Reihe Informatik
003 / 2009

Scalability Transformations on Declarative Applications

Alexander Böhm   Carl-Christian Kanne

University of Mannheim
alex|cc@db.informatik.uni-mannheim.de

Scalability Transformations on Declarative Applications

Alexander Böhm
University of Mannheim
Germany
alex@db.informatik.uni-mannheim.de

Carl-Christian Kanne
University of Zurich
Switzerland
kanne@ifi.uzh.ch

ABSTRACT

Many current distributed applications are based on the exchange of XML messages. Scaling such applications to the high processing volume demanded by Internet-scale deployment typically requires costly redesign and coding.

In this paper, we investigate how a declarative specification of such applications can simplify the task of deploying them on a large number of host machines. In our model, applications are represented as a graph of message queues connected by message flow rules. The state of application instances is encoded in the message history of the queues and accessed using XQuery expressions. We show how to split such an application into distributable fragments using graph partitioning and discuss different algorithms for placing the fragments on hosts. Typically, an initial application specification contains data dependencies that place an upper limit on the number of fragments, and hence the number of usable machines. We describe transformations that increase the number of possible fragments by converting data dependencies into message flow. An evaluation using the TPC-App benchmark and a runtime system prototype confirms the feasibility and performance benefits of this approach.

1. INTRODUCTION

To be successful, modern applications need to have very small times between idea and realization. At the same time, they must be able to scale to a large number of concurrently active instances to avoid becoming the victims of their own success. Utility computing provide access to the required processing and storage capacities [19, 34]. However, on the software side, creating scalable applications remains a difficult task that cannot be fully automated.

Typical architectures follow an approach where messages are organized in queues, state is stored in DBMS, and application code is written in imperative languages. We believe that this model can be greatly improved by changing two aspects. Firstly, by using a declarative rule language to describe the processing logic. Secondly, by exclusively modeling state using the message history and accessing it using declarative expressions. This programming model is investigated in the Demaq project [7, 8] in the context of XML messaging (e.g. Web Services). So far, we have focused on developer productivity and execution efficiency on single hosts that participate in distributed processes.

.

In this paper, we turn to the parallel execution of declarative application programs. In this context, the unified data model for representing state and messages is showing great promise, because we can simply transform state (in our case: history) access to messaging, and vice versa. This is the core of our approach: Instead of designing a general, distributed run-time system, we model deployment of an application as a source-level transformation that turns a non-distributed application specification into a set of programs that can be executed on the various machines of a cluster. All required messaging becomes explicit, and the resulting programs can be run on the local run-time systems.

The challenges of this approach are to automate as much of this parallelization process as possible and to assist developers in making and implementing design choices when parallelizing an application.

To deal with these challenges, we represent the message queues and rules of an application in form of graphs that represent message flow and message history access. Distribution of the application functionality becomes a matter of graph partitioning. Further, we present "scalability transformations" that modify the application structure by introducing messaging operations that allow for more partitions of the graph. These transformations represent a toolbox of methods to increase the number of hosts an application can be run on.

In particular, we show how

- to decompose a declarative messaging application into independent fragments that can be executed on different host machines
- to assign application fragments to hosts based on workload information
- to increase the number of fragments by
  - converting message history (state) access to message flow
  - replicating fragments based on partitions of the message history
- the performance of a benchmark application (TPC-App) can be improved by our techniques

The remainder of the paper is organized as follows. We discuss our programming model in Sec. 2. Sec. 3 gives an overview of the TPC-App benchmark that we use as a running example throughout the paper. In Sec. 4, we discuss how the independent fragments of an application can be automatically identified and deployed to a cluster of hosts. Sec. 5 discusses how applications can be rewritten to yield more fragments and thus can potentially be deployed on a larger number of machines. We evaluate the benefits of the proposed techniques in Sec. 6 before briefly reviewing related work in Sec. 7. Sec. 8 concludes the paper.

2. PROGRAMMING MODEL

Our programming model describes the application logic of a node in a distributed XML messaging application using two fundamental components: Queues and rules.

XML message queues (Sec. 2.1) provide asynchronous communication facilities and allow for reliable and persistent message storage. Declarative rules (Sec. 2.2) operate on these message queues and are used to implement the application logic. The Demaq execution model (Sec. 2.3) captures the typical behavior of message-driven applications in a few simple rules and guarantees that need to be provided by the corresponding run-time system.

2.1 XML Message Queues

Distributed messaging applications are based on *asynchronous* data exchange. Queue data structures offer efficient message storage and retrieval operations while preserving the order of incoming data. Queues also allow to decouple the retrieval of a message from its processing. This is particularly useful for an application to keep interacting with communication partners in periods of high load.

In our model, queues do not only serve as buffers for external communication, but also to provide the persistent memory for a node. Instead of deleting messages after processing, they are only marked as processed and remain accessible.

All state changes in our model are reflected by messages. Consequentially, the current state of an application instance is expressible as a declarative query against the message history. For this query to yield the correct result, messages have to be retained as long as they are necessary to compute the state of an instance. In Demaq, we allow application developers to directly specify a condition that must be met by the messages that are sufficient to represent the current application state. Access to the message history then yields the smallest suffix which contains such a set of relevant messages.

The logical model of our message queues is based on the XQuery Data Model (XDM) [16]. XDM is the data model of most XML query languages including XPath 2.0, XSLT 2.0 and XQuery. For our purposes, XDM is particularly suited, as its fundamental type is the ordered sequence, which nicely captures message queue structures.

2.1.1 Physical Queues

Our programming model incorporates two different kinds of queues. *Gateway queues* provide communication facilities for the interaction with remote systems. There are two different kinds of gateway queues, inbound and outbound. Messages placed into outbound gateway queues are sent, while inbound gateway queues contain messages that have been received from remote nodes. Various protocols are supported, including HTTP, SMTP, SOAP, and various binary Demaq-to-Demaq protocols that avoid to fully serialize XML for communication between hosts of a single cluster, with a format very similar to the protocol buffers of [20].

Queues are also used as persistent storage containers. These *basic queues* allow applications to define intermediate steps in their control flow and also to simply store data.

As a result, messages received from remote communication endpoints and local state representation are handled in a uniform manner, simplifying application development and distribution.

2.1.2 Virtual Queues (Slicings)

To declare which portions of the message history are relevant in particular contexts, the Demaq language incorporates the concept of virtual queues, called *slicings*.

Slicings can be seen as a kind of parameterized view [31] that extends the concept of data independence to the application state. They support compact rule formulation by giving a name to frequent parameterized expressions. Further, the explicit declaration of relevant message subsets can be used for optimization purposes, e.g. by indexing or materializing slicings.

Slicings are used to simplify the implementation of recurring design patterns in messaging and workflow applications, such as "correlation sets" in BPEL , or "conversations" in XL [17]. Additionally, they can be used to join control flow after executing several tasks in parallel, or to establish synchronization points and milestones [33] within an application.

