

On-line Sensing Task Optimization for Shared Sensors

Arsalan Tavakoli
University of California Berkeley
Berkeley, CA, USA
arsalan@cs.berkeley.edu

Aman Kansal and Suman Nath
Microsoft Research
Redmond, WA, USA
{kansal,sumann}@microsoft.com

ABSTRACT

Shared sensing infrastructures that allow multiple applications to share deployed sensors are emerging and Internet protocol based access for such sensors has already been prototyped and deployed. As a large number of applications start accessing shared sensors, the efficiency of resource usage at the embedded nodes and in the network infrastructure supporting them becomes a concern. To address this, we develop methods that detect when common data and common stream processing is requested by multiple applications, including cases where only some of the data is shared or only intermediate processing steps are common. The communication and processing is then modified to eliminate the redundancies. Specifically, we use an interval-cover graph to minimize communication redundancies and a joint data flow graph optimization to remove computational redundancies. Both methods operate online and allow application requests to be dynamically added or removed. The proposed methods are evaluated using applications on a road traffic sensing infrastructure.

Categories and Subject Descriptors

H.3.5 [Online Information Services]: Data Sharing

General Terms

Algorithms, Design

Keywords

wireless sensor networks, multi-query optimization, computation sharing

1. INTRODUCTION

Sensing systems now allow sensors to be shared among multiple users and applications [21, 4, 11, 12]. Open interfaces using the Internet protocol and web services have been prototyped to facilitate such shared access to sensors [10, 3, 20]. Multiple applications can use such sensing infrastructures to provide new functionalities using live sensor data. Also, within a single application, multiple users can access different data based on their needs [15]. As

the numbers of applications and users within applications grow, the amount of data to be provided from the sensors and the amount of computation performed on that data go up. This increases the load on the sensing infrastructure, limiting the number of allowable concurrent application requests. “Hot” sensors, i.e., ones that contain events of interest to several users, are likely to become especially overloaded.

Consider, as an example, the road traffic sensors deployed by the Department of Transportation on several roads, to measure the volume and average speed of traffic for the covered road segments. In a shared system, multiple sensing applications, such as driving directions computation, traffic characterization [27], congestion prediction [7], cab fleet management, or urban planning tools, may obtain data streams from these sensors. In existing systems each application obtains data directly from sensors and performs computations in isolation, which is not efficient. As an illustration of a specific application using these sensors, consider the following. A commuter wishes to avoid getting stuck in traffic on her way home from work. To choose a good departure time, she wants to calculate the average travel time on a route covering k road segments every 15 minutes in a time window extending from 3pm to 7pm, and then take the minimum over all of these, repeating for each day of the work-week. Similar data collection and computation tasks may also be submitted by many other commuters within the same city. The routes specified in the tasks may contain common road segments and have overlapping departure time windows. Clearly, fetching data from a sensor only once for multiple tasks will help resource constrained sensors avoid expensive communication, and computing the route latencies for shared segments along routes will allow the infrastructure to support a larger number of users. It is thus important to eliminate the computational and communication overlap among application requests, *to make the system more scalable* and to avoid excess load on hot sensors.

The problem of minimizing redundancies in the processing of multiple data streams has been considered in multi-query optimization for databases [22, 25]. Most such techniques statically analyze a fixed set of queries to find a globally optimal execution plan. As pointed out in [14], such a “compile-time” approach is prohibitively complex for many real-world streaming scenarios and is unable to handle dynamic arrival and departure of queries. To address these limitations, recent works [14, 13, 18] proposed techniques that optimize the stream processing on-the-fly. However, in the context of shared sensor networks, such existing techniques suffer from the following limitations:

No Optimization for Communication Redundancy: Each task requires data from one or more sensors periodically. Existing stream based query optimization techniques are geared toward centralized databases and do not consider the communication overheads of ob-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IPSN'10, April 12–16, 2010, Stockholm, Sweden.

Copyright 2010 ACM 978-1-60558-955-8/10/04 ...\$10.00.

taining the data, but this aspect becomes important for sensor networks. Our methods optimize the communication cost by re-using data collection across multiple tasks with similar, though not necessarily identical, temporal requirements.

Insufficient Optimization for Computational Redundancy: Existing techniques are unable to take full advantage of commonality in computation, especially when only intermediate computations overlap and when computation involves a sequence of multiple aggregation operations. Our system extracts such partial computation overlaps.

We present Task-Cruncher, a system that takes advantage of the spatio-temporal redundancy among multiple sensing tasks. Task-Cruncher makes the following contributions:

To address the first limitation mentioned above, we abstract the overlap between sampling requirements of different tasks as an interval-cover graph. We show that a dominating vertex set of the graph corresponds to an optimal sampling schedule. We also present a novel on-line algorithm to find this optimal schedule and show that it can be efficiently implemented on TelosB motes.

To address the second limitation, Task-Cruncher models tasks as data flow graphs and applies a novel on-line algorithm to dynamically optimize the graph structure as tasks enter and leave the system. The algorithm has a low overhead such that cost of graph optimization does not exceed the achieved savings. The algorithm can handle tasks with arbitrary combinations of processing primitives.

Thirdly, we evaluate Task-Cruncher using a one-year long data trace from Washington State Department of Transportation traffic sensors, as well as an image stitching application representing application specific computational constraints. Results show that our techniques can reduce resource usage by up to 45% compared to existing techniques, and achieve savings of over 70% of the theoretical maximum.

2. SYSTEM OVERVIEW

This section describes the key design considerations and the overall system architecture for Task-Cruncher.

2.1 Design Considerations

The design of Task-Cruncher must consider the specific constraints presented by a shared sensing substrate.

Communication. The first design consideration arises due to the resource constraints at the sensors themselves. Many sensors may be wireless and battery operated. Some sensors may be connected to the Internet using low bandwidth connections or remote links. They may be running on low-end processors. These constraints make it crucial to minimize the communication resource usage. A key design goal for Task-Cruncher is thus to *minimize redundancy in communication*. If data requests from multiple applications have similar temporal characteristics, the sensors should send the minimum amount of data that satisfies all applications. By reducing the communication load on each sensor, the number of supported applications can be increased.

Computation. A second design consideration is the computational resource overhead in processing the sensor streams. This computation is likely to be performed at central servers that maintain an index of available sensors and make them available to applications. Even though the server may be wall-powered and have a relatively high bandwidth connection to the Internet, the large number of sensors and applications served imply that the server resource usage per sensor and per task for performing online data processing on sensor streams cannot be very high. As the number of tasks increases, it is no longer efficient to perform the aggrega-

tion required by multiple applications in isolation. Suppose two wireless cameras are imaging a scene and many applications request the panoramic image generated by stitching the images from two cameras. Clearly, if the server can detect that more than one application has requested the result of the same computation, in this case the panoramic image generated from the images from the same two sensors, performing the stitching once per frame update interval and sending the stitched image to all requesting applications is preferred.

However, the detection of such computational overlap is not trivial. The set of sensors streams to be aggregated for one application may have only a partial overlap with that of another. Some of the intermediate steps required for two computations may be common, even when the final results are different (e.g. one application may request a sum of values, while another may request the mean, where the sum is computed as an intermediate step towards computing the mean). Another key design goal for Task-Cruncher is thus to *minimize the computational redundancy* in view of the spatio-temporal characteristics of the tasks served.

