A Distributed Technique For Dynamic Operator Placement In Wireless Sensor Networks

Georgios Chatzimilioudis
Computer Science Department
University of California
Riverside, California 92507
gchatzim@cs.ucr.edu

Nikos Mamoulis
Computer Science Department
Hong Kong University
Pokfulam Road, Hong Kong
nikos@cs.hku.hk

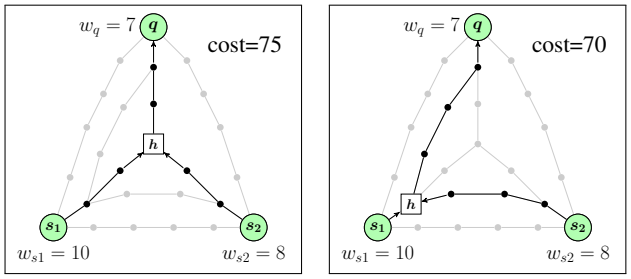
Dimitrios Gunopulos
Dept of Informatics and Telecommunications
University of Athens
15784 Ilisia, Greece
dg@di.uoa.gr

***Abstract*—We present an optimal distributed algorithm to adapt the placement of a single operator in high communication cost networks, such as a wireless sensor network. Our parameter-free algorithm finds the optimal node to host the operator with minimum communication cost overhead. Three techniques, proposed here, make this feature possible: 1) identifying the special, and most frequent case, where no flooding is needed, otherwise 2) limitation of the neighborhood to be flooded and 3) variable speed flooding and eves-dropping. When no flooding is needed the communication cost overhead for adapting the operator placement is negligible. In addition, our algorithm does not require any extra communication cost while the query is executed. In our experiments we show that for the rest of cases our algorithm saves 30%-85% of the energy compared to previously proposed techniques. To our knowledge this is the first optimal and distributed algorithm to solve the 1-median (*Fermat node*) problem.**

I. Overview and Motivation

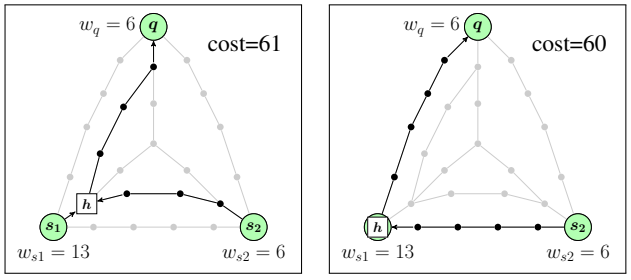
Network applications often need to perform in-network query processing. Sensor networks are being deployed in the physical or urban environment to benefit scientific research or security surveillance. An example of a query in a network, which is monitoring traffic in a busy downtown area, could be "How many cars took the same route of passing through intersections A, B and C?". To avoid the cost of communicating all the data lists from the nodes in regions A, B and C to the querying node, the query must be executed in-network. Data lists generated on the source nodes are fed into operators on intermediate nodes that combine several lists from different sources. The amount of data is reduced due to the selectivity of the operators and the data that reaches the querying node is the final answer.

An operator, that is involved in the in-network processing, can be placed on a node of the network. It takes in elements from source nodes, processes them, and sends the output to either another operator node or to the sink. Shipping elements over an edge in the graph imposes a cost that is dependent on the weight of the elements. Therefore, the placement of an operator can greatly affect the cost of answering a query since it affects the number of edges the elements have to travel over and the weight of the elements, since usually the output weight is not the sum of the input weights.



(a) Before (b) After

Fig. 1. Example of optimal operator placement: (a) Data flow during query execution before the operator placement is optimized, and (b) data flow during query execution after the operator placement is optimized. The *Fermat* node is an *external* node. The cost represents the cost of our objective function, not the actual communication cost.



(a) Before (b) After

Fig. 2. Example of optimal operator placement: (a) Data flow during query execution before the operator placement is optimized, and (b) data flow during query execution after the operator placement is optimized. The *Fermat* node is a *datanode*. The cost represents the cost of our objective function, not the actual communication cost.

It is typical to have continuous queries that require an answer over a continuous period of epochs. In most applications the sources and operators are not producing the same weight of elements in every epoch. Similarly, nodes in the network might be mobile resulting in different hop-distances between nodes in every epoch. Therefore, the initial operator placement might not be good enough for future epochs. It is a large overhead to re-run the algorithms for finding a good placement for the operators of the query. Instead, the technique followed in literature is to *update* the placement of just the operators that are affected by the weight change in order to keep the

978-0-7695-4048-1/10 $26.00 © 2010 IEEE
DOI 10.1109/MDM.2010.16

cost of query execution in the next epoch to a minimum. This operator placement update needs to be done with the least amount of communication cost overhead possible.

In Figures 1 and 2 let the nodes $s1, s2$ and q be the sources and the sink (henceforth called altogether *datanodes*) that send/receive data from the binary operator hosted at node h. Let w_i be the weight of the data to be sent from node n_i. We can see that by picking the right node to host the operator with the right distances from the *datanodes*, we can reduce the objective function for the communication cost of executing the query (difference between Figures 1(a), 2(a) and 1(b), 2(b)). Depending on the data loads and the path lengths, the optimal node to place the operator can be either an *external* node (figure 1), or one of the *datanodes*, i.e. a source or sink (figure 2). Our algorithm finds the optimal new placement for an operator while creating far less communication cost overhead than previous work. Note that we do not assume that the communication cost can be computed by summing the data-load sent over each link. We just use this as an objective function to estimate the actual communication cost. In our experiments we use a more accurate model for the communication cost.

Especially in high communication-cost applications minimizing the communication cost is the key issue. High communication cost networks play an important role in real world applications, as much as they do in research. The communication cost can be posed by monetary, temporal, resource or energy demands. As an example of a high communication cost network we will use a wireless sensor network throughout the paper. Wireless sensors have very limited energy resources. The task that has by far the highest demand in energy on a wireless sensor is the transmission and reception of data. Thus, the cost to pay for communicating is in form of energy. Minimizing the total energy consumed makes the whole network more energy sufficient, and minimizing the maximum energy consumption per node increases the network's lifetime.

