KODA: Knit-Program Optimization by Dependency Analysis

Megan Hofmann
Khoury College of Computer Sciences, Northeastern University
Boston, Massachusetts, USA

ABSTRACT

Digital knitting machines have the capability to reliably manufacture seamless, textured, and multi-material garments, but these capabilities are obscured by limiting CAD tools. Recent innovations in computational knitting build on emerging programming infrastructure that gives full access to the machine's capabilities but requires an extensive understanding of machine operations and execution. In this paper, we contribute a critical missing piece of the knitting-machine programming pipeline–a program optimizer. Program optimization allows programmers to focus on developing novel algorithms that produce desired fabrics while deferring concerns of efficient machine operations to the optimizer. We present KODA, the Knit-program Optimization by Dependency Analysis method. KODA re-orders and reduces machine instructions to reduce knitting time, increase knitting reliability, and manage boilerplate operations that adjust the machine state. The result is a system that enables programmers to write readable and intuitive knitting algorithms while producing efficient and verified programs.

CCS CONCEPTS

• **Human-centered computing**;

KEYWORDS

Program Optimization, Machine Knitting, Digital Fabrication

ACM Reference Format:
Megan Hofmann. 2024. KODA: Knit-Program Optimization by Dependency Analysis. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3654777.3676405

1 INTRODUCTION

Despite the advanced manufacturing technologies available to textile designers, there is still a substantial gap between what fabrics machines can make and what designers can readily create using computer-aided textile design tools [54]. In the space of machine knitting, an emerging area of research is developing systems that give designers full access to the capabilities of these machines rather than limiting them to template garments (e.g., hats, socks, sweaters). With these tools, designers can knit 3D models [18, 35, 37], create actuating devices [2, 11, 20, 43], and design soft sensors [1, 27, 39, 40].

However, the infrastructure for building these knitting CAD tools is still in its infancy. While knit designers are primarily concerned with exploring the complex space of knitted structures, knit programmers must develop algorithms that translate these structures into machine-level instructions. Machine languages such as Knitout [30, 31] give programmers full access to the capabilities of the machine and a higher-level programming interface in general-purpose languages such as Python and JavaScript [46] to build their knitting algorithms. More recently, the KnitScript programming language [15, 17] provides a more direct conduit for knit programmers to generate machine code in a domain-specific language that conveys the state of the machine and handles knitting-specific errors.

But, this is where the infrastructure ends—hardware-specific languages that assume the programmer can manage both the complexity of the space of knitted objects and the low-level hardware constraints and efficiency considerations that make fabrication feasible. In practice, many advances in machine knitting have explored different approaches to efficient machine knitting [24–26], while explorations of the novel and exciting materials these efficient knitting algorithms may enable are frequently limited to simple swatches and objects that cannot be combined to form more advance garments and objects.

In this paper, we address a critical gap in knit-programming infrastructure: efficiency. While the full space of knitted materials requires various solutions and domain-specific tools, our aim is to develop a unifying optimization method that reduces a set of knitting machine instructions into an efficient program that may increase the reliability of the knitting process. By optimizing low-level code, our approach is compatible with a wide range of systems. This enables programmers to disregard efficiency and focus on developing tools that unlock the machine knitting's potential.

We present KODA (Knit-program Optimization by Dependency Analysis), a system for optimizing knitting machine code that is compatible with the Knitout machine language [30, 31] and higher level programming languages that generate knitout (e.g., [15, 46]). KODA includes three components. First, we provide an expanded knitgraph representation that covers the complexity of knit objects, including loop and stitch placement, float arrangements between multiple yarns, and braided wales. Second, we develop a dependency graph representation of knitted programs that models the relationships between knitting operations and the knitted structures they produce. This dependency graph is constructed through a single pass through a knitting program. Third, we provide a knit-program optimization method that uses dependency analysis to identify an efficient program that will produce the same knitted structure. We evaluate this system through a series of benchmark knitting programs and demonstrate that the system produces semantically correct programs that are often substantially more efficient, increase knitting reliability, and fabricate correctly.

UIST '24, October 13–16, 2024, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0628-8/24/10
https://doi.org/10.1145/3654777.3676405

2 RELATED WORK

There is growing interest in novel machine knitting techniques (e.g., [2–4, 12, 20, 27, 28]) due to emerging opportunities to create novel and impactful textile technologies such as medical devices [9, 22, 33, 50], assistive technologies [44, 47], soft-robotics [21, 42, 43], and wearables [7, 38, 49]. This growing interest has led to the development of new programming infrastructure for automatic knitting machines, generalized representations of knitted structures [16, 37, 51], and knit-scheduling methods [24–26].

2.1 Knit-Programming Infrastructure

Programming, while less accessible than design tools, is a powerful tool for digital fabrication. Programming language concepts can support diverse areas such as carpentry [48, 52], quilt patterning [23], and knitting self-embedded structures [53]. Unlike other domains, programming is often the only way to take full advantage of machine knitting. Many recent advances in machine knitting rely on McCann et al.'s [31] knitting assembly language, Knitout [30]. Knitout is rarely written directly; usually, it is generated by programs written with APIs for general-purpose languages (e.g., [46]) or the domain-specific scripting language, KnitScript [15, 17]. Using these programming tools, we can reason about the knitting process algorithmically and develop new knitted structures.

2.2 Program Dependency Analysis

Unfortunately, the most understandable and reusable code is rarely the most efficient. In practice, efficient code may introduce errors in edge cases and execute in a way that wastes computing resources. Fortunately, program verification and optimization are mature areas of software engineering. Despite the plethora of established programming optimization methods, new domain-specific optimization problems are a frequent source of innovation in domains that rely on domain-specific languages (e.g., [6, 41]). While knit programming implies novel physical constraints, it may benefit from tried and true optimization methods.

One such method is dependence analysis [8, 19, 36], which models a program as a set of interdependent nodes in a directed graph. For manufacturing, this graph is an acyclic graph. All programs executed by following paths through this graph will produce a program with the same semantics as an original. For example, dependence analysis has been used to optimize programs by removing dead-code [13]–statements that do not affect the program execution. Not only would removing dead code from a knitting program improve execution time, but it may also reduce wear and tear on the knit object by limiting the wear and tear on the yarn incurred with every operation. Similarly, dependency analysis has been used to schedule parallel programming [5]. In the context of machine knitting, this approach may identify clusters of operations that can be scheduled in parallel.

3 SYSTEM OVERVIEW

To bridge the gap between the intuitive knitting algorithms and the efficient programs that are needed to reliably knit, we present the KODA method for *Knit-program Optimization by Dependency Analysis*. KODA is built on three components: an expanded knitgraph representation of knitted structures, a dependency graph structure that models semantically identical programs that produce those structures, and a dependency analysis method that clusters operations into an efficient ordering of carriage passes.

KODA improves the efficiency of knitout programs [30] that are compiled into instructions for a variety of knitting machines. KODA does not optimize higher-level knitting programs (e.g., KnitScript [15, 17]). However, by optimizing the intermediary knitout code produced by these tools, we have created an optimizer that is compatible with a wide range of knit-programming pipelines.

4 EXPANDING KNITGRAPHS

To optimize a knitting program that produces a specific knitted structure, we must have a complete representation of that structure. We expand on the loop-based knitgraph structure defined by Hofmann et al. [16]. As in prior work, we define a knitgraph $K(\mathbb{L}, \mathbb{S})$ as a set of loops, $\mathbb{L}$, connected by the set of stitch-edges $\mathbb{S}$ where the stitch-edge $s_{u,v}$ indicates that the loop u is pulled through the loop v to form a stitch. We expand on this knitgraph structure to represent two missing characteristics of knitted objects. First, we adapt Lin and McCann's [24] Artin-Braid representation of loop-crossings to represent how loops are tangled and crossed to form cables and decreases. Second, we define a new yarn representation that models both loops and their relative position to floats in the yarns. For convenient reference, Table 1 provides a summary of relevant knitting terms and the notation we use in this paper.

4.1 Loop and Wale Braids

Knitted structures are not necessarily grid-like; crossing loops form unique knitted structures such as decreases and cables. In knitting, a *wale* refers to a series of loops connected by stitches, such as the stitches that form columns in a rib pattern. Lin and McCann demonstrated [24] that we can represent crossing wales as a braid where each wale is a strand of the braid. Each strand in the braid can cross over or under its neighboring strands. The opportunity to cross wale strands occurs at each loop along the wale.

We represent all of the crossings of wales in the knitgraph using a braid-graph $B_K(\mathbb{L}, \mathbb{B})$ which includes a node for every loop in the knitgraph and a set of directed edges $\mathbb{B}$. A crossing edge $b_{u,v} \in \mathbb{B}$ implies that the loop u is passed to the right and crosses the loop v. The loop u can cross either over v (i.e., $b^+_{u,v}$) or under v (i.e., $b^-_{u,v}$). We use the convention that edges are directed by rightward crossings. Thus an edge from $b^+_{u,v}$ also implies that v is crossed leftward under u, and $b^-_{u,v}$ implies that v is crossed leftward over u.

4.2 Float

Prior representations of knitgraphs assume only one yarn in the structure. However, multiple yarns are required for multi-material structures. In these cases, a knitted structure is defined both by the order of loops formed on the yarn and the position of loops relative to the floats between them. Modeling floats assists in managing *slack* [31], which constrains the placement of loops on the needle bed. Once a float is formed, the length of yarn between them is finite and should those loops be transferred further away from each other the yarn may tear or bend needles. Managing slack constraints is critical to many transfer planning algorithms (e.g., [26, 32, 34]).

Table 1: The set of notation we use to describe knitted structures and the knitgraphs that represent them.