A slicing defines a family of *slices*, where each slice consists of all the messages with the same value for a particular part of the message (*slice key*). For each slicing, the function to map a message to its corresponding slice key is specified in the Demaq language using an XPath expression. The evaluation of the expression on the message's root node yields the slice key.

2.2 Declarative Rules

In our model, the application logic is specified as a set of declarative rules that operate on messages and queues. Each rule describes how to react to a single new message in a queue. Depending on the structure and content of this message, rule execution results in the creation of new messages. These result messages can either become the input for another rule, or be sent to a remote system using a gateway queue.

Our rule language is built on the foundation of XQuery [5]. It allows developers to directly access and interact with XML fragments stored in message queues. Thus, there is no mismatch between the type system of the application programs and the underlying communication format. Additionally, existing tools, query processing and optimization techniques can be adapted to our application language.

2.2.1 Rule Structure

Rules consist of rule heads and rule bodies. *Rule heads* simply define the name of a rule and associate it with a single queue or slicing. The *rule body* consists of a single XQuery expression. Whenever a message gets inserted into this queue or slicing, the rule body is evaluated with this message as the context item.

Optionally, an *error queue* can be defined for each application rule. Whenever a runtime error is encountered dur-

ing the execution of this rule, a corresponding notification message is sent to the associated error queue. Thus, errors can be handled by rules defined on the corresponding error queue. If no error queue is defined, error notifications are inserted into a system-provided, default error queue.

2.2.2 Rule Effect

Every application rule describes how to react to a message by creating new messages and enqueuing them into local or gateway queues. While XQuery allows for the creation of arbitrary XML fragments, it does not incorporate any primitives for performing side effects. In our model, this is a severe restriction, as there is no possibility to modify the content of the queues underlying our application rules.

We adopt the update method introduced by the XQuery Update Facility [11] to perform side effects on the message store. Every application rule is an updating expression that produces a (possibly empty) list of messages that have to be incorporated into the message store by enqueuing them into corresponding queues. Demaq extends the XQuery Update Facility with an additional `enqueue message` update primitive.

2.2.3 Message History Access

Our model encodes application state as queries against the message history. Hence, we need to extend XQuery to allow access to this history. This can be done without requiring changes to the syntax or semantics of XQuery by providing external functions. Our language incorporates functions for accessing the sequence of XML messages in a particular slice (`qs:slice`) and to retrieve the slicekey of a message.

2.3 Execution Model

The fundamental behavior of messaging applications can be described as a simple loop that (1) decides which message(s) to process, (2) determines the reaction to that message based on its content and the application state, and (3) effects the reaction by creating new messages. In existing systems, this loop is mostly coded by hand, optimizing for the requirements of each application. Implementations of the Demaq model may use any form of processing loop(s) that obeys the following constraints:

1. Each message is processed exactly once. This means that the evaluation of all rules defined for the message's queue and slicings are triggered once for every message.

2. Rules are evaluated by determining the result of the rule body as defined by XQuery (update) semantics, extended by the access function definitions described in Sec. 2.2.3. The result is a sequence of pending update operations in the form of messages to enqueue.

3. The overall result of rule evaluation for a message is the concatenation of the pending actions of the individual rules in some non-deterministic order.

4. Processing the pending actions for a message is atomic, i.e. after a successful rule evaluation all result messages are added to the message history in one atomic transaction, which also marks the trigger message as processed.

5. All rule evaluations for the same trigger message see the same snapshot of the message history, which contains all messages enqueued prior to the trigger message, but none of the messages enqueued later.

This list includes strong transactional guarantees necessary to implement reliable state-dependent applications, but still allows many alternative strategies to couple message processing to a transactional message store. Note that the above model does not allow for message store transactions that span rules. However, application developers do have some control over the amount of decoupled, asynchronous execution: The expressive power of XQuery allows the bundling of complex processing steps into single rules, which are executed in a single transaction and hence allow to constrain the visibility of intermediate results to concurrent transactions. Further, application developers can isolate intermediate messages in local queues that are not accessed by conflicting control paths.

3. SAMPLE APPLICATION: TPC-APP

Throughout this paper, we use the TPC-App benchmark [32] as a running example. TPC-App is an application server and web services benchmark proposed by the Transaction Processing Performance Council (TPC). Its objective is to evaluate the performance of transactional application server systems, including complex application logic, remote messaging operations and persistent data management.

3.1 Application Domain

TPC-App implements the application logic of a book distributor. It provides business partners with web service interfaces, i.e. all communication is based on XML messages. The services provided by the book distributor include browsing the product catalog, adding new products, managing master data, as well as ordering and order tracking functionality. While each of these services can be accessed separately by sending XML messages with a particular schema, their functionality is tightly coupled as they access conjoint data such as customer master data, product information or order status. Apart from its own services, the application interacts with external web services e.g. for credit card verification, product delivery, stock management, etc. Again, all this communication is based on XML messages.

3.2 Demaq Implementation

Our implementation of TPC-App consists of about 1000 lines of Demaq code with a total of 32 application rules, 35 queues and 17 slicings. This code also includes the "external" service emulations required by TPC-App. Figure 1 depicts the queues and message flow of the application[1]. In this graph, message flow is represented using black solid edges. If a rule defined on a queue accesses the message history of another queue, this is represented using a dashed red edge.

Figure 2 shows an exemplary rule that implements TPC-App's "product detail" web service together with the corresponding queue and slicing definition expressions. This service allows customers to retrieve detailed product information for a list of product identifiers. In Demaq, the entire functionality, including message and master data access as well as the construction of the result message can be realized within a single application rule.

[1]The complete application, a workload driver and message templates are available at http://www.demaq.net/.

Figure 1: Application Structure (Queues), Message Flow and Data Dependencies of TPC-App

Lines 1-3 contain the queue definition expression for three basic queues that are used for persistent data storage (`items` and `authors`) and intermediate message flow. Two slicings are defined on these queues in order to access the item and author master data based on their respective identifier (lines 5-7).

