DxTer: A Program Synthesizer for Dense Linear Algebra

Bryan Marker     Don Batory     C. T. Shepherd

The University of Texas at Austin
{bamarker,batory,cts}@cs.utexas.edu

Abstract
DxTer is a tool that generates a search space of implementations for operations in dense linear algebra and uses cost functions to select automatically the most efficient implementation from the space. We describe how DxTer generates this space and the optimizations it uses to reduce the size of the space to improve its runtime, such as employing phases of generation.

1. Introduction

Design by Transformation (DxT) [19], pronounced “dext”, codifies domain knowledge as transformations. These transformations are applied implicitly by domain experts to implement and optimize algorithms in high-performance code.

DxT represents algorithms and their implementations as pipe-and-filter (PnF) graphs [27]. Expert knowledge is encoded as graph transformations that are used to map domain operations to their classical algorithms, to choose machine-specific implementation details, and to replace inefficient compositions of components with more efficient compositions. The benefit of DxT is a simple way to formalize and mechanize domain expertise.

Our inspiration for DxT is rule-based relational query optimization, now over 20 years old and a standard fixture in modern database systems [17]. Given a design (relational algebra expression), algebraic identities are applied to rewrite this design into a more detailed and efficient form. These identities (transformations) encode fundamental domain-specific knowledge about how algorithms fit together to provide database functionality. By searching the space of logically equivalent designs either heuristically or completely, efficient implementations of queries are produced. DxT is an application of this paradigm to other domains.

We developed a prototype named DxTer (pronounced “dexter”) to automatically generate a space of different algorithms that implement a given operation or composition of operations. This is the same space that an expert searches, employing experience and intuition to choose which transformations are better to apply than others. Currently, DxTer uses a breadth-first search to generate all implementations and uses cost estimates to choose the “best”.

We use DxTer to generate dense linear algebra (DLA) algorithm implementations for distributed-memory systems. It generates code for a domain-specific language called Elemental [22], which targets DLA for distributed-memory machines. Our experiments have shown DxTer to automatically generate code that is the same as or better than that hand-crafted by experts. Such is the case in Figure 1 which shows the performance of two implementations of a two-sided triangular solve: DxTer-produced “Optimized” code and ScALAPACK code. (ScALAPACK is hand-crafted and the most widely-used package in DLA [7]).

Due to the combinatorial nature of DxT (i.e. implementation choices an expert makes are combinatorial), DxTer generates a large space of implementations. This paper focusses on the strategies we have developed to reduce the time DxTer takes to produce and search this space. In a 2-phased process, we have reduced the time by as much as a factor of 53 in the first phase and 2.2 in a second phase. The contribution of this paper is to explain these phases in detail.

![Two-Sided Triangular Solve Performance on BlueGene/P](image)

**Figure 1.** Two-sided triangular solve (Hegst) performance. Two-thirds of the machine’s best possible performance is at the top of the graph.
This paper is a companion to [19]. In [19], we presented the DxT approach, some DLA examples, and performance graphs for algorithms generated by DxTer. In this paper, we briefly summarize the DxT approach in Section 2. From Section 3 and beyond, we explain the novel inner workings of DxTer that enabled us to produce the results of [19]. In particular, we explain how DxTer represents algorithms as graphs and encodes graph transformations. We explain how a search space of implementations is formed by exhaustively applying transformations to an input graph (a composition of high-level DLA operations). We explain the impact of each of several optimizations used in DxTer to limit the time it takes to generate a combinatorially-sized search space and produce the performance graphs in [19] for algorithms automatically generated by DxTer.

2. Design by Transformation

One approach an expert can take to implement, parallelize, and optimize an algorithm is to apply transformations step-by-step. She considers how to code algorithms for the target hardware. She looks for inefficient compositions of components and replaces them with more efficient combinations. DxT makes it easy to encode and catalog such knowledge as transformations, so that they can be better understood and mechanically applied by a system like DxTer. In this section we explain the basics of DxT. A more complete introduction can be found in [19].

2.1 Pipe-and-Filter Graphs

DxT represents functionality as pipe-and-filter or data-flow graphs [27]. DxTer works specifically on directed acyclic multigraphs (DAGs). A DAG can represent either an abstract algorithm or low-level code for a particular architecture depending on the types of nodes it contains.

A DAG node can have multiple inputs and multiple outputs. An edge or connector is drawn in the direction of data flow and has properties of that data. For example, the way data is distributed over a process grid or the size of matrices are important properties that can be assigned to an edge. A node in a DAG represents a computation or operation to be performed. With DxT we call a node with no implementation details an abstraction or interface. It has preconditions on input ports and postconditions on output ports. A matrix multiplication abstraction, for example, has a precondition that the three matrix inputs (A, B, and C) conform in size and a postcondition that the output is C + A × B. There are no implementation details with an abstraction.

