Architecture Design by Transformation

Taylor L. Riché[1], Don Batory[1], Rui Gonçalves[2], and Bryan Marker[1]

[1] University of Texas at Austin, Austin, TX 78712 USA
{riche, batory, bamarker}@cs.utexas.edu
[2] Universidade do Minho, Braga, Portugal
rgoncalves@di.uminho.pt

UTCS Technical Report TR-10-39

Abstract. We show how transformations organize and explain the designs of legacy pipe-and-filter-architectures. We start with an elementary architecture and progressively transform it to a detailed executable architecture. In this paper, we (a) present an MDE-based foundation of how domain-specific design knowledge can be organized by transformations, (b) illustrate how complex pipe-and-filter architectures created by domain experts can be recovered and explained by applying transformations, and (c) validate our approach with examples from parallel database-query processing and high-performance matrix computations.

1 Introduction

In the past two years, we reengineered legacy applications in multiple domains and confronted long-standing problems: how can complex pipe-and-filter architectures be explained in a simple way, so that non-experts (such as students, engineers, and faculty) can understand and appreciate expert-created designs? And what principles can organize domain-specific design knowledge yet be general enough to be useful in different domains?

Twenty years ago, *knowledge-based software engineering (KBSE)* addressed these problems with mixed success [14,35,58]. Their ideas were correct—namely that domain-knowledge can be expressed as transformations and that automated and semi-automated design tools could aid engineers to design reliable systems. The success of KBSE was limited by its era: *model-driven engineering (MDE)* and its emphasis on transformation-based designs were unknown then, *component-based software engineering (CBSE)* and *software architectures (SA)* were nascent, and experience in building large systems by transformations simply did not exist.

We depart from the classical KBSE approach that focused on microscopic transformations ($\mathtt{a} + \mathtt{a} \rightarrow \mathtt{2a}$) and required thousands or tens of thousands of rewrites to synthesize a single program [14,58]. In contrast, we use transformations that express relationships between abstractions (i.e. interfaces) and their implementing components. This enables us to derive large programs with complex architectures by applying tens (not thousands) of transformations to an initial and elementary architecture to synthesize a target architecture. The target architecture was never conceived in these terms, and this makes architectural recovery both interesting and challenging. But doing so provides a clean prescription for explaining and reproducing architectures whose inner-workings would

otherwise remain inaccessible to a vast majority of software engineers and computer scientists.

The contribution of this paper is a synthesis of known ideas to tackle the problems of the introductory paragraph. The ideas of refinement and optimization are not new, nor is the use of transformations in deriving programs new. What is new is:

- the scale at which our transformations operate compared to prior work,
- how we define and organize domain-specific design knowledge as grammars, and
- the simplicity of our approach that makes architecture design-by-transformation practical.

We demonstrate this practicality by illustrating transformations used in classical parallel-query-processing architectures of database machines and high-performance dense-linear-algebra algorithms used on distributed-memory machines. We explain how we reproduced these programs in exactly the way we describe them. We present a tool that enables designers to 1) specify domain-specific design knowledge as grammars graphically, 2) explore the design space by building sentences (i.e. architectures) of these grammars interactively, and 3) demonstrate proofs-of-concept of our approach.

Finally, although our paper is definitely not about *formal methods (FMs)*, we suggest connections that FMs can contribute to our work. We start by reviewing the principles behind our work.

2 The Principles of Refinement and Optimization

2.1 Refinement

A pipe-and-filter *architecture* is a directed graph of boxes and connectors that defines the implementation of a system. A *box* is a component with input and output ports. A *connector* is a communication path for messages pointing in the direction of dataflow from an output port to an input port(s).

A filter architecture is an elementary example (Figure 1a). It consists of a single box `FILTER` that takes a stream of photographs `P` as input, examines each photograph `p` in `P`, and outputs `p` only if some criteria is satisfied. A shorter stream `Q` is produced. `FILTER` may have other parameters, such as a photograph type and filtering criteria. We elide these details without loss of generality.

An architectural *transformation* is a mapping—a graph rewrite—of an input architecture to an output architecture. Suppose there are multiple (but semantically equivalent) filtering components, $\texttt{FILTER}_1$ and $\texttt{FILTER}_2$, each with its own distinct performance characteristics. A transformation could replace the `FILTER` box of Figure 1a with the $\texttt{FILTER}_2$ box resulting in Figure 1b. Another transformation replaces the `FILTER` box of Figure 1a with its map-reduce counterpart in Figure 1c, which shows that an input stream can be split into substreams, each substream is filtered in parallel, and the output substreams are merged [23].

We organize transformations as productions of a grammar: let `A` be an abstract box (a box that defines only the input/output ports and—at least informally—box semantics), and $\texttt{G}_1, \texttt{G}_2, \ldots$ denote architectures (graphs) that implement `A`:

$$\texttt{A} \ : \ \texttt{G}_1 \mid \texttt{G}_2 \mid \ldots \ ;$$

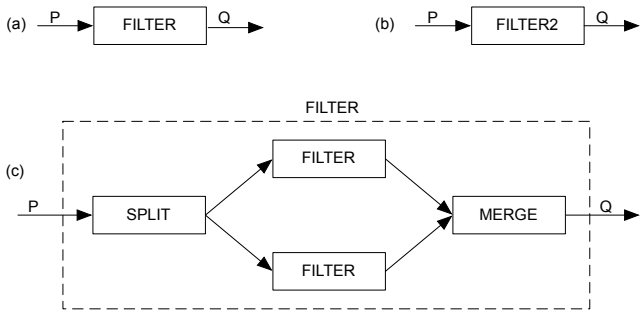


Fig. 1. Pipe and Filter Architectures

As an example, if `FILTER` of Figure 1a denotes an abstract filter box and everything else is an implementation, we have:

$$\texttt{FILTER} \ : \ \texttt{FILTER}_1 \ | \ \texttt{FILTER}_2 \ | \ \texttt{MapReduceFilter(} \ \texttt{FILTER} \ \texttt{)} \ ;$$

Where `MapReduceFilter( FILTER )` denotes the architecture of Figure 1c that is parameterized by an implementation of the `FILTER` abstraction. In general, wherever abstract box `A` appears, any of its implementing graphs $\texttt{G}_\texttt{i}$ can appear. Replacing an abstraction with an implementation is *refinement* [54].

Of course, the $\texttt{G}_\texttt{i}$ graphs can reference other abstractions and these abstractions have their own productions. The collection of such productions is a grammar, and the set of graphs that can be constructed by substituting implementations for abstractions defines the *language* or *domain* of systems that can be synthesized by refinement.[3]

2.2 Optimization

Grammars define the legal compositions of refinements, but refinements alone are insufficient for synthesizing efficient systems [55]. Consider the following grammar, where uppercase letters denote abstractions and lowercase are concrete boxes (i.e. components):

$$\texttt{A} \ : \ \texttt{a B c} \ | \ \ldots \ ;$$
$$\texttt{B} \ : \ \texttt{b} \ | \ \ldots \ ;$$

[3] The resulting grammar is context-free. We expect more generally that such grammars will be context-sensitive, where certain combinations of implementations may be incompatible or may be required [8]. Context-sensitivity arises because certain properties of implementations are *not* exposed by an abstraction. These properties distinguish different implementations (e.g. performance), and may be used to constrain legal combinations of implementations.

An example is given in [45]. Directed and undirected graph components implement the same interface. (The distinction between directed and undirected is invisible to the interface). A component that implements a strongly-connected graph algorithm requires directed graphs. A grammar that enforces this constraint is context-sensitive.

A sentence of this grammar is `abc`. Suppose box composition `bc` implements abstraction `Z` which has the following production:

```
Z : bc | q ;
```

Further, domain-experts know that composition `bc` is inefficient and `bc` can be replaced by box `q`, which is faster. This is accomplished by *abstracting* (i.e. reparsing) sentence `abc` to `aZ` and then refining to a faster program by replacing `Z` with `q` to yield `aq`. Abstraction followed by refinement is the essence of architectural *optimization*. Note why such optimizations arise: refinement introduces inefficiencies because of component boundaries. The pairing of `bc` arises because `c` comes from the refinement of `A` and `b` comes from the refinement of `B`. Dissolving component boundaries exposes inefficiencies which can be removed by optimizations.

