Master Thesis
Ecole Polytechnique Fédérale de Lausanne (EPFL)

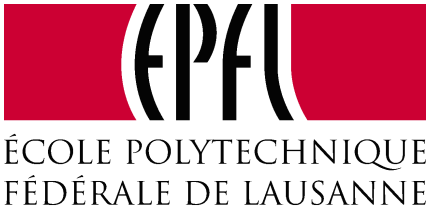
Better Loop Fusion for LMS

Vera Salvisberg
July 31, 2014

Student:
Vera Salvisberg
Bellevuestr. 18
CH-3095 Spiegel b. Bern
vera.salvisberg@epfl.ch

EPFL Supervisor:
Prof. Martin Odersky
martin.odersky@epfl.ch

Supervisor:
Prof. Oyekunle A. Olukotun
kunle@stanford.edu
Stanford University



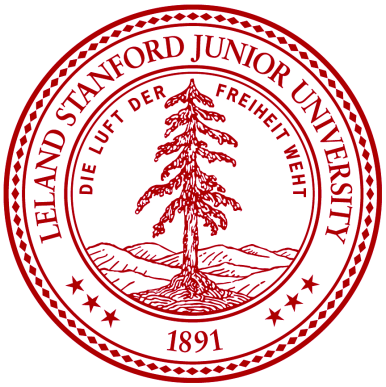


Table of Contents

1. Introduction

Domain-specific languages (DSLs) are an active area of research, because they enable domain experts to express their problems in a simple and concise manner, without having to be programming experts as well. While the advantage of having a language tailored to fit a domain is obvious as far as the productivity of the programmer using the DSL is concerned, we also need to consider the performance of the resulting system, and the cost of developing the DSL.

There are two broad categories of DSLs, stand-alone and embedded. For a stand-alone DSL, the whole toolchain needs to be implemented: the grammar, the parser, the compiler-linker or an interpreter with a virtual machine. An embedded DSL on the other hand is programmed as a library in an existing host language, and takes advantage of the existing infrastructure. While the additional layers of indirection have a performance penalty, most applications do not justify the effort of developing a stand-alone DSL.

The LMS [1] and Delite [2] frameworks developed at EPFL and Stanford University provide the best from both worlds: while the DSL can be programmed in Scala, the frameworks allow for code generation that removes the layers of indirection (LMS), and provide constructs for parallelizing code with minimal effort (Delite). Developing a DSL becomes easy through the existing libraries and semantic building blocks.

One main advantage of using a DSL framework is that general optimizations can be implemented once and are then available for all DSLs at no extra cost. The subject of my master project was to implement one such optimization from the compiler world, namely loop fusion. Loop fusion is an important optimization for all languages that feature list comprehensions and translate their high-level operations into loop-based representations. By combining a processing pipeline into a single computation intermediate datastructures can be removed, and further optimizations are enabled when side-by-side loops are fused.

LMS already had an implementation of loop fusion, but it did not cover all operations and used an old architecture that made integration with other optimizations difficult. The goal of my master project was to use the new transformer framework to implement a more comprehensive algorithm with a cleaner interface.

2. Background

Loop Fusion is an optimization that can increase performance as well as decrease memory and code size. It benefits all loop-based representations of procedures and data, and has been implemented in many compilers of functional languages with an emphasis on list processing. Loop fusion is an active area of research, related work from the Haskell world includes Stream Fusion [3] as well as Iteratees [4].

There are two different flavors of loop fusion. The simpler one is called horizontal or side-by-side fusion and fuses adjacent loops iterating over the same range. The second one is vertical or pipeline fusion, where a producer and a consumer of data are fused, removing the need for the intermediate data structure.

2.1 Horizontal Fusion

Horizontal fusion of two loops is possible under the following conditions:

H1. They are in the same scope
H2. They iterate over the same range
H3. Neither depends on the other
H4. Their effects can be interleaved

For example:

```
for (i <- 0 until 100) { pure_body() }
for (i <- 0 until 100) { effectful_body() }
```

```
for (i <- 0 until 100) {
  optimized { pure_body(); effectful_body() }
}
```

While a programmer probably wouldn't write those two loops next to each other, this pattern happens frequently in the context of embedded DSLs, where a high-level DSL program is lowered into a loop-based representation. Fusing two loops horizontally saves the cost of iteration of one loop, but the main benefit comes from further optimizations. As a consequence of the fusion, the two loop bodies are now in the same inner scope of the fused loop, and optimizations such as common subexpression elimination run on the combined body.

In terms of memory accesses, horizontal fusion can dramatically improve performance, in particular if both loops read the same data. In theory it could also cause a performance degradation if the combined body accessed data in a way that didn't trigger prefetching anymore or caused thrashing.

