Resource-Aware Application State Monitoring

Shicong Meng, *Student Member*, *IEEE*, Srinivas Raghav Kashyap,
Chitra Venkatramani, and Ling Liu, *Senior Member*, *IEEE*

Abstract—The increasing popularity of large-scale distributed applications in datacenters has led to the growing demand of distributed application state monitoring. These application state monitoring tasks often involve collecting values of various status attributes from a large number of nodes. One challenge in such large-scale application state monitoring is to organize nodes into a monitoring overlay that achieves monitoring scalability and cost effectiveness at the same time. In this paper, we present REMO, a REsource-aware application state MOnitoring system, to address the challenge of monitoring overlay construction. REMO distinguishes itself from existing works in several key aspects. First, it jointly considers intertask cost-sharing opportunities and node-level resource constraints. Furthermore, it explicitly models the per-message processing overhead which can be substantial but is often ignored by previous works. Second, REMO produces a forest of optimized monitoring trees through iterations of two phases. One phase explores cost-sharing opportunities between tasks, and the other refines the tree with resource-sensitive construction schemes. Finally, REMO also employs an adaptive algorithm that balances the benefits and costs of overlay adaptation. This is particularly useful for large systems with constantly changing monitoring tasks. Moreover, we enhance REMO in terms of both performance and applicability with a series of optimization and extension techniques. We perform extensive experiments including deploying REMO on a BlueGene/P rack running IBM's large-scale distributed streaming system—System S. Using REMO in the context of collecting over 200 monitoring tasks for an application deployed across 200 nodes results in a 35-45 percent decrease in the percentage error of collected attributes compared to existing schemes.

Index Terms—Resource-aware, state monitoring, distributed monitoring, datacenter monitoring, adaptation, data-intensive

1 Introduction

Recently, we have witnessed a fast growing set of large-scale distributed applications ranging from stream processing [1] to applications [2] running in Cloud datacenters. Correspondingly, the demand for monitoring the functioning of these applications also increases substantially. Typical monitoring of such applications involves collecting values of metrics, e.g., performance related metrics, from a large number of member nodes to determine the state of the application or the system. We refer to such monitoring tasks as *application state monitoring*. Application state monitoring is essential for the observation, analysis, and control of distributed applications and systems. For instance, data stream applications may require monitoring the data receiving/sending rate, captured events, tracked data entities, signature of internal states, and any number of application-specific attributes on participating computing nodes to ensure stable operation in the face of highly bursty workloads [3] [4]. Application provisioning may also require continuously collecting performance attribute values such as CPU usage, memory usage, and packet size distributions from application-hosting servers [5].

One central problem in application state monitoring is organizing nodes into a certain topology where metric values from different nodes can be collected and delivered. In many cases, it is useful to collect detailed performance attributes at a controlled collection frequency. As an example, fine-grained performance characterization information is required to construct various system models and to test hypotheses on system behavior [1]. Similarly, the data rate and buffer occupancy in each element of a distributed application may be required for diagnosis purposes when there is a perceived bottleneck [3]. However, the overhead of collecting monitoring data grows quickly as the scale and complexity of monitoring tasks increase. Hence, it is crucial that the monitoring topology should ensure good monitoring scalability and cost effectiveness at the same time.

While a set of monitoring-topology planning approaches have been proposed in the past, we find that these approaches often have the following drawbacks in general. First of all, existing works either build monitoring topologies for each individual monitoring task (TAG [6], SDIMS [7], PIER [8], join aggregations [9], REED [10], operator placement [11]), or use a static monitoring topology for all monitoring tasks [11]. These two approaches, however, often produce suboptimal monitoring topologies. For example, if two monitoring tasks both collect metric values over the same set of nodes, using one monitoring tree for monitoring data transmission is more efficient than using two, as nodes can merge updates for both tasks and reduce per-message processing overhead. Hence, multimonitoring-task-level topology optimization is crucial for monitoring scalability.

Second, for many data-intensive environments, monitoring overhead grows substantially with the increase of

- *S. Meng is with the Georgia Institute of Technology, 37975 GT Station, Atlanta, GA 30332. E-mail: smeng@cc.gatech.edu.*
- *S.R. Kashyap is with Google Inc., 1600 Amphitheatre Parkway Mountain View, CA 94043. E-mail: raaghav@gmail.com.*
- *C. Venkatramani is with the IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: chitrav@us.ibm.com.*
- *L. Liu is with the Georgia Institute of Technology, KACB 3340, Georgia Tech, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: lingliu@cc.gatech.edu.*

Manuscript received 26 June 2011; revised 5 Dec. 2011; accepted 9 Feb. 2012; published online 29 Feb. 2012.
Recommended for acceptance by J. Zhang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-06-0426.
Digital Object Identifier no. 10.1109/TPDS.2012.82.

monitoring tasks and deployment scale [7] [12]. It is important that the monitoring topology should be resource sensitive, i.e., it should avoid monitoring nodes spending excessive resources on collecting and delivering attribute values. Unfortunately, existing works do not take node-level resource consumption as a first-class consideration. This may result in overload on certain nodes which eventually leads to monitoring data loss. Moreover, some assumptions in existing works do not hold in real-world scenarios. For example, many works assume that the cost of update messages is only related with the number of values within the message, while we find that a fixed per-message overhead is not negligible.

Last but not the least, application state monitoring tasks are often subject to change in real-world deployments [13]. Some tasks are short term by nature, e.g., ad hoc tasks submitted to check the current system usage [14]. Other tasks may be frequently modified for debugging, e.g., a user may specify different attributes for one task to understand which attribute provides the most useful information [13]. Nevertheless, existing works often consider monitoring tasks to be static and perform one-time topology optimization [10] [11]. With little support for efficient topology adaptation, these approaches would either produce suboptimal topologies when using a static topology regardless of changes in tasks, or introduce high adaptation cost when performing comprehensive topology reconstruction for any change in tasks [15].

In this paper, we present REMO, a resource-aware application state monitoring system, that aims at addressing the above issues. REMO takes node-level available resources as the first class factor for building a monitoring topology. It optimizes the monitoring topology to achieve the best scalability and ensures that no node would be assigned with excessive monitoring workloads for their available resources.

REMO employs three key techniques to deliver cost-effective monitoring topologies under different environments. In the conference version [15] of this paper, we introduced a *basic topology planning algorithm*. This algorithm produces a forest of carefully optimized monitoring trees for a set of static monitoring tasks. It iteratively explores cost-sharing opportunities among monitoring tasks and refines the monitoring trees to achieve the best performance given the resource constraints on each node. One limitation of the basic approach is that it explores the entire search space for an optimal topology whenever the set of monitoring tasks is changed. This could lead to significant resource consumption for monitoring environments where tasks are subject to change. In this journal paper, we present an *adaptive topology planning algorithm* which continuously optimizes the monitoring topology according to the changes of tasks. To achieve cost effectiveness, it maintains a balance between the topology adaptation cost and the topology efficiency, and employs cost-benefit throttling to avoid trivial adaptation. To ensure the efficiency and applicability of REMO, we also introduce a set of *optimization and extension techniques* in this paper. These techniques further improve the efficiency of resource-sensitive monitoring tree construction scheme, and allow REMO to support popular monitoring features such as In-network aggregation and reliability enhancements.

We undertake an experimental study of our system and present results including those gathered by deploying REMO on a BlueGene/P rack (using 256 nodes booted into Linux) running IBM's large-scale distributed streaming system—System S [3]. The results show that our resource-aware approach for application state monitoring consistently outperforms the current best known schemes. For instance, in our experiments with a real application that spanned up to 200 nodes and about as many monitoring tasks, using REMO to collect attributes resulted in a 35-45 percent reduction in the percentage error of the attributes that were collected.

To our best knowledge, REMO is the first system that promotes resource-aware methodology to support and scale multiple application state monitoring tasks in large-scale distributed systems. We make three contributions in this paper.

- We identify three critical requirements for large-scale application state monitoring: *the sharing of message processing cost among attributes*, *meeting node-level resource constraints*, and *efficient adaptation toward monitoring task changes*. Existing approaches do not address these requirements well.
- We propose a framework for communication-efficient application state monitoring. It allows us to optimize monitoring topologies to meet the above three requirements under a single framework.
- We develop techniques to further improve the applicability of REMO in terms of runtime efficiency and supporting new monitoring features.

The rest of the paper is organized as follows: Section 2 identifies challenges in application state monitoring. Section 3 illustrates the basic monitoring topology construction algorithm, and Section 4 introduces the adaptive topology construction algorithm. We optimize the efficiency of REMO and extend it for advanced features in Sections 5 and 6. We present our experimental results in Section 7. Section 8 describes related works and Section 9 concludes this paper.

2 System Overview

In this section, we introduce the concept of application state monitoring and its system model. We also demonstrate the challenges in application state monitoring, and point out the key questions that an application state monitoring approach must address.

2.1 Application State Monitoring

Users and administrators of large-scale distributed applications often employ application state monitoring for observation, debugging, analysis, and control purposes. Each application state monitoring task periodically collects values of certain attributes from the set of computing nodes over which an application is running. We use the term attribute and metric interchangeably in this paper. As we focus on monitoring topology planning rather than the actual production of attribute values [16], we assume values of attributes are made available by application-specific tools or management services. In addition, we target at datacenter-like monitoring environments where any two nodes can communicate with similar cost (more details in Section 3.3). Formally, we define an application state monitoring task t as follows:

Definition 1. *A monitoring task $t = (A_t, N_t)$ is a pair of sets, where $A_t \subseteq \bigcup_{i \in N_t} A_i$ is a set of attributes and $N_t \subseteq N$ is a set*

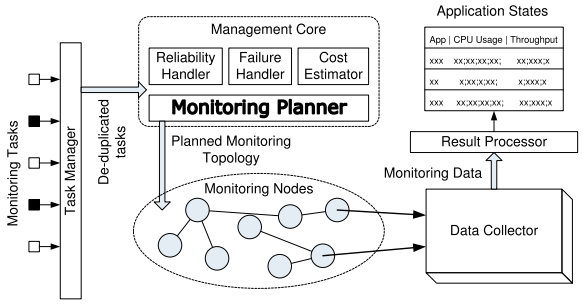


Fig. 1. A high-level system model.