The `create rule` expression (line 9) declares a new rule with name `handleProductDetail` that is executed every time a message is enqueued in the `productDetail` queue. The product identifiers the customer is interested in are extracted from the customer's message using a path expression (line 13). For each identifier, the corresponding item master data is retrieved using a `qs:slice` function call (line 14). Another slice function call is used to retrieve the author of each individual item. XQuery element constructors are used to assemble the result message (lines 19 to 25). Finally, the result message is enqueued into the `outgoingMessages` gateway queue (line 26), from where it is delivered to the requesting customer.

```
1  create queue productDetail kind basic mode persistent;
2  create queue items kind basic mode persistent;
3  create queue authors kind basic mode persistent;
4
5  create slicing itemByID queue items value /item/id;
6  create slicing authorsByID
7    queue authors value /author/authorID;
8
9  create rule handleProductDetail for productDetail
10 let $request as node() := /ProductDetail
11 let $result as node() :=
12 <ProductDetailResponse><ProductDetailResult>{
13   for $itemID in $request/ITEM_ID/text()
14   let $item as node():=qs:slice($itemID,"itemByID")/item
15   let $authorID := $item/authorID/text()
16   let $author as node() :=
17       qs:slice($authorID, "authorsByID")/author
18   return(
19   <ITEM_ID>{$itemID}</ITEM_ID>,
20   <ITEM_TITLE>{$item/title/text()}</ITEM_TITLE>,
21   <AUTHOR_FNAME>{$author/firstname/text()}</AUTHOR_FNAME>,
22   <AUTHOR_LNAME>{$author/lastname/text()}</AUTHOR_LNAME>,
23   (:skipping additional element constructors:)
24   <ITEM_COST>{$item/cost/text()}</ITEM_COST>)
25 }</ProductDetailResult></ProductDetailResponse>
26 return enqueue message $result into outgoingMessages;
```

Figure 2: TPC-App "Product Detail" Web Service

The rule shows up in Fig. 1 as three edges starting from `productDetail`. One message flow edge to `outgoingMessages`, and two red history access edges to `authors` and `items`.

4. DISTRIBUTION TRANSFORMATIONS

As discussed before, our goal is to increase the performance and scalability of declarative applications by distributing them to clusters of hosts based on commodity hardware.

We use a transformative approach to distribute Demaq applications. By transformative, we mean that we do not create a "distributed" Demaq run-time system, but instead rewrite individual Demaq applications into a set of host programs, one for each host. There are two reasons for this. Firstly, the Demaq language is designed to describe applications that take part in distributed systems, so it is quite simple to add code to a Demaq program that deals with the distribution. Secondly, guaranteeing a "single system view" for a distributed runtime system requires design choices for issues related to distribution [35] that are best solved in an application-specific way. Our approach using a distribution-aware language before and after the transformation allows to fine-tune the application, for example to address consistency issues.

We will now describe the mechanisms for distribution of declarative Demaq applications in more detail. In a first step, we perform a functional decomposition of the application into fragments that allow for local rule processing on each host without changing application semantics. We then describe how to map these fragments to hosts. Given a fragmentation with a corresponding allocation of hosts, we can easily create the required subprograms for each host in a final step.

4.1 Application Fragmentation

There is a natural decomposition of a queue-based messaging application into parts that can be processed by separate hosts. We can simply assign each queue to a host that is responsible for processing that queue's messages. However, since our model allows message history access, we need to be careful to distribute queues on hosts in a way that allows to locally process rules. Without local rule processing, we would incur communication costs not only for sending messages, but also for remote history access.

We call a subset of an application's queues a *fragment* of the application. A fragment is *independent* iff processing its rules only requires access to the message histories of the

queues in this fragment. Different independent fragments may be placed on different hosts without changing the application's semantics. A *fragmentation* of an application is a set of disjoint, independent fragments whose union equals the set of all queues of the application. A *maximal fragmentation* is the largest such set of independent fragments.

Our objective is to find a maximal fragmentation of an application, in order to have as many choices as possible to create a balanced distribution of queues to hosts.

The declarative application specification of rules simplifies to determine a maximal fragmentation. We use the rule bodies to construct a data dependency graph with the application's queues as nodes. There is an edge from queue q_1 to queue q_2 iff a rule defined on q_1 accesses the message history of q_2 (by means of a qs:slice() call).

An example for such a dependency graph is shown in Fig. 1 for our TPC-App implementation. That figure contains the data dependency graph if only the dashed, red edges are considered. The solid black edges represent messages sent between queues, which does not induce a data dependency between the queues.

An edge between two queues means that they cannot be part of different independent fragments. Hence, each connected component represents one independent fragment of the application that cannot be fragmented further. Creating a maximal fragmentation is simply a matter of finding all connected components of the data dependency graph.

4.2 Host Allocation

After the independent fragments have been identified using the dependency graph, the next step is to assign these fragments to the available machines.

4.2.1 Problem Statement

We want to find a map from the application's fragments to a set of host machines H. This should be done in a way that equally distributes the rule evaluation workload among hosts and - at the same time - minimizes network communication by co-locating frequently communicating fragments if possible.

Our fragmented application can be represented as a *fragment graph* $G(V, E, w, c)$, with the vertexes representing fragments. The edges represent communication between fragments, such that there is an edge between v_1 and v_2 iff rule evaluation on a queue from v_1 may cause a message to be enqueued at a queue of v_2. We assume a weight function c that assigns a rule evaluation cost to each vertex. The graph edges are weighted with network traffic using a network cost function w that reflects message count and size in a single cost measure based on the latency and bandwidth of the connections.

A *k-cut* of the graph consists of k disjoint sets of vertexes, such that their union is the complete graph. The value of a cut is the sum of the weights of those edges that connect vertexes in different sets. In our case, we want to find a minimum balanced $|H|$-cut, i.e. a cut for which the sum of node weights in each set is equal (or as equal as possible given a set of weights), and the value of the cut is minimal. Unfortunately, this optimization problem is NP-hard even for $|H| = 2$ (MINIMUM BALANCED CUT), and we have to consider heuristics.

We will first describe our method to determine the weights for a given application program and then discuss various heuristics to solve the problem.

4.2.2 Estimated Application Workload

Cost estimation of XQuery evaluation is difficult and highly dependent on the actual data/messages. Hence, we do not attempt to accurately estimate costs by looking only at the application specification and a workload description. Instead, we rely on statistics collected while running the application with a sample workload on a single host. This profile information is used to guide the partitioning process and to derive a solution that fits the expected workload [10, 14, 23]. The relevant profile information we collect includes the total size and number of messages exchanged between the queues of the application and the rule execution effort (elapsed time) for each individual queue.

4.2.3 Partitioning Heuristics

We evaluate three different heuristics to find good host allocations for the fragments. While there are many more strategies (such as Genetic Algorithms or Simulated Annealing), the focus of this paper is not on an evaluation of partitioning heuristics, and we leave a detailed investigation as future work.

First-Fit Bin Packing Heuristic.

Our first heuristic ignores network communication, assuming that XQuery rule evaluation is the dominant factor when processing Demaq programs. We allocate a given set of fragments to a given maximum number of hosts by solving instances of the BIN PACKING problem with the available hosts as the bins the application fragments are assigned to.

Instead of trying to find a solution with a minimum number of fixed sized bins, our objective is to find a minimal bin size that allows to allocate all fragments using less than a maximum number of available hosts. We approach this problem by performing a binary search on the bin size and solving the bin packing instance at each binary search step. Since bin packing itself is NP-hard, we use the first-fit heuristic to approximate the optimal fragment distribution. First-fit is simple to implement, efficient and produces relatively good solutions [25].