Figure 2a is an elementary example. Abstract box boundaries are indicated by dashed lines. The left box processes stream `H` by box `F`. The right box immediately processes its input by box $\mathtt{F}^{-1}$, the inverse of `F`. Figure 2b dissolves these boundaries, exposes the inefficiency, and reveals the identity abstraction of which (`F` followed by $\mathtt{F}^{-1}$) is an implementation. Figure 2c shows the optimized architecture.

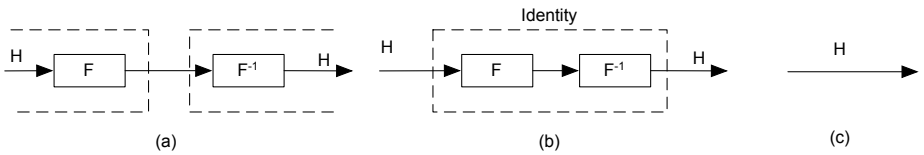


Fig. 2. Architecture Optimizations

2.3 Perspectives

Organization. Grammars are a compact way to organize and define domain-specific design knowledge. Each production defines a set of transformations where the abstraction of a production can be replaced with any of its implementations. From a formal methods perspective, this set of (abstraction, implementation) pairs defines theorems to be proven, i.e. each refinement correctly implements its abstraction.

Design. The design of a domain-specific system is an exercise in creating and applying transformations to derive that system. Sometimes existing transformations are sufficient. Other times, new rewrites (such as introducing new implementations of previously-known abstractions or new optimizations) are postulated during a design, which are to be tested or verified later.

Existing Tools. Well-known tools for data flow and pipe-and-filter architectures, like LabVIEW [51], Weaves [34], and Simulink [57] support refinement, but interestingly *not* optimization. Domain-specific optimizations are essential to explain the case studies that we consider later. Further, they represent a fundamental omission in existing tools.

Executability. Our models are executable. This can be seen in Figure 1. A $\texttt{FILTER}_2$ box is created (typically by hand) to make the architecture in Figure 1a executable. When we refine to Figure 1c, we create new SPLIT and MERGE boxes and link them with FILTER boxes. The result is another executable architecture. After each refinement or optimization, we reuse existing concrete boxes or implement new boxes to make our architectures executable. In this way, we can execute and test (as we discuss below) our architectures at successively detailed levels of abstraction. This is useful in architectural development and implementation.

Testing. An appealing property of our approach is that we can define a battery of tests for each abstraction. All implementations of that abstraction must pass these tests. So the tests for the FILTER abstraction correspond to "unit" tests for the $\texttt{FILTER}_1$ and $\texttt{FILTER}_2$ boxes, and as "integration" tests for the map-reduce architecture of Figure 1c. That is, logically the input-output response of a single FILTER box and its parallel filtering counterpart should be indistinguishable. The same holds for optimizations.

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

Challenge and Experience. Our case studies were not originally expressed in terms of refinements and optimizations. It took time for us to polish the derivations that we present. A contribution of our paper are the transformational designs of these legacy applications. More on this in Section 5.2.

All of our case studies necessarily involve domain-specific knowledge, i.e. facts that are not widely-known. Our examples illustrate exactly the principles that we just described, and that (a) domain-specific knowledge about system design can be organized by these principles and (b) with a small amount of domain-specific expertise, these designs can be appreciated and replicated by others.

3 Hash Joins in Database Machines

Gamma was (and perhaps still is) the most sophisticated relational database machine built in academics [24]. 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 [24]; our presentation is a derivation from first principles. Each step (refinement or optimization) that we make has been proven correct [10], and we have implemented our design exactly as we presented it and verified our implementation with tests.

A *hash join* is an implementation of a relational equi-join; it takes two streams $(\texttt{A}, \texttt{B})$ of tuples as input and produces their equi-join $(\texttt{A} \bowtie \texttt{B})$ 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 3a shows the executable HJOIN architecture that we start with.

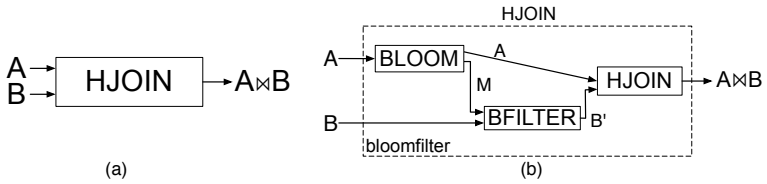


Fig. 3. Hash Join Architecture

3.1 Bloom Filtering Refinement

Joins are among the most expensive database operations. Gamma makes an ingenious use of Bloom filters [17]—a simple approximation of semijoins—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 3b.

Here is 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`. Stream `B` is read one tuple at a time. Each `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 $\mathtt{B}'$ is the result.

3.2 Parallelizing Refinements

Next, we refine the `BLOOM`, `BFILTER`, and `HJOIN` boxes by replacing each with their map-reduce versions (Figure 4). A `BLOOM` box is parallelized by hash-splitting its input stream `A` into substreams $\mathtt{A}_1 \ldots \mathtt{A}_n$, creating a `BLOOM` filter $\mathtt{M}_1 \ldots \mathtt{M}_n$ for each substream, coalescing $\mathtt{A}_1 \ldots \mathtt{A}_n$ back into `A`, and merging bit maps $\mathtt{M}_1 \ldots \mathtt{M}_n$ into a single map `M`.

A `BFILTER` box is parallelized by hash-splitting its input stream `B` into substreams $\mathtt{B}_1 \ldots \mathtt{B}_n$, where the same hash function that splits stream `A` is used to split stream `B`. Map `M` is decomposed into submaps $\mathtt{M}_1 \ldots \mathtt{M}_n$ and substream $\mathtt{B}_i$ is filtered by $\mathtt{M}_i$. The reduced substreams $\mathtt{B}'_1 \ldots \mathtt{B}'_n$ are coalesced into stream $\mathtt{B}'$.

The parallelization of the `HJOIN` box is textbook [7]: both input streams `A`, `B` are hash-split on their join keys using the same hash function as before. Each stream $\mathtt{A}_i$ is joined with stream $\mathtt{B}_j$ ($\mathtt{i}, \mathtt{j} \in 1 \ldots \mathtt{n}$), yielding $\mathtt{n}^2$ `HJOIN` boxes. Since an equi-join is computed, we know $\mathtt{A}_i \bowtie \mathtt{B}_j = \emptyset$ for all $\mathtt{i} \neq \mathtt{j}$ (as equal keys must hash to the same value). Thus, a single abstract `HJOIN` box is replaced by `n` `HJOIN` boxes instead of $\mathtt{n}^2$ boxes as all other `HJOIN` boxes have provably null outputs. By merging the joins of $\mathtt{A}_i \bowtie \mathtt{B}_i$ ($\mathtt{i} \in 1 \ldots \mathtt{n}$), $\mathtt{A} \bowtie \mathtt{B}$ is produced as output.

Figure 4 shows the result of applying all three parallelization refinements to Figure 3b.