Term	Definition	Example Notation	Section
Loop	A loop of yarn (i.e., fiber) that make up a knitted structure.	$l \in \mathbb{L}$: The loop l is in the set of loops $\mathbb{L}$. $l \in y$: The loop l is formed by the yarn y.	4
Stitch Edge	Stitches are formed by pulling a loop through other loops. A stitch-edge is a directed edge in a knitgraph that represents that one loop u is pulled through another loop v. The set of stitch-edges in a knitGraph is denoted $\mathbb{S}$.	$s_{u,v} \in \mathbb{S}$: The loop v is pulled through the loop u.	4
Course	A horizontal row of stitches in a knit fabric, usually, but not always connected to the same yarn.	NA	NA
Wale	A wale is an ordered set of stitches where the child loop of each stitch is pulled through the next in the wale.	NA	4.1
Wale crossing	Loops can cross over each other causing wales to cross. This is modeled as crossings in a braid structure.	$b_{u,v} \in \mathbb{B}$: the loop u crosses the loop v. $b^{+}_{u,v}$: The loop u crosses over the loop v. $b^{-}_{u,v}$: The loop u crosses under the loop v.	4.1
Yarn	A yarn is a fiber that makes up the loops in a knitted structure. In a knitgraph, a yarn is a set of loops in their knitting order.	$y \in \mathbb{Y}$: The yarn y is in the set of yarns used in a knitgraph.	4.2
Float	A float is the length of yarn between two connected loops. Directed edges represent floats between loops on the same yarn. Floats, like wales, can cross over each other loops.	$f \in \mathbb{F}_y$: The float f is in the set of floats on the yarn y. $f_{u,v}$: There is a float from the loop u to the loop v. $f^{+}_{u,v}$: The set of loops that the float from u to v crosses over. $f^{-}_{u,v}$: The set of loops that the float from u to v crosses under.	4.2
Needle	The mechanism that forms, holds, knits and transfers loops. Needles are arranged on a front and back needle bed.	n_i: A needle a the ith index on a needle bed. $\mathbb{N}_{front}$: The ordered set of needles on the front bed. $\mathbb{N}_{back}$: The ordered set of needles on the back bed. $\mathbb{L}_n$: The set of loops held on the needle n. $\mathbb{L}_{[n_i,n_j]}$: The set of loops held on needles from indices i to j.	4.3
Carriage Pass	A set of operations executed on the knitting machine by passing a carriage that actuates needles across the needle beds.	NA	6.2

To capture these relationships, each knitgraph has a set of yarns, $\mathbb{Y}$, representing the yarn-wise ordering of loops. Unlike prior work, we expand on the definition of a yarn to model both loops and floats–the yarn lengths between loops. A yarn, y, is a graph with loops as nodes (i.e., $l \in y$) and directed float edges connecting each loop, $f \in \mathbb{F}_y$. The direction of a float edge determines the time ordering of loops. That is, if a loop u is formed with the yarn y and then loop v is formed with y, then there will be a float $f_{u,v}$. If there is a path from a loop u to v along a yarn, then the loop v must be knit after the loop u. We refer to the first loop formed on y as its head and the last loop formed on the yarn as its tail.

The length of a float is set by the distance between the needles that form the loops. Float length is used to define slack constraints. Transferring loops to new needles that are further apart than the length of their connecting float will stretch the float beyond and risks tearing the yarn. Some stretch in a float is possible, depending on machine settings and the fiber. Thus, the original float length and a stretch tolerance define the upper limit on the distance between the needles holding the loops connected by a float.

Just as the loops in wales can cross over each other, loops can cross over or under floats. Careless placement of floats produces unseemly colorwork with strands crossing over the desired image. To represent these relationships, we give each float-edge, $f_{u,v}$, two properties. First, we denote the set of loops the float passes over as $f^{+}_{u,v}$. Conversely, $f^{-}_{u,v}$ denotes the set of loops the float passes under.

4.3 Forming Knitgraphs

Knitted structures are formed by executing a small set of operations in a knitting program. Lin et al. [25] provide a formal definition of the operations of a knitting machine and each of their effects on the state of the machine. We generate our knitgraph structure using a virtual knitting machine that models these processes. We use a virtual machine model from the KnitScript interpreter [15, 17].

A knitting machine consists of front and back beds of aligned needles. The set of all needles is denoted $\mathbb{N}$, and we refer to the front and back beds as $\mathbb{N}_{front}$ and $\mathbb{N}_{back}$, respectively. Needles in each bed have increasing indices from zero on the left to the right. Each needle can form, hold, and transfer the loops in the knitted structure. We denote the set of loops currently held on a needle n as $\mathbb{L}_n$ and the set loops held on all needles over a range between two needles n_0 and n_1 as $l \in \mathbb{L}_{[n_0,n_1]}$. The machine has a set of yarn carriers that hold and position the yarns while knitting.

Knitting operations orchestrate the actions of needles and yarns and can update a knitted structure in three ways: *making a loop* (algorithm 1), *stitching a loop through loops on a needle* (algorithm 2), and *moving loops from one needle to another* (algorithm 3).

4.3.1 Forming New Loops. Algorithm 1 describes the updates to a knitgraph for making a new loop on a specified needle with a specified yarn. First, the loop is formed and added to the loop set of the knitgraph and the yarn. Next, if the yarn has prior loops, we form a new float edge between the last loop made on the yarn and the new loop. This float passes over loops formed on the back bed, and under loops formed on the front bed.

4.3.2 Stitching Loops. Algorithm 2 describes the updates to a knitgraph caused by pulling a loop through the loops on a given needle (i.e., stitching), which forms stitch edges to the given loop.

4.3.3 Moving Loops. Algorithm 3 describes the updates to a knitgraph caused by moving a loop on one needle to another needle in

Input n**:** The needle to form the loop on.
Input y**:** The yarn to form the loop at the end of.
Output l**:** The loop formed.
$l \leftarrow$ new tail loop on y;
add l to $\mathbb{L}_K$;
if *y has a loop l_y prior to l* **then**
  add $f_{l_y,l}$ to $\mathbb{F}_y$;
  if *l_y is currently on a needle n_y* **then**
    $f^+_{l_y,l} \leftarrow \{\hat{l} \in \mathbb{L}_{n_y,n}\} \cup \mathbb{N}_{front}$;
    $f^-_{l_y,l} \leftarrow \{\hat{l} \in \mathbb{L}_{n_y,n}\} \cup \mathbb{N}_{back}$;
  end
end
return l;

Algorithm 1: Make_Loop

Input l**:** The loop to pull through other loops in the stitch.
Input $\mathbb{L}_n$**:** The loops on a needle to be stitched.
Output $\mathbb{S}_l$**:** The set of stitch-edges produced.
$\mathbb{S}_l \leftarrow \{s_{l_n,l} \forall l_n \in \mathbb{L}_n\}$;
add $\mathbb{S}_l$ to $\mathbb{S}_K$;
return $\mathbb{S}_l$;

Algorithm 2: Stitch_Loops

an xfer or split operation. Moving loops affect both the placement of floats and the crossing of wales in the knitgraph.

Input l**:** The loop on needle n and formed on the yarn y.
Input $\hat{n}$**:** The needle to place the loop on.
Output $\mathbb{B}_l$**:** The resulting set of crossing edges.
if *$\exists l_y \in \mathbb{L}_y$ and on $n_y | f_{l_y,l} \in \mathbb{F}_y$* **then**
  $f^+_{l_y,l} \leftarrow \{\hat{l} \in \mathbb{L}_{n_y,n}\} \cup \mathbb{N}_{front}$;
  $f^-_{l_y,l} \leftarrow \{\hat{l} \in \mathbb{L}_{n_y,n}\} \cup \mathbb{N}_{back}$;
end
if *$\exists l_y \in \mathbb{L}_y$ and on $n_y | f_{l,l_y} \in \mathbb{F}_y$* **then**
  $f^+_{l,l_y} \leftarrow \{\hat{l} \in \mathbb{L}_{n_y,n}\} \cup \mathbb{N}_{front}$;
  $f^-_{l,l_y} \leftarrow \{\hat{l} \in \mathbb{L}_{n_y,n}\} \cup \mathbb{N}_{back}$;
end
$\mathbb{B}_l \leftarrow \{\}$;
if *$\exists l_s \in \mathbb{L}_k | \exists s_{l_s,l} \in \mathbb{S}_K$* **then**
  for $\hat{l} \in \mathbb{L}_{[n,\hat{n}]}$ **do**
    if *$\exists \hat{l_s} \in \mathbb{L}_k | \exists s_{\hat{l_s},\hat{l}} \in \mathbb{S}_K$* **then**
      add $l \times \hat{l}$ to $\mathbb{B}_l$; `/*see Equation 1*/`
    end
  end
  add $\mathbb{B}_l$ to $\mathbb{B}_K$;
end
return $\mathbb{B}_l$;

Algorithm 3: Move_Loop

First, we consider the placement of floats. Moving a loop repositions both the float it ends and the float it starts. This new float position may over-stretch the yarn, violating slack constraints. If the distance between the new start and end needles is greater than the length of a stretched float, an error is reported to the user. If all of the loops involved in a float are held on needles, this will shift the float past a new set of needles. The float will now be positioned relative to the loops on those needles. Thus, we update the over- and under-sets of each float using the same process that defined them when making a loop (see algorithm 1 lines 9-10).

Next, we consider the crossing of wales. Moving a loop, between two needles may cause it to cross loops held on needles between the transferring needles. Two loops can only cross if they are not the beginning of a wale. The prior loops in the wale will anchor the location of the wale and cause the crossing loop to form a braid. That is, each crossing loop must be stitched through at least one other loop (i.e., $\exists l_s | s_{l_s,l} \in \mathbb{S}_K$). Similar to Lin and McCann [24], we use the convention of recording rightward crossings. Thus, moving a loop rightward will create crossing edges over back bed loops and under front bed loops. If the loop is moving leftward, the orientation of the edge is reversed. To simplify notation, we define the cross operator between two loops as a shorthand for determining the orientation of a crossing edge between two loops (Equation 1).

$$u \times v = \begin{cases} b^+_{u,v} & \text{if } n_u < n_v \text{ and } n_v \in \mathbb{N}_{back} \\ b^-_{u,v} & \text{if } n_u < n_v \text{ and } n_v \in \mathbb{N}_{front} \\ b^+_{v,u} & \text{if } n_u > n_v \text{ and } n_u \in \mathbb{N}_{back} \\ c^-_{v,u} & \text{if } n_u > n_v \text{ and } n_u \in \mathbb{N}_{front} \end{cases} \tag{1}$$

4.3.4 Interpreting Operations on Knitgraphs. From these three update methods, we can interpret knitting operations (knit, tuck, xfer, split) as updates to a knitgraph and update the state of the virtual knitting machine. We can render a knitgraph generated by a specific knitting program by interpreting machine operations using these update processes.

A tuck instruction forms a new loop on a specified needle for each yarn in the specified carrier set. Knit instructions similarly form a new loop and then stitch it through the loops on the specified needle. The original loops on the needle are dropped from the needle, leaving the new loops behind. Xfer operations move loops on one specified needle to a different specified needle on the opposite bed. Finally, the split operation makes a new loop (like tucking), stitches that new loop through its predecessors on the needle (like knitting), but unlike knitting, moves those predecessors to another target needle on the opposite bed (like transferring).

