

Extraction of MDE Architectures from Parallel Streaming Applications

Taylor L. Riché and Don Batory
Department of Computer Science
The University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{riche,batory}@cs.utexas.edu

ABSTRACT

We present an incremental approach to extract an MDE architecture from parallel streaming applications. We begin with a sequential pipe-and-filter architecture as our initial model. We then use transformations to *refine* (the standard technique that reveals hierarchical detail), *extend* (expose extra ports, functionality, components, and connectors), and *optimize* (break encapsulation boundaries to achieve efficiency or availability) this model to derive a parallel architecture. We show the generality of our approach with two significant case studies: join parallelizations in database machines and asynchronous crash-fault-tolerant servers.

1. INTRODUCTION

We recently confronted the challenge of re-engineering a complex streaming or dataflow application into a pipe-and-filter architecture with the aim of eventually implementing it on a *Model Driven Engineering (MDE)* platform. Our target was a state-of-the-art *Asynchronous Crash Fault Tolerant (ACFT)* server [12]. We found this server to be so complicated that we needed a way to convince ourselves and others that we understood its design. As we were not domain experts, it was not obvious to us how or why the server worked. We needed a structured way to explain and build our version of this system.

We used *stepwise development (SWD)* to achieve our goals. SWD is a fundamental technique that controls complexity in program development [4, 26, 51] and software architectures [8, 22, 23, 39, 41]. We created an *MDE architecture* using SWD by starting with an executable model of a sequential pipe-and-filter architecture. We then used transformations to *refine* (the standard SWD technique that reveals hierarchical detail), *extend* (expose extra ports, functionality, components, and connectors), and *optimize* (break encapsulation boundaries to achieve efficiency or availability) this model to derive an executable parallel architecture that faithfully captures the decisions made by domain-experts. The result is an easy-to-understand *explanation* and *prescription* of how to recreate that system.

In this paper, we present the results of the first part of our project: an SWD approach to extract an MDE architecture from parallel streaming applications and a demonstration of the generality of our

approach with two non-trivial case studies: the ACFT server and join parallelization in relational database machines. We validate our approach by manually recreating these case studies in the incremental way that we presented them. The next and ongoing phase of our project is to automate these manual tasks on an MDE platform.

The contributions of this paper are:

- **Approach:** We present a novel approach to extract MDE architectures from parallel streaming applications.
- **Architectural Optimizations:** Our case studies presented situations where we had to break abstraction boundaries and then identify and optimize new abstractions to achieve efficient or fault-tolerant designs. Extant SWD literature is silent on such optimizations which are essential to our work.
- **Architectural Extensions:** Classical refinement does not extend application semantics [39]. It is common in SWD to incrementally extend application *and* component semantics.
- **Executable Models:** SWD traditionally starts with a non-executable specification (or model) and ultimately maps it to an executable. In contrast, we start with a *sequential, executable* architecture that we incrementally refine, extend, and optimize to produce a *parallel* architecture with desired performance or availability properties. After each derivation step, our architectures are *always* executable.
- **Case Studies:** We present case studies of two challenging real-world applications and discuss our recreation of them.

2. PREFACE TO OUR WORK

Our case studies are known for their fundamental contributions to fault-tolerance and database machines, respectively. Their designs were *not* conceived in terms of transformations and were *not* built with the aid of software architectural models. However, the novelty, indeed genius, of their designs can be expressed in terms of the transformations or sequence of transformations that were *implicitly used by their authors*. (Exposing these transformations and making them explicit is one of our contributions). There may be no a priori reason or justification for why their authors chose these particular transformations, other than they were necessary for that system or that they introduced a novel algorithm or protocol.

Our explanations are no substitute for domain-expertise; they are intended to complement, encode, and structure domain knowledge for others to follow. Expressing designs by transformations may be novel to domain-experts, but the end result is rarely surprising to them. Further, the assumptions we use are standard fare for their domains. On the other hand, non-experts may find the assumptions

or the choice of transformations unintuitive. What software engineers should take from our paper is a general approach to reveal architectural details of parallel streaming applications incrementally by applying transformations.

Finally, readers may find that the descriptions of our case studies may make them look deceptively simple and straightforward. This clarity was achieved by effort on our part. All told, it took us over four months to refresh our domain knowledge, and to recover and polish the designs for this paper. Patience and some domain expertise are required.

3. DESIGN PRINCIPLES

An *architecture* is a directed graph of boxes and connectors that defines the implementation of a system. A box is a component and a connector is a communication path for messages or tuples pointing in the direction of dataflow.

A sort architecture is an elementary example (see Figure 1(a)). It consists of a single box *SORT* that takes a stream of tuples *A* as input and produces a sorted stream of tuples *sort(A)* as output. *SORT* works by reading stream *A* into memory, sorting its tuples and outputting the sorted stream. *SORT* has other parameters, such as a sort key and a tuple comparison function. We elide these details without loss of generality.

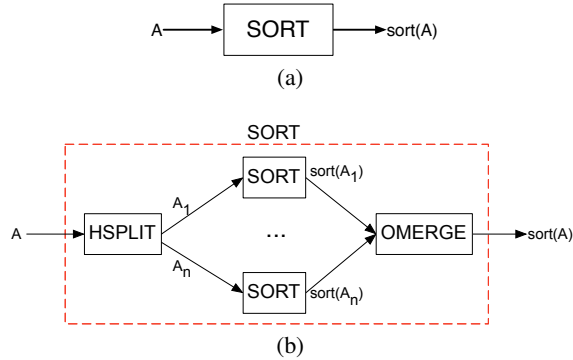


Figure 1: Sort Architecture

An *architectural transformation* or simply *transformation* is a mapping of an input architecture to an output architecture. In effect, a transformation is a graph rewrite. We use three kinds of transformations: refinements, optimizations, and extensions.

3.1 Refinements

A *refinement* exposes hierarchical implementation detail of a single box. Figure 1(b) shows a classical database refinement of a *SORT* box that shows how *SORT* is parallelized by *map-reduce* [14, 21]: the input stream *A* is hash-partitioned on keys by the *HSPLIT* box, producing substreams $A_1 \dots A_n$. All tuples in stream A_i hash to value $i \in 1 \dots n$. Each substream is sorted on the same sort key, producing sorted streams $sort(A_1) \dots sort(A_n)$ which are then combined by an ordered merge, producing stream $sort(A)$.

3.2 Optimizations

An *optimization* breaks encapsulations to realize non-functional properties (e.g., efficiency or fault-tolerance). Consider the projection-sort architecture of Figure 2(a). A stream of tuples *A* enters projection box *PROJ*, where *PROJ* removes unnecessary fields from *A* tuples. The resulting stream A' is then sorted, yielding stream *B*.