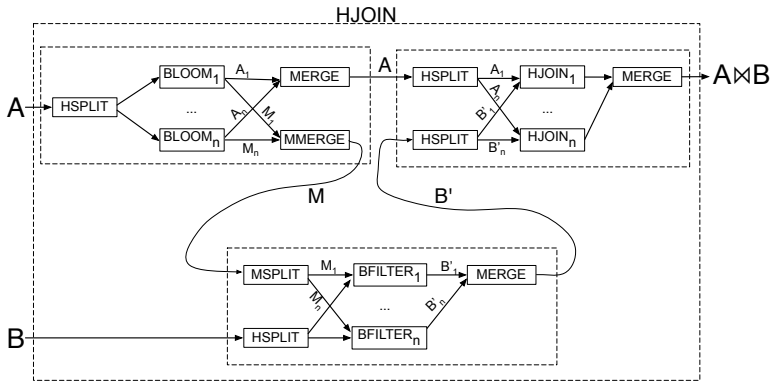


Fig. 4. Parallel Refinements

3.3 Optimizations

A primary goal of Gamma was to determine performance increases in joins that could be gained by parallelization. The architecture of Figure 4 has three *serialization bottlenecks* which degrade performance. Consider the MERGE of substreams $A_1 \ldots A_n$ into A, followed by a HSPLIT to reconstruct $A_1 \ldots A_n$. *There is no need to materialize* A: the (MERGE, HSPLIT) pair is the identity map: $A_i \rightarrow A_i$ ($i \in 1 \ldots n$). The same applies for the (MERGE, HSPLIT) pair for collapsing and reconstructing substreams $B'_1 \ldots B'_n$. The removal of (MERGE, HSPLIT) pairs eliminates two serialization bottlenecks. See Figure 5a.

The third bottleneck combines maps $M_1 \ldots M_n$ into M and then decomposes M back into $M_1 \ldots M_n$. The (MMERGE, MSPLIT) pair is also the identity map: $M_i \rightarrow M_i$ ($i \in 1 \ldots n$). This optimization removes the (MMERGE, MSPLIT) boxes and reroutes the streams appropriately.[4] See Figure 5b.

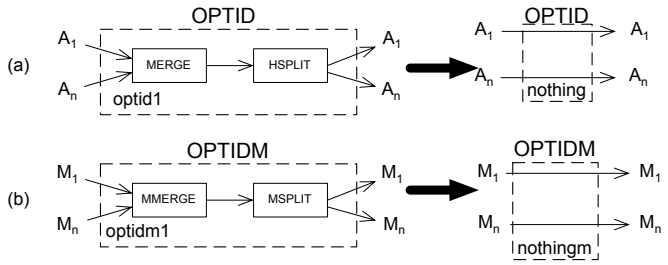


Fig. 5. Hash Join Optimizations

[4] 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 $M_1 \ldots M_n$ into M by boolean disjunction. For each i, MSPLIT extracts row i from M and zeros out the rest of M_i.

Figure 6 shows the result of all three optimizations, which is the design Gamma uses to parallelize a single join. Also note Figure 6 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 [51], Weaves [34], or Lagniappe [56] can stitch the boxes of Figure 6 together to produce an executable.

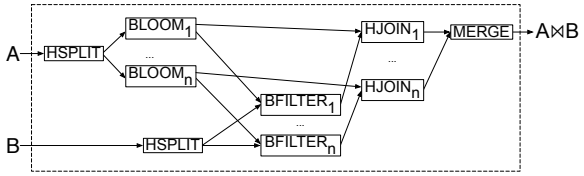


Fig. 6. Gamma Parallelized Hash Join

This is not the last word on Gamma's parallel join architecture [9]. Additional transformations are used to optimize cascading hash joins, i.e. the output of one join is the input to another join. See the Appendix for more details.

3.4 Perspective

Grammar. The grammar that is used in Gamma's design is listed below, where `CAPITAL` names denote abstractions and `lowercase` names are primitive boxes. As before, `bloomfilter`(`BLOOM`, `BFILTER`, `HJOIN`) refers to the parameterized architecture of Figure 3b, where parameters specify implementations of the `BLOOM`, `BFILTER`, and `HJOIN` abstractions. The rest of the grammar is interpreted similarly.

```
  HJOIN : hjoin
        | bloomfilter( BLOOM, BFILTER, HJOIN )
        | parallelhjoin( HSPLIT, HJOIN, MERGE ) ;
  BLOOM : bloom
        | parallelbloom( HSPLIT, BLOOM, MERGE, MMERGE ) ;
 HSPLIT : hsplit ;
BFILTER : bfilter
        | parallelfilter( HSPLIT, MSPLIT, BFILTER, MERGE ) ;
  MERGE : merge ;
 MMERGE : mmerge ;
 MSPLIT : msplit ;
   OPTID : optid1( HSPLIT, MERGE )
        | nothing ;
 OPTIDM : optidm1( MSPLIT, MMERGE )
        | nothingm ;
```

Evaluation and Correctness. As mentioned earlier, every (abstraction, implementation) pair represents a theorem to be proven. Doing so lays the foundation for *correct-by-construction* designs, e.g. if the initial design is correct and all transformations applied to it are correct, then the final design is correct [39]. We have proofs of correctness (or can cite proofs of correctness) for every production [10]. Further, 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 Gamma as an assignment to upper-division software engineering classes totalling over 60 students in Spring and Fall 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 could be completed, 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 graph of Figure 6. Implementing and testing Gamma hash joins is an interesting exercise in correct-by-construction development: we were assured at every step that our Gamma models were provably correct and with tests that our implementations were demonstrably correct.

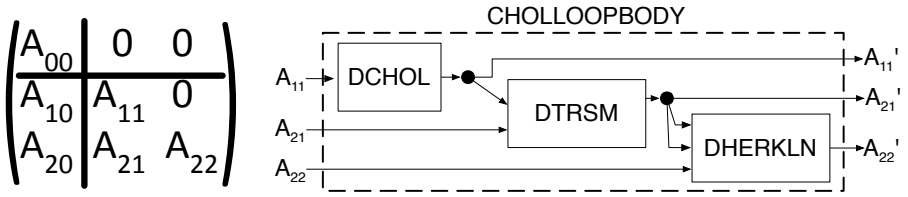
4 Distributed Memory Cholesky Factorization

Elemental [52] is a distributed-memory, dense linear algebra library that uses a highly-structured, objected-oriented programming style to manage data distributions and computations. It is the next-generation of PLAPACK [2,64], one of the two most popular packages used by the high-performance computing community—the other is ScaLAPACK [5,16]. Among the advantages of Elemental is (a) better performance and scalability than PLAPACK and ScaLAPACK [52], (b) its code is easier to read and reason about because of the data distributions it uses and the way that computations are parallelized [52], and (c) its programming style allows one to mechanically optimize algorithms by distributing data and computation, but this task is currently done manually by an expert. There is nothing novel performed by the expert because the techniques have been known for many years and are formulaic, but they have never been cast in a systematic approach that uses abstractions and refinements. By doing so, we can encode the domain-specific details of transformations and optimizations into a format that allows a system to automate a process previously done only by hand.

Cholesky factorization is a common technique for solving a system of linear equations and is a representative program that illustrates transformations that exist in the dense-linear-algebra domain. The result of a Cholesky factorization of a symmetric, positive definite matrix $\mathtt{A}$ is a lower-triangular matrix $\mathtt{L}$ such that $\mathtt{A} = \mathtt{L} \cdot \mathtt{L}^*$, where $\mathtt{L}^*$ denotes the conjugate transpose of $\mathtt{L}$.

The Cholesky algorithm we use marches through the matrix being factored from the top-left to the bottom-right, and can be characterized entirely by the updates found in the body of the loop that strides through the matrix [65]. The updates operate on the blocks of the matrix labelled $\mathtt{A}_{11}$, $\mathtt{A}_{21}$, and $\mathtt{A}_{22}$ in Figure 7a, which holds a block of the diagonal ($\mathtt{A}_{11}$), the panel below the diagonal block ($\mathtt{A}_{21}$), and the block to the bottom-right of the diagonal ($\mathtt{A}_{22}$). The horizontal and vertical lines of Figure 7a move

down and to right, respectively, with each iteration, thus marching through the matrix and indexing different sub-blocks at each iteration. The initial loop body, CHOLLOOPBODY, is shown in Figure 7b. Blocks $\mathtt{A}_{11}$, $\mathtt{A}_{21}$, and $\mathtt{A}_{22}$ of the original matrix are inputs, and the updated versions $\mathtt{A}'_{11}$, $\mathtt{A}'_{21}$, and $\mathtt{A}'_{22}$ are output. The factorization only touches the lower-triangular portion of the matrix and at each iteration—blocks other than $\mathtt{A}_{11}$, $\mathtt{A}_{21}$, and $\mathtt{A}_{22}$ are unchanged. Sequentially replicating and composing panels of Figure 7b unrolls the loop, where successive iterations cause $\mathtt{A}_{00}$ to grow in size and $\mathtt{A}_{21}$ and $\mathtt{A}_{22}$ to shrink. $\mathtt{A}_{11}$ remains the same size, chosen for the best performance, with the possible exception of the final iteration.