Dynamic Adaptation. As multiple independent applications use the shared sensors, tasks may be added and removed as per changing application requirements. As a result, both communication and computation redundancy elimination methods must *adapt dynamically* to changing tasks being concurrently served. Also, the cost of detecting the redundancies should not be greater than the savings achieved by exploiting the redundancies. The design of Task-Cruncher requires methods to identify all spatio-temporal overlaps in an on-line manner and when advantageous, optimize the processing to share the computation for common portions of multiple tasks.

Flexibility. Since the sensing infrastructure is shared by a vast range of applications, many different types of computation should be supported. Hence, we allow for arbitrary data processing and aggregation operations to be specified by composing primitive operations. However, exploiting a partial overlap between two tasks relies on breaking down a computation into the overlapping and non-overlapping sets. Many primitive operations used in sensing tasks are both *distributive* and *algebraic* (such as Sum, Max, Average, etc.) where the final result can be computed from partial results computed over disjoint partitions of input values. Breaking down such computations is straightforward. For other operations, breaking down of the computation may have constraints on the ordering or set of values that may be processed together. For instance, if the processing operation is stitching of images, only sensor subsets with overlapping fields of view can be stitched. In these cases, the breakdown of the computation must respect the constraints of the computation. The methods in Task-Cruncher extract overlap while respecting the correctness of the computation. Data processing tasks where the computational constraints do not allow any break down of the operations are performed without removing redundancies.

In addition to the redundancy minimization performed in Task-Cruncher, sensors may perform additional optimizations such as entering sleep states or low power listening modes when there are no application requests. Computational operations that do not depend on other sensors may be pushed to the sensors [6, 9, 16]. Such optimizations are complimentary to Task-Cruncher.

2.2 System Architecture

Task-Cruncher consists of the two key components (Figure 1) discussed below.

The *communication redundancy minimizer* receives streaming requirements for all tasks that use this sensor. It determines the

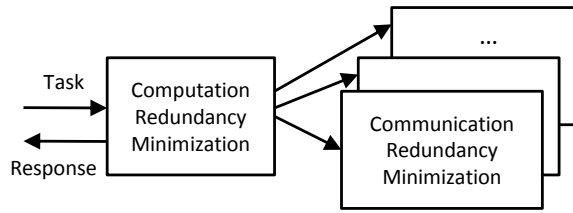


Figure 1: Task-Cruncher system architecture

minimum common set of time instances at which samples must be collected and uploaded to satisfy all tasks, and updates this set as tasks are added or removed. This module is implemented at the individual sensors as it does not require information from other sensors. This allows the communication overhead to be saved at the sensor itself. As shown in Section 5 using an implementation on TelosB motes, the overhead is extremely low. Detailed design of this module is presented in section 3.

The *computational redundancy minimizer* receives all tasks submitted by applications and jointly optimizes the computation across these tasks. This module must operate at a central point through which all tasks pass. A natural location for this module then is an index server that maintains a database of all available shared sensors for applications, since this server would likely be contacted by all applications to discover the sensors of interest. In SenseWeb [12] for instance, the central server that coordinates sensor access among applications can implement this module. This service may also be distributed across multiple servers as discussed in section 6, but for our system we assume a centralized server.

System Operation: The system operates as follows. An application that needs to sense the environment submits a sensing task specifying the spatial region to be covered, the time window for which the data is to be streamed, the sampling period, desired spatial density of sampling, the type of sensors to be used, and related characteristics. The application also specifies the computation to be performed on the data such as averaging over time, summation over space, evaluating maxima, or application specific combinations of operations. The task is received by the central server, and the relevant sensors are determined based on the spatial coverage and sensor types desired, using an index of all sensors stored at the server. The temporal sampling requirements of each task are sent to the relevant sensors. The sensor is thus only responsible for collecting the required data and the communication redundancy minimizer at the sensor uploads sufficient data to satisfy all tasks. The computation redundancy minimizer at the central server will compute the metrics requested by applications in an optimized manner, using the sensor data streams.

3. COMMUNICATION REDUNDANCY

Communication redundancy arises when a sensor is requested to collect data for multiple tasks. Typically, the task element local to a sensor involves collecting samples according to the temporal requirements of the task. Hence, the key issue in communication redundancy reduction is managing the overlap along the temporal axis. If the temporal requirements (e.g., interval between samplings) of two or more tasks are exactly the same within the overlapping portion of their time windows, minimizing this redundancy is trivial. However, in practice, the temporal requirements may not be exactly identical but have certain similarity. This similarity can be leveraged to reduce the number of samples uploaded, resulting

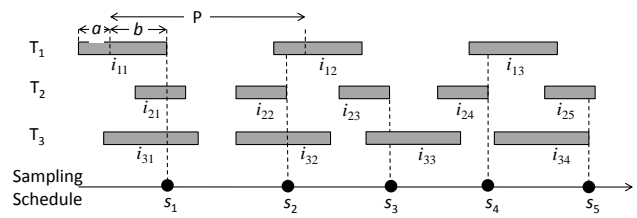


Figure 2: Sampling window sequences of three tasks

in direct resource savings at the sensor, and in the communication network overall.

3.1 Data Acquisition in Task-Cruncher

Consider a set of tasks $\{T_i, i = 1 \dots n\}$ served by a sensor. In Task-Cruncher, each task T_i defines its periodic sampling requirement with three parameters: a period p_i , a negative tolerance a_i , and a positive tolerance b_i , all in seconds. The tolerance parameters define a *sampling window* $[t - a_i, t + b_i]$, where t is a multiple of the period p_i , within which the sensor must take a sample at least once to satisfy the task. In practice large negative tolerances are often acceptable because the application wants to sample at least every p_i seconds and more frequent sampling is only avoided for efficiency reasons. Thus, as long as an application can accept faster samples, negative tolerances exist. Large positive tolerances are less frequent and the system may use $b_i = 0$ (our evaluation uses $b_i \leq 0.1p_i$).

Figure 2 illustrates sampling windows, as shaded rectangles, for three tasks. For each task, the above parameters define a *sampling window sequence*, such that the sensor must sample at least once within every sampling window. For a single task, this can be achieved, for example, by sampling and uploading at the beginning of each sampling window.

The parameters a_i and b_i reflect the flexibility that a sensor has. If the sensor intelligently determines the sampling instances so that a single sample can satisfy multiple tasks, data communication overhead is reduced. For example, in Figure 2, samples collected by the sensor at time s_1 satisfies three tasks, and we say that the sample at time s_1 covers the sampling windows i_{11} , i_{21} , and i_{31} . A *sampling schedule* is given by a sequence of time instances that cover all sampling windows of all active tasks. The instances s_1, \dots, s_5 in Figure 2 show an example sampling schedule. Designing the communication redundancy minimizer for Task-Cruncher, is equivalent to solving the following problem:

Minimum sampling schedule problem: *Given multiple sampling window sequences for tasks served by a sensor, find the sampling schedule with the minimum number of samples.*

3.2 Interval-Cover Graph

We abstract the above problem with an *interval-cover graph* $G = (V, E)$ constructed as follows. Each sampling window corresponds to a vertex in V , with the following attributes: a window $[s, e]$, start-time s , and end-time e . There is a directed edge $(i \rightarrow j) \in E$ if and only if the end-time of i intersects with window of j . We call a vertex j a neighbor of a vertex i if and only if there is an edge $i \rightarrow j$. Figure 3 shows the interval-cover graph for the sampling window sequences in Figure 2, where the vertex labeled mn corresponds to the sampling window i_{mn} .