We illustrate DxT with a prototypical example from DLA: Cholesky factorization [31]. This operation takes a symmetric, positive-definite (SPD) matrix A and forms a lower-triangular matrix L such that A = LLT. Figure 2 shows the DAG of the loop body of one algorithmic variant of Cholesky factorization, known as Variant 3 [31]. The loop exposes sub-matrices at each iteration, three of which are inputs to this DAG. The three boxes in the DAG are abstractions that output the result of some computation (the specifics of which are not important). The outputs of the DAG are updated sub-matrices. This DAG does not specify how this algorithm is mapped to an architecture; it is a hardware-independent specification of a Cholesky factorization. To generate code, we replace all abstractions with concrete implementations. For that we use transformations.

![Figure 2. Cholesky variant 3 loop body as a data-flow graph. This uses abstractions (solid boxes) for component operations. Implementations must be chosen for these boxes.](image)

2.2 Graph Transformations

DxT presently uses two kinds of transformations: refinements and optimizations. Other kinds of transformations are discussed in Section 7.

**Refinements.** A refinement replaces an abstraction with an implementation—a graph of one or more nodes. The replacement subgraph can contain other abstractions, which other refinements, in turn, can instantiate with implementations. Eventually, every node represents a concrete component (e.g. a call to a specific library function for a particular machine) at which point code that implements the graph can be directly produced. Figure 3 shows some of the refinements used for the abstractions in Cholesky.

In distributed memory code, data is distributed among processors in different ways to enable parallel computation. Data must be communicated between distributions which is an expensive operation. Boxes with arrows in their label, such as \([M_C, M_R] \rightarrow [*,*]\), are data redistribution operations, which can be refined to code that translates data from one distribution into another. The other boxes, with textual names, are computations. We avoid further describing what particular boxes represent in our DLA examples because these details are not important to understand DxT or DxTer.

**Optimizations.** An optimization replaces a subgraph with two or more connected nodes with another subgraph. Optimizations encode how an expert replaces a piece of functionality with a more efficient implementation. Figure 4 (Top) shows an optimizing transformation used in Cholesky factorization that removes an unnecessary redistribution node. Figure 4 (Bottom) shows an optimization to alleviate possible numerical stability issues.²

¹ Effectively, these are architectures whose PnF graphs have no loops – a common situation.

² We use this as a convenient example to demonstrate DxT and DxTer as it is a common transformation experts apply to DLA code. We do not (yet) deal with other numerical stability transformations.
3. DAG Representation in DxTer

DxTer generates and manipulates DAGs. In Model Driven Engineering (MDE), DxTer has an implicit metamodel that requires its conforming DAGs to satisfy particular constraints. Although we use MDE terminology here to explain the DAGs we use, we have found no MDE tool that allows us to define our transformations and generate DAGs that we need.

3.1 Node Class Hierarchy

DAGs are represented by nodes that reference each other. Nodes have input references to the nodes that provide input data; these are stored in an ordered list. Nodes also have consumer references to the nodes that use produced data. Because nodes can produce more than one output, consumed references are marked with an output number.\(^3\)

A node is an instance of a subclass of Node, a class which defines functionality common to all DAG nodes. All abstractions are subclasses of Node, and implementations of an abstraction are subclasses of that abstraction’s class. For example, gemm is a concrete node that implements the matrix multiplication abstraction by calling a library function called gemm. Matrix multiplication is a subclass of Node, and gemm is a subclass of matrix multiplication. This structure makes it easy to add new abstractions and concrete implementations of abstractions to DxTer. Although all classes (with the exception of Node) of this hierarchy can be instantiated, concrete nodes are instances of the terminal classes. Instances of all other classes represent abstractions.

3.2 Node Properties

Nodes can be queried on the properties of output data or of the node itself. Properties for DLA code include matrix size, data distribution, and variable names. Output properties can be a function of input properties. For example, redistribution nodes communicate data across the network from one distribution to another, so they affect the output distribution type. Often output properties are propagated directly from inputs. Each class has methods to return “provides” functionality.

Concrete nodes have an additional cost property, which DxTer uses to determine if one DAG encodes a better implementation than another. A cost function for DLA code is estimated lower-bound runtime based on how much computation is performed and data is communicated across the network [6]. Each node type can calculate its own runtime cost based on the size of inputs and their data distribution. The cost of an entire DAG is calculated by summing the costs of each node. Thus, a DAG has only one representation in DxTer. Properties, like cost, are derived by analyzing a DAG. By summing the costs of all nodes on a DAG, implementations can be compared to choose the “best” generated code.