(a) Blocking of Matrix A (b) Initial Implementation of Cholesky Factorization

Fig. 7. Cholesky Factorization

Distribution	Location of data in matrix
$[*,*]$	All processes store all elements
$[\mathtt{M_C},\mathtt{M_R}]$	Process $(\mathtt{i}\%\mathtt{r},\mathtt{j}\%\mathtt{c})$ stores element $(\mathtt{i},\mathtt{j})$
$[\mathtt{M_C},*]$	Row i of data stored redundantly on process row i % r
$[\mathtt{M_R},*]$	Row i of data stored redundantly on process column i % c
$[\mathtt{V_C},*]$	Rows of matrix wrapped around process grid in column-major order

Table 1. Elemental Data Distributions in a $\mathtt{p} = \mathtt{r} \times \mathtt{c}$ Process Grid

Operation	Explanation
DCHOL	Distributed Cholesky Factorization
LCHOL	Local Cholesky Factorization, all process compute the same results
DTRSM	Distributed triangular solve with multiple right hand sides
LTRSM	Local triangular solve with multiple right hand sides, all processors compute an equal portion of the result
DHERKLN	Distributed Hermitian (rank-k) update
LHERKLN	Local Hermitian (rank-k) update, all processors compute an equal portion of the result
$[\mathtt{x},\mathtt{y}] \rightarrow [\mathtt{z},\mathtt{w}]$	Redistribute data from Elemental distribution $[\mathtt{x},\mathtt{y}]$ to $[\mathtt{z},\mathtt{w}]$

Table 2. Operations in Cholesky Factorization

The termination condition is for $\mathtt{A}_{11}$, $\mathtt{A}_{21}$, and $\mathtt{A}_{22}$ to be of zero size. In short, the input matrix A of the algorithm enters at the far left of the replicated panels, and the output matrix L exits at the far right [36,37].

A distributed-memory machine has p processing cores. Elemental creates p processes, one per core, and views these processes as a 2-dimensional $\mathtt{r} \times \mathtt{c}$ grid, where the values of r and c are determined experimentally. The default distribution of matrix data, labeled $[\mathtt{M}_\mathtt{C}, \mathtt{M}_\mathtt{R}]$, uses a 2-dimensional, cyclic mapping. Listed in Table 1 are other data distributions that are obtained by communication between the processors that involve rows, columns, or all of the grid [52]. Table 2 lists the boxes (redistributions and computations) that are relevant to our Cholesky example.

Figure 7b shows a refinement of CHOLLOOPBODY as a cascading sequence of update boxes. The input and output data distributions for each box are the standard distribution in Elemental, $[\mathtt{M}_\mathtt{C}, \mathtt{M}_\mathtt{R}]$. Dot ($\bullet$) is simply a notation to indicate that a stream is replicated; there is no code for $\bullet$. The important point is that data can be redistributed or communicated between all processors using redistributions. Internally, the boxes in Figure 7b redistribute data; refining their implementations provides subsequent opportunities for optimizations to reduce costly redistributions. The purpose is to accrue the benefits of improved load balance with the cost of the communication required to obtain that improvement.

4.1 Refinements

Each of the boxes in Figure 7b are abstractions that can be implemented in multiple ways. Each implementation has different performance characteristics due to the ways computation is parallelized and data is distributed. Figure 8 shows one way to refine each of these abstractions.

4.2 Redundant Redistribution Optimization

After applying the refinements of Figure 8, we flatten the design to eliminate abstraction boundaries. Now consider Figure 9, which shows the composition of the right-most box of DCHOL composed with the upper-left-most box of DTRSM. For lack of a better name, we call this abstraction OPT1. The input to OPT1 has the distribution $[*, *]$, which is then changed to $[\mathtt{M}_\mathtt{C}, \mathtt{M}_\mathtt{R}]$ to produce output K, and then redistributed back to $[*, *]$ to produce output J. Figure 9 shows a more compact and efficient way to achieve this same goal.

4.3 Redistribution Refinement

Now look at the left-hand side of Figure 10, which depicts a sub-architecture of DHERKLN. Each of the depicted redistributions can be refined into a series of more primitive redistributions, as shown in the right-hand side of Figure 10. Note that there is a replicated distribution in this rewrite, namely $[\mathtt{M}_\mathtt{C}, \mathtt{M}_\mathtt{R}] \rightarrow [\mathtt{V}_\mathtt{C}, *]$, that is unnecessary. Figure 11 shows the rewrite that improves performance, whose abstraction we call OPT2.

