X-CSR: Dataflow Optimization for Distributed XML Process Pipelines

Daniel Zinn* Shawn Bowers† Timothy McPhillips† Bertram Ludäscher*,†

***Abstract*— XML process networks are a simple, yet powerful programming paradigm for loosely coupled, coarse-grained dataflow applications such as data-centric scientific workflows. We describe a framework called Δ-XML that is well-suited for applications in which pipelines of data processors modify parts ("deltas") of XML data collections while keeping the overall collection structure intact. We show how to optimize the execution of Δ-XML process networks by minimizing the data shipping cost in distributed settings. This X-CSR[1] optimization employs static type inference based on XML Schema to determine the XML stream fragments that are relevant to a processor, allowing irrelevant fragments to be bypassed ("shipped") to downstream pipeline steps. Finally, we present evaluation results for a real-world scientific workflow, which shows the practical feasibility of X-CSR. A long version of this paper is available as [1].**

I. Introduction

We adopt a simple and flexible model for designing XML process networks called Δ-XML, and show how Δ-XML pipelines can be efficiently executed within distributed environments. Δ-XML has, for example, application in a scientific workflow setting as discussed in [2]. Δ-XML is well-suited for applications in which pipeline components (or *actors*) modify only parts (i.e., *deltas*) of incoming XML data streams (see Fig. 1a). Δ-XML supports explicit typing of data and actors based on XML Schema, leading to opportunities for analysis and optimization of XML process networks. Below, we describe how to minimize the overall cost of shipping data between distributed processing components.

II. Δ-XML Actor Configurations

We view XML data as (i) unranked, labeled, ordered trees (for typing purposes), and (ii) as token sequences of XML events (for streaming purposes), ignoring XML attributes for simplicity. Data nodes contain (binary) data whose specific representation is unknown to the framework, but understood by actors.

Δ-XML schemas are based on XML Schema [3] and are formalized via regular tree grammars [4]. A type declaration (production rule) has the form $T ::= \langle \mathtt{t} \rangle\, \mathcal{R}$, where the left-hand side (*lhs*) consists of a *type* T, and the right-hand side (*rhs*) consists of a label (*tag*) $\langle \mathtt{t} \rangle$ and a regular expression over types $\mathcal{R}$. Regular expressions are defined using the symbols "." (sequence), "|" (alternation or union), "?" (optional), "$*$" (zero or more), "$+$" (one or more), and "ϵ" (empty word). We omit "." when clear from the context.

A Δ-XML *schema* τ is defined as a finite set of type declarations. Every schema τ implies a set of labels $\mathbf{L}_\tau$ and types $\mathbf{T}_\tau$. Per convention, we use S as our "start type" and require each schema to have a type declaration for S such that S does not occur in the *rhs* of a declaration in τ. Similar to XML Schema, we also require that content models be deterministic.[2] For convenience, we assume schemas are non-recursive. By convention, the types $Z, Z_1, Z_2, \ldots$ are used to denote base data.

Each actor A is assigned a *configuration* (or *signature*) $\Delta_A = \langle \sigma, \tau_\alpha, \tau_\omega \rangle$ that consists of *replacement* (or *match*) *rules* σ for selecting and replacing relevant subtrees of an input stream, an *input selection schema* (or *read scope*) τ_α, and an *output replacement schema* (or *write scope*) τ_ω. Each replacement rule is of the form $X \to \mathcal{R}$, where X is a type of τ_α, and $\mathcal{R}$ is a regular expression over types of τ_ω. Intuitively, a replacement rule states that for any relevant fragment of type X in the input stream, the actor A will put in place of X an output of type $\mathcal{R}$. We call X a *match type*, and each type in $\mathcal{R}$ a *replacement type*. We require the match types of a configuration to be unreachable from each other within the input selection schema τ_α, thus avoiding ambiguous or nested matches. Additionally, each match type occurs in exactly one match rule of σ.[3] We also require that the result of applying σ to a channel schema τ results in a schema τ as defined above. For example, see Fig. 1a in which channel schemas are consecutively transformed by actor signatures.

III. Optimizing Δ-XML Pipelines with X-CSR

The X-CSR optimization transforms (compiles) Δ-XML pipelines into equivalent process networks optimized for reducing data-shipment costs. The result of the transformation introduces *distributors* and *mergers* as new components into the original pipeline. During pipeline execution, *distributors* route data directly to actors for which the data is relevant, and thus around actors for which the data is irrelevant. Similarly, *mergers* automatically receive, combine, and provide routed data to subsequent actors. X-CSR exploits schema information and actor configurations to construct the appropriate distributors. Note that the X-CSR optimization is performed only prior to workflow execution and thus does not affect the original pipeline design (it is invisible to regular workflow designers and users).