Both *PROJ* and *SORT* boxes are refined individually to their parallel counterparts in Figure 2(b). Both *HSPLIT* boxes partition

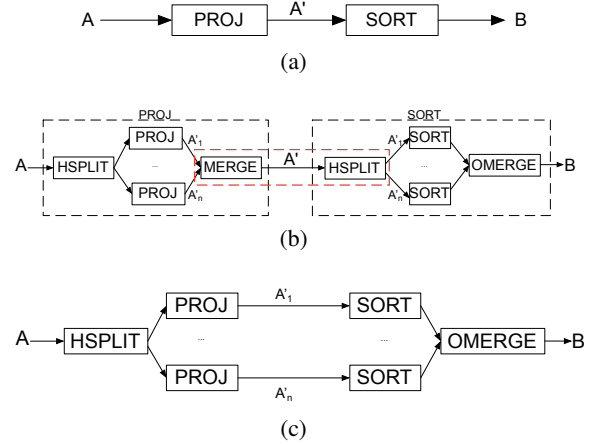


Figure 2: Refinement and Optimization

streams using the same hash function. Note that the *MERGE* box serializes substreams $A'_1 \dots A'_n$ into stream A' and then *HSPLIT* reconstructs *these same substreams*. An optimization removes the *MERGE* and *HSPLIT* pair, as their composition is the identity map. Figure 2(c) shows the optimized projection-sort architecture.

Another optimization is a *rotation*. A rotation reorders stateless computations. Figure 3(a) shows substreams $A_1 \dots A_n$ are combined by a *MERGE* (which is part of one abstraction, the incomplete box on the left) and split into substreams $B_1 \dots B_k$ by a *HSPLIT* (which is part of another abstraction, the incomplete box on the right). Unlike the previous *PROJ* – *SORT* example, here the composition of *MERGE* and *HSPLIT* (the abstraction in the dashed box) is *not* the identity map as streams *A* and *B* are split using different hash functions. The key property of Figure 3(a) is that each tuple of an A_i substream is hash-routed to a unique B_j substream. This property is preserved by rotating (swapping) the composition order of *MERGE* and *HSPLIT* (Figure 3(b)): each substream A_j is *HSPLIT* into substreams $A_{j1} \dots A_{jk}$ and then substreams $A_{1j} \dots A_{nj}$ are combined by *MERGE* to form substream B_j for all $j \in 1 \dots k$. Rotations are essential in the architectures that we examine.

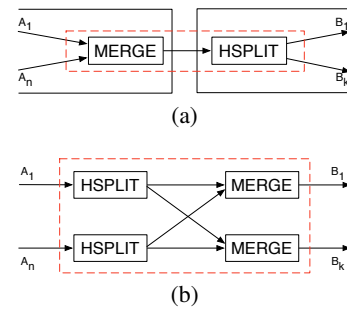


Figure 3: Rotations

3.3 Extensions

A *model or architecture extension* augments the semantics of a model by adding new capabilities and ports to existing boxes and adding new boxes and connectors. A common special case is *box extension*, where new capabilities and ports are added to a box. Extending a box is equivalent to adding one or more features, which can be accomplished by preprocessors (e.g. `#ifdef` inclusion of

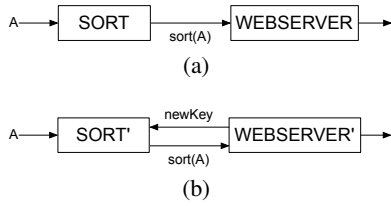


Figure 4: Extensions

extra code) or by more sophisticated means [4]. We say that a box X is *extended* to Y (written $X \rightsquigarrow Y$).

Figure 4(a) shows an architecture with a *SORT* and *WEBSERVER* box. The *WEBSERVER* takes sorted tuples and creates a webpage of the sorted results. We extend this architecture to Figure 4(b) by $SORT \rightsquigarrow SORT'$ and $WEBSERVER \rightsquigarrow WEBSERVER'$ with new ports and adding a feedback connector labeled *newKey*. A *newKey* message changes the key that *SORT'* uses to sort the incoming tuples (e.g. switching from last names to ID numbers in a patient database or artists to album titles in a music player).

Refinement is different than extension. Refinement preserves box semantics; it maps an abstract box to a streaming architecture that preserves the semantics of the abstract box. The abstract box is not altered and new relationships external to this box are not added. In contrast, model extension enhances the semantics of a model by elaborating existing boxes with new ports, new functionality, and new boxes and connectors. Well-known tools for data flow architectures, like Labview [41], Weaves [23], and Click [30] support refinement, but *not* extension. Our use of the terms refinement and extension follows that used in formal methods [45].

3.4 Model Executability

Our models are always executable. This can be seen in Figure 1(a). A *SORT* box is created (typically by hand) to make the architecture in Figure 1a executable. When we refine to Figure 1b, we create new *HSPLIT* and *OMERGE* boxes and link them with *SORT* boxes. The result is another executable architecture.

An appealing property of refinements is that we can reuse input-output tests of the *SORT* box as “integration” tests for the architecture of Figure 1(b). That is, logically we should not be able to distinguish the input-output response of a single *SORT* box from its parallel sorting counterpart or other refinements.

Similar arguments hold for optimizations and extensions. We can reuse tests for a box to ensure that its input-output response holds for each box extension. Of course, new tests must be created to evaluate the correctness of extensions to a box’s functionality. In any case, at all times our models are executable. In future work, we explore how our approach not only leads to a way of *recreating* a system but also *testing* it as well.

In the next sections, we demonstrate the power of these simple ideas by using them to explain sophisticated streaming architectures that were created by experts in very different domains.

4. HASH JOINS IN DATABASE MACHINES

Gamma was (and perhaps still is) the most sophisticated relational database machine built in academics [15]. It was created in the late 1980s and early 1990s without the aid of modern software architectural models. We focus on Gamma’s join parallelization, which is typical of modern relational database machine designs. What is new in this section is our presentation of Gamma. Published descriptions are informal [15]; our presentation is a derivation from first principles.

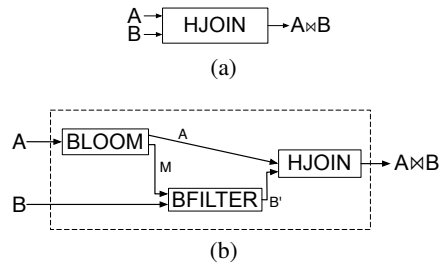


Figure 5: Hash Join Architecture

A *hash join* is an implementation of a relational equi-join; it takes two streams (A, B) of tuples as input and produces their equi-join $(A \bowtie B)$ as output. The basic hash join algorithm is simple: read all tuples of stream A into a main-memory hash table, where the join key of A tuples are hashed. Then read stream B , one tuple at a time. By hashing a B tuple’s join key, one can quickly identify all A tuples that join with the B tuple. This algorithm has linear complexity in that each A and B tuple is read only once. Figure 5(a) shows the executable *HJOIN* architecture that we start with.

4.1 Bloom Filtering Refinement

Joins are among the most expensive database operations. Gamma makes an ingenious use of Bloom filters [7] to reduce the number of tuples to join. It uses two new boxes: *BLOOM* (to create the filter) and *BFILTER* (to apply the filter). This refinement of *HJOIN* is shown in Figure 5(b).

Here’s how the refinement works: the *BLOOM* box takes a stream of tuples A as input and outputs exactly the same stream A along with a bitmap M . The algorithm first clears M . Each tuple of A is read, its join key is hashed, the corresponding bit (indicated by the hash) is set in M , and the A tuple is output. After all A tuples are read, M is output. M is the *Bloom filter*.