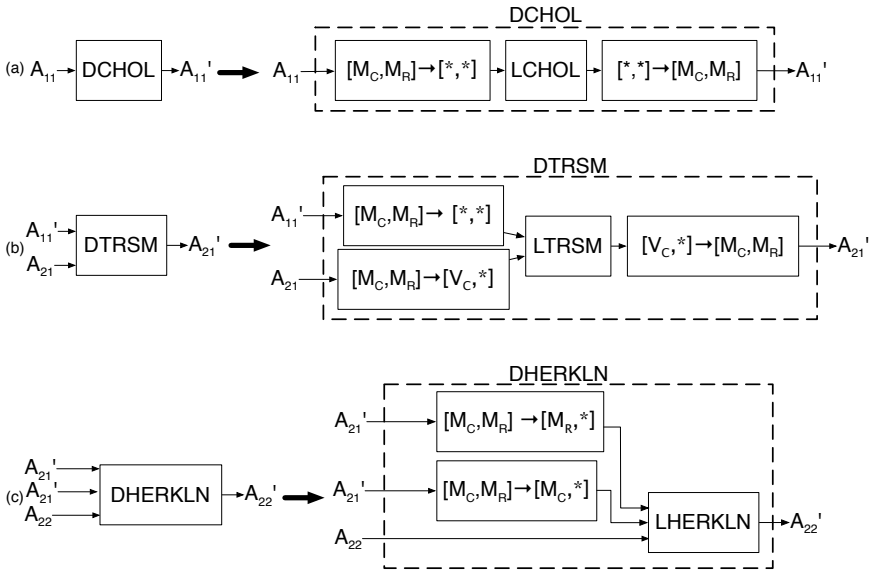


Fig. 8. Refinements of the DCHOL, DTRSM, and DHERKLN Boxes

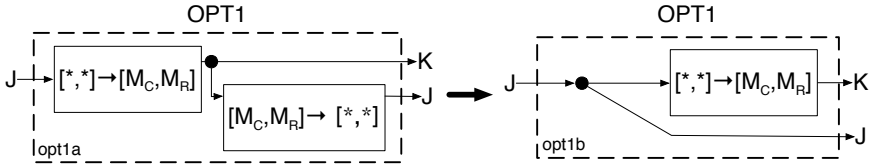


Fig. 9. The OPT1 Redistribution Optimization

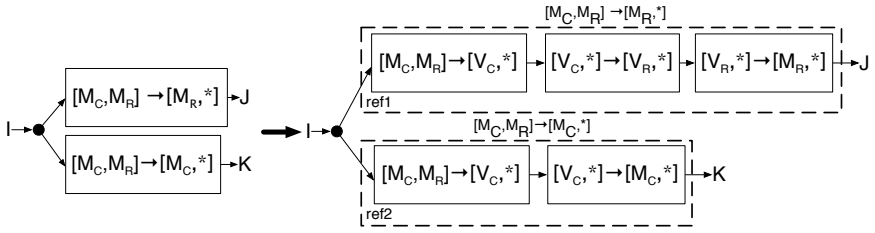


Fig. 10. Refinements of Two Redistributions

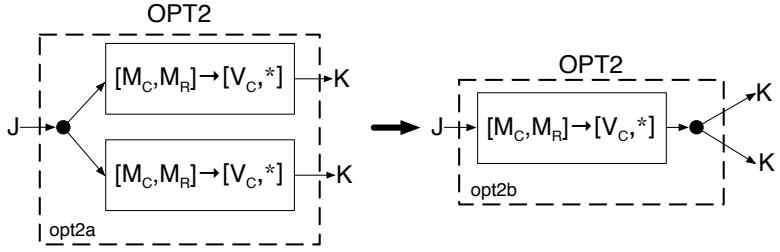


Fig. 11. The OPT2 Redistribution Optimization

4.4 Another Redistribution Optimization

There is one last redistribution optimization. Look at Figure 12, which contains a cascading series of three redistribution boxes, whose abstraction we call `OPT3`. The left-most box of `OPT3` comes from the right-most box of `DTSRM` (Figure 8b). The remaining boxes of `OPT3` are two of the cascading boxes of Figure 10. As in the previous optimization, there is a more compact and efficient way to achieve this same goal. Figure 12 shows this rewrite. Figure 13 depicts the final architecture of our Cholesky program.

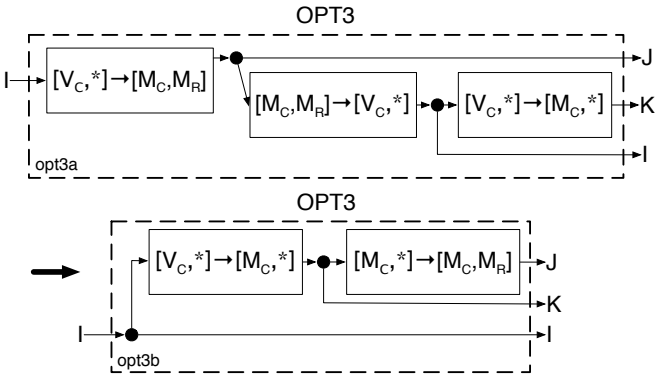


Fig. 12. The `OPT3` Redistribution Optimization

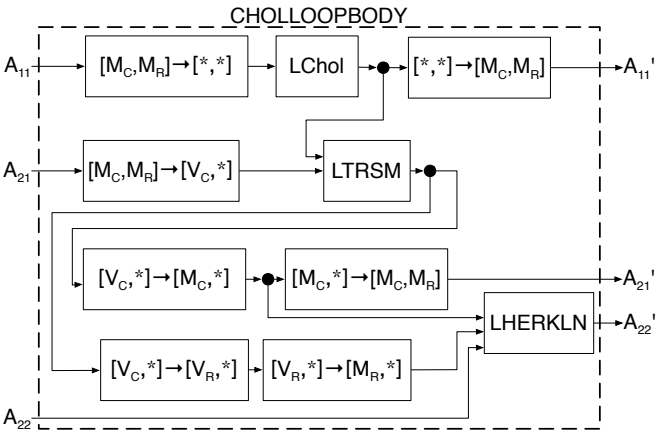


Fig. 13. Final Architecture

4.5 Perspective

Grammar. The grammar of the Cholesky Loop Body is shown below.

$$
\begin{aligned}
\texttt{CHOLLOOPBODY} &: \texttt{cholloopbody(}\ \texttt{DCHOL, DTRSM, DHERKLN}\ \texttt{)}\ ; \\
\texttt{DCHOL} &: \texttt{dchol(}\ \texttt{MCMR2}{*}{*}\texttt{, LCHOL, }\ {*}{*}\texttt{2MCMR}\ \texttt{)}\ ; \\
\texttt{DTRSM} &: \texttt{dtrsm(}\ \texttt{MCMR2}{*}{*}\texttt{, MCMR2VC}{*}\texttt{, LTRSM, VC}{*}\texttt{2MCMR}\ \texttt{)}\ ; \\
\texttt{DHERKLN} &: \texttt{dherkln(}\ \texttt{MCMR2MR}{*}\texttt{, MCMR2MC}{*}\texttt{, LHERKLN}\ \texttt{)}\ ; \\
\texttt{LCHOL} &: \texttt{lchol}\ ; \\
\texttt{LTRSM} &: \texttt{ltrsm}\ ; \\
\texttt{LHERKLN} &: \texttt{lherkln}\ ; \\
\texttt{OPT1} &: \texttt{opt1a(}\ {*}{*}\texttt{2MCMR, MCMR2}{*}{*}\ \texttt{)} \\
&\quad \mid \texttt{opt1b(}\ {*}{*}\texttt{2MCMR}\ \texttt{)}\ ; \\
\texttt{OPT2} &: \texttt{opt2a(}\ \texttt{MCMR2VC}{*}\ \texttt{)} \\
&\quad \mid \texttt{opt2b(}\ \texttt{MCMR2VC}{*}\ \texttt{)}\ ; \\
\texttt{OPT3} &: \texttt{opt3a(}\ \texttt{VC}{*}\texttt{2MCMR, MCMR2VC}{*}\texttt{, VC}{*}\texttt{2MC}{*}\ \texttt{)} \\
&\quad \mid \texttt{opt3b(}\ \texttt{VC}{*}\texttt{2MC}{*}\texttt{, MC}{*}\texttt{2MCMR}\ \texttt{)}\ ; \\
\texttt{MCMR2}{*}{*} &: [\texttt{M}_\texttt{C}, \texttt{M}_\texttt{R}] \rightarrow [*, *]\ ; \\
\texttt{MCMR2VC}{*} &: [\texttt{M}_\texttt{C}, \texttt{M}_\texttt{R}] \rightarrow [\texttt{V}_\texttt{C}, *]\ ; \\
\texttt{MCMR2MR}{*} &: [\texttt{M}_\texttt{C}, \texttt{M}_\texttt{R}] \rightarrow [\texttt{M}_\texttt{R}, *] \\
&\quad \mid \texttt{ref1(}\ \texttt{MCMR2VC}{*}\texttt{, VC}{*}\texttt{2VR}{*}\texttt{, VR}{*}\texttt{2MR}{*}\ \texttt{)}\ ; \\
\texttt{MCMR2MC}{*} &: [\texttt{M}_\texttt{C}, \texttt{M}_\texttt{R}] \rightarrow [\texttt{M}_\texttt{C}, *] \\
&\quad \mid \texttt{ref2(}\ \texttt{MCMR2VC}{*}\texttt{, VC}{*}\texttt{2MC}{*}\ \texttt{)}\ ; \\
{*}{*}\texttt{2MCMR} &: [*, *] \rightarrow [\texttt{M}_\texttt{C}, \texttt{M}_\texttt{R}]\ ; \\
\texttt{VC}{*}\texttt{2MCMR} &: [\texttt{V}_\texttt{C}, *] \rightarrow [\texttt{M}_\texttt{C}, \texttt{M}_\texttt{R}]\ ; \\
\texttt{VC}{*}\texttt{2VR}{*} &: [\texttt{V}_\texttt{C}, *] \rightarrow [\texttt{V}_\texttt{R}, *]\ ; \\
\texttt{VR}{*}\texttt{2MR}{*} &: [\texttt{V}_\texttt{R}, *] \rightarrow [\texttt{M}_\texttt{R}, *]\ ; \\
\texttt{VC}{*}\texttt{2MC}{*} &: [\texttt{V}_\texttt{C}, *] \rightarrow [\texttt{M}_\texttt{C}, *]\ ;
\end{aligned}
$$

Evaluation. Each box in Figure 13 can be mapped, one-to-one, to a single function call in Elemental code. The resulting code is the same as the hand-optimized version produced by the expert developer of Elemental. The performance of that code, seen in Figure 14, outperforms the existing leading packages in this field. These and other results were obtained on Argonne National Laboratory's IBM Blue Gene/P machine [52]. This experiment was run on 8192 cores with a theoretical peak performance of 27 TFlops.