Optimization goals and assumptions. The X-CSR optimization is designed to minimize the total amount of data that is shipped during workflow execution. This is expected to decrease the duration of workflow execution, i.e., the wall-clock time associated with the enactment of a single workflow run, when the following conditions hold:

[1] **X**ML **C**ut, **S**hip, and **R**eassemble; pronounced *"X-scissor"*

*Dept. of Computer Science, UC Davis; †UC Davis Genome Center, Contact: {dzinn, sbowers, tmcphillips, ludaesch}@ucdavis.edu

[2] Also called *non-ambiguous* in [3].

[3] Non-deterministic actions can be captured using the "|" operator in the replacement expression $\mathcal{R}$.

(1) The computations carried out by individual actors are expensive in terms of memory or other node resources, requiring actors to be distributed across multiple nodes. **(2)** The number and/or size of data items passed between actors is large such that shipping data is time-consuming relative to the time needed to perform computations on the data. **(3)** The overall computation presents opportunities for pipeline parallelism (a form of concurrency with multiple dependent actors executing simultaneously). **(4)** Initializing the Δ-XML framework on all nodes takes little time relative to the total workflow execution time.

These assumptions are often true of data-driven scientific workflows [5], [6], which typically work over large and complex data sets. To simplify the presentation below, we assume each actor in a pipeline is staged on a different host, and that data can be shipped between any two hosts.

The general deployment pattern for mergers and distributors is shown in Fig. 1. The original Δ-XML pipeline with input schema τ is shown in Fig. 1a, and the result of applying X-CSR to this pipeline is depicted in Fig. 1b. For example, the first distributor D_0 is configured as follows. Because actor A_1 contains only F as a match type in σ_1, and the relevant type path leading to F is $S/A/D/F$, each F fragment in the input stream (with their corresponding context) are routed directly to A_1. Each G fragment in the original input stream is routed directly to A_2, since A_2 has a match type G and G is not used by A_1 (but is in τ). Similarly, E fragments in the original input stream are directly routed to A_3. Finally, because fragments of type B and C in τ are not used by any actor in the pipeline, these fragments are directly bypassed to the final merger M_4. Fig. 1c depicts how instances of the input schema τ are split by the various distributors inserted into the pipeline by X-CSR. See [1] for a detailed description of the algorithm for partitioning the input schemas of actors in Δ-XML pipelines.

Labeling Holes. To merge bypassed fragments, distributors add hole markers ($\circ$) into the stream, and bypassed fragments are assigned filling tags ($\bullet$). To match fillings with holes, each hole is labeled according to its corresponding channel number. Without such labels, a merger such as M_2 would not know whether a given hole corresponds to a fragment sent via channel $d^{\bullet}_{02}$ (and should thus be filled), or if the hole corresponds to a fragment sent via channel $d^{\bullet}_{03}$ (and should therefore be ignored because merger M_3 will fill the hole). Note that labeling holes using only the index of the merger that will fill in the fragment is *not* sufficient. For example, the merger M_3 when receiving a hole labeled with "M_3" cannot determine which bypassing channel it is supposed to obtain the filling from. However, it is not necessary to uniquely number every hole and filling pair, since the order of tokens is preserved along channels.

Distributors. At runtime, distributors continuously map incoming tokens to type symbols in their input schemas. This is done via a Glushkov automata, which we generate for the distributor's input schema (cf. [7]). Based on the partitioning of the schema, we associate a destination with each state and add holes and fillings as appropriate. During runtime, we therefore need to do only O(1) work to determine the destination of any given token.

Mergers. Mergers simply align hole and filler pairs. A merger M_i will sequentially read from the primary input stream (i.e., $d^{\circ}_{i-1,i}$), forwarding all tokens in the stream except holes labeled with $\circ_{jk}$ where $k = i$. When such a hole $\circ_{ji}$ is read, it reads a new filling from channel j and inserts the fragment back into the stream. Therefore, mergers also have complexity of O(1) for each token they read.

IV. Theoretical Evaluation of X-CSR

Optimality of X-CSR. X-CSR-optimized pipelines do not exhibit unnecessary shippings of XML fragments (including data nodes).[4] Notice that as soon as a fragment is produced by an actor (or provided to the first actor of a pipeline), X-CSR finds the closest actor downstream that has the fragment in scope. The fragment is then directly sent to this actor without passing through intermediate actors (as in the unoptimized case). Because the fragment must be received by this actor to guarantee correctness, this shipping is indeed necessary, and thus optimal.

