# XCHANGE: High Performance Data Morphing in Distributed Applications

Jay Lofstead and Karsten Schwan

College of Computing, Georgia Institute of Technology

{lofstead, schwan}@cc.gatech.edu

Abstract

Distributed applications in which large volumes of data are exchanged between components that generate, process, and store or display data are common in both the high performance and enterprise domains. A key issue in these domains is mismatches of the data being generated with the data required by end users or by intermediate components. Mismatches are due to the need to customize or personalize data for certain end users, or they arise from natural differences in the data representations used by different components. In either case, mismatch correction – data morphing -- requires either servers or clients to perform extensive data processing. This paper describes automated methods and associated tools for morphing data in overlay networks that connect data producers with consumers. These methods automatically generate data transformation codes from declarative specifications, 'just in time', i.e., when and as needed. By describing data transformations declaratively, code generation can take into account the current nature of the data being generated, the current needs of data sinks, and the current resources available in the overlay connecting sources to sinks. In addition, code generation can consider the shared requirements of multiple consumers, to reduce redundant data transmissions and transformations. Data morphing is realized with the XCHANGE toolset, and in this paper, it is applied to both high performance and enterprise applications. Runtime generation and deployment of data morphing codes for filtering and transforming the large data volumes exchanged in a high performance remote data visualization is shown to result in improved network usage, able to generate code that matches the data volumes exchanged to available network resources. Morphing codes dynamically generated and deployed for an enterprise application in the healthcare domain demonstrates the importance of generating code so as to improve server scalability by reducing server loads.

Keywords: Data Transformation, Data Morphing, Dynamic Code Generation, Overlay Networks, XML, Declarative Specification, Distributed and Enterprise Applications, High Performance Data Exchange.

Technical Areas: Internet Computing and Applications, Data Management, Autonomic Computing

## 1. Introduction

A large class of service-oriented applications uses distributed components to capture, process, and display data in real-time across heterogeneous underlying execution platforms [58, 59, 60, 61]. A specific example is a scientific collaboration in which data is streamed from a running simulation to multiple end users. Here, large data volumes are accompanied by substantial processing demands when different end users require individualized views of simulation outputs. Another example is in the healthcare domain, where a hospital must share patient data across many distributed subsystems. Here, scalability concerns limit the overheads acceptable for the data processing and conversions needed to match data representations to the diverse needs of individual subsystems.

This paper addresses a general issue for the data-driven, service-oriented distributed applications described above: how to efficiently deal with dynamic mismatches in the data formats output by one application component (i.e., data source) vs. the inputs required by others (i.e., data sinks). The approach taken models the problem as one of semantic data mismatches, and it dynamically compiles explicit transformation specifications to the network overlay used to disseminate data from sources to sinks. Dynamic compilation and code deployment capitalize on the data streaming nature of data-driven applications, by eliminating repetitive data transformation actions and sharing selected transformation results across multiple data sinks.

Our research addresses many data mismatches. Examples range from row- vs. column-major data layouts and their associated impacts on the performance of parallel applications [25], to changes in time or length scales for data emitted vs. consumed by multi-scale modeling applications [30], to resolution changes or per-client requirements for data output by some simulation model vs. what is displayed by an end user device (e.g., see [1] or [31]). We also address mismatches in data acquisition, as with satellite data repeatedly processed until ingested by some simulation model or sensor data processed, filtered, and transformed prior to being used for some command and control function [32]. Finally, this work addresses applications where privacy or proprietary concerns require data to be filtered or transformed prior to being released to another software component [33].

A specific set of data mismatches studied in this paper occurs in the SmartPointer [1] framework for real-time scientific collaboration, which can be used to create per-user and per-device custom views of scientific data. For the molecular dynamics (MD) simulation data interactively viewed with SmartPointer, some users may be interested in viewing copper atoms and their positions, for instance, whereas others are interested in the bond forces occurring in the simulated material. Using the traditional client-server approach to distributed computing, a straightforward approach to viewing such data is to send all of it to each sink and then have each sink perform the processing necessary to implement its specific data view. For the large data volumes emitted by modern MD applications, this will generate additional and needless data transport and processing actions. The alternate approach of moving data conversion operations to the source has been shown useful [53], but it is limited the implied strain on source-side host CPU resources [34, 54] and/or the degrees of data expansion experienced at the source.

This paper describes the XCHANGE approach to dealing with data mismatches and transformations in distributed applications. The approach dynamically constructs and automatically populates with data morphing codes the overlay networks that connect sources to sinks. Runtime code generation and deployment are based on meta-information about data exchanges between participating parties, including (1) type information based on which data mismatches between outputs and inputs are diagnosed and corrected via automatically generated code and (2) transformations explicitly described by user-specified, meta-level instructions from which efficient binary codes are generated at runtime, whenever or wherever needed.

An interesting attribute of XCHANGE explored in this paper is the ability to combine methods that dynamically generate data transformation codes with the deployment of these codes into the overlay. Specifically, by combining both, code generation can be customized to take into account certain overlay characteristics, such as the processing and network resources available across multiple overlay nodes. Such customization uses real-time monitoring data about the resources available in the overlay. Code generation and deployment are driven by application-specific metrics, one being to create data conversions and map them to overlay nodes to reduce total network usage [35, 36], another one minimizing resource consumption [37] or some global metric like end-to-end delay [14]. In all such cases, by moving data morphing operations ‘into’ the overlay, improved scalability or performance can be attained [14].

XCHANGE is implemented as a set of tools that can be associated with any overlay-based messaging infrastructure, given the availability of meta-information about the type and structure of the data being exchanged between distributed application components. Our current implementation, targeting the high performance domain, uses the Portable Binary I/O format (PBIO) [2] to describe the structure of the binary data being moved. Efficient runtime PBIO binary data representations are combined with higher level data descriptions as XML schemas [11], using the XMIT [9] library. The overlay messaging infrastructure used in our experiments with XCHANGE is EVPath, evolved from the ECho high performance publish/subscribe infrastructure [7]. EVPath provides the overlay abstractions and general monitoring and control interfaces required by toolsets like XCHANGE.

In this context, the XCHANGE implementation provides the following new functionality: (1) high level descriptions of data transformations, processed by a transformation descriptor parser, (2) an optimizer that carries out cross-request optimizations on the code being generated, (3) a code distributor for mapping the optimized graph of transformations onto available overlay nodes, and (4) a transformation deployment daemon -- controller -- able to dynamically place transformation codes into the overlay. Jointly, XCHANGE and EVPath, therefore, offer functionality much beyond that of network-level multicast implementations like MBone [5], and they are applicable to a wide range of underlying execution platforms and network topologies, in contrast to projects like OMNI [6] aiming to create the best optimization scheme for overlay networks.

The advantages of using XCHANGE for implementing distributed data transformations are multi-fold:

- *Efficient data morphing through runtime code generation*. By explicitly describing data transformations, code implementing them can be generated automatically and dynamically, in the forms needed for the currently 'best' deployment to distributed overlay nodes. When bandwidth is limited, for example, generation and deployment can be optimized for unnecessary data transmission. Furthermore, runtime code generation makes it feasible to optimize data transformations (1) across entire affected overlay paths and (2) across all involved data sinks and sources. Sample optimization techniques used in this paper include the recognition and elimination of replicated data transformations and the use of operation dependencies to appropriately distribute operations across different overlay nodes. The outcome of such analyses is an optimized scheme for dynamically evolving data from a source to each of its sinks.
- *Dynamic, incremental deployment*. XCHANGE implements effective techniques for incorporating new requests into existing overlays, with small impact on the currently running application. Such dynamic deployment also affords us the opportunity to consider reconfiguring an existing overlay network based on changing conditions.
- *High performance*. By directly operating on binary data, XCHANGE-generated codes are applicable to both traditional high performance applications moving large data volumes and to applications that require low overheads due to high request volumes. That is, XCHANGE does not require application data to be in XML form, rather, it uses XML-based data format descriptions to generate efficient binary codes that operate on

the binary data being described. The motivation is to avoid (1) the overheads caused by runtime XML parsing and (2) the typically 10-fold data expansions experienced when using XML data encodings.

An interesting attribute of XCHANGE not highlighted in this paper is that its code generation and deployment schemes are replaceable since they are based on explicit meta-information maintained by the middleware about the overlay to which code is being deployed.