Input n**:** The needle to tuck on.
Input *carriers***:** The set of carriers to tuck with.
Output $\mathbb{L}_{tuck}$**:** The loops formed.
$\mathbb{L}_{tuck} \leftarrow$;
for $y \in$ *carriers* **do**
  $l \leftarrow$ Make_New_Loop(n, y);
  add l to $\mathbb{L}_n$;
  add l to $\mathbb{L}_{tuck}$;
end
return $\mathbb{L}_{tuck}$;

Algorithm 4: Tuck

Input n: The needle to knit on.
Input *carriers*: The set of carriers to knit with.
Output $\mathbb{L}_{knit}, \mathbb{S}_{knit}$:The loops and stitch-edges formed.
$\mathbb{L}_{knit} \leftarrow \emptyset$;
$\mathbb{S}_{knit} \leftarrow \emptyset$;
$\mathbb{L}_{prior} \leftarrow \mathbb{L}_n$;
$\mathbb{L}_n \leftarrow \emptyset$;
for $y \in carriers$ **do**
  $l \leftarrow$ Make_Loop(n, y);
  add Stitch_Loops$(l, \mathbb{L}_{prior})$ to $\mathbb{S}_{knit}$;
  add l to $\mathbb{L}_n$;
  add l to $\mathbb{L}_{knit}$;
end
return $\mathbb{L}_{knit}, \mathbb{S}_{knit}$;

Algorithm 5: Knit

Input n: The needle to transfer loops from.
Input $\hat{n}$: The needle to transfer loops to.
Output $\mathbb{B}_{xfer}$: The wale crossings formed.
$\mathbb{B}_{xfer} \leftarrow \emptyset$;
for $l_n \in \mathbb{L}_n$ **do**
  add Move_Loop$(l_n, \hat{n})$ to $\mathbb{B}_{splxferit}$;
end
$\mathbb{L}_n \leftarrow \emptyset$;
add $\mathbb{L}_{prior}$ to $\mathbb{L}_{\hat{n}}$;
return $\mathbb{B}_{xfer}$;

Algorithm 6: Xfer

Input n: The needle to split on.
Input $\hat{n}$: The needle to transfer old loops to.
Input *carriers*: The set of carriers to split with.
Output $\mathbb{L}_{split}, \mathbb{S}_{split}, \mathbb{B}_{split}$: The loops, stitch-edges, and wale crossings formed.
$\mathbb{L}_{split} \leftarrow \emptyset$;
$\mathbb{S}_{split} \leftarrow \emptyset$;
$\mathbb{L}_{prior} \leftarrow \mathbb{L}_n$;
$\mathbb{B}_{split} \leftarrow$ Xfer$(n, \hat{n})$;
for $y \in carriers$ **do**
  $l \leftarrow$ Make_Loop(n, y);
  add Stitch_Loops$(l, \mathbb{L}_{prior})$ to $\mathbb{S}_{split}$;
  add l to $\mathbb{L}_n$;
  add l to $\mathbb{L}_{split}$;
end
return $\mathbb{L}_{split}, \mathbb{S}_{split}, \mathbb{B}_{split}$;

Algorithm 7: Split

5 KNITTING DEPENDENCY GRAPH

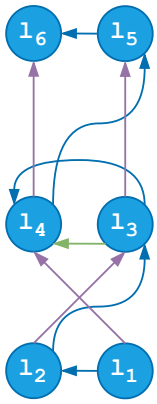
There may be many programs that produce any given knitted structure. We use dependency graphs to represent the space of programs that can produce the knitted structure created by a source program. Conceptually, a dependency graph is a directed acyclic graph where each node is a machine operation. An edge from one operation, u, to another operation, v, implies that v cannot be executed until u has been executed. Thus, any topological ordering of the dependency graph will produce a knitted structure with the included dependencies. We define a set of dependencies between knitting operations based on our knitgraph representation to ensure these dependency graphs produce a consistent knitted structure.

Given a program, $\mathbb{P}$, with an ordered set of machine operations that form the knitgraph K, we create a dependency graph $G(\mathbb{P}, \mathbb{D})$ where each edge $d_{u,v} \in \mathbb{D}$ indicates that the operation $v \in \mathbb{P}$ must occur after the operation $u \in \mathbb{P}$. Any topological ordering of the dependency graph, $\hat{\mathbb{P}}$, will produce the same knitgraph, K.

We form a dependency graph as we execute the original program. Note that each knitting operation (i.e., algorithms 5-7) returns the components of the knitgraph that it generated or updated. As we execute each operation in the program, we add the operation as a node to the dependency graph and associate it with the resulting structures (e.g., loops, stitches, and wale-crossings) that the operation created in the knitgraph. Each operation can have the following properties: the set of loops it formed (and the floats we can infer from this), the set of loops it positioned either by forming or moving (and the floats we can infer from this), the stitch-edges it created, and the wale-crossings it created. Each loop in the knitgraph is associated with the operation that formed it, the last operation to position it, and the last operation that passed a float across it. So, for each new operation, we can find a dependency to other operations that involve the same loops or edges connected to those loops.

```
tuck - f2 y; forms loop 1
tuck - f1 y; forms loop 2
knit + f1 y; forms loop 3
rack 1
xfer f1 b2; moves loop 3 to right
rack 0
knit + f2 y; forms loop 4
rack -1
xfer f2 b1; moves loop 4 to left
rack 0
xfer b1 f1; moves loop 4 to front
xfer b2 f2; moves loop 3 to front
knit - f2 y; forms loop 5
knit - f1 y; forms loop 6
```

(a) Twist Program



(b) Knitgraph

Figure 1: A knitting program that twists two stitches and the resulting knitgraph. Loop nodes are notated with their time-order index. Blue edges represent yarn-edges between loops, forming floats. Purple edges show stitch edges, connecting loops together. Green edges show wale crossing edges.

Consider an example illustrated in Figure 1. Suppose we execute a tuck operation, t_1, that returns the loop l_1. We associate this loop with t_1 and add it as a node in the dependency graph. Then we execute another tuck, t_2, with the same yarn that returns l_2 and forms the float f_{l_1,l_2}. We associate this new loop with the second tuck operation and add it to the dependency graph. Observe that f_{l_1,l_2} involves the loop l_1 with t_1 marked as its creating operation. From this, we can infer a dependency from t_1 to t_2 and add this edge to the dependency graph. Next, suppose we execute a knit operation k_3 that forms a new loop l_3 and stitches it through l_2 and

returns this as the stitch-edge s_{l_2,l_3}. We associate the stitch edge and new loop with the operation k_3 and form a dependency between t_2 and k_3 that ensures that t_2 forms l_2 before l_2 is knit by k_3. Also note that because there is a path from t_1 to k_3, any topological ordering will ensure that both tucked loops are formed before knitting.

To ensure we capture all of the features in the knitgraph, we detect five dependency types after executing each operation in the program and updating the knitgraph: *Yarn-order*, *Loop-Position*, *Float-Position*, *Stitch*, and *Wale-Crossing* dependencies.

5.1 Yarn-Order Dependency

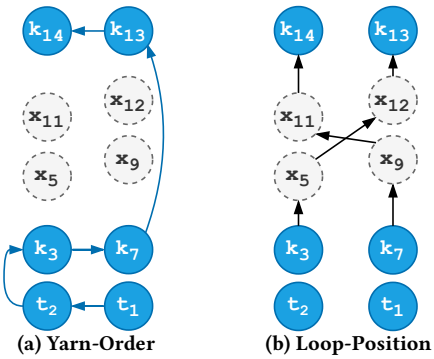
Yarn-order dependencies ensure the loops of each yarn are formed in the correct order. For example, the dependency created between two tuck operations that use the same yarn (e.g., t_1 and t_2 in Figure 2a). When executing a program, we check for yarn-order dependencies whenever an operation produces a new loop. We then look to see if there is any float in the knitgraph that leads to the returned loop. If there is, the proceeding loop will be associated with whatever operation created the loop. We can then create a yarn-order dependency between these two operations.

5.2 Loop-Position Dependency

Similar to yarn-order dependencies, we must keep track of the relationship between operations on loops and the position of those loops on needles. For example, Figure 2b shows the operation k_3 forms the loop l_3, and the transfer operation x_5 moves this loop to a new needle. We identify a Loop-Position dependency from k_3 to x_5 because they are both operating on the loop l_3. The subsequent transfer of the loop l_3 by transfer x_{12} will only create a loop-position dependency between x_5 and x_{12} because after executing x_5 the loop l_3 resets its last moving operation from k_3 to x_5.

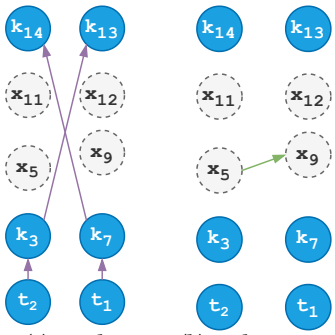
5.3 Stitch Dependencies

Stitch dependencies ensure stitches are formed in the wale-wise order of the original program. The formation of a dependency between t_2 and k_3 in Figure 3a shows how stitch dependencies are formed. Note that these dependencies can be created even if there are many intermediate steps between forming a loop and stitching through it. For example, k_7 produces the loop l_4 and stitches it



(a) Yarn-Order (b) Loop-Position

Figure 2: The yarn-order and loop-position dependency graphs for the twist program in Figure 1a. Operation nodes are labeled with their abbreviated operation type and line number (e.g., t_1 for the tuck on line 1).



(a) Stitch (b) Wale-Crossing

Figure 3: The stitch and wale-crossing dependency graphs for the twist program in Figure 1a.

through the l_1 formed by t_1. Despite the many operations executed between them, we identify the stitch dependencies from t_1 to k_7.

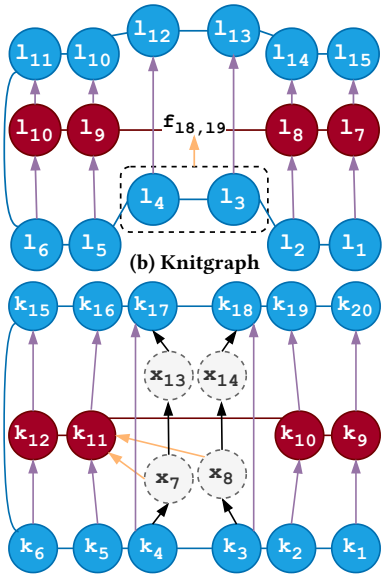
5.4 Wale-Crossing Dependencies

When an operation moves a loop, it may form a wale-crossing edge. For example, the transfer operation x_9 moves the loop l_3 to the left and creates a wale-crossing where l_4 crosses rightward under l_3 (i.e., $b^{-}_{l_4,l_3}$). This creates a wale-crossing dependency between the operation that last placed l_4, x_5, and the operation x_9. An example wale-crossing dependency is shown in Figure 3b.

5.5 Float-Position Dependencies

```
1  knit - f6 b
2  knit - f5 b
3  knit - b4 b
4  knit - b3 b
5  knit - f2 b
6  knit - f1 b
7  xfer b3 f3
8  xfer b4 f4
9  knit - f6 r
10 knit - f5 r
11 knit - f2 r
12 knit - f1 r
13 xfer f3 b3
14 xfer f4 b4
15 knit + f1 b
16 knit + f2 b
17 knit + b3 b
18 knit + b4 b
19 knit + f5 b
20 knit + f6 b
```

(a) Program



(b) Knitgraph (c) Dependency Graph

Figure 4: Colorwork rib where the red yarn floats over the blue purled loops. Float-position dependencies are orange.