The boxes in the above graphs or the equivalent operations in Elemental are complex and include a lot of functionality. This functionality is largely implemented using wrapper code that calls into the standardized, well-established BLAS, LAPACK, and

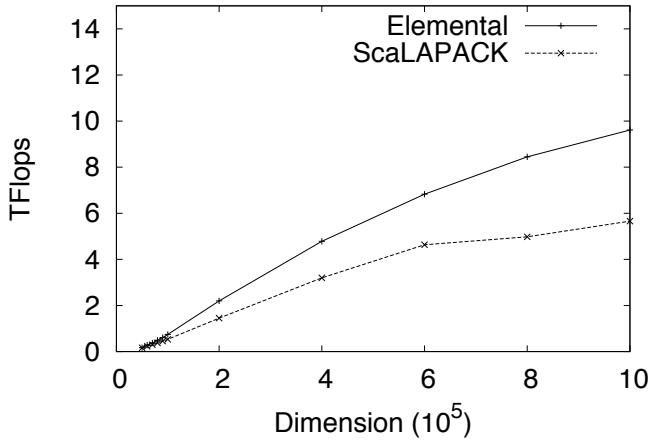


Fig. 14. Cholesky performance on 8192 cores

MPI routines [44,25,4,59]. For now we do not consider those details in our grammar, but it may be useful to do so in the future to expose more opportunities for optimizations.

Correctness. The loop body shown in Figure 7b is derived from the Cholesky factor of the input matrix; it is provably correct. Furthermore, each of the refinements we applied above can be proven to maintain the correctness of their abstractions. (Indeed, the sequential code with which we start was formally derived to be correct [65].) Thus, the results of the above methods create provably correct implementations of Cholesky factorization.

As for the correctness of the actual program, when translating an optimized algorithm to Elemental code, there is little room for error as we use a one-to-one mapping. The functionality expected from the boxes above is the exact same as the functionality expected from the mapped lines in Elemental code. Elemental code calls functions in libraries that are not proven correct, but they are so widely used that correctness is trusted throughout this domain. Furthermore, the wrapper code that calls those libraries is sufficiently formulaic and well-tested to trust as well. Therefore, the correctness of the code derived using our methods and implemented in Elemental can be trusted.

5 Tools, Methodology and Experience

5.1 Tools

We are developing a tool using the Eclipse Graphical Modeling Framework [27] and Epsilon [29]. Our goal is to create an machine-assisted environment for designers to develop pipe-and-filter architectures, to postulate new refinements and new optimizations, and to apply transformations to incrementally derive a specific architecture.

Models Our tool allows developers to express the abstractions of a domain, the algorithms that implement them, the optimizations, and derived architectures. It provides five types of objects: (1) *Abstraction*, (2) *Algorithm*, (3) *Pattern*, (4) *Input*, and (5) *Output*, and two types of associations: (1) *Connector* and (2) *Implementation*. Figure 15 shows a UML diagram of the tool's metamodel.

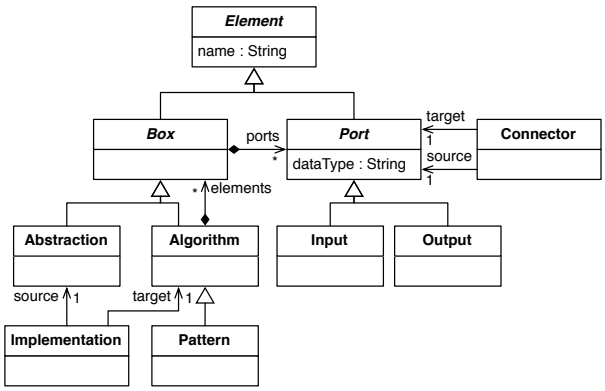


Fig. 15. Tool's Metamodel

An *Abstraction* object with its input and output ports models an abstract box. An *Algorithm* object models an algorithm. In addition to its input and output ports, we add other boxes and connectors inside, creating a dataflow graph that specifies an algorithm's details. An *Implementation* association pairs an abstraction to each of its algorithms, creating a grammar with abstractions on the left and algorithms on the right. Metamodel constraints ensure that all input and output ports of an abstraction are present in each of its implementing algorithms.

Figure 16 shows part of the grammar that we need to derive the example from Section 3. We have the abstraction `HJOIN`, and two algorithms, `bloomfilter` and `parallelhjoin`. The dashed arrows are *Implementation* associations.

A *pattern* box specifies an algorithm to be abstracted. Like an algorithm box, it is paired with an abstraction and other plug-compatible algorithms. Patterns are distinguished from algorithms in that our tool searches for patterns when optimizing designs.

Transformations Given an initial architecture, designers can replace any abstraction box with an algorithm that implements it. If there is more than one algorithm for the abstraction, the user will be asked to choose one. If a desired algorithm is not present, users can extend the list of algorithms and postulate his/her own designs.

Refinements produce a hierarchy of dataflow graphs that require optimizations. Our tool has a *flatten* transformation that removes modular boundaries. A user can then abstract a set of boxes and then refine to a new implementation. These two steps can be executed separately, using the *abstract* transformation, followed by the *refinement* transformation, or as a single step, using the *optimize* transformation. To help the user,

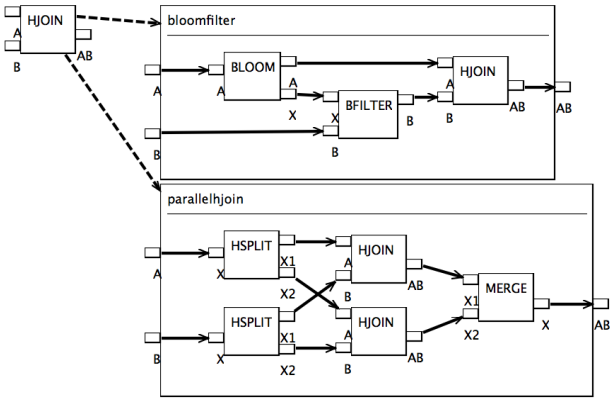


Fig. 16. `HJOIN` Abstraction and its `bloomfilter` and `parallelhjoin` Algorithms

we also provide the *find pattern* transformation, that identifies the sets of boxes present in the program that can be abstracted and then optimized.

Further Work Our tool will soon offer a transformation that converts an architecture to code. We have code for all the examples in this paper, but have not made the last step to connect our tool to our implemented components. Further, our plan is to enrich grammars with additional information, such as port data types and cost functions for each box. Doing so would allow us to type check designs, and cost functions would allow us to automate the derivation of optimized programs. Further, our tool will support extension transformations—besides refinements and optimizations—that add features (i.e. additional capabilities) to existing architectures [60].

5.2 Methodology and Experience

We tackled a general problem: there are many domains where applications use pipe-and-filter architectures, such as virtual instrumentation [51], data processing [31], graphics processing [6], and cloud computing [3,33] that could benefit from our work.

We faced five challenges: (1) Identify principles that allowed us to develop architectures in different domains incrementally. (2) Acquire an understanding of the Gamma and Cholesky (dense matrix computation) domains. And (3) decompose the Gamma and Cholesky architectures, as their original designs were certainly not expressed by transformations. Although choosing principles, understanding domains, and decomposing sounds sequential, early on they were concurrent tasks, eventually converging to the results in this paper.

Our approach works because there are natural abstractions in domains that engender multiple implementations. In databases, the `JOIN` abstraction was known in the early 1970s. Many `JOIN` algorithms were invented in the following 10-15 years, often being focal points in database conferences. The same holds for linear algebra, where multiple algorithms for implementing common abstractions (e.g. matrix multiply) abound.