This paper demonstrates the advantages of using XCHANGE for dynamic data morphing in distributed systems with a distributed scientific data visualization, which consists of a single server sending display data to multiple sinks each requiring different data transformations. For this application, a deployment scheme optimized to reduce network usage demonstrates a 64% reduction in network usage vs. a sink-hosted transformation model and a 14% reduction over a source-hosted transformation model. A second use of XCHANGE for an application in the healthcare domain focuses on maintaining performance while offloading the transformation CPU workload from endpoints. One scenario uses a representative ‘results’ message from a laboratory system to a transactional system, where the two end points use different data organizations and have different scaling requirements. By transforming message formats in an intermediate node rather than at the endpoints, tasks can be offloaded from resource-poor or highly loaded end points without increases in required communication bandwidth, without discernible effects on or even improvements in end-to-end performance. Similar outcomes are observed for another scenario that evaluates the dynamic placement into the overlay of on-the-fly calculation to pre-calculate certain results desired by end points, such as calculating a Body Mass Index (BMI) across departments. Finally, micro-benchmarks show that dynamic code generation is not the limiting factor in XCHANGE performance, but rather, limitations are due to the need to dynamically construct and/or reconfigure the overlay itself, involving a distributed overlay control infrastructure and system level actions.

The remainder of this paper is organized as follows. Related work is discussed in Section 2, followed by an overview of the XCHANGE Architecture and Implementation in Section 3. Section 4 discusses and evaluates the XCHANGE implementation. Applications and Usage are discussed in Section 5. Section 6 discusses both the scientific and healthcare applications, and then uses them to illustrate the technical operation of XCHANGE and the optimizations enabled by it. Conclusions and future work appear in Section 7, followed by references in Section 8.

## 2. Related Work

Related work falls into four categories. The first uses database-style constructs to transform and combine data with an overlay network linking sources and sinks, as common in publish/subscribe infrastructures [7, 40, 62] and in systems that aim to provide rich data access models like DataCutter [13] and MOCHA [17, 19], STREAM [20], Aurora [21], and In-Transit [14]. Since data manipulations are expressed as SQL-style operations, technical issues addressed by these systems include dynamic query routing and the deployment (i.e., mapping) of query graphs to overlay nodes. In comparison, our work combines the generation of data manipulation codes with the mapping process, thereby offering an additional degree of freedom in how data transformations are carried out. We share with such systems the assumption that the data being manipulated is represented in some well-structured form.

The second category concerns tools that use XML syntax to describe querying of data sources. YATL [3] provides a declarative language for transforming a XML document into a new one based on the query issued. YATL is built as extensions to the ML language. In contrast, we manipulate data in its native binary form, using XML at the meta-level, to specify the data types and transformations being applied to binary data.

A third category concerns file system constructs designed for combining distributed data sources into a single virtual system. For example, the SRB [23] project is developing a Data Grid Language (DGL) [24] to describe how to route data within a grid environment, coupled with transformations on the metadata associated with file data. In comparison, we focus on transforming actual file data in addition to the metadata about files [52, 55]. However, the research described in this paper does not explicitly support file system APIs instead focusing on live data streaming and manipulation in response to data and QoS needs dynamically stated by sinks.

Fourth, an entirely different technology offering functionality similar to XCHANGE are the rich programming methods offered by Java-based infrastructures. Here, runtime deployment is supported by functionality like Enterprise Java Beans (EJBs) [38]. The techniques presented in this paper may be interpreted as an approach that addresses the potentially many data transformations occurring across bean interactions, but our implementation does not explicitly address this software environment. For instance, EJBs require a fixed type interface dictating the order and decomposition of necessary transformation operations at design time and the deployment descriptors and interfaces are codified separately from the actual transformations. Our work lacks the richness of an EJB implementation, but for the applications we consider, that penalty is offset by the fact that the actual decomposition and ordering of transformations can be completely rearranged at any time necessary to account for changes in resource availabilities or new client requests, without any coding changes.

We share research goals with DataCutter [13, 15], which delivers client-specific data visualizations to multiple end points using data virtualization abstractions, but there are some key differences with our work. First, DataCutter uses C++ codes for custom filter functions thereby imposing programming tasks on end users. Second, for automatically generated filters, DataCutter uses an SQL-style approach for data selection, whereas we use comparatively richer descriptions for both filter and transformation operations (from which manipulation codes are generated). In addition, in place of a ‘flat’ SQL-like model, XCHANGE end users can manipulate data in the native forms with which they are familiar. This moves end users from having to worry about the mechanics of transforming and using their data to describing the relationships between source data formats and desired sink formats and then just using the data. It also affords us the opportunity to optimize similar requests across multiple clients, by combining common portions of transformations from multiple sinks into a single operation and/or by splitting transformations for distribution to the most appropriate locations on the paths from sources to sinks.

YATL [3] also has explicit transformation descriptions, created as extensions to ML, for query and transformations, but as stated earlier, YATL only operates on XML-formatted data. Our approach permits use of a variety of specification languages, including those built on existing standards like MathML [12] and XML Schema. We also do not limit the source formats of data, which may be XML or any other understandable format such as binary data created on the fly.

Finally, this work and our previous research with ECho [7] share the ability to dynamically install application-specific stream filters and transformers with systems like Infopipes [18].

## 3. XCHANGE Architecture and Implementation

### 3.1 Overview

This section describes how the XCHANGE toolset interfaces with overlay messaging frameworks. Figure 1 depicts a simple overlay network connecting multiple sources to sinks, where the main component of interest is the ‘controller’, which interacts with overlay nodes to install new transformations or change existing ones. The messaging system used in our research is EVPath, a second-generation implementation of the high performance ECho publish/subscribe infrastructure. EVPath is focused on the efficient transport of high-volume or high-rate data streams, where high performance is attained in part by using efficient binary formats for runtime data representation.

These format descriptions are accessible to XCHANGE -generated code. EVPath also offers control methods for overlay management, to dynamically create or delete new nodes, to install new data transformers or filters, and/or to deploy code that changes how messages are routed across the overlay. Scalable methods for overlay deployment and management are described in [14]; they are not relevant to this paper's experimental results. Here, we simply use these methods to experiment with alternative realizations of data manipulations and their mappings to different overlay nodes.

Data transformations in EVPath and XCHANGE involve the following components:

- *Source*. A source system generates a single copy of a data record for downstream consumption, with some well-defined source format.
- *Sink*. Consumes the data in some desired target format.
- *Transformation Description*. Describes the source and sink data formats and the transformation operations necessary to perform the transformation. One of these is required for each sink.
- *Optimizer/Distributor*. A process responsible for collecting the various sink requests, generating the appropriate overlay network, starting, and managing the flow of data.
- *Intermediate Node*. A node in the overlay network capable of hosting some portion of the transformation operations required. An intermediate node is not restricted from residing on the same physical device(s) as the source or sink(s).
- *Controller*. On each node capable of hosting transform(s), potentially including the source and sink nodes, the controller daemon understands the data traffic flowing through the node and the filters and transforms to be applied to each distinct data stream. This is also the component responsible for compiling the transformation description generated for the intermediate form into the EVPath stone physical layout for the node. Once the node has been configured, the controller is not involved in the data path.

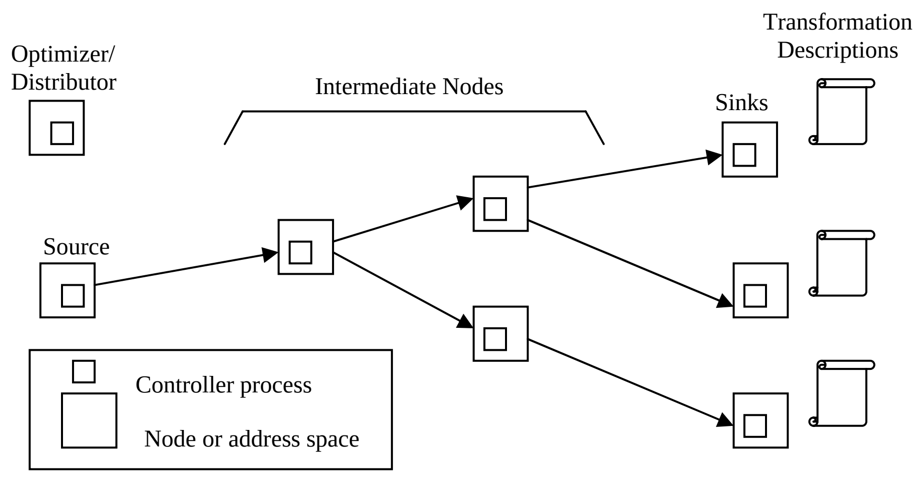


Figure 1: Overlay Messaging with XCHANGE

The experimental configuration used in this paper’s measurements employs a three sink, single source test system, comparing a client-server configuration with sink-hosted and also source-hosted transformations. We assume the availability of adequate intermediate nodes to host the various intermediate-stage transforms. This assumption is not fundamental and only has bearing on the final mapping from a logical deployment graph onto physical nodes.