A dominating set of graph $G = (V, E)$ is a set of vertex $V' \subseteq V$ such that every vertex not in V' is a neighbor of at least one vertex in V' .

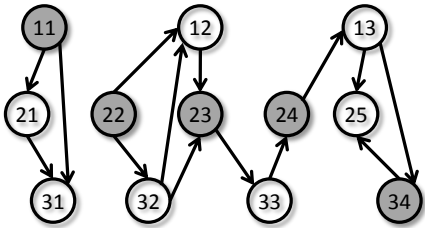


Figure 3: Interval-cover graph of the sampling window sequences in Figure 2. Shaded vertices constitute a minimum dominating vertex set.

CLAIM 3.1. *A dominating set of an interval-cover graph gives a feasible sampling schedule, where the sampling instances are given by the end-times of vertices in the dominating set.*

To see the correctness of the above claim, note that when a vertex v_1 is a neighbor of a vertex v_2 (i.e., there is an edge from v_2 to v_1), sampling at v_2 's end-time satisfies v_1 as well, since by definition the window of v_1 intersects with the end-time of v_2 . Thus, sampling at the end points of all vertices in the dominating set satisfies the sampling requirements of all the vertices, including the ones not in the dominating set.

CLAIM 3.2. *Any feasible sampling schedule corresponds to a dominating set in the interval-cover graph.*

To show the correctness of this claim, we modify a given feasible schedule S to an equivalent canonical schedule S' with the same number of samplings as follows: move every sampling instance in S to the earliest end-time of windows containing the sampling instance (thus, each sampling instance in S' is the end-time of some sampling window). Suppose the sampling window w_i corresponds to the vertex v_i in G . Without loss of generality, consider a sampling instance at the end-time of window w_1 and intersecting a set of windows $W = \{w_2, w_3, \dots, w_k\}$. Thus, in G , there will be edges between v_1 to all vertices in $\{v_2, v_3, \dots, v_k\}$. In other words, v_1 will dominate $\{v_2, v_3, \dots, v_k\}$. Since S' is a feasible schedule, its sampling instances cover all windows, and hence the vertices corresponding to the sampling instances dominate all the vertices in G . Thus, S' gives a dominating set in G .

The above two claims imply that solving the minimum sampling schedule problem is equivalent to finding the minimum dominating set in the interval-cover graph. The shaded nodes in Figure 3 show a minimum dominating set.

Finding a minimum dominating set of a general graph is NP-Hard. However, the problem can be solved in polynomial time for many special classes of graphs, including for widely-studied interval graphs [5]. Note that, an interval-cover graph is not equivalent to an interval graph; in an interval graph, two vertices are neighbors if and only if two intervals overlap with each other at any point, while in interval-cover graph, neighborhood is determined by the intersection of one interval and the end-time of another interval. Thus, minimum dominating set of interval graphs can not directly be used for our purpose, since it represents a set of intervals, while our solution requires finding the exact time points for sampling. For example, in Figure 2, if we abstract the intervals as an interval graph, a minimum dominating set will be given by the intervals $\{i_{11}, i_{12}, i_{23}, i_{13}\}$ since these intervals overlap with all other intervals. However, in our solution, a minimum dominating set will be given by the set of 5 sampling times shown in Figure 2, or the set of 5 shaded nodes in Figure 3.

Algorithm 1 GreedySample()

Require: Decides when the sensor needs to collect samples

Definitions:

G : an interval-cover graph

- 1: $G \leftarrow \emptyset$
 - 2: On arrival of a task T_i with parameters (p_i, a_i, b_i) , do $AddVertex(G, t' - a_i.t' + b_i)$, where t' is the next instance of time which is a multiple of p_i
 - 3: On departure of a task T_i , remove the corresponding vertex from G (and remove relevant edges)
 - 4: **if** current time $t ==$ the smallest end-time of any vertex v in G **then**
 - 5: Collect sensor sample at time t
 - 6: **for** each neighbor n of v **do**
 - 7: Remove the vertex n from G
 - 8: $AddVertex(G, n.t_1 + p, n.t_2 + p, p)$
-

Algorithm 2 AddVertex(G, t_1, t_2, p)

Require: Given a sampling window $[t_1, t_2]$ and its period p , add a vertex in the interval graph G

- 1: Create a vertex v with start-time t_1 , end-time t_2 , window $[t_1, t_2]$, and period p
 - 2: Add edges from v to existing vertices in G whose window overlaps with t_2
 - 3: Add edges from existing vertices to v if their end-times overlap with the window $[t_1, t_2]$
-

Also note that our target algorithm needs to work online with a dynamically changing interval-cover graph, since new tasks may be received by the sensor and old tasks may expire at any time. Moreover, since tasks are periodic, their sampling window sequences can be very long, resulting in a very large graph for offline processing (Figure 2 shows only parts of the sampling window sequences of tasks; each sequence can extend arbitrarily to the left.)

3.3 An Optimal Online Algorithm

We now present an online algorithm to determine, on the fly, the optimal minimum sampling schedule for a given set of sampling window sequences. The algorithm maintains an interval-cover graph G that has exactly one vertex for each task, representing the current or the next sampling window of the task. Each vertex maintains the start- and the end-time of the corresponding sampling window. After the current sampling window of a task expires, the corresponding vertex is deleted from G and a new vertex with the next sampling window is added to it. As tasks arrive and depart, corresponding vertices are added to or deleted from G . Thus, G evolves over time and it captures the information required for our algorithm to make optimal online sampling decisions. Our algorithm can be easily implemented on low power sensor nodes using a small memory footprint as shown in section 5.

Algorithm 1 shows the pseudocode of our greedy algorithm. Given the current instance of the interval-cover graph G , the algorithm considers its vertices in ascending order of their end-times. Suppose, at some point it considers the vertex v . Then, the algorithm collects a sample at the end-time of v . In addition, it removes all v 's neighbors from G . Intuitively, the window of any neighbor n of v contains the end-time of v , and thus taking a sample at that end-time automatically satisfies the sampling window of n . Since we do not need to create additional samplings for n , it can be safely removed. However, since the tasks are periodic, and G contains only the current or the next sampling window of a task, the deletion of n must be followed by addition of a new vertex, corresponding to the next sampling window of the task, to G . The process goes on until all tasks depart from the system.

The following theorem shows our algorithm's optimality:

THEOREM 3.1. *The algorithm `GreedySample` collects the minimum number of samples to cover all sampling windows of all tasks.*

Proof. The proof is by induction. We show that the number of samples PC collected by `GreedySample` is no more than the cost PC' of any other algorithm \mathcal{A} (i.e., \mathcal{A} can be the optimal offline algorithm designed by an oracle).

In the base case of induction, the graph G consists of only one vertex, and \mathcal{A} must take at least one sample to cover it. Therefore, $PC'(G) \geq 1$. `GreedySample` generates only one sampling (at the end-time of the vertex). Thus, $PC(G) = 1 \leq PC'(G)$. Now consider an arbitrary interval-cover graph G . Without loss of generality, suppose vertex X has the earliest end-time. Now partition the graph G into $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that V_1 includes X and all its neighbors and $V_2 = V - V_1$. According to the induction hypothesis, `GreedySample` uses the minimum number of samplings for G_2 . Since the interval of X is disjoint of any interval in G_2 , samplings needed for G_2 do not cover the vertex X . Thus, \mathcal{A} will need at least one sampling (besides the samplings required for G_2) to cover G_1 . i.e., $PC'(G) \geq 1 + PC(G_2)$. `GreedySample` generates exactly one sampling for G_1 , and thus $PC(G) = 1 + PC(G_2) \leq PC'(G)$. This proves the theorem. \square

Implementation with priority queues. The description of Algorithm `GreedySample` explicitly maintains the current interval-graph, which needs to be updated every time a sample is collected, a task arrives, or a task departs. However, we observe that edges of the graph are required only just after the samples are collected, in order to determine the neighboring nodes of the current vertex (in Line 6 of Algorithm 1). Thus, the maintaining all the edges of the graph all the time is not strictly necessary.