3.3 DAG Restrictions

Each node type has restrictions on inputs and outputs. Matrix multiplication, for example, only allows three inputs and one output. There are also restrictions on how the input sizes conform and their data distributions, such as the inner dimension of A and B must be the same size and have the same distribution for gemm. Further, the “m” dimension of A and the “n” dimension of B should match the “m” and “n” dimensions for C, too.

Restrictions are placed on the shape of a DAG, such as its acyclicity and that it is weakly connected, meaning that if the directed edges were changed to undirected edges there is a path from any node to any other node. There are also restrictions such as any node A stored as a consumer of node B should also have B in its input list. In DxTer each node and each DAG is given an opportunity to check that all restrictions are obeyed. These checks are made during a “sanity check” step. If a node or DAG is “insane” (which can arise if a transformation is invalid) DxTer halts execution. In MDE

---

\(^3\) We omit the output number in the graphical representation of DLA DAGs as very few DLA nodes produce more than one output. When they do, we label the edges with variable names, which clarifies which output is used.
3.4 Transformations

Each transformation is encoded as a class with two functions. The first is boolean CanApply, which takes a node in a DAG, called box, and determines if box is part of a subgraph that can be replaced by the transformation. A node in the replaced subgraph is chosen to be the target box for the transformation, the one around which the transformation is written. The target node for a transformation can be any node in the subgraph being replaced. It is chosen for convenience while coding the transformation. The second function Apply takes box and applies the transformation to the DAG.

Recall Figure 4 (Bottom). Suppose the target box is the Triangular Inversion operation (TriInv). The CanApply function first checks if the box given as input is of the required type (TriInv). It then checks if one of the consumers of box is a Trmm operation. If so, CanApply returns true; otherwise false.

The Apply function first adds a Trsm node to the DAG and adds the appropriate inputs to it, box’s input and the other input of Trmm. It then redirects the consumers of Trmm to instead use Trsm as inputs. Finally, Trmm is removed from the DAG. If TriInv has no consumers at this point, then it is also removed. In general, transformations recursively remove nodes whose only consumers are removed. This is effectively an optimizing transformation to remove nodes that produce unused output (i.e. unnecessary computation or communication), but it is applied as a final step with most transformations instead of as a separate transformation. Figure 5 shows these steps graphically. Only the beginning and ending subgraphs represent the same functionality.

There are other transformations that do the same except for slight variations on properties such as data distribution. The transformations in Figure 3 are examples. It is useful, then, to represent these variations as instances of a templated transformation. This allows for many similar transformations to be represented in a compact way. For example, many abstractions have refinements that only differ in the way that data is distributed. Instead of creating a separate transformation for each refinement, one transformation can be created and templated over the redistribution used. This reduces the time it takes for an expert to encode her knowledge. Further, it reduces the chance of a coding error because the same code is used for many transformations.

4. Search Space

DxTer is given an input DAG that represents the functionality to be implemented. This DAG contains one or more abstraction nodes and any number of concrete nodes. DxTer maintains a separate set of DAGs that implement the input DAG’s functionality, which is the search space. DxTer tries to apply all transformations to all nodes on all DAGs. For each transformation that can be applied, the DAG is copied and the transformation is applied to the copy. The newly-created DAG is compared to all existing DAGs; if the new DAG is unique, it is added to the search space. This is necessary because there are commuting diagrams of transformations. Once a transformation is applied to a node (to form a new DAG), that node is marked such that the transformation is never re-applied to the node (which would result in the same DAG as was already created). These markings are not carried over to DAG copies when applying a transformation, though. If markings were copied, combinations of transformations would not be applied to a starting DAG that had those transformations applied separately.

4.1 Phases

Transformations are applied in two phases similar to how we have observed experts implement DLA code: first one parallelizes and then optimizes redistribution of data. In the first phase, which we call the distributed parallelization (DP) phase, DxTer applies optimizations written in terms of abstraction nodes and refinements to parallelize abstractions for distributed memory. These transformations choose implementation details and make higher-level algorithm changes (e.g. exploiting associativity of abstractions). The redistribution optimization (RO) phase mostly applies transformations that deal with redistribution boxes. DP transformations add redistribution boxes, which can generate code directly. This code is the best implementation for a single redistribution, but may not be optimal in combination with other redistributions, which refining transformations explore. Locally sub-optimal implementations allow for optimizing transformations to replace inefficient communication subgraphs with more efficient implementations, achieving designs that are best overall. The RO phase is

---

3 As mentioned earlier, we know of no MDE tool that generates a space of implementations in the way we need.

4 We use this approach to encode transformations as it is sufficiently general-purpose to encode even complex transformations. We do not know of existing tools that provides this general functionality in a way that works with DxTer.

5 Often two transformations can be applied in either order to produce the same DAG.
where Elemental experts and DxTer spend most of their time exploring options.