2.2 Vertical Fusion

Vertical fusion combines a producer and a consumer into a fused loop which computes the consumer directly. As an example, let's consider a domain-specific language where arrays are represented as loops in the intermediate representation. The DSL user writes the following program:

```
val prod = arrayFromFunction(10)({ i => 2*i + 3 })
val cons = prod.map({ x => 2*i*x })
```

The generated Scala code without fusion could look as follows:

```
val prodB = new ArrayBuilder.ofInt
for (i <- 0 until 10) {
  val x0 = 2*i
  val x1 = x0 + 3
  prodB += x1
}
val prod = prodB.result
val consB = new ArrayBuilder.ofInt
for (i <- 0 until prod.length) {
  val x0 = 2*i
  consB += x0*prod(i)
}
val cons = consB.result
```

And this is the generated code of the fused version if `prod` isn't used anywhere else:

```
val consB = new ArrayBuilder.ofInt
for (i <- 0 until 10) {
  val x0 = 2*i
  val x1 = x0 + 3
  consB += x0*x1
}
val cons = consB.result
```

Now the access of the producer has been replaced with the element of the producer at that index, and the fused loop iterates over the range of the producer loop. Vertical fusion thus removes all dependencies between producer and consumer, computing the consumer directly through the fused loop. The example shows how the common subexpression elimination (CSE) optimization was enabled by the fusion, saving a multiplication (`x0`) for each array element.

The fused version also doesn't use an intermediate data structure, saving not only the time for allocation and memory access, but also the space in memory. However, if `prod` is used later on, it needs to be allocated and computed as well. If the producer loop was just generated side by side with the fused loop, the work would be duplicated. Instead, the producer and the fused loop are now in a shape that allows them to be fused horizontally. And in the spirit of taking advantage of existing optimizations, CSE will automatically de-duplicate the computation of the producer in the combined body.

The running example with vertical and then horizontal fusion would generate the following code, with only one loop that computes both arrays at the same time. CSE eliminated the double computation of `x0` and `x1`:

```
val prod_b = new ArrayBuilder.ofInt
val cons_b = new ArrayBuilder.ofInt
for (i <- 0 until 10) {
  val x0 = 2*i
  val x1 = x0 + 3
  prod_b += x1
  cons_b += x0*x1
}
val prod = prod_b.result
val cons = cons_b.result
```

The new loop fusion algorithm is based on this observation and is divided into the following stages. The first transformer does the first half of vertical fusion, namely replacing each consumer with a fused loop that computes it directly, without any dependencies on the producer. After each transformer, the scheduler runs and triggers general optimizations like CSE and DCE. In particular, dead code elimination (DCE) removes the producers that aren't used anywhere else. Next, the horizontal fusion transformer combines surviving producers with their fused loops as well as horizontally fuses all other sets of loops that satisfy the conditions. After running the general optimizations again, a scheduler extension brings the program into the final shape, and CSE is important at this stage to deduplicate the producer body.

The conditions for vertical fusion are:

V1. The consumer iterates over the length of the producer
V2. The consumer body only uses the producer at the current index, but is otherwise independent
V3. Their effects can be interleaved and the producer value is pure
V4. They are in the same exact scope

The next chapter will give a brief overview of LMS and introduce the main concepts used in the implementation of the new loop fusion.

3. Introduction to LMS

Lightweight modular staging (LMS) [1] is a framework for creating domain-specific languages (DSLs) embedded in Scala by harnessing the type system for multi-stage compilation. The following will introduce the concepts necessary for understanding the implementation of loop fusion.

LMS transforms a program written in a DSL into an intermediate representation (IR). The IR consists of nodes that capture the domain-specific operations as well as general language constructs. The abstract syntax tree (AST) contains all of those nodes in a sea-of-nodes and static single assignment (SSA) representation.

3.1 AST: Sea-of-Nodes in SSA Form

Let's illustrate these terms with an example. Here is a snippet of DSL code that creates an array with a single element, which depends on a condition:

```
val y = if (x > 0) singleton(x + 6)
        else singleton(x - 6)
```

For this project we don't need to know how the program gets turned into an AST, so we'll skip that step and directly look at the corresponding AST:

```
TP(Sym(1),OrderingGT(Sym(0),Const(0)))
TP(Sym(2),IntPlus(Sym(0),Const(6)))
TP(Sym(3),Singleton(Block(Sym(2))))
TP(Sym(4),IntMinus(Sym(0),Const(6)))
TP(Sym(5),Singleton(Block(Sym(4))))
TP(Sym(6),IfThenElse(Sym(1),Block(Sym(3)),Block(Sym(5))))
```

Each line is a *statement* or *TP*, a "typed pair" of a symbol and a definition. The *symbol* wraps an integer identifier, contains type information and carries source context information for debugging (which line in which file a particular statement came from). The right-hand side of each TP contains a *definition*, which is an IR-node defined either by a DSL (`Singleton` in the example) or already contained in LMS like the `IfThenElse`.

The AST is in SSA (static single assignment) form because the definitions aren't nested (e.g. the second and third line could have been combined into: `TP(Sym(3), Singleton(IntPlus(Sym(0)),Const(6)))`). Instead, each definition is statically assigned to a unique symbol, and definitions can only refer to *expressions* - either previously defined symbols or constant nodes (`Const(6)` for example).

This representation is called sea-of-nodes because it doesn't impose any structure, it's just an unordered set of nodes which can float around, without a notion of scopes. There is a partial ordering between nodes defined by dependencies, but this information follows from the nodes themselves and isn't stored separately in the AST. Dependencies define whether a statement has to be computed *before* another. While the sea-of-nodes representation has maximum flexibility for optimizations, code generation ultimately needs to emit the statements in a particular sequence that doesn't violate the partial order, and with scopes (assuming the target language has scopes). The process of finding such a sequence is called *scheduling*.