Exploiting this observation with an efficient implementation that requires two priority queues Q_1 and Q_2 of vertices. Q_1 maintains vertices such that the front vertex has the smallest start-time t_{Q_1} , while Q_2 maintains them such that the front vertex has the smallest end-time t_{Q_2} . The queues maintain only the vertices of G ; directed edges between vertices are implicitly given by the order of the vertices in the queues. When tasks arrive, new vertices are created and added to Q_1 and when they depart, corresponding vertices are deleted from Q_1 or Q_2 , depending on which queue they are in at that time. At time $t = t_{Q_1}$, all vertices with start-time equals to t_{Q_1} are dequeued from Q_1 and inserted into Q_2 . At time $t = t_{Q_2}$, a sampling is added to the sampling schedule, and all vertices in Q_2 are dequeued, updated with their next sampling windows (by adding their periods), and inserted to Q_1 . It is easy to see that at time $t = t_{Q_2}$, the vertex at the front of Q_2 has directed edges to all the vertices in Q_2 in G , since the windows of all the vertices in Q_2 contains t_{Q_2} . In other words, Q_2 essentially captures all the neighbors of the front vertex of Q_2 at time $t = t_{Q_2}$. Thus, scheduling only one sampling at $t = t_{Q_2}$ covers all the vertices in Q_2 , and hence can be deleted from G (i.e., removed from Q_2).

In effect, the algorithm is sampling at the latest time possible for the task with the earliest deadline. The priority queues enable efficient book-keeping that reduce the execution overhead of the algorithm to a small number of scalar comparisons and pointer updates.

4. COMPUTATIONAL REDUNDANCY

This section describes techniques to reduce redundancy in computation due to overlap (including partial overlap in intermediate steps) among tasks.

4.1 Task Model

To formally specify the processing required by a task, we use

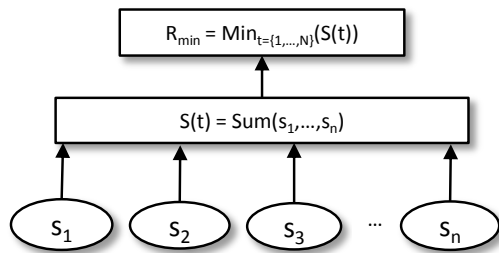


Figure 4: An example task in traffic sensing.

the following task model. Each task is represented as a directed acyclic graph (DAG), denoted $T_i(S_i)$ where S_i represents the set of sensors used.

Consider the road traffic sensing application introduced in section 1. The example task of finding the best departure time by computing the travel times for one specified route over a desired time window may be represented as the DAG shown in Figure 4. The *leaf nodes* (labeled s_1, s_2, \dots, s_n) represent the data received from sensors on the n road segments comprising the route. *Intermediate nodes* represent computations. Each intermediate node is characterized by a *type*, P , that specifies the computation it performs. For example, the intermediate level node labeled $S(t)$ represents the computation of the total travel time and the highest layer node, R , represents the computation of the minimum of the travel times over multiple time instances. Such DAG's can be used to represent complicated tasks consisting of several intermediate processing steps. The task may extend over several time instances. When a task is received, such a task-DAG is created for it. The leaf nodes are generated by selecting relevant sensors from an index of available sensors, based on the region specified using a geographical polygon, a travel route, or by certain properties of sensors such as all sensors in elevators or bird-houses.

The DAGs from tasks currently served are combined into a data flow graph, $\mathcal{F}(S)$, over the set S of sensors. Minimization of redundancy involves dynamically optimizing the structure of this data flow graph $\mathcal{F}(S)$ as tasks enter and leave the system.

Consider as a simple example, two tasks $T_1(S_1)$ and $T_2(S_2)$ to be traffic application tasks similar to the task in Figure 4 but on slightly different routes: $S_1 = \{s_1, s_2, \dots, s_5\}$ while $S_2 = \{s_3, s_4, \dots, s_7\}$. Writing the two task-DAG's such that the common sensor nodes are not repeated¹ leads to the data flow graph $\mathcal{F}(S)$ shown in Figure 5. Assuming temporal similarities have already been addressed by the communication redundancy minimizer, we only focus on computational redundancies for sensor data samples received with a common timestamp.

In Figure 5, the $S(t)$ nodes have a partial overlap in computation: the calculation of the sum $s_3 + s_4 + s_5$. To understand the procedure used to remove such overlaps, consider first a *computation*:

$$C_{Sum} = s_1 + s_2 + s_3 + s_4 \quad (1)$$

This computation may be viewed as consisting of the following primitive operations:

$$s_1 + s_2 = c_1 \quad c_1 + s_3 = c_2 \quad c_2 + s_4 = C_{Sum}$$

¹Writing the duplicate nodes only once has the same effect as the multi-query aggregation technique used in [17] that takes a union of the regions covered by multiple queries to generate a single combined query, where only tasks with the same temporal characteristics were considered.

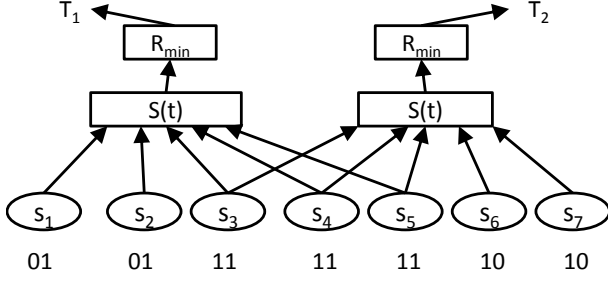


Figure 5: Data flow graph with two tasks. Binary labels at the bottom represent bitmasks for Algorithm 3

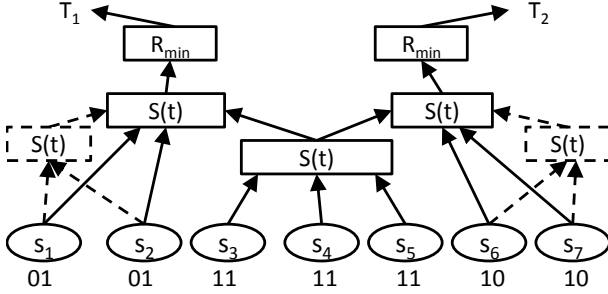


Figure 6: Optimized data flow graph.

A computation node may be split up in multiple ways, such as $s_2 + s_3 = c_1$ could be the first operation in the above example. The key to minimizing redundancies is to split the node in such a way that the overlapping operations are separated. For instance, in Figure 5, the intermediate nodes may be split to avoid redundant computation, leading to an optimized data flow graph shown in Figure 6 (ignore dashed lines and nodes for now).

4.2 Reduction Algorithm

Task-Cruncher converts an unoptimized data flow graph (such as Fig. 5), into an optimized one (Fig. 6) using the procedure summarized in Algorithm 3, named *ShareComputation()*.