## 3.2. Technical Overview

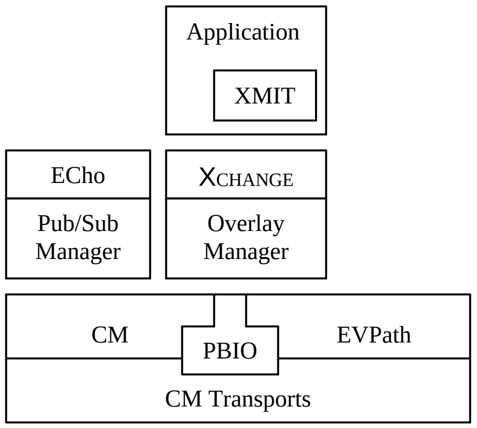


Figure 2: Software Architecture of XCHANGE

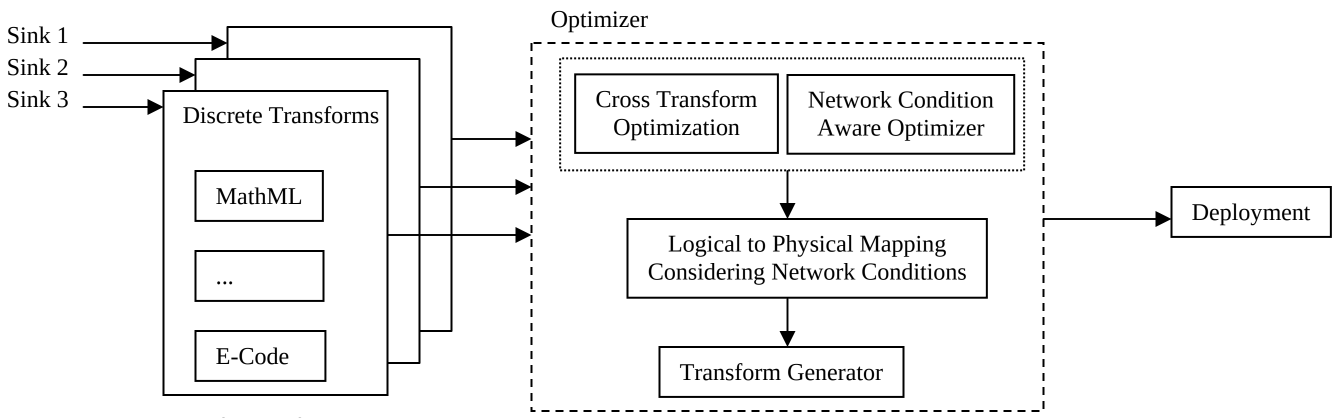
Figure 2 describes the software architecture of EVPath and XCHANGE. The base layer uses EVPath and the PBIO [2] binary data formats, which jointly implement an efficient binary transport of typed data with automatic conversion to local platform formats. In addition to providing the binary transmission format, with XMIT [9], PBIO is also capable of converting data to/from XML on the fly. This expands the scenarios where PBIO can be used effectively to improve communication performance. The CM (Connection Manager) [8] portion manages connections, and EVPath enhances the basic functionality of CM with concrete representations for data manipulations (termed ‘stones’) and with the aforementioned control methods for deploying stones, linking them to other stones, and deleting them [7]. The set of ‘stepping stones’ crossed by formatted data events create dynamic paths across the overlay network, and stones can be linked to form arbitrary graphs. At each stone, filters and translation code can be installed to alter the way in which the data being streamed is routed and manipulated. Data filters and transforms are written in E-Code [10]. Higher-level messaging semantics, like publish/subscribe, are implemented by control methods layered on top of the EVPath infrastructure. We currently support two different sets of semantics, one being publish/subscribe, the other implementing an information flow model. The top layer consists of the end-user application and the XMIT tool [9], which can compile an XML schema [11] into a PBIO data format description and vice versa.

## 3.3 Controller

The first component original to the research described in this paper is the *controller*, which is realized as a daemon installed on all potentially participating nodes in the network. In addition to handling maintenance tasks, this daemon includes a code and intermediate type generator based on the data transformation descriptor provided by the application. Two other interesting components are a centralized *optimizer* that generates the transformation flow graph for all connected clients and a *distributor* that maps the transformation flow graph onto the physical network, installs the transforms and filters on the appropriate nodes, and starts or continues the flow of data from the source to the sinks.

## 3.4 Optimizer

The general architecture of the optimizer and its role in dynamic code generation are depicted in Figure 3.


Sink 1
Sink 2
Sink 3
Discrete Transforms
MathML
...
E-Code
Optimizer
Cross Transform Optimization
Network Condition Aware Optimizer
Logical to Physical Mapping Considering Network Conditions
Transform Generator
Deployment
Figure 3: Overview of Transformation Generation Process

The controlling entity of the framework is the *transformation descriptor* file. That is, each client provides an XML file describing the source and destination formats, using an XML schema for each and a series of declarative transformation descriptions of how to adapt the source data format into the destination data format. The optimizer collects these files to create the composite logical graph of all requests ordering the transformations by merging common subgraphs first and then branching out to more unique transformations. The optimization process consists of finding a path from the source into the network that share transforms with those being deployed. When the transforms assigned to the logical node starting at the root of the graph are a subset of the new transforms being deployed, the procedure can then look for a child node to continue to overlap the existing network. Otherwise, common elements are extracted into a new ancestor node and a new branch is created for the remaining transforms.

Building the logical deployment involves distributing transforms across the various logical nodes. Type safety is maintained by visiting each logical node and ‘fixing up’ the incoming and outgoing data types once the logical deployment is complete. The current implementation uses a simple O(*n*) approach, where n is the number of nodes in the graph. Proper generation of source element propagation information is an important part of this process. Generally, our procedure collects the data elements generated in each node as it traverses down to the leaves and brings back the required source elements as it exits the child node visits. With this process, we can correctly assemble the intermediate types representing both the data elements generated so far and the source elements required by nodes downstream from the current node. The final step in this process is to map the logical to the physical overlay network for proper descriptor generation and deployment.

Since there are many complex steps in our process, we will next discuss some microbenchmarks to examine the potential performance issues and bottlenecks with architectural and design decisions made in our system.

## 4. XCHANGE Implementation - Discussion and Evaluation

This section discusses and evaluates some of the implementation choices made for XCHANGE. The purposes are to position XCHANGE as a set of tools useful for high performance applications and scalable server systems and to identify potential bottlenecks in the XCHANGE implementation.

*Using XML-described Transformations.* A common criticism of using XML for high-end server applications is the relatively high cost of data parsing. XCHANGE does not experience such costs in the data fast path due to its binary representations of data and data transformations. Instead, with XCHANGE, the parsing costs experienced at runtime concern code generation, that is, we must parse the XML-described contents of transformation description files. For our XML parser, we are using the XML C Parser and Toolkit of Gnome (LibXML2) [16] in C++ with the SAX parser. To isolate actual parsing time, we first evaluate the most minimal valid XML for our system. We then evaluate the test examples used for the scientific and healthcare applications described in this paper. For both, results show that parsing overheads are less than 1 microsecond, thereby demonstrating the viability of frequent runtime code generation for long-running data-driven applications.