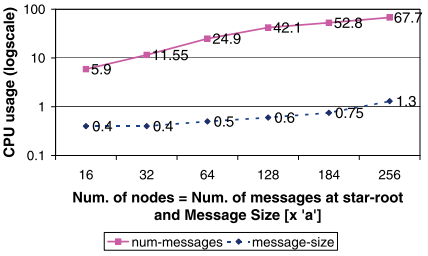


Fig. 2. CPU usage versus increasing message number/size.

of nodes. In addition, t can also be represented as a list of node-attribute pairs (i, j), where $i \in N_t, j \in A_t$.

2.2 The Monitoring System Model

Fig. 1 shows the high-level model of REMO, a system we developed to provide application state monitoring functionality. REMO consists of several fundamental components:

Task manager takes state monitoring tasks and removes duplication among monitoring tasks. For instance, monitoring tasks

$$t_1 = (\{cpu\_utilization\}, \{a, b\}) \text{ and}$$
$$t_2 = (\{cpu\_utilization\}, \{b, c\})$$

have duplicated monitored attribute $cpu\_utilization$ on node b. With such duplication, node b has to send $cpu\_utilization$ information twice for each update, which is clearly unnecessary. Therefore, given a set of monitoring tasks, the task manager transforms this set of tasks into a list of node-attribute pairs and eliminates all duplicated node-attribute pairs. For instance, t_1 and t_2 are equivalent to the list $\{a\text{-}cpu\_utilization, b\text{-}cpu\_utilization\}$ and $\{b\text{-}cpu\_utilization, c\text{-}cpu\_utilization\}$, respectively. In this case, node-attribute pair $\{b\text{-}cpu\_utilization\}$ is duplicated, and thus, is eliminated from the output of the task manager.

Management core takes deduplicated tasks as input and schedules these tasks to run. One key subcomponent of the management core is the *monitoring planner* which determines the interconnection of monitoring nodes. For simplicity, we also refer to the overlay connecting monitoring nodes as the *monitoring topology*. In addition, the management core also provides important support for reliability enhancement and failure handling. **Data collector** provides a library of functions and algorithms for efficiently collecting attribute values from the monitoring network. It also serves as the repository of monitoring data and provides monitoring data access to users and high-level applications. **Result processor** executes the concrete monitoring operations including collecting and aggregating attribute values, triggering warnings, etc.

In this paper, we focus on the design and implementation of the monitoring planner. We next introduce monitoring overhead in application state monitoring which drives the design principles of the monitoring planner.

2.3 Monitoring Overhead and Monitoring Planning

On a high level, a monitoring system consists of n monitoring nodes and one central node, i.e., data collector. Each monitoring node has a set of observable attributes $A_i = \{a_j | j \in [1, m]\}$. Attributes at different nodes but with the same subscription are considered as attributes of the same type. For instance, monitored nodes may all have locally observable CPU utilization. We consider an attribute as a continuously changing variable which outputs a new value in every unit time. For simplicity, we assume all attributes are of the same size a and it is straightforward to extend our work to support attributes with different sizes.

Each node i, the central node or a monitoring node, has a capacity b_i (also referred to as the resource constraint of node i) for receiving and transmitting monitoring data. In this paper, we consider CPU as the primary resource for optimization. We associate each message transmitted in the system with a per-message overhead C, and, the cost of transmitting a message with x values is $C + ax$. This cost model is motivated by our observations of monitoring resource consumption on a real-world system which we introduce next.

Our cost model considers both per-message overhead and the cost of payload. Although other models may consider only one of these two, our observation suggests that both costs should be captured in the model. Fig. 2 shows how significant the per-message processing overhead is. The measurements were performed on a BlueGene/P node which has a quad core 850 MHz PowerPC processor. The figure shows an example monitoring task where nodes are configured in a star network where each node periodically transmits a single fixed small message to a root node over TCP/IP. The CPU utilization of the root node grows roughly linearly from around 6 percent for 16 nodes (the root receives 16 messages periodically) to around 68 percent for 256 nodes (the root receives 256 messages periodically). Note that this increased overhead is due to the increased number of messages at the root node and not due to the increase in the total size of messages. Furthermore, the cost incurred to receive a single message increases from 0.2 to 1.4 percent when we increase the number of values in the message from 1 to 256. Hence, we also model the cost associated with message size as a message may contain a large number of values relayed for different nodes.

In other scenarios, the per-message overhead could be transmission or protocol overhead. For instance, a typical monitoring message delivered via TCP/IP protocol has a message header of at least 78 bytes not including application-specific headers, while an integer monitoring data is just 4 bytes.

As Fig. 3 shows, given a list of node-attribute pairs, the monitoring planner organizes monitoring nodes into a forest of monitoring trees where each node collects values for a set of attributes. The planner considers the aforementioned per-message overhead as well as the cost of attributes

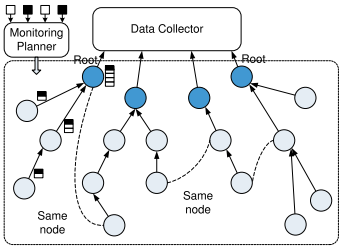


Fig. 3. An example of monitoring planning.

transmission (as illustrated by the black and white bar in the left monitoring tree) to avoid overloading certain monitoring nodes in the generated monitoring topology. In addition, it also optimizes the monitoring topology to achieve maximum monitoring data delivery efficiency. As a result, one monitoring node may connect to multiple trees (as shown in Figs. 3 and 4c). Within a monitoring tree T, each node i periodically sends an update message to its parent. As application state monitoring requires collecting values of certain attributes from a set of nodes, such update messages include both values locally observed by node i and values sent by i's children, for attributes monitored by T. Thus, the size of a message is proportional to the number of monitoring nodes in the subtree rooted at node i. This process continues upward in the tree until the message reaches the central data collector node.

2.4 Challenges in Monitoring Planning

From the users' perspective, monitoring results should be as accurate as possible, suggesting that the underlying monitoring network should maximize the number of node-attribute pairs received at the central node. In addition, such a monitoring network should not cause the excessive use of resource at any node. Accordingly, we define the monitoring planning problem (MP) as follows:

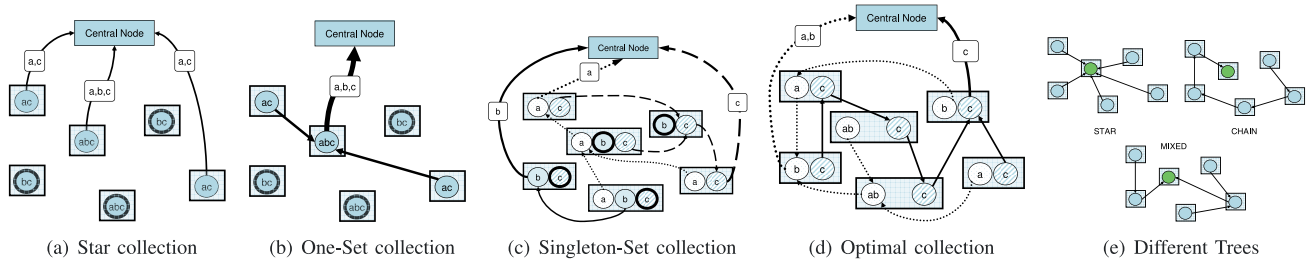
Problem Statement 1. *Given a set of node-attribute pairs for monitoring* $\Omega = \{\omega_1, \omega_2, \ldots, \omega_p\}$ *where* $\omega_q = (i, j)$, $i \in N$, $j \in A$, $q \in [1, p]$, *and resource constraint* b_i *for each associated node, find a parent* $f(i, j), \forall i, j$, *where* $j \in A_i$ *such that node* i *forwards attribute* j *to node* $f(i, j)$ *that maximizes the total number of node-attribute pairs received at the central node and the resource demand of node* i, d_i, *satisfies* $d_i \leq b_i, \forall i \in N$.

NP-completeness. When restricting all nodes to only monitor the same attribute j, we obtain a special case of the monitoring planning problem where each node has at most one attribute to monitor. As shown by Kashyap et al. [17], this special case is an NP-complete problem. Consequently, the monitoring planning problem is an NP-complete problem, since each instance of MP can be restricted to this special case. Therefore, in REMO, we primarily focus on efficient approaches that can deliver reasonably good monitoring plan.

We now use some intuitive examples to illustrate the challenges and the key questions that need to be addressed in designing a resource-aware monitoring planner. Fig. 4 shows a monitoring task involving 6 monitoring nodes where each node has a set of attributes to deliver (as indicated by alphabets on nodes). The four examples (a), (b), (c), and (d) demonstrate different approaches to fulfill this monitoring task. Example (a) shows a widely used topology in which every node sends its updates directly to the central node. Unfortunately, this topology has poor scalability, because it requires the central node to have a large amount of resources to account for per-message overhead. We refer to the approach used in example (a) as the *star collection*. Example (b) organizes all monitoring nodes in a single tree which delivers updates for all attributes. While this monitoring plan reduces the resource consumption (per-message overhead) at the central node, the root node now has to relay updates for all node-attribute pairs, and again faces scalability issues due to limited resources. We refer to this approach as *one-set collection*. These two examples suggest that achieving certain degree of load balancing is important for a monitoring network.

However, load balance alone does not lead to a good monitoring plan. In example (c), to balance the traffic among nodes, the central node uses three trees, each of which delivers only one attribute, and thus achieves a more balanced workload compared with example (b) (one-set collection) because updates are relayed by three root nodes. However, since each node monitors at least two attributes, nodes have to send out multiple update messages instead of one as in example (a) (star collection). Due to per-message overhead, this plan leads to higher resource consumption at almost *every* node. As a result, certain nodes may still fail to deliver all updates and less resources will be left over for additional monitoring tasks. We refer to the approach in example (c) as *singleton-set collection*.