The first step initializes $\mathcal{F}(S)$ to the DAG of the first task accepted. When the i -th task is added, Step 3 computes the merged $\mathcal{F}(S)$ by extending the set of leaf nodes in $\mathcal{F}(S)$ to include any additional sensors required by T_i . The leaf nodes send the sampling component of the task to the sensor and receive the responses. The subsequent steps reduce redundant computations, as follows.

Identification of Groups: The leaf nodes are referred to be at level $l = 0$. At the lowest layer of $\mathcal{F}(S)$, an *analysis set*, A , of nodes is formed, where A consists of all lowest level nodes used by $T_i(S_i)$ and their siblings. The siblings of a set S_i with respect to a type P , denoted as set $Sibling_P(S_i)$, are defined as nodes that share at least one common parent node of type P with any node in S_i . For the example above, when $T_2(S_2)$ is added, the siblings of set S_2 w.r.t type $P = Sum$ are sensors s_1, s_2 as they share a common parent of type Sum with $s_3 \in S_2$.

Suppose sensors in A serve k tasks. Each leaf node in A is assigned a *bitmask*, $B = [b_1, b_2, \dots, b_k]$ where b_i represents a bit value and has the value 1 if this leaf node services task i , and 0 otherwise. For nodes that existed earlier in $\mathcal{F}(S)$, this step results in updating their old bitmask. For our previous example, the bitmasks assigned to each leaf node are shown at the bottom in Figure 5 w.r.t.

Algorithm 3. *ShareComputation()*

1. Initialize $\mathcal{F}(S) = T_1$ where $S = S_1$
 2. Whenever new task i arrives, insert T_i as follows:
 3. $S = S \cup S_i$
 4. Level, $l = 0$. Type = P
 - (a) Analysis set, $A_{l,P} = S_i \cup Sibling_P(S_i)$
 - (b) Identify: To each node s_k in $A_{l,P}$, assign bitmask $B = [b_1, b_2, \dots, b_i]$ where bit $b_j = 1$ iff node s_k serves T_j .
 - (c) Group: For each unique bitmask generated, group as set G the nodes with that bitmask. Assign a parent node, g , of type P to each group G if computational constraints allow computing P for G separately.
 - (d) Vertical Merge: For all parent nodes, g , assigned (new or existing) above, if g has only one parent and that parent is the same type, merge that node into its parent. If g has only one child merge that into its child.
 - (e) Horizontal Merge: Any two assigned parent nodes (new or existing) that have the exact same children are joined into one. The vertical and horizontal merge steps are repeated while feasible at this level.
 - (f) Repeat steps 4a to 4e for all types for which there were parents in T_i at this level.
 5. Level, $l = l + 1$. Type = P
 - (a) Analysis set, $A_{l,P} =$ nodes at level l in T_i and their siblings w.r.t parents of type P .
 - (b) Repeat steps 4b to 4d for this level.
 - (c) Repeat above steps 5a to 5b for each type.
 6. Repeat step 5 for all levels.
-

to parents of type $P = Sum$. The bitmask used is similar to those in [14, 13, 18].

Splitting Calculations: Suppose N unique bitmasks are generated. At step 4c, nodes with a common bitmask are grouped together. The computation is distributed over the groups (if computational constraints allow) and for each group a parent node that carries out the computation is assigned.

Certain groups may already have a suitable parent, g , of type P . For others, a new intermediate parent node is generated. For cases where a new g is required, sometimes computational constraints may not allow separately performing a computation over set G , (such as a panorama may not be stitched if interleaving sensor images are left out of G) and then no new g will be assigned.

The new parent nodes are now connected to feed into the older parents of the lower layer nodes. When adding a new parent, if the child and the new parent already share an upstream node, this may lead to double counting of data in computations. So in this case, the child keeps only the longest path to the upstream node. Following the previous example, grouping the unique bitmasks shown in Figure 5 leads to the structure shown in Figure 6 where the new parent nodes added for groups $\{01\}$ and $\{10\}$ are shown using dashed lines while that for group $\{11\}$ is shown with a solid line.

Merge. Next, at step 4d, all non-leaf nodes that have only one parent of the same type as themselves, are merged into their parent. For instance, the dashed $S(t)$ nodes on the far right and left in Figure 6 are merged with their only parent, leading to the $\mathcal{F}(S)$ consisting of the nodes with solid lines only. Also at this step, all nodes that have only one child are merged into the child. This may happen when a single node has a unique bitmask and a new parent is created for it. Sometimes, when a new task is added, with the

same sensor footprint as the previously running tasks, difference in bitmasks for the new task may cause a duplicate parent node to occur. Step 4e removes such duplicates. Neither of the graph changes at the merge steps affect the correctness of the computation.

Some tasks may have more than one type of parents at the same level for the sensor nodes. If this is the case, step 4f specifies that an analysis set must be considered with respect to each parent type.

Repetition at Upper Layers: Similar steps are repeated at each higher layer. Note that the siblings forming an analysis set have a common parent type but may themselves may not be of the same type. In our previous example, only the right-most $S(t)$ node (solid lines) corresponding to T_2 comprises the analysis set as it has no siblings. In this example, repeating similar steps does not change $\mathcal{F}(S)$ and Figure 6 (solid lines only) gives the optimized $\mathcal{F}(S)$ after addition of task T_2 .

Task Entry-Exit Dynamics: In the above algorithm, only the analysis set and the portions of $\mathcal{F}(S)$ affected by the analysis set are scanned, rather than the entire $\mathcal{F}(S)$. This is advantageous for dynamic insertion and deletion of tasks. Task exits are handled simply by removing the highest node in \mathcal{F} that is serving this task. This recursively disables the links from child nodes and all nodes with no upstream links drop off $\mathcal{F}(S)$.

Optimality: This incremental algorithm does not guarantee that all redundancies are eliminated. To eliminate redundancy completely a scan of the entire graph $\mathcal{F}(S)$ may be necessary, rather than considering one analysis set at a time. One approach would be to take a logical AND of all possible pairs of unique bitmasks in an extended analysis set consisting of all nodes (at the same level), select the pair with the largest match in the result of AND, and form a new partial node for this subset. Reiterating after setting this subset to 0 until no matching subsets remain eliminates all redundancies. This become computationally inefficient with large system workloads. Evaluations of Algorithm 3 (Section 5) show that computations are reduced by over 70%. Since, reduction will always be lower than 100% because all calculations cannot be dropped, this suggests that the remaining margin for improvement is not very large.

Comparison With Existing Methods: Task-Cruncher improves upon existing methods along the following dimensions: First, existing methods for computational redundancy reduction [14, 13, 18] are designed for a single common aggregation operation. Our method identifies partial overlaps in computation even with different overall aggregation operations.

Second, our method works for tasks consisting of multiple steps of processing as opposed to a single aggregation step.

Third, even for a single aggregate computation, our method is able to reduce the redundancy by a greater amount than existing methods. For example, consider three tasks $T_1(S_1)$, $T_2(S_2)$, and $T_3(S_3)$ where $S_1 = \{1, 2\}$, $S_2 = \{1, 2, 3\}$, and $S_3 = \{1, 2, 3, 4\}$. The tasks $\{T_i\}_{i=1}^3$ each compute simply the sum of sensors in sets S_i respectively. Figure 7(a) shows the computation performed by [14] after optimization for this example. There are three unique sets of streams ((1,2), 3, and 4), leading to the bitmasks shown. Each set undergoes partial aggregation and then the partial aggregates relevant to each task are aggregated in the final aggregation step. A total of 4 sum operations is used (one operation for the sum node labeled c1, another for c2 and two operations for c3). On the other hand, Figure 7(b) demonstrates the data flow graph formed by Task-Cruncher. Algorithm 3 achieves better re-use and uses only 3 sum operations, fewer than needed by existing methods.