The *BFILTER* box takes Bloom filter M and a stream of tuples B as input, and eliminates B tuples that cannot join with A tuples. The algorithm begins by reading M . B is read one tuple at a time; the B tuple’s join key is hashed, and the corresponding bit in M is checked. If the bit is unset, the B tuple is discarded as there is no A tuple to which it can be joined. Otherwise the B tuple is output. Stream B' is the result. Given the behaviors of the *BLOOM*, *BFILTER*, and *HJOIN* boxes, it is easy to prove the refinement of Figure 5(a) to Figure 5(b) does indeed produce $A \bowtie B$ [3].

4.2 Parallelizing Refinements

Next, we refine the *BLOOM*, *BFILTER*, and *HJOIN* boxes by replacing each with their map-reduce versions (Figure 6). A *BLOOM* box is parallelized by hash-splitting its input stream A into substreams $A_1 \dots A_n$, creating a *BLOOM* filter $M_1 \dots M_n$ for each substream, coalescing $A_1 \dots A_n$ back into A , and merging bit maps $M_1 \dots M_n$ into a single map M .

A *BFILTER* box is parallelized by hash-splitting its input stream B into substreams $B_1 \dots B_n$, where the same hash function that splits stream A is used to split stream B . Map M is decomposed into submaps $M_1 \dots M_n$ and substream B_i is filtered by M_i . The reduced substreams $B'_1 \dots B'_n$ are coalesced into stream B' .

The parallelization of the *HJOIN* box is textbook [2]: both input streams A, B are hash-split on their join keys using the same hash function as before. Each stream A_i is joined with stream B_j ($i, j \in 1 \dots n$), yielding n^2 *HJOIN* boxes. Since an equi-join is computed, we know $A_i \bowtie B_j = \emptyset$ for all $i \neq j$ (as equal keys must hash to the same value). Thus, a single abstract *HJOIN* box is replaced by n

HJOIN boxes instead of n^2 boxes as all other *HJOIN* boxes have provably null outputs. By merging the joins of $A_i \bowtie B_i$ ($i \in 1 \dots n$), $A \bowtie B$ is produced as output.

Figure 6 shows the result of applying all three parallelization refinements to Figure 5(b). Each of the *BLOOM*, *BFILTER*, and *HJOIN* parallelizations have proofs of correctness [3].

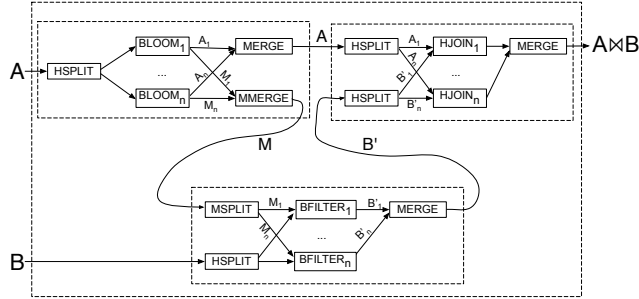


Figure 6: Parallel Refinements

4.3 Optimizations

A primary goal of Gamma was to determine performance increases in joins that could be gained by parallelization. The architecture of Figure 6 has three *serialization bottlenecks* which degrade performance. Consider the *MERGE* of substreams $A_1 \dots A_n$ into A , followed by a *HSPLIT* to reconstruct $A_1 \dots A_n$. There is no need to materialize A : the (*MERGE*, *HSPLIT*) pair is the identity map: $A_i \rightarrow A_i$ ($i \in 1 \dots n$). The same applies for the (*MERGE*, *HSPLIT*) pair for collapsing and reconstructing substreams $B'_1 \dots B'_n$. The removal of (*MERGE*, *HSPLIT*) pairs eliminates two serialization bottlenecks.

The third bottleneck combines maps $M_1 \dots M_n$ into M and then decomposes M back into $M_1 \dots M_n$. The (*MMERGE*, *MSPLIT*) pair is also the identity map: $M_i \rightarrow M_i$ ($i \in 1 \dots n$). This optimization removes the (*MMERGE*, *MSPLIT*) boxes and reroutes the streams appropriately.¹

Figure 7 shows the result of all three optimizations, which is the design Gamma uses to parallelize a single join. Also note Figure 7 is an executable architecture, as are all architectures that we present. Remember that users provide the source code for each box. Tools, such as Labview [41], Weaves [23], or Lagniappe [43] can stitch the boxes together given an architectural model to produce an executable.

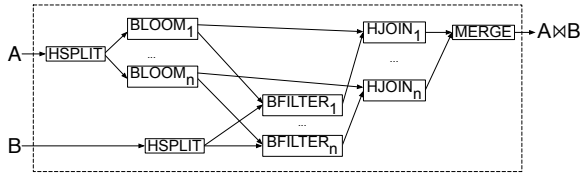


Figure 7: Hash Join Optimizations

¹ There are many ways in which *MMERGE* and *MSPLIT* can be realized. The simplest is this: M is a $n \times k$ bitmap. The join key of an A tuple is hashed twice: once to determine the row of M , the second to determine the column within the selected row. Thus, all tuples of substream A_i hash to row i of M . *MMERGE* combines $M_1 \dots M_n$ into M by boolean disjunction. For each i , *MSPLIT* extracts row i from M and zeros out the rest of M_i .

4.4 Cascading Joins

Figure 7 is not the last word on Gamma's architecture. Rotations Section 5.2.3 are used to optimize the processing of cascading joins, where the output of one join becomes the input of another (see Figure 8(a)). Figure 8(b) reveals the partial internals of these *HJOIN* boxes where the output of the first join is formed by merging substreams $C_1 \dots C_n$ into stream C and then C is immediately hash-split into substreams $D_1 \dots D_k$. This is another serialization bottleneck. Unlike the bottlenecks in Section 4.3, cascading joins use different join keys, so that the partitioning of C before its merge is different than the partitioning of C after the hash-split ($n \neq k$).

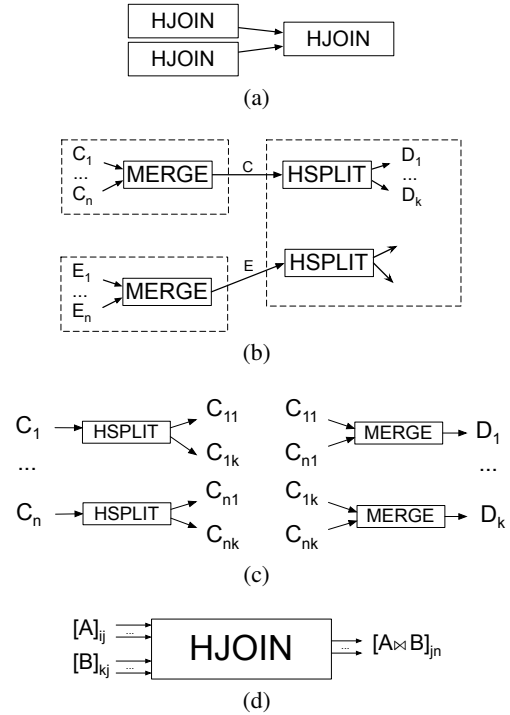


Figure 8: Rotation of *MERGE* and *HSPLIT*

Here is where refinement is insufficient to derive a streaming architecture; encapsulation boundaries must be broken to eliminate serialization bottlenecks. Rotations remove these bottlenecks, swapping (*MERGE*, *HSPLIT*) pairs with appropriate (*HSPLIT*, *MERGE*) pairs (Figure 8(c)). Each C_i is hash-split into k substreams ($C_{i1} \dots C_{ik}$) and sets of n substreams ($C_{1j} \dots C_{nj}$) are merged into stream D_j ($j \in 1 \dots k$). This rotation preserves the property that tuples of C whose hash-value is j are assigned to stream D_j , while eliminating a serialization bottleneck. A drawback is bookkeeping: between the *HSPLIT* and *MERGE* boxes is an $n \times k$ matrix of substreams, which we denote by $[C]_{nk}$.