3.2 Scheduling, Blocks and Loops

The scheduler starts with the final statement and recursively covers all dependencies:
`TP(Sym(6),IfThenElse(Sym(1),Block(Sym(3)),Block(Sym(5))))`

The way the `IfThenElse` node is defined, it takes an expression for the first argument, which is the condition, and then it takes two blocks for the then- and else-branches. A *block* is just a wrapper for an expression, called the block result. The block doesn't carry any additional information, but it signals to the scheduler that the statements needed to compute the block result need special treatment.

When the scheduler encounters the `IfThenElse`, it sees that the statement computing Sym(1) is a dependency and has to be scheduled before. But what about the blocks? How about the following pseudo-code:

```
val x1 = OrderingGT(Sym(0),Const(0))
val x3 = Singleton(IntPlus(Sym(0),Const(6)))
val x5 = Singleton(IntMinus(Sym(0),Const(6)))
val x6 = if (x1) x3 else x5
```

Clearly only one of the singleton arrays will ever be used, so the scheduler shouldn't just treat the block results like dependencies and schedule them before the `IfThenElse`. Instead, the DSL author can override the `symsFreq` function, which tells the scheduler how blocks should be handled. In the case of the `IfThenElse`, both blocks are marked as *cold,* meaning that the scheduler should not compute them eagerly before the node in question. The opposite happens for loops, where the loop body is marked as *hot*, meaning that whatever can be computed before the loop (and therefore only once instead of at each iteration) should be scheduled before the loop.

In the running example, the scheduler will decide that symbol 1 should be first, and then schedule the IfThenElse. It will also compute that for the then-block it needs statements 2 followed by 3 and for the else-block 4 followed by 5.

With the schedule sorted out, the next and last step is *code generation*, which is again defined by the DSL author. As an example let's take Scala code generation. The code generator walks through the schedule, starting by emitting:

```
TP(Sym(1),OrderingGT(Sym(0),Const(0))) ----> "val x1 = x0 > 0"
```

The next statement in the schedule is the `IfThenElse`, which now has nested scopes. Its code generation will call `emitBlock` for each of the inner scopes:

```
TP(Sym(6),IfThenElse(Sym(1),Block(Sym(3)),Block(Sym(5))))
----> "if (x1) {" emitBlock(Sym(3)) "} else {" emitBlock(Sym(5)) "}"
```

`EmitBlock` then triggers code generation for all missing dependencies required to compute the block result. In the case of the then-branch, `x2` is required to compute the block result `x3`, but hasn't been emitted yet, so it gets passed to code generation now. This is the final code:

```
val x1 = x0 > 0
val x6 = if (x1) {
  val x2 = x0 + 6
  val x3 = Array(x2)
  x3
} else {
  val x4 = x0 - 6
  val x5 = Array(x4)
  x5
}
```

Some IR nodes define symbols they need internally, but which aren't the left-hand side of any TP. Therefore when the scheduler encounters that symbol, it doesn't know how to satisfy the dependency. This is the case for the loop index variable for example. DSLs override `boundSyms` to return any internal symbols a node might have, which means for the scheduler that the symbols will be defined by code generation of the node, but only for the blocks of the node (e.g. the loop body).

Since the loop block is marked as hot, the scheduler needs to move as many statements out of the loop body as possible. But everything that depends (directly or indirectly) on the loop index and is necessary to compute the block result cannot be moved out of the loop body, so this is the minimum set that cannot be moved. For example, the following DSL snippet defines an array of size 10 by computing the values as a function of the array index:

```
val y = array(10)({ i => x + 5 + 2*i })
```

In the generated code, the computation of `x+5` has been moved out of the loop, but the rest depends on the index variable `i` and is necessary to compute the value at the current index, so it can't be moved:

```
val x3 = x0 + 5
val x7 = new Array[Int](10)
for (x1 <- 0 until 10) {
  val x4 = 2 * x1
  val x5 = x3 + x4
  x7(x1) = x5
}
```

3.3 Optimizations, Transformers and Mirroring

Some of the most important compiler optimizations are dead code elimination (DCE), code motion and common subexpression elimination (CSE). All three happen automatically in LMS.

DCE is a natural consequence of how scheduling works. Since the scheduler pulls in dependencies starting from the final node, dead code will never get scheduled because by definition nothing depends on it. Similarly, code motion happens because the sea-of-nodes AST doesn't capture any scopes, and therefore the scheduler is free to order the code in whichever way seems best, as long as the partial order of the dependencies is satisfied and ideally respecting the hot/cold heuristic.

CSE is achieved because each new IR node is compared against the existing ones and if it already exists the same symbol is used, otherwise a new statement is created in the AST and the new symbol returned. The scheduler then decides how many times the statement has to be emitted, depending on the final scopes. For example:

```
print(x + 1)
val y = if (x > 0) {
  if (x > 10) x + 5 else x + 1
} else x + 5
```

The AST contains only two `IntPlus` nodes: `TP(Sym(1),IntPlus(Sym(0),Const(1)))` and `TP(Sym(5),IntPlus(Sym(0),Const(5)))`. In the generated code, `x1` is only emitted once, whereas the definition for `x5` can still be found twice because it is in different scopes:

```
val x1 = x0 + 1
val x2 = println(x1)
val x3 = x0 > 0
val x7 = if (x3) {
  val x4 = x0 > 10
  val x6 = if (x4) {
    val x5 = x0 + 5; x5
  } else { x1 }
  x6
} else {
  val x5 = x0 + 5; x5
}
```

Domain-specific optimizations can often be implemented as parts of the AST construction. As a simple example, instead of creating a definition for `IntPlus(Const(4),Const(5))`, the DSL can directly replace it with `Const(9)`.