*Generating logical operator graphs.* Logical graph generation for client requests is a somewhat more complex process. For instance, for two clients, to create a logical graph consists of a single pass extraction of all identical operators between the two requests, using a $n^2$ type algorithm, where $n$ is the number of transforms in a client request. Once this single pass is made, those identical transforms, if any, are assigned to the root node, and the remaining transforms are assigned appropriately to a child node for each of the requesting clients. This second step is an O(1) operation. Similar steps are taken when adding a new client request to an existing logical overlay. First, the $O(n^2)$ algorithm extracts identical operators using the root node of the graph and the new client request. The next assignment step is an O(1) operation when no transforms overlap or when only some of the root node's transforms overlap. If all of the root node's transforms overlap, child nodes are evaluated for possible overlaps, followed by an appropriate assignment process including a potentially recursive call, if needed. Since this recursive process can become expensive, several optimizations have been implemented. First, we check the type of the transform and then check the component pieces for a match, thereby short-circuiting the process whenever there is mismatch. Second, since we store the transforms in some parsed format, we do not need to reparse the transforms for each comparison or rely on a problematic textual comparison to find identical transforms. The resulting graph generation cost is $O(mn^2)$, where *m* is the depth of the logical graph and *n* is the number of transforms in the client request. In practice, unless there are large numbers of similar transforms and a large number of clients, this process ends after a few iterations. This is also the case for measurements with the actual transformations used in the applications described in this paper, which exhibit graph generation times of less than 1 microsecond for each case.

*Mapping operator graphs to the overlay and to physical machines.* Using the mapping algorithm described in [14], initial deployments of logical graphs to overlays across 10s of machines can be carried out in timeframes ranging from 200 to 800 milliseconds. Steps taken involve the transmission of XML layout descriptors to overlay controllers, the parsing and compilation of the XML down to machine appropriate code, and the assembly of the intranode data flow and transforms from the incoming stream to each of the outgoing streams.  Since our transforms have all been generated during the logical and physical mapping steps, we have little more than a simple translation operation to change our XML descriptions into PBIO schemas and ECL. The cost of creating and linking stones and deploying and having the ECL compiled for the transforms is sufficiently small that all of our testing examples showed a << 1 microsecond duration for any of our test deployments.

*Summary.* Micro-benchmarks show that by far, the slowest part of the system is deployment. Based on our experiments in [14], this time grows roughly linearly with the number of nodes. We next describe the applications built and evaluated with XCHANGE, followed by explanations of some of the optimizations afforded by its use.

## 5. Applications and Usage

### 5.1. Scientific Collaboration

The SmartPointer framework for online scientific collaboration [1] is used to evaluate XCHANGE utility and performance. A specific configuration of SmartPointer is one in which the framework is used to generate visualizations of certain molecular dynamics processes for different clients. A small depiction of molecular data uses a 640x480x3 byte color image as source data. The two clients each require different image formulations. The first client has limited bandwidth and display capabilities and desires a 320 by 240 by 1 byte grayscale image, where the conversion to grayscale is performed by the equation $p = (R \times 0.299 + G \times 0.587 + B \times 0.114) + 0.5$. The second does not really care about color and would rather be sure there is sufficient bandwidth to handle the data load, thus requiring a 640 by 480 by 1 byte grayscale image instead of the source format. The third client desires a full color, full size image.

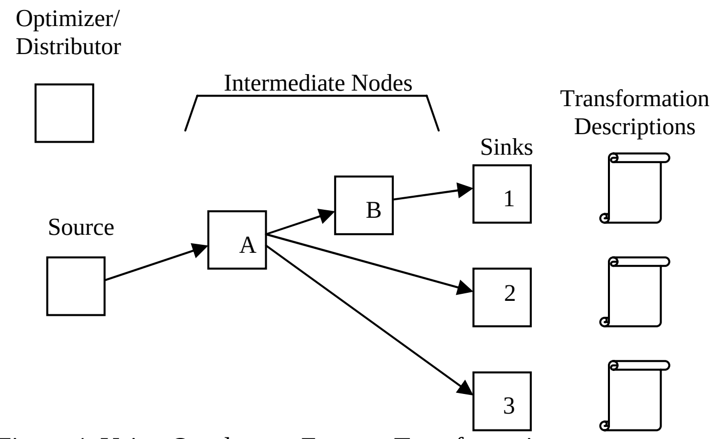


Figure 4: Using Overlays to Execute Transformations

In the above diagram, Sink 1 has its transformations split across two nodes. Sink 2 has its single transformation on one intermediate node. Sink 3 does not need any transformations and just extracts a copy of the original data stream. The optimizer recognizes the identical grayscale conversion for Sinks 1 and 2 merges the two operations. For comparison, measurements are also taken when all of the transformations are hosted on the source and separately, when each sink hosts its own transformations.

To generate a broader understanding of the impact of data volume, we vary the example to include different sized source images. In additional tests, we manually scale all image sizes up by 50% and 100% and also down by 50% to compare the relative savings our approach offers in each of the four scenarios. To maintain a consistent comparison base, we also scale the client requests similarly for each test.

## 5.2 Healthcare

Consider a general hospital infrastructure consisting of a wide variety of specialized departmental systems and more general systems for cross-departmental use [39]. These systems exchange information through event messaging, such as an admission event, ordering a lab test, or the emission of a set of lab results. Each departmental system maintains a private data repository tracking patient state information to guide data retention and messaging operations. Since hospitals typically buy specific systems for each department, a typical installation is comprised of a large number of different vendor products. In response, the HL-7 [26] standard for healthcare messaging codifies the format and content of inter-system messages. For cross-platform compatibility, the format is text-based with markers to separate variable width fields. Issues with the standard and its use include the following. First, its text format incurs parsing overheads that could be overcome with alternate binary representations. Second, each message generated by some system will almost certainly need to be translated to be understood or be usable by the receiving system, constituting another potential advantage of a binary representation. Third, since HL-7 is sufficiently flexible, occasionally ambiguous, and sometimes improperly implemented, the development of custom interfaces is typical, as evidenced by multiple companies providing them [27, 28, 29]. Finally, for all such tasks, budget limitations constrain the fashion in which response times can be kept sufficiently low with rising patient data volumes.

One sample scenario consists of a typical hospital with ADT, transactional repository, order management, laboratory management, and medical imaging, among other systems. The ADT system is used to track admissions, discharges, and transfers among hospital locations for patients. Each ADT event must be copied to a transactional repository for per-patient, general use and to various departmental systems that need to know patient status or other patient attributes. For example, when a patient must have some laboratory tests performed, basic patient identification information must be available in the laboratory system to be able to link the results of the tests back to the patient records. Some patient attributes like pregnancy status may also be required in order to generate accurate results. Data generation events like ADT messages happen for each patient event or activity. Some of these include medical imaging, dietary orders, respiratory treatments, and pharmacy dispensing. By applying our system in this domain redundant calculations across departments can be consolidated into the overlay network reducing compute load across all affected departments.

One key area where healthcare applications could benefit from our system is clinical decision support systems [49]. Image differencing techniques combined with discoveries in diagnosis have combined to afford the opportunity to provide automatic advanced warning of potentially serious conditions without physician involvement. By performing trending across medical images, developments of areas of interest can be noted and flagged for physician review [50]. This assists the physician by alerting them to the possibility of an abnormal condition as quickly as can be determined rather than waiting for a physician to decide to look for the possibility of a particular condition. With the constant lowering of reimbursements for medical procedures, the hospital system is most interested in these systems. Through early detection of conditions, cheaper remedies may be able to be employed reducing the costs to the hospital generating a higher reimbursement percentage for the care provided. The image transfer savings using our framework have been established in [56]. Evaluating the image analysis portions of this scenario requires complex image analysis algorithms that are beyond the scope of this paper.

Our approach is to apply the XCHANGE framework to manipulate data in binary forms thereby removing parsing and reducing transformation costs and distribute the data transformation costs along the transmission paths. Two concrete examples are described in this paper. The first represents a physician ordering a complex lab test to generate panels of multiple results each generated from testing a few different specimens. For example, a complete metabolic panel consists of approximately 15 discrete values generated from separate urine and blood specimens. The lab system is configured to run a particular set of tests for each order based on the specimen type. It matches the specimen accession number to the electronically transmitted lab order and performs the required tests. The results are sent back as a list of components to the ordering system for further processing. In order for the results to be useful to the physician, they require to have units adjusted, new composite results generated by combining some of the discrete values, and separating them into subpanels. All of this work is performed to transform data into something the transactional system can understand. The second example is the calculation of the Body Mass Index (BMI) [42]. While the calculation itself is not complex [43], it may need to be done across three or more departments in the hospital. The dietary department uses it to help determine the type of diet to provide to the patient [44]. Respiratory uses it as part of the calculation of respiratory functions [45]. The pharmacy uses it to determine appropriates or the dosages for prescribed medicines [46]. Large, multi-facility hospital systems can have 30-50 or even more individual hospitals and outpatient facilities [47, 48]. Savings can be attained by centralizing all of the IT systems for all of these facilities. In these scenarios, the frequency of the BMI calculations increases beyond a trivial amount of CPU through volume alone. By employing our system, we can pre-calculate this value once on the fly as the data is transferred to all of these departments, thereby freeing the specialized departmental systems to focus on the core tasks they are designed for.