Both when a loop is formed and when it is moved, the floats associated with that loop may pass over or under loops on other needles. With each of these operations, we associate the operation with the loops that the float passes over. When those floats are operated on, either by stitching or moving, we create a dependency between the last operation to affect the float and the operation acting on the loop positioned either in front of or behind the float.

Figure 4 shows a sample program that knits with two yarns b (blue) and r (red). The blue yarn forms a ribbed structure with columns of knit and purl loops (formed on the back bed). The red yarn passes the float between loops l_8 and l_9 in front of the blue purled loops l_3 and l_4. To do this, the blue purls are transferred to the front bed (i.e., x_7, x_8) and then returned to the back bed to be knit (x_{13}, x_{14}). The knit operation k_{11} forms this float, creating a float-position dependency between the k_{11} and the transfers.

6 KNITTING DEPENDENCY ANALYSIS

A dependency graph describes the space of knit programs that will produce the knitted structure produced by the original program. However, not all topological orderings of the dependency graph will produce an efficient knit program. We provide an optimization method that uses dependency analysis and leverages information about the machine knitting process to find an efficient program. This optimization process has two stages: *transfer dependency reduction* and *carriage-pass dependency analysis*. Dependency reduction allows us to reduce the number of operations in a knitting program without modifying the knitted structure. Carriage-pass dependency analysis enables us to order the remaining operations to make efficient use of machine resources.

6.1 Transfer Dependency Reduction

Given a knitting program with only tucks and knits, removing any operation would remove a loop from the knitgraph and substantially alter the structure. The place to look for operations that can be removed is in transfer operations. It is possible to undo the effect of wale crossings in a knitted structure. This would have the added benefit of making the optimized program more reliable because every transfer operation risks dropping the transferred loop or straining and tearing the yarn. A trivial example is transferring a loop and then transferring it to the original needle.

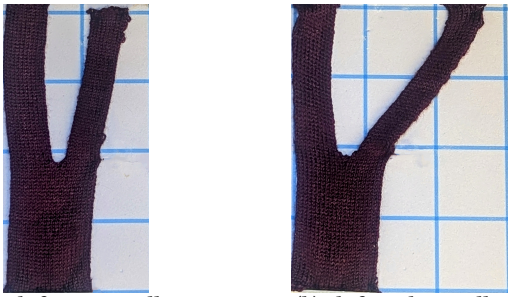
Suppose we execute the transfer x = *xfer f2 b2* and then the transfer x^{-1} = *xfer b2 f2*. The operation x^{-1} inverts wale-crossings formed by the x operation. Clearly, a more efficient knitting program would omit these two operations. However, suppose between x and x^{-1} we knit two loops to form a float across the transferring needles (i.e., $\mathbb{P} = \{x, k_0 =$*knit - f4 r*$, t_1 =$*knit - f1 r*$, x^{-1}\}$. Figure 8c show the differences in the loop and float positions when these transfers are removed (Figure 8a) and included (Figure 8b). The difference between the resulting knitted samples illustrates why it is critical that knitgraphs represent both wale and float crossings and that these relationships are reflected in the dependency graph.

To identify the transfer that can be removed, we define an inverted transfer series and a method for identifying them in a dependency graph. A transfer series is an ordered set of transfer operations where only the first transfer depends on any operations outside the series, and only the last transfer is dependent on operations outside the transfer series. A transfer series can be inverted if all of the effects of the first transfer in the series are undone by the time the last transfer in the series occurs. That is, the transfer series undoes all wale and float crossings it creates, and the transferred loops arrive at their original position. In this case, a transfer series can be removed from the dependency graph.

Input $G(\mathbb{P}, \mathbb{D})$**:** The dependency graph produced by execution of the program $\mathbb{P}$.
Input $\mathbb{X}$**: A transfer series in the graph** G **that ends with the operation** x**.**
$\mathbb{O}_{succ} \leftarrow \{o \in \mathbb{P} | \exists d_{x,o} \in \mathbb{D}\}$;
if $|\mathbb{O}_{succ}| == 1$ *or* $o \in \mathbb{O}_{succ}$ *is a transfer* **then**
  add o to $\mathbb{X}$;
  if $\mathbb{X}$ *is invertable* **then**
    remove $\mathbb{X}$ from G;
    return $\emptyset$;
  else
    return Extend_Transfer_Series($G, \mathbb{X}$);
  end
else
  return $\mathbb{X}$;
end

Algorithm 8: Extend Transfer Series

To find all invertible transfer series in the dependency graph, we iterate over every transfer operation in the original program execution order. Given a transfer x, we form a new transfer series $\mathbb{X} = \{x\}$. We then attempt to extend the transfer series based on its successor operations in the dependency graph (algorithm 8). If x has only one successor and it is a transfer operation, we can add that successor to the transfer series and extend it to the following successors. Once we have found an invertible transfer series, we remove the series from the dependency graph. Series that cannot be inverted are critical to the knitting program, and we retain them.



(a) Shift Two Needles **(b) Shift Eight Needles**

Figure 5: Two Y-shapes with the right tube shifted to the right every four courses. These KnitScript programs were optimized with KODA. No necessary transfers were removed.

6.1.1 Maintaining Slack. Notably, a critical change in state that can go unnoticed is changes that affect slack constraints. If a transfer series moves floats in order to maintain slack, the xfer series cannot be inverted, even if the loops arrive at the original destination. This is an example of why representing floats, in addition to loops, in the knitgraph structure is critical to dependency analysis.

Consider how a knitter may organize transfers to spread out two connected tubes forming a Y shape (Figure 5). When shifting the right tube to the right, a naive knitter will transfer each loop on the front side of the tube across to the opposite bed, rack to the desired shifting distance, and then back over to the newly shifted

positions. Then, they will repeat this with the back of the tube. However, if the desired shifting distance is greater than the allowed slack between the front and back loops, this will stretch the yarn too much. Instead, the shift must be completed in multiple smaller shifts (e.g., racking 1 or 2 needles at most). While this may appear to result in many redundant transfers, each transfer creates float-position dependencies with their neighbors. Because these float-position dependencies are not invertible, these critical transfers are maintained after transfer reduction.

6.2 Carriage-Pass Dependency Analysis

The time a knitting program takes to execute is defined by the number of *carriage passes* that must be completed. The carriage is an actuator that moves across the needle beds, actuating needles in a sequence of operations. To execute a knitting program in order, the carriage will move from needle to needle in one carriage pass until it reaches an operation on a needle the carriage has already passed. To complete the next operation, the carriage pass will end, and a new carriage pass will start in the opposite direction. Carriage passes are also limited to specific operation types. Knits and tucks can be performed in their own pass, but splits and xfers must be on independent passes. The state of the machine, such as the active yarn carrier or the alignment of the beds (i.e., their racking), can only be changed between carriage passes. For example, the program in Figure 1a is executed in seven carriage passes: $\{t_1, t_2\}$, $\{k_3\}$, $\{x_5\}$, $\{k_7\}$, $\{x_9\}$, $\{x_{11}, x_{12}\}$, and $\{k_{13}, k_{14}\}$.

Carriage passes are critical to knitting time because knitting a set of operations all in one carriage passes is substantially faster than knitting them in individual carriage passes. Thus, we define the execution time of a knitting program by the number of carriage passes executed. This can vary even with a constant operation set.

Given a knitting operation, o, the carriage pass c can receive o if it meets three criteria. First, the racking of o must match the racking of c (i.e., the needles share the same alignment). While interpreting a program into a dependency graph, we label each operation with the racking at which it was executed. Second, we consider if the type of operation is compatible with c. Knit and tuck operations can be executed in the same carriage pass. However, splits can only be executed in carriage passes with other splits, and transfers can only be executed in carriage passes with other transfers. Third, o must continue the order of c without changing directions. Knit, tuck, and split operations must be knit in a specified direction. If o has a specified direction (i.e., + or -), c must operate in the same direction. We can only add o to c if the needle o operates on comes after the needle operated on by the last operation in c in that direction. Transfer passes can freely re-order transfers, so the direction does not matter. By convention, we execute a transfer pass by sorting from the leftmost needle to the rightmost needle.

6.2.1 Carriage-Pass Formation. To reduce the number of carriage passes produced by a knitting program, we analyze the dependencies in the dependency graph and form a new Carriage-Pass dependency graph that reduces the set of topological orderings of the program. By clustering operations into carriage passes, we reduce the space of the program's topological orderings and are likely to find a more efficient solution.

A Carriage-Pass dependency graph is similar to our program dependency graph in that it represents the dependencies between operations in a program. However, instead of nodes representing individual operations, they represent carriage passes, which are a collection of operations that can be executed in a specific sequence. Thus, we define a carriage pass dependency graph $G(\mathbb{C}, \mathbb{D})$ by a set of carriage passes $\mathbb{C}$ and the directed dependency edges between them, $\mathbb{D}$. An edge $d_{u,v} \in \mathbb{D}$ indicates that there is at least one dependency between an operation in the carriage pass u and an operation in the carriage pass v.

Input $G_P(\mathbb{P}, \mathbb{D}_P)$**:** The dependency graph produced by execution of the program $\mathbb{P}$.
Output $G_C(\mathbb{B}, \mathbb{D}_C)$**:** The dependency graph between carriage-passes that cluster the operations in $\mathbb{P}$.
$\mathbb{B}_{open} \leftarrow \emptyset$;
for $o \in \mathbb{P}$ **do**
  $o_{descendants} \leftarrow$ Descendent operations of o in G_P;
  $c_o \leftarrow \emptyset$;
  for $c \in \mathbb{B}_{open}$ **do**
    if *c can receive o and* $c \cap o_{descendants} = \emptyset$ **then**
      add o to the end of c;
      $c_o \leftarrow c$;
      Break;
    end
  end
  if $c_o = \emptyset$ **then**
    $c_o \leftarrow \{o\}$;
    add c_o to $\mathbb{B}_{open}$;
  end
  for $\hat{o} \in \mathbb{P} | \exists d_{\hat{o},o} \in \mathbb{D}_P$ **do**
    $\hat{c} \leftarrow \hat{c} \in \mathbb{B} | \hat{o} \in \hat{c}$;
    add $d_{\hat{c},c}$ to $\mathbb{D}_C$;
    if $d_{\hat{c},c}$ *is a yarn-order dependency* **then**
      remove $\hat{c}$ from $\mathbb{B}_{open}$;
    end
  end
end
return $G_C(\mathbb{B}, \mathbb{D}_C)$;

Algorithm 9: Carriage Pass Dependency Analysis

We start forming a carriage-pass dependency graph by creating an empty set of open carriage passes. This set holds all carriage passes that can receive new operations. We then iterate through the operations in the original program, searching for an open carriage pass to receive them. For an operation o, we greedily search for an open-carriage pass, c, that can be extended by o without creating a cycle in the carriage pass dependency graph. A cycle would be formed if there is a path from the operation o to any operation in the carriage pass c in the program dependency graph. Note that the graph must remain acyclic to find a topological ordering of carriage passes. If no carriage pass can receive o, then we will create a new open carriage pass. After o is added to a carriage pass, we update the dependencies. For each predecessor operation on which o depends, we find its carriage pass. If that predecessor is not

in o's carriage pass, we add a dependency between the predecessor and o's carriage passes.