The DP and RO phases reflect the layering one finds in DLA code. A DLA expert first parallelizes abstractions and makes high-level changes to algorithms (DP phase). Once these choices are made, experts optimize and refine lower-level abstractions like redistribution operations (RO phase). For the DLA examples that we have studied, two phases seem sufficient.\(^7\)

As of the time of this writing, DxTer’s approach to generating the search space is a brute-force breadth-first search.\(^8\) When all DAGs in the search space are generated, we say the space is fully expanded; at this point cost estimates are attached to each implementation and the “best” is chosen.

4.2 DAG Culling

Once DAGs have been as refined as far as possible with DP-phase transformations, those that still contain abstraction nodes are removed from the search space. We call this culling. Recall Cholesky has three abstractions (Figure 2). After the DP phase, the Chol abstraction is on many DAGs with all of the ways to implement Trsm and TriRK together, the Trsm abstraction is on DAGs with implementations of Chol and TriRK, etc. DAGs with an abstraction cannot generate code (because abstractions cannot generate code). Since RO-phase transformations only deal with concrete node types, DAGs containing abstractions can be culled without removing useful DAGs from the search space. As there is a combinatorial number of culled DAGs, this substantially reduces the size of the search space and, therefore, DxTer runtime.\(^9\) After culling, the RO phase runs. Culling uses knowledge of the domain’s transformations to improve the search substantially. Without culling DxTer would take days to complete a complex algorithm implementation.

4.3 Simplifiers

In addition to DP and RO phase transformations, there are simplification transformations, which we call simplifiers. These are transformations that do not require a DAG to be copied. Instead, they are applied directly as simplifiers are always worth applying. An example is removing redundant communication, something an expert would always do.

Simplifiers are run after each DP and RO transformation. That means a graph to be transformed by a non-simplifier is copied, the non-simplifier transformation is applied, and then zero or more simplifiers are applied (whatever can be applied). Then the resulting graph is compared to existing graphs. Only if it is unique is it added to a PSet. Simplifiers effectively prune the search space of implementations that are always more costly (no matter what other optimizations are performed later). For the Cholesky example, simplifying transformations reduces the search space of implementations from 2,744 to 174. Using domain knowledge, we classify some transformations as simplifiers to reduce the search space substantially; for complex algorithms simplifiers are essential to reduce the search space from hundreds of millions of implementations to hundreds of thousands.

5. Possibilities and Possibility Sets

Each of the three abstractions in Figure 2 can be refined in multiple ways. The result is a combinatorial explosion of implementing algorithms. Up to a point, the ways to implement the Chol abstraction can be explored independently from the ways to implement Trsm and TriRK. For that reason, DxTer explores implementations of abstractions independently and in parallel when possible. After all implementations are generated, DxTer takes the cross product of these implementation sets to allow optimizations to be applied.

This is similar to how an expert deals with parallelizing and optimizing an algorithm. She focuses on each of the operations separately, thinking about implementation options. Here, we do that by considering all possible implementations of operations in isolation. Then, the expert considers how the choices affect each other, and optimizes.

5.1 Use of Possibility Sets

A possibility is a weakly connected subgraph that implements an abstraction. (Note: the abstraction of a possibility may be a single operation or a composition of operations). A possibility set (Pset) is a set of possibilities—a set of implementations of the same abstraction. The abstraction of a Pset has input and output ports that are connected to the nodes of each possibility. A Pset is always connected to the DAG from which its abstraction originated. This enables nodes of possibility DAGs to query input nodes for data properties.

The DAG in Figure 2 starts with three different Psets, one for Chol, Trsm, and TriRK. Initially each set contains a single possibility: the abstract version of each operation. DxTer transforms the possibilities in each Pset independently, adding new implementations as they are found. Once the Psets are fully expanded (modulo their culling, which we discuss shortly in Section 5.2) their cross products are formed, yielding a single large Pset that contains the implementations of the 3-input, 3-output abstraction of Figure 2. We call the cross-product of Psets merging.

After a pair of Psets has been merged, DP-phase transformations are applied to the result, and this continues until all Psets have been merged and fully expanded.\(^10\) Figure 4 (Bottom) shows the reason for reapplying DP-phase transformations: it is a rewrite that deals with two connected

\(^7\)When we deal with even lower-level operations, such as the sequential BLAS [13], additional phases will be needed.

\(^8\)Many search techniques have been explored in AI literature. We will investigate options in the future as our search space gets larger and our needs become clearer.

\(^9\)Without culling between the DP and RO phases, runtime jumps from hours to days for large problems. At this point, any experiment to document this is not interesting and is just a waste of power.

\(^10\)Pset merging is done in parallel.
abstractions in the same graph. Pairing graphs (containing one of these abstract nodes) enables an optimization to be applied to their product.