Using the XCHANGE framework offloads message transformation processing from the primary systems. The first experiment consists of the transmission of a set of 16 components. The receiving system requires components to be decomposed into 3 panels with 5 components each. One of the components needs to have the units scaled to a range more familiar to physicians, and one of the components will be the aggregate of two components with a simple multiplicative relationship of ($Z = Y \times X$). Other domains have analogous examples. In engineering simulations, each simulation component typically requires adjustments of its output for it to be suitable as inputs for other simulations. Sample adjustments include dealing with error tolerances, unit translations, format changes, and others. The second experiment consists of calculating the BMI on an ADT message transmission. For simplicity, we have stripped a typical ADT admission event down to some relevant fields. We test simulating volume to show we can achieve savings in a high message volume environment.

## 5.3 Describing Transformation Operations

The transformation generation process shown in Figure 3 uses XML-based descriptions, where each transformation description is an XML node named “assign” consisting of a “dst” and a “src” element. The “dst” element describes the destination format in which the result of the transformation should be stored. This may be either a simple item or a compound item to describe a path through the type “tree” to the element. For an array or block element, this is likely just the name of the array or block rather than a particular element. The “src” element consists of one of several items describing both the element(s) from which the destination element is formed and the operation to apply. For example, a 50% reduction in height and width of a 2-dimensional array of, say, a bitmap

```
<assign>
  <dst>
    <compound>
      <item name="pixel" subscript="x"/>
      <item name="line" subscript="y"/>
    </compound>
  </dst>
  <src>
    <reduce factor="2">
      <compound>
        <item name="pixel" subscript="x"/>
        <item name="line" subscript="y"/>
      </compound>
    </reduce>
  </src>
</assign>
```

image (25% of original data size), could be achieved by using a “reduce” element listing the factor and the source element(s). The code generation at the intermediate node daemon understands how to generate the proper looping and other control constructs to apply the operation to the range of elements specified. The subscript attribute of the item node represents how to apply the mapping of the elements. This could be used to perform a transposition of a matrix.

For formulaic math operations, the MathML [12] standard is employed. This example shows a Body Mass Index (BMI) calculation.

```
<assign>
  <dst>bmi</dst>
  <src>
    <apply>
      <times/>
      <apply>
        <divide/>
          <ci>weight</ci>
          <apply>
            <times/>
            <ci>height</ci>
            <ci>height</ci>
          </apply>
      </apply>
      <cn>703</cn>
    </apply>
  </src>
</assign>
```

By default, all elements referenced in a “src” node are assumed to be from the incoming data stream. As with the MathML standard, the ci tag represents an identifier. We use our type navigation syntax within that element to navigate to the proper data member for that portion of the formula. We would also use that structure if we were to need a previously calculated destination element as part of the new calculation. In that case, we would add an attribute of the compound element called “src” to be of the value “dst”. To handle the non-compound cases, we also have a “simple” element that can host the “src” attribute as well.

We have also added appropriate attributes for element references to determine whether the source elements are from the original source type or from a derived item generated from a previous transform.

The deployment of the generated transformation flow graph onto appropriate nodes currently uses a simple latency metric to determine appropriate nodes. More complex and efficient deployment algorithms have been studied elsewhere [14] and are beyond the scope of this paper.

## 6. Experimental Evaluation

To evaluate the efficacy of XCHANGE, two different scenarios are examined using two different metrics. The first scenario emulates the scientific applications implemented with SmartPointer, by evaluating the costs associated with data transport (and conversion) from a single source to three sinks. The second scenario is representative of event message flows in a hospital’s computing infrastructure, using a single source and a single sink. Two different metrics are used, one focusing on the network resource, the other considering processing overheads.

The scientific application is driven by limitations in network bandwidth. Such limitations are due to both the large data volumes that commonly occur and the fact that an inappropriate implementation would result in the source data volume becoming the union of all potential client needs. The metric measured is end-to-end performance, since online collaboration is not feasible without reasonable delay constraints on the data streams being produced, transported, and displayed. An important contribution to performance is the CPU load incurred to encode, transform, and decode messages. In both the scientific and healthcare scenarios, the end node machines may not be able to deliver services to meet the needs of multiple clients. The approach taken in our research includes offloading processing to intermediate nodes.

Measurements are performed on an IBM BladeCenter with 14 HS20 blade servers installed (hostnames awing1-awing14). Each blade has dual 2.8GHz Xeon processors with 1GB RAM and a 1 Gb/s NIC card running RH Linux 9.0. Source, intermediate, and sink nodes participating are each hosted on a different blade. Our base testing setup is sufficiently fast that for our timed tests, a single test step took less time than the quantum of 1 microsecond measurable on this system. To compensate, we ran 100,000 iterations and divided the total time by 100,000 to gain a representative per-iteration time.

## 6.1 Scientific Example:

This example demonstrates the importance of being able to dynamically generate code based on the locations of transforms, as determined by the resources available at runtime. If the source node can perform all data morphing and if there are sufficient network resources, then the best solution is to place morphing at the source. Client-hosted data morphing has the greatest impact on network resources. By being able to select where to place the code generated from the transforms and how to order them, we can adjust data morphing to avoid consumption of the currently scarcest resource, i.e., for source, overlay nodes, sinks, and the networks connecting them.

We could have achieved similar performance results with different technologies, but at several costs. First, current technologies like EJBs are capable of creating a system decomposed as we do here, but the writer is limited to the static decomposition determined at design time. Our approach affords decomposition decisions and code generation both at runtime and with optimization that consider the efficiencies desired based on current conditions. Second, if a client must perform data morphing, its end-user application must understand the source data format and perform the transform operations. Our approach offloads this non-core task from such applications and from programmers and provides performance gains as outlined below. Third, if the transforms were to be placed on the source, we would be limited by the available resources on the server. Our approach affords the advantages of source-based dynamic data morphing as also supported by systems like ECho, but surpasses them by supporting automatic code generation and deployment to other nodes along the network path in the overlay network. Our approach is more flexible than any of the static or even the dynamic approaches traditionally used and affords the opportunity to adapt the code deployed based on the limiting resources in the system.

Measurements of the network bandwidths required for each of source, sink, and overlay hosting of the data morphing code are presented below.

XCHANGE generates the following graph automatically:

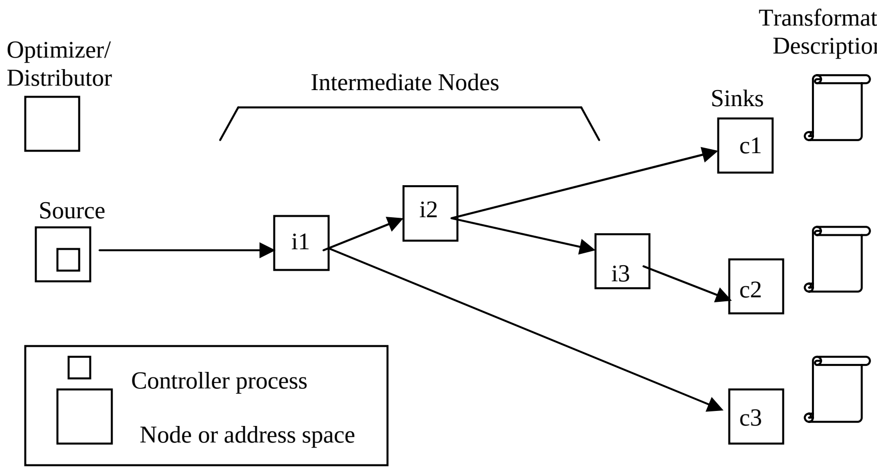


Figure 5: Generated Graph Deployment

For this set of experiments, XCHANGE is configured to only morph data on intermediate nodes. The nodes labeled i1, i2, and i3 will have the code generated from the various transforms installed. The items in the graphs below correspond to the links pictured in this graph.