This process creates an increasingly large set of carriage passes. In the worst case, this is equal to the number of operations in the program. All transfer passes remain open and can receive new operations. However, a carriage pass involving a yarn (knits, tucks, splits) closes when a new carriage pass is formed with a yarn-order dependency between them.

6.2.2 Managing Machine State. An experienced knit programmer may be wondering where in the dependency graph we introduce all of the knitting operations that do not operate on needles or directly modify the knitgraph. For example, the rack operations required to change the alignment of needle beds before transferring or the carrier managing operations (e.g., inhook, outhook, releasehook) that introduce and cut the yarns. After we clustered operations into carriage passes, we re-introduced these types of non-knitting instructions to the carriage-pass dependency graph. Essentially, we treat the pause between carriage passes where these operations are executed as another carriage-pass dependency that will influence the topological ordering of the final optimized program.

Each carriage pass is dependent on a specific state of the knitting machine. Each operation in a carriage pass must be executed at the same racking. So, we introduce a new racking operation that each carriage pass depends on. Next, all operations that involve yarns (i.e., knit, tuck, split) require the yarn to be activated by an *inhook* operation. For each yarn, we create an inhook operation with the carrier that holds that yarn and add a dependency to the carriage pass that holds the first operation to form a loop with the yarn (i.e., the yarn's head). Inhook operations activate the *yarn-inserting-hook* that holds the head of the yarn until it has been stabilized by forming multiple loops. This hook blocks needles near the first loop formed on the yarn and prevents split and transfer passes. Thus, when we identify a transfer pass or a pass that contains one of these blocked needles, we add a dependency to a releasehook operation that releases the hook and unlocks the needles. Finally, the yarns in a knitgraph must be cut free from the machine after the last loop of each yarn is formed. Thus, if a carriage pass is followed by no other carriage passes with yarn-order dependencies, we can introduce a new dependency to an *outhook* operation. The outhook operation requires the yarn-inserting-hook. So, if no releasehook was already introduced by blocked needles, we introduce a new releasehook operation and make the outhook dependent on it.

The resulting carriage-pass dependency graph describes a reduced space of knitting programs that will produce the original knitted structure but may more efficiently use carriage passes to cluster operations. We find an optimized program in this dependency graph using a topological sort. We implement our graph structures using the Network X Graph library in Python and use their topological sorting method [14, 29]. We take a final pass on this resulting program to remove redundant rack operations. Finally, we identify each releasehook operation and attempt to shift it earlier in the final execution order. A releasehook may not be required until it blocks a needle, but ideally, it is released after a carriage pass that forms enough loops to hold the yarn stable in the knitgraph (e.g., ten loops) and before the next carriage pass that moves in the same direction as the carriage pass that used the yarn after an inhook operation.

The final program is clear of unnecessary operations, ordered to efficiently use carriage passes, handles carriers and racking, and will produce the same knitgraph as the original program. In the following section, we describe a set of demonstrative knitting programs that we use to evaluate the efficacy of the KODA method.

7 DEMONSTRATION

Many knitted structures are created by pairing stitch operations with a series of transfers that reposition the resulting loop for the next stitch. The simplest example, as described by Lin et al. [25], is forming a knit stitch (on the front bed) and a purl stitch (on the back bed). To switch between knits and purls, a loop must first be knit and then transferred to the opposite bed. More complex structures, such as decreases and cables, are formed by transferring loops to the opposite bed and then transferring them to the original bed at a new needle. When hand-knitting, these stitch and transfer operations are handled one at a time. However, the knits and transfers should be clustered in carriage passes to machine knit these efficiently.

In the following set of demonstrations, we describe how programmers may write programs based on this hand-knitting intuition while producing inefficient results. In the following section, we will evaluate KODA by optimizing the programs produced in these demonstrations for semantic correctness, fabrication correctness, manufacturing efficiency, and computational complexity.

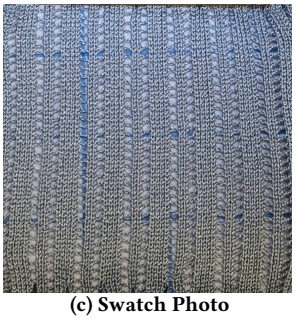
7.1 Lace and Decreases

```
knit + f1 y; Pass 1
xfer f2 b2; Pass 2
rack 1;
xfer b2 f3; Pass 3
rack 0;
tuck + f2 y; Pass 4
knit + f3 y; Pass 4
xfer f5 b5; Pass 5
rack -1;
xfer b5 f4; Pass 6
rack 0;
knit + f4 y; Pass 7
tuck + f5 y; Pass 7
```

(a) Intuitive Method

```
xfer f2 b2; Pass 1
xfer f5 b5; Pass 1
rack 1;
xfer b2 f3;  Pass 2
rack -1;
xfer b5 f4; Pass 3
rack 0;
knit + f1 y; Pass 4
tuck + f2 y; Pass 4
knit + f3 y; Pass 4
knit + f4 y; Pass 4
tuck + f5 y; Pass 4
```

(b) Efficient Method



(c) Swatch Photo

Figure 6: Example code of a method forming two paired decreases using an approach similar to hand knitting and an efficient method based on the slider algorithm [26].

Lace patterns are made by pairing decreases and increases to create holes in the fabric. This introduces complex wale structures where new wales start in the middle of knitting and merge at decreases. Figure 6c shows an example lace pattern of alternating left and right decreases paired with increases that form eyelets. Decreases are formed by moving a loop onto a neighboring loop and then knitting through the stacked loops. Hand knitters will complete these transfers and knits all at once and may intuitively program them in this way (Figure 6a). However, to efficiently knit multiple decreases in a single course of knitting–as in lace patterns–the transfers needed to stack loops should be consolidated into one pass, followed by a knitting pace to complete all the decreases (Figure 6b).

As described in Lin et al's [26] schoolbus method, an optimal approach to stacking loops into decreases is to consolidate transfers across to the opposite bed and consolidate stacking transfers based on the racking. For example, leftward (r=1) and rightward (r=-1) transfers require two separate transfer passes for stacking—totaling three transfer passes. Decreases across multiple racking magnitudes will require a minimum of one transfer pass for each racking and the additional holding transfer pass.

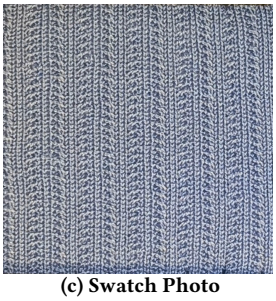
7.2 Cables and Wale-Crossings

```
xfer f2 b2; Pass 2
xfer f3 b3; Pass 2
rack 1;
xfer b2 f3; Pass 3
rack -1;
xfer b3 f2; Pass 4
rack 0;
knit + f2 y; Pass 5
knit + f3 y; Pass 5
xfer f4 b4;
xfer f5 b5; Pass 6
rack -1;
xfer b5 f4; Pass 7
rack 1;
xfer b4 f5; Pass 8
rack 0;
knit + f4 y; Pass 9
knit + f5 y; Pass 9
```

(a) Intuitive Method

```
xfer f2 b2; Pass 1
xfer f3 b3; Pass 1
xfer f4 b4; Pass 1
xfer f5 b5; Pass 1
rack 1;
xfer b2 f3;  Pass 2
rack -1;
xfer b3 f1; Pass 3
xfer b5 f4; Pass 3
rack 1;
xfer b4 f5; Pass 4
rack 0;
knit + f2 y; Pass 5
knit + f3 y; Pass 5
knit + f4 y; Pass 5
knit + f5 y; Pass 5
```

(b) Efficient Method

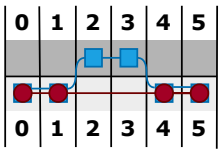


(c) Swatch Photo

Figure 7: Example code of a method forming two alternating cables using an approach similar to hand knitting and an efficient method that consolidates transfers.

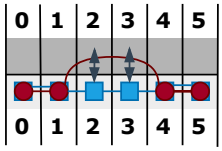
Cables are similar to lace patterns in that they introduce complex wale structures. Cables braid wales by crossing them at specific locations. Figure 7c shows an example cable pattern of alternating leftward and rightward cables crossing two loops over a third loop. Like decreases, hand knitters will form cables by crossing their loops and then immediately knitting across them (Figure 7a). However, the transfers that rearrange these loops should be consolidated when knitting multiple cables in a single course (Figure 7b).

A right-leaning cable is formed in three carriage passes: transferring a set of loops to the opposite bed, then transferring the left half of the loops to the freed needles on the right side of the cable, then transferring the right loops to the left side. Switching the order of left and right transfers will switch this to a left-leaning cable. Like forming decreases, cables can be inefficiently formed (cable-by-cable) or efficiently clustered into carriage passes. In the optimal case, if there are both left and right-leaning cables in a course, this will require five carriage passes.

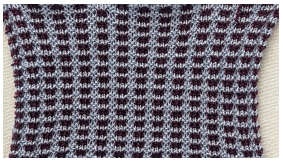


(a) Float without Transfers

(c) Result without Transfers



(b) Float with Transfers



(d) Result with Transfers

Figure 8: Demonstration of the effect redundant transfers have on the float positions of yarns.

7.3 Colorwork and Float Arrangement

While lace and cable patterns introduce complexity in the wales of a knitted structure, colorwork introduces complexity in float placement. Colorwork is created using multiple yarns. The floats between each yarn must be carefully placed relative to the loops formed on the other yarn. A common hand-knitting colorwork technique called Fair-Isle is done by knitting needles with different color yarns to form intricate, detailed color patterns. While knitting, the hand knitter will pull all of the yarns through the pattern—meticulously handling the placement of floats. To do this on a knitting machine, a naive programmer may move the carriers in tandem, transferring loops to cover floats as needed (Figure 9a).

