International Journal of Foundations of Computer Science
© World Scientific Publishing Company

# Resource Allocation Strategies for Constructive In-Network Stream Processing*

Anne Benoit and Veronika Rehn-Sonigo and Yves Robert
*École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France*
{*anne.benoit,veronika.sonigo,yves.robert*}*@ens-lyon.fr*
*http://graal.ens-lyon.fr/∼*{*abenoit,vsonigo,yrobert*}

Henri Casanova
*University of Hawai'i at Manoa, 1680 East-West Road, Honolulu, HI 96822, USA*
*henric@hawaii.edu*
*http://navet.ics.hawaii.edu/∼casanova/*

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

In this paper we consider the operator mapping problem for in-network stream processing applications. In-network stream processing consists in applying a tree of operators in steady-state to multiple data objects that are continually updated at various locations on a network. Examples of in-network stream processing include the processing of data in a sensor network, or of continuous queries on distributed relational databases. We study the operator mapping problem in a "constructive" scenario, i.e., a scenario in which one builds a platform dedicated to the application by purchasing processing servers with various costs and capabilities. The objective is to minimize the cost of the platform while ensuring that the application achieves a minimum steady-state throughput. The first contribution of this paper is the formalization of a set of relevant operator-placement problems, and a proof that even simple versions of the problem are NP-complete. Our second contribution is the design of several polynomial time heuristics, which are evaluated via extensive simulations and compared to theoretical bounds for optimal solutions.

*Keywords*: in-network stream processing, trees of operators, operator mapping, optimization, complexity results, polynomial heuristics.

## 1. Introduction

In this paper we consider the execution of applications structured as trees of operators. The leaves of the tree correspond to basic data objects that are spread over different servers in a distributed network. Each internal node in the tree denotes the

*Note that a short version of this paper appeared in the proceedings of APDCM'09, the 11th Workshop on Advances in Parallel and Distributed Computational Models, in conjunction with IPDPS 2009, Rome, Italy, May 2009. IEEE Computer Society Press.

aggregation and combination of the data from its children, which in turn generate new data that is used by the node's parent. The computation is complete when all operators have been applied up to the root node, thereby producing a final result. We consider the scenario in which the basic data objects are constantly being updated, meaning that the tree of operators must be applied continuously. The goal is to produce final results at some desired rate.

The above problem, which is called *stream processing* [1], arises in several domains. An important domain of application is the acquisition and refinement of data from a set of sensors [2]. For instance, a video surveillance application in which the sensors are cameras located at different locations over a geographical area is outlined in [2]. The goal of the application could be to show an operator monitored area in which there is significant motion between frames, particular lighting conditions, and correlations between the monitored areas. This can be achieved by applying several operators (filters, image processing algorithms) to the raw images, which are produced/updated periodically. Another example arises in the area of network monitoring [3, 4, 5]. In this case the sources of data are routers that produce streams of data pertaining to packets forwarded by the routers. One can often view stream processing as the execution of one or more "continuous queries" in the relational database sense of the term (e.g., a tree of join and select operators). A continuous query is applied continuously, i.e., at a reasonably fast rate, and returns results based on recent data generated by the data streams. Many authors have studied the execution of continuous queries on data streams [6, 7, 8].

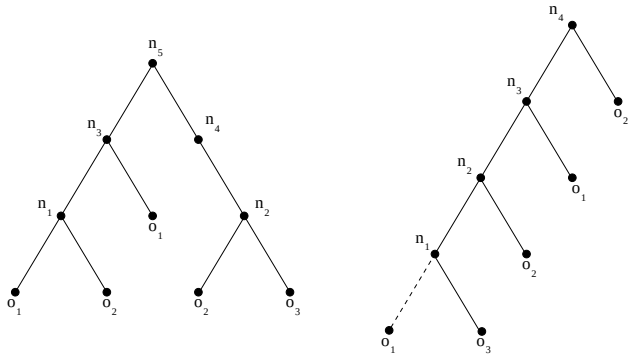
In practice, the execution of the operators on the data streams must be distributed over the network. In some cases, for instance in the aforementioned video surveillance application, the cameras that produce the basic objects do not have the computational capability to apply any operator effectively. Even if the servers responsible for the basic objects have sufficient capabilities, these objects must be combined across devices, thus requiring network communication. A simple solution is to send all basic objects to a central compute server, but it proves unscalable for many applications due to network bottlenecks. Also, this central server may not be able to meet the desired target rate for producing results due to the sheer amount of computation involved. The alternative is to distribute the execution by mapping each node in the operator tree to one or more compute servers in the network (which may be distinct or co-located with the devices that produce/store and update the basic objects). One then talks of *in-network* stream processing. Several in-network stream processing systems have been developed [4, 9, 10, 11]. These systems all face the same question: to which servers should one map which operators?

In this paper we address the operator-mapping problem for in-network stream processing. This problem was studied in [2, 12, 13]. The work in [13] studied the problem for an ad-hoc objective function that trades off application delay and network bandwidth consumption. In this paper we study a more general objective function. We first enforce the constraint that the rate at which final results are produced, or *throughput*, is above a given threshold. This corresponds to a Quality

of Service (QoS) requirement of the application, which is almost always desirable in practice (e.g., up-to-date results of continuous queries must be available at a given frequency). Basic objects may be replicated at multiple locations, i.e., available and updated at these locations. In terms of the computing platform we consider a "constructive" scenario: either the user can build the platform from scratch using off-the-shelf components, or computing and network units are rented by a cloud provider (e.g. [14]). In a recent paper [15], we have studied the operator-mapping problem in a "non-constructive" scenario, where, given a set of compute and network elements, we attempt to use as few resources as possible. Also, in [15], we tackle the problem for multiple concurrent applications that contend for the servers, each with its own QoS requirement. In this paper, our goal is to construct a distributed network dedicated to the given application, which minimizes the monetary cost while ensuring that the desired throughput is achieved. In terms of the tree of operators, we consider general binary trees and discuss relevant special cases (e.g., left-deep trees [16]).