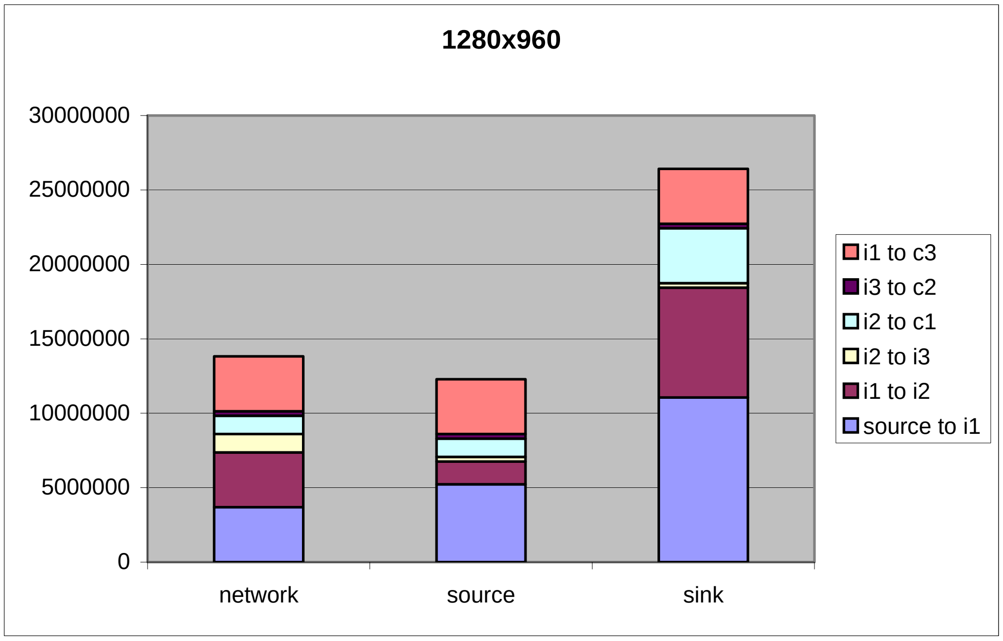


Figure 6: Scientific Example Performance Graph for a 1280x960 Image Size

The same relative outcomes of all experiments support the claim that XCHANGE can generate efficient service overlay networks. This graph showing the 1280x960 image size tests is representative. The network hosted model for dynamically generated data morphing code is nearly as bandwidth efficient overall as the source hosted model. Of particular note is that the "last mile" portion of the transmission has been offloaded from both ends, moving the bottlenecks into the network infrastructures with have generally better bandwidths. Using techniques from the high performance domain, node loads like a tree fan-out [57] or similar techniques, we can reduce these bottlenecks further. One proposed future investigation would evaluate schemes for determining the efficacy and appropriate conditions to duplicate intermediate nodes to further reduce single link bottlenecks.

## 6.2 Healthcare Examples:

Our high performance claim has been validated with these examples. We demonstrate that we can offload the source and sink systems by hosting these operations in the overlay network without impacting the end-to-end performance. We measure the end-to-end time for each scenario and observed no measurable difference between hosting the code generated from the transforms in the network and hosting them later on the client. The details of these tests are included below.

Our first healthcare example is the lab results reconfiguration. This test uses three nodes: a lab system (source), an intermediate node (messaging hub), and a sink (transactional system). Our experiments demonstrate that there was no difference in the end-to-end time (2.00 seconds for 10000 events) or in the volume of data hosted among any of our models. In this case, it is most important for us to be able to maintain or exceed the performance rather than having strictly better metric results. In these scenarios, the volume of data is rarely significantly different between the source and sink models. Our goal is to offload computing work with little to no impact rather than reducing what is transmitted. Measurements demonstrate our ability of XCHANGE to move operations into the network with no impact on end-to-end latencies.

The second healthcare example is the calculation of the Body Mass Index (BMI) as part of an admission event. Different from the lab example above, we choose to simulate the possible event load for a busy night across all hospitals in a large provider network. We consider a volume of 600 patients [51] across 50 hospitals for a total of 30000 events. To generate the events as quickly as possible, three source systems feed a single intermediate node that propagates all results to three different departmental systems. Measurements show no discernible differences in end-to-end latency when hosting the calculation on the intermediate vs. the destination node. We observe that all 30000 events are processed in 1.03 seconds when hosting it on the server. When hosting on the intermediate node or on the sink, we reduce the time to 0.85 seconds in both cases, a 17% savings. Therefore, by pulling the calculation from the server into the overlay network, the time to transmit events and the CPU load on the are reduced.

## 6.3 Results Summary

Experimental results demonstrate that the dynamic, resource-aware generation and deployment of data morphing code can improve network resource utilization and result in greater scalability, with no negative effects on end-to-end latency. The scientific example demonstrates the ability of XCHANGE to reduce the per link resource costs and/or adapt to the available CPU resources, as appropriate for current conditions. Unlike the source or sink hosted models, we achieve the flexibility to respond to conditions with the most appropriate deployment of the generated code to achieve the lowest resource usage. Results show a 64% savings over sink hosted transforms and a 14% savings over source hosted transformations. The healthcare example again validates these results. Both sets of results demonstrate that there is no impact on end-to-end performance when hosting data morphing code in the network as compared to a sink hosted model. For the BMI calculation example, by hosting morphing code in the network, we demonstrate a 17% improvement over the source hosted model.

## 6.4 Discussion of Experimental Results

Our experiments highlight the three core contributions of the paper. First, code generation costs are small, particularly when compared to deployment delays. That is, initial deployment performance dominates generation costs by taking 100s of milliseconds for 10s of nodes. Second, it is important to dynamically generate the code being deployed, as demonstrated by the scientific example. It shows high variability in network link costs for hosting generated code on the source, in the network, or on the sink. By evaluating current resource availabilities, code generation can change how morphing code is organized and then deploy the generated code differently to reduce the impact on the most constrained resource. The healthcare examples show that generated code can be deployed to the sink or into the network, without impacting end-to-end performance. This flexibility is afforded by the ability of XCHANGE to perform resource-sensitive dynamic code generation. Third, the performance of XCHANGE is shown suitable for a production system. The science example demonstrates the advantage of adapting generated code to the highest performance configuration, given current resource availability. The healthcare example shows that we can adapt the deployment of generated code to use the best end-to-end performance model. Below, we detail some of the specific features that contribute to the success of our approach.

One of the key features of this work is that code generation is combined with the dynamic placement of transformation operations into an overlay network between sources and sinks. In comparison, a sink-hosted data transformation may have undue client requirements, and it can require substantial network bandwidths. Source-hosted transformations may not be feasible due to CPU limitations or just because of the sheer number of clients. Additional advantages of our approach of combining the generation of transformation code with code deployment are gained through the combination of identical operations. Specifically, results attained with the SmartPointer application (see Section 5) demonstrate a 14% reduction in data traffic compared to a source-hosted approach and a 64% reduction compared to a sink-hosted approach for a single source/two sink configuration. For our healthcare messaging example, we observe no end-to-end performance penalties when deploying this structure.

The base example shown above assumes a network bandwidth constrained environment. If this is not the case, a different optimization strategy can be employed. Evaluating the CPU cost, and network bandwidth impact of transforms (neutral, reduces data volume, or increases data volume), or other factors, different optimization strategies can be employed in the generation of the logical graph and the physical mapping. As described above, we choose to limit the granularity of our optimizations at this point. As our first pass, we perform simple full common expression elimination rather than the more complex common subexpression elimination. Our distributed transforms and the requirement to properly generate intermediate types add some additional complexity we have chosen not to explore at this time. Other possibilities like common subexpression elimination, operation reordering, dead code elimination, and other techniques common from the compilers field could also be employed. We have evaluated some other useful optimization approaches and techniques in [14].

Another key feature of this work is the ability to add new clients at runtime and merge them into a running overlay network. Once the initial deployment of the system has been performed, a new client can add itself to the data stream in a fashion similar to how the initial clients registered for a transformed version of the data. In this case, the previous logical mapping has the new request merged into it, noting which nodes have been modified. Next, the logical to physical mapping can determine which nodes are affected by the new request. Rather than interrupt the current running system, the new network of nodes is deployed in parallel, setting up the full subscription model. Once it has been fully deployed, the source node has its subscribers updated, sending the data over the new overlay structure. At this time, we do not handle the transfer of any privately stored state in a transform function. Future extensions to this work will handle such stateful cases.