Finally, in [14], the bitmasks are computed for each data tuple as it arrives, i.e. this computational overhead is paid at the tuple arrival rate. In our approach, the bitmasks are computed only when

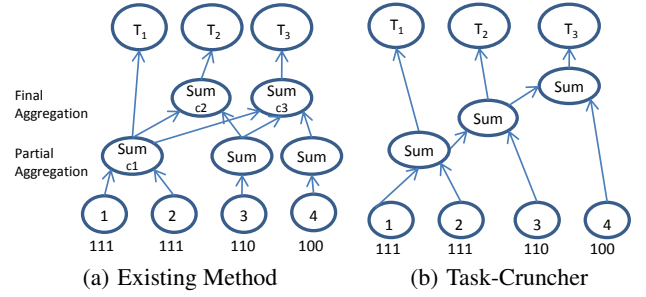


Figure 7: Data Flow Graphs

a new task is added or removed, i.e., only at the task arrival rate, which can be significantly lower than the tuple collection rate.

4.3 Algorithm Analysis

The reduction in operations achieved by Algorithm 3 may be analyzed as follows. Define \mathcal{T} to be the set of currently executing tasks, S be the set of sensors actively servicing tasks, and \mathcal{B} as the set of unique bitmasks currently active. Let B_{freq} denote the number of sensors with a unique bitmask B , and τ_B be the number of tasks served by bitmask B . The average number of tasks served by a node, X_{avg} becomes:

$$X_{avg} = \frac{1}{|S|} \sum_{B \in \mathcal{B}} \tau_B \cdot B_{freq} \quad (2)$$

Assuming each node produces one piece of data per period, this one datum will be involved in X_{avg} operations, and so the total number of operations, C_0 , without any computation sharing becomes:

$$C_0 = |S| \cdot X_{avg} - |\mathcal{T}| = \left(\sum_{B \in \mathcal{B}} \tau_B \cdot B_{freq} \right) - |\mathcal{T}| \quad (3)$$

$|\mathcal{T}|$ is subtracted because there are no *inter-task* operations.

In Task-Cruncher, counting an operation over every two values as one operation, the number of operations, C_1 , at the intermediate layers is:

$$C_1 = |S| - |\mathcal{B}| \quad (4)$$

since each sensor in $|S|$ leads to one operation except the $|\mathcal{B}|$ sensors at group boundaries. Also, the average number of tasks each bitmask serves is:

$$Y = \frac{1}{|\mathcal{B}|} \sum_{B \in \mathcal{B}} \tau_B \quad (5)$$

Using Y , the number of operations, C_2 , in combining the intermediate results into final responses is:

$$C_2 = |\mathcal{B}| \cdot Y - |\mathcal{T}| = \left(\sum_{B \in \mathcal{B}} \tau_B \right) - |\mathcal{T}| \quad (6)$$

Putting (4) and (6) together, the total number of operations, C_3 , is:

$$C_3 = |S| + \left(\sum_{B \in \mathcal{B}} \tau_B \right) - |\mathcal{B}| - |\mathcal{T}| \quad (7)$$

For Task-Cruncher the worst case is $|\mathcal{B}| = |S|$ when there is no grouping possible ($|\mathcal{B}|$ can not be greater than $|S|$). In this case, $B_{freq} = 1 \forall B \in \mathcal{B}$, and $C_3 = C_0$. Intuitively, this makes sense as it means that each node has a unique bitmask and no partial aggregates can be used for sharing computation. On the other hand, as $|\mathcal{B}|$ gets smaller, B_{freq} grows, leading to greater savings. Thus, as the number of tasks sharing each bitmask grows, more savings

are obtained. This analysis only considered tasks requiring a single computation step; similar analysis can be applied to multi-step tasks, replacing S with an analysis set made up of aggregate nodes. Practically achieved gains with realistic workloads are shown in section 5.

5. PERFORMANCE EVALUATION

We study the savings in computation and communication achieved using the following real world datasets:

Sensor Deployment: We use Washington State Department of Transportation’s traffic sensors in Seattle area. These sensors report average speed and traffic volume for freeway road segments. We obtained the measurements from all 492 sensors in that area for a period of 1 year.

Task Requests: We obtained data regarding which spatial regions across the metropolitan area were accessed on Microsoft mapping services by Internet users. This data serves as a proxy for spatial regions of interest for sensing applications.

Additionally, since the real world sensors above are scalar sensors and the common aggregation operations on these are fully distributive, we also consider a simulated deployment of image sensors where the aggregation operation is image-stitching and the overlap in fields of view of the cameras restricts the computation sharing. This deployment uses 100 cameras, with each sensor having an overlapping field of view only with two neighboring sensors. User requests are generated from a Zipfian distribution, with skew parameter 1.2, that models higher interest in more popular sensors as a power law distribution, inspired by the real user requests dataset that was also seen to be skewed.

For the application workloads, we consider a variety of computational tasks described along with the simulations.

Server-side Implementation: Our implementation of the data flow graph uses hashtables extensively, sacrificing space efficiency on the server in order to achieve speedier lookup times, meant to enable real time processing of a large number of task streams. All computation delays reported are for running Task-Cruncher on a 2.4GHz dual core system with 2GB RAM. We compare our system to a base case with no redundancy reduction, as well as our implementation of the algorithm from [14].

Sensor-side Implementation: The implementation on the sensor side is developed in TinyOS on TelosB motes. While the algorithm is provably optimal, it is also efficient to implement and uses only 110 lines of nesC code (with the priority queues implemented using arrays), resulting in 696 bytes of ROM footprint, and 25 bytes of RAM usage.

5.1 Experiment 1

In our first set of experiments, we use the following representative varied workloads:

W1: Commuters want to find the minimum travel time for their commutes over a given time window.

W2: An entity wants to know commuters’ average commute time during different periods of the day.

W3: A coffee retail chain wants to determine the maximum number of vehicles on roads near its stores to figure out how many promotional items to produce.

W4: A radio station wants to determine the peak number of vehicles in a given area over a set time window to determine prime scheduling.

W5: Department of Transportation wants to determine the period when the average number of vehicles in an area is minimal, to schedule construction.

Tasks from each workload are generated with random sampling

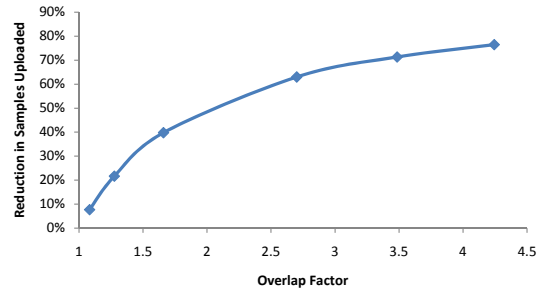


Figure 8: Sensor sample upload reduction

intervals (between 30 seconds and 5 minutes with random positive/negative tolerances ranging from 0-10%) and random aggregation time windows. Sensor sets used for each task are randomly chosen, based on a set of 4200 routes for route-oriented workloads (W1 and W2), and 3150 geographic rectangles from the Internet mapping service user trace for area-oriented workloads (W3, W4, and W5). The number of concurrent tasks was varied from 5 to 60 with each workload type equally represented. We define *overlap factor* to be the number of tasks each sensor node serves, averaged across all nodes. The overlap factor indicates the level of spatial redundancy among workloads. The above workloads use four different types of computational operators (sum, minimum, maximum, average). Each experiment was run 5 times, with the plotted data points representing the average (error bars show standard deviation in measured computation latency).