Our contributions in this paper are as follows. First, we formalize a set of relevant operator-placement problems (Sections 2 and 3). Second, we establish complexity results, showing that all problems turn out to be NP-complete but that linear program problem formulations can obtained (Section 4). Third, we propose several common-sense heuristics for solving the operator mapping problem (Section 5). These heuristics are based on intuitive ideas combined with standard tree traversal techniques, and constitute a reasonable set of approaches for solving our NP-complete problem. Fourth, we evaluate results obtained with the heuristics and compare their results to a bound on the optimal that we obtain via the linear programming formulation of the problem (Section 6).

## 2. Models




a) Standard tree. b) Left−deep tree.

Fig. 1. Examples of applications structured as a binary tree of operators.

**Application model.** We consider an application that can be represented as a set of operators, $\mathcal{N} = \{n_1, n_2, \dots\}$. These operators are organized as a binary tree, as shown in Fig. 1. Operations are initially performed on basic objects, which are made available and continuously updated at given locations in a distributed network. We denote the set of basic objects $\mathcal{O} = \{o_1, o_2, \dots\}$. The leaves of the tree are thus the basic objects, and several leaves may correspond to the same object, as illustrated in Fig. 1. Internal nodes represent operator computations.

For an operator $n_i$ we define $Leaf(i)$ as the index set of the basic objects that are needed for the computation of $n_i$, if any; $Child(i)$ as the index set of the node's children in $\mathcal{N}$, if any; and $Parent(i)$ as the index of the node's parent in $\mathcal{N}$, if it exists. We have the constraint that $|Leaf(i)| + |Child(i)| \leq 2$ since our tree is binary. We define $f(I) = \cup_{i \in I} f(i)$, where $I$ is an index set and $f$ is $Leaf$, $Child$ or $Parent$. In other terms, all three functions can be applied to sets and return sets. If $|Leaf(i)| \geq 1$, then operator $n_i$ needs at least one basic object for its computation. We call such an operator an *al-operator* (for "almost leaf"). We suppose that each node needs the data from its children. As our trees are binary, we can model an operator $n_i$ that needs only one basic object or only one intermediate result (and no other result from any other node) by adding an "empty" leaf (i.e., with a zero-sized basic object) to this operator.

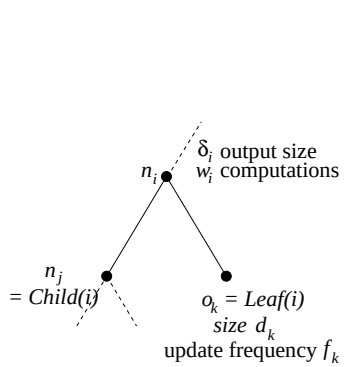
The application must be executed so that it produces final results, where each result is generated by executing the whole operator tree once, at a target rate. We call this rate the application *throughput* $\rho$ and the specification of the target throughput is a QoS requirement for the application. Each operator $n_i \in \mathcal{N}$ must compute (intermediate) results at a rate at least as high as the target application throughput. Conceptually, a server executing an operator consists of two concurrent threads that run in steady-state. One thread periodically downloads the most recent copies of the basic objects corresponding to the operator's leaf children, if any. For our example tree in Fig. 1(a), $n_1$ needs to download $o_1$ and $o_2$ while $n_3$ downloads only $o_1$ and $n_5$ does not download any basic object. Note that these downloads may simply amount to constant streaming of data from sources that generate data streams. Each download has a prescribed cost in terms of bandwidth based on application QoS requirements (e.g., so that computations are performed using sufficiently up-to-date data). A basic object $o_k$ has a size $\delta_k$ (in bytes) and needs to be downloaded by the processors that use it with frequency $f_k$. Therefore, these basic object downloads consume an amount of bandwidth equal to $rate_k = \delta_k \times f_k$ on each network link and network card through which this object is communicated. The other thread receives data from the operator's non-leaf children, if any, and performs some computation using downloaded basic objects and/or data received from other operators. The operator produces some output that needs to be passed to its parent operator. The computation of one instance of operator $n_i$ requires $w_i$ operations (e.g., floating point operations), and produces an output of size $\delta_i$.

In this paper we sometimes consider *left-deep* trees, i.e., binary trees in which the right child of an operator is always a leaf. These trees arise in practical settings [16]

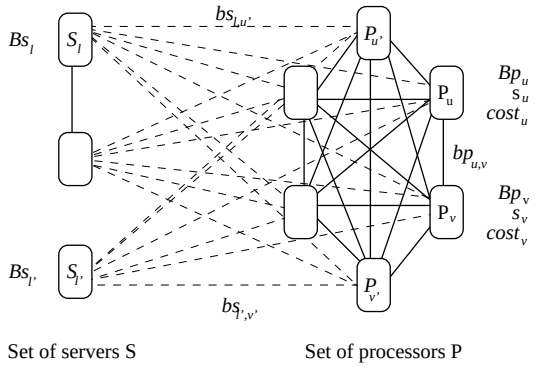
and we show an example of left-deep tree in Fig. 1(b). Here $Child(i)$ and $Leaf(i)$ have cardinal 1 for every operator $n_i$ but for the bottom-most operator, $n_j$, for which $Child(j)$ has cardinal 0, and $Leaf(j)$ has cardinal 1 or 2 depending on the application. Our application model is depicted in Fig. 2(a).

**Platform model.** The target distributed network is a fully connected graph (i.e., a clique) interconnecting a set of resources $\mathcal{R} = \mathcal{P} \cup \mathcal{S}$, where $\mathcal{P}$ denotes compute servers, or *processors* for short, and $\mathcal{S}$ denotes data servers, or *servers* for short. Servers hold and update basic objects, while processors apply operators of the application tree. Each server $S_l \in \mathcal{S}$ (resp. processor $P_u \in \mathcal{P}$) is interconnected to the network via a network card with maximum bandwidth $Bs_l$ (resp. $Bp_u$). The network link from a server $S_l$ to a processor $P_u$ has bandwidth $bs_{l,u}$; on such links the server sends data and the processor receives it. The link between two distinct processors $P_u$ and $P_v$ is bidirectional with bandwidth $bp_{u,v} (= bp_{v,u})$ shared by communications in both directions. In addition, each processor $P_u \in \mathcal{P}$ is characterized by a compute speed $s_u$ measured in operations per time units.