The above examples reveal two fundamental aspects of the monitoring planning problem. First, *how to determine the number of monitoring trees and the set of attributes on each?* This is a nontrivial problem. Example (d) shows a topology which uses one tree to deliver attribute a, b, and another tree to deliver attribute c. It introduces less per-message overhead compared with example (c) (singleton-set collection) and is a more load-balanced solution compared with example (b) (one-set collection). Second, *how to determine the topology for*



(a) Star collection (b) One-Set collection (c) Singleton-Set collection (d) Optimal collection (e) Different Trees

Fig. 4. Motivating examples for the topology planning problem.

nodes in each monitoring tree under node-level resource constraints? Constructing monitoring trees subject to resource constraints at nodes is also a nontrivial problem and the choice of topology can significantly impact node resource usage. Example (e) shows three different trees. The star topology (upper left), while introducing the least relaying cost, causes significant per-message overhead at its root. The chain topology (upper right), on the contrary, distributes the per-message overhead among all nodes, but causes the most relaying cost. A "mixed" tree (bottom) might achieve a good tradeoff between relaying cost and per-message overhead, but it is determine its optimal topology.

3 The Basic REMO Approach

The basic REMO approach promotes the resource aware multitask optimization framework, consisting of a two phase iterative process and a suite of multitask optimization techniques. At a high level, REMO operates as a *guided local search* approach, which starts with an initial monitoring network composed of multiple independently constructed monitoring trees, and iteratively optimizes the monitoring network until no further improvements are possible. When exploring various optimization directions, REMO employs cost estimation to guide subsequent improvement so that the search space can be restricted to a small size. This guiding feature is essential for the scalability of large-scale application state monitoring systems.

Concretely, during each iteration, REMO first runs a *partition augmentation* procedure which generates a list of most promising candidate augmentations for improving the current distribution of monitoring workload among monitoring trees. While the total number of candidate augmentations is very large, this procedure can trim down the size of the candidate list for evaluation by selecting the most promising ones through cost estimation. Given the generated candidate augmentation list, the *resource-aware evaluation* procedure further refines candidate augmentations by building monitoring trees accordingly with a resource-aware tree construction algorithm. We provide more details to these two procedures in the following discussion.

3.1 Partition Augmentation

The partition augmentation procedure is designed to produce the attribute partitions that can potentially reduce message processing cost through a guided iterative process. These attribute partitions determine the number of monitoring trees in the forest and the set of attributes each tree delivers. To better understand the design principles of our approach, we first briefly describe two simple but most popular schemes, which essentially represent the state-of-the-art in multiple application state monitoring.

Recall that among example schemes in Fig. 4, one scheme (example (c)) delivers each attribute in a separate tree, and the other scheme (example (b)) uses a single tree to deliver updates for all attributes. We refer to these two schemes as the *Singleton-set partition scheme (SP)* and the *One-set partition (OP) scheme*, respectively. We use the term "partition" because these schemes partition the set of monitored attributes into a number of nonoverlapping subsets and assign each subsets to a monitoring tree.

Singleton-set partition. Specifically, given a set of attributes for collection A, singleton-set partition scheme divides A into $|A|$ subsets, each of which contains a distinct attribute in A. Thus, if a node has m attributes to monitor, it is associated with m trees. This scheme is widely used in previous work, e.g., PIER [8], which constructs a routing tree for each attribute collection. While this scheme provides the most balanced load among trees, it is not efficient, as nodes have to send update messages for each individual attribute.

One-set partition. The one-set partition scheme uses the set A as the only partitioned set. This scheme is also used in a number of previous work [11]. Using OP, each node can send just one message which includes all the attribute values, and thus, saves per-message overhead. Nevertheless, since the size of each message is much larger compared with messages associated with SP, the corresponding collecting tree cannot grow very large, i.e., contains limited number of nodes.

3.1.1 Exploring Partition Augmentations

REMO seeks a middle ground between these extreme solutions—one where nodes pay lower per-message overhead compared to SP while being more load-balanced and consequently more scalable than OP. Our partition augmentation scheme explores possible augmentations to a given attribute partition P by searching for all partitions that are *close* to P in the sense that the resulting partition can be created by modifying P with certain predefined operations.

We define two basic operations that are used to modify attribute set partitions.

Definition 2. *Given two attribute sets A_i^P and A_j^P in partition P, a **merge** operation over A_i^P and A_j^P, denoted as $A_i^P \bowtie A_j^P$, yields a new set $A_k^P = A_i^P \cup A_j^P$. Given one attribute set A_i^P and an attribute α, a **split** operation on A_i^P with regard to α, denoted as $A_i^P \triangleright \alpha$, yields two new sets $A_k^P = A_i^P - \alpha$.*

A merge operation is simply the union of two set attributes. A split operation essentially removes one attribute from an existing attribute set. As it is a special case of set difference operation, we use the set difference sign $(-)$ here to define split. Furthermore, there is no restriction on the number of attributes that can be involved in a merge or a split operation. Based on the definition of merge and split operations, we now define neighboring solution as follows:

Definition 3. *For an attribute set partition P, we say partition P' is a **neighboring solution** of P if and only if either $\exists A_i^P, A_j^P \in P$ so that $P' = P - A_i^P - A_j^P + (A_i^P \bowtie A_j^P)$, or $\exists A_i^P \in P, \alpha \in A_i^P$ so that $P' = P - A_i^P + (A_i^P \triangleright \alpha) + \{\alpha\}$.*

A neighboring solution is essentially a partition obtained by make "one-step" modification (either one merge or one split operation) to the existing partition.

Guided partition augmentation. Exploring all neighboring augmentations of a given partition and evaluating the performance of each augmentation is practically infeasible, since the evaluation involves constructing resource-constrained monitoring trees. To mitigate this problem, we use a guided partition augmentation scheme which greatly reduces the number of candidate partitions for evaluation.

The basic idea of this guided scheme is to rank candidate partitions according to the *estimated* reduction in the total capacity usage that would result from using the new partition. The rationale is that a partition that provides a large decrease in capacity usage will free up capacity for more attribute value pairs to be aggregated. Following this, we evaluate neighboring partitions in the decreased order of their estimated capacity reduction so that we can find a good augmentation without evaluating all candidates. We provide details of the gain estimation in the appendix which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.82. This guided local-search heuristic is essential to ensuring the practicality of our scheme.

3.2 Resource-Aware Evaluation

To evaluate the objective function for a given candidate partition augmentation m, the resource-aware evaluation procedure evaluates m by constructing trees for nodes affected by m and measures the total number of node-attribute value pairs that can be collected using these trees. This procedure primarily involves two tasks. One is *constructing a tree* for a given set of nodes without exceeding resource constraints at any node. The other is for a node connected to multiple trees to *allocate its resources* to different trees.

3.2.1 Tree Construction

The tree construction procedure constructs a collection tree for a given set of nodes D such that no node exceeds its resource constraints while trying to include as many nodes as possible into the constructed tree. Formally, we define the tree construction problem as follows:

Problem Statement 2. *Given a set of n vertices, each has x_i attributes to monitor, and resource constraint b_i, find a parent vertex $p(i), \forall i$, so that the number of vertices in the constructed tree is maximized subject to the following constraints where u_i is the resource consumed at vertex i for sending update messages to its parent*

1. *For any vertex i in the tree, $\sum_{p(j)=i} u_j + u_i \leq b_i$.*
2. *Let y_i be the number of all attribute values transmitted by vertex i. We have $y_i = x_i + \sum_{p(j)=i} x_j$.*
3. *According to our definition, $u_i \leq C + y_i \cdot a$.*

The first constraint requires that the resource spent on node i for sending and receiving updates should not exceed its resource constraint b_i. The second constraint requires a node to deliver its local monitored values as well as values received from its children. The last constraint states that the cost of processing an outgoing message is the combination of per-message overhead and value processing cost. The tree construction problem, however, is also NP-complete [17] and we present heuristics for the tree-construction problem.

To start with, we first discuss two simple tree construction heuristics

Star. This scheme forms "star"-like trees by giving priority to increasing the breadth of the tree. Specifically, it adds nodes into the constructed tree in the order of decreased available capacity, and attaches a new node to the node with *the lowest height* and sufficient available capacity, until no such nodes exist. STAR creates bushy trees and consequently pays low relay cost. However, owing to large node degrees, the root node suffers from higher per-message overhead, and consequently, the tree cannot grow very large.

Chain. This scheme gives priority to increasing the height of the tree, and constructs "chain"-like trees. CHAIN adds nodes to the tree in the same way as STAR does except that it tries to attach nodes to the node with *the highest height* and sufficient available capacity. CHAIN creates long trees that achieve very good load balance, but due to the number of hops each message has to travel to reach the root, most nodes pay a high relay cost.

STAR and CHAIN reveal two conflicting factors in collection tree construction—resource efficiency and scalability. Minimizing tree height achieves resource efficiency, i.e., minimum relay cost, but causes poor scalability, i.e., small tree size. On the other hand, maximizing tree height achieves good scalability, but degrades resource efficiency. The adaptive tree construction algorithm seeks a middle-ground between the STAR and CHAIN procedures in this regard. It tries to minimize the total resource consumption, and can trade off overhead cost for relay cost, and vice versa, if it is possible to accommodate more nodes by doing so.

Before we describe the adaptive tree construction algorithm, we first introduce the concept of saturated trees and congested nodes as follows:

Definition 4. *Given a set of nodes N for tree construction and the corresponding tree T which contains a set of nodes $N' \subset N$, we say T is **saturated** if no more nodes $d \in (N - N')$ can be added to T without causing the resource constraint to be violated for at least one node in T. We refer to nodes whose resource constraint would be violated if $d \in (N - N')$ is added to T as **congested** nodes.*

The adaptive tree construction algorithm iteratively invokes two procedures, the *construction* procedure and the *adjusting* procedure. The construction procedure runs the STAR scheme which attaches new nodes to low-level existing tree nodes. As we mentioned earlier, STAR causes the capacity consumption at low-level tree nodes to be much heavier than that at other nodes. Thus, as low-level tree nodes become congested we get a saturated tree, the construction procedure terminates and returns all congested nodes. The algorithm then invokes the adjusting procedure, which tries to relieve the workload of low level tree nodes by reducing the degree of these nodes and increasing the height of the tree (similar to CHAIN). As a result, the adjusting procedure reduces congested nodes and makes a saturated tree unsaturated. The algorithm then repeats the constructing-adjusting iteration until no more nodes can be added to the tree or all nodes have been added.