We begin by examining the performance of communication redundancy reduction, Algorithm 1. Figure 8 plots the reduction in the number of sensor samples uploaded. Noting that the reduction will always be lower than 100%, the algorithm achieves a large fraction of the achievable savings in the simulated scenario. Put in absolute terms, this is a reduction of nearly 4000 uploads per period (in the 60 task trial), which relieves a significant strain on underlying sensor resources.

Figure 9 portrays computational benefits. Note that in addition to spatial computation (i.e. computation that uses data from multiple sensors) the tasks also contain temporal computation (such as taking average/minimum over a time window of the task result from sampling periods within that time window). Our implementation includes some of the existing techniques for reducing temporal computational overlap, in particular the paned window method from [14].

The savings are significant: reduction in operations plateaus at approximately 75%, while the reduction in total computation cost reaches 70%. While the temporal computation savings are the same as existing work (and these are the dominant savings observed at low overlap factors) we see that Task-Cruncher performs increasingly better than [14] at high overlap factors by better exploiting the spatial computational overlaps, performing roughly only 55% as many operations².

Optimization Overhead: It is also interesting to observe the magnitude of the optimization overheads. Performing the optimizations when a new task is introduced incurs a computation cost (task removal is cheap), and this reduces the savings from reduction in operations for executing the task. As this cost is amortized over the

²The reduction in total calculation cost with [14] cannot be fairly compared: the reduction in operations is algorithm dependent and is reproduced correctly, but total calculation cost is hardware and implementation dependent and we did not have access to the implementation used in [14].

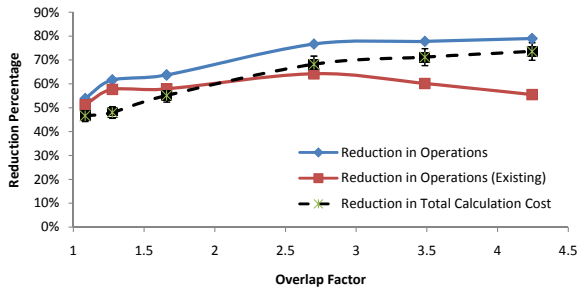


Figure 9: Reduction in operations and computation time for Experiment 1.

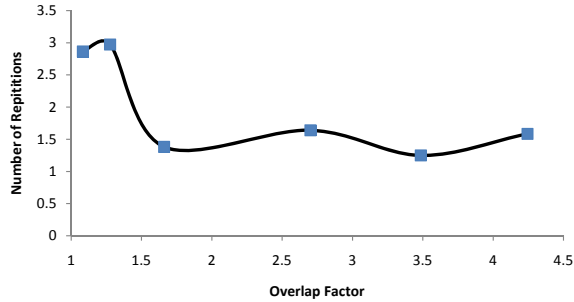


Figure 10: Break-even points for Experiment 1

lifetime of the task, the natural question is how many times should the operations in the task be repeated to recoup this initial task insertion overhead. Figure 10 depicts this *breakeven point* as a function of the overlap factor. At all but the lowest overlap levels, less than 2 repetitions suffice. Since we designed the system primarily for long-lived tasks repeating the computations for many time instances, we imagine that most workloads will benefit from the optimization process. When several short tasks are also served, the system can compare the life of the task with the empirical break-even times, such as shown in figure 10, and optimize $\mathcal{F}(S)$ only if the task lasts longer than a threshold duration.

5.2 Experiment 2

Next we evaluate computation redundancy reduction in a workload without temporal computation overlap to further highlight the difference between Task-Cruncher and existing work. We use a single type of tasks - the traffic application tasks depicted in Figure 4, with identical temporal parameters. The route for each task is selected randomly from over 4200 choices and determines the sensor set S_i used by it. We vary the number of concurrent tasks from 1 to 60 and measure the overlap factor for each set of concurrent tasks as generated. The gains at similar overlap factors are lower compared to the previous experiment due to the absence of temporal computation overlap. Figure 11 depicts the reduction in operations and total calculation cost. The overlap factor is higher due to less variability in workloads. Task Cruncher still outperforms existing approaches, particularly at higher overlap.

A further observation of interest is the difference between the total savings and the reduction in number of operations. The percentage reduction in total cost is computed with respect to the total cost of the naïve approach. The costs of adding new tasks and optimizing $\mathcal{F}(S)$ are included in the total cost for Task-Cruncher aside from the operations themselves. The overhead incurred in optimizing $\mathcal{F}(S)$ reduces the savings at lower overlaps.

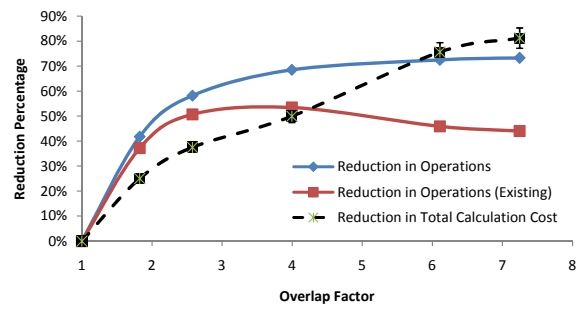


Figure 11: Reduction in operations and execution time in Experiment 2.

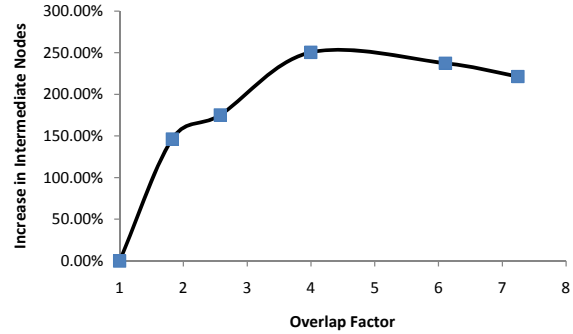


Figure 12: Increase in intermediate nodes for Experiment 2

Surprisingly, as overlap increases, the total cost savings in Task-Cruncher increase beyond the reduction in the number of operations. To understand this, we look at the overheads more carefully. Part of the overhead is the addition of new nodes in $\mathcal{F}(S)$ and part of it is the computation of optimizing $\mathcal{F}(S)$. The increase in the number of new nodes added in the optimized $\mathcal{F}(S)$ is plotted in Figure 12. A key observation from the graph is that after a high enough overlap, very few new intermediate nodes are added as the existing nodes are re-used more often. The naïve approach continues to add more nodes for each new task - causing the percentage of new nodes added to decrease for the optimized graph. The total cost saving includes not only the reduction in operations but also the reduction in node generation overhead. When the node generation cost saving exceeds the graph optimization costs, it further adds to the savings in operations. As a reference for the absolute time scale, the execution time including the overhead of task graph optimization ranged from 0.6ms to 2ms, normalized per task, on a 3MHz dual core processor, serving the workloads described.