Our *distributed Fermat node search* algorithm (*dFNS*) achieves two goals: finding the best node to place an operator and minimize communication cost doing so. To achieve this it 1) identifies the special case where no flooding is needed, 2) if flooding is needed, it minimizes the flooding radius, and 3) uses variable speed flooding and eves-dropping. Our algorithm is parameter-free, decentralized, optimal and outperforms previously proposed methods in minimizing communication cost overhead.

As shown in our experiments, there is a high chance (56%-85%) that the optimal node to place the operator is a *datanode* (source or sink), like in the example of Figure 2. Such a case can be identified by our algorithm and the operator is simply placed on the optimal *datanode* without any further communication cost to find the optimal operator node.

In any other case, *dFNS* finds the optimal operator node (*Fermat* node) by extending a flood from each of the *datanodes*. We generate a set of possible distance combinations that the new hosting node can have to produce a smaller hosting cost. Using these candidate distance combinations *dFNS* calculates the minimum possible radius for each flood, guaranteeing that the nodes that participate are kept to a minimum without compromising the optimality of the algorithm.

We adapt our proposed algorithm to existing work in wireless sensor networks. Using an existing framework for answering multi-predicate snapshot queries, we extend the framework to deal with continuous queries. The framework answers continuous queries in epochs and adapts the operator placement to data load changes. In our experimental evaluation we compare against the only other existing distributed algorithm for operator placement updates and show that using our proposed algorithm we can save $30\% - 80\%$ of the communication cost overhead.

In the following section we present previous work in this area. We formulate our problem definition and preliminary annotation in section III to be able to describe our algorithm in detail in section IV. In section V the framework in which our algorithm is implemented is described and in section VI we present our thorough experimental evaluation that shows the efficiency of our algorithm.

II. Previous work and the differences

The vast majority of the literature on operator placement in wireless sensor networks focuses on finding a good operator placement at query initialization as described in the introduction. Those algorithms are centralized; i.e., the basestation knows the location of the sensors or has complete knowledge about the network [1][2][3][4][5].

Ying et al [6] propose a distributed algorithm to do the same task as above, namely static operator placement. Nodes exchange information with their neighbors iteratively until they find the optimal placement for all given operators. Any node that has found a better cost for routing data or placing the operator, broadcasts this information to its neighbors. This algorithm is suited only for initial operator placement for queries with many operators, since it involves every node inside the network. Further, using this technique, it is hard to guarantee convergence, optimality, and low communication cost overhead.

Instead of sticking to a static plan, dynamic environments require adaptive query processing. A comprehensive survey on adaptive query processing is presented by Deshpande et al [7]. They categorize all techniques proposed that focus on using runtime feedback to modify query processing in a way that provides better response time, more efficient CPU utilization or network utilization. Our work would fall under the category of adaptive join processing with non-pipelined execution.

Next, we cite literature that deals specifically with operator placement adaptation, picking a new hosting node for one of the operators. There are two categories here: algorithms that pick the best neighboring node as the new host and converge to the optimal operator placement with time, and algorithms that find the best hosting node immediately. The former method is also called operator migration and we will call the later method *placement update*.

An alternative to operator placement update is operator *migration*, where the operator is moved gradually from one node to the next node towards the optimal placement. Algorithms following this principle are simple and their decision making is only local. On the other hand, it takes several epochs of query execution to reach the optimal operator placement. For the same reason, these methods suffer greatly from oscillating changes, that might force it to migrate an operator to a different direction before even reaching the optimal placement. Further, they are prone to local minima and impose extra cost during query execution in order to probe for a better operator host on every neighbor; [8][9][10] are works in this category.

Finding directly the optimal hosting node is the approach adopted in this paper. This problem is the same as the 1-median problem or single facility location problem in graphs. There is extensive literature on centralized algorithms for this problem [11], but not on distributed algorithms. In a distributed environment we can not adapt any of the centralized algorithms, since they all require that a central authority knows the topology of the whole network.

Zoe Abrams and Jie Liu in their paper named "Greedy is Good" (GIG) [12] propose a decentralized solution for the 1-median problem in graphs. They try to find the optimal hosting node of a single operator by flooding a small neighborhood around each *datanode*. It follows the intuition that the optimal hosting node will be somewhere close to all the datanodes. Their algorithm, *GIG*, aims to minimize the nodes involved in the flood by making use of some parameters set by the user. Surprisingly, they do not aim to minimize the number of messages exchanged by those nodes and thus the communication cost overhead is not minimized. Further, their algorithm does not guarantee to find the optimal operator node as we will see in the example in Figure 3.

We propose a parameter-free algorithm based on the same principles as *GIG*, but show how using the right techniques the right heuristics we can achieve a 30%-100% energy reduction compared to *GIG*. Some extra points that distinguishes our work from previous work is the following:

- our algorithm is distributed and we only collect a negligible amount of network information.
- we do not assume any location awareness for the nodes. It follows that we cannot use geographical routing to our advantage.
- our algorithm does not impose any overhead during the query execution phase.
- our algorithm is parameter-free, thus its efficiency is independent of any user input.
- our algorithm guarantees optimality.

III. Distributed Fermat Node Search: Preliminaries

Assume that in the network seen in Figure 1(a) the colored nodes are 3 customers s_1, s_2 and q. Each customer i needs quantity w_i from a commodity produced by a service that is currently hosted in node h. The cost of servicing customer i is the cost of sending weight w_i over the shortest path from node h to i. Find the node, that minimizes the cost of servicing the customers, to host the service. This is also known as the 1-median problem and can be extended to an arbitrary number of customers. Equivalently in wireless sensor networks we have an operator that collects data from a number of sources and sends the result of the operation to a sink. In Figures 1 and 2 we are dealing with binary operators (two sources s_1 and s_2, one sink q). Note that there are no restrictions in the relation between the quantities w_i, thus we can use any kind of operator.

We assume that sending data of weight w from node i to the operator host h and sending the same amount of data from operator host h to node i imposes the same cost. This is why we generalize and call both, sources and sinks, *datanodes*. Now the problem of finding the optimal operator placement is similar to the Fermat point problem [13], the three factory problem [14], and to the 1-median problem or single facility location problem. We call the optimal node to place the operator *Fermat* node and formulate our problem as follows:

Fermat node (or 1-median) problem definition
Given a weighted graph $G(N, L)$ and a set of datanodes $D \subset N$, find the Fermat node f in the graph that minimizes the cost of shipping data from the nodes in D to node f.