3.3 Discussion

REMO targets at datacenter-like environments where the underlying infrastructure allows any two nodes in the network can communicate with similar cost, and focuses on the resource consumption on computing nodes rather than that of the underlying network. We believe this setting fits for many distributed computing environments, even when computing nodes are not directly connected. For instance,

communication packets between hosts located in the same rack usually pass through only one top-of-rack switch, while communication packets between hosts located in different racks may travel through longer communication path consisting of multiple switches or routers. The corresponding overhead on communication endpoints, however, is similar in these two cases as packet forwarding overhead is outsourced to network devices. As long as networks are not saturated, REMO can be directly applied for monitoring topology planning.

when the resource consumption on network devices needs to be considered, e.g., networks are bottleneck resources, REMO cannot be directly applied. Similarly, for environments where internode communication requires nodes to actively forward messages, e.g., peer-to-peer overlay networks and wireless sensor networks, the assumption of similar cost on communication endpoints does not hold as longer communication paths also incur higher forwarding cost. However, REMO can be extended to handle such changes. For example, its local search process can incorporate the forwarding cost in the resource evaluation of a candidate plan. We consider such extension for supporting such networks as our future work.

4 Runtime Topology Adaption

The basic REMO approach works well for a static set of monitoring tasks. However, in many distributed computing environments, monitoring tasks are often added, modified or removed on the fly for better information collection or debugging. Such changes necessitate the adaptation of monitoring topology. In this section, we study the problem of runtime topology adaptation for changes in monitoring tasks.

4.1 Efficient Adaptation Planning

One may search for an optimal topology by invoking the REMO planning algorithm every time a monitoring task is added, modified or removed, and update the monitoring topology accordingly. We refer to such an approach as *REBUILD*. REBUILD, however, may incur significant resource consumption due to topology planning computation as well as topology reconstruction cost (e.g., messages used for notifying nodes to disconnect or connect), especially in datacenter-like environments with a massive number of mutable monitoring tasks undergoing relatively frequent modification.

An alternative approach is to introduce minimum changes to the topology to fulfill the changes of monitoring tasks. We refer to such an approach as *DIRECT-APPLY* or *D-A* for brevity. D-A also has its limitation as it may result in topologies with poor performance over time. For instance, when we continuously add attributes to a task for collection, D-A simply instructs the corresponding tree to deliver these newly added attribute values until some nodes become saturated due to increased relay cost.

To address such issues, we propose an efficient adaptive approach that strikes a balance between adaptation cost and topology performance. The basic idea is to look for new topologies with good performance and small adaptation cost (including both searching and adaptation-related communication cost) based on the modification to monitoring tasks. Our approach limits the search space to topologies that are close variants of the current topology in order to achieve efficient adaptation. In addition, it ranks candidate adaptation operations based on their estimated cost-benefit ratios so that it always performs the most worthwhile adaptation operation first. We refer to this scheme as *ADAPTIVE* for brevity.

When monitoring tasks are added, removed or modified, we first applies D-A by building the corresponding trees with the tree building algorithm introduced in Section 3 (no changes in the attribute partition). We consider the resulting monitoring topology after invoking D-A as *the base topology*, which is then optimized by our ADAPTIVE scheme. Note that the base topology is only a virtual topology plan stored in memory. The actual monitoring topology is updated only when the ADAPTIVE scheme produces a final topology plan.

Same as the algorithm in Section 3, the ADAPTIVE scheme performs two operations, merging and splitting, over the base topology in an iterative manner. For each iteration, the ADAPTIVE scheme first lists all candidate adaptation operations for merging and splitting, respectively, and ranks the candidate operations based on estimated cost effectiveness. It then evaluates merging operations in the order of decreasing cost effectiveness until it finds a valid merging operation. It also evaluates splitting operations in the same way until it finds a valid splitting operations. From these two operations, it chooses one with the largest improvement to apply to the base topology.

Let T be the set of reconstructed trees. To ensure efficiency, the ADAPTIVE scheme considers only merging operations involving at least one tree in T as candidate merging operations. This is because merging two trees that are not in T is unlikely to improve the topology. Otherwise, previous topology optimization process would have adopted such a merging operation. The evaluation of a merging operation involves computationally expensive tree building. As a result, evaluating only merging operations involving trees in T greatly reduces the search space and ensures the efficiency and responsiveness (changes of monitoring tasks should be quickly applied) of the ADAPTIVE scheme. For a monitoring topology with n trees, the number of possible merging operations is $C_n^2 = \frac{n(n-1)}{2}$, while the number of merging operations involving trees in T is $|T| \cdot (n-1)$ which is usually significantly smaller than $\frac{n(n-1)}{2}$ as $T << n$ for most monitoring task updates. Similarly, the ADAPTIVE scheme considers a splitting operation as a candidate operation only if the tree to be split is in T.

The ADAPTIVE scheme also minimizes the number of candidate merging/splitting operations it evaluates for responsiveness. It ranks all candidate operations and always evaluates the one with the greatest potential gain first. To rank candidate operations, the ADAPTIVE scheme needs to estimate the cost effectiveness of each operation. We estimate the cost effectiveness of an operation based on its estimated benefit and estimated adaptation cost. The estimated benefit is the same as $g(m)$ we introduced in Section 3. The estimated adaptation cost refers to the cost of applying the merging operation to the existing monitoring topology. This cost is usually proportional to the number of

edges modified in the topology. To estimate this cost, we use the lower bound of the number of edges that would have to be changed.

4.2 Cost-Benefit Throttling

The ADAPTIVE scheme must ensure that the benefit of adaption justifies the corresponding cost. For example, a monitoring topology undergoing frequent modification of monitoring tasks may not be suitable for frequent topology adaptation unless the corresponding gain is substantial. We employ cost-benefit throttling to apply only topology adaptations whose gain exceeds the corresponding cost. Concretely, when the ADAPTIVE scheme finds a valid merging or splitting operation, it estimates the adaptation cost by measuring the volume of control messages needed for adaptation, denoted by M_{adapt}. The algorithm considers the operation cost effective if M_{adapt} is less than a threshold defined as follows:

$$Threshold(A_m) = (T_{cur} - \min\{T_{adj,i}, i \in A_m\}) \cdot (C_{cur} - C_{adj}),$$

where A_m is the set of trees involved in the operation, $T_{adj,i}$ is the last time tree i being adjusted, T_{cur} is the current time, C_{cur} is the volume of monitoring messages delivered in unit time in the trees of the current topology, and C_{adj} is the volume of monitoring messages delivered in unit time in the trees after adaptation. $(T_{cur} - \min\{T_{adj,i}, i \in A_m\})$ essentially captures how frequently the corresponding trees are adjusted, and $(C_{cur} - C_{adj})$ measures the efficiency gain of the adjustment. Note that the threshold will be large if either the potential gain is large, i.e., $(C_{cur} - C_{adj})$ is large, or the corresponding trees are unlikely to be adjusted due to monitoring task updates, i.e., $(T_{cur} - \min\{T_{adj,i}, i \in A_m\})$ is large. Cost-benefit throttling also reduces the number of iterations. Once the algorithm finds that a merging or splitting is not cost-effective, it can terminate immediately.

5 Optimization

The basic REMO approach can be further optimized to achieve better efficiency and performance. In this section, we present two techniques, efficient tree adjustment and ordered resource allocation, to improve the efficiency of REMO tree construction algorithm and its planning performance, respectively.

5.1 Efficient Tree Adjustment

The tree building algorithm introduced in Section 3 iteratively invokes a construction procedure and an adjusting procedure to build a monitoring tree for a set of nodes. One issue of this tree building algorithm is that it generates high-computation cost, especially its adjusting procedure. To increase the available resource of a congested node d_c, the adjusting procedure tries to reduce its resource consumption on per-message overhead by reducing the number of its branches. Specifically, the procedure first removes the branch of d_c with the least resource consumption. We use b_{d_c} to denote this branch. It then tries to reattach nodes in b_{d_c} to other nodes in the tree except d_c. It considers the reattaching successful if all nodes of b_{d_c} is attached to the tree. As a result, the complexity of the adjusting procedure is $O(n^2)$ where n is the number of nodes in the tree.

We next present two techniques that reduces the complexity of the adjusting procedure. The first one, branch based reattaching, reduces the complexity to $O(n)$ by reattaching the entire branch b_{d_c} instead of individual nodes in b_{d_c}. It trades off a small chance of failing to reattaching b_{d_c} for considerable efficiency improvement. The second technique, Subtree-only searching, reduces the reattaching scope to the subtree of d_c, which considerably reduces searching time in practice (the complexity is still $O(n)$). The subtree-only searching also theoretically guarantees the searching completeness.

5.1.1 Branch-Based Reattaching

The above adjusting procedure breaks branches into nodes and moves one node at a time. This per-node-reattaching scheme is quite expensive. To reduce the time complexity, we adopts a branch-based reattaching scheme. As its name suggests, this scheme removes a branch from the congested node and attaches it entirely to another node, instead of breaking the branch into nodes and performing the reattaching. Performing reattaching in a branch basis effectively reduces the complexity of the adjusting procedure.

One minor drawback of branch-based reattaching is that it diminishes the chance of finding a parent to reattach the branch when the branch consists of many nodes. However, the impact of this drawback is quite limited in practice. As the adjusting procedure removes and reattaches the smallest branch first, failing to reattach the branch suggests that nodes of the tree all have limited available resource. In this case, node-based reattaching is also likely to fail.