At the beginning of the RO phase, all Psets have been merged. The subgraphs in the RO phase can be explored independently to find all implementations and then merged and transformed further. Therefore, DxTer forms new Psets from subgraphs of each of the possibilities generated by the DP phase before the RO phase begins.

As RO-phase transformations generally affect redistribution nodes, we currently use a simple heuristic that chooses subgraphs of redistribution nodes. For a different domain, a different heuristic could be used—any weakly-connected subgraphs could be used. Since Psets are eventually all merged, the choice of subgraphs only improves performance, it does not affect the final results. The improvement with this, as shown in Section 6.3, is an over 50% reduction in runtime for our two complicated examples when Psets are re-formed at the beginning of the RO phase.

5.2 Culling During Merging

We said earlier that DAGs with abstractions are culled between the DP and RO phases because they cannot generate code (which is our goal) and no RO-phase transformations apply to them (so no new DAGs are generated from them). Similarly, most possibilities with abstractions can be culled when merging Psets because refinements have already been applied.

The Chol abstraction, for example, is on some possibility in the Chol Pset. There is no reason to generate the cross product of the possibility with the Chol abstraction with all of the possibilities of the Trsm and TriRk Psets because no transformation could be applied to the resulting possibilities that has not already been applied (i.e. only refining transformations involve Chol). This culling can significantly speed up the time it takes to generated the search space in the DP phase because it limits the search space of graphs that are eventually thrown away (and the search space is already combinatorial in size).

Abstractions found in optimizing transformations like that of Figure 4 (Bottom) prohibit DxTer from culling some possibilities, though. TriInv or Trmm might be found separately on possibilities of two different Psets, so the subgraph pattern the transformation applies to will not be formed on a single possibility graph until the Psets are merged. Therefore, DxTer cannot cull the possibilities with either abstraction so that the cross product can be formed and the transformation can be applied. Fortunately, there are not many optimizing transformations found thus far that apply to abstractions (most apply to concrete boxes), so culling during merging is still beneficial.

Currently, node types are manually marked to specify which can be culled and which cannot. Specifically, all abstract node types are marked unless they are found in the graph that some optimizing transformation replaces. In the future, automatic analysis of transformations could be used to find this information.

5.3 Loops

Dense linear algebra algorithms are largely loop-driven. When implementing an algorithm in distributed-memory, an expert focuses on the loop body, which is why we optimize only the body of the Cholesky algorithm’s loop in Figure 2. Some algorithms contain multiple loops, each of which can be implemented and optimized independently. In some cases, those loops can be fused (merged) to further optimize the loop body by re-using intermediate data distributions. In [18] there is an explanation of the analysis required for loop fusion. Loops represent some functionality, the loop body can have multiple implementations, and loops can be fused (read: merged) under some conditions. These are three characteristics similar to Psets, so loops are implemented as a flavor of Psets—they are Psets that generate code (the loop and indexing code) and store information about how the loop body alters data. Loop fusion is the same as Pset merging, but it is only done when the analysis shows the fused loop maintains correctness [18].

5.4 Inlining Possibility Sets

Abstractions are often put into individual Psets to improve performance while expanding them. As abstractions can be implemented using graphs that can contain other abstractions (implemented with their own Psets), hierarchical Psets are common in DxTer.

For the same reason Psets are merged (i.e. to enable cross-Pset DAGs to be replaced by transformations), hierarchical Psets are inlined. When a hierarchy of Psets is fully expanded, the lowest-level Pset is inlined and DxTer attempts to further expand the hierarchy, repeating this until all possible Psets are inlined. A Pset representing a loop cannot be inlined because the Pset itself generates code for the loop.

When inlining a Pset, a duplicate of the owning DAG is created for each possibility and the possibility’s DAG is inlined into the duplicate. All nodes on the duplicate DAG are redirected from being connected to the Pset’s ports to the nodes on the possibility DAG being inlined, and those nodes are connected directly to the nodes on the duplicated owning DAG. Not counting loop Psets, inlining and merging leaves DxTer with a single Pset at the end of each phase.

6. Results

This section focuses on DxTer’s performance and the results of design decisions and optimizations described above. We first show performance results on generated code (ultimately, this is of primary interest to end-users). But the real focus of this section and this paper is on the efficiencies that we have obtained in producing this code.
6.1 Generated Code Performance

For the operations described in this section and all others tested with DxTer, the code generated and chosen as best is effectively the same as that which is hand-crafted by an expert. In a few cases, it is better. In two cases, DxTer applied optimizations when the expert forgot to apply them. In one case, the Elemental expert wrote incorrect code because he made a mistake while implementing an algorithm; DxTer generated the code correctly. After showing the DxTer-generated code, the expert applied these optimizations to his code. In [19] we show more performance graphs.