5.3 Experiment 3

Next we evaluate computation reduction with a more complex processing operation, image stitching, where the nature of computation constrains the groups of sensors that can be processed together. Each of the 100 image sensors simulated has a field of view that overlaps only with neighboring two sensors and the tasks ask for panoramas stitched over valid sets of sensors. Figure 13 shows the results for computation redundancy reduction. The methods from existing work do not apply to this type of computation and we show the reduction in operations compared to a base case where no computational redundancy is removed. Again, a significant reduction in computation is observed. Here, savings do not necessarily grow with increasing overlap factor since the computation sharing

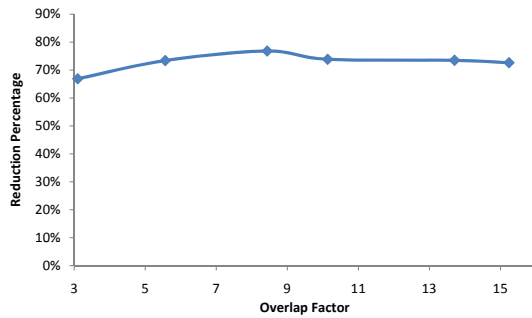


Figure 13: Reduction in operations for Experiment.

is restricted to neighboring sensors and the differences in aggregation steps for different tasks saturate the savings at high overlap.

6. DISCUSSION

This work demonstrated several methods to minimize redundancy in communication and computation. Here we discuss limitations and possible extensions to these methods.

In our present design we did not force any special requirements on the network infrastructure between the sensors and the server. If programmable elements such as geo-distributed servers or gateways for the sensors exist, then the data flow graph $\mathcal{F}(S)$, can be partitioned among such servers, with each server hosting parts of the graph relevant to its sensors. While the $\mathcal{F}(S)$ generated in our method can be distributed according to sensors served by different servers, a further research problem could consider the optimization of $\mathcal{F}(S)$ by jointly taking computational overlap and network topology into account.

Additional overlap in computation may be detected by inferring the semantics of computations performed by different combinations of primitive operations, i.e., when different DAG structures result in the same semantic computation. Semantic equivalence sometimes exist among data themselves. For example, Department of Transportation traffic sensors provide both speed and volume data. However, the physical transducer in the sensor only measures volume, and speed is actually computed from volume data. One approach to leverage such relationships among data is to use semantic web techniques that can automatically infer such relationships.

7. RELATED WORK

Computational overlap among streaming databases queries has been considered before [14]. Our system differs in that it is designed specifically for a sensing infrastructure, considering the cost of communication from sensors. Another difference is in the types of computational overlap addressed: we allow sequences of multiple aggregation primitives and optimize for partial computation overlaps, unlike the single aggregation step assumed in prior work.

Computational overlap among multiple queries were also studied in [8]. The goal however was to distribute the computations required across multiple nodes and minimize network traffic by transmitting partially aggregated results. The computation overhead of these techniques is high. Further, as stated in [8], the optimization only holds for a static query set and must be recomputed from scratch each time a new task is added or removed. Our methods reduce computational load and have a low task insertion-deletion overheads.

Streaming sensor data has also been considered in [17, 24, 1,

26, 23] using techniques such as query rewriting and in-network aggregation. Our work differs from these in both the optimization objectives and techniques. A shared and heterogeneous sensor network is considered in [19] but the focus is on executing tasks in a distributed manner rather than eliminating redundancy.

Common subexpression elimination in compiler optimization [2] is also related in principle, yet is often static, has no concept of temporal variance, and has no notion of communication cost. Multi-query optimization using directed acyclic graphs has been explored before ([22, 25]) but relies on static analysis of execution plans for queries, involving stateful operators such as *Join*. They do not consider temporal parameters and communication sharing, as they are not focused on stream processing.

8. CONCLUSIONS

We presented Task-Cruncher, a system to reduce communication and computation loads when multiple concurrent applications use a shared sensing substrate. Our communication redundancy reduction methods can reduce the number of samples required even when the temporal requirements are not exactly identical. The proposed algorithm was shown to be optimal, operates online, and has very low overhead. Further, we also minimize computational redundancies, not only when multiple tasks are performing the same aggregation operation but also when partial overlaps exist at intermediate steps. Task insertion-deletion is supported and savings achieved are close to optimal. These techniques enable efficient use of large scale sensing infrastructures by multiple applications. Our implementation not only prototypes the proposed methods but also combines the use of existing methods to yield a greater combined advantage.

9. REFERENCES

- [1] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *International Conference on Mobile Data Management*, May 2007.
- [2] J. Cocke. Global common subexpression elimination. In *Proceedings of symposium on Compiler Optimization*, 1970.
- [3] A. Dunkels. Full TCP/IP for 8-bit architectures. In *ACM MobiSys*, 2003.
- [4] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell. The bikenet mobile sensing system for cyclist experience mapping. In *Sensys*, 2007.
- [5] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.
- [6] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *MobiSys*, 2005.
- [7] E. Horvitz, J. Apacible, R. Sarin, and L. Liao. Prediction, expectation, and surprise: Methods, designs, and study of a deployed traffic forecasting service. In *UAI*, 2005.
- [8] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD*, 2007.
- [9] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SensSys*, 2004.
- [10] J. W. Hui and D. E. Culler. IP is dead, long live IP for wireless sensor networks. In *ACM Sensys*, 2008.

- [11] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Mui, E. Shih, H. Balakrishnan, and S. Madden. Cartel: A distributed mobile sensor computing system. In *Sensys*, 2006.
- [12] A. Kansal, S. Nath, J. Liu, and F. Zhao. Senseweb: An infrastructure for shared sensing. *IEEE Multimedia*, 14(4), 2007.
- [13] S. Krishnamurthy, M. J. Franklin, G. Jacobson, and J. M. Hellerstein. The case for precision sharing. In *VLDB*, 2004.
- [14] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, June 2006.
- [15] B. Kusy, E. Cho, K. Wong, and L. Guibas. Enabling data interpretation through user collaboration in sensor networks. In *Workshop on Applications, Systems, and Algorithms for Image Sensing (ImageSense)*, November 2008.
- [16] P. Levis and D. Culler. Mate: a tiny virtual machine for sensor networks. In *ASPLOS*, 2002.
- [17] M. Li, T. Yan, D. Ganesan, E. Lyons, P. Shenoy, A. Venkataramani, and M. Zink. Multi-user data sharing in radar sensor networks. In *SenSys*, 2007.
- [18] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.
- [19] M. Ocean, A. Bestavros, and A. Kfoury. snbench: Programming and virtualization framework for distributed multitasking sensor networks. In *Proceedings of the Second Int'l Conference on Virtual Execution Environments*, June 2006.
- [20] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: Design and implementation of interoperable and evolvable sensor networks. In *ACM Sensys*, 2008.
- [21] O. Riva and C. Borcea. The urbanet revolution: Sensor power to the people! *IEEE Pervasive Computing*, 6(2):41–49, 2007.
- [22] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhoje. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
- [23] J. Shneidman, P. Pietzuch, M. Welsh, M. Seltzer, and M. Roussopoulos. A cost-space approach to distributed query optimization in stream based overlays. In *Proceedings of the 1st IEEE International Workshop on Networking Meets Databases*, 2005.
- [24] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *DCOSS*, pages 307–321, 2005.
- [25] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, 2006.
- [26] S. Xiang, H. B. Lim, K.-L. Tan, and Y. Zhou. Two-tier multiple query optimization for sensor networks. In *ICDCS*, 2007.
- [27] J. Yoon, B. Noble, and M. Liu. Surface street traffic estimation. In *MobiSys*, 2007.