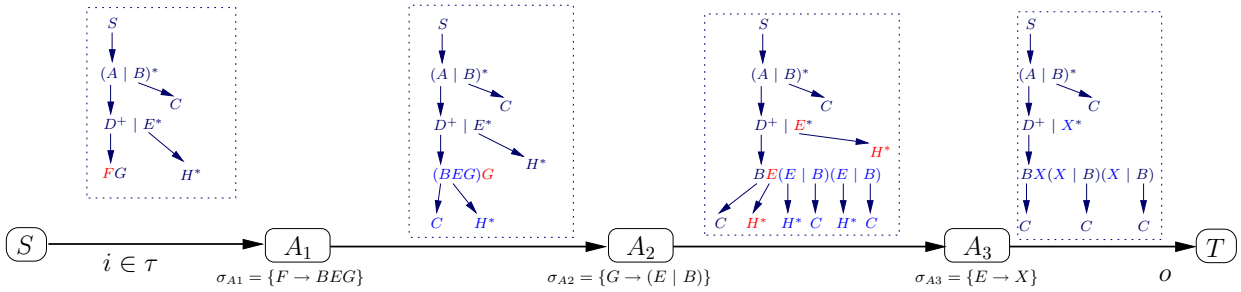
Saved shipping volume. In an unoptimized workflow, shipping data node $\#_{\mathtt{d}}$ from an actor A_i to an actor A_{i+n}, the closest downstream actor that has $\#_{\mathtt{d}}$ in scope, involves shipping $\text{sizeof}(\#_{\mathtt{d}}) * n$ bytes. The optimized version will directly send the data to A_{i+n} and will thus only ship $\text{sizeof}(\#_{\mathtt{d}})$ bytes, resulting in a savings of $\text{sizeof}(\#_{\mathtt{d}}) * (n-1)$ bytes. In other words, the saved shipping cost is *linear* in the number of bypassed actors as well as in the size of the total data nodes (in general, shipped fragments) involved in the shipping optimization. Thus, in X-CSR the more data shipped, the larger the savings.

Compile-time work. Constructing the X-CSR-optimized pipeline involves two main steps: Inferring distributor specifications and building distributor Glushkov automata. Inferring distributor specifications (cf. [1]) is of complexity O(N^2) where N is the number of actors in the pipeline. Since there are O(N^2) direct connections in a pipeline, this complexity is optimal as it is the theoretical lower bound for considering all direct connections. Building the internal Glushkov automata for the distributors is possible in quadratic time with respect to the length of the schema [8]. The PTIME complexity is reasonable in practice since in many scientific workflows, e.g., the schema themselves are rather small—in contrast to the amount and size of data (within a schema instance) that is typically shipped.

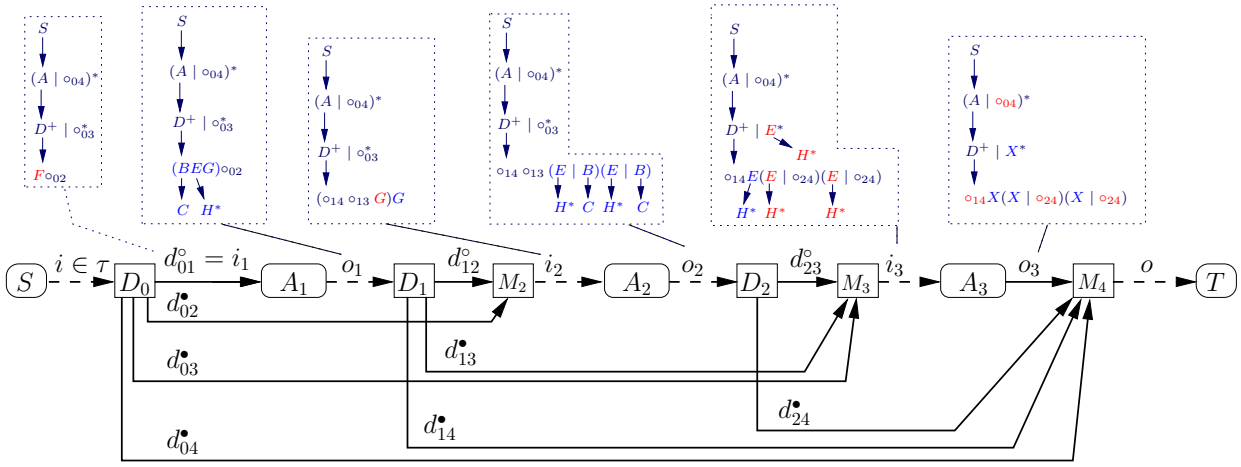
Run-time work. When the X-CSR-optimized pipeline is run, roughly three-times the number of actors are started compared to the unoptimized (original) pipeline, because a distributor and a merge actor is started for each workflow actor. Because starting actors is only a small portion of the overall pipeline execution (and is also done in parallel), we do not expect the runtime overhead to be significant. While data is streamed through the pipeline, both mergers and distributors