As a practical example, consider the SmartPointer application described above. Consider the case where the system is started with two clients, the full resolution feed and the greyscale version. After some time, a third client with very limited bandwidth wishes to see the visualization and asks for a greyscale, 50% image reduction version. The optimizer/distributor will evaluate the impact on the previous logical graph noting any node changes, check the mapping to the physical deployment to discover which nodes need to be updated, deploy the new transforms setting up the appropriate channels, and start the feed over the new overlay network. Once the feed has begun, the distributor can decommission the now unused transform deployments. Detailed evaluation of this process will be presented in a future paper.

## 7. Conclusions and Future Work

As this paper has shown, our approach of performing dynamic code generation for transforms considering resource availability and performing even rudimentary optimizations like merging common operations can lead to significant performance improvements over static approaches. In our tests, we measure a 64% improvement compared with sink hosted data transformations and a 14% improvement compared with source hosted transformations. For our end-to-end measurements using our healthcare example, we observe that it is possible to host operations in the overlay network with either no impact or a 17% improvement in the end-to-end time compared with a source hosted model. At a micro-benchmark level we note that merging in new clients is the slowest overall step. As this step occurs outside of the data path and is still measured at < 1 microsecond (ignoring deployment), such runtime configuration is clearly viable. As we scale, we note that performance will degrade based on the number of transforms used in each client and the depth of the shared transform tree.

Our results are also in congruence with those from message passing in parallel systems [41]. By coordinating over various clients, we can reduce the impact of adding more clients to the composite system.

The future directions of this research are many. First, different deployment schemes accounting for performance metrics such as CPU availability, network bandwidth availability, and node locality can be developed and evaluated. We will also investigate how to make incremental changes to an existing deployment. For example, it is interesting to determine different workable approaches for adding new clients to an existing deployment, with minimal impact on the currently operating application. Another would be to determine some metrics to gauge at what time an incrementally deployed network is sufficiently less efficient than a newly deployed network to justify redeploying. When redeploying, developing guidelines on how to handle state when migrating operations from one node to another. Both of these ideas assume that we do not need to move transforms from one node to another. Considering how to do these operations on a running system and moving transforms adds another set of challenges to investigate.

As the complexity of the available operators increases, the dependencies may grow. Investigating issues around how to optimize graphs of operations based on desired performance characteristics is important. These would include considerations of how and when to duplicate nodes that would normally be merged in order to generate a more optimal data flow. For example, when there is a sufficiently large bottleneck for a particular node, it may be worth duplicating the transforms on another node and splitting the traffic over the multiple nodes.

One area where we have done some initial work is to evaluate doing low-level common subexpression elimination rather than performing the optimizations at the full expression level. This has already been done to some degree, but using a SQL-style data model in Gryphon [40]. Other typical compiler techniques for optimization will also be explored.

An interesting related problem concerns how to integrate the requests of several clients from multiple sources all participating in the same data stream. Adapting and extending current work on multicast systems would be required.

Related to the previous item is the issue of multiple similar sources serving a set of clients. Determining how and when to combine operators to form a graph with disparate sources feeding into a network that ultimately connects with the various sinks. This is important for cases where data is stored in multiple formats that are needed by several clients. One possible approach may be to meld the two data streams into a coherent model.

## 8. References

[1] M. Wolf, Z. Cai, W. Huang, and K. Schwan, "SmartPointers: personalized scientific data portals in your hand," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Supercomputing '02)*: IEEE CS Press, 2002, pp. 1-16.

[2] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener, "Efficient Wire Formats for High Performance Computing'", *ACM Supercomputing 2000*, (SC 2000), Nov. 2000.

[3] S. Cluet, S. Jacqmin, and J. Simeon. "The New YATL: Design and Specifications." Technical Report, INRIA, 1999.

[4] Suman Banerjee, Christopher Kommareddy, Koushik Kar, Bobby Bhattacharjee, Samir Khuller, "Construction of an efficient overlay multicast infrastructure for real-time applications," IEEE INFOCOM 2003 - The Conference on Computer Communications, vol. 22, no. 1, Mar 2003 pp. 1521-1531.

[5] Eriksson, H., "MBONE: The Multicast Backbone," Communications of the ACM, Vol. 37, No. 8, 1994, pp. 54-60.

[6] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Rob Strom, Daniel Sturman, and Wei Tao. "Information Flow Based Messaging Middleware for Building Loosely Coupled Distributed Applications." *Proceedings of the Middleware Workshop held in conjunction with the Int'l Conference on Distributed Computing Systems (ICDCS),* Austin, May 1999.

[7] Greg Eisenhauer, Fabian Bustamente and Karsten Schwan. "Event Services for High Performance Computing," *Proceedings of High Performance Distributed Computing (HPDC-2000).*

[8] Greg Eisenhauer. "The Connection Manager Library". August 17, 2004. Unpublished.

[9] Patrick Widener, Greg Eisenhauer, and Karsten Schwan. "Open Metadata Formats: Efficient XML-Based Communication for High Performance Computing." *Proc. Tenth IEEE International Symposium on High Performance Distributed Computing-10 (HPDC-10),* San Francisco, California, August 6-9, 2001.

[10] Greg Eisenhauer. "Remote Application-level Processing through Derived Event Channels in ECho," Unpublished.

[11] W3C XML Schema. http://www.w3.org/XML/Schema.

[12] W3C MathML. http://www.w3.org/Math.

[13] Michael Beynon, Renato A Ferreira, Tahsin M Kurc, Alan Sussman, Joel H Saltz, "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems", *Eighth NASA Goddard Conference on Mass Storage Systems and Technologies/Seventeenth IEEE Symposium on Mass Storage Systems*, 2000, pp. 119-133.

[14] Vibhore Kumar, Brian F Cooper, Zhongtang Cai, Greg Eisenhauer, Karsten Schwan. "Resource-Aware Distributed Stream Management using Dynamic Overlays." *25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005),* Columbus, Ohio, USA.

[15] Weng, L.; Gagan Agrawal; Catalyurek, U.; Kur, T.; Narayanan, S.; Saltz, J. "An approach for automatic data virtualization," High performance Distributed Computing, 2004. *Proceedings. 13th IEEE International Symposium on*, Vol., Iss., 4-6 June 2004 Pages: 24-33.

[16] XML C Parser and Toolkit. http://xmlsoft.org/.

[17] Manuel Rodríguez-Martínez , Nick Roussopoulos, "MOCHA: a self-extensible database middleware system for distributed data sources", *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, p.213-224, May 15-18, 2000, Dallas, Texas, United States.

[18] C. Pu, K. Schwan, and J. Walpole. "Infosphere project: System support for information flow applications." *ACM SIGMOD Record*, 30(1), March 2001.

[19] Rodriguez-Martinez, M.; Roussopoulos, N.; McGann, J.M.; Kelley, S.; Mokwa, J.; White, B.; Jala, J.; "Integrating distributed scientific data sources with MOCHA and XRoaster." Scientific and Statistical Database Management, 2001. SSDBM 2001. *Proceedings of the Thirteenth International Conference on Distributed Data Management,* 18-20 July 2001 Pages: 263 – 266.

[20] S Babu, J Widom "Continuous Queries over Data Streams." *SIGMOD Record* 30(3):109-120 (2001)

[21] D Carney, U Cetintemel, M Cherniack, C Convey, S Lee, G Seidman, M Stonebraker, N Tatbul, S Zdonik. "Monitoring Streams: A new class of data management applications." *In Proceedings of the 27th International Conference on Very Large Databases*, Hong Kong, August 2002.

[22] R. Koster, A. Black, J. Huang, J. Walpole, C. Pu. "Infopipes for composing distributed information flows." *Proceedings of the 2001 International Workshop on Multimedia Middleware*. Ontario, Canada, 2001.

[23] Arcot Rajasekar, Michael Wan, Reagan Moore, Wayne Schroeder, George Kremenek, Arun Jagatheesan, Charles Cowart, Bing Zhu, Sheau-Yen Chen, Roman Olschanowsky. “Storage Resource Broker - Managing Distributed Data in a Grid,” *Computer Society of India Journal, Special Issue on SAN*, Vol. 33, No. 4, pp. 42-54 Oct 2003.