We tested performance on the BlueGene/P cluster Surveyor at Argonne National Labs. We used 64 nodes, each with 4 IBM PowerPC 450 processors running at 850 MHz, which have a combined theoretical peak performance of 870.4 GFLOPS.\(^{11}\) Two-thirds of the machine’s peak performance is shown at the top of graphs.

Figures 1 and 6 show performance results for Hegst, described below, and SPD matrix inversion, respectively. We compare performance of the DxTer-generated code (which is the same as the hand-written code by the Elemental expert) against ScaLAPACK [7], which is the most widely-used package in the field. It is important to note that ScaLAPACK is hand-written; DxTer outperforms ScaLAPACK by over 20\%, which is significant. The Elemental language allows one to code better implementations than ScaLAPACK and its structure allows DxTer to generate that code automatically [19].

For SPD matrix inversion, we show results for two generated codes. The “Optimized” version is the “best” chosen by DxTer. It comes from applying a transformation to fuse three loops by analyzing properties of the loops (the loop body can be optimized further than three separate loop bodies). The “Non-fused” version does not have fused loops. We note the difference because to determine if loops can be fused is a complicated transformation that only few experts attempt. DxTer has this transformation encoded, so it can apply it automatically for many algorithms with multiple loops. For comparison ScaLAPACK does not have fused-loop code.

An expert manually implementing these algorithms and optimizing them can take many hours to reach the same code (or lower-performing or incorrect code) that DxTer automatically produces. The key here is that DxTer enables an expert to encode the knowledge she would use and then automatically applies it repeatedly no matter how many operations are to be implemented.

Next, we examine the performance of DxTer itself.

6.2 Transformations

DxTer currently implements a wide variety of DLA algorithms [19]. In Figure 7, we show the number of transformations encoded, broken down by their type. The “Total Count” column is the number of transformations of each type. Many transformations are templatized. In the “Unique Count” column, we list the number of unique transformations, counting each templatized transformation as one.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Total Count</th>
<th>Unique Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP Phase</td>
<td>70</td>
<td>40</td>
</tr>
<tr>
<td>RO Phase</td>
<td>55</td>
<td>10</td>
</tr>
<tr>
<td>Simplification</td>
<td>655</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 7. Count of encoded DLA transformations.

Due to the nature of DLA software, many of the non-templatized transformations are similar. Most take one argument, have the same preconditions, and parallelize in the same way. Such transformations were easy to create from existing code. Some only required copying and pasting code with minor changes to the functions called in the generated code. Most transformations are about ten lines of code for each CanApp1y and App2y method. Only a handful reach 20 or more lines of code. Overall, the DxTer code base is 15,000 lines of C++ code. About 8,000 lines are domain-specific DLA code and the remainder is domain-independent DxTer code (e.g. graph generation and search code).

6.3 Search Space

We want to understand the complexity of options an expert considers and the power of DxTer to automate this exploration. To do so, we look at four prototypical DLA algorithms. First is the Cholesky algorithm in Figure 2 (called variant 3) and an alternate algorithm called variant 2 [31]. Next, we tested an operation called two-sided triangular solve, or Hegst, which performs the update \(A := L^{-1}AL^{-T}\) where \(A\) is general matrix and \(L\) is a lower-triangular matrix [23]. Hegst has a relatively complex loop body with 8 abstractions. Lastly, we tested an algorithm to invert a SPD matrix [3], which consists of three loops that can be fused. One of those loops is Cholesky variant 3.
Figure 8. Comparison of DP phase with and without culling during Pset merging and count of DAGs without abstractions at the end of DP phase.

<table>
<thead>
<tr>
<th></th>
<th>DP with culling</th>
<th>DP without culling</th>
<th>Speedup</th>
<th>RO starting count</th>
</tr>
</thead>
<tbody>
<tr>
<td># Graphs</td>
<td>Time (secs)</td>
<td># Graphs</td>
<td>Time (secs)</td>
<td></td>
</tr>
<tr>
<td>Cholesky var 2</td>
<td>28</td>
<td>55</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cholesky var 3</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hegst</td>
<td>288</td>
<td>3,888</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>SPD Inverse</td>
<td>6,343</td>
<td>87,145</td>
<td>1,070</td>
<td>53.5</td>
</tr>
</tbody>
</table>

Figure 9. Comparison of RO phase with and without forming Psets at the beginning, demonstrating the performance improvement using Psets. The final number of implementation DAGs are shown. Additional, the time it takes to calculate their costs and find the “best” is also shown.