For the objective function that we use in our algorithm we assume that the cost of shipping data from node u to node v is proportional to the data load w_u to be shipped and the weight of the path used. The path weight $W(u, v)$ is equal to the sum of the weights of all links $l \in L$ that make up path (u, v): $W(u, v) = \sum_{l \in links(u,v)} w_l$, where w_l is the weight of the link $l \in L$. The cost of shipping data from node u to node v is defined as

$$t(u, v) = w_u * W(u, v)$$

This simplified version is used only as an objective function in our algorithm to estimate communication cost. Note that the computation of the actual energy consumed by the network when transmitting a message over a path is more complicated. In the network simulator we used to run our experiments the communication cost model is much more realistic.

Hosting cost, c, is the cost of sending data from the nodes in D to the hosting node h. It is equal to

$$c = \sum_{d \in D} t(d, f) \tag{1}$$

To minimize this cost we need to find the *Fermat* node and place the operator there. Finding the *Fermat* node involves a number of nodes that need to exchange messages. This imposes a communication overhead. The problem we solve in this paper is the following:

Our problem definition *Given a weighted graph $G(N, L)$, with identical link weights, and a set of weighted datanodes $D \subset N$, solve the Fermat node problem with minimum overhead.*

The communication cost in a wireless sensor network is the energy consumed for performing communication. The total

communication cost is the sum of the energy consumed by each node in the network. The maximum communication cost is the maximum energy consumed by a single node. By minimizing the number of nodes involved and the messages exchange between them, we keep the total communication cost and the maximum communication cost per node to a minimum.

Networks are inherently distributed, thus no node has global knowledge about the network topology. This rules out the application of one of many proposed algorithms in literature (Section II), that solve the *Fermat node problem*. We propose a fully distributed algorithm, that does not require the gathering of network information in order to compute the *Fermat* node. In the rest of the paper we will make extensive use of the following notions, that are formally defined here:

Shortest path length is the length of the shortest path between two nodes, i.e. u and v, and is denoted as $|(u, v)|$. We assume that the graph has bidirectional links, thus $|(u, v)| = |(v, u)|$.

Datanodes is the set of nodes D that either transmit data (*source nodes*) or receive data (*sink node*) to/from the node that hosts the m-ary operator (*hosting node*). The opposite of the datanode set is the *external nodes* set $X = (N - D)$. *Leader* node is the node that decides on initiating and terminating the *dFNS* algorithm.

Note that we assume error-free readings, otherwise we would need specialized techniques for probabilistic or model-base query execution [15]. We also do not assume any correlation between data that could assist us in saving energy during query execution [16]. The only information we need is what nodes the data is coming from/going to and the size in bytes. Our framework operates independently of how an operator placement update is triggered or oscilating updates, due to the rapid changes in the network, are avoided.

Distance Combination, α, denotes the k-ary set of shortest path lengths from all datanodes in D to the *hosting* node h.

$$\alpha = [\alpha_1, \alpha_2, .., \alpha_k] = [|(d_1, h)|, |(d_2, h)|, .., |(d_k, h)|]$$

where $d_i \in D$ and $k = |D|$. Each distance combination α has its hosting cost c_α. Note, when we have an m-ary operator it means we have m inputs and one output. It follows that the number of datanodes is $m + 1$ and thus $m + 1 = k$.

Flooding is the task of broadcasting data from one node to all its neighbors and repeating this for each neighbor. Each node broadcasts the data only once. By setting a restriction to the flooding *radius*, the broadcast message travels only *radius* hops away (Hops-To-Live = *radius*). This limits the nodes in the network that are flooded.

IV. Our Distributed Fermat Node Search Algorithm

We assume that a node h, that hosts an operator with datanodes D, knows the shortest path distances between any pair of datanodes in D. Note that the datanodes D of a single operator are only a very small subset of the nodes in the network ($|D| << |N|$). This information can be piggy-backed from each datanode $d \in D$ to node h, since there is direct unicast communication between them. The task of retrieving this information for each datanode d can be performed with efficient algorithms proposed in literature, such as doubling broadcast distance. Other than the datanodes of an operator, no other node in the network need to know their distance to any other node.

Each datanode d has its own hosting cost c_d. We call *best* datanode b the datanode with the minimum hosting cost $c_b = min\{c_d\}_{\forall d \in D}$. Using b as the solution to the *Fermat* problem is called *datanode solution*. There are cases where it is impossible for an *external* node to have better hosting cost than datanode b. Identifying those cases is simple and imposes no communication cost. All our techniques make use of hosting cost c_b of the best node.

A. Candidate Nodes

Candidate nodes are called the nodes in the network that have a hosting cost less than the hosting cost c_b of the best datanode. We need to compare all candidate nodes in order to find the actual *Fermat* node. Minimizing the number of candidate nodes is one of the key features of our algorithm. Note that there can be several nodes with the same minimum hosting cost, thus there can be several *Fermat* nodes. We just need to pick one of them.

To be able to calculate the hosting cost of an *external* node, we need to know its distance to the datanodes. Although external nodes might serve as relay nodes, they never communicate directly with any datanode, thus we cannot assume that they know their distance to each datanode in advance. To find the distance from an *external* node to each datanode we can initiate a flood from each datanode counting hops.

Nodes inside the intersection of all floods know the distance to all datanodes. This is true since we assume that the flood reaches a node over the shortest path from the initiator. These nodes can now calculate their hosting cost and, if it is smaller than c_b, they become candidate *Fermat* nodes. Candidate nodes report their hosting cost to the *leader* node, that decides what node is the actual *Fermat* node.

By reducing the number of candidate nodes, and therefore the messages (reports) sent to the *leader* node, we can save on communication cost. *dFNS* includes the hosting cost c_b of the best datanode in the initial flooding message as a cost threshold. Nodes, that have a hosting cost higher than this cost threshold, are not considered candidate nodes. Nodes that have a better hosting cost designate themselves as *Fermat* candidates and update the cost threshold inside the flooding message before it gets forwarded. We also let candidate nodes eves-drop messages sent by their neighbors in order to increase the probability that a message with a lower cost threshold is received to minimize the number of candidates.