Resources operate under the full-overlap, bounded multi-port model [17]. In this model, a resource $R_u$ can be involved in computing, sending data, and receiving data simultaneously. Note that servers only send data, while processors engage in all three activities. A resource $R$, which is either a server or a processor, can be connected to multiple network links (since we assume a clique network). The "multi-port" assumption states that $R$ can send/receive data simultaneously on multiple network links. The "bounded" assumption states that the total transfer rate of data sent/received by resource $R$ is bounded by its network card bandwidth ($Bs_l$ for server $S_l$, or $Bp_u$ for processor $P_u$). Our platform model is depicted in Fig. 2(b).




(a) Notations in the application tree.




(b) The platform model. The set of servers $S$ is fixed, whereas the set of processors $P$ has to be determined.

Fig. 2. Notations and illustration of the application and the platform model.

**Mapping model and constraints.** Our objective is to map operators, i.e., internal nodes of the application tree, onto processors. Each operator is mapped onto a single processor, which computes all the results for that operator. While this rule prevents multiple processors to share different computations related to the same operator [18], it leads to an easier schedule construction and implementation. Note however that different operators can be assigned to the same processor, in order to optimize resource utilization. As explained in the application model section, if a tree node has leaf children it must continuously download up-to-date basic objects, which consumes bandwidth on its processor's network card. If there is only one operator on processor $P_u$, while the processor computes for the $t$-th final result it sends to its parent (if any) the data corresponding to intermediate results for the $(t-1)$-th final result. It also receives data from its non-leaf children (if any) for computing the $(t+1)$-th final result. All three activities are concurrent (see description of the platform model). When different operators are assigned to the same processor, the same overlap happens, but possibly on different result instances. The time required by each activity must be summed for all operators to determine the processor's computation time.

We assume that a basic object can be duplicated, and thus be available and updated at multiple servers. We assume that duplication of basic objects is achieved in some out-of-band manner specific to the target application. For instance, this could be achieved via the use of a distributed database infrastructure that enforces consistent data replication. In this case, a processor can choose among multiple data sources when downloading a basic object. Conversely, if two operators have the same basic object as a leaf child and are mapped to different processors, they must both continuously download that object (and incur the corresponding network overheads).

We denote the mapping of the operators in $\mathcal{N}$ onto the processors in $\mathcal{P}$ using an allocation function $a$: $a(i) = u$ if operator $n_i$ is assigned to processor $P_u$. Conversely, $\bar{a}(u)$ is the index set of operators mapped on $P_u$: $\bar{a}(u) = \{i \mid a(i) = u\}$.

We also introduce new notations to describe the location of basic objects. Processor $P_u$ may need to download some basic objects from some servers. We use $download(u)$ to denote the set of $(k, l)$ couples where object $o_k$ is downloaded from server $S_l$ by processor $P_u$.

Given these notations we can now express the constraints for the required application throughput, $\rho$. Essentially, each processor has to communicate and compute fast enough to achieve this throughput, which is expressed via a set of constraints. Note that a communication occurs only when a child or the parent of a given tree node and this node are mapped on different processors. In other terms, we neglect intra-processor communications.

• Each processor $P_u$ cannot exceed its computation capability:

$$\forall P_u \in \mathcal{P}, \quad \sum_{i \in \bar{a}(u)} \rho \cdot \frac{w_i}{s_u} \leq 1 \tag{1}$$

• $P_u$ must have enough bandwidth capacity to perform all its basic object downloads and all communication with other processors. This is expressed by the following constraint, in which the first term corresponds to basic object downloads, the second term corresponds to inter-node communications when a node is assigned to $P_u$ and some of its children nodes are assigned to another processor, and the third term corresponds to inter-node communications when a tree node is assigned to $P_u$ and its parent node is assigned to another processor:

$$\forall P_u \in \mathcal{P}, \sum_{(k,l)\in download(u)} rate_k + \sum_{j\in Child(\bar{a}(u))\setminus\bar{a}(u)} \rho.\delta_j + \sum_{j\in Parent(\bar{a}(u))\setminus\bar{a}(u)} \sum_{i\in Child(j)\cap\bar{a}(u)} \rho.\delta_i \leq Bp_u \quad (2)$$

• Server $S_l$ must have enough bandwidth capacity to support all the downloads of the basic objects it holds at their required rates:

$$\forall S_l \in \mathcal{S}, \sum_{P_u\in\mathcal{P}} \sum_{(k,l)\in download(u)} rate_k \leq Bs_l \quad (3)$$

• The link between server $S_l$ and processor $P_u$ must have enough bandwidth capacity to support all possible object downloads from $S_l$ to $P_u$ at the required rate:

$$\forall P_u \in \mathcal{P}, \forall S_l \in \mathcal{S}, \sum_{(k,l)\in download(u)} rate_k \leq bs_{l,u} \quad (4)$$

• The link between processor $P_u$ and processor $P_v$ must have enough bandwidth capacity to support all possible communications between the tree nodes mapped on both processors. This constraint can be written similarly to constraint (2) above, but without the cost of basic object downloads, and with specifying that $P_u$ communicates with $P_v$:

$$\forall P_u, P_v \in \mathcal{P}, \sum_{\substack{j\in Child(\bar{a}(u))\\ \cap\bar{a}(v)}} \rho.\delta_j + \sum_{\substack{j\in Parent(\bar{a}(u))\\ \cap\bar{a}(v)}} \sum_{\substack{i\in Child(j)\\ \cap\bar{a}(u)}} \rho.\delta_i \leq bp_{u,v} \quad (5)$$

Note that in our model we do not allow for a processor to serve as a repository for basic objects, i.e., forwarding updated basic objects to other processors. Instead, all basic objects must be obtained from authoritative sources, i.e., the servers themselves. It would be straightforward to extend our model by allowing a "virtual" server to be co-located with a processor. That server would share network card and network link with the processor so that the processor's network resources would be used by other processors downloading the basic object from the virtual server. Each processor would then have an attached such virtual server. The above equations could be rewritten taking such virtual servers into account, at the cost of more cumbersome notations.