Gamma's *HJOIN* architecture generalizes from Figure 5(a) to Figure 8(d): a single *HJOIN* box takes matrices $[A]_{ij}$ and $[B]_{kj}$ of substreams as input (stream A is hash-partitioned into $i \times j$ disjoint substreams and B into $k \times j$ disjoint substreams) and produces a matrix $[A \bowtie B]_{jn}$ of substreams as output ($A \bowtie B$ is hash-partitioned into $j \times n$ disjoint substreams). Rotations arise in parallel database architectures generally and Gamma's architecture in particular.

4.5 Validation

We implemented the complete Gamma architecture by hand as described above using Java threads and pipes. A more convincing

validation was giving the design/derivation of Figure 7 as an assignment to an upper-division software engineering class of 30 students in Spring 2010. Other than discovering that serialized objects do not work well with Java pipes (but if String-based serializers are used there is no problem), the assignment was straightforward, confirming that our MDE architecture was both explanatory and prescriptive for system reconstruction.

In presenting this material to graduate database students, we observed that it is easier to remember the *derivation* of Gamma’s architecture than the designs of Figure 7 or Figure 8d. Implementing and testing Gamma hash joins is an interesting exercise in incremental development: we can be assured at every step that our models are provably correct and with tests that our implementations are demonstrably correct.

5. ACFT SERVERS

Our original interest in streaming architectures stemmed from the desire to transform a vanilla server to a *Synchronized Crash Fault Tolerant (SCFT)* server and then to a ACFT server. Several hard-coded mappings are used in practice [24, 53]. *Here we describe a recently proposed mapping that is used in state-of-the-art fault-tolerant servers* [12]. We begin by reviewing client-server basics. *Readers should note the same principles for architectural refinement and optimization that we used in developing Gamma’s architecture are used here in developing the ACFT architecture.*

Note: Hash joins require a modicum of domain-knowledge to explain; ACFT servers require a bit more. We indent additional ACFT-specific comments into **Notes** to distinguish them from our domain-independent discussions.

5.1 Client-Server Basics

We consider *request-processing applications (RPAs)* with multiple clients sending requests (a.k.a. *messages*) to a server. Client requests can read or write the server’s internal state, which persists across requests. For each request, the server receives the message, updates its state, and sends a response back to the client. That servers have state is important: *crash-fault tolerance (CFT)* of non-stateful servers is trivial.²

RPAs have a cylinder topology representing the cyclic flow of request-response. We unroll the cylinder, as shown in Figure 9a, by breaking the seam along dotted lines. Figure 9b shows a typical architecture with clients $C_1 \dots C_n$ and server S . Each client sends messages to the server. Messages from different clients are serialized into a single stream by multiplexer \triangleright . The stream of responses from the server are demultiplexed into multiple output streams, one per client, by \triangleleft . Response messages wrap around the dotted lines.

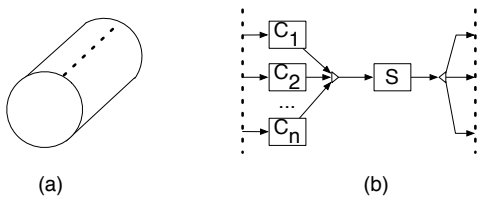


Figure 9: Unrolled Cylinder Topology

Note: A CFT server must uphold two properties: safety and liveness. Loosely, *safety* means the server is always in a “good”

²Replicating a stateless application is trivial as consistency across replicas is guaranteed by the fact that the replicas never change.

state, for varying definitions of “good”, and *liveness* means that the server eventually makes progress [40]. The server designs that we study are safe by providing a serializable view of client-request processing [42]; serialization proofs of the protocols we use are given in [32]. Further, these protocols uphold a liveness property called “eventual liveness” [12]. As consensus in an asynchronous system is impossible [19], the existence of occasional periods of a well-behaved network is assumed, thus allowing both SCFT and ACFT servers to make progress.

5.2 Synchronous CFT

Crash fault tolerance is the ability of a service to survive a number of failures. A *crash failure* occurs when a box stops processing messages—no messages pass through a failed box and a failed box cannot create new messages. The failure of network boxes— \triangleright , \triangleleft , \odot (reliable broadcast [40]), and \bullet (broadcast)—are treated identically to software boxes. Failure is self-contained, meaning that failures do not propagate across box boundaries.

Note: In this paper we assume that each box executes on its own machine. Normally, multiple boxes are mapped to a single machine. The rules for machine failure are simple: if a machine fails, all boxes on that machine fail.

Note: All requests and boxes in the system are considered benign. Malicious behavior (or even benign behavior outside the accepted application protocol) is assumed to not occur. Work on *Byzantine Fault Tolerance (BFT)* relaxes these assumptions [11, 12, 33]. *In any case, the mapping of a vanilla client-server model to a SCFT model is unaffected.* BFT transformations begin with an ACFT model and map to a BFT model.

The technical objective of SCFT is to eliminate *Single Points of Failure (SPoF)* by replicating functionality [44]. An SPoF is the failure of a single box that causes the entire server abstraction to fail. Our initial design (Figure 9(b)) has three SPoFs: the server S , the serializer \triangleright , and the demultiplexer \triangleleft . Client failures should never cause a server abstraction to fail in an SCFT model as servers should expect and gracefully handle non-responsive clients.

5.2.1 Agreement Refinement

The first transformation $SCFT_1$ adds a single node A^\perp between the clients and server. The *agreement node* A^\perp implements an ordered queue of messages, collecting messages from clients and passing messages one at a time to the server. In effect, A^\perp makes the network queue explicit, materializing a placeholder for subsequent refinements. Figure 10 shows the architecture after $SCFT_1$.

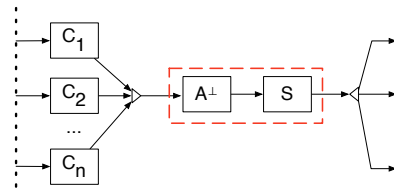


Figure 10: After the $SCFT_1$ Refinement

5.2.2 Replication Refinements

The next transformations, $SCFT_{2a}$ and $SCFT_{2b}$, replicate the agreement and server boxes to improve system availability, i.e. to make the server abstraction more resilient to crashes. See Figure 11.

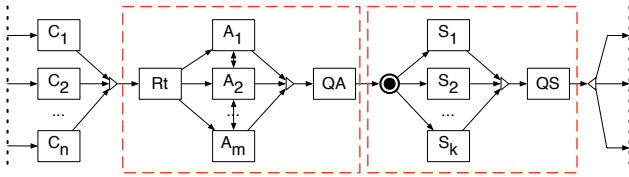


Figure 11: Fully Replicated A^\perp and S Abstractions

First consider the replicating-server refinement $SCFT_{2b}$. It replicates the server S k -times, indicated by boxes $S_1 \dots S_k$, and adds three new boxes: \odot , \triangleright , and QS . Box \odot reliably broadcasts an incoming message to each server replica: if one server replica receives a message then all server replicas receive the message—such a strong guarantee is necessary for correctness. Replicas receive the message, update their state, and send responses. \triangleright serializes all responses and box QS collects a quorum of identical responses. Once QS receives matching responses from a sufficient number of S replicas, QS transmits a single response to the client, thus maintaining the abstraction of a single server.