<table>
<thead>
<tr>
<th></th>
<th>RO with Psets</th>
<th>RO without Psets</th>
<th>Speedup</th>
<th>Final count</th>
<th>Cost Analysis (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td># Graphs</td>
<td>Time (secs)</td>
<td>Time (secs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cholesky var 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>172</td>
<td>1</td>
</tr>
<tr>
<td>Cholesky var 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>Hegst</td>
<td>7,142</td>
<td>14,670</td>
<td>2.05</td>
<td>119,964</td>
<td>6,287</td>
</tr>
<tr>
<td>SPD Inverse</td>
<td>5,860</td>
<td>12,903</td>
<td>2.2</td>
<td>337,899</td>
<td>3,854</td>
</tr>
</tbody>
</table>

Figure 8 lists the results of these tests. We show the time to apply all DP-phase transformations and the number of generated graphs before the final culling. We list results both with culling during set merging and without culling during set merging and show the number of DAGs that result after culling between the DP and RO phase; these are the graphs that are used in the RO phase. We do not test results without culling between the DP and RO phase because the search space is unmanageably large.\(^{13}\)

For the Hegst and SPD inverse algorithms, there are many more component operations to parallelize and consequently more complicated DAGs to optimize than the simple Cholesky algorithms. For this reason, many more implementation graphs are formed in the combinatorial search space. With a larger search space, the benefit of culling while merging Psets becomes apparent. For SPD inversion, culling improves DP phase performance by a factor of $53 \approx 1070\text{ms}/20\text{ms}$.

Figure 9 shows the results of the RO phase. *DxTer spends the bulk of runtime in the RO phase, so optimizing it provides the greatest benefit.* We tested the time it took to apply all transformations without forming Psets and we tested with multiple Psets formed. This demonstrates the benefit of using Psets in DxTer.\(^{14}\) Either way, the same number of DAGs were created (shown in the figure); these represent implementations of the input algorithm (i.e. each node in the DAG represents parallel code). Lastly, we timed how long it took to perform the cost analysis to find the cheapest graph.

For Hegst and SPD inversion, forming Psets and independently exploring subgraphs reduced runtime by 51% and 55%, respectively. This is a significant improvement in the overall runtime. Eventually, our transformations will break through more layers of code and optimize at lower levels such as within the sequential computation kernels called by the currently-generated code. Those transformations will cause the number and size of DAGs to become substantially larger. We expect the benefit of reforming Psets at the beginning of other phases will become even greater. It might be possible to use culling during Pset merging in other phases or possibly even avoid merging altogether (as no optimizing transformation would require such merging). This will be an area of research as our work is extended.

One might wonder why DxTer takes longer to complete the RO phase and to choose a “best” implementation for Hegst than SPD Inverse when Hegst has many fewer DAGs to analyze. The SPD Inverse search space includes DAGs with Psets that represent loops that are not fused; there are 3 loops and they can be fused in 4 ways. Unfused loops partition the DAG into portions that are each limited in the number of implementation options but still contribute to the combinatorial size of the search space. The unfused loops are simple, so they take less time to expand and analyze for cost, but they still increase the search space substantially. All Hegst implementations, on the other hand, are complex DAGs, so they take longer to generate and analyze.

---

\(^{12}\) These results were obtained on a quad-core Intel Core i7 processor with hyper-threading running 8 threads.

\(^{13}\) As stated earlier, we do not test performance without culling between the DP and RO phase because the experiment would take days and no new code-generating implementations would be created.

\(^{14}\) The time it takes to form Psets at the beginning of the RO phase is relatively trivial (and included in the timing figures).
6.4 Parallelism

Possibility transformation, sanity checking, and Pset merging are all performed in parallel in DxTer\(^{15}\) using OpenMP parallel for loops and simple synchronization. For Cholesky there is not much of a speedup because the runtime is only two seconds. For Hegst and SPD Inverse, there is a 1.38 and 1.14 speedup, respectively, running eight threads on four cores. In our opinion, these results are not great (yet). We used rather basic parallelism, and see only a small speedup. Never-the-less, search parallelism is an interesting avenue to explore as we scale up our work.

7. Related Work

DxT was inspired by relational query optimization (RQO) [17, 26]. With RQO an SQL query is represented as a relational algebra expression. Algebraic identities are applied to generate a search space of equivalent expressions and cost functions of component operations are used to estimate the performance of their implementations. By choosing the expression with the cheapest cost, the most efficient implementation of the query is selected. DxT is essentially rule-based RQO of the late 1980s [17] applied to other domains.

DxTer is related to knowledge-based software engineering (KBSE) [2, 4, 14, 21, 25], where domain knowledge is expressed as transformations that can be automatically or semi-automatically applied. KBSE may have been limited by the size of its search spaces (or simply the ideas never caught on). In DxT and DxTer, we have found significant domains (e.g. DLA) whose PnF architectures have “short” derivations, meaning that the number of transformations needed to map from an abstraction to an algorithm is small (~ 15). It is this shortness that enables DxT and DxTer to work well.