Bond Energy Algorithm.

The Bond Energy Algorithm (BEA) [26] is used in the context of vertical fragmentation of a relational schema into sets of related attributes [27, 28] based on access similarity. This is similar to our problem of host allocation, where an application is partitioned into sets of related fragments based on access similarity as measured by network traffic between fragments.

As input to the algorithm, we use the weighted adjacency matrix A of the fragment graph. We do not distinguish between incoming and outgoing traffic, and construct a symmetric matrix with $a_{ij} = w((v_i, v_j)) + w((v_j, v_i))$. The algorithm then permutes the rows and columns of the matrix, trying to maximize the similarity of neighboring entries. The result is a reordered matrix in block diagonal form where structurally similar rows/columns are "clumped" together.

BEA yields a one-dimensional ordering the fragments where strongly related fragments are close together. We still need to divide this ordering into partitions that are balanced and minimize the communication overhead. In a next step, we

use the partitioning technique proposed by Navathe et. al. [27], which bisects the block diagonal matrix into two sub-partitions. The BEA and binary partitioning steps are recursively applied to these sub-partitions until a given number of target hosts is reached. In each partitioning step, we choose the best cut using an objective function that balances partitions by minimizing the maximum CPU load assigned to a single host and the total aggregated network communication.

Spectral Bisection.

In many application areas, spectral graph partitioning is used to solve optimization problems related to graph cuts. One such area is load balancing in parallel computing [22], which inspires us to also employ a spectral algorithm for our host allocation problem. We only sketch the core idea of the method here, and refer to [22] for details. The algorithm uses a weighted Laplacian matrix as representation for the fragment graph, which is essentially the same matrix as used by the BEA above, but with additional diagonal elements that equal the sum of all network traffic entering or leaving a fragment. Spectral graph partitioning now computes the eigenvectors of this matrix. The first nontrivial eigenvector represents a partitioning of the graph vertexes that can be used as a heuristic for a minimal balanced cut. Sorting the graph vertexes by their corresponding components in that eigenvector again yields an ordering of our fragments that reflects their structural relationships, much as the permuted matrix of the BEA does. After computing the ordering, we again apply a recursive partitioning scheme similar to Navathe et. al. [27] to derive the fragment allocations.

4.3 Deployment

After the fragments have been mapped to the available hosts, the final step in the distribution process is to transform the initial application into a set of host programs. Each individual host program contains the application code of the corresponding fragments. This includes queue and slicing definitions as well as application rules.

In order to allow for host-to-host message exchange, additional gateway queues have to be created. This includes incoming gateway queues to receive messages that need to be processed by local fragments from other machines, as well as outgoing gateway queues that allow to send messages to remote fragments. Additionally, local application rules that enqueue messages into queues belonging to a remote fragment have to be rewritten to use the outgoing gateway queues instead.

5. SCALABILITY TRANSFORMATIONS

The previous sections discuss the distribution of "vanilla" applications as originally written by the developer. In practice, the message history accesses of such applications often limit the amount of parallelism and distribution that can be achieved automatically. The method of choice to reduce these dependencies is to manually redesign and modify the application to replicate some of the data and control information and ship it between nodes.

In our approach, we describe scalability-improving transformations systematically as source-level rewrites, in a formal and succinct manner that simplifies their application, and at the same time allows local decisions on where to apply them that do not affect other parts of the application.

We present full source code only for a few rewrites and discuss only the structure of the remaining transformations to save space. In a final section, we discuss how to control application of the rewrites.

5.1 Breaking History Access Chains

Whenever a rule r defined on a queue q accesses the history of another queue s, our approach requires q and s to be in the same fragment. This prohibits the mapping of q and s to different hosts. This transitively applies to other queues if they access the history of q, and may cause unrelated queues to share a fragment without necessity. The left side of Fig. 3 shows such a situation, with rules on p accessing q and rules on q accessing s. As a result, p must be in the same fragment as s.

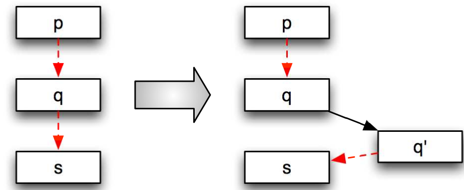

Figure 3: Broken History Access Chain

We can decouple p from s simply by copying the current message from q to a new queue q' and processing the rule r on that queue instead. This converts history access to message flow, and allows to put p and s into different partitions. The history of q is still intact and must reside in the same fragment as p, but processing of r can be done in a separate fragment. The desired outcome is shown on the right side of Fig. 3.

5.1.1 *Simple Version*

The actual source-level rewrite to perform depends on the rule body r. If r does not access the history of q, the rewrite is very simple. Before the rewrite, the code looks like this:

```
create rule r for q
  body
```

This code is replaced with

```
create rule r' for q
  enqueue message . into q';

create queue q' ...

create rule r'' for q'
  body
```

The result is as described above. Note that we do not need to keep q''s history, as only the current message is necessary, whereas we still keep q's history for rules defined on p. This reduced amount of required history can be specified using the retention mechanisms explained in Sec. 2.1.

5.1.2 *Transforming History Accesses*

If there is a message history access to q in r, the situation is slightly more complex. For ease of exposition, assume that this self-history access of q is through a single slicing t (the method below can be easily extended to more than one slicing). In this case, we can "annotate" the context message

with the required history slices of t before sending it to q'. In the rule body evaluated on q', we replace all accesses to slices of t with accesses to the "enriched" context message.

Applied to our "before" code from above, the corresponding rewritten code is

```
create rule r' for q
  let $t:=<slicing name="t">
            { for $k in slicekeysfrombody
              return <slice value="{$k}">
                       {qs:slice($k,"t")}
                     </slice> }
          </slicing>
  return enqueue message <pkg> <origmsg> { . } </origmsg>
                                 { $t }
                              </pkg> into q';

create queue q' ...

create rule r'' for q'
  body'
```

where `body'` is rule r's body in which all accesses to the current message have been replaced with `/pkg/origmsg`, and all calls of the form `qs:slice($k,"t")` have been replaced with expressions that access the slices encoded in the enriched message, such as

`/pkg/slicing[@name eq 't']/slice[@value eq $k]/*`.

The expression `slicekeysfrombody` is obtained by transforming `body` into an XQuery expression that returns a sequence of all the slicekeys required in the evaluation of `body`. Using the rewrite function $[\![\cdot]\!]^s_D$ as defined below, we have `slicekeysfrombody`$:=[\![$`body`$]\!]^t_\emptyset$.

As above, the history of q' does not need to be retained, only the currently unprocessed messages have to be stored.

In the TPC-App example, this rewrite allows us to cut the history access edges from the `verifiedOrderStatus` queue to the `orders` and `shipping` queues. Consequentially, the order and shipping related functionality (the ten queues in the lower right corner of Figure 1) becomes an independent fragment.