## 3. Problem Definitions

The overall objective of the operator-mapping problem is to ensure that a prescribed throughput is achieved while minimizing a cost function. The user must buy or rent processors (with various computing speed and network card bandwidth specifications) and build the distributed network dedicated to the application. For this "constructive" problem, the cost function is simply the actual monetary cost of the purchased processors. This problem is relevant to, for instance, the surveillance application mentioned in Section 1.

We assume that some standard interconnect technology is used to connect all the processors together ($bp_{u,v} = bp$). We also assume that the same interconnect technology is used to connect each server to processors ($bs_{l,u} = bs_l$). We consider the case in which the processors are homogeneous because only one type of CPUs and network cards can be purchased ($Bp_u = Bp$ and $s_u = s$). We term the corresponding problem Hom. We also consider the case in which the processors are heterogeneous with various compute speeds and network card bandwidth, which we term LAN.

Homogeneity in the platform as described above applies only to processors and not to servers. Servers are always fixed for a given application, together with the objects they hold. We sometimes consider variants of the problem in which the servers and application tree have particular characteristics. We denote by HomS the case in which all servers have identical network capability ($Bs_l = Bs$) and communication links to processors ($bs_{l,u} = bs$). We can also consider the mapping of particular trees, such as left-deep trees (LDTree) and/or homogeneous trees with identical object rates $rate_k = rate$ and number of operations $w_i = w$ (HomA). Also, we can consider application trees with no communication cost ($\delta_i = 0$, NoComA). All these variants correspond to simplifications of the problem, and we simply append HomS, LDTree, HomA, and/or NoComA to the problem name to denote these simplifications.

Another possible scenario is to tackle the operator-mapping problem in a "non-constructive" way, where an existing platform is targeted. The goal is then to use a subset of this platform so that the prescribed throughput is achieved while minimizing a cost function. This scenario is not subject of this paper, but the interested reader may see [15] and [19].

## 4. Complexity

Without surprise, most problem instances are NP-hard, because downloading objects with different rates on two identical servers is the same problem as 2-Partition [20]. But from a theoretical point of view, it is important to assess the complexity of the simplest instance of the problem, i.e., mapping a fully homogeneous left-deep tree application with objects placed on a fully homogeneous set of servers, onto a fully homogeneous set of processors: Hom-HomS-LDTree-HomA-NoComA (or LDT-Hom for short). It turns out that even this problem is difficult, due to the combinatorial space induced by the mapping of basic objects that are

shared by several operators. Note that the corresponding non-constructive problem is exactly the same, see [15].

**Definition 1.** *The problem* LDT-HOM *consists in minimizing the number of processors used in the application execution. $K$ is the prescribed throughput that should not be violated.* LDT-HOM-DEC *is the associated decision problem: given a number of processors $N$, is there a mapping that achieves throughput $K$?*

**Theorem 2.** LDT-HOM-DEC *is NP-complete.*

**Proof.** First, LDT-HOM-DEC belongs to NP. Indeed, given an allocation of operators to processors and the download list $download(u)$ for each processor $P_u$, we can check in polynomial time that we use no more than $N$ processors, that the throughput of each enrolled processor respects $K$: $K \times |\bar{a}(u)| \frac{w}{s} \leq 1$ , and that bandwidth constraints are respected.

To establish the completeness, we use a reduction from 3-Partition, which is NP-complete in the strong sense [20]. We consider an arbitrary instance $\mathfrak{I}_1$ of 3-Partition: given $3n$ positive integer numbers $\{a_1, a_2, \ldots, a_{3n}\}$ and a bound $R$, assuming that $\frac{R}{4} < a_i < \frac{R}{2}$ for all $i$ and that $\sum_{i=1}^{3n} a_i = nR$, is there a partition of these numbers into $n$ subsets $I_1, I_2, \ldots, I_n$ of sum $R$? In other words, are there $n$ subsets $I_1, I_2, \ldots, I_n$ such that $I_1 \cup I_2 \ldots \cup I_n = \{a_1, a_2, \ldots, a_{3n}\}$, $I_i \cap I_j = \emptyset$ if $i \neq j$, and $\sum_{j \in I_i} a_j = R$ for all $i$ (and $|I_i| = 3$ for all $i$). Because 3-Partition is NP-complete in the strong sense, we can encode the $3n$ numbers in unary and assume that the size of $\mathfrak{I}_1$ is $O(nM)$, where $M = \max_i\{a_i\}$.

We build the following instance $\mathfrak{I}_2$ of LDT-HOM-DEC:

• The object set is $\mathcal{O} = \{o_1, ..., o_{3n}\}$, and there are $3n$ servers each holding an object, thus $o_i$ is available on server $S_i$. The rate of $o_i$ is $rate = 1$ , and the bandwidth limit of the servers is set to $Bs = 1$.

• The left-deep tree consists of $|\mathcal{N}| = nR$ operators with $w = 1$. Each object $o_i$ appears $a_i$ times in the tree (the exact location does not matter), so that there are $|\mathcal{N}|$ leaves in the tree, each associated to a single operator of the tree.

• The platform consists of $n$ processors of speed $s = 1$ and bandwidth $Bp = 3$. All the link bandwidths interconnecting servers and processors are equal to $bs = bp = 1$.

• Finally we ask whether there exists a solution matching the bounds $1/K = R$ and $N = n$.

The size of $\mathfrak{I}_2$ is clearly polynomial in the size of $\mathfrak{I}_1$, since the size of the tree is bounded by $3nM$. We now show that instance $\mathfrak{I}_1$ has a solution if and only if instance $\mathfrak{I}_2$ does.