Now consider the replicating-agreement refinement $SCFT_{2a}$. It (1) extends $A^\perp \rightsquigarrow A$ to implement an agreement protocol, (2) replicates A m -times as boxes $A_1 \dots A_m$, and (3) adds a routing box Rt before the replicas and a serializer \triangleright and quorum box QA after the replicas. Rt forwards incoming client messages to a subset of A replicas, where the actual subset is determined by an agreement protocol. The replicas $A_1 \dots A_m$ implement an agreement protocol that guarantees a quorum-decided linear order in which client messages should be processed [34, 35]. A replicas communicate with each other to determine the next client message to process. Each A replica votes, sending its “next” message to the quorum box QA . QA forwards a single message to the server if it receives identical messages from a sufficient number of replicas. In this way, $SCFT_{2a}$ maintains the behavior of and interface to a single A^\perp box.³

In summary, domain experts replicate the A box m -times and the S box k -times. The specific values of m and k depend on the number of faults f to tolerate and the agreement protocol. Common assignments set $m = 2f + 1$ and $k = f + 1$.

Note: Yin et al. explain this difference in replica count by noticing that the quorums necessary to prove the protocols correct are larger for messages coming from the agreement portion (QA) than for messages coming from the execution (QS) [52]. Since a server, on average, requires more computational resources than agreement, and thus needs a more powerful and more expensive machine, using fewer server replicas in the architecture is desirable.

$SCFT_{2a}$ and $SCFT_{2b}$ commute as the order in which they are applied does not matter. However, both must be applied to guarantee that the system tolerates the failures of up to f server and f agreement boxes. Figure 11 is the result of applying them to Figure 10.

³ Observe that $SCFT_{2a}$ refines abstract box A^\perp to a streaming architecture that requires concrete box A^\perp to be extended to A ($A^\perp \rightsquigarrow A$). Object-oriented programmers would recognize this as inheritance: A^\perp is a superclass and A is its subclass. The semantics of A^\perp does not change, but A extends both A^\perp 's semantics and implementation. Thus, there can be a very close relationship between refinement and extension.

5.2.3 Optimizations

Our original architecture of Figure 9(b) had three SPoFs; our current design in Figure 11 has eight! (From left to right in Figure 11, they are the \triangleright , Rt , \triangleright , QA , \odot , \triangleright , QS , and \triangleleft boxes.) Here is where refinement is insufficient to derive a streaming architecture; encapsulation boundaries must be broken to remove SPoFs.

In Figure 12 we dissolve existing abstraction boundaries and identify three new abstractions that circumscribe two or three boxes, all of which are SPoFs. For example, the left-most abstraction contains SPoF boxes \triangleright and Rt . The middle contains three SPoF boxes (\triangleright , QA , and \odot). And the right-most contains three SPoF boxes (\triangleright , QS , and \triangleleft). In the following, we explain how rotations revise the implementations of each abstraction to eliminate SPoFs.

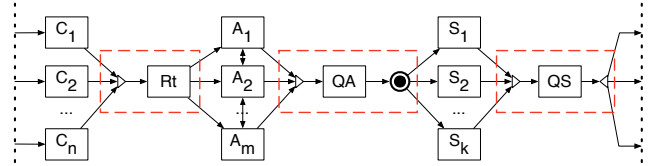


Figure 12: Abstractions with Multiple SPoFs

Consider the left-most abstraction of Figure 12, which we show again in Figure 13(a) comprising the sequence (\triangleright , Rt). The $SCFT_{3a}$ rotation swaps the order of these boxes (see Figure 13(b)). Box Rt is replicated once for each client and \triangleright is replicated once for each A_i . Instead of serializing all requests and then routing, we route client requests immediately and serialize requests before each A replica. The property that each client request is sent to a subset of A replicas is preserved by $SCFT_{3a}$. As expected, the interface of n input channels and m output channels is maintained. But now, Rt and \triangleright boxes are no longer SPoFs in Figure 13(b).

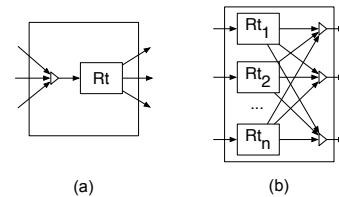


Figure 13: The $SCFT_{3a}$ Rotation

The middle abstraction of Figure 12, replicated in Figure 14(a), consists of the box sequence (\triangleright , QA , \odot). Transformation $SCFT_{3b}$ is a pair of rotations that modify the order (\triangleright , QA , \odot) to (\triangleright , \odot , QA) (Figure 14(b)) and finally to (\odot , \triangleright , QA) (Figure 14(c)). Instead of taking a quorum of responses from A replicas and reliably broadcasting the result, we reliably broadcast the results of all A replicas and take a quorum at each server replica. The property that a quorum-decided request from replicated A boxes is delivered to all server replicas is preserved by $SCFT_{3b}$.

One last optimization is needed: it is non-obvious but well-known to domain experts. Reliable broadcast is very expensive.⁴ By replicating \odot at each of the A_i boxes, we can take advantage of the fact that quorums are taken at each server and replace the reliable broadcast box \odot with the normal (unreliable) network broadcast box \bullet that is simple to implement. The abstraction in Figure 14(d) now contains no SPoFs, and also represents a standard and efficient way to implement a reliable crossbar [12].

⁴In fact, experts often use an agreement cluster to implement reliable broadcast!

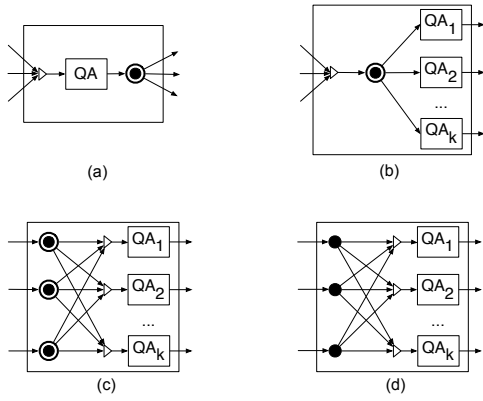


Figure 14: The $SCFT_{3b}$ Rotations and The Reliable Broadcast Optimization

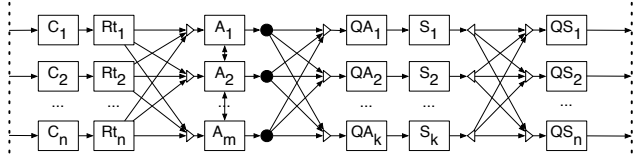


Figure 15: Synchronous Crash Fault Tolerant Architecture

Similarly, transformation $SCFT_{3c}$ is a pair of rotations that is applied to the right-most abstraction in Figure 12, which maps the box sequence $\langle \triangleright, QS, \triangleleft \rangle$ to $\langle \triangleright, \triangleleft, QS \rangle$ and then to $\langle \triangleleft, \triangleright, QS \rangle$, thereby eliminating all SPoFs. Figure 15 is the result of applying $SCFT_{3a}$ – $SCFT_{3c}$ to Figure 12.