5.1.3 Determining Slice Keys at Run-Time

The rewrite rule to embed slice contents into an enriched message discussed above must be able to determine the slice keys required to evaluate the rule body for a specific input message. If these slice keys are not constants but depend on the input message, they need to be computed at run-time. We define a function $[\![\cdot]\!]^s_D$ which transforms an XQuery body expression into an XQuery expression that may be used to compute the list of required slice keys for slicing s at run-time.

To define our function, we assume a normalized form of XQuery expressions called XQuery Core [15], where every FLWOR expression only contains a single `for` or `let` followed by a `return`, and `where` clauses are replaced by `if then else` conditionals. Further, we assume that every simple expression is decomposed into single operations with only variables as arguments. Operators, updating primitives and XML constructors are treated like function calls. For example, $\$x + 1$ is normalized to

```
let $one:= 1 return ( let $r := $x + $one return $r )
```

When applied to such a normalized expression, our function descends down the syntax tree, recursively applying itself to the subexpressions. The noteworthy exceptions to recursive propagation are shown in Fig. 4. We never want to return actual results of operations, hence rewrite variable references to empty sequences wherever they are not used as function arguments (1). We return any used slice keys for slicing s, but do not use results of slice access, and drop the corresponding variable definitions (2). We remember the dropped variable in D in the recursive call. This removes all code from the expression that depends on slice contents. For regular function calls or slice calls on other slices than s, we progress recursively unless one of the arguments does not exist, in which case we also drop the result variable (3). Finally, if a conditional depends on a dropped variable, we try to determine the slice keys in both branches (4).

Note that we cannot determine slice keys if the referenced slice key depends on another access to the same slicing. In this case, we return an error in (2), and cannot apply our rewrite[2].

As an example, rewriting the rule body

```
for $i in //productID
return count(qs:slice($i,"productsByID"))
```

for slicing `productsByID` yields the rewritten expression

```
for $i in //productID
return $i
```

as desired.

5.2 Replication of Fragments

So far, we have discussed functional fragmentation of the application, where different functional aspects of the application run on different hosts. Obviously, there is an upper limit to the number of fragments, as we cannot have more fragments than queues. For typical applications, this is a severe limitation if we want to scale beyond a few dozen machines. In these cases, it is desirable to partition the data, and to run the same part of the application for each of the data partitions in parallel.

It turns out that by applying another program transformation, we can use our fragmentation algorithm in unmodified form to also distribute the same fragment to multiple machines. This program transformation is relatively simple: We replicate those parts of the application structure that are supposed to run in parallel. This makes the parallelization opportunities explicit, and we can run our fragmentation algorithms on the resulting program.

The challenge with this approach is to identify application fragments that can be replicated without changing semantics. We will now describe a conservative criterion for parellizability that is very simple to implement. We also define relaxing program transformations that create more parallelizable fragments in a given application.

5.2.1 Fragment Locality

Our approach assumes that we have a function g that maps each message m to a group $g(m) \in G$ from a sufficiently large set of groups G (we will discuss the choice of g and G below). Let us denote with $M_r(m)$ the set of messages from the history accessed for the evaluation of a single rule r on a single message m. We say a rule r is *local* with respect to g, iff for all possible messages m, the messages in $M_r(m)$ belong to the same partition, i.e. we can find a group $g_m \in G$ such that $g(m') = g_m$ for $m' \in M_r(m)$. We say a fragment f of an application is local for g iff all the rules defined on queues and slicings of f are local for g.

[2]Actually, we can apply a more sophisticated rewrite in that case, which we omit here due to space constraints.

$$(1) \quad [\![\$x]\!]^s_D := ()$$

$$(2) \quad \left[\!\!\left[\begin{array}{l} \text{let } \$x \ := \ \text{qs:slice}(\$k, s) \\ \text{return } expr \end{array} \right]\!\!\right]^s_D := \begin{cases} (\$k, [\![expr]\!]^s_{D \cup \{\$x\}}) & \text{if } \$k \notin D \\ \textbf{see text} & \text{if } \$k \in D \end{cases}$$

$$(3) \quad \left[\!\!\left[\begin{array}{l} \text{let } \$x \ := \ \text{func-or-op}(\$v_1, \ldots, \$v_n) \\ \text{return } expr \end{array} \right]\!\!\right]^s_D := \begin{cases} \begin{array}{l} \text{let } \$x \ := \ \text{func-or-op}(\$v_1, \ldots, \$v_n) \\ \text{return } [\![expr]\!]^s_D \end{array} & \text{if no } \$v_i \in D \\ [\![expr]\!]^s_{D \cup \{\$x\}} & \text{if any } \$v_i \in D \end{cases}$$

$$(4) \quad [\![\text{ if } \$x \text{ then } expr_1 \text{else } expr_2 \ ]\!]^s_D := \begin{cases} \text{if } \$x \text{ then } [\![expr_1]\!]^s_D \text{ else } [\![expr_2]\!]^s_D & \text{if } \$x \notin D \\ ([\![expr_1]\!]^s_D \,, [\![expr_2]\!]^s_D) & \text{if } \$x \in D \end{cases}$$

Figure 4: Rewrite Rules for Determining Required Slice Keys in Rule Bodies

Locality is a useful property because we can deploy multiple copies of a local fragment, each processing a different partition of the message history, up to one copy for each group in G.

Locality is not decidable for arbitrary rule bodies and arbitrary functions g, due to the expressive power of XQuery. However, a locality check that catches many practical cases of locality is straightforward to implement. A trivial case is a rule without any message history accesses: It is always local. Another simple case is the use of slice keys as partitioning function g. If a rule accesses only messages from a single slice, then it is local (because by definition, all messages in the same slice have the same slice key).

We omit a detailed discussion of more advanced techniques to detect locality due to a lack of space. Note that even the simple cases explained above cover many practical cases. Particularly, this can be ensured by first running the fragmentation algorithm of Sec. 4, which reduces a fragment's data dependencies to a minimum.

5.2.2 *Replication of Local Fragments*

Our method to execute several instances of a local fragment f in parallel makes the fragment's locality "explicit" by replacing f by one copy of f for each message group in G. The copies of f have the same rules and slicings defined as f, and the rules are changed only slightly: Every enqueue operation on f's queues (from any rule in the application) must be replaced by a proper "dispatch" code that calculates the message's g_m value, and sends the message to the proper copy of f.

An example of its application is visualized in Fig. 5. Before the transformation, there are three fragments, of which $f2$ is local. In the transformed program, there are many copies of $f2$, of which three are shown. Message flow from the untransformed program is unchanged, but applies to all copies of $f2$ (we have omitted some message flow edges for clarity). However, the data dependencies are local in each copy.