Suppose first that $\mathfrak{I}_1$ has a solution. We map all operators corresponding to occurrences of object $o_j$, $j \in I_i$, onto processor $P_i$. Each processor receives three distinct objects, each coming from a different server, hence bandwidths constraints are satisfied. Moreover, the number of operators computed by $P_i$ is equal to $\sum_{j \in I_i} a_i = R$, and the required throughput is achieved because $KR \leq 1$. We have thus built a solution to $\mathfrak{I}_2$.

Suppose now that $\mathfrak{I}_2$ has a solution, i.e., a mapping matching the bound $1/K = R$ with $n$ processors. Due to bandwidth constraints, each of the $n$ processors is assigned at most three distinct objects. Conversely, each object must be assigned to at least one processor and there are $3n$ objects, so each processor is assigned exactly three objects in the solution, and no object is sent to two distinct processors. Hence, a processor must compute all operators corresponding to the objects it needs to download, which directly leads to a solution of $\mathfrak{I}_1$ and concludes the proof. □

Note that problem LDT-Hom-Dec becomes polynomial if one adds the additional restriction that no basic object is used by more than one operator in the tree. In this case, one can simply assign operators to $\lceil |\mathcal{N}| \times w/s \rceil$ arbitrary processors in a round-robin fashion.

*Linear Programming Formulation:* We also provide a formulation of the optimization problem as an integer linear program (ILP), but due to lack of space we refer the interested reader to [19].

## 5. Heuristics

In this section we propose several polynomial heuristics to solve the most realistic LAN operator-placement problem. The code for all of them is available on the web [21]. We say that the heuristics can "purchase" processors, or "sell back" processors, until a final set of needed processors is determined.

Each heuristic works in two steps: (i) an operator placement heuristic determines the number of processors that should be acquired, and decides which operators are assigned to which processors; (ii) a server selection heuristic decides from which server each processor downloads all needed basic objects.

**Operator placement heuristics.** Note that in most of these heuristics, only the most powerful processors and network cards are acquired. However, these are later replaced by the cheapest ones that still fulfill throughput requirements. This is done just after the server selection step, as a third "downgrade" step, aiming at minimizing the cost.

*Random:* While there are some unassigned operators, the Random heuristic picks one of these unassigned operators randomly, say *op*. It then acquires the cheapest possible processor that is able to handle *op* while achieving the required application throughput. If there is no such processor, then the heuristic considers *op* along with one of its children operators or with its parent operator. This second operator is chosen so that it has the most demanding communication requirements with *op* (in an attempt to reduce communication overhead). If no processor can be acquired that can handle both operators together, then the heuristic fails. If the additional operator had already been assigned to another processor, this last processor is sold back.

*Comp-Greedy:* The Comp-Greedy heuristic first sorts operators in non-increasing order of $w_i$, i.e., most computationally demanding operators first. While there are unassigned operators, the heuristic acquires the most expensive processor available and assigns the most computationally demanding unassigned operator to it. If this operator cannot be processed on this processor so that the required throughput is achieved, then the heuristic uses a grouping technique similar to that used by the Random heuristic (i.e., grouping the operator with its child or parent operator with which it has the most demanding communication requirement). If after this step some capacity is left on the processor, then the heuristic tries to assign other operators to it. These operators are picked in non-increasing order of $w_i$, i.e., trying to first assign to this processor the most computationally demanding operator.

*Comm-Greedy:* The Comm-Greedy heuristic attempts to group operators to reduce communication costs. It picks the two operators that have the largest communication requirements. These two operators are grouped and assigned to the same processor, thus saving costly communication between both processors. There are three cases to consider: (i) both operators were unassigned, in which case the heuristic simply acquires the cheapest processor that can handle both operators; if no such processor is available then the heuristic acquires the most expensive processor for each operator; (ii) one of the operators was already assigned to a processor, in which case the heuristic attempts to accommodate the other operator as well; if this is not possible then the heuristic acquires the most expensive processor for the other operator; (iii) both operators were already assigned on two different processors, in which case the heuristic attempts to accommodate both operators on one processor and sell back the other processor; if this is not possible then the current operator assignment is not changed.

*Subtree-Bottom-Up:* This heuristic (Subtree-BU in the following) first acquires as many most expensive processors as there are al-operators and assigns each al-operator to a distinct processor. The heuristic then tries to merge the operators with their parent on a single machine, in a bottom-up fashion (possibly returning some processors). Consider a processor on which one or more operators have been assigned. The heuristic first tries to allocate as many parent operators of the currently assigned operators to this processor. If some parent operators cannot be assigned to this processor, then one or more new processors are acquired. This mechanism is used until all operators have been assigned to processors.

*Object-Grouping:* For each basic object, this heuristic counts how many operators need this basic object. This count is called the "popularity" of the basic object. The al-operators are then sorted by non-increasing sum of the popularities of the basic objects they need. The heuristic starts by acquiring the most expensive processor and assigns to it the first al-operator. The heuristic then attempts to assign to it as many other al-operators that require the same basic objects as the first al-operator, taken in order of non-increasing popularity, and then as many non al-operators as possible. This process is repeated until all operators have been assigned.

*Object-Availability:* This heuristic takes into account the distribution of basic objects on the servers. For each object $o_k$, the number $av_k$ of servers handling $o_k$ is calculated. Al-operators in turn are treated in increasing order of $av_k$ of the basic objects they need to download. The heuristic tries to assign as many al-operators downloading object $o_k$ as possible on a most expensive processor. The remaining internal operators are assigned similarly to Comp-Greedy, i.e., in decreasing order of $w_i$ of the operators.

**Server selection heuristics.** Once an operator placement heuristic has been applied, each al-operator is mapped on a processor, which needs to download basic objects required by the operators. Thus, we need to specify from which server this download should occur. For the Random heuristic, once the mapping of operators onto processors is fixed, we associate randomly a server to each basic object a processor has to download.

For all other heuristics, we use a more sophisticated heuristic, using three loops. The first loop assigns objects that are held exclusively by a single server. If not all downloads can be guaranteed, the heuristic fails. The second loop associates as many downloads as possible to servers that provide only one basic object type. The last loop finally tries to assign the remaining basic objects that must be downloaded. For this purpose, objects are treated in decreasing order of $nbP/nbS$, where $nbP$ is the remaining number of processors that need to download the object, and $nbS$ is the number of servers where the object still can be downloaded. In the decision process, servers are considered in decreasing order of the minimum between the remaining bandwidth capacity of the servers network card, and the bandwidth of the communication link.