B. Calculating All Candidate Distance Combinations

Before looking for the actual candidate nodes in the network we calculate all possible distance combinations that would qualify a node as a candidate node. These *candidate distance*

combinations are calculated at the datanodes without any communication with neighbors. This is done in order to be able to restrict the communication cost while searching for the actual candidate nodes inside the network. Most of the notation used here is defined in section III.

The datanode computes all candidate distance combinations A and their respective hosting cost c_α, $\alpha \in A$. This is the basic building block for our algorithm. To efficiently compute this set we use information about the shortest path lengths between the datanodes D. The distance combination that violate the shortest path length between the datanodes (triangle inequality) and have a greater hosting cost than c_b, the hosting cost of the best datanode b, are discarded. Formally the restrictions for each distance combination α are:

$$\begin{aligned} |(d_i, d_j)| &\leq a_i + a_j, \quad \forall i, j \in D \\ c_\alpha &< c_b \end{aligned} \tag{2}$$

The distance combination with the minimum hosting cost is called *ideal* distance combination and is denoted as ϵ. Depending on the network, a node with the distance combination ϵ might exist or not. If a candidate node has the ideal hosting cost c_ϵ, then no further action is needed to distinguish it as the *Fermat* node.

The algorithm we propose to compute the distance combinations is optimized to find the set fast and effectively, pruning combinations that do not satisfy the constraints in Equation (2) early. We start from the distance combination that corresponds to picking the best datanode b as the *Fermat* node. In this distance combination the value for $|(b, f)|$ will be 0. The other distances start from the minimum value possible that satisfies the constraints. We recursively increment each distance by 1. The pseudocode is shown in Algorithm 1. This algorithm returns the set of all possible distance combinations that would result in a smaller hosting cost than c_b. It also designates the *ideal* distance combination ϵ.

Algorithm 1 . CDCGenerator($distanceList$, i)

Require: list of datanodes and their loads,
distance between every pair of datanodes
1: $l_{current}$ =minimumDistance($distanceList$,i)
2: $distanceList \leftarrow l_{current}$
3: **while** ($distanceList$ satisfies constraints AND $l_{current}$ <maximumDistance($distanceList$,i)) **do**
4: **if** $distanceList.size ==$ number of datanodes **then**
5: C $\leftarrow distanceList$
6: update $bestCombination$
7: **else**
8: C $\leftarrow$ CDCGenerator($distanceList$, $i+1$)
9: **end if**
10: $l_{current} = l_{current} + 1$
11: remove last entry of $distanceList$
12: $distanceList \leftarrow l_{current}$
13: **end while**
14: **return** C

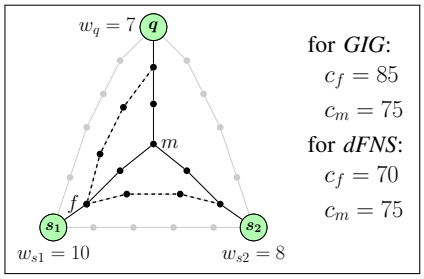


Fig. 3. *GIG* [12] is not an optimal algorithm. An example where *GIG* misses the optimal operator placement (f). This happens because the distances from candidate nodes to the datanodes are overestimated.

The function $maximumDistance(distanceList, i)$ returns the maximum distance that a node can have from datanode d_i so that it satisfies the constraints of Equation 2 and does not exceed the maximum distance between d_i and d_j, where $j > i$.

C. No Flooding Cases

If we get a distance combination set, that is empty when running the *CDCGenerator()* algorithm, it means that there cannot exist an *external* datanode with better hosting cost than the best datanode b. In those special cases, no flooding is needed to look for external *candidate* nodes. Node b is the optimal new operator host and our algorithm terminates by placing the operator there. Contrary to their characterization as *special*, these cases comprise 56%-85% of the cases as shown by experiments.

D. Flooding Radius

Flooding the whole network from each datanode in D poses a very big communication cost. Our algorithm efficiently restricts the flooding radius, guaranteeing at the same time that the *Fermat* node will be found. For this it uses the *candidate distance combinations*.

The same intuition is used in the *GIG* algorithm [12], only they use a suboptimal method to restrict the flooding. In addition, *GIG* cannot guarantee optimality since the distance from an external node to a datanode can be overestimated. This can be seen in Figure 3. According to *GIG* flooding is extended until all floods intersect, in this case node m. Then m broadcasts a message to every node inside the flooding union, which would be every node in this example, counting hops from m. This way the distance $|(x, d)|$ from a node x to a datanode d is calculated as $|(x, d)| = |(x, m)|+|(m, d)|$, which is clearly an overestimation. In our example the distance between node f and q is incorrectly estimated as $|(f, q)| = 5$ by *GIG* (following the solid edges) and correctly as $|(f, q)| = 4$ by *dFNS* (following the dashed edges). As a result the *GIG* algorithm would chose node m as the new operator node, although the actual *Fermat* node and optimal new operator node is f. The hosting costs estimated by *GIG* and our algorithm can also be seen in Figure 3.

There is a maximum radius that each datanode has to flood in order to be able to reach every candidate distance combi-

nation. The maximum radius is set to guarantee completeness, i.e. if there is an *external* node with hosting cost better that c_b of the best datanode it will be found. For the maximum radius of datanode d_i we use the maximum value of $\alpha_i \forall \alpha \in A$.

E. Flooding Speed

Increasing the likelihood that the floods will intersect at the *Fermat* node first, increases the likelihood that more nodes inside the flood intersection will receive a lower cost threshold. This in turn leads to fewer nodes reporting to the lead node as *Fermat* candidates, thus saving on communication cost. We define a primary speed for each flood in order for them to reach the *ideal* distance combination ϵ at the same time point.

After the floods reach the *ideal* distance combination ϵ they will keep expanding until they reach the maximum radius. We define a secondary speed for the foods that will make their intersection grow faster toward the distance combinations with the lower hosting costs.

The flooding speed is implemented by delaying the relay (broadcast) of the flooding message at every node. More specifically, a timeout is defined at flood initialization, that each node should obey before re-broadcasting the flooding message. This timeout is defined by multiplying the estimated time it takes for the message to travel over one hop by a delay factor. Based on the *ideal* distance combination ϵ, we compute the primary delay factor pf of the flood for each datanode $d_i \in D$ as follows:

$$pf(i) = max\{e_i\}_{\forall i}/\epsilon_i - 1$$

When the delay factor is 0 then the flooding message gets forwarded immediately.