5.3 Recovery from Asynchrony

Up to this point, failure is permanent. If a box fails, it no longer responds and has no hope of ever responding again. An SCFT architecture supports f failures of A replicas and f failures of S replicas. When either limit is exceeded, the entire system is in a failed state, the one-correct-server abstraction is violated, and safety and liveness are no longer guaranteed.

In this section, we explain how domain-experts relax restrictions on liveness. An ACFT architecture guarantees liveness even with an occasional poorly behaved network or temporary box failure (and always guarantees safety), assuming the network eventually enters a sufficiently long well-behaved period. That is, liveness is guaranteed *even with occasional network asynchrony*.

In effect, ACFT limits the situations where a client sees an unresponsive server abstraction. The mechanisms that support asynchrony also allow the architecture to mask some failures by enabling a box to be restarted and catch up. A box that fails and recovers is considered correct, albeit slow. The other boxes will continue to make progress; a recovering box may not be able to catch up immediately. If other boxes fail, the remaining fast servers may have to wait for the recovering servers to catch up (as required by quorum boxes), at which point the system can resume servicing client requests [12]. Note, boxes may still *crash* and enter a state where they will never again process requests; asynchrony support cannot mask all failures.

5.3.1 Where We Are Going

Figure 16 shows a road map (or commuting diagram) of where we are and where we are going. We began with the topmost left

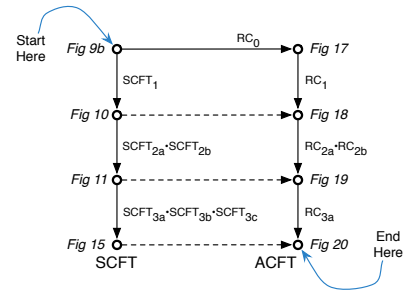


Figure 16: Roadmap in Our ACFT Designs

node (labeled Figure 9b) and incrementally refined and optimized downward until we reached the bottommost left node (Fig. 15). In the next section, we extend our initial architecture of Figure 9b by horizontally mapping it to Figure 17, and then applying a series of vertical refinements and optimizations similar to those used earlier to produce the ACFT architecture of Figure 20.

All horizontal arrows in Figure 16 are architecture extensions. Although we discuss only the top-most extension RC_0 , the other dashed extensions do indeed exist. Readers may notice that the right-side vertical transformations (e.g. RC_1) that we will shortly introduce can be explained as extensions to their corresponding left-side vertical transformations (e.g. $SCFT_1$).

5.3.2 ACFT Server Extension

RC_0 extends a server box S that has no asynchrony recovery support to a server box RS that does ($S \rightsquigarrow RS$). Recovering from asynchrony includes both logging and checkpointing, and the ability to load a checkpoint and replay the log upon restart. Most modern request-processing servers implement this functionality, such as Hadoop [24] and Zookeeper [53]. The extension $S \rightsquigarrow RS$ adds a “recovery” feature to a server to correctly implement the checkpointing of application-specific data structures. Figure 17 shows the result of applying RC_0 to Figure 9b.

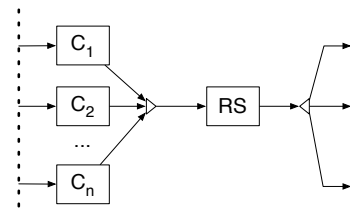


Figure 17: After Extension RC_0

5.3.3 Asynchronous Agreement Refinement

Recall that refinement $SCFT_1$ introduces an A^\perp box in front of S . A^\perp is part of the greater abstraction of one correct server. For the entire abstraction to be recoverable, all of its stateful internal boxes must be recoverable, thus the functionality of A^\perp must be extended to RA ($A^\perp \rightsquigarrow RA$). Further, the recovery algorithm of the server RS is extended, as it needs information from the RA box to fully recover. See Figure 18. This “extra” information is not essential at this stage of design, but like the introduction of the A box, it is a placeholder for subsequent refinements to elaborate. Providing this “extra” information is part of $RS \rightsquigarrow RS'$.

The other boxes in Figure 10 that flank the A^\perp - S abstraction, namely \triangleright and \triangleleft , are unaffected by recovery as they are stateless; the recovery of a stateless box is a simple restart.

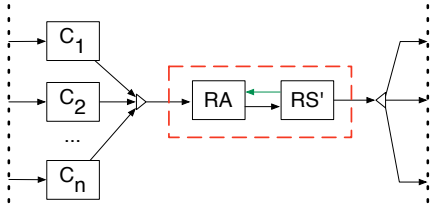


Figure 18: After Transformation RC_1

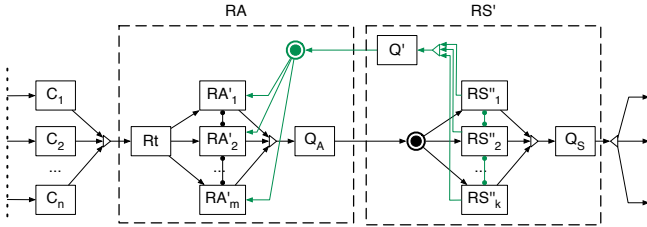


Figure 19: After the RC_{2a} and RC_{2b} Transformations

In summary, refinement RC_1 maps box RS of Figure 17 to that shown in Figure 18. This implementation uses two extended boxes $A^\perp \rightsquigarrow RA$, $RS \rightsquigarrow RS'$, and adds a new connector from RS' to RA to allow RS' to query RA for checkpoint information.⁵

5.3.4 Replication Refinements

The next refinements, RC_{2a} and RC_{2b} , respectively add asynchrony recovery to replicated agreement and server boxes. Although RC_{2a} and RC_{2b} are commutative, they are dependent on $SCFT_{2a}$ and $SCFT_{2b}$ respectively as they extend those refinements. Figure 19 shows the result of applying RC_{2a} and RC_{2b} to Figure 18.

Server Replication.

When a recoverable server RS' is replicated, it needs to be able to communicate with other server replicas. The reason lies in the recovery algorithm for replicated servers: when a server replica crashes, other servers continue to make progress. When the failed server recovers—it may be behind—other client messages may have been processed in the interim. For the failed server to catch up, it must fetch checkpoint state from other server replicas.

Refinement RC_{2b} maps box RS' of Figure 18 to the implementation shown in Figure 19. This implementation uses an extended box $RS' \rightsquigarrow RS''$ to account for server recovery (discussed above) and for the following situation. After processing a fixed number of client messages, the agreement box asks the server for its current checkpoint. The agreement box can only accept a checkpoint if it receives a quorum of matching checkpoints from server replicas.⁶ Further, when a server crashes and attempts to recover, it asks the agreement box for the latest checkpoint. This “Help, I need to recover!” message comes from just one server, and the agreement box does not wait for a quorum (as one will never come). A special quorum box Q' (a) takes quorums of checkpoint messages from servers and (b) immediately passes along recovery messages of servers that have fallen behind.

⁵We represent multiple communication channels in the same direction with just one connection for brevity.

⁶By receiving occasionally checkpoints from the server, the agreement nodes can implement recovery without the requirement of an infinite replay log. This optimization is obviously desirable.

Agreement Replication.