Once servers have been selected, processors are downgraded if possible: each processor is replaced by a less expensive model that fulfills the CPU and network card requirements of the allocation.

## 6. Simulation Results

**Simulation methodology.** In the literature, to the best of our knowlege, there are no precise models of query streaming applications that would allow us to conduct a narrow, but representative, experimental study. However, we note that applications can be constructed based on arbitrary components. Consequently, we opted for using randomly generated applications with sets of parameters that span a reasonably large range of possible configurations. By contrast, we simulate the use of hardware resources that correspond to real-world configurations and prices.

All our simulations use randomly generated binary operator trees with at most $N$ operators, which we vary. All leaves correspond to basic objects, and each basic object is chosen randomly among 15 different types. For each of these 15 basic object types, we randomly choose a fixed size. In simulations, *small object sizes* are in the $\delta_k \in [5, 30]$ MB range, whereas *large object sizes* are in the $\delta_k \in [450, 530]$

MB range. The download frequency for basic objects is either *low* ($f_k = 1/50s$) or *high* ($f_k = 1/2s$). Recall that the download rate for object $o_k$ is then computed as $rate_k = \delta_k \times f_k$.

The number of operations $w_i$ of an operator $n_i$ (a non-leaf node in the tree) depends on its children $l$ and $r$ (basic object or operator): $w_i = (\delta_l + \delta_r)^\alpha$, where $\alpha$ is a constant fixed for each simulation run, and $\delta$ is either the size of the basic object, or the amount of data sent by the child operator. A small value of $\alpha$ corresponds to data-intensive operators while a large value corresponds to compute-intensive operators. Although in the real world each operator would have a different $\alpha$ value, in our simulation we use the same value for all operators. We opt for this simple model because it makes it straightforward to study the impact of data- and compute-intensiveness on our heuristics and because we are not aware of any models of representative operator complexities for query streaming application in the literature. The same principle is used for the output size of each operator, setting for all simulations $\delta_i = \delta_l + \delta_r$. The application throughput $\rho$ is fixed to 1 for all simulations. Throughout the whole set of simulations we use the same server architecture: we dispose of 6 servers, each of them equipped with a 10 GB network card. The 15 different types of objects are randomly distributed over the 6 servers. We assume that servers and processors are all interconnected by a 1 GB link. The rest of the platform can be purchased at the costs from Table 1 (configurations of Intel's high-end, rack-mountable server, PowerEdge R900). Note that we do not account for the cost of the network switch used to interconnect the processors and servers. In case the switch needs to be purchased as well, then its cost could be factored into the network card costs. Noting that more expensive switches may be needed for higher-capacity network cards, high-end network cards may end up being significantly more expensive, which would change our simulation results.

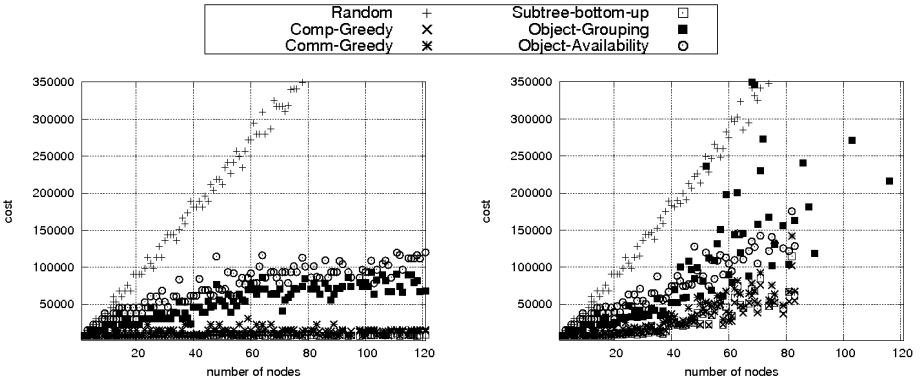
Table 1. Incremental costs for increases in processor performance or network card bandwidth relative to a $7,548 base configuration (based on data from the Dell Inc. web site, as of early March 2008).

| Processor | | | Network Card | | |
|---|---|---|---|---|---|
| Performance (GHz) | Cost ($) | Ratio (GHz/$) | Bandwidth (Gbps) | Cost ($) | Ratio (Gbps/$) |
| 11.72 | 7,548 + 0 | $1.55 \times 10^{-3}$ | 1 | 7,548 + 0 | $1.32 \times 10^{-4}$ |
| 19.20 | 7,548 + 1,550 | $1.93 \times 10^{-3}$ | 2 | 7,548 + 399 | $2.51 \times 10^{-4}$ |
| 25.60 | 7,548 + 2,399 | $2.38 \times 10^{-3}$ | 4 | 7,548 + 1,197 | $4.57 \times 10^{-4}$ |
| 38.40 | 7,548 + 3,949 | $3.12 \times 10^{-3}$ | 10 | 7,548 + 2,800 | $9.66 \times 10^{-4}$ |
| 46.88 | 7,548 + 5,299 | $3.43 \times 10^{-3}$ | 20 | 7,548 + 5,999 | $14.76 \times 10^{-4}$ |

**Results.** Due to lack of space, we only present results for selected sets of significant experiments (see [19] for more results).