For more complex optimizations, LMS provides transformers and mirroring. Mirroring means recreating IR nodes with possibly changed dependencies. A transformer walks through a schedule, changing existing nodes and/or creating new ones. It typically changes a few and for most nodes just calls mirror, which recreates the node, but also recurses into any blocks that the node might have.

For many transformations it is sufficient to look at the current and past statements to decide how to process a statement. These transformers are called *forward transformers*, because they move from the beginning to the end of the schedule and transform the statements one by one. Furthermore, they are generally *substitution transformers*, which means that they maintain a map from old symbols to new expressions. So whenever a statement is changed, all dependencies (which can only come after the statement) need to be updated. This happens through mirroring: each node applies the transformer to its dependencies, the transformer returns the substituted expressions instead of each old symbol and the node then recreates itself with the transformed dependencies. The construction of the new node might again trigger domain-specific optimizations as seen above.

After each transformer, the scheduler runs again. This means that CSE, DCE and code motion happen automatically after each transformer. In particular, all the old statements that aren't used anymore won't make it into the new schedule. For the loop fusion project this is important because DCE removes producers that aren't used anymore.

The loop fusion transformers are *preserving forward substitution transformers* because they use an additional optimization to preserve statements/symbols whose dependencies haven't changed whenever possible. This keeps the AST small, which is good for compilation times.

3.4 Effects, Reflect and Reify

The effect system is the last part of LMS needed to understand the implementation of loop fusion. If scheduling is driven only by dependencies, what happens with the following DSL snippet?

```
print(x)
val y = if (x > 0) singleton(x) else {
  print("empty")
  print("Arr")
  emptyArray[Int]
}
print(y)
```

If the block result of the else-branch was just the emptyArray, then the two inner print statements would never get scheduled. Instead, effects are tracked as additional dependencies with two special nodes, *Reflect* and *Reify*. The reflect node wraps an effectful definition and adds effect information to it. A reify node wraps a block result expression and tracks all effects that were reflected in the block. As a consequence, effectful statements are tied to their block and cannot be moved outside, which is the behavior one would expect - print at every iteration of the loop, print only if that branch is executed.

In the running example, the AST looks as follows (with the statements of the then-branch in color):

```
TP(Sym(1),Reflect(Print(Sym(0)),Summary(...),List()))
TP(Sym(2),OrderingGT(Sym(0),Const(0)))
TP(Sym(3),Singleton(Block(Sym(0))))
TP(Sym(4),Reflect(Print(Const(empty)),Summary(...),List(Sym(1))))
TP(Sym(5),Reflect(Print(Const(Array)),Summary(...),List(Sym(4))))
TP(Sym(7),EmptyArray(Sym(6)))
TP(Sym(8),Reify(Sym(7),Summary(...),List(Sym(4), Sym(5))))
TP(Sym(9),Reflect(IfThenElse(Sym(2),Block(Sym(3)),Block(Sym(8))),
  Summary(...),List(Sym(1))))
TP(Sym(10),Reflect(Print(Sym(9)),Summary(...),List(Sym(9))))
TP(Sym(11),Reify(Sym(10),Summary(...),List(Sym(1), Sym(9), Sym(10))))
```

Each `Reflect` node has three parts:

- The wrapped definition
- The summary describing the nature of the effect. Printing is an I/O effect for example, and reads/writes of mutable state are a different kind of effect

- The list of dependencies: this captures the partial ordering between effectful statements. Not all effects are serialized, reads and writes are only serialized if they access the same mutable state, and I/O effects are tracked separately. But if there are preceding nodes of the same effect class(es), then the dependencies list tracks the predecessor of each type.

For the statement defining symbol 1, the print has no dependencies since there are no previous I/O effects. For symbol 4, the dependency list now tracks Sym(1), and for symbol 5 it contains Sym(4), the preceding effectful node.

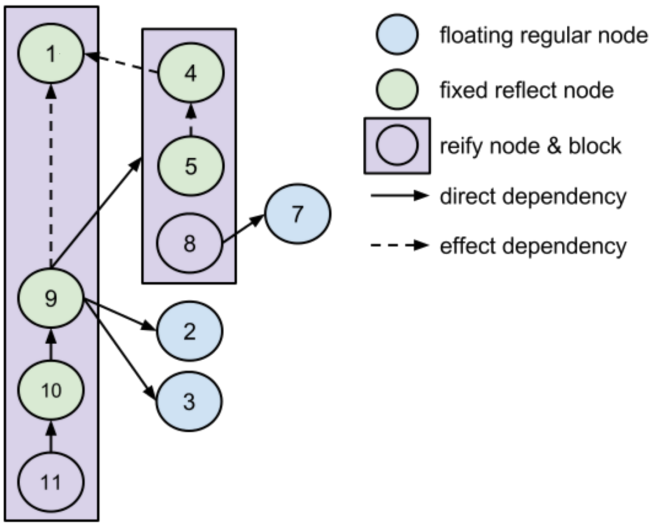
Next comes a `Reify` node. It has three parts that look similar, but play a slightly different role:

- The first part is now an expression instead of a definition. This is the block result value. But as seen previously, it is not sufficient to track only the block result value if effects should get executed. So the reify node summarizes all effects that have been reflected in its block.
- The summary is a combination of all summaries of the reflect nodes in the block
- The list contains all reflect nodes from the block

The else-branch block contains this reify symbol 8 instead of the block result value (symbol 7).