Once $SCFT_{2a}$ replicates agreement box RA , the transformation RC_{2a} extends replicated RA boxes to RA' by adding new message handlers such as one for receiving the current checkpoint on which the RA' replicas must agree.⁷ Further, a reliable broadcast (\odot) is introduced that sends all incoming “Help” messages from the server abstraction to all agreement replicas $RA'_1 \dots RA'_m$.

5.3.5 Optimizations

Once again, we are at a point where refinement is inadequate to complete our design; optimizations are still needed. In Figure 19 there is a sequence of boxes ($\triangleright, Q', \odot$) that connect server replicas to agreement replicas. All three boxes are SPoFs and need to be replaced by the reliable crossbar abstraction highlighted in Section 5.2.3.

Transformation RC_3 is a pair of rotations that modify the order ($\triangleright, Q', \odot$) to ($\triangleright, \odot, Q'$) and then to ($\odot, \triangleright, Q'$). Further, RC_3 replaces the reliable broadcast \odot with the normal broadcast \bullet . RC_3 removes SPoFs and preserves the property that all replica RA' boxes receive quorum-decided messages from server replicas (or “Help!” messages from recovering servers). Figure 20 is our final ACFT server architecture.

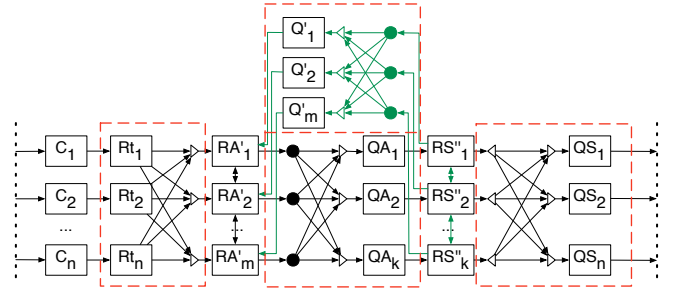


Figure 20: Asynchronous Crash Fault Tolerant Architecture

5.4 Validation

We implemented both the SCFT and ACFT architectures by hand, incrementally as described above, using Python 3 atop a custom-built, light-weight streaming-application framework (code for this case study can be downloaded from [20]). Each box was a process; messages were transferred via sockets. As we did with Gamma, we tested the initial architecture and each subsequent architecture that was derived—all tests passed thus validating our transformations. As future work we hope to collaborate with our ACFT colleagues to formally prove the correctness of individual transformations. As mentioned earlier, Clements et al. have a proof of the *composite* transformation for the ACFT architecture [12], but not proofs for *individual* transformations. Although special cases of the ACFT proof reduce to proofs of specific compositions of transformations, it is unclear whether proofs of individual transformations are ultimately simpler than proofs of a single composite transformation.

Interestingly, rotations were unfamiliar to our ACFT colleagues. Their informal designs went directly from Figure 10 to Figure 15, avoiding our incremental construction of a reliable crossbar. This suggests how we can explain intuitive designs in a principled way.

⁷As part of the transformation, the agreement protocol must now not only agree upon the next client request to transmit, but also on the checkpoint to save to stable storage.

6. RELATED WORK

Twenty-five years ago, we modeled the architectures of commercial database systems as a composition of refinements [5], what we now call an MDE architecture. We have since shown that similar ideas generalize to describe software product lines [4, 47]. We expect, and have no reason to question, that the ideas (refinement, extension, and optimization) presented here will apply to domains beyond parallel streaming architectures.

Our approach is MDE: we create high-level models of streaming architectures and transform them into low-level models of streaming architectures. What is unusual is our reliance on *endogenous transformations*—transformations whose domain and co-domain are the same. Most of the MDE literature focuses *exogenous transformations*—mappings whose domain and co-domain are different [37]. For example, each architectural style has its own meta-model. Prior work mapped architectures of one style to an architecture of another [1, 17, 25, 48]. Endogenous transformations have been used sparingly, *not for incremental architectural development as we do*, but to simulate architecture execution (e.g., adding and removing clients in a client-server system) [10, 38]. The few cases where endogenous transformations are composed to produce MDE designs [54, 47, 49] deal with simple transformations that encode extensions as model deltas, not refinements and optimizations that we present.

There is a rich collection of papers on architecture refinement; we limit our discussions to key papers for lack of space. Traditional approaches start with an abstract architecture or specification and then apply refinements to progressively expose more hierarchical detail on how the abstract architecture is implemented. Some researchers have shown that their refinements can be verified, not violating any of the original design’s properties [6, 38, 25, 39]. Our work differs from traditional refinement in three ways: 1) We start with an executable architecture and apply endogenous transformations to expose implementation details and maintain architecture executability. 2) We are unaware of prior SWD work that uses architectural optimizations, in particular rotations, in the development of streaming systems. And 3) the role of extension in SWD architectural design is under-appreciated. Broy seems to lay the mathematical foundations for extension [8], but we are unaware of a recent system that puts his ideas into practice.

Practical tools for building streaming architectures also have a long history. Labview [41] (marketed since the mid-1980s) and Weaves [23] are platforms for executing streaming architectures. Refinement is their sole abstraction; *user-defined optimizations and extensions are absent*. Other component-based systems follow a similar approach [9, 13, 22, 50]. More recent tools are StreamIt [46] and Click [30]. StreamIt is a general language for expressing stateless streaming applications; it uses map-reduce for automatic parallelization. Click demonstrated the feasibility of programmable routers using streaming architectures. With few exceptions [23, 30, 46], all of the above-cited tools are aimed at *bottom-up development*; *none handle architectural optimizations or extensions*. In short, our work describes an MDE way to synthesize domain-specific streaming architectures that could be implemented by these tools, so our contribution is more methodological than tool-specific.

Our work is an example of software architecture recovery [16, 31]. Classic research focuses on tools, data exchange formats, and metrics for extracting and clustering information from source (code, makefiles, documentation, etc.) and application execution traces to reconstruct an architecture [18, 27, 36]. Our approach is different: our decomposition is based on semantics and not metrics; our source is our understanding of a domain and the application plus that which we can gain from domain-experts, rather than from

code and execution traces. Our work is more in line with architectural recovery using MDE, where a system is described by multiple viewpoints (metamodels) and their views (models) [18]. But even here our work is different, as we start with a well-known viewpoint (a pipe-and-filter architecture) and stress the role of transformations to derive an architecture’s design. Few papers in this area emphasize both the descriptive nature of extracted models and their prescriptive ability to reconstruct a system.

Finally, our work may be an example of the *correct by construction* paradigm: if the initial architecture is correct, and its transformations are correct, the resulting architecture is correct. We have proven each of the transformations using Gamma’s architecture to be correct [3]. There is a proof of correctness for the entire ACFT architecture [12], but not proofs for individual transformations.

7. CONCLUSIONS

We showed how SWD can be used to extract an MDE architecture from parallel streaming architectures. Attributes of our approach that are desirable for incremental development are that our architectures are (a) always executable and (b) derived top-down. We use traditional refinements (the standard technique for revealing hierarchical detail), but we also used extensions (to elaborate semantics and expose extra ports, boxes and connections) and optimizations (to break encapsulation boundaries to achieve efficiency or availability). The integration of refinements, extensions, and optimizations are *all* needed to explain the complexities of today’s modern streaming architectures.

