efficiency of MicroPulse in a large-scale sensor network, as this is provided by the Intel540 dataset. Figure 4, shows that MicroPulse requires only 3,446mJ on average (i.e., the mean of all three queries) while Cougar requires as much as 7,281mJ for the acquisition of values from all 540 nodes. This shows that MicroPulse retains a significant competitive advantage over Cougar even for larger-scale wireless sensor networks. Note that we have omitted the presentation of the TAG algorithm as it has a very high energy cost (i.e., $189,707mJ$).

We have repeated the experiment for the MTF and MTA queries and summarize the results in Table 3. For all queries we noticed that the MicroPulse-to-Cougar performance ratio is slightly increased (i.e., 47%) compared to the re-

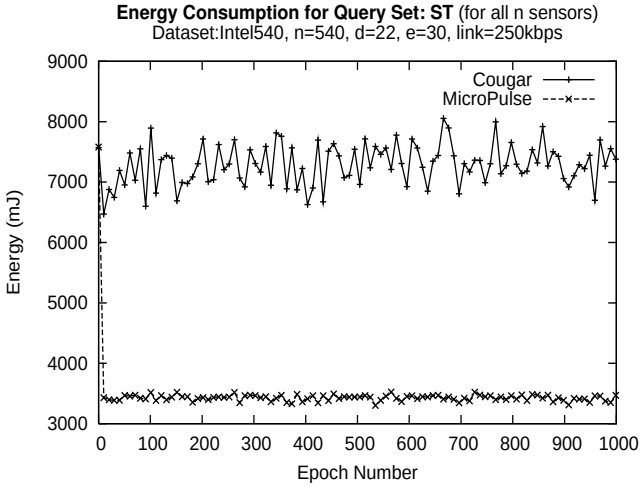

Figure 4. Energy Consumption for the Intel540 dataset with the query MTF.

Table 3. Energy (in mJ) for the Intel540 Dataset

	ST	MTF	MTA
TAG	189,691±53	189,707±49	189,670±51
Cougar	7,269±374	7,317±376	7,257±376
MicroPulse	3,431±140	3,510±126	3,398±132

spective performance ratio noticed with the Intel54 dataset (which was only 6%). Such an increase was expected as larger networks have a higher probability of transient network conditions and arbitrary failures. The above characteristics are causes that lead to the disruption of the query routing tree synchrony. Nevertheless, the MicroPulse approach is still 53% more energy efficient than Cougar under these limitations, thus can have many practical applications in large-scale environments.

5 Conclusions

This paper studies a workload-aware optimization technique of the query routing tree in Wireless Sensor Networks. In particular, we study the problem of optimizing the length of the waking window in order to minimize the consumption of energy. Our ideas are established on profiling recent data acquisition activity and on identifying the bottlenecks using an in-network execution of the Critical Path Method. We have provided an elaborate description of energy-conscious algorithms for disseminating the critical path cost and for maintaining such a cost in a distributed manner. Our qualitative and quantitative comparison with other predominant data acquisition frameworks has shown that MicroPulse offers tremendous energy reductions under realistic conditions. In the future we plan to investigate additional modules that can yield collision-aware query routing trees.

Acknowledgements: This work was supported by the US NSF under projects S-CITI (#ANI-0325353) and AQSIOS (#IIS-0534531), the EU under the project mPower (#034707) and the Cyprus Research Foundation under the project GEITONIA.

References

[1] Crossbow technology inc., http://www.xbow.com/.
[2] Intel lab data, http://db.csail.mit.edu/labdata/labdata.html.
[3] Zigbee specification 053474r06, version 1.0, 2004.
[4] U. Cetintemel, A. Flinders, and Y. Sun. Power-efficient data dissemination in wireless sensor networks. In *ACM MOBIDE*, 2003.
[5] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.
[6] J. Gross and J. Yellen. *Graph theory and its applications*. CRC Press, Inc., Boca Raton, FL, USA, 1999.
[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *ASPLOS*, 2000.
[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *ACM MOBICOM*, 2000.
[9] P. Levis. Tinyos implementation documentation, 2007.
[10] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys*, 2003.
[11] Q. Li, J. Beaver, A. Amer, P. K. Chrysanthis, A. Labrinidis, and G. Santhanakrishnan. Multi-criteria routing in wireless sensor-based pervasive environments. In *Pervasive Computing*, 2005.
[12] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *OSDI*, 2002.
[13] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. *ACM SIGMOD*, 2003.
[14] A. Sharaf, J. Beaver, A. Labrinidis, and K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *VLDB*, 2004.
[15] D. Sharma, V. I. Zadorozhny, and P. K. Chrysanthis. Timely data delivery in sensor networks using whirlpool. In *DMSN*, 2005.
[16] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *ACM SenSys*, 2004.
[17] A. Woo and D. E. Culler. A transmission control scheme for media access in sensor networks. In *MOBICOM*, 2001.
[18] Y. Yao and J. Gehrke. Query processing in sensor networks. *CIDR*, 2003.
[19] V. Zadorozhny, P. K. Chrysanthis, and A. Labrinidis. Algebraic optimization of data delivery patterns in mobile sensor networks. In *DEXA*, 2004.
[20] D. Zeinalipour-Yazti, P. Andreou, P. K. Chrysanthis, G. Samaras, and A. Pitsillides. The micropulse framework for adaptive waking windows in sensor networks. *DISN Workshop (MDM)*, 2007.
[21] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. *USENIX FAST*, 2005.