The `IfThenElse` node is now also wrapped in a `Reflect` node. The dependencies in the reflect node are the outward dependencies of the reflects in the inner scope, e.g. the dependencies on statements outside of the block. This is why statement 9 just tracks a dependency on symbol 1, the `IfThenElse` node reflects the dependencies of its inner blocks on statements in the *exact scope* of the node, where exact scope designates a scope without its inner scopes.

The last statement is again a `Reify` node, because the top level scope is treated exactly as all inner scopes, and its block result is the last print statement.



The schema illustrates how the `Reify` nodes fix the `Reflect` nodes in their block, whereas the scheduler can move pure nodes around. Now all nodes are reachable from the last statement (11) and thus the effectful statements will be scheduled and emitted too, as long as there are correct `Reflect` and `Reify` nodes to track all effects.

4. Scheduler Implementation

The old loop fusion was implemented completely in the scheduler and came only as the very last step before code generation. The advantage was that it could manipulate the whole AST at will, but the disadvantage was that the algorithm was difficult to understand and not integrated with other optimizations. This became a problem for the GPU multidimensional loop analysis, which collects information about loops that is used for code generation. But since loop fusion ran in between, the final AST had different loops and symbols.

Another Delite optimization is the array-of-struct to struct-of-array (AoS to SoA) transformation [5]. While most of it was implemented as a transformer, the last step also required fusing the loops it created. Since there was no way to communicate this to the old loop fusion, SoA had to implement another scheduler extension to pre-fuse its loops.

For these reasons my new implementation consists of two transformers (first vertical, then horizontal fusion) and a minimal scheduler extension with an open interface. The scheduler part is necessary because there is a fundamental difference between the AST that transformers and analysis passes operate on, and the AST that the scheduler feeds to code generation.

4.1 TTPs and Fat Nodes

The AST presented in section 3.1 is a set of TPs, each a pair of a symbol and a definition. But this means that each definition defines exactly one symbol, so how can a fused loop return multiple results? The scheduler that runs for code generation uses *fat nodes*, which define multiple symbols with one *fat definition* through a new type of statements: *TTPs*. A TTP is the fat version of a TP, and it contains three parts:

- lhs: the left-hand side is the list of symbols it defines
- mhs: the middle-hand side is the list of the original thin TPs that were combined into this fat TTP
- rhs: the right-hand side is one fat definition that calculates all the symbols at once

Let's look at an example of horizontal loop fusion:

```
val a1 = array(10)({ i => 2*i })
val a2 = array(10)({ i => 2*i + 3 })
```

The DSL in use defines and processes arrays through loops. The loop node is called `SimpleLoop` and contains the range, then the loop index variable, and last an element representing the loop body. `ArrayElem` creates an array where each element is the result of applying the function to the index variable:

```
TP(Sym(2),IntTimes(Const(2),Sym(1)))
TP(Sym(3),SimpleLoop(Const(10),Sym(1),ArrayElem(Block(Sym(2)))))

TP(Sym(5),IntTimes(Const(2),Sym(4)))
TP(Sym(6),IntPlus(Sym(5),Const(3)))
TP(Sym(7),SimpleLoop(Const(10),Sym(4),ArrayElem(Block(Sym(6)))))
```

↓

```
val x3 = new Array[Int](10)
for (x1 <- 0 until 10) {
  val x2 = 2 * x1
  x3(x1) = x2
}
val x7 = new Array[Int](10)
for (x4 <- 0 until 10) {
  val x5 = 2 * x4
  val x6 = x5 + 3
  x7(x4) = x6
}
```

After loop fusion, the second loop has been mirrored to also use `x1` as index, with `x11` as the new loop body and `x12` as loop symbol. The AST contains one fat TPP for the fused loop:

```
TP(Sym(2),IntTimes(Const(2),Sym(1)))
TP(Sym(11),IntPlus(Sym(2),Const(3)))
TTP(List(Sym(3), Sym(12)),
    List(SimpleLoop(Const(10),Sym(1),ArrayElem(Block(Sym(2)))),
         SimpleLoop(Const(10),Sym(1),ArrayElem(Block(Sym(11))))),
    SimpleFatLoop(Const(10),Sym(1),
      List(ArrayElem(Block(Sym(2))),
           ArrayElem(Block(Sym(11))))))
```

↓

```
val x3 = new Array[Int](10)
val x12 = new Array[Int](10)
for (x1 <- 0 until 10) {
  val x2 = 2 * x1
  val x11 = x2 + 3
  x3(x1) = x2
  x12(x1) = x10
}
```

4.2 Transformers on TTPs?

Loop fusion ultimately produces TTPs. The example above shows that the body of the fused loop isn't just the body of the first one concatenated with the second one, one multiplication has been CSE'd. So ideally other transformers and analysis passes would operate on the fat AST with TTPs. Unfortunately, the current architecture of LMS doesn't support this because mirroring is only defined for thin nodes, and therefore the transformer framework can't handle TTPs. Changing the signature of mirroring would require changing not only the LMS framework and all built-in nodes, but every single DSL ever built would have to change. The effect system would have to change too, since it is built to reflect and reify single blocks. For these reasons I decided to keep the last step of loop fusion in the scheduler and find another way for transformers and analysis passes to get fusion information.