[4]Note that optimality is with respect to the available actor configuration information: If the signature of an actor claims that data is used by an actor, then this data will be shipped to the actor, even though the signature might not capture the actor's real behavior.

(a) Original pipeline



(b) X-CSR optimized pipeline



(c) Schema partitioning performed by distributors

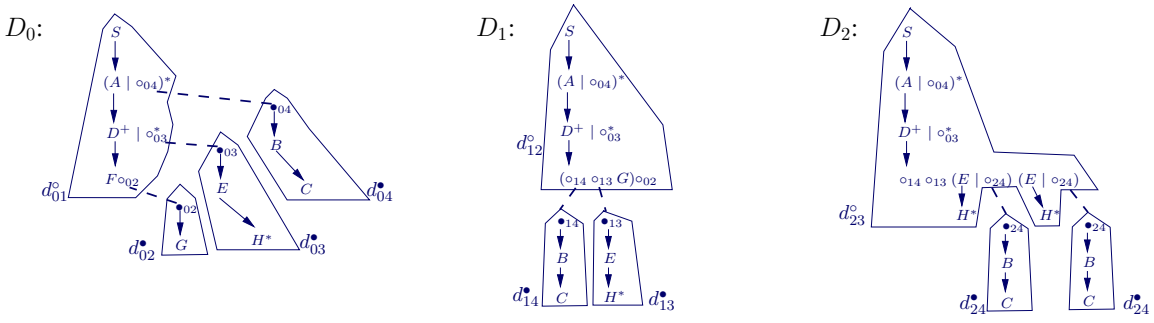


Fig. 1. X-CSR ("X-scissor") in action: (a) pipeline design (unoptimized) with actor *signatures* σ_{A_i} (part of *configurations* Δ_{A_i}), initial input channel schema τ and inferred intermediate schemas (dash-boxed schema trees, above channels) and final inferred schema (after A_3). (b) optimized (system-generated) design: The sub-network $M_2 \rightsquigarrow A_2 \rightsquigarrow D_2$ (M_2 and D_2 reside on the same host as A_2) shows the general pattern: A_2 receives, via the *merger* M_2, all parts relevant for its read-scope, then performs its local operation. The *distributor* D_2 removes parts that are irrelevant to the subsequent actor A_3 and ships them further downstream, but not before leaving marked "holes" behind where the removal was done. This allows downstream mergers to pair bypassed fragments with the holes from which they were originally taken; (c) distributors D_0, D_1, and D_2 split the schemas on the wire as indicated.

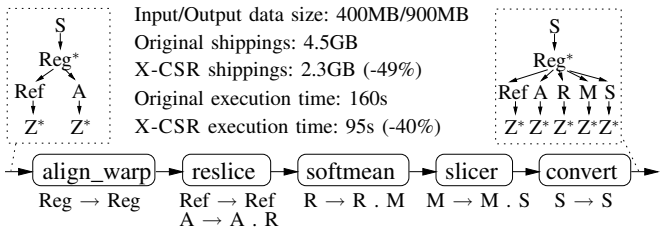


Fig. 2. fMRI workflow with input/output schemas and configurations. Statistics for original and X-CSR-optimized runs.

perform O(1) work for each token they receive. So although collection tokens must travel through roughly three times as many actors in the transformed pipeline, this overhead is also not significant since the majority of the time will be spent on actor computation and shipping data between actors.

V. Experimental Evaluation of X-CSR

Our implementation of the X-CSR optimization has successfully been used to reduce data shipment costs in both real-world scientific workflows and synthetic examples [1]. Here we briefly describe the optimization benefits of X-CSR applied to a workflow for studying functional Magnetic Resonance Imaging (fMRI) results [9], [10] in which voxel images of human brain scans are analyzed.