Challenge (4) was understanding and viewing an architecture as the result of applying transformations or algebraic rewrites to an elementary architecture. Query optimization in databases is understood in terms of relational algebra, and applying rewrite rules (algebraic identities) to map an inefficient relational algebra expression to an efficient relational algebra expression. Linear algebra had the benefit of a mathematical foundation from the beginning. Relating design to algebra (or transformations) is an inherently structured approach to architecture creation. It was one of the tenets of KBSE and is fundamental to our work.

Another reason why our approach works is that pipe-and-filter architectures often have a nice "functional" or stateless property, raising similarities to techniques used in optimizing functional programs [41]. Gamma hash joins and Cholesky factorization algorithms are certainly stateless. We have applied these same ideas to recover transformational designs of crash-fault-tolerant servers that are *stateful* [55], so we believe the paradigm we are advancing is broader, but has predecessors in various forms in other fields. More on this in Related Work (Sec. 6).

We argue that the role of grammars—especially recursive grammars—in organizing domain-specific design transformations is fundamental. We noticed long ago [11] that grammars are naturally recursive when describing transformational designs of software. The recursive nature of the `FILTER` production of Section 2.1 (where `MapReduceFilter` recursively calls the `FILTER` abstraction) is typical: recursion is how patterns (implementations) can be composed in combinatorial numbers of ways.

Challenge (5) was to explain the genius of legacy designs in a simple manner: it took us months to polish the derivations of our case studies. Developing the insights behind these designs takes effort. As we have matured, our ability to reconstruct legacy designs as transformational designs takes less time, but is definitely not a trivial task.

6 Related Work

Twenty-five years ago, we modeled the architectures of commercial database systems as a composition of refinements [13]. We have since shown that similar ideas generalize to describe software product lines [12,62]. We expect, and have no reason to question, that refinement and optimization (and ultimately extension [60]) applies far beyond the domains we mentioned in this paper.

Our approach leverages MDE: we start with an elementary architecture and transform it into complex architectures. What is unusual is our reliance on *endogenous transformations*—transformations whose domain and co-domain are the same. MDE literature focuses on *exogenous transformations*—mappings whose domain and co-domain are different [48]. For example, each architectural style has its own metamodel. Prior work mapped architectures of one style to an architecture of another [1,28,38,63]. 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) [21,49]. The few cases where endogenous transformations are composed to produce MDE designs [62,66,68] deal with simple transformations that encode extensions as model deltas, not the 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 [15,38,49,50]. 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 work that uses architectural optimizations. And 3) the role of transformations in architectural design (as we presented it) is under-appreciated. Broy seems to lay a mathematical foundations for extension [18,19], but we are unaware of a recent system that puts his ideas into practice.

Practical tools for building pipe-and-filter architectures also have a long history. LabVIEW [51] (marketed since the mid-1980s), Weaves [34], and Simulink [57] are platforms for executing such architectures. Refinement is their sole abstraction; *user-defined optimizations are absent*.[5] Other component-based systems follow a similar approach [20,22,32,67]. More recent tools are StreamIt [61] and Click [42]. 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. To the best of our knowledge, none of these tools support architectural optimizations. It came as a big surprise to us that optimizations (as well as extensions) are absent in these tools. Not having them is a fundamental omission.

SPIRAL [53] is an approach in the *digital signal processing (DSP)* domain to implement automatically DSP operations in architecture-specific, high-performance code. SPIRAL recursively applies refinements to the sub-operations to generate many implementations and uses machine learning techniques with feedback from compiling and timing implementations to choose the best performing option. The mathematical nature of SPIRAL is very similar to the linear algebra domain explored in Section 4. In both cases, many potential implementations can be created, but the runtime of DSP operations is much less than that of dense linear algebra algorithms. Therefore, it is likely not feasible to run potential implementations of the Cholesky factorization to find the best as SPIRAL does. Instead, we can use theoretically proven lower-bound cost models for the implementations' sub-operations. Then, we can choose the theoretically best implementation of all possibilities by summing the sub-operations' costs to get each implementations' cost. We already have a prototype system that does just that [47]. It applies the transformations described in Section 4 as well as many more, generates hundreds of possible implementations, and analytically determines the implementation in Figure 13 is the optimal for the target machine across a wide range of problem sizes.

Our work is an example of software architecture recovery [26,43]. 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 [30,40,46]. Our approach is different: our decom-

[5] Simulink provides the ability to abstract a set of boxes and encapsulate them into a single box. However, there is no model specifying which subgraphs can be abstracted or the alternative implementations.

position is based on semantics and not metrics; we start by understanding the domain and 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) [30]. 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 a practical foundation for a *correct-by-construction* paradigm for architecture creation: if the initial architecture is correct, and its transformations are correct, the resulting architecture is correct. We have proofs that the transformations used in Gamma's architecture are correct [10], and comparable statements can be made for the Cholesky architecture as well. Here is where formal methods may make a fundamental contribution to our approach in the future.

7 Conclusions

Expressing design knowledge as transformations has rich historical roots. With the advent of MDE, component-based software engineering, and experience in synthesizing large systems transformationally, taking the next step to express architecture design by transformations is latent.

We have shown how the application of tens (not thousands) of transformations can explain the designs of legacy pipe-and-filter-architectures in a MDE fashion. Although our descriptions of our case studies look simple and straightforward, it took considerable effort on our part to understand their domains and to identify and polish their core abstractions and transformations to produce the results of this paper. In time, after we examine more applications in these domains, we believe that we can collect sufficient domain knowledge as transformations to create *knowledge-based design assistant* tools to help designers create (or automate) designs in well-understood domains [35].

We use grammars to encode domain-specific design knowledge; sentences of a grammar define the domain of architectures that can be synthesized by refinement. Further applying optimizations (also expressed as productions in our grammars) expands the domain of architectures that can be synthesized. Our case studies show that optimizations are essential in recovering transformation-based designs of legacy systems. Further, our approach is general: although we focused on two case studies from disparate domains, we cited many other domains to which our approach could be applied.

Perhaps the most significant contribution of our work is its simplicity: our experience has shown that the ideas are eminently teachable and understood by non-domain experts, such as undergraduates and graduates. We believe this simplicity shows that design-by-transformation of software architectures is practical.

Acknowledgments. We gratefully acknowledge helpful feedback from R. van de Geijn and J. Poulson (UTexas), 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 (UTexas), and S. Trujillo (IKERLAN) on earlier

versions. Batory and Riché are supported by the NSF's Science of Design Project CCF 0724979. Riché was additionally supported by NSF's Computer Systems Research Grant CNS 0509338. Bryan Marker was partially supported by NSF grants OCI-0850750 and CCF-0917167, and a Sandia National Laboratory fellowship. Rui Gonçalves is supported by Portuguese Science Foundation (FCT) grant SFRH/BD/47800/2008.

References

1. M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Arch. into a Design. In *UML*, 1999.
2. P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y.-J. J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.
3. Amazon web services. `http://aws.amazon.com/`.
4. E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
5. E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Lapack for distributed memory architectures: Progress report. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 625–630, Philadelphia, 1992. SIAM.
6. E. Angel. *Interactive Computer Graphics: A Top-Down Approach using OpenGL*. Addison-Wesley, 2009.
7. F. Baru. DB2 Parallel Edition. *IBM Sys. Journal*, 34(2), 1995.
8. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, Sept. 2005.
9. D. Batory. A Shorter Derivation of the Gamma Architecture. Fall CS378 Final, 2010.
10. D. Batory. The Gamma DB Machine Architecture. In Preparation, 2010.
11. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1:255–298, 1992.
12. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30, June 2004.
13. D. S. Batory. Modeling the Storage Architectures of Commercial Database Systems. *ACM Trans. on Database Systems*, 10:463–528, 1985.
14. I. Baxter. Practical issues in building knowledge-based code synthesis systems. *6th Annual Reuse Workshop (WISR'93)*, 1993.
15. J. P. Bernhard and B. Rumpe. Stepwise Refinement of Data Flow Architectures. Technical Report TUM-19746, TU München, 1997.
16. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.
17. B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.
18. M. Broy. Compositional Refinement of Interactive Systems. *JACM*, 44(6), 1992.
19. M. Broy. Multifunctional Software Systems: Structured Modeling and Specification of Functional Requirements. *Science of Computer Programming*, 2010.
20. E. Bruneton, T. Coupaye, and J. Stefani. The Fractal Component Model. `http://fractal.ow2.org`, 2004.