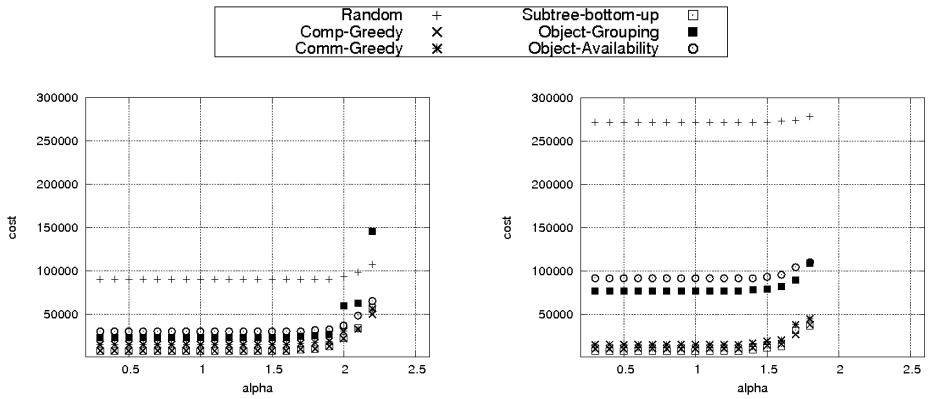
In the first set of simulations, we study the behavior of the heuristics when the download frequency is high (1/2s) and object sizes small (5-30MB). Fig. 3 shows the cost as the number of nodes $N$ in the tree increases, with a fixed computation factor $\alpha$. As expected, Random performs poorly. Subtree-BU achieves the best costs. All Greedy heuristics exhibit similar performance, poorer than Subtree-BU. Perhaps surprisingly, the heuristics that pay special attention to basic objects, Object-Grouping and Object-Availability, perform poorly. With a larger value of $\alpha$ (cf. Fig. 3(b)) the operator tree size becomes a more limiting factor. For trees with more than 80 operators, almost no feasible mapping can be found. However, the relative performance of our heuristics remains almost the same, with two notable features: a) Object-Grouping still finds some mappings for operator trees with up to 120 operators; b) Comp-Greedy performs as well as and sometimes better than Subtree-BU when the number of operators increases.

In the second set of simulations, we keep a high download frequency and small object sizes, but we rather explore the behavior of the heuristics when $N$ is fixed and the computation factor $\alpha$ increases, see Fig. 4. Up to a threshold, the $\alpha$ parameter has no influence on the heuristics' performance. When $\alpha$ reaches the threshold, the solution cost of each heuristic increases until $\alpha$ exceeds a second threshold after which solutions can no longer be found. Depending on the number of operators both thresholds have lower or higher values. In the case of small operator trees with only 20 nodes, (see Fig. 4(a)), the first threshold is for $\alpha$=1.7 and the second at $\alpha$=2.2 (vs. $\alpha$=1.6 and $\alpha$=1.8 for operator trees of size 60, as seen in Fig. 4(b)). Subtree-BU behaves in both cases the best, whereas Random performs the poorest. Object-Grouping and Object-Availability change their position in the ranking: for small trees Object-Grouping behaves better, for larger trees it is outperformed by Object-Availability. The Greedy heuristics are between Subtree-BU and the object sensitive heuristics.




(a) $\alpha = 0.9$. (b) $\alpha = 1.7$.

Fig. 3. Simulations with high frequency and small object sizes, increasing $N$.




(a) $N = 20$. (b) $N = 60$.

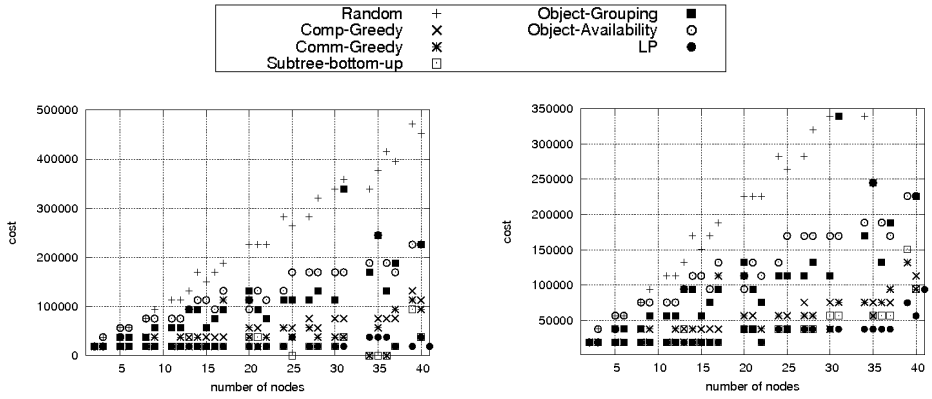
Fig. 4. Simulations with high frequency and small object sizes, increasing $\alpha$.

With the same experimental setting but large object sizes (450-530MB), the results are similar except that no feasible solution can be found as soon as the trees exceed 45 nodes. In general, Subtree-BU still achieves the best costs, but at times it is outperformed by Comm-Greedy. Subtree-BU even fails in two cases (the server selection does not succeed because of bandwidth limitation), while other heuristics find a solution. Please refer to [19] for the detailed results.

The behaviors of the heuristics with low download frequencies ($f_k = 1/50s$) are almost the same as for high frequency. In general the heuristics lead to the same operator mapping, but in some cases the purchased processors have less powerful network cards.

In another set of experiments, we study the influence of download rates on the solution. Recall that the download rate of a basic object $o_k$ is computed by $rate_k = f_k \times \delta_k$. A first result is that frequencies smaller than $1/10s$ have no further influence on the solution. All heuristics find the same solutions for a fixed operator tree (see figures in [19]). For frequencies between $1/2s$ and $1/10s$, the solution cost changes. In general the cost decreases, but for $N = 160$ the cost for the Object-Grouping heuristic increases. Furthermore, the heuristic ranking remains: Subtree-BU, followed by the Greedy heuristics, followed by the object sensitive ones, and Random. Interestingly, the costs of Object-Availability decrease with the number of operators. In this case the number of operators that need to download a basic object increases, and hence the privileged treatment of basic objects in order of availability on servers becomes more important. We conclude that the level of replication of basic objects on servers may matter for application trees with specific structures and download frequencies, but that in general we can consider that this parameter has little or no effect on the heuristics' performance.

The last set of experiments is dedicated to the evaluation of our heuristics versus




(a) $\alpha = 0.9$. (b) $\alpha = 1.1$.