To calculate the secondary delay factor sf we reverse the order of the pf. The datanode d_i with the maximum pf will have a secondary delay factor equal to the minimum pf. The datanode d_j with the minimum pf will have a secondary delay factor equal to the maximum pf and so on. The intuition about this is that the cheapest distance combinations will be the ones with minimum values satisfying the triangle inequality between the datanodes.

F. The dFNS *algorithm*

Here we describe the *dFNS* algorithm as general steps taken inside the network. Assume each operator node has some pre-specified criteria that decide whether an operator placement update is needed or not. These criteria could involve monitoring the change in the data loads of the datanodes, the change in the location of the datanodes, the remaining energy on the operator node, and estimations on whether an operator placement update would be worth the cost overhead for a cheaper query execution in the next epoch. What happens after this decision is taken is described next and shown in pseudocode in Algorithm 2.

Assume there is an operator placed on node h, that sends/receives data from datanodes D. Thus, it knows the data loads for every node in D. When the criteria of node h to update the operator are met, node h becomes the *leader* node and initiates the *dFNS* algorithm (Algorithm 2). It calculates all candidate distance combinations using Algorithm 1. If the candidate distance combination set is empty, the *leader* informs all datanodes that the new operator placement has changed to b. Otherwise, if there are candidate distance combinations for external nodes, the *leader* computes the time-point to initiate flooding for synchronization. Without synchronization variable speed flooding would not have the desired effect. The *leader* sends a message to all datanodes in D containing the time point to initiate the flooding and the candidate distance combinations.

Using the candidate distance combination set, datanode d_i can calculate the hosting cost c_b of the *best* datanode. It also can compute the minimum and maximum radius, and the primary and secondary speed of its flood, described in Section IV-E and IV-D respectively. d_i prepares a flooding message that contains the cost threshold set to c_b, the timeout needed to realize the primary speed, the timeout needed to realize the secondary speed, the minimum radius, and the maximum radius of the food. d_i initiates its flooding at the given time-point broadcasting its flooding message. All the candidate nodes send their report to the *leader*. After all reports are received, the *leader* calculates the best candidate node, informs the datanodes about the new operator host and passes on any information regarding the operator to the new host node.

Algorithm 2 . The general *dFNS* steps

Steps taken by *leader* node:

1: $A =$ CDCGenerator($\oslash, 0$)
2: **if** $A = \oslash$ **then**
3: Place operator on b
4: **else**
5: Set timepoint t for initiating flood
6: $m \leftarrow t, A$
7: send message m to D
8: **end if**
9: timeout until all candidate nodes have reported
10: choose best candidate as new operator host
11: inform datanodes about new operator host
12: send operator information to new operator host

Steps taken by each datanode $d_i \in D$:

1: compute *minimum* and *maximum* radius
2: compute *primary* and *secondary* speed
3: initiate flood at timepoint t

When an external node n receives a flooding message from datanode d_i for the first time it stores it and performs a series of checks. If n is not beyond the minimum radius then it just forwards the message. Any consecutive receptions of the same message are ignored. Otherwise, if n has received a message from all the datanodes in D it can calculate its hosting cost c_n. If c_n is smaller than the cost threshold contained in any of the flooding messages, node n updates the cost threshold inside every flooding message that was not yet forwarded and

stores c_n. Also, n sets a timeout to report to the *leader* node as a candidate node. A final check that node n performs when receiving a message from datanode d_i is whether it is not on the maximum radius so it can forward the message it received.

Every candidate node that has not reported yet to the *leader* performs eves-dropping on its neighbors. When such a candidate node receives a message containing a lower cost threshold than its hosting cost, it cancels the timeout for reporting to the *leader* node and withdraws its designation as a candidate node. This way the number of candidate node reports sent to the *leader* node are minimized. The pseudocode is omitted due to lack of space.

G. *Optimality of* dFNS

Our algorithm always finds the node in the network that minimizes the hosting cost as defined in objective function 1.

The dFNS *algorithm is optimal*

Proof: First, all possible distance combinations A, that have a better hosting cost than the *best* datanode, are found using Algorithm 1. This is true since the algorithm is exhaustive. The radius for the flood of datanode d_i is set to the maximum value of $\alpha_i \forall \alpha \in A$. This guarantees that all possible nodes with distance combinations equal to the ones in the candidate set A will be inside the intersection of all floods. This allows them to calculate their hosting cost and become candidate nodes.

V. Initial Operator Placement

Our distributed Fermat Node Search algorithm (*dFNS*) can be used in any framework that optimizes continuous queries, to always keep the operator placement optimal. In addition, frameworks that are made to optimize snapshot queries can be adapted for answering continuous queries by using *dFNS*. The query execution is divided in epochs. As soon as the query execution terminates, an operator node checks whether its operator meets the placement update criteria. Details of these criteria are orthogonal to this work. If those are met, *dFNS* is triggered and the operator placement is optimized before the next epoch. Note, that *dFNS* has no overhead whatsoever during query execution. The overhead is most of the time negligible even when an operator placement update takes place. We have a communication cost overhead only in the less frequent cases, where a flood is needed to find the new optimal operator host.

Most of the previously proposed algorithms for initial operator placement (Section II) are centralized, assuming global knowledge of the network. When answering snapshot queries, the initial placement should be as optimized as possible, which cannot be achieved without collecting network information. For continuous queries, however, the quality of the initial operator placement is less of an issue, as the query executes for several epochs. A rough initial placement is calculated with the information that is locally present or is collected locally without significant overhead, avoiding the collection of global network knowledge. After the query execution in the first epoch is over, we can check the criteria for each operator and, if they are met, run *dFNS* to optimize the operator placement for consecutive epochs.

We use the algorithm proposed in Chatzimilioudis et al [5] for finding an initial operator placement. We exploit the mandatory query dissemination to collect some information about the network with minimum overhead. Every node, that receives the query dissemination and has data needed for answering the query, sends to the querying node its position and a summary of its data. Techniques for building a summary of small size and high information have been previously proposed in the literature [17][18][19][20]. Using this information the query node can roughly estimate the hop-distance between the datanodes and the selectivity of each operator.