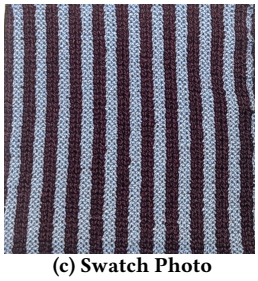
Consider a rib pattern made of alternating columns of knit stitches and purl stitches. Adding colorwork to these patterns can be done by knitting with a second yarn only on the columns of knit stitches. However, as we show in Figure 8, the placement of the pulled stitches will alter the float placement of the fabric. Figure 9c shows an example of the colorwork rib pattern where the red yarn is only knit on front bed stitches and floats under the blue purl stitches. To do this correctly, the purled stitches must first be transferred to the front bed. Then, the colored stitches can be knit on the knit wales. After, the purled stitches can be transferred back to the

```
xfer f2 b2; Pass 1
knit + b2 blue; Pass 2
xfer b2 f2; Pass 3
xfer f4 b4; Pass 3
knit + b4 blue; Pass 4
xfer b4 f4; Pass 5
knit + f1 red; Pass 6
knit + f3 red; Pass 6
knit + f5 red; Pass 6
```

(a) Intuitive Method

```
xfer f2 b2; Pass 1
xfer f4 b4; Pass 1
knit + b2 blue; Pass 2
knit + b4 blue; Pass 2
xfer b2 f2; Pass 3
xfer b4 f4; Pass 3
knit + f1 red; Pass 4
knit + f3 red; Pass 4
knit + f5 red; Pass 4
```

(b) Efficient Method



(c) Swatch Photo

Figure 9: Example code that forms alternating columns of blue purls and red knits using an approach similar to hand knitting and an efficient method that consolidates transfers.

back bed in preparation for continuing to knit the rib pattern. These transfers can be inefficiently executed stitch-by-stitch or optimally clustered into knitting and transfer passes (Figure 9b).

8 OPTIMIZATION BENCHMARKS

For our optimization method to be effective, it must meet three criteria. First, it must be semantically correct; that is, the resulting knitted structure should be identical to the original program's. Second, the optimized program should fabricate correctly without dropping stitches or tearing yarns. Third, the optimizer should reduce or maintain execution time measured by carriage passes.

We have produced a set of benchmark swatch programs based on our demonstration swatches to evaluate these criteria. Each program is written in the higher-level knitting language, KnitScript [15, 17][1]. The width in stitches and height in the courses of the swatch are parameterized. We provide three benchmark categories: lace, cable, and colorwork. For each category, we have developed three programs. The first is an inefficient, stitch-by-stitch approach to knitting these structures in a regular pattern. The second is an optimal program that generates the same structure. The third is an inefficient program that generates the swatch stitch-by-stitch but randomizes the placement of these structures (see Figure 10).

Additionally, we include a randomized program that uses all techniques (i.e., lace, cables, and colorwork) in one pattern (see Figure 10d). An experienced knit programmer can define an optimal program for structure by recognizing the opportunity to cluster transfer operations into carriage passes. However, combining these methods increases the complexity of transfer planning substantially. This demonstrates the benefit of program optimizers—they allow programmers to write intuitive solutions rather than efficient code.

[1]KnitScript code to generate these benchmarks is available in the KODA code base at https://github.com/mhofmann-Khoury/koda-knitout

Thus, we provide a final benchmark program that randomly selects a structure to apply to each cluster of six stitches.

9 EVALUATION

We generated swatches using both the inefficient and ground-truth benchmarks to validate our optimizer. We then optimized each inefficient swatch program. We make three comparisons between swatches to verify semantic correctness, correct fabrication, and carriage-pass efficiency. We verify semantic correctness by comparing the knitgraph of the un-optimized program and the optimized program. Any variation in the knitgraphs would result in different knitted structures and violate the requirements of our optimizer. Samples are correctly fabricated if the ground-truth program and optimized swatches are knit without torn yarns or dropped loops and appear identical. The inefficient swatches could not be reliably knit despite multiple attempts. We measure the efficiency of each program in carriage passes counted in the Apex 4 software [45].

We generated two data sets of swatch programs from our benchmarks that show the effect of optimization on program efficiency across varied swatch sizes and randomized patterns.

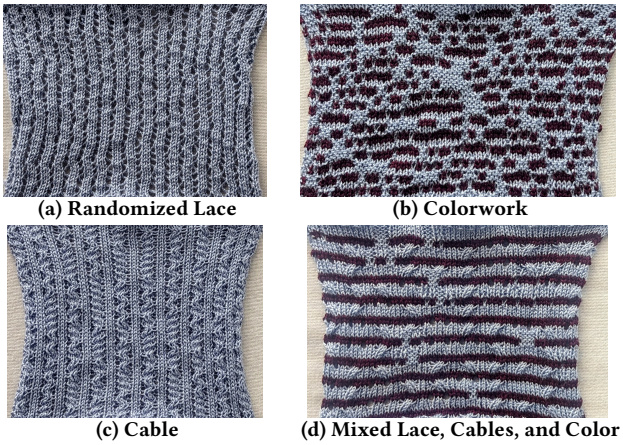
9.1 Standardized Swatch Generation by Size

For the first data set, we generated the standardized lace, cable, and colorwork patterns in a range of sizes. Each swatch is knit to be a square with the same number of stitches and courses. We increment the size parameter by six stitches, ensuring that there is sufficient space for added repetitions of the lace, cable, and color work patterns. Figure 11 shows the effect of swatch size on carriage pass count. We see that size causes an exponential increase in the number of carriage passes for the unoptimized programs. This is because as the size increases, the number of lace, cable, and color work structures increases exponentially. When made stitch by stitch, this causes an exponential increase in transfer passes. Alternately, the optimized swatch programs increase their carriage passes linearly. This is because the optimal solution will cluster transfers across stitch structures and the number of carriage passes only increases relative to the number of courses in the swatch.

9.2 Randomized Swatch Generation

Our next data set uses the randomized swatch benchmarks. We fixed the size of the swatches to 82 stitches by 82 courses. When optimized, most swatches of this size can be knit in under 5 minutes and allow for a wide variation of knitted structures. We knit ten unique randomized swatches for each randomized benchmark (i.e., lace, cable, colorwork, mixed patterns). Table 2 summarizes the mean and standard deviation of carriage pass counts across each of these benchmarks in the unoptimized and optimized case.

Note that the resulting number of carriage passes was fixed after optimization for the lace and cable work patterns. For these benchmarks, the optimizer successfully clustered the transfers required between courses regardless of the position of decreases or cables in each course. This results from the greedy carriage pass consolidation approach. When there are multiple carriage passes for the multiple yarns in a course, the optimal transfer passes are sometimes split across each yarn's carriage pass. This results in a



(a) Randomized Lace (b) Colorwork

(c) Cable (d) Mixed Lace, Cables, and Color

Figure 10: Example randomized swatches.

sub-optimal program; however, there are still significant reductions in carriage passes overall.

9.3 Semantic Correctness

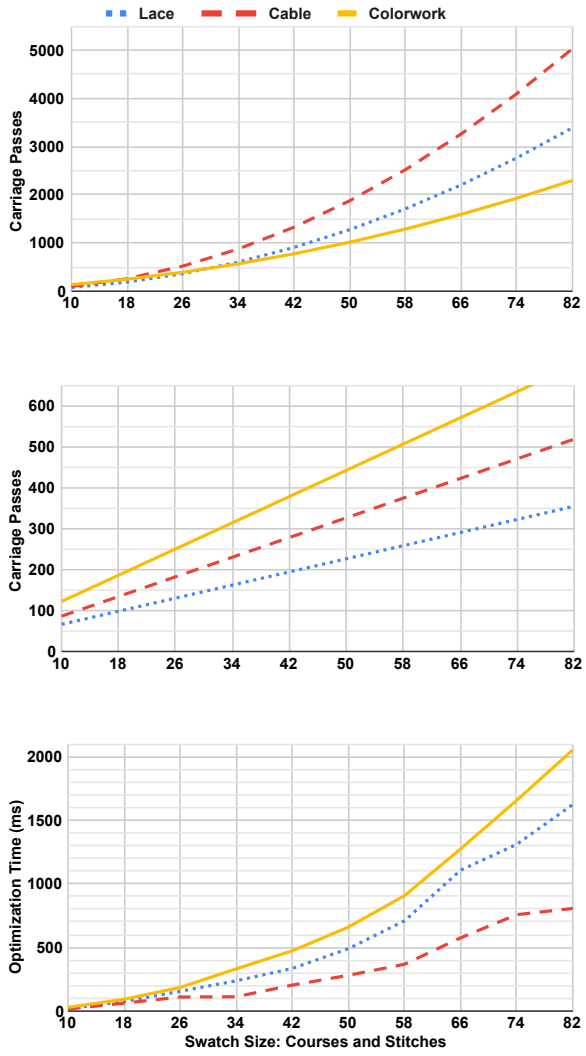
For each pair of inefficient and optimized swatch programs, we compared the resulting knitgraph produced by executing the program on our virtual knitting machine. Across all pairs in both data sets, the resulting knitgraphs always matched exactly. This shows that the optimizer produced semantically correct results.

9.4 Fabrication Correctness

We attempted to knit the inefficient programs for each standardized benchmark with a size of 82 stitches by 82 courses. However, despite multiple attempts and adjustments of machine parameters such as loop length and knitting speed, we were unsuccessful at producing swatches without numerous yarn tears and dropped stitches. This is despite the fact that the knitted structures are valid, the instructions violate no machine constraints (e.g., slack constraints), and the machine-specific software raised no warnings. This demonstrates that program efficiency can be critical to reliable fabrication, not just a measure of manufacturing time. Instead of comparing the fabricated results of our optimized swatches to the inefficient programs, we compared them to the fabricated results of the ground-truth programs for standardized lace, cable, and colorwork.

We knitted each pair of optimized and ground-truth swatches with 82 stitches by 82 courses. We knit these swatches on a Shima Seiki SWG91N2 15 gauge v-bed knitting machine using Puma Stretch 2/28 NM yarns [10], a carriage speed of .8 m/s, and a stitch size of 40. All samples were knitted without error, and we could not identify any variation between the optimized and ground-truth swatches. We knit three random samples for each benchmark in our randomized data set under the same conditions. Because we have no ground truth for these samples and the inefficient solutions were not knittable, we cannot verify the fabrication correctness of these samples. However, the resulting swatches were consistent in appearance and knitted without error. No loops appear dropped, no yarns are torn, and no floats are misplaced.

9.5 Efficiency After Optimization



(c) Time to Optimize Program in milliseconds.

Figure 11: Effect of swatch size in stitches and courses on carriage pass counts across standardized pattern benchmarks.

The substantial reduction in carriage passes across all generated swatches shows that the optimizer can consistently improve the efficiency of knitting programs by clustering operations in carriage passes. For the standardized benchmark patterns, the number of carriage passes after optimization was often equal to the number of carriage passes produced by the optimal ground-truth programs. This does not necessarily imply that the optimizer will always produce an optimal program. The tendency to find an optimal program may depend on the types of structures being knit. For example, we saw consistent results when optimizing lace and cable patterns but variance when knitting with colorwork. When multiple techniques are used in one program, the optimizer performs the most poorly. However, most programs reduced the carriage pass counts by half in these cases.