5.1.2 Subtree-Only Searching

The tree adjusting procedure tries reattaching the pruned branch to all nodes in the tree except the congested node denoted as d_c. This adjustment scheme is not efficient as it enumerates almost every nodes in the tree to test if the pruned branch can be reattached to the node. It turns out that testing nodes outside d_c's subtree is often unnecessary. The following theorem suggests that testing nodes within d_c's subtree is sufficient as long as the resource demand of the node failed to add is higher than that of the pruned branch.

Theorem 1. *Given a saturated tree T outputted by the construction procedure, d_f the node failed to add to T, a congested node d_c and one of its branches b_{d_c}, attaching b_{d_c} to any node outside the subtree of d_c causes overload, given that the resource demand of d_f is no larger than that of b_{d_c}, i.e., $u_{d_f} \leq u_{b_{d_c}}$.*

Proof. If there exists a node outside the subtree of d_c, namely d_o, that can be attached with b_{d_c} without causing overload, then adding d_f to d_o should have succeeded in the previous execution of the construction procedure, as $u_{d_f} \leq u_{b_{d_c}}$. However, T is a saturated tree when adding d_f, which leads to a contradiction. □

Hence, we improve the efficiency of the original tree building algorithm by testing all nodes for reattaching only when the resource demand of the node failed to add is higher than that of the pruned branch. For all other cases, the algorithm performs reattaching test only within the subtree of d_c.

5.2 Ordered Resource Allocation

For a node associated with multiple trees, determining how much resource it should assign to each of its associated trees is necessary. Unfortunately, finding the optimal resource allocation is difficult because it is not clear how much resource a node will consume until the tree it connects to is built. Exploring all possible allocations to find the optimal one is clearly not an option as the computation cost is intractable.

To address this issue, REMO employs an efficient on-demand allocation scheme. Since REMO builds the monitoring topology on a tree-by-tree sequential basis, the on-demand allocation scheme defers the allocation decision until necessary and only allocates capacity to the tree that is about to be constructed. Given a node, the on-demand allocation scheme assigns all current available capacity to the tree that is currently under construction. Specifically, given a node i with resource b_i and a list of trees each with resource demand d_{ij}, the available capacity assigned to tree j is $b_i - \sum_{k=1}^{j-1} d_{ij}$. Our experiment results suggest that our on-demand allocation scheme outperforms several other heuristics.

The on-demand allocation scheme has one drawback that may limit its performance when building trees with very different sizes. As on-demand allocation encourages the trees constructed first to consume as much resource as necessary, the construction of these trees may not invoke the adjusting procedure which saves resource consumption on parent nodes by reducing their branches. Consequently, resources left for constructing the rest of the trees is limited.

We employ a slightly modified on-demand allocation scheme that relieves this issue with little additional planning cost. Instead of not specifying the order of construction, the new scheme constructs trees in the order of increasing tree size. The idea behind this modification is that small trees are more cost efficient in the sense that they are less likely to consume much resource for relaying cost. By constructing trees from small ones to large ones, the construction algorithm pays more relaying cost for better scalability only after small trees are constructed. Our experiment results suggest the ordered scheme outperforms the on-demand scheme in various settings.

6 Extensions

Our description of REMO so far is based on a simple monitoring scenario where tasks collect distributed values without aggregation or replication under a uniform value updating frequency. Real-world monitoring, however, often poses diverse requirements. In this section, we present three important techniques to support such requirements in REMO. The In-network-aggregation-aware planning technique allows REMO to accurately estimate per-node resource consumption when monitoring data can be aggregated before being passed to parent nodes. The reliability enhancement technique provides additional protection to monitoring data delivery by producing topologies that replicate monitoring data and pass them through distinct paths. The heterogeneous-update-frequency supporting technique enables REMO to plan topologies for monitoring tasks with different data updating frequencies by correctly estimating per-node resource consumption of such mixed workloads. These three techniques introduce little to no extra planning cost. Moreover, they can be incorporated into REMO as plugins when certain functionality is required by the monitoring environment without modifying the REMO framework.

6.1 Supporting In-Network Aggregation

In-network aggregation is important to achieve efficient distributed monitoring. Compared with holistic collection, i.e., collecting individual values from nodes, In-network aggregation allows individual monitoring values to be combined into aggregate values during delivery. For example, if a monitoring task requests the SUM of certain metric m on a set of nodes N, with In-network aggregation, a node can locally aggregate values it receives from other nodes into a single partial sum and pass it to its parent, instead of passing each individual value it receives.

REMO can be extended to build monitoring topology for monitoring tasks with In-network aggregation. We first introduce a *funnel function* to capture the changes of resource consumption caused by In-network aggregation. Specifically, a funnel function on node i, $fnl_i^m(g_m, n_m)$, returns the number of outgoing values on node i for metric m given the In-network aggregation type g_m and the number of incoming values n_m. The corresponding resource consumption of node i in tree k for sending update message to its parent is,

$$u_{ik} = C + \sum_{m \in A_i \cap A_k^P} a \cdot fnl_i^m(g_m, n_m), \tag{1}$$

where $A_i \cap A_k^P$ is the set of metrics node i needs to collect and report in tree k, a is the per value overhead and C is the per message overhead. For SUM aggregation, the corresponding funnel function is $fnl_i^m(SUM_m, n_m) = 1$ because the result of SUM aggregation is always a single value. Similarly, for TOP10 aggregation, the funnel function is $fnl_i^m(TOP10_m, n_m) = \min\{10, n_m\}$. For holistic aggregation we discussed earlier, $fnl_i^m(HOLISTIC_m, n_m) = n_m$. Hence, (1) can be used to calculate per-node resource consumption for both holistic aggregation and In-network aggregation in the aforementioned monitoring tree building algorithm. Note that it also supports the situation where one tree performs both In-network and holistic aggregation for different metrics.

Some aggregation functions such as DISTINCT, however, are data dependent in terms of the result size. For example, applying DISTINCT on a set X of 10 values results in a set with size ranging from 1 to 10, depending how many repeated values A contains. For these aggregations, we simply employ the funnel function of holistic aggregation for an upper bound estimation in the current implementation of REMO. Accurate estimation may require sampling-based techniques which we leave as our future work.

6.2 Reliability Enhancements

Enhancing reliability is important for certain mission critical monitoring tasks. REMO supports two modes of reliability enhancement, *same source different paths (SSDP)* and *different sources different paths (DSDP)*, to minimize the impact of link and node failure. The most distinct feature of the reliability enhancement in REMO is that the enhancement is done by rewriting monitoring tasks and requires little modification to the original approach.

The SSDP mode deals with link failures by duplicating the transmission of monitored values in different monitoring trees. Specifically, for a monitoring task $t = (a, N_t)$ requiring SSDP support, REMO creates a monitoring task $t' = (a', N_t)$ where a' is an alias of a. In addition, REMO restricts that a and a' would never occur in the same set of a partition P during partition augmentation, which makes sure messages updating a and a' are transmitted within different monitoring trees, i.e., different paths. Note that the degree of reliability can be adjusted through different numbers of duplications.

When a certain metric value is observable at multiple nodes, REMO also supports the DSDP mode. For example, computing nodes sharing the same storage observe the same storage performance metric values. Under this mode, users submit monitoring tasks in the form of $t = (a, N_{identical})$ where $N_{identical} = N(v_1), N(v_2), \ldots, N(v_n)$. $N(v_i)$ denotes the set of nodes that observe the same value v_i and $N_{identical}$ is a set of node groups each of which observes the same value. Let $k = \min\{|N(v_i)|, i \in [1, n]\}$. REMO rewrites t into k monitoring tasks so that each task collects values for metric a with a distinct set of nodes drawn from $N(v_i), i \in [1, n]$. Similar to SSDP model, REMO then constructs the topology by avoiding any of the k monitoring tasks to be clustered into one tree. In this way, REMO ensures values of metrics can be collected from distinct sets of nodes and delivered through different paths.

6.3 Supporting Heterogeneous Update Frequencies

Monitoring tasks may collect values from nodes with different update frequencies. REMO supports heterogeneous update frequencies by grouping nodes based on their metric collecting frequencies and constructing per-group monitoring topologies. When a node has a single metric a with the highest update frequency, REMO considers the node as having only one metric to update as other metrics piggyback on a. When a node has a set of metrics updated at the same highest frequencies, denoted by A_h, it evenly assigns other metrics to piggyback on metrics in of A_h. Similarly, REMO considers the node as having a set of metrics A_h to monitor as other metrics piggyback on A_h. We estimate the cost of updating with piggybacked metrics for node i by $u_i = C + a \cdot \sum_j freq_j / freq_{max}$ where $freq_j$ is the frequency of one metric collected on node i and $freq_{max}$ is the highest update frequency on node i.

Sometimes metric piggybacking cannot achieve the precise update frequency defined by users. For example, if the highest update frequency on a node is 1/5 (msg/sec), a metric updated at 1/22 can at best be monitored at either 1/20 or 1/25. If users are not satisfied with such an approximation, our current implementation separates these metrics out and builds individual monitoring trees for each of them.

7 Experimental Evaluation

We undertake an experimental study of our system and present results including those gathered by deploying REMO on a BlueGene/P rack (using 256 nodes booted into Linux) running IBM's large-scale distributed streaming system—System S.

Synthetic data set experiments. For our experiments on synthetic data, we assign a random subset of attributes to each node in the system. For monitoring tasks, we generate them by randomly selecting $|A_t|$ attributes and $|N_t|$ nodes with uniform distribution, for a given size of attribute set A and node set N. We also classify monitoring tasks into two categories—1) **small-scale** monitoring tasks that are for a small set of attributes from a small set of nodes, and 2) **large-scale** monitoring tasks that either involves many nodes or many attributes.