The explicit representation of parallelizable application fragments allows our host allocation methods from Sec. 4.2 to be used on the resulting program. Note that, given proper workload information, the allocation can take nonuniform workloads into account. For example, several copies of a fragment that are less often used may share a host (possibly with other, unrelated fragments), whereas heavily used copies of the same fragment get assigned a host of their own. It may even be desirable to apply further rewrites to only the heavily used copy to distribute it over several hosts.

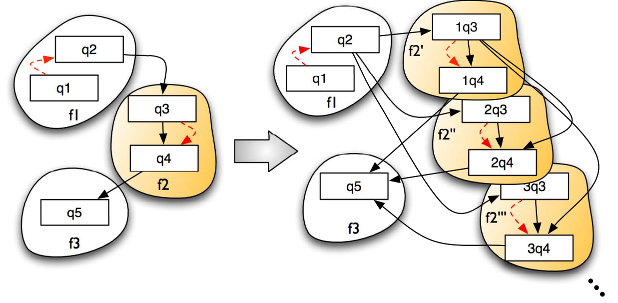

Figure 5: Local Fragment Replication

5.2.3 *Message History Partitioning Functions*

The application of the transformation described above hinges on the ability to find a partitioning function g for the message history. A straightforward choice for this function are the slice key definitions supplied in the application specification. They group related messages in the application domain, making it desirable to keep each such group on a single host.

For example, in the TPC-App benchmark, the stock master data can be replicated to several machines based on the `stocksByItemID` slicing which provides access to stock master data based on the item identifier. For dispatching messages enqueued into the `stocks` queue to the right replica, the itemID slicekey is computed in advance and is used to route the message to the right replica.

It is not always a good idea to directly use slice keys for partitioning, however. The domain of the slice keys may be unbounded (e.g. for customer numbers). Even if it is known at compile-time, a very large number of fragment copies may cause problems for the host allocation algorithms from Sec. 4.2. For this reason, we use only the least significant bits of the slice keys for partitioning. The number of bits used depends on the slice key frequency distribution. The details are beyond the scope of this paper, but the idea is to have the most heavily used slice key partition require less processing resources than the most processing intensive, non-replicated fragment in the remainder of the application.

We can also model round-robin distribution using this method. If there is no history access in the fragment, it is local and we can arbitrarily dispatch messages to the replicated copies. By introducing a slicing that partitions by the least significant bits of a message sequence number, we

can achieve a uniform distribution of messages to fragment copies.

5.3 Transforming Non-Local Fragments

Replicating local fragments as explained above is very effective at scaling declarative applications. However, the technique is directly applicable only to applications in which local fragments cover a significant fraction of the total processing effort. Below, we discuss program transformations that convert non-local fragments to local ones, thus allowing to increase the parallelizable portion of an application.

5.3.1 *Replication of Slicings*

A fragment is typically not local if it contains accesses to two different slicings that are defined on the same queue, but with different slice keys. For example, the items in the TPC-App application are accessed by using both the `itemsByID` slicing, which allows to retrieve item master data based on the unique item identifier and the `itemsBySubject` slicing, which retrieves items based on their subject.

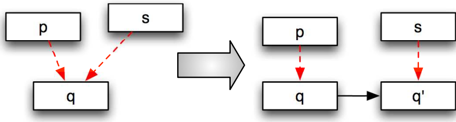

Figure 6: Queue Replication to Decouple Slicings

A simple method to parallelize such application patterns is the replication of message histories. This can be captured as a simple source-level transformation if the two different slicings are accessed in different rules. Given a queue q, on which two slicings are defined which cause a fragment to be non-local, we can introduce an additional queue q' that contains the same messages as q. This replication of messages can simply be achieved adding a rule to q that sends a copy of every message to q'. Further, we change the definition of one of the original two slicings to refer to q' instead of q. The structure of this rewrite is visualized in Fig. 6. Again, we convert a data dependency edge into a message flow edge, transforming a single non-local fragment into two (possibly) local fragments.

5.3.2 *Data Shipping Transformations*

The transformations discussed so far may fail to decompose an application into sufficiently small components for adequate parallel processing. As a last resort, the application developer may have no choice but to remotely access data that belongs to another fragment.

Due to space constraints, we address only one specific application pattern where explicit data shipping may help. Assume that we access multiple slices of one slicing in a single rule. Typically, we cannot guarantee that the slices belong to a single partition of the message history and, hence, the fragment containing this rule is non-local. To remove this multiple data dependency, we can add additional queues to provide access to remote slice content using new, local rules, as follows.

We replace the original rule r with a new "fork" rule. This rule computes the slice keys required by r for the input message, using an expression obtained from r as explained in Sec. 5.1.3. It sends one request message for each slice key to a helper queue and also forwards the input message to a new "merge" queue. On the helper queue, a single new rule extracts the slice key from the input message, accesses the corresponding slice, and packages the result into a reply message, which is sent to the "merge" queue. The merge queue contains a rule that checks whether the original input message and replies for all slice keys have been received. If so, it evaluates r with the original qs:slice() calls replaced with accesses to the reply messages, similar to the rewrite in Sec. 5.1.2. In this rewritten version of the original program, there are only local rules, as only single slice calls are used in each rule.

The TPC-App benchmark contains a web service that allows to retrieve item information for a user-defined set of itemIDs. The corresponding Demaq rule uses the `items-ByID` slicing to retrieve detailed information for each of the user-supplied identifiers. Using the above steps, the application code can be rewritten in order to avoid the corresponding fragment to be non-local and thus potentially become a scalability bottleneck.

As in the example above, we can model data shipping protocols as source-level transformations instead of hardcoding a specific data shipping protocol into the run-time system (by providing a remote procedure call-based version of the `qs:slice()` function). This way, we can more easily control which parts of an application may use data shipping, and also combine the technique with other rewrites in a single transformation process.

5.4 Rewrite Control

Our scalability transformations are concise representations of techniques that application developers can employ to systematically increase the parallelizable portion of an application. They are easy to understand, because they only affect small parts of an application, and are expressed in the same language as the original application program. This allows the developer to look at the rewritten programs, to understand consequences of rewrites at the source level, and to fine-tune them.

We do not discuss how to control the application of our rewrites in this paper. Of course, it would be desirable to fully automate the rewrite process. This is possible only to a limited extent, as application distribution involves additional latency, concurrency issues and sources of errors [35]. Not all techniques used by developers to improve scalability leave application semantics unchanged, and this includes our rewrites. For example, when ordering at an online store, some items are shown as being in stock. Some time after ordering them, the store reports them as unavailable in an email message, because - for scalability reasons - the store does not have a single consistent database for all its aspects. Instead, asynchronous messages and application-specific protocols are used, and certain inconsistencies are tolerated and repaired after they have been detected.