VI. Experimental Evaluation

The experiments were run on an Intel Core2 Duo CPU at 2.5GHz with a 4GB RAM. We implemented the algorithms in Java and used J-Sim [21] as our network simulator. We used the energy model of J-Sim to account for the energy spent by the network when transmitting data.

Comparison is done against the algorithm proposed by Zoe Abrams and Jie Liu in [12], noted as *GIG*. The authors' implementation was not available, so we reimplemented *GIG* with clarifications from the authors. This algorithm needs two parameters from the user in order to run: the radius α of the initial flood and a function $G()$ defining how the radius is increased for each consecutive flood. Its performance heavily depends on these parameters. For our experimental setup we use the optimal values $\alpha = 1$ and $G(r_{new}) = r_{previous} + 1$, which are the same as in the original paper. Those choices are optimal since most of the resulting networks have the datanodes in close proximity and only a small flooding radius of 1 or 2 hops is needed. We also implemented the variable speed flooding function that is only suggested as a future optimization in [12]. This function sets the flooding speed of datanode i to be inversely proportional to the data load of datanode i.

For the experiments we create a network with 512 nodes randomly scattered in a physical space of size 1000x1000. We randomly place the datanodes in a square region of size 200x200 at the center of the space. This is done so that the flooding process can reach a large number of nodes without hitting the edge of the network, and we get more accurate results regarding the efficiency of the algorithms. This is the same network setup as in [12].

We run experiments for m-ary operators with $m = 2, 3, 4$, thus we have $k = 3, 4, 5$ datanodes respectively, showing the efficiency of the compared algorithms. For each value of k we run 80 simulations. The amount of the communication cost overhead is immediately dependent on how far the datanodes are from each other, since the further apart the bigger the floods will have to be. Therefore, our experiments are grouped by metric h. We sum the distances from the datanodes to the *Fermat* node returned by the algorithm in the equation:

$$h = \sum_{\forall d_i \in D} |(d_i, f)|$$

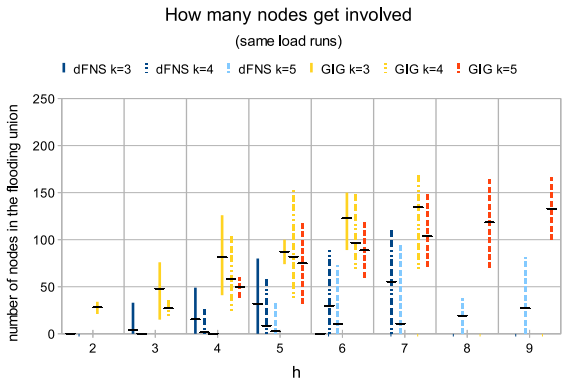


Fig. 4. Number of nodes involved in the flooding process (minimum, average and maximum value) using the same load for each datanode. When lines are missing it means there were no simulation runs possible for the combination of k and h values.

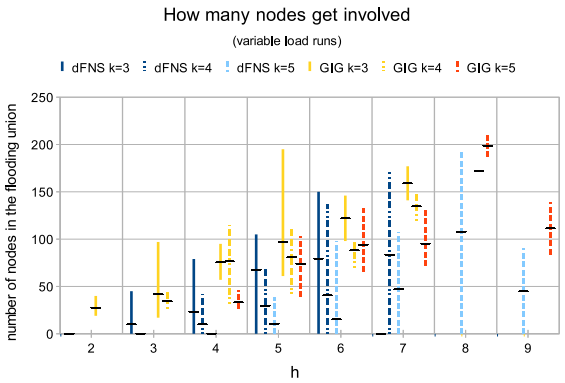


Fig. 5. Number of nodes involved in the flooding process (minimum, average and maximum value) using the variable load for each datanode. When lines are missing it means there were no simulation runs possible for the combination of k and h values.

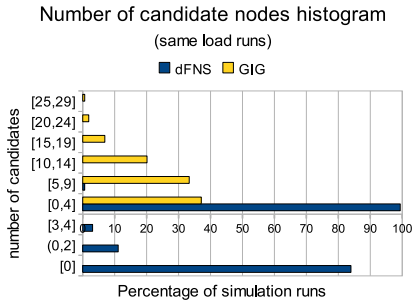


Fig. 6. Number of candidate nodes that report to the leader node. The less candidates the less communication needed. Beneath the x-axis the group [0,4] is divided into more detailed groups to show the distribution for *dFNS*.

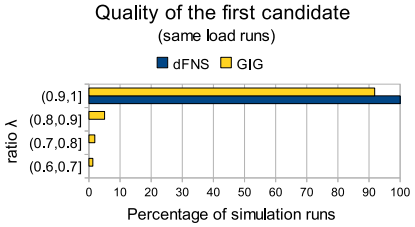


Fig. 7. Quality of the first candidate node encountered while flooding (or the best of the first set). Quality is expressed by dividing the hosting cost of the actual *Fermat* node to the first candidate. The closer the ratio is to 1 the better the variable speed flooding function of the algorithm used.

Reproducing Experiments Of GIG

For the first set of experiments we copied the operator migration experiments conducted in [12]. The authors used the same data load w for all the datanodes and used three metrics: number of nodes involved in the flooding process (Figure 4), number of candidate nodes (Figure 6) and quality of the first candidate (Figure 7). All figures denoted as "same load runs" belong to the first set of experiments. We got approximately the same results for the *GIG* algorithm as in [12]. Our proposed algorithm (*dFNS*) outperforms *GIG* in this first set of experiments.

In Figure 4 the minimum, average and maximum number of nodes involved in the flooding process is shown when running each algorithm for $k = 3, 4, 5$ datanodes. The simulation runs are grouped by h, how far apart the datanodes are. For each value of h the leftmost three lines belong to *dFNS*, whereas the rightmost three lines belong to *GIG*. Some lines are missing, since not all combinations of k and h are possible. For example, when we have $k = 5$ distinct datanodes it is impossible to find an operator node whose sum of distances to the datanodes is less than 4, $h < 4$. We can have $h = k - 1$ only if the *Fermat* node returned by the algorithm is one of the datanodes itself and every other datanode is only 1 hop away from the *Fermat* datanode.