We re-engineered hand-built systems—that were created by domain experts with no formal software architecture models—to show that it is feasible to recreate these systems using model-based techniques (our ultimate goal). We were able to encode deep domain knowledge in terms of simple transformations and demonstrate the generality of our approach with two case studies from disparate domains: join parallelizations in database machines and asynchronous crash-fault-tolerant servers. Finally, we validated our approach by manually recreating these case studies in the incremental way that we presented them, showing that our models provide both an easy-to-understand *explanation* and *prescription* of how to reconstruct these systems.

Although our MDE architectures look simple and straightforward, it took effort on our part to understand the domains of these case studies, to identify and polish their core abstractions and transformations to produce and validate our derivations. But the end product is worth the effort: the simplicity of our transformations makes explaining complex designs straightforward. Our experience indicates that the progressive revealing of design details can be appreciated by engineers who lack domain knowledge. Incremental development is not just “cute” (i.e., nice to have); we believe it is ultimately indispensable for future software development that eventually integrates design, construction, verification, and testing.

The next phase of our project is to implement the Gamma and ACFT architectures on an MDE platform (e.g. [28, 29]). Because of the simplicity of our designs and the regularity (predictability) of the repetitive (e.g. wrappers) code that is needed, we hope that an MDE implementation will streamline their development.

Acknowledgments: We gratefully acknowledge helpful feedback from U. Eisenecker (Leipzig), G. Heineman (WPI), G. Karsai (Vanderbilt), E. Hehner (Toronto), H. Vin (Tata Consulting), C. Lengauer (Passau), M. Azanza (Basque Country), A. Clement (Texas), and S. Trujillo (IKERLAN) on earlier versions. Batory and Riché are supported by the NSF’s Science of Design Project CCF 0724979 and NSF’s Computer Systems Research Grant CNS 0509338.

8. REFERENCES

- [1] M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Arch. into a Design. In *UML*, 1999.
- [2] F. Baru. DB2 Parallel Edition. *IBM Sys. Journal*, 34(2), 1995.
- [3] D. Batory. The gamma db machine architecture. unpublished manuscript, 2010.
- [4] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30, June 2004.
- [5] D. S. Batory. Modeling the Storage Architectures of Commercial Database Systems. *ACM Trans. on Database Systems*, 10:463–528, 1985.
- [6] J. P. Bernhard and B. Rumpe. Stepwise Refinement of Data Flow Architectures. Technical Report TUM-19746, TU München, 1997.
- [7] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.
- [8] M. Broy. Compositional refinement of interactive systems. *JACM*, 44(6), 1992.
- [9] E. Bruneton, T. Coupaye, and J. Stefani. The Fractal Component Model. <http://fractal.law2.org>, 2004.
- [10] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. L. Lafuente. Graph-Based Design and Analysis of Dynamic Software Arch. In *LNCS*. Springer-Verlag, 2008.
- [11] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT, Jan. 2001.
- [12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. L. Riché. UpRight Cluster Services. In *SOSP*, Oct. 2009.
- [13] J. Cobleigh, L. Osterweil, A. Wise, and B. Lerner. Containment units: A hierarchically composable architecture for adaptive systems. In *FSE*, 2002.
- [14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, Dec. 2004.
- [15] D. J. Dewitt, S. Ghandeharizadeh, D. Schneider, A. B. H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE ToKADe*, 2(1):44–62, 1990.
- [16] S. Ducasse, D. Pollet, and L. Poyet. A Process-Oriented Software Arch. Reconstruction Taxonomy. In *CSMR*, 2007.
- [17] A. Egyed, N. Mehta, and N. Medvidovic. SW Connectors and Refinement in Family Arch. In *IWSAPF*, 2000.
- [18] J. Favre. Cacophony: Metamodel-driven architecture recovery. In *WCRE*. IEEE Computer Society, 2004.
- [19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.
- [20] Cft case study. <http://code.google.com/p/stepwise-ft/>.
- [21] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2009.
- [22] D. Garlan. Style-Based Refinement for Software Architecture. In *ISAW*, 1996.
- [23] M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *ICSE*, 1991.
- [24] Hadoop. <http://hadoop.apache.org/core/>.
- [25] R. Heckel and S. Thöne. Behavior-Preserving Refinement Relations Between Dynamic Soft. Arch. In *WADT*, 2004.
- [26] E. Hehner. Predicative Programming Part 1. *CACM*, 1984.
- [27] R. Holt, A. Winter, and A. Schurr. GXL: Toward a Standard Exchange Format. In *Reverse Engineering*, Nov. 2000.
- [28] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MTP Workshop at MODELS*, 2005.
- [29] G. Karsai, A. Ledeczi, S. Neema, and J. Sztipanovits. The MIC Tool suite: Metaprogrammable Tools for Embedded Control System Design. In *IEEE CCACSD*, Oct. 2006.
- [30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.
- [31] R. Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In *LNCS 5413*, 2009.
- [32] R. Kotla. *xbft: Byzantine Fault Tolerance with High Performance, Low Cost, and Aggressive Fault Isolation*. PhD thesis, The University of Texas at Austin, 2008.
- [33] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, Oct. 2007.
- [34] L. Lamport. The Part-Time Parliament. *ACM ToCS*, 16(2):133–169, 1998.
- [35] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos Register. In *IEEE SRDS*, 2007.
- [36] O. Maqbool and H. Babri. Hierarchical Clustering for Software Arch. Recovery. *IEEE TSE*, pages 759–780, 2007.
- [37] T. Mens, K. Czarnecki, and P. V. Gorp. A Taxonomy of Model Transformations. In *GraMoT*, 2005.
- [38] D. L. Métayer. Describing Software Arch. Styles Using Graph Grammars. *IEEE TSE*, 24(7):521–533, 1998.
- [39] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architectural Refinement. *IEEE TSE*, 21:356–372, 1995.
- [40] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, 2nd edition, 2003.
- [41] National Instruments LabView 8. <http://www.ni.com/labview/>.
- [42] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, 1979.
- [43] T. L. Riché, H. M. Vin, and D. Batory. Transformation-Based Parallelization of Request-Processing Architectures. In *MODELS*, October 2010.
- [44] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Sept. 1990.
- [45] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.
- [46] W. Thies. *Lang. and Compiler Support for Stream Programs*. PhD thesis, MIT, Feb. 2009.
- [47] S. Trujillo, D. Batory, and O. Diaz. Feature Oriented Model Driven Develop.: A Case Study for Portlets. In *ICSE*, 2007.
- [48] T. Tseng, J. Aldrich, D. Garlan, and B. Schmerl. Semantic Issues in Arch. Refinement. Technical report, CMU, 2004.
- [49] M. Volter and I. Groher. Product Line Implementation using AO and MD Software Development. In *SPLC*, 2007.
- [50] S. Wang, G. S. Avrunin, and L. A. Clarke. Arch. Building Blocks for Plug-and-Play System Design. In *CBSE*, 2006.
- [51] N. Wirth. Program Development by Stepwise Refinement. *CACM*, 14(4):221–227, 1971.
- [52] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Servers. In *SOSP*, 2003.
- [53] Zookeeper. <http://hadoop.apache.org/zookeeper>.
- [54] S. Zschaler and et. al. VML*: A Family of Languages for Variability Management in Software Product Lines. In *SLE*, 2009.