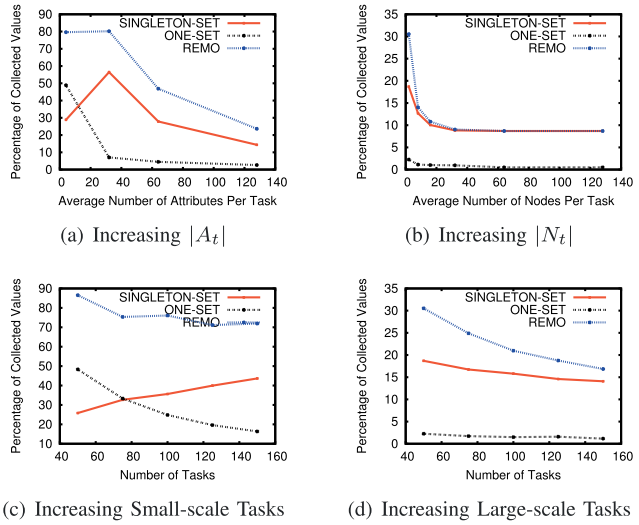
To evaluate the effectiveness of different topology construction schemes, we measure the percentage of attribute values collected at the root node with the monitoring topology produced by a scheme. Note that this value should be 100 percent when the monitoring workload is trivial or each monitoring node is provisioned with abundant monitoring resources. For comparison purposes, we apply relatively heavy monitoring workloads to keep this value below 100 percent for all schemes. This allows us to easily compare the performance of different schemes by looking at their percentage of collected values. Schemes with higher percentage of collected values not only achieve better monitoring coverage when monitoring resources are limited, but also have better monitoring efficiency in terms of monitoring resource consumption.

Real system experiments. Through experiments in a real system deployment, we also show that the error in attribute value observations (due to either stale or dropped attribute values) introduced by REMO is small. Note that this error can be measured in a meaningful way only for a real system and is what any "user" of the monitoring system would perceive when using REMO.

System S is a large-scale distributed stream processing middleware. Applications are expressed as dataflow graphs that specify analytic operators interconnected by data streams. These applications are deployed in the System S runtime as processes executing on a distributed set of hosts, and interconnected by stream connections using transports such as TCP/IP. Each node that runs application processes can observe attributes at various levels such as at the analytic operator level, System S middleware level, and the OS level. For these experiments, we deployed one such System S application called *YieldMonitor* [18], that monitors a chip manufacturing test process and uses statistical stream processing to predict the yield of individual chips across different electrical tests. This application consisted of over 200 processes deployed across 200 nodes, with 30-50 attributes to be monitored on each node, on a BlueGene/P cluster. The BlueGene is very communication rich and all compute nodes are interconnected by a 3D Torus mesh network. Consequently, for all practical purposes, we have a fully connected network where all pairs of nodes can communicate with each other at almost equal cost.

7.1 Result Analysis

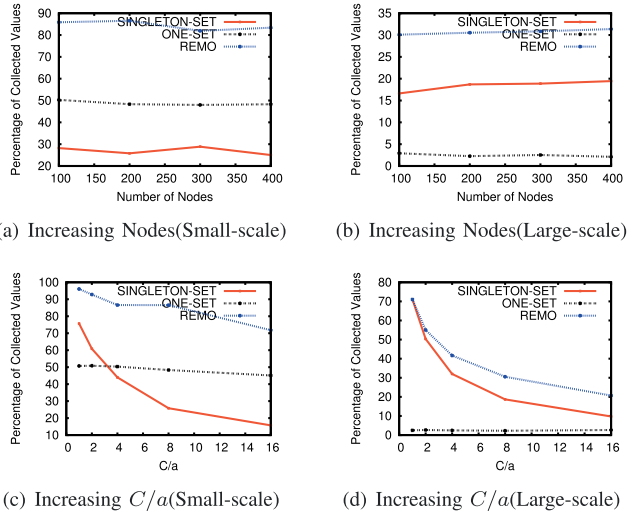
We present a small subset of our experimental results to highlight the following observations amongst others. First, REMO can collect a larger fraction of node-attribute pairs to serve monitoring tasks presented to the system compared to simple heuristics (which are essentially the state-of-the-art). REMO *adapts* to the task characteristics and outperforms each of these simple heuristics for all types of tasks and system characteristics, e.g., for small-scale tasks, a collection



(a) Increasing $|A_t|$ (b) Increasing $|N_t|$

(c) Increasing Small-scale Tasks (d) Increasing Large-scale Tasks

Fig. 5. Comparison of attribute set partition schemes under different workload characteristics.



(a) Increasing Nodes(Small-scale) (b) Increasing Nodes(Large-scale)

(c) Increasing C/a(Small-scale) (d) Increasing C/a(Large-scale)

Fig. 6. Comparison of attribute set partition schemes under different system characteristics.

mechanism with fewer trees is better while for large-scale tasks, a collection mechanism with more trees is better. Second, in a real application scenario, REMO also significantly reduces percentage error in the observed values of the node-attribute pairs required by monitoring tasks when compared to simple heuristics.

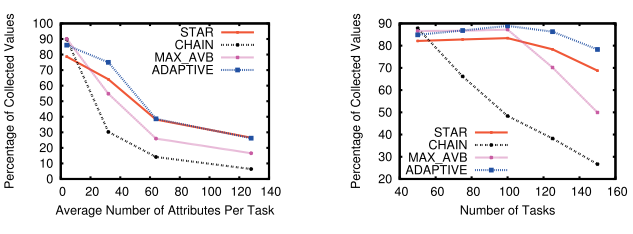
Varying the scale of monitoring tasks. Fig. 5 compares the performance of different attribute set partition schemes under different workload characteristics. In Fig. 5a, where we increase the number of attributes in monitoring tasks, i.e., increasing $|A_t|$, our partition augmentation scheme (REMO) performs consistently better than singleton-set (SINGLETON-SET) and one-set (ONE-SET) schemes. In addition, ONE-SET outperforms SINGLETON-SET when $|A_t|$ is relatively small. As each node only sends out one message which includes all its own attributes and those received from its children, ONE-SET causes the minimum per-message overhead. Thus, when each node monitors relatively small number of attributes, it can efficiently deliver attributes without suffering from its scalability problem. However, when $|A_t|$ increases, the capacity demand of the low-level nodes, i.e., nodes that are close to the root, increases significantly, which in turn limits the size of the tree and causes poor performance. In Fig. 5b, where we set $|A_t| = 100$ and increase $|N_t|$ to create extremely heavy workloads, REMO gradually converges to SINGLETON-SET, as SINGLETON-SET achieves the best load balance under heavy workload which in turn results in the best performance.

Varying the number of monitoring tasks. We observe similar results in Figs. 5c and 5d, where we increase the total number of small-scale and large-scale monitoring tasks, respectively.

Varying nodes in the system. Fig. 6 illustrates the performance of different attribute set partition schemes with changing system characteristics. In Figs. 6a and 6b, where we increase the number of nodes in the system given small- and large-scale monitoring tasks, respectively, we can see SINGLETON-SET is better for large-scale tasks while ONE-SET is better for small-scale tasks, and REMO performs much better than them in both cases, around 90 percent extra collected node-attribute pairs.

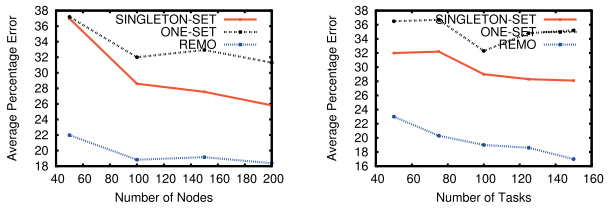
Varying per-message processing overhead. To study the impact of per-message overhead, we vary the C/a ratio under both small- and large-scale monitoring tasks in Figs. 6c and 6d. As expected, increased per-message overhead hits the SINGLETON-SET scheme hard since it constructs a large number of trees and, consequently, incurs the largest overhead cost while the performance of the ONE-SET scheme which constructs just a single tree degrades more gracefully. However having a single tree is not the best solution as shown by REMO which outperforms both the schemes as C/a is increased, because it can reduce the number of trees formed when C/a is increased.

Comparison of tree-construction schemes. In Fig. 7, we study the performance of different tree construction algorithms under different workloads and system characteristics. Our comparison also includes a new algorithm, namely *MAX_AVB*, a heuristic algorithm used in TMON [17] which always attaches new node to the existing node with the most available capacity. While we vary different workloads and system characteristics in the four figures, our adaptive tree construction algorithm (ADAPTIVE) always performs the best in terms of percentage of collected values. Among all the other tree construction schemes, STAR performs well when workload is heavy, as suggested by Figs. 7a and 7b. This is because STAR builds trees with minimum height, and thus, pays minimum cost for



(a) Increasing $|A_t|$ (b) Increasing Small-scale Tasks

Fig. 7. Comparison of tree construction schemes under different workload and system characteristics.



(a) Increasing Workload (b) Increasing Tasks

Fig. 8. Comparison of average percentage error.

relaying, which can be considerable when workloads are heavy. CHAIN performs the worst in almost all cases. While CHAIN provides good load balance by distributing per-message overhead in CHAIN-like trees, nodes have to pay high cost for relaying, which seriously degrades the performance of CHAIN when workloads are heavy (performs the best when workloads are light as indicated by the left portions of both Figs. 7a and 7b). MAX_AVB scheme outperforms both STAR and CHAIN given small workload, as it avoids over stretching a tree in breadth or height by growing trees from nodes with the most available capacity. However, its performance quickly degrades with increasing workload as a result of relaying cost.