The workflow consists of the major steps as shown in Fig. 2: align_warp, reslice, softmean, slicer, and convert. In our example, five different regions of the brain are studied with each region grouped under one ⟨Reg⟩ label. A region contains one reference image, which is stored under ⟨Ref⟩, and multiple anatomy images taken from different subjects. Four of these anatomy images are used and each is stored under ⟨A⟩. Each image has associated image files. We model all image files and header files as base data. The first step is to align all anatomy images with the associated reference image (align_warp). This step requires access to all input data and produces a "warp" specification which we store with the anatomy images; we therefore do not modify the collection structure. In the second step, "reslice" takes the reference image data and anatomy image data (with warp information) and creates "resliced" versions of the anatomy images. We put these in a new collection labeled with ⟨R⟩. The signature models access to the reference images (Ref → Ref) and the creation of the new ⟨R⟩ collection as a sibling to the anatomy collection (A → A . R). Then, the "softmean" actor averages all reference images together to create one averaged image ⟨M⟩; which is then sliced to create several (in our example three) 2D images ⟨S⟩. The last step converts each 2-D image to a common image format and stores the result also beneath ⟨S⟩.

For provenance reasons, we want to preserve the intermediate data products and therefore append new data products without deleting the input data in each step. As a consequence, the output schema shown in Fig. 2 contains all input and intermediate data involved in data processing. For performance reasons, each of these steps is run on a single node and data is shipped from node to node. In our example with 400MB of input data, 4.5GB of total data needs to be shipped. Deploying the X-CSR optimization, which sends all intermediate data that is not used by downstream actors to the very end, reduces the amount of data shipped to 2.3GB (a reduction of 49%). The optimization not only saves network bandwidth, but the workflow also executes in 95s as opposed to 160s in the unoptimized case, speeding up execution time by 40%. We averaged the times from 3 workflow runs in which runtimes varied by less than 4 seconds.

Overhead. The startup time of the unoptimized version was 0.4 seconds whereas the optimized workflow needed 0.6 seconds[5] to start up. That is, all compile-time work for X-CSR is performed in less than 0.2 seconds, which is less than 0.2% of the overall execution time (both optimized and unoptimized). We observed similar optimization results on another workflow which we describe in [1].

VI. Conclusion

We have presented a formal model of Δ-XML data types and actor configurations—read scopes, write scopes, and replacement rules—using standard XML typing approaches [11]. Based on Δ-XML, we use a schema propagation approach that is used by X-CSR to determine optimal routings of stream fragments to appropriate actors in a pipeline. We have also described the X-CSR optimization and have shown its effectiveness on a real-world scientific workflow. Specifically, X-CSR can significantly reduce the overall cost of shipping large volumes of data in cases where actors are staged across distributed hosts.

Acknowledgements. Work supported in part through NSF grants IIS-0630033, OCI-0722079, IIS-0612326, DBI-0533368, and DOE grant DE-FC02-01ER25486.

References

[1] D. Zinn, S. Bowers, T. McPhillips, and B. Ludäscher, "Dataflow optimization for distributed XML process pipelines," UC Davis, Tech. Rep. CSE-2008-15, 2008.

[2] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher, "Scientific Workflow Automation for Mere Mortals," *Future Generation Computer Systems*, 2008, in press.

[3] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 4, pp. 660–704, 2005.

[4] M. Mani and D. Lee, "XML to relational conversion using theory of regular tree grammars," *VLDB Workshop on EEXTT*, 2002.

[5] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific Workflow Management and the Kepler System," *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[6] N. Podhorszki, B. Ludäscher, and S. Klasky, "Workflow Automation for Processing Plasma Fusion Simulation Data," in *2nd Workshop on Workflows in Support of Large-Scale Science (WORKS)*, Monterey Bay, June 2007.

[7] A. Brüggemann-Klein and D. Wood, "One-unambiguous regular languages," *Information and Computation*, vol. 142, no. 2, pp. 182–206, 1998.

[8] A. Brüggemann-Klein, "Regular expressions into finite automata," *Theoretical Computer Science*, vol. 120, pp. 197–213, 1993.

[9] Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde, "A notation and system for expressing and executing cleanly typed workflows on messy scientific data," *SIGMOD Rec.*, vol. 34, no. 3, pp. 37–43, 2005.

[10] L. Moreau et al., "The first provenance challenge," *Concurrency and Computation: Practice and Experience*, 2007.

[11] H. Hosoya, J. Vouillon, and B. C. Pierce, "Regular expression types for XML," *TOPLAS*, 2005.

[5] Both measures had a variance smaller than 0.05s throughout 10+ runs