Table 2: Summary of optimization statistics across 10 randomized swatch programs for each pattern type.

Pattern Type	Unoptimized Carriage Passes		Optimized Carriage Passes		Percentage of Original Program		Optimization Time (ms)	
	Mean	StD	Mean	StD	Mean	StD	Mean	StD
Lace	1748.0	0.0	353.8	0.6	20.24	0.04	1172.3	67.3
Cable	2568.0	0.0	534.0	0.0	20.79	0.00	2340.3	349.2
Colorwork	915.6	28.7	605.2	14.2	66.12	1.31	2258.7	165.7
Mixed	1994.4	41.7	1009.6	24.0	50.64	1.71	3087.8	317.9

9.6 Computation Time: Scalability and Usability

Beyond the manufacturing consequences of inefficient knitting, the principal benefit of program optimization is manufacturing time. To be usable and useful to programmers, this benefit must be weighed against the computing time needed to optimize a program. As shown in Figure 11c and Table 2, across all of our benchmark tests, the optimizer never took longer than four seconds to optimize a knitting program. The computational complexity of our dependency analysis method is dependent on the number of operations in a program (i.e., $|\mathbb{P}|$) and the number of dependency edges between them (i.e., $|\mathbb{D}|$)—$O(|\mathbb{P}| \times |\mathbb{D}|)$. This is shown in the exponential trend in optimization times shown in Figure 11c; as the number of stitches and courses–and thus operations–increases, the optimization time increases. The computational complexity increases faster for patterns with more dependencies, such as the float dependencies in the colorwork benchmarks. Regardless, these computation times are still insignificant compared to the gained speed in knitting–a mechanical and physical process, where carriage passes happen on an order of seconds and a reduction of carriage passes by even half can save minutes on small samples and hours on full garments.

10 DISCUSSION AND LIMITATIONS

Knit program optimization is only a small part of the knit programming infrastructure needed to hasten the creation of novel knitting CAD tools. This work exists alongside advances in domain-specific languages [17, 31], other approaches to transfer planning [24–26], and the creation of new knit structures that are not encapsulated by knitgraphs. However, we aim for KODA to be a valuable tool in this pipeline that enables knit programmers to focus on creating novel structures rather than writing efficient programs. This may enable new knit programmers who are not deeply familiar with the idiosyncrasies of knitting machines to build efficient and reliable programs. It may also enable researchers to explore knitting algorithms where an efficient solution is unknown or too complex to create in a readable knit program.

We identify three key limitations of the KODA system. First, KODA does not necessarily produce an optimal program. Second, our method does not encapsulate the placement of floats while they are on the carriers (e.g., miss instructions), which may lead to inefficient kickbacks. Third, KODA requires an original knit program and does not handle *machine scheduling* where a knitgraph is converted into a set of instructions.

10.1 KODA and Optimum Knitting

Finding an optimal order of knitting operations for each knit program would be ideal. KODA will produce this optimal set of operations in many cases (e.g., lace and cable benchmarks). However, we cannot guarantee that it will produce the optimal results. When clustering operations into carriage passes (see algorithm 9), we make a greedy decision to merge an operation into the first open carriage pass that we discover. This will only produce one carriage pass for operations involving a yarn. However, there may be multiple open transfer passes that can accept the operation for transfers. Unfortunately, which transfer pass would produce the most efficient program cannot be determined at this stage, and the system would need to build multiple carriage pass dependencies based on all possible transfer passes that could receive the transfer. Across all of these carriage pass dependency graphs, we could identify the most efficient one because it has the fewest final carriage passes. An efficient way of identifying these could involve methods from general-purpose programming optimization such as e-Graphs (e.g., [52]). This remains an area for future work.

Transfer Dependency reduction is one way to reduce the number of open transfer passes during carriage pass consolidation. When there are many redundant transfers, this may increase the efficacy of our greedy method. However, we find that the benefits of transfer reduction are often limited since an entire carriage pass of transfers must be reduced to have an effect on the final manufacturing time of a knitting program. Unlike carriage pass consolidation, which can readily undo common knitting approaches used by novice programmers (e.g., transferring decreases immediately before knitting), transfer reduction is not directly tied to common knit programming approaches. While not represented in our benchmarks, we expect that transfer reduction is more beneficial when connecting multiple transfer algorithms together. For example, Hofmann et al's [17] sheet-layering algorithm, embedded in the KnitScript language [15], introduces transfer passes between knitting operations to maintain a programmer-defined state of floats crossings. However, it is common to immediately undo these transfers when knitting patterns that use transfer for purling, lace, and cables. Transfer reduction can undo these redundant operations created by using two separate coding techniques in one knitting program, allowing programmers to use these tools more intuitively.

10.2 Modeling Loose Yarns

Our expanded knitgraph representation ignores the state of the yarn until it has been fixed into the knit structure by forming a loop. However, knit programmers often use miss operations to position a yarn relative to the knit structure for more precise control over floats being formed. Purposeful tangling and detangling of yarns can be done using these operations that are lost during optimization. Notably, this tangling is only done between active carriers, and the final placement of floats in the knitgraph cannot be changed. Future

work should incorporate the yarn-carrier dependencies in addition to the dependencies in the knitgraph.

10.3 Beyond Scheduled Knitting

Optimizing a given knit program (i.e., schedule) is feasible with KODA because we use the original order of operations to form a knitgraph and dependency graph. However, converting a given knitgraph (e.g., one produced by a design tool) into an optimal program without this existing schedule requires more advanced methods for knit-scheduling. Prior work has explored this for some limited types of knit structures (e.g., 3D surfaces [37], textures [16]), but capturing the full space of knitted structures in a single knitting algorithm may not be feasible. Indeed, we did not benchmark KODA against the outputs of existing tools because, to make manufacturing feasible, a great deal of effort has been spent to make the output of these programs efficient. This highlights the value of a system like KODA. Program optimization allows researchers to search for flexible machine knitting algorithms that will serve as the basis for knitting CAD tools. KODA frees programmers from hardware-specific constraints and efficient knitting practices.

11 CONCLUSION

We present KODA (Knit program Optimization by Dependency Analysis). KODA is a knit-program optimization method that can increase the efficiency of a knit program written in the low-level knitout language while maintaining the original knitted structure. KODA relies on an expanded knitgraph representation that encapsulates the complex relationships between loops, stitches, wales, and floats in a knitted structure. We can model the resulting knitgraph and the dependencies between operations that formed it through a single pass of a knitting program. By analyzing the dependencies between transfer operations in the graph, we can identify redundant operations that can be removed. By merging operations into dependent carriage passes, we can organize the knit program into more efficient sets of operations, and from this, we can find an optimized instruction order. We provide four sets of benchmark swatch programs that generate complex randomized structures and conduct an evaluation that shows that KODA can reduce inefficient programs into efficient solutions that are reliable for machine knitting.

ACKNOWLEDGMENTS

This work was funded by NSF Grants 2327137 and 2341880. This work could not have been completed without the endless patience of students who used KnitScript and the earliest versions of KODA in their coursework and research.

REFERENCES

[1] Roland Aigner, Mira Alida Haberfellner, and Michael Haller. 2022. SpaceR: Knitting Ready-Made, Tactile, and Highly Responsive Spacer-Fabric Force Sensors for Continuous Input. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 68, 15 pages. https://doi.org/10.1145/3526113.3545694

[2] Lea Albaugh, Scott Hudson, and Lining Yao. 2019. Digital Fabrication of Soft Actuated Objects by Machine Knitting. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300414

[3] Lea Albaugh, Scott E Hudson, and Lining Yao. 2023. Physically Situated Tools for Exploring a Grain Space in Computational Machine Knitting. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 736, 14 pages. https://doi.org/10.1145/3544548.3581434

[4] Lea Albaugh, James McCann, Scott E. Hudson, and Lining Yao. 2021. Engineering Multifunctional Spacer Fabrics Through Machine Knitting. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 498, 12 pages. https://doi.org/10.1145/3411764.3445564

[5] Todd M. Austin and Gurindar S. Sohi. 1992. Dynamic dependency analysis of ordinary programs. *SIGARCH Comput. Archit. News* 20, 2 (apr 1992), 342–351. https://doi.org/10.1145/146628.140395

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[7] Tiago M. Fernández-Caramés and Paula Fraga-Lamas. 2018. Towards The Internet of Smart Clothing: A Review on IoT Wearables and Garments for Creating Intelligent Connected E-Textiles. *Electronics* 7, 12 (2018), 405. https://doi.org/10.3390/electronics7120405

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[9] Ada Ferri, Maria Rosaria Plutino, and Giuseppe Rosace. 2019. Recent trends in smart textiles: Wearable sensors and drug release systems. *AIP Conference Proceedings* 2145, 1 (08 2019), 020014. https://doi.org/10.1063/1.5123575 arXiv:https://pubs.aip.org/aip/acp/article-pdf/doi/10.1063/1.5123575/14193705/020014_1_online.pdf

[10] Silk City Fibers. 2024. Puma Stretch Cone Yarn, NM 2/28. https://www.silkcityfibers.com/products/puma-stretch-cone-yarn [Online; accessed 1. Apr. 2024].

[11] Jack Forman, Ozgun Kilic Afsar, Sarah Nicita, Rosalie Hsin-Ju Lin, Liu Yang, Megan Hofmann, Akshay Kothakonda, Zachary Gordon, Cedric Honnet, Kristen Dorsey, Neil Gershenfeld, and Hiroshi Ishii. 2023. FibeRobo: Fabricating 4D Fiber Interfaces by Continuous Drawing of Temperature Tunable Liquid Crystal Elastomers. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 19, 17 pages. https://doi.org/10.1145/3586183.3606732

[12] Gozde Goncu-Berk, Burak Karacan, and Ilke Balkis. 2022. Embedding 3D Printed Filaments with Knitted Textiles: Investigation of Bonding Parameters. *Clothing and Textiles Research Journal* 40, 3 (2022), 171–186. https://doi.org/10.1177/0887302X20982927 arXiv:https://doi.org/10.1177/0887302X20982927

[13] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is Static Analysis Able to Identify Unnecessary Source Code? *ACM Trans. Softw. Eng. Methodol.* 29, 1, Article 6 (jan 2020), 23 pages. https://doi.org/10.1145/3368267

[14] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. 2008. Exploring network structure, dynamics, and function using NetworkX. https://www.osti.gov/biblio/960616

[15] Megan Hofmann. 2023. knit-script. https://pypi.org/project/knit-script [Online; accessed 1. Jul. 2023].