[24] Weinberg, J.; Arun Jagatheesan; Ding, A.; Faerman, M.; Hu, Y.; “Gridflow description, query, and execution at SCEC using the SDSC matrix”. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing,* 4-6 June 2004 Page(s):262 – 263.

[25] Jeyarajan Thiyagalingam, Olav Beckmann and Paul H. J. Kelly. “Is Morton layout competitive for large two dimensional arrays, yet?” To appear in Concurrency And Computation: Practice and Experience (special issue on Compilers for Parallel Computing, accepted February 2004).

[26] Health Level Seven. http://www.hl7.org.

[27] Healthcare Communications, Inc. Cloverleaf. http://www.healthcare.com/.

[28] SeeBeyond. e*Gate. http://www.seebeyond.com/

[29] MDI Solutions. MD Link. http://www.mdisolutions.com/

[30] Clayton, J.D. and McDowell, D.L., “A Multiscale Multiplicative Decomposition for Elastoplasticity of Polycrystals,” *International Journal of Plasticity*, Vol. 19, No. 9, 2003, pp. 1401-1444.

[31] Bao, H., Bielak, J., Ghattas, O., O’Hallaron, D., Kallivokas, L., Shewchuk, J., and Xu, J. “Earthquake ground motion modeling on parallel computers.” *In Proceedings of Supercomputing ’96* (Pittsburgh, PA, Nov. 1996).

[32] S. Madden and M. J. Franklin. “Fjording the stream: An architecture for queries over streaming sensor data.” In *ICDE Conference,* 2002.

[33] Ada Gavrilovska, Sanjay Kumar, and Karsten Schwan. “The Execution of Event-Action Rules on Programmable Network Processors.” *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure* (OASIS 2004), held in conjunction with ASPLOS-XI, Boston, MA, Oct. 2004.

[34] Radhakrishnan, S. “Speed Web delivery with HTTP compression: A look at the page-delivery effects of data compression in HTTP 1.1”. *IBM Developer Works*. 22 July 2003. http://www-106.ibm.com/developerworks/web/library/wa-httpcomp/

[35] Dong Zhou, Karsten Schwan, Greg Eisenhauer, Yuan Chen . “JECho - Interactive High Performance Computing with Java Event Channels,” *in the Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS 2001),* April 2001.

[36] Zhongtang Cai, Greg Eisenhauer, Christian Poellabauer, Karsten Schwan and Matthew Wolf. “IQ-Services: Resource-Aware Middleware for Heterogeneous Applications,” *Proc. of the 13th Heterogenous Computing Workshop (HCW 2004)*, invited paper, April 2004, Santa Fe, NM.

[37] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. “Cluster-based scalable network services.” *In Proceedings of the Sixteenth ACM Symposium on Operating System Principles,* San Malo, France, Oct. 1997.

[38] Enterprise Java Beans. http://java.sun.com/products/ejb/.

[39] Hasselbring, Wilhem. “Extending the Schema Architecture of Federated Database Systems for Replicating Information in Hospital Information Systems,” *Engineering Federated Database Systems EFDBS ’97*, June 1997, pp 33-44.

[40] Banavar, Guruduth, et. al. “Information Flow Based Event Distribution Middleware,” *Electronic Commerce and Web-based Applications/Middleware*, 1999, pp 114-121.

[41] Ming ZHU, Wentong CAI, Bu-Sung LEE. “Communication Optimization in Network-based Parallel Computing” *Second International Conference on Information, Communications & Signal Processing* (ICICS'99), 7-10 December 1999, Venue: Hotel Mandarin Singapore, Singapore.

[42] BMI – Body Mass Index: Introduction | DNPA | CDC. Retrieved October 15, 2005 from http://www.cdc.gov/nccdphp/dnpa/bmi/index.htm.

[43] BMI - Body Mass Index: BMI Formula for Adults | DNPA | CDC. Retrieved October 15, 2005 from http://www.cdc.gov/nccdphp/dnpa/bmi/bmi-adult-formula.htm.

[44] New Jersey State Act Enforcement Letter. Retrieved October 15, 2005 from http://www.state.nj.us/health/hcsa/hospfines/ber070104.pdf.

[45] De Lorenzo A, Petrone-De Luca P, Sasso G, Carbonelli M, Rossi P, Brancati A: “Effects of Weight Loss on Body Composition and Pulmonary Function. Respiration” 1999;66:407-412 (DOI: 10.1159/000029423) [http://content.karger.com/ProdukteDB/produkte.asp?Aktion=ShowPDF&ProduktNr=224278&Ausgabe=226555&ArtikelNr=29423&filename=29423.pdf]

[46] The Legal Practice of Pharmacy in Ohio. http://pharmacy.ohio.gov/leglprac-040901.htm

[47] Triad Hospitals, Inc. http://www.triadhospitals.com/

[48] Hospital Corporation of America (HCA). http://www.hcahealthcare.com/

[49] Amit X. Garg; Neill K. J. Adhikari; Heather McDonald; M. Patricia Rosas-Arellano; P. J. Devereaux; Joseph Beyene; Justina Sam; R. Brian Haynes "Effects of Computerized Clinical Decision Support Systems on Practitioner Performance and Patient Outcomes: A Systematic Review" *JAMA 2005* 293: 1223-1238

[50] Georgia D. Tourassi. "Journey toward Computer-aided Diagnosis: Role of Image Texture Analysis." *Radiology 1999* 213: 317-320.

[51] Private conversation with Cheryl Bronczyk, McKesson Implementation Project Manager October 24, 2005.

[52] Zhongtang Cai, Greg Eisenhauer, Qi He, Vibhore Kumar, Karsten Schwan, Matthew Wolf. "IQ-Services: Network-Aware Middleware for Interactive Large-Data Applications." *2nd International Workshop on Middleware for Grid Computing, MGC-2004.* Toronto, Canada.

[53] Greg Eisenhauer, Fabian Bustamante and Karsten Schwan. "A Middleware Toolkit for Client-Initiated Service Specialization." *Principles of Distributed Computing (PODC 2000) Middleware Symposium*, July 18-20, 2000.

[54] Wiseman Y., Schwan K. & Widener P. "Efficient End to End Data Exchange Using Configurable Compression" *Proc. The 24th IEEE Conference on Distributed Computing Systems* (ICDCS 2004), Tokyo, Japan, pp. 228-235, 2004.

[55] Abbasi H., Wolf M., Schwan K., Eisenhauer G., Hilton A.. "XCHANGE: Coupling Parallel Application in a Dynamic Environment" *Proceedings of Cluster 2004,* San Diego, CA, 2004.

[56] Himanshu Raj, Karsten Schwan, Ripal Nathuji (2005). "M-ECho: A Middleware for Morphable Data-Streaming in Pervasive Systems." *Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services,* 2005, Seattle, WA.

[57] Ron Brightwell and Lee Ann Fisk. "Scalable Parallel Application Launch on Cplant." *Proceedings of SC2001*. November 10-16, 2001, Denver, Colorado.

[58] V.Oleson, K.Schwan, G.Eisenhauer, B.Plale, C.Pu, D.Amin. "Operational Information Systems - An Example from the Airline Industry." *First Workshop on Industrial Experiences with Systems Software WIESS 2000,* October 2000.

[59] R. Ferraro, T. Sato, G. Brasseur, C. Deluca, and E. Guilyard, “Modeling the Earth System,” presented at *International Geoscience & Remote Sensing Symposium* (invited paper), 2003.

[60] A. Dai, A. Hu, G. A. Meehl, W. M. Washington, and W. G. Strand, “North Atlantic Ocean circulation changes in a millennial control run and projected future climates,” *Journal of Climate*, (in press).

[61] M. Parashar, H. Klie, U. Catalyurek, T. Kurc, V. Matossian, J. Sltz, and M. F. Wheeler, “Application of Grid-Enable Technologies for Solving Optimization Problems in Data-Driven Reservoir Studies,” presented at *4th International Conference on Computer Science* (ICCS 2004), Krakow, Poland, 2004.

[62] P. R. Pietzuch and J. M. Bacon. Hermes: “A Distributed Event-Based Middleware Architecture.” In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems* (DEBS'02), pages 611-618, Vienna, Austria, July 2002.