One would expect that *GIG* always involves less nodes in its flooding than *dFNS*, since it stops flooding as soon as the floods intersect for the first time. As Figure 4 shows, though, *dFNS* has a far smaller mean value of the number of nodes involved in flooding than *GIG*. All this can be attributed to the fact that *dFNS* identifies the special cases where a datanode is the *Fermat* node, and avoids flooding. Those are the frequent cases where the number of nodes involved is zero.

For the second metric, we plot the number of candidate nodes produced by each algorithm in a histogram in Figure 6. The less candidate nodes, the fewer candidate node message have to be sent to the leader node to decide on the best candidate. We can see that *dFNS* has never more than 4 candidates. This happens because our algorithm looks for candidates only in the intersection of its extended floods, whereas *GIG* looks for candidates inside the whole union of its floods. Below the x-axis the group of [0-4] candidate is broken down just to show the distribution for our algorithm.

In Figure 7 we can see the quality of the first candidate. The quality is expressed by the ratio λ equal to the hosting cost of the actual *Fermat* node over the hosting cost of the first candidate node found. If $\lambda = 1$ it means the first candidate

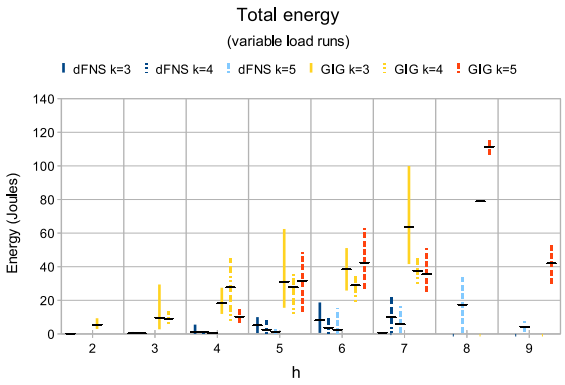


Fig. 8. Total energy consumed (minimum, average and maximum value). When lines are missing it means there were no simulation runs possible for the combination of k and h values.

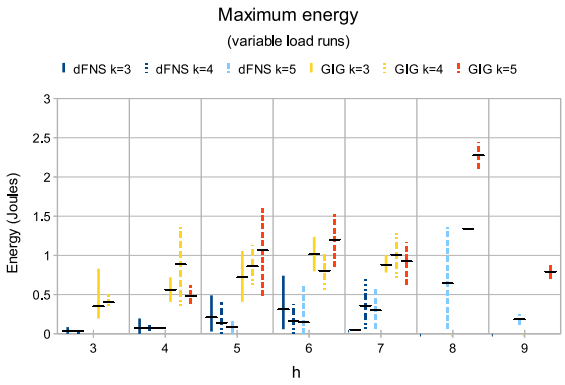


Fig. 9. The maximum energy consumed by a single node (minimum, average and maximum value). When lines are missing it means there were no simulation runs possible for the combination of k and h values.

node is the actual *Fermat* node. The first time all floods intersect, there are one or more candidate nodes inside the intersection, which will all report to the leader. The one with the best hosting cost is called the first candidate node. This is how the first candidate is defined for both algorithms. The quality of the first candidate depends solely on the variable speed flooding function used. We can see that our variable speed flooding function has always a better first candidate.

The simulations described so far were conducted solely for the purpose of matching the experiments in [12], in order to compare our algorithm head on against *GIG*. The second set of experiments is a fairer comparison between the two algorithms, since in real world applications the datanodes usually have different data loads.

We run all of the above experiments again with variable data loads. We varied the loads of the datanodes slightly with a Gaussian distribution around weight w used in the previous experiments. A greater variation in the loads would leave us with only a few runs where no datanode is a Fermat node, which is the only case where we can study these heuristics. The results regarding the first metric can be seen in Figure 5. We excluded the results for the other two metrics because the result were identical to the "same load" simulations seen in Figures 6 and 7.

Apart from better efficiency, *dFNS* also finishes faster since it does not use incremental flooding, where the network is flooded repeatedly until an intersection is found. *dFNS* floods once to a predefined restricted neighborhood.

Actual Communication Cost Overhead

We also conducted our own experiments using as metrics the total energy and the maximum energy per node consumed for finding the new *Fermat* node. After all, this is what we are trying to minimize with our algorithm. These are more important metrics compared to the above and the ones that actually define the performance of the algorithms.

Figures 8 and 9 show that *dFNS* clearly has a smaller energy overhead for determining the optimal hosting node. *GIG*'s limited cost flooding results in reflooding the neighborhood incrementally, thus yields a very big total energy cost. In addition it refloods the whole flooding union to look for candidate nodes. In our algorithm very often we do not need to flood in the first place. This keeps the mean value of total energy low. In the case where flooding is needed, our algorithm might use slightly larger flood radii, but it floods only once, and no further communication between nodes is needed to find any candidate nodes.

TABLE I
PERCENTAGE OF SIMULATION RUNS WHERE A DATANODE IS THE OPTIMAL NODE TO PLACE THE OPERATOR AND THUS NO FLOODING IS NEEDED

	$k=3$	$k=4$	$k=5$
same load	85%	84%	83%
variable load	68%	66%	56%

To simulate the energy in the previous experiments, we used the following parameters for our sensors: power consumption for transmission 0.660 Watts and power transmission for reception 0.395 Watts. The data rate of the radio is set to 19.2 kbps and the load of each transmission is 1Kb.

These differences are affected by the fact that *dFNS* takes advantage of the cases where a datanode is a *Fermat* node in order to save energy by avoiding flooding. We can see in Table I that the majority of cases have a datanode that is the actual *Fermat* node and thus we do not need to look any further for the optimal operator placement.