Our rewrites reflect this. For example, when replicating a queue in Sec. 5.3.1, the replicated queue may lag behind the the original queue and will not always be in sync instantaneously. The big advantage of our approach is that we can decide to locally give up consistency for small portions of the application. Instead of using a scalable, but inconsistent method for the whole application, we can decide for each rewrite step whether to tolerate the consequences in terms of lost consistency or extra messaging cost. We can also use

more elaborate rewrites that include an implementation of some consistency protocol in the rewritten program.

Our programming model and "library" of rewrites allows to formally describe techniques for improving scalability in a representation close to the application's source code. Currently, we are investigating several alternatives for integrating the rewrites into a toolchain. One possibility is to annotate the source code with hints about the applicability of rewrites, partitioning functions, or local consistency requirements. This can either be done inline or using a separate "scalability descriptor".

Another approach is assisted manual rewriting. For this purpose, we are extending our graphical Demaq editor [8] with a "scalability assistant". Running the distribution transformations within the editor then allows to graphically display fragmentation and bottlenecks, and suggest applicable rewrites.

6. EVALUATION

In this section, we verify the feasibility of our approach. We do not attempt a comprehensive benchmark of various approaches here, but take a first step by discussing how to apply our transformation-based method to the TPC-App benchmark. The application is of reasonable size that qualifies it as a realistic example, and the application domain is one that certainly requires scalability.

For this purpose, we implemented an application rewriting framework based on the rule rewrites and allocation heuristics discussed in the previous sections. It uses workload estimations collected by the profiling component of the Demaq runtime system to transform a given application into a set of resulting application that can be deployed on the target hosts without manual code modifications.

Test Setup. Our testbed consists of four hosts, each of them equipped with an Intel Pentium 4 CPU at 2.80GHz and 2 GB RAM, running Ubuntu Linux 8.10. The hosts are connected by a 100MBit switched Ethernet. The Demaq runtime system was deployed on each host and used for application execution. We refer the interested reader to [7, 8, 6] for a comprehensive description of the system components and its performance.

We investigate the web service interactions per second that can be achieved when running an example workload conforming to TPC-App's web service interaction mix for setups with different numbers of machines and partitioning heuristics. We compare the results to the service interactions per seconds that are achieved when running the same workload on a single host. Figure 7 depicts the speedup that can be achieved depending on the number of involved hosts and whether scalability rewrites (Sec. 5) have been used.

Distribution Effect. The baseline for our experiments is the number of web service interactions per second that can be achieved when running the application on a single host. By deploying the application to two hosts allows us to increase the performance by a factor of 1.71. For more than two hosts, a maximum speedup factor of 1.77 is achieved using the bin-packing heuristic. Although there are 12 independent fragments in the original specification, the distribution of rule evaluation effort is so skewed that more hosts have no performance benefits.

Rewrite Effect. By applying the message replication technique discussed in Sec. 5.1 to the TPC-App implementation, we can increase the number of independent fragments

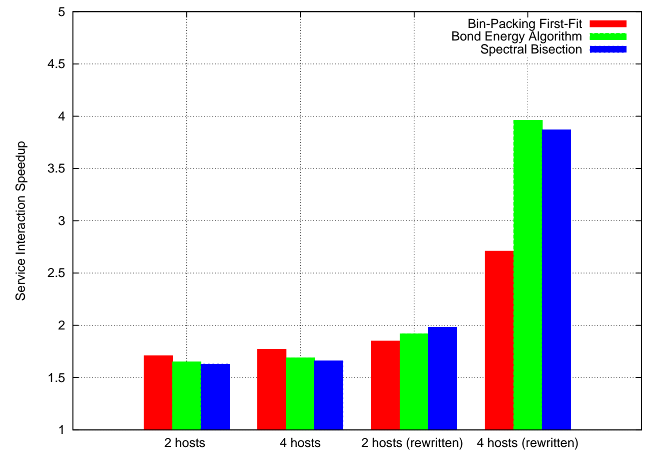

Figure 7: TPC-App Speedup Factor Relative to Single-Host Execution

and thus allow for a more fine-grained deployment. Running the rewritten application on a single host slightly decreases the number of service interactions that can be achieved by a factor of 0.99. This is not surprising as the rewritten application performs slice data shipping (see Sec. 5.1.2) instead of using more efficient index-based local disk access.

For a setup with two hosts, we achieve a performance speedup factor of 1.85 using Bin Packing or even 1.98 for spectral bisection.When deploying application fragments on four hosts, scalability rewrites become essential to benefit from the additional machines. For the rewritten application, the performance can be increased by a factor of 2.71 using the communication-unaware bin packing heuristic. Using spectral bisection and BEA allows to improve performance by a factor of 3.87 and 3.96, respectively. Using the scalability rewrites and these communication-aware heuristics allows to benefit from all machines in our testbed.

Choice of Host Allocation Heuristic. The effect of the host allocation heuristic appears to depend on the number of fragments available. In the non-rewritten application, the network-agnostic method is faster. The more fragments are available, the more important the impact of communication becomes, and the two communication-aware methods dominate, although they both perform similarly.

Further rewriting TPC-App. For the performance experiments in the last section, we used the message replication technique (Sec. 5.1) to partition the TPC-App application into 14 independent fragments. Using this basic source-level rewrite thus allowed us to significantly increase the application performance compared to the non-rewritten version when deploying the application to four hosts.

As our testbed was limited to four hosts, no additional rewrites were necessary to increase the number of fragments in order to deploy the application to a greater number of machines. However, in a setting where more machines are available, the other rewrites discussed in Sec. 5 can be used to increase the number of fragments significantly.

In our TPC-App example application, all product-related operations (the "product detail" web service discussed in Sec. 3.2 and the service for creating new products) belong to the same application fragment, as they require access to the item master data using corresponding slicings. To further

increase application scalability, this fragment can be rewritten in order to be deployed on multiple machines. In a first step, the two different slicings that are used to access the item master data (by itemID and subject) can be separated using the rewrite of Sec. 5.3.1. Next, the multiple slice function calls used by the product detail web service (line 6 in Figure 2) can be rewritten (Sec. 5.3.2), allowing to deploy the item master data management on an arbitrary number of machines. Similarly, other fragments can be decomposed, thus significantly improving the scalability.

7. RELATED WORK

Our techniques for automatic application fragmentation and distribution, as well as the Demaq programming model and runtime system embrace and intersect with work from a multitude of domains and research areas. We briefly review the most closely related work in the following sections.

Application Servers. Today, distributed applications are usually executed by multi-tier application servers [2]. For XML messaging applications, these tiers typically consist of queue-based communication facilities (e.g. [18, 24]), a runtime component executing the application logic, and a database management system that provides persistent state storage. An additional transaction processing monitor ensures that transactional semantics are preserved across these tiers.