21. 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.
22. J. Cobleigh, L. Osterweil, A. Wise, and B. Lerner. Containment units: A hierarchically composable architecture for adaptive systems. In *FSE*, 2002.
23. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, Dec. 2004.
24. 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.
25. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, Mar. 1988.
26. S. Ducasse, D. Pollet, and L. Poyet. A Process-Oriented Software Arch. Reconstruction Taxonomy. In *CSMR*, 2007.
27. Eclipse graphical modeling framework. `http://www.eclipse.org/gmf`.
28. A. Egyed, N. Mehta, and N. Medvidovic. SW Connectors and Refinement in Family Arch. In *IWSAPF*, 2000.
29. Epsilon. `http://www.eclipse.org/gmt/epsilon/`.
30. J. Favre. Cacophony: Metamodel-driven architecture recovery. In *WCRE*. IEEE Computer Society, 2004.
31. H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2009.
32. D. Garlan. Style-Based Refinement for Software Architecture. In *ISAW*, 1996.
33. Google app engine. `http://code.google.com/appengine/`.
34. M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *ICSE*, 1991.
35. C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. *Kestrel Institute Technical Report KES.U.83.2*, 1983.
36. J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, Dec. 2001.
37. J. A. Gunnels and R. A. van de Geijn. Formal methods for high-performance linear algebra libraries. In R. F. Boisvert and P. T. P. Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001.
38. R. Heckel and S. Thöne. Behavior-Preserving Refinement Relations Between Dynamic Soft. Arch. In *WADT*, 2004.
39. E. Hehner. Predicative Programming Part 1. *CACM*, 1984.
40. R. Holt, A. Winter, and A. Schurr. GXL: Toward a Standard Exchange Format. In *Reverse Engineering*, Nov. 2000.
41. S. L. P. Jones and A. L. M. Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1-3):3–47, 1998.
42. 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.
43. R. Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In *LNCS 5413*, 2009.
44. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979.
45. R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GPCE*, 2001.
46. O. Maqbool and H. Babri. Hierarchical Clustering for Software Arch. Recovery. *IEEE TSE*, pages 759–780, 2007.
47. B. Marker, J. Poulson, and R. van de Geijn. Towards Automatic Optimization of Dense Linear Algebra Algorithms for Distributed Memory Machines. *In Preparation*, 2011.

48. T. Mens, K. Czarnecki, and P. V. Gorp. A Taxonomy of Model Transformations. In *GraMoT*, 2005.
49. D. L. Métayer. Describing Software Arch. Styles Using Graph Grammars. *IEEE TSE*, 24(7):521–533, 1998.
50. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architectural Refinement. *IEEE TSE*, 21:356–372, 1995.
51. National Instruments LabView 8. `http://www.ni.com/labview/`.
52. J. Poulson, B. Marker, J. R. Hammond, N. A. Romero, and R. van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 2010. submitted.
53. M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.
54. refinement. Program refinement. `http://en.wikipedia.org/wiki/Program_refinement`.
55. T. L. Riché and D. Batory. Extraction of MDE Architectures From Parallel Streaming Applicaitions. Technical Report TR-10-38, University of Texas at Austin, Dept. of CS, 2011.
56. T. L. Riché, H. M. Vin, and D. Batory. Transformation-Based Parallelization of Requst-Processing Architectures. In *MODELS*, October 2010.
57. Simulink. `http://www.mathworks.com/products/simulink/`.
58. D. R. Smith and E. A. Parra. Transformational approach to transportation scheduling. *8th Knowledge-Based Software Engineering Conference*, 1993.
59. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.
60. J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.
61. W. Thies. *Lang. and Compiler Support for Stream Programs*. PhD thesis, MIT, Feb. 2009.
62. S. Trujillo, D. Batory, and O. Diaz. Feature Oriented Model Driven Develop.: A Case Study for Portlets. In *ICSE*, 2007.
63. T. Tseng, J. Aldrich, D. Garlan, and B. Schmerl. Semantic Issues in Arch. Refinement. Technical report, CMU, 2004.
64. R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.
65. R. A. van de Geijn and E. S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.
66. M. Volter and I. Groher. Product Line Implementation using AO and MD Software Development. In *SPLC*, 2007.
67. S. Wang, G. S. Avrunin, and L. A. Clarke. Arch. Building Blocks for Plug-and-Play System Design. In *CBSE*, 2006.
68. S. Zschaler and et. al. VML*: A Family of Languages for Variability Management in Software Product Lines. In *SLE*, 2009.

A Cascading Joins in Gamma

Optimizations are needed to speed-up the processing of cascading joins, where the output of one join becomes the input of another (see Figure 17a). Figure 17b reveals the partial internals of HJOIN boxes where the output of the first join is formed by merging substreams $C_1 \ldots C_n$ into stream C and then C is immediately hash-split into substreams $D_1 \ldots D_k$. This is serialization bottleneck. Unlike the bottlenecks in Section 3.3, 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 ($n \neq k$).

Here is where refinement is insufficient to derive a streaming architecture; encapsulation boundaries must be broken to eliminate serialization bottlenecks. A particular transformation, called a rotation, is used to remove these bottlenecks. A *rotation* swaps the order of operations. Box sequence A followed by B (denoted A;B) is rotated to $B_n \otimes A_k$, which represents the cross bar of $B_1 \ldots B_n$ with $A_1 \ldots A_k$.

MERGE;HSPLIT in Figure 17b is rotated to $HSPLIT_n \otimes MERGE_k$ in Figure 17c. Each C_i is hash-split into k substreams ($C_{i1} \ldots C_{ik}$) and sets of n substreams ($C_{1j} \ldots C_{nj}$) are merged into stream D_j ($j \in 1 \ldots k$). 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 $n \times k$ array of substreams, which we denote by $[C]_{nk}$.

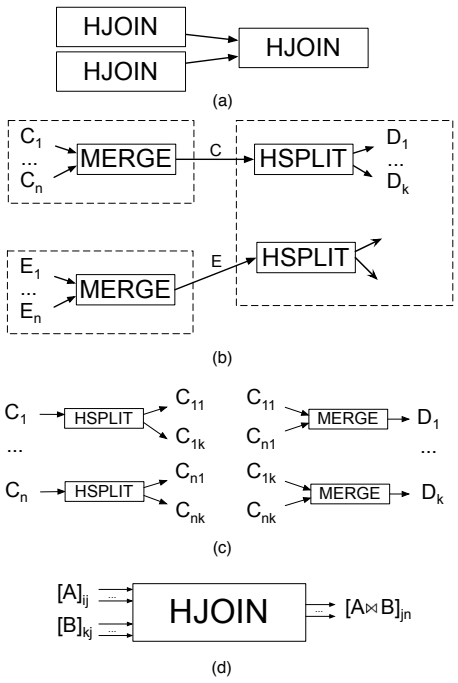


Fig. 17. Rotation of MERGE and HSPLIT

Gamma's HJOIN architecture generalizes from Figure 3a to Figure 17d: a single HJOIN box takes arrays $[A]_{ij}$ 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 array $[A \bowtie B]_{jn}$ 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.

One additional production should be added to Gamma's grammar to express these design transformations:

$$\begin{aligned} \texttt{ROTATION} : \texttt{MERGE;HSPLIT} \\ \mid \texttt{HSPLIT}_n \otimes \texttt{MERGE}_k \ ; \end{aligned}$$