As a side note, pure analysis passes can't use the fat AST either, because there is no notion of a body consisting of several blocks. The blocks in a fat definition are only combined at the very end by their code generation, which calls the scheduler with the list of blocks, and the scheduler then collects all the effect dependencies and manages the CSE. However, since almost all nodes are only thin nodes, the changes might be more limited and it might in fact be sufficient to change the `SimpleLoop` and `IfThenElse` nodes to allow for a fat traversal.

4.3 CombineTTPScheduling and CanBeFused

The extension to the scheduler that handles fusion can be found in `CombineTTPScheduling.scala`. While the initial loop fusion only checked and fused loops, this extension can fuse all kinds of fat nodes. In particular, the new loop fusion requires combining `IfThenElse` nodes with the same condition to fuse filter nodes (see section 6.10). The new scheduler extension should therefore handle these two and all future cases of combining several TPs into a TTP.

To this effect, I introduced the `CanBeFused` trait, which can be mixed into DSL nodes and tracks fusion information for them. Currently, it is used in `IfThenElse` and `SimpleLoop` nodes. All transformers that need to fuse nodes can now set the fusion information, and the scheduler extension handles the combination for all of them. In particular, the SoA transformation doesn't need its own scheduler extension anymore.

`CombineTTPScheduling` collects all sets of nodes per `CanBeFused.fusionSetID` and calls `combineFat` on each set. The DSLs override it to return the right kind of TTP, e.g. a set of loops will be combined into a `SimpleFatLoop-TTP`. This design delegates the responsibility to each DSL and allows `CombineTTPScheduling` to be very concise, easy to understand and general so it can handle other types of fusion in the future. It runs in a single pass instead of needing multiple iterations until it converges, and it just traverses the full list of all statements that aren't dead, without needing to focus on each exact scope separately.

Since the TTP usually just groups a set of existing definitions into a fat definition, the scheduler doesn't introduce any new symbols. The preparation has already been done by the transformers. Previously, the fusion scheduler changed loops and symbols and therefore code generation saw a different set of loops and symbols than the analysis passes which ran before (in particular, this made loop fusion incompatible with GPU multidimensional loop analysis).

4.4 Fused Scopes

Some analysis passes and transformers need to know about scopes. The horizontal fusion transformer for example can only fuse two loops if they're in the same exact scope, so it maintains a datastructure capturing the set of loops seen in each scope. But fusion has the effect of combining inner scopes, for example two bodies of two loops will form one body of the fused loop. So when processing the second body, all loops from the first body are also candidates for horizontal fusion.

Such transformers can combine their per-scope datastructures by querying `CanBeFused` information to see which nodes have been fused. They still need to pattern match on the type of node, since a loop has one body, whereas an `IfThenElse` has two branches that need to be combined respectively. But this solution allows them to operate on the post-fusion AST without having to redesign LMS and change all DSLs.

But how about those optimizations happening on the fused bodies? Since no new definitions are created (only new fat definitions regrouping each set of thin definitions), no domain-specific optimizations are triggered. DCE cannot happen since fusion just groups things differently, if something was alive before it will still be in use after. Code motion is also not likely, since the same hot-cold heuristics will be used for fat loops and ifs.

The only optimization that needs some consideration is CSE. Indeed, this is what happened in the code example. Quick reminder about how CSE works: There is only one statement and symbol for each definition in the AST, and the scheduler only emits it if it isn't already in scope where it is used. So even if a statement was used in both loops, it will only be emitted once for the fused loop. If the transformer or analysis pass is sensitive to CSE, it can just track all the symbols seen per exact scope. In this way, when transforming the second scope it can ignore the symbols that were already encountered.

In conclusion, the new scheduler extension uses the general interface of `CanBeFused` and can thus cover a wide range of fusion-like operations. While TTPs still only live here and in the code generation phase, transformers with per-scope datastructures can easily query this information to combine the datastructures of scopes that will be fused. Furthermore, analysis results stay valid since the scheduler doesn't change any definitions or symbols.

5. Horizontal Transformer

This chapter starts by explaining the underlying mechanics of both fusion transformers and then covers the implementation of the simpler one, namely the horizontal fusion transformer. It is less complex than the vertical transformer because it just combines side-by-side loops, whereas vertical (or pipeline) fusion replaces parts of one loop with the other and needs to know the type of each, e.g. fusing a map with a reduce works differently from a filter with a map.