Readers familiar with MDE will see a lot of similarities between DxT and MDE. Indeed, MDE concepts influence our work. We find MDE terminology naturally applies to DxT and DxTer. By using it, we can raise our level of discussion, without overwhelming readers with low-level details. DxT exclusively uses endogenous transformations (whose domain and codomain are the same). MDE literature and tools tend to focus on exogenous transformations to map from one architecture to another [1, 9, 15, 30], with model refactorings as exceptions. Another deviation from most existing MDE work is that optimizations in the domains we have investigated require transformations across abstraction boundaries—after choosing refinements, optimizations are needed to remove inefficiencies.

Practical tools for building PnF architectures have a long history. LabVIEW [20], Weaves [12], and Fractal [5] are platforms for executing hierarchical PnF architectures, but they only support refinements. User-defined optimizations are necessary for the DLA domain. Other component-based systems follow a similar approach [5, 8, 11, 32]. More recent tools are StreamIt [29] and Click [16]. StreamIt is a general language for expressing stateless PnF applications; it uses map-reduce for automatic parallelization. Click demonstrated the feasibility of programmable routers using PnF architectures. With few exceptions [12, 16, 29], the above-cited tools are really aimed at bottom-up development; none handle architectural optimizations. In short, DxT is an MDE way to synthesize domain-specific PnF architectures that could be implemented by these tools, but these tools cannot express all transformations used by DxT.

The SPIRAL project [24] and auto-tuning approaches like ATLAS [33] have similar goals to DxT. They generate code specifically for a target architecture. SPIRAL employs rewrite rules to change the algorithm used and tune parameters for the target architecture. Auto-tuning changes algorithmic parameters like block size and unrolling factors, but generally uses the same algorithm. In both cases, there is a search space of possibilities as with DxT. For each implementation under consideration, ATLAS and SPIRAL generate code, compile it, and run it on a particular architecture to guide the search (instead of cost functions as in DxTer). They use performance information to search the space for the “best” algorithm and parameters. For distributed-memory DLA algorithms, the runtime is too long to perform an online search to find the “best” implementation. As our results have shown, cost functions are sufficient to find the same implementation as an expert generates, so online learning is not necessary. Online learning techniques could theoretically be worked into DxTer if cost functions cannot be created for components represented in the future.

In [10], a DLA compiler similar to DxTer is described. There, the input is a high-level description of an operation and the output is an algorithm for it. This algorithm is similar to what we expect as the input to DxTer. The rewrite rules they use expose math properties to, for example, invert a matrix with different approaches based on data properties. Further, rules are applied to reuse common subexpressions to reduce computation. This is similar to DxTer, but it works at a higher-level, dealing more with the DLA math than the implementation of that math in code.

Equality saturation is presented in [28] to limit redundant searching; this is similar to our Psets. Psets correspond to equivalence classes. Intermediate data reuse is not represented in their system as it is in DxTer. Instead, the best implementation from each equivalence class is chosen and the resulting code is transformed to reuse data. If DxTer were to do this, it would miss transformations that find the “best” program. Said another way, DxTer would get stuck in a local minimum. DxTer represents re-used intermediate data in the starting DAG and merges Psets to allow more optimizations than can be represented in [28].

\(^{15}\)Cost analysis could be parallelized, but is not currently.
8. Conclusion

We showed how DxT encodes expert knowledge about algorithm implementation and optimization in terms of graph transformations. Further we explained how DxTer automatically applies these transformations to produce a space of feasible implementations, and how it selects the “best”. The code that DxTer generates is the same as or better than an expert for a handful of indicative algorithms and can do so without human assistance. To our knowledge, this is the first time that this has been accomplished.

The space that DxTer generates is a combinatorial explosion of possibilities, so optimizations to avoid unnecessary work (searching) were developed to reduce DxTer’s run time. We use simplifying transformations to prune the search space of sub-optimal implementations and applied transformations in phases so DAGs can be removed, or culled, between phases when they are no longer useful. We partition DAGs into subgraphs and form sets of implementations for each so they can be explored independently and then later merged to produce implementations of full-graphs. All of these optimizations significantly reduce the amount of time it takes for DxTer to identify efficient implementations.

DxTer is a prototype to test the DxT approach, so it is actively being improved as DxT is further developed. This paper presents the current state of DxTer and the lessons we have learned so far. We expect this to be the first paper of many to describe DxTer’s inner workings, which will change as DxTer evolves.

Acknowledgments

We thank the other members of the FLAME team for their support and especially Jack Poulson for his help understanding Elemental. Marker was sponsored by a fellowship from Sandia National Laboratories and an NSF Graduate Research Fellowship under grant DGE-1110007. Batory is supported by the NSF’s Science of Design Project CCF 0724979. This research was also partially sponsored by NSF grants OCI-0850750 and CCF-0917167 as well as by a grant from Microsoft. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References