[16] Megan Hofmann, Lea Albaugh, Ticha Sethapakadi, Jessica Hodgins, Scott E. Hudson, James McCann, and Jennifer Mankoff. 2019. KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/3332165.3347886

[17] Megan Hofmann, Lea Albaugh, Tongyan Wang, Jennifer Mankoff, and Scott E Hudson. 2023. KnitScript: A Domain-Specific Scripting Language for Advanced Machine Knitting. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 21, 21 pages. https://doi.org/10.1145/3586183.3606789

[18] Benjamin Jones, Yuxuan Mei, Haisen Zhao, Taylor Gotfrid, Jennifer Mankoff, and Adriana Schulz. 2021. Computational Design of Knit Templates. *ACM Trans. Graph.* 41, 2, Article 16 (dec 2021), 16 pages. https://doi.org/10.1145/3488006

[19] Ken Kennedy and John R. Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Jin Hee (Heather) Kim, Kunpeng Huang, Simone White, Melissa Conroy, and Cindy Hsin-Liu Kao. 2021. KnitDermis: Fabricating Tactile On-Body Interfaces Through Machine Knitting. In *Designing Interactive Systems Conference 2021* (Virtual Event, USA) *(DIS '21)*. Association for Computing Machinery, New York, NY, USA, 1183–1200. https://doi.org/10.1145/3461778.3462007

[21] Jin Hee (Heather) Kim, Shreyas Dilip Patil, Sarina Matson, Melissa Conroy, and Cindy Hsin-Liu Kao. 2022. KnitSkin: Machine-Knitted Scaled Skin for Locomotion. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 391, 15 pages. https://doi.org/10.1145/3491102.3502142

[22] Jin Hee (Heather) Kim, Joan Stilling, Michael O'Dell, and Cindy Hsin-Liu Kao. 2023. KnitDema: Robotic Textile as Personalized Edema Mobilization Device. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 472, 19 pages. https://doi.org/10.1145/3544548.3581343

[23] Mackenzie Leake, Gilbert Bernstein, Abe Davis, and Maneesh Agrawala. 2021. A mathematical foundation for foundation paper pieceable quilts. *ACM Trans. Graph.* 40, 4, Article 65 (jul 2021), 14 pages. https://doi.org/10.1145/3450626.3459853

[24] Jenny Lin and James McCann. 2021. An Artin Braid Group Representation of Knitting Machine State with Applications to Validation and Optimization of Fabrication Plans. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1147–1153. https://doi.org/10.1109/ICRA48506.2021.9562113

[25] Jenny Lin, Vidya Narayanan, Yuka Ikarashi, Jonathan Ragan-Kelley, Gilbert Bernstein, and James Mccann. 2023. Semantics and Scheduling for Machine Knitting Compilers. *ACM Trans. Graph.* 42, 4, Article 143 (jul 2023), 26 pages. https://doi.org/10.1145/3592449

[26] Jenny Lin, Vidya Narayanan, and James McCann. 2018. Efficient Transfer Planning for Flat Knitting. In *Proceedings of the 2nd Annual ACM Symposium on Computational Fabrication* (Cambridge, Massachusetts) *(SCF '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3213512.3213515

[27] Yiyue Luo, Kui Wu, Tomás Palacios, and Wojciech Matusik. 2021. KnitUI: Fabricating Interactive and Sensing Textiles with Machine Knitting. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 668, 12 pages. https://doi.org/10.1145/3411764.3445780

[28] Yiyue Luo, Kui Wu, Andrew Spielberg, Michael Foshey, Daniela Rus, Tomás Palacios, and Wojciech Matusik. 2022. Digital Fabrication of Pneumatic Actuators with Integrated Sensing by Machine Knitting. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 175, 13 pages. https://doi.org/10.1145/3491102.3517577

[29] Udi Manber. 1989. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., United States. https://doi.org/10.5555/534662

[30] James McCann. 2020. The "Knitout" (.k) File Format. https://textiles-lab.github.io/knitout/knitout.html [Online; accessed 24. Feb. 2023].

[31] James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. 2016. A Compiler for 3D Machine Knitting. *ACM Trans. Graph.* 35, 4, Article 49 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925940

[32] James McCann, Vidya Narayanan, and Gabrielle Ohlson. 2017. Autoknit: A Public-Domain Reimplementation of "Automatic Machine Knitting of 3D Meshes". https://github.com/textiles-lab/autoknit.

[33] M. McKnight, T. Agcayazi, H. Kausche, T. Ghosh, and A. Bozkurt. 2016. Sensing textile seam-line for wearable multimodal physiological monitoring. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, New York, NY, USA, 311–314. https://doi.org/10.1109/EMBC.2016.7590702

[34] Rahul Mitra, Erick Jimenez Berumen, Megan Hofmann, and Edward Chien. 2024. Singular Foliations for Knit Graph Design. In *ACM SIGGRAPH 2024 Conference Papers* (Denver, CO, USA) *(SIGGRAPH '24)*. Association for Computing Machinery, New York, NY, USA, Article 38, 11 pages. https://doi.org/10.1145/3641519.3657487

[35] Rahul Mitra, Liane Makatura, Emily Whiting, and Edward Chien. 2023. Helix-Free Stripes for Knit Graph Design. In *ACM SIGGRAPH 2023 Conference Proceedings* (<conf-loc>, <city>Los Angeles</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(SIGGRAPH '23)*. Association for Computing Machinery, New York, NY, USA, Article 75, 9 pages. https://doi.org/10.1145/3588432.3591564

[36] S. Muchnick. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA. https://books.google.com/books?id=Pq7pHwG1_OkC

[37] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. 2018. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.* 37, 3, Article 35 (aug 2018), 15 pages. https://doi.org/10.1145/3186265

[38] Rita Paradiso, Laura Caldani, and Maria Pacelli. 2014. Chapter 3.1 - Knitted Electronic Textiles. In *Wearable Sensors*, Edward Sazonov and Michael R. Neuman (Eds.). Academic Press, Oxford, 153–174. https://doi.org/10.1016/B978-0-12-418662-0.00003-9

[39] Andreas Pointner, Thomas Preindl, Sara Mlakar, Roland Aigner, Mira Alida Haberfellner, and Michael Haller. 2022. Knitted Force Sensors. In *Adjunct Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22 Adjunct)*. Association for Computing Machinery, New York, NY, USA, Article 77, 3 pages. https://doi.org/10.1145/3526114.3558656

[40] Andreas Pointner, Thomas Preindl, Sara Mlakar, Roland Aigner, and Michael Haller. 2020. Knitted RESi: A Highly Flexible, Force-Sensitive Knitted Textile Based on Resistive Yarns. In *ACM SIGGRAPH 2020 Emerging Technologies* (Virtual Event, USA) *(SIGGRAPH '20)*. Association for Computing Machinery, New York, NY, USA, Article 21, 2 pages. https://doi.org/10.1145/3388534.3407292

[41] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2023. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2* (1 ed.). Association for Computing Machinery, New York, NY, USA, Article 39, 12 pages. https://doi.org/10.1145/3596711.3596751

[42] Vanessa Sanchez, Kausalya Mahadevan, Gabrielle Ohlson, Moritz A. Graule, Michelle C. Yuen, Clark B. Teeple, James C. Weaver, James McCann, Katia Bertoldi, and Robert J. Wood. 2023. 3D Knitting for Pneumatic Soft Robotics. *Advanced Functional Materials* n/a, n/a (2023), 2212541. https://doi.org/10.1002/adfm.202212541 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/adfm.202212541

[43] Vanessa Sanchez, Conor J. Walsh, and Robert J. Wood. 2021. Textile Technology for Soft Robotic and Autonomous Garments. *Advanced Functional Materials* 31, 6 (2021), 2008278. https://doi.org/10.1002/adfm.202008278 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/adfm.202008278

[44] Margaret Ellen Seehorn, Gene S-H Kim, Aashaka Desai, Megan Hofmann, and Jennifer Mankoff. 2022. Enhancing Access to High Quality Tangible Information through Machine Embroidered Tactile Graphics. In *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication* (Seattle, WA, USA) *(SCF '22)*. Association for Computing Machinery, New York, NY, USA, Article 23, 3 pages. https://doi.org/10.1145/3559400.3565586

[45] Shima Seiki. 2023. SHIMA SEIKI | Computerized Flat Knitting Machines, Design System/Software, CAD/CAM Systems. https://www.shimaseiki.com [Online; accessed 24. Feb. 2023].

[46] textiles lab. 2023. knitout-frontend-js. https://github.com/textiles-lab/knitout-frontend-js [Online; accessed 5. Apr. 2023].

[47] Tongyan Wang, Jennifer Mankoff, and Megan Hofmann. 2022. Fabricating Accessible Designs with Knitting Machines. In *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication* (Seattle, WA, USA) *(SCF '22)*. Association for Computing Machinery, New York, NY, USA, Article 25, 3 pages. https://doi.org/10.1145/3559400.3565584

[48] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry compiler. *ACM Trans. Graph.* 38, 6, Article 195 (nov 2019), 14 pages. https://doi.org/10.1145/3355089.3356518

[49] Kui Wu, Marco Tarini, Cem Yuksel, James McCann, and Xifeng Gao. 2022. Wearable 3D Machine Knitting: Automatic Generation of Shaped Knit Sheets to Cover Real-World Objects. *IEEE Transactions on Visualization and Computer Graphics* 28, 9 (2022), 3180–3192. https://doi.org/10.1109/TVCG.2021.3056101

[50] Kai Yang, Beckie Isaia, Laura J.E. Brown, and Steve Beeby. 2019. E-Textiles for Healthy Ageing. *Sensors* 19, 20 (2019), 4463. https://doi.org/10.3390/s19204463

[51] Cem Yuksel, Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2012. Stitch Meshes for Modeling Knitted Clothing with Yarn-Level Detail. *ACM Trans. Graph.* 31, 4, Article 37 (jul 2012), 12 pages. https://doi.org/10.1145/2185520.2185533

[52] Haisen Zhao, Max Willsey, Amy Zhu, Chandrakana Nandi, Zachary Tatlock, Justin Solomon, and Adriana Schulz. 2022. Co-Optimization of Design and Fabrication Plans for Carpentry. *ACM Trans. Graph.* 41, 3, Article 32 (mar 2022), 13 pages. https://doi.org/10.1145/3508499

[53] Amy Zhu, Adriana Schulz, and Zachary Tatlock. 2023. Exploring Self-Embedded Knitting Programs with Twine. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design* (Seattle, WA, USA) *(FARM 2023)*. Association for Computing Machinery, New York, NY, USA, 25–31. https://doi.org/10.1145/3609023.3609805

[54] Jingwen Zhu and Hsin-Liu (Cindy) Kao. 2022. Scaling E-Textile Production: Understanding the Challenges of Soft Wearable Production for Individual Creators. In *Proceedings of the 2022 ACM International Symposium on Wearable Computers* (Cambridge, United Kingdom) *(ISWC '22)*. Association for Computing Machinery, New York, NY, USA, 94–99. https://doi.org/10.1145/3544794.3558475