5.1 Fusion Transformers

As explained in section 3.3, both fusion transformers are preserving forward substitution transformers. Their skeleton looks as follows:

```
trait XXXLoopFusionTransformer extends PreservingFixpointTransformer {
  // per-scope datastructure definitions

  // collection of all per-scope objects

  // variable for current per-scope object

  override def reflectBlock[A](block: Block[A]): Exp[A]

  override def transformStm(stm: Stm): Exp[Any]

  // helpers to check fusion conditions

  // helpers to change fused nodes
}
```

`ReflectBlock` is called for each new block/scope. The transformer loads the correct per-scope datastructure, and then calls `super.reflectBlock`, which will call `transformStm` on each statement that is scheduled to be in the inner scope, in the order of the schedule. If some statements have nested blocks, they are reflected before proceeding. Once all statements have been transformed, the transformer resets the per-scope datastructure to the outer one and returns the transformed block result.

`TransformStm` is usually structured as a pattern match. The transformer has three choices: return the original symbol, mirror the definition in the statement, or it can return a different expression (symbol or constant). The default behavior for a preserving transformer is to return the old symbol whenever possible, but mirror if one of the following conditions is true:

- The statement contains symbols (dependencies or bound symbols) that need to be substituted: mirroring is the mechanism to recreate a node with changed dependencies
- The statement contains blocks or functions: mirroring calls `reflectBlock` on those to recurse into the inner scopes, otherwise only the top-level scope could be transformed
- The statement is effectful: As described in section 3.4, the effect system is based on two special nodes: `Reflect` and `Reify`. Conceptually, `Reify` collects all effects (marked by `Reflect` nodes) from a block and states them together with the block result. So when `reflectBlock` returns the new block result, it has to wrap it in a `Reify` containing all effects of the block. The way this is implemented, `super.reflectBlock` takes a snapshot of the AST before and after the transformation, and then collects all effectful statements in the difference. Therefore all effectful statements have to be mirrored, so they are recreated and visible in the difference.

5.2 Datastructures

Now we can examine the `HorizontalLoopFusionTransformer` in more detail. Its role is to fuse side-by-side loops as well as side-by-side ifs, given the following four properties hold:

H1. They are in the same scope
H2. They iterate over the same range/have the same condition
H3. Neither depends on the other
H4. Their effects can be interleaved

Since more than just two nodes can be fused together, horizontal fusion effectively partitions the set of all loop&if nodes *S*. The subsets forming the partition will be called *fusion sets* (abbreviated as *fset*). By the definition of a partition, the union of all fusion sets is S, each fusion set is non-empty (but might be a singleton) and the fusion sets are mutually exclusive.

Because of H1, the exact scopes pre-partition S, since no fusion set can contain nodes from two different scopes. The per-scope datastructure of the horizontal transformer therefore only needs to capture the fusion sets of the associated scope. It is called `FusionScope` and offers the following interface:

- lookup of set by symbol (for both ifs and loops)
- lookup of candidate set(s) by shape for loops and by condition for ifs
- record a new fusion set
- record addition of nodes to an existing fusion set
- record new symbol after transformation

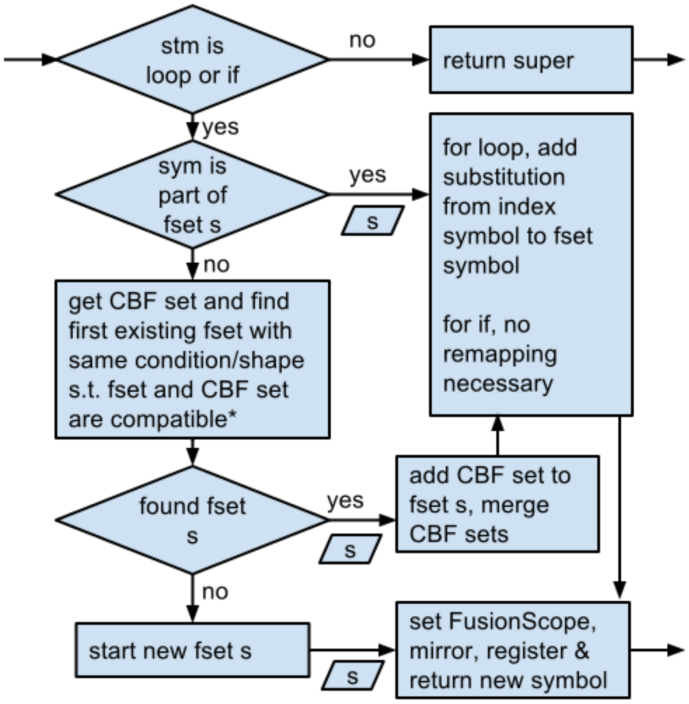
Each loop fusion set has a reference to the `FusionScope` object for its inner (combined) scope, and the if-sets have two, one for the then-branch and one for the else-branch. The transformer has a map from block results to `FusionScopes`, and before mirroring a loop for example, the transformer puts a mapping from the block result of the body to the correct `FusionScope`. When `reflectBlock` is called on the block, it retrieves the saved `FusionScope`

instead of creating a new one, and thus all bodies of a fusion set are mirrored with the same per-scope object, so loops can be fused even if they will only be in the same scope once the outer loops are fused into a TTP.