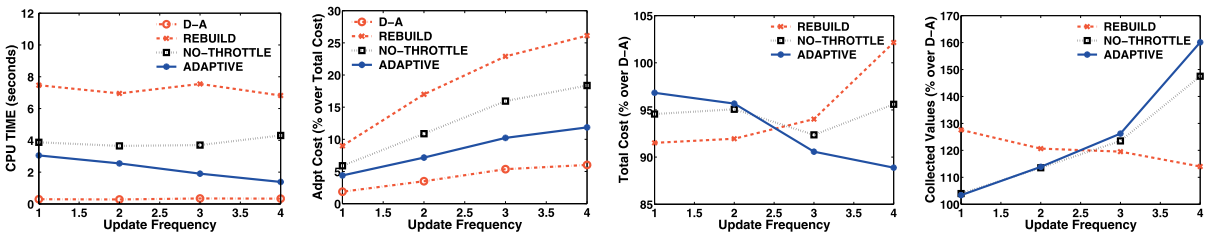
Real-world performance. To evaluate the performance of REMO in a real-world application. we measure the average percentage error of received attribute values for synthetically generated monitoring tasks. Specifically, we measure average percentage error between the snapshot of values observed by our scheme and compare it to the snapshot of "actual" values (that can be obtained by combining local log files at the end of the experiment). Fig. 8a compares the achieved percentage error between different partition schemes given increasing number of nodes. Recall that our system can deploy the application over any number of nodes. The figure shows that our partition augmentation scheme in REMO outperforms the other partition schemes. The percentage error achieved by REMO is around 30-50 percent less than that achieved by SINGLETON-SET and ONE-SET. Interestingly, the percentage error achieved by REMO clearly reduces when the number of nodes in the system increases. However, according to our previous results, the number of nodes has little impact on the coverage of collected attributes. The reason is that as the number of nodes increases, monitoring tasks are more sparsely distributed among nodes. Thus, each message is relatively small and each node can have more children. As a result, the monitoring trees constructed by our schemes are "bushier," which in turn reduces the percentage error caused by latency. Similarly, we can see that REMO gains significant error reduction compared with the other two schemes in Fig. 8b where we compare the performance of different partition schemes under increasing monitoring tasks.

Runtime adaptation. To emulate a dynamic monitoring environment with a small portion of changing tasks, we continuously update (modify) the set of running tasks with increasing update frequency. Specifically, we randomly select 5 percent of monitoring nodes and replaces 50 percent of their monitoring attributes. We also vary the frequency of task updates to evaluate the effectiveness of our adaptive techniques.

We compare the performance and cost of four different schemes: 1) DIRECT-APPLY (D-A) scheme which directly applies the changes in the monitoring task to the monitoring topology. 2) REBUILD scheme which always performs full-scale search from the initial topology with techniques we introduced in Section 3. 3) NO-THROTTLE scheme which searches for optimized topology that is close to the current one with techniques we introduced in Section 4. 4) ADAPTIVE scheme is the complete technique set described in Section 4, which improves NO-THROTTLE by applying cost-benefit throttling to avoid frequent topology adaptation when tasks are frequently updated.

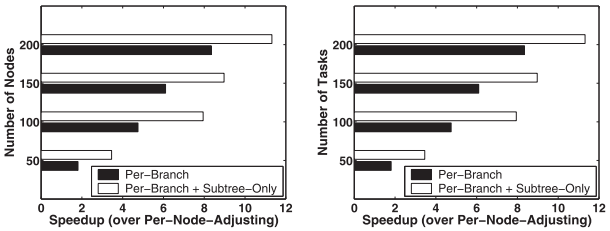
Fig. 9a shows the CPU time of running different planning schemes given increasing task updating frequency. The X-axis shows the number of task update batches within a time window of 10 value updates. The Y-axis shows the CPU time (measured on a Intel CORE Duo 2 2.26 GHz CPU) consumed by each scheme. We can see that D-A takes less than 1 second to finish since it performs only a single round tree construction. REBUILD consumes the most CPU time among all schemes as it always explores the entire searching space. When cost-benefit throttling is not applied the NO-THROTTLE scheme consumes less CPU time. However, its CPU time grows with task update frequency which is not desirable for large-scale monitoring. With throttling, the adaptive schemeincurs even less CPU time (1-3 s) as it avoids unnecessary topology optimization for frequent updates. Note that while the CPU time consumed by ADAPTIVE is higher than that of D-A, it is fairly acceptable.

Fig. 9b illustrates the percentage of adaptation cost over the total cost for each scheme. Here, the adaptation cost is measured by the total number of messages used to notifying monitoring nodes to change monitoring topology (e.g., one such message may inform a node to disconnect from its current parent node and connect to another parent



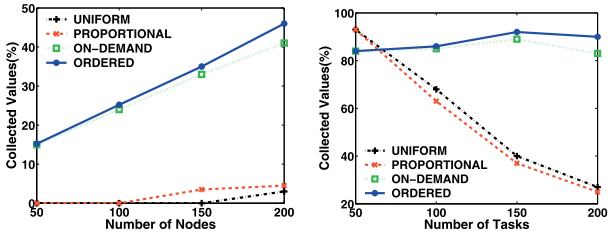
(a) CPU Time Consumption (b) Adaptation Cost (c) Total Cost (d) Percentage of Collected Values

Fig. 9. Performance comparison of different adaptation schemes given increasing task updating frequencies.



(a) Increasing Nodes (b) Increasing Tasks

Fig. 10. Speedup of optimization schemes.



(a) Increasing Nodes (b) Increasing Tasks

Fig. 11. Comparison between resource allocation schemes.

node). Similarly, the total cost of a scheme is the total number of messages the scheme used for both adaptation and delivering monitoring data. REBUILD introduces the highest adaptation cost because it always pursues the optimal topology which can be quite different from the current one. Similar to what we observed in Fig. 9a, NO-THROTTLE achieves much lower adaption cost compared with REBUILD does. ADAPTIVE further reduces adaptation cost which is very close to that of D-A, because cost-benefit throttling allows it to avoid unnecessary topology optimization when task updating frequency grows.

Fig. 9c shows the scheme-wise difference of the total cost (including both adaptation and data delivery messages). The Y-axis shows the ratio (percentage) of total cost associated with one scheme over that associated with D-A. REBUILD initially outperforms D-A as it produces optimized topology which in turn saves monitoring communication cost. Nevertheless, as task updating frequency increases, the communication cost of adaptation messages generated by REBUILD increases quickly, and eventually the extra cost in adaptation surpasses the monitoring communication cost it saves. NO-THROTTLE shows similar growth of total cost with increasing task updating frequency. ADAPTIVE, however, consistently outperforms D-A due to its ability to avoid unnecessary optimization.

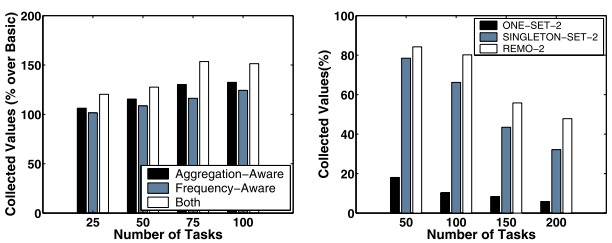
Fig. 9d shows the performance of schemes in terms of collected monitoring attribute values. The Y-axis shows the percentage of collected values of one scheme over that of D-A. Note that the result we show in Fig. 9c is the generated traffic volume. As each node cannot process traffics beyond its capacity, the more traffic generated, the more likely we observe miss-collected values. With increasing task updating frequency, the performance of REBUILD degrades faster than that of D-A due to its quickly growing cost in topology optimization (see Figs. 9b and 9c). On the contrary, both NO-THROTTLE and ADAPTIVE gain an increasing performance advantage over D-A. This is because the monitoring topology can still be optimized with relatively low adaptation cost with NO-THROTTLE and ADAPTIVE, but continuously degrades with D-A, especially with high task updating frequency.

Overall, ADAPTIVE produces monitoring topologies with the best value collection performance (Fig. 9d), which is the ultimate goal of monitoring topology planning. It achieves this by minimizing the overall cost of the topology (Fig. 9c) by only adopting adaptations whose gain outweighs cost. Its searching time and adaptation cost, although slighter higher than schemes such as D-A, is fairly small for all practical purposes.

Optimization. Figs. 10a and 10b show the speedup of our optimization techniques for the monitoring tree adjustment procedure, where the Y-axis shows the speedup of one technique over the basic adjustment procedure, i.e., the ratio between CPU time of the basic adjustment procedure over that of an optimized procedure. Because the basic adjustment procedure reattaches a branch by first breaking up the branch into individual nodes and performing a per-node-based reattaching, it takes considerably more CPU time compared with our branch-based reattach and subtree-only reattach techniques. With both techniques combined, we observe a speedup at up to 11 times, which is especially important for large distributed systems. We also find that these two optimization techniques introduce little performance penalties in terms of the percentage of values collected from the resulting monitoring topology($<2\%$).

Figs. 11a and 11b compare the performance of different tree-wise capacity allocation schemes, where UNIFORM divides the capacity of one node equally among all trees it participates in, PROPORTIONAL divides the capacity proportionally according to the size of each tree, ON-DEMAND and ORDERED are our allocation techniques introduced in Section 5.2. We can see that both ON-DEMAND and ORDERED consistently outperform UNIFORM and PROPORTIONAL. Furthermore, ORDERED gains an increasing advantage over ON-DEMAND with growing nodes and tasks. This is because large number of nodes and tasks cause one node to participate into trees with very different sizes, where ordered allocation is useful for avoiding improper node placement, e.g., putting one node as root in one tree (consuming much of its capacity) while it still needs to participate in other trees.

Extension. Fig. 12a compares the efficiency of basic REMO with that of extended REMO when given tasks that involves In-network aggregation and heterogeneous update frequencies. Specifically, we apply MAX In-network aggregation to tasks so that one node only needs to send the largest value to its parent node. In addition, we randomly



(a) Advanced Tasks (b) Replication Support

Fig. 12. Performance of extension techniques.

choose half of the tasks and reduce their value update frequency by half. The Y-axis shows values collected by REMO enhanced with one extension technique, normalized by values collected by the basic REMO. Note that collected values for MAX In-network aggregation refer to values included in the MAX aggregation, and are not necessarily collected by the root node.

The basic REMO approach is oblivious to In-network aggregation. Hence, it tends to overestimate communication cost of the monitoring topology, and prefers SINGLETON-SET-like topology where each tree delivers one or few attributes. As we mentioned earlier, such topologies introduce high per-message overhead. On the contrary, REMO with aggregation-awareness employs funnel functions to correctly estimate communication cost and produces more efficient monitoring topology. We observe similar results between the basic REMO and REMO with update-frequency-awareness. When both extension techniques are combined, they can provide an improvement close to 50 percent in terms of collected values.