How Good Is dFNS *In Finding External* Fermat *Nodes*

It is clear that our algorithm successfully identifies the special cases where flooding can be avoided. Here we evaluate our algorithm for the other case by collecting information only from the simulation runs where the *Fermat* node is an external node and flooding is needed (*floody runs*). Figures 10 and 11 show that *dFNS* still outperforms *GIG*, although the savings are less significant compared to the cases where no flooding is needed. Figure 10 shows the minimum, mean and maximum

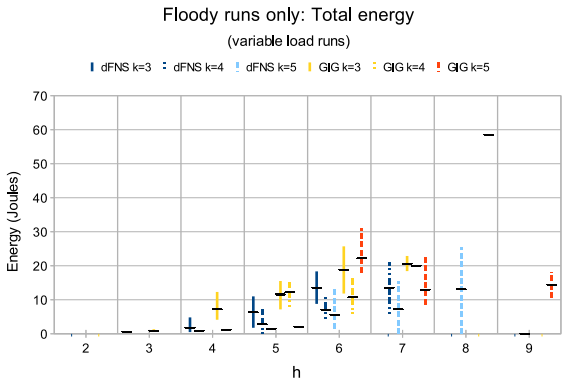


Fig. 10. Total energy consumed (minimum, average and maximum value) only for simulation runs that needed to use flooding in order to find the *Fermat* node. When lines are missing it means there were no simulation runs possible for the combination of k and h values.

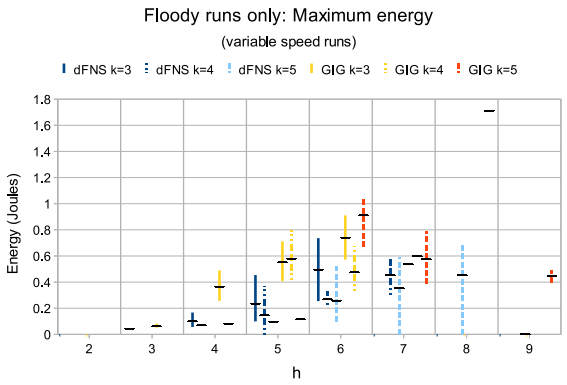


Fig. 11. The maximum energy consumed by a single node (min, avg and max value) only for simulation runs that needed to use flooding in order to find the *Fermat* node. When lines are missing it means there were no simulation runs possible for the combination of k and h values.

values of the total energy consumed for finding the external *Fermat* node. Similarly, Figure 11 shows the minimum, mean and maximum values of the maximum energy per node.

VII. Summarizing our contribution

We present an optimal distributed algorithm to update the placement of a single operator achieving minimum cost for executing continuous queries. Our algorithm imposes minimal communication cost overhead for finding the optimal node to host the operator. Previous work in WSN has only proposed approximate/heuristic algorithms. Besides the advantage of optimality, our experiments show that the cost overhead of our algorithm is reduced by $50\% - 100\%$ compared to previously proposed techniques. Our distributed Fermat Node Search algorithm (*dFNS*) can be used in any framework that optimizes continuous queries and has specific criteria for triggering operator placement updates. *dFNS* can be seamlessly integrated to keep the operator placement optimal.

Acknoledgments: This work was supported by the SemsorGrid4Env and the MODAP European Commission projects, the NSF IIS-0534781 grant, NSF Award 0627191 and HKU 714907E from Hong Kong RGC

References

[1] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *PODS '05: Proceedings of the twenty-fourth symposium on Principles of database systems*. New York, NY, USA: ACM, 2005, pp. 250–258.

[2] N. Jain, R. Biswas, N. Nandiraju, and D. Agrawal, "Energy aware routing for spatio-temporal queries in sensor networks," *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 3, pp. 1860–1866 Vol. 3, March 2005.

[3] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2006, p. 71.

[4] A. Pathak and V. K. Prasanna, "Energy-efficient task mapping for data-driven sensor network macroprogramming," in *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 516–524.

[5] G. Chatzimilioudis, H. Hakkoymaz, N. Mamoulis, and D. Gunopulos, "Operator placement for snapshot multi-predicate queries in wireless sensor networks," in *MDM 2009: Proceedings of the 10th International Conference on Mobile Data Management*, May 2009. [Online]. Available: http://www.cs.ucr.edu/ gchatzim/snapMPQ.pdf

[6] L. Ying, Z. Liu, D. F. Towsley, and C. H. Xia, "Distributed operator placement and data caching in large-scale sensor networks," in *INFOCOM*. IEEE, 2008, pp. 977–985.

[7] A. Deshpande, Z. G. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[8] K. Oikonomou, I. Stavrakakis, and A. Xydias, "Scalable service migration in general topologies," *A World of Wireless, Mobile and Multimedia Networks, International Symposium on*, vol. 0, pp. 1–6, 2008.

[9] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," *Telecommunication Systems*, vol. 26, no. 2-4, pp. 389–409, 2004.

[10] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer, "Network-aware operator placement for stream-processing systems," in *ICDE*, 2006, p. 49.

[11] R. L. F. Pitu B. Mirchandani, *Discrete Location Theory*. New York, NY, USA: Wiley, July 1990.

[12] Z. Abrams and J. Liu, "Greedy is good: On service tree placement for in-network stream processing," *Distributed Computing Systems, International Conference on*, vol. 0, p. 72, 2006.

[13] E. Weiszfeld, "Sur le point pour lequel la somme des distances de n points donnes est minimum," in *Tohoku Mathematics Journal*, vol. 43, 1937, p. 355386.

[14] I. Greenberg and R. A. Robertello, "The three factory problem," in *Mathematical Association of America*, 1965.

[15] A. Deshpande, C. Guestrin, and S. Madden, "Model-based querying in sensor networks," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 1764–1768.

[16] A. Deshpande, "Prdb: Managing large-scale correlated probabilistic databases (abstract)," in *SUM*, ser. Lecture Notes in Computer Science, L. Godo and A. Pugliese, Eds., vol. 5785. Springer, 2009, p. 1.

[17] C. Estan and J. Naughton, "End-biased samples for join cardinality estimation," *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pp. 20–20, April 2006.

[18] R. J. Lipton, J. F. Naughton, and D. A. Schneider, "Practical selectivity estimation through adaptive sampling," 1990, pp. 1–11.

[19] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias, "Spatio-temporal aggregation using sketches," in *ICDE*. IEEE Computer Society, 2004, pp. 214–226.

[20] B. H. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Comm. of the ACM*, vol. 13, no. 7, p. 422, July 1970.

[21] J. Kacer, "Discrete event simulations with j-sim," in *IRE '02: Proceedings of the second workshop on Intermediate Representation Engineering for virtual machines, 2002*, Maynooth, County Kildare, Ireland, 2002, pp. 13–18. [Online]. Available: http://j-sim.cs.uiuc.edu/