Fig. 5. Simulations to compare the heuristics' performances to the LP performance on homogeneous platforms.

a lower bound given by the solution of our ILP. We use the commercial Cplex 11 solver to solve our linear program. Unfortunately, the ILP is so enormous that, even when using only 5 possible groups of processors and using trees with 30 operators, the ILP description file could not be opened in Cplex. For trees with 20 operators, Cplex returns the optimal solution, which consists in all cases in buying a single processor. Therefore, we decided to compare the heuristic solution with the optimal solution only in a homogeneous setting, in which there is only a single processor type. In this case we can skip the downgrading step after the server allocation step. Both for $\alpha$ values lower and higher than 1, Subtree-BU finds the optimal solution in most of the cases (see Figs. 5(a) and 5(b)).

The same ranking of the heuristics holds in the homogeneous setting: Subtree-bottom up, the Greedy heuristics, followed by Object-Grouping, Object-Availability and finally Random. Focusing on the Greedy heuristics, we observe that in most cases Comm-Greedy achieves the best cost.

**Summary of results.** Results show that all our more sophisticated heuristics perform better than the simple random approach. Unfortunately, the object sensitive heuristics, Object-Grouping and Object-Availability, do not show the desired performance. We believe that in some situations these heuristics could lead to good performance, but this is not observed on our set of random application configurations. We have found that Subtree-BU outperforms other heuristics in most situations and also produces results very close to the optimal (for the cases in which we were able to determine the optimal). There are some cases for which Subtree-BU fails. In such cases our results suggest that one should use one of our Greedy heuristics.

## 7. Conclusion

In this paper we have studied the problem of resource allocation for in-network stream processing. We formalized several operator-placement problems. We have focused more particularly on a "constructive" scenario in which one aims at minimizing the cost of a platform that satisfies an application throughput requirement. The complexity analysis showed that all problems are NP-complete, even for the simpler cases. We have derived an integer linear programming formulation of the various problems, and we have proposed several polynomial time heuristics for the constructive scenario. We compared these heuristics through simulation, allowing us to identify one heuristic that is almost always better than the others, Subtree-BU. Finally, we assessed the absolute performance of our heuristics with respect to the optimal solution of the linear program for homogeneous platforms and small problem instances. It turns out that the Subtree-BU heuristic almost always produces optimal results.

An interesting direction for future work is the study of the case when multiple applications must be executed simultaneously so that a given throughput must be achieved for each application (see preliminary results in [15]). In this case a clear opportunity for higher performance with a reduced cost is the reuse of common subexpression between trees [22, 23]. Another direction is the study of applications that are mutable, i.e., whose operators can be rearranged based on operator associativity and commutativity rules. Such situations arise for instance in relational database applications [6].

## References

[1] B. Badcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Intl. Conf. on Very Large Data Bases*, 2004, pp. 456–467.

[2] U. Srivastava, K. Munagala, and J. Widom, "Operator Placement for In-Network Stream Query Processing," in *Proceedings of the 24th Intl. Conf. on Principles of Database Systems*, 2005.

[3] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck, "Gigascope: high-performance network monitoring with an SQL interface," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002, pp. 623–633.

[4] R. van Rennesse, K. Birman, D. Dumitriu, and W. Vogels, "Scalable Management and Data Mining Using Astrolabe," in *Proceedings from the First Intl. Workshop on Peer-to-Peer Systems*, 2002.

[5] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs, "Reclaiming Network-wide Visibility Using Ubiquitous End System Monitors," in *Proceedings of the USENIX Annual Technical Conf.*, 2006.

[6] J. Chen, D. J. DeWitt, and J. F. Naughton, "Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries," in *Proceedings of ICDE*, 2002.

[7] B. Plale and K. Schwan, "Dynamic Querying of Streaming Data with the dQUOB System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 422–432, 2003.

[8] J. Kräme and B. Seeger, "A Temporal Foundation for Continuous Queries over Data streams," in *Proceedings of the Intl. Conf. on Management of Data*, 2005, pp. 70–82.
[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable distributed stream processing," in *Proc. of the CIDR Conf.*, January 2003.
[10] L. Chen, K. Reddy, and G. Agrawal, "GATES: a grid-based middleware for processing distributed data streams," *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pp. 192–201, 4-6 June 2004.
[11] D. Logothetis and K. Yocum, "Wide-Scale Data Stream Management," in *Proceedings of the USENIX Annual Technical Conference*, 2008.
[12] Y. Ahmad and U. Cetintemel, "Network aware query processing for stream-based applications," in *Proceedings of the International Conference on Very Large Data Bases*, 2004, pp. 456–467.
[13] P. Pietzuch, J. Leflie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," in *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, 2006.
[14] "Amazon Elastic Compute Cloud (Amazon EC2)," *http://aws.amazon.com/ec2/*.
[15] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert, "Resource Allocation for Concurrent In-Network Stream-Processing Applications," in *Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, HeteroPar'09*, 2009.
[16] Y. E. Ioannidis, "Query optimization," *ACM Computing Surveys*, vol. 28, no. 1, pp. 121–123, 1996.
[17] B. Hong and V. K. Prasanna, "Adaptive allocation of independent tasks to maximize throughput," *IEEE Trans. Parallel Distributed Systems*, vol. 18, no. 10, pp. 1420–1435, 2007.
[18] O. Beaumont, A. Legrand, L. Marchal and Y. Robert, "Steady-state scheduling on heterogeneous clusters," *Int. J. of Foundations of Computer Science*, vol. 16, no. 2, pp. 163-1942005.
[19] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert, "Resource Allocation Strategies for In-Network Stream Processing," LIP, ENS Lyon, France, Research Report 2008-20, June 2008. Available at *http://graal.ens-lyon.fr/∼abenoit/*.
[20] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
[21] "Source Code for the Heuristics," *http://graal.ens-lyon.fr/∼vsonigo/code/query-streaming/*.
[22] V. Pandit and H. Ji, "Efficient in-network evaluation of multiple queries," in *HiPC*, 2006.
[23] K. Munagala, U. Srivastava, and J. Widom, "Optimization of continuous queries with shared expensive filters," in *PODS '07: Proc. of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2007.