Fig. 12b compares the efficiency of REMO with replication support and two alternative techniques. The SINGLETON-SET-2 scheme uses two SINGLETON-SET trees to deliver values of one attribute separately. The ONE-SET-2 scheme creates two ONE-SET trees, each of which connects all nodes and delivers values of all attributes separately. REMO-2 is our Same-Source-Different-Path technique with a replication factor of 2, i.e., values of each attribute are delivered through two different trees. Compared with the two alternative schemes, REMO-2 achieves both replication and efficiency by combining multiple attributes into one tree to reduce per-message overhead. As a result, it outperforms both alternatives consistently given increasing monitoring tasks.

8 Related Work

Much of the early work addressing the design of distributed query systems mainly focuses on executing single queries efficiently. As the focus of our work is to support multiple queries, we omit discussing these work. Research on processing multiple queries on a centralized data stream [19], [13], [20], [21] is not directly related with our work either, as the context of our work is distributed streaming where the number of messages exchanged between the nodes is of concern.

A large body of work studies query optimization and processing for distributed databases (see [22] for a survey). Although our problem bears a superficial resemblance to these distributed query optimization problems, our problem is fundamentally different since in our problem individual nodes are capacity constrained. There are also much work on multiquery optimization techniques for continuous aggregation queries over physically distributed data streams [21], [23], [24], [25], [19], [13], [20], [26]. These schemes assume that the routing trees are provided as part of the input. In our setting where we are able to choose from many possible routing trees, solving the joint problem of optimizing *resource-constrained* routing tree construction and multitask optimization provides significant benefit over solving only one of these problems in isolation as evidenced by our experimental results.

More recently, several work studies efficient data collection mechanisms. CONCH [27] builds a spanning forest with minimal monitoring costs for continuously collecting readings from a sensor network by utilizing temporal and spatial suppression. However, it does not consider the resource limitation at each node and per-message overhead as we did, which may limit its applicability in real-world applications. PIER [8] suggests using distinct routing trees for each query in the system, in order to balance the network load, which is essentially the SINGLETON-SET scheme we discussed. This scheme, though achieves the best load balance, may cause significant communication cost on per-message overhead.

9 Conclusions

We developed REMO, a resource-aware multitask optimization framework for scaling application state monitoring in large-scale distributed systems. The unique contribution of the REMO approach is its techniques for generating the network of monitoring trees that optimizes multiple monitoring tasks and balances the resource consumption at different nodes. We also proposed adaptive techniques to efficiently handle continuous task updates, optimization techniques that speedup the searching process up to a factor of 10, and techniques extending REMO to support advanced monitoring requirements such as In-network aggregation. We evaluated REMO through extensive experiments including deploying REMO in a real-world stream processing application hosted on BlueGene/P. Our experimental results show that REMO significantly and consistently outperforms existing approaches.

Acknowledgments

This work is partially supported by NSF grants from CISE CyberTrust program, NetSE program, and CyberTrust Cross Cutting program, and an IBM faculty award and an Intel ISTC grant on Cloud Computing.

References

[1] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, Implementation, and Evaluation of the Linear Road Bnchmark on the Stream Processing Core," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2006.

[2] B. Hayes, "Cloud Computing," *Comm. ACM*, vol. 51, no. 7, pp. 9-11, 2008.

[3] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive Control of Extreme-Scale Stream Processing Systems," *Proc. IEEE 26th Int'l Conf. Distributed Computing Systems (ICDCS)*, 2006.

[4] J. Borkowski, D. Kopanski, and M. Tudruj, "Parallel Irregular Computations Control Based on Global Predicate Monitoring," *Proc. Int'l Symp. Parallel Computing in Electrical Eng. (PARELEC)*, 2006.

[5] K. Park and V.S. Pai, "Comon: A Mostly-Scalable Monitoring System for Planetlab," *Operating Systems Rev.*, vol. 40, no. 1, pp. 65-74, 2006.

[6] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "Tag: A Tiny Aggregation Service for Ad-Hoc Sensor Networks," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI)*, 2002.

[7] P. Yalagandula and M. Dahlin, "A Scalable Distributed Information Management System," *Proc. SIGCOMM*, pp. 379-390, 2004.

[8] R. Huebsch, B.N. Chun, J.M. Hellerstein, B.T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A.R. Yumerefendi, "The Architecture of Pier: An Internet-Scale Query Processor," *Proc. Second Conf. Innovative Data Systems Research (CIDR)*, 2005.

[9] G. Cormode and M.N. Garofalakis, "Sketching Streams through the Net: Distributed Approximate Query Tracking," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB)*, pp. 13-24, 2005.
[10] D.J. Abadi, S. Madden, and W. Lindner, "Reed: Robust, Efficient Filtering, and Event Detection in Sensor Networks," *Proc. 31st Int'l Conf. Very Large Databases (VLDB)*, 2005.
[11] U. Srivastava, K. Munagala, and J. Widom, "Operator Placement for In-Network Stream Query Processing," *Proc. ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS)*, pp. 250-258, 2005.
[12] C. Olston, B.T. Loo, and J. Widom, "Adaptive Precision Setting for Cached Approximate Values," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '01)*, 2001.
[13] S. Krishnamurthy, C. Wu, and M.J. Franklin, "On-the-Fly Sharing for Streamed Aggregation," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '06)*, pp. 623-634, 2006.
[14] S. Ko and I. Gupta, "Efficient On-Demand Operations in Dynamic Distributed Infrastructures," *Proc. Second Workshop Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.
[15] S. Meng, S.R. Kashyap, C. Venkatramani, and L. Liu, "Remo: Resource-Aware Application State Monitoring for Large-Scale Distributed Systems," *Proc. IEEE 29th Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 248-255, 2009.
[16] K. Marzullo and M.D. Wood, *Tools for Constructing Distributed Reactive Systems.* Cornell Univ., 1991.
[17] S.R. Kashyap, D. Turaga, and C. Venkatramani, "Efficient Trees for Continuous Monitoring," 2008.
[18] D.S. Turaga, M. Vlachos, O. Verscheure, S. Parthasarathy, W. Fan, A. Norfleet, and R. Redburn, "Yieldmonitor: Real-Time Monitoring and Predictive Analysis of Chip Manufacturing Data," 2008.
[19] R. Zhang, N. Koudas, B.C. Ooi, and D. Srivastava, "Multiple Aggregations over Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2005.
[20] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker, "No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams," *ACM SIGMOD Record*, vol. 34, no. 1, pp. 39-44, 2005.
[21] S. Madden, M.A. Shah, J.M. Hellerstein, and V. Raman, "Continuously Adaptive Continuous Queries over Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2002.
[22] D. Kossmann, "The State of the Art in Distributed Query Processing," *ACM Computing Surveys*, vol. 32, no. 4, pp. 422-469, 2000.
[23] R. Huebsch, M.N. Garofalakis, J.M. Hellerstein, and I. Stoica, "Sharing Aggregate Computation for Distributed Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2007.
[24] A. Silberstein and J. Yang, "Many-to-Many Aggregation for Sensor Networks," *Proc. 27th Int'l Conf. Data Eng. (ICDE)*, pp. 986-995, 2007.
[25] S. Xiang, H.-B. Lim, K.-L. Tan, and Y. Zhou, "Two-Tier Multiple Query Optimization for Sensor Networks," *Proc. 27th Int'l Conf. Distributed Computing Systems (ICDCS)*, p. 39, 2007.
[26] J. Borkowski, "Hierarchical Detection of Strongly Consistent Global States," *Proc. Third Int'l Symp. Parallel and Distributed Computing/Third Int'l Workshop Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar)*, pp. 256-261, 2004.
[27] A. Silberstein, R. Braynard, and J. Yang, "Constraint Chaining: On Energy-Efficient Continuous Monitoring in Sensor Networks," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2006.

Shicong Meng currently working toward the PhD degree in the College of Computing at Georgia Tech. He is affiliated with the Center for Experimental Research in Computer Systems (CERCS) where he works with professor Ling Liu. His research focuses on performance, scalability, and security issues in large-scale distributed systems such as cloud datacenters. He is a student member of the IEEE.

Srinivas Raghav Kashyap received the PhD degree in computer science from the University of Maryland at collage park in 2007. Currently, he is a software engineer at Google Inc. and was a postdoctoral researcher at IBM T.J. Watson Research Center before moving to Google.

Chitra Venkatramani received the PhD degree in computer science from SUNY at Stony Brook in 1997, and was also engaged in various projects around multimedia data streaming and content distribution at IBM's Watson labs. She is a research staff member and manager of the Distributed Streaming Systems Group at the IBM T.J. Watson Research Center. The focus of her team is on research related to System S, a software platform for large scale, distributed stream processing. Her team explores issues in high performance, large-scale distributed computing systems, including advanced computation and communication infrastructures, fault tolerance, scheduling, and resource management for data-intensive applications. She holds numerous patents and publications and is also a recipient of IBM's Outstanding Technical Achievement Award.

Ling Liu is a full professor in the School of Computer Science at Georgia Institute of Technology. There she directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of data intensive systems with the focus on performance, availability, security, privacy, and energy efficiency. He has served as general chair and PC chairs of numerous IEEE and ACM conferences in data engineering, distributed computing, service computing, and cloud computing fields and is a coeditor-in-chief of the 5 volume Encyclopedia of Database Systems (Springer). Currently, she is on the editorial board of several international journals. He has published more than 300 International journal and conference articles in the areas of databases, distributed systems, and internet computing. She is a recipient of the Best Paper Award of ICDCS 2003, WWW 2004, the 2005 Pat Goldberg Memorial Best Paper Award, and 2008 international conference on Software Engineering and Data Engineering. Her current research is primarily sponsored by NSF, IBM, and Intel. She is a senior member of the IEEE.