Application servers allow for the convenient deployment of applications in distributed and heterogeneous environments. However, their use entails several problems which are discussed in the literature. Significant functional overlap and redundancy between the different tiers wastes resources [21, 29], and configuration and customization in typical multi-layer, multi-vendor environments with limited native XML support is complex and brittle [2]. Further, frequent representation changes between data formats (XML, format of the runtime component, relational database management system) decrease the overall performance [17].

Data Stream Management Systems. Data stream management systems (DSMS) and languages (e.g. [1, 3, 13]) are targeted at analyzing, filtering and aggregating items from a stream of input events, again producing a stream of result items. Several stream management systems rely on declarative programming languages to describe patterns of interest in an event or message stream. In most cases, these languages extend SQL with primitives such as window specification, pattern matching, or stream-to-relation transformation [4].

In contrast to application servers that provide reliable and transactional data processing, stream management systems aim at low latency and high data throughput. To achieve these goals, data processing is mainly performed in main memory (e.g. based on automata [13] or operators [1]). Thus, in case of application failures or system crashes, no state recovery may be performed, and data can be lost.

XML Query and Programming Languages. For an XML message processing system such as Demaq, choosing a native XML query language such as XQuery [5] as a foundation for a programming language is a natural choice. However, these query languages lack the capability to express application logic that is based on the process state - they are functional query languages with (nearly) no side effects. There are various approaches [9, 12, 17] to evolve XQuery into a general-purpose programming language that can be used without an additional host programming language.

Application Analysis and Partitioning. There are several approaches that aim at automatic application partitioning for imperative programming languages such as Java [10, 14, 30] or Microsoft's Component Object Model (COM) [23]. Guided by profiling information collected from running the application to be partitioned, the proposed techniques try to decompose the application into independent parts that can be deployed on different machines.

Generally, all these approaches suffer from particularities of the underlying imperative programming languages. This includes dealing with Java system classes that contain platform-specific, native code and thus cannot be distributed, a coarse-grained partitioning as COM components or Java objects cannot be automatically decomposed to yield better load distribution as well as the necessity for complex runtime system modifications or even remote method invocation (RMI) overhead.

8. CONCLUSION

We have investigated the parallelization opportunities of a novel programming model for distributed XML messaging applications. The model is based on message queues as many existing approaches, but uses declarative rules to specify the application logic, and retains the message history to represent state. Given the task of executing such applications on a large scale, we need to deploy them across multiple hosts. A natural graph-based representation of the application programs for this model allows to use simple algorithms to partition them into independent fragments and allocate them to hosts.

Our programming language and execution model already incorporate powerful communication primitives, such that a distributed run-time system for our language would have to duplicate some functionality. Instead, we have taken a transformation-based approach, where the deployment of a program causes the single initial application to be rewritten into programs for the different hosts, with the required inter-host communication made explicit.

The application logic may contain dependencies that prohibit distribution of functionality across many hosts. For cases like this, we have shown how the unified data model for state and messages allows us to formulate various techniques for improving scalability using relatively simple source-level rewrites. They are much less difficult to apply than the redesign and manual recoding required by traditional methods. We have also shown how to represent data partitioning as a source-level rewrite technique, such that functional and data-based decomposition can be treated uniformly.

An evaluation using the TPC-App application server benchmark and the Demaq runtime system confirmed the practical feasibility and performance benefits of the proposed approach.

Acknowledgments. We thank Simone Seeger for her helpful comments on the manuscript and Christian Tilgner who maintained the testbed for running the experiments. We also thank Balthasar Biedermann, Vassil Hristov, Dennis Knochenwefel, Martin Krämer, Andreas Kremer and Erich Marth for their contributions to the Demaq system.

Part of this work was supported by the German Research Foundation (DFG) under grant MO 507/12-1.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applictions*. Springer-Verlag, 2004.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. Technical report, W3C, January 2007.

[6] A. Böhm and C.-C. Kanne. Processes are data: A programming model for distributed applications. Technical report, University of Mannheim, 2009.

[7] A. Böhm, C.-C. Kanne, and G. Moerkotte. Demaq: A foundation for declarative XML message processing. In *CIDR*, pages 33–43, 2007.

[8] A. Böhm, E. Marth, and C.-C. Kanne. The Demaq system: declarative development of distributed applications. In *SIGMOD Conference*, pages 1311–1314, 2008.

[9] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. *Computer Networks*, 39(5):645–660, 2002.

[10] B. J. Bradel and T. S. Abdelrahman. Automatic trace-based parallelization of Java programs. In *ICPP*, page 26. IEEE Computer Society, 2007.

[11] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Siméon. XQuery Update Facility 1.0. Technical report, W3C, August 2008.

[12] D. D. Chamberlin, M. J. Carey, D. Florescu, D. Kossmann, and J. Robie. Programming with XQuery. In *XIME-P*, 2006.

[13] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[14] R. Diaconescu, L. Wang, Z. Mouri, and M. Chu. A compiler and runtime infrastructure for automatic program distribution. In *IPDPS*. IEEE Computer Society, 2005.

[15] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. Technical report, W3C, January 2007.

[16] M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model (XDM). Technical report, W3C, January 2007.

[17] D. Florescu, A. Grünhagen, and D. Kossmann. XL: a platform for Web Services. In *CIDR*, 2003.

[18] C. B. Foch. Oracle streams advanced queuing user's guide and reference, 10g release 2 (10.2), 2005.

[19] Google. Google app engine. `http://code.google.com/appengine/`.

[20] Google. Google protocol buffers. `http://code.google.com/apis/protocolbuffers`.

[21] J. Gray. Thesis: Queues are databases. In *Proceedings 7th High Performance Transaction Processing Workshop. Asilomar CA.*, 1995.

[22] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, Jan 1995.

[23] G. Hunt and M. Scott. The Coign automatic distributed partitioning system. Technical Report 96.05, Microsoft Research, Microsoft Corporation, February 1999.

[24] IBM. WebSphere MQ, 2007. `http://www-306.ibm.com/software/integration/wmq/index.html`.

[25] D. Johnson, A. Demers, J. Ullman, and M. Garey. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, Jan 1974.

[26] W. McCormick, P. Schweitzer, and T. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20(5):993–1009, October 1972.

[27] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.

[28] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall International, Inc., 1999.

[29] M. Stonebraker. Too much middleware. *SIGMOD Record*, 31(1):97–106, 2002.

[30] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In B. Magnusson, editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204. Springer, 2002.

[31] M. Toyama. Parameterized view definition and recursive relations. In *ICDE*, pages 707–712. IEEE Computer Society, 1986.

[32] Transaction Processing Performance Council (TPC). TPC BENCHMARK App (Application Server) Specification Version 1.3. Technical report, February 2008.

[33] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[34] W. Vogels. Web services at amazon.com. In *IEEE SCC*. IEEE Computer Society, 2006.

[35] J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In J. Vitek and C. F. Tschudin, editors, *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1996.