Vertical fusion and other transformers like SOA that run before horizontal fusion use the `CanBeFused` trait to mark sets of nodes that need to be fused together. These *CBF sets* impose an additional constraint on the partition, because if the nodes aren't fused, the optimizations will result in duplication of computation. Therefore CBF sets may not be split across multiple fusion sets, and their inner scopes also have to be combined.

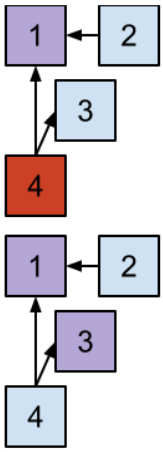
5.3 Algorithm

The algorithm for `HorizontalLoopFusionTransformer.transformStm` proceeds as follows:



All loops in an fset need to use the same index symbol, so that they can be combined into a fat loop with that index. Otherwise some bodies would have to be changed in the scheduler. So when a new fset is created for a loop that cannot be fused with any existing set, the index variable of that loop is picked. For each loop added to an existing fset, the transformer adds a substitution from the old index variable to the one of the set. Mirroring takes care of the rest, as all statements in the loop body are transformed the index gets replaced according to the substitution map.

5.4 Greedy Coloring



What happens if several fusion set candidates are compatible with a CBF set in the box marked with an asterisk in the algorithm? As the dependency graph on the right shows, some fusion decisions will prevent fusion opportunities further down the road. Node 2 depends on node 1, so they aren't fused. Node 3 could be fused with 1 or 2, we arbitrarily pick 2. But then node 4 cannot be fused with the set (2,3) anymore, since it depends on 3. But it also can't be fused with node 1. Had we fused node 3 with node 1, then we could have done two fusions instead of just one (the lower picture).

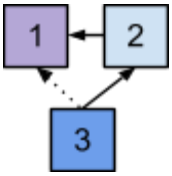
The most important constraint is that vertically fused and SOA loops need to be fused, but this is guaranteed since each CBF set is a subset of an fset. After that, it is difficult to define what a good partition is. For example it would be great if loops that access the same data could be fused. Maybe long loops with simple bodies should be prioritized, because the time saved by only having one iteration is more significant. Maybe the algorithm should actually favor small fusion sets, because fused loops that access too much data can cause thrashing and interfere with prefetching.

In future work, it would be interesting to investigate annotations so that users can mark loops that should or shouldn't be fused. Another Delite member is working on a debugger to visualize performance aspects, and is interested in developing hooks or an autotuner that could also provide feedback to fusion.

For the time being, I use the number of fusions as a metric, call it f. The number of fsets k is related to it by the total number of nodes n as follows: $n = k + f$
This holds by induction: in the base case of $n = 0$, there are no fusion sets and no fusions. If it holds for $n = k + f$, then it holds for $n+1$, since the next node can either:

- form a new fset: $n+1 = (k+1) + f$
- be fused with an existing set: $n+1 = k + (f+1)$

Since the number of nodes n in the AST is fixed, maximizing f is the same as minimizing k. The problem now becomes the graph/vertex coloring problem, since we want to find an assignment using the minimum number of colors (corresponding to fsets) such that no two neighboring nodes (corresponding to nodes that transitively depend on each other) have the same color. The dependency graph needs to be modified as follows: Instead of just using direct dependencies, the graph also needs directed edges for all transitive dependencies. Once those are added, it can be treated as an undirected graph, called the coloring graph in the following, which is then the input to the graph coloring problem.



In the example on the right, the directed edge from 3 to 1 needs to be added, otherwise 3 and 1 could be colored in the same color. The coloring graph in this case is K3 and thus needs three colors.

Graph coloring is NP-complete in the general case, but the coloring graph described above belongs to a special class of graphs called comparability graphs, which are defined exactly as the graphs resulting from connecting comparable nodes according to a partial order relation on a set of nodes, in this case the dependency relation.

While the greedy graph coloring algorithm (process each node in sequence and assign it to the first available color) generally does not find an optimal assignment, there is an order for each graph such that the greedy algorithm results in an optimal assignment - that is, the minimum number of colors is used. And since the coloring graph is a comparability graph, it belongs to the class of perfectly orderable graphs, for which any topological ordering of the nodes results in an optimal assignment when using the greedy algorithm [6]. As it turns out, the scheduler orders the nodes such that no node can come before any of its dependencies, so the transformer sees the nodes in a topological order and can thus use the greedy algorithm.

Therefore the horizontal transformer uses the algorithm of picking the first compatible fusion set. It is very simple and fits in the framework of the forward transformers: each node can be assigned to a set immediately when it is processed and doesn't need to be revisited. It is optimal for the metric chosen if the loops in a CBF set are considered as already fused into one loop.

5.5 Performance

Most parts of the diagram representing the horizontal fusion algorithm are either just part of the transformer (linear time overall) or constitute hashmap operations (contributing at most linear time overall). The most interesting part is the compatibility check marked with an asterisk in the diagram: On one side there's the CBF set of the current node, which might only consist of the node itself. On the other side, there's a list of candidate fsets with the same characteristic (same condition for ifs, same range for loops).

Because the transformer uses the greedy algorithm to pick the candidate, it needs to find the first fset that is compatible. This means that all loops in the CBF set and in the fset satisfy the conditions H3 and H4: neither depends on the other and their effects can be interleaved. For the second condition, there is currently no good way in LMS to check whether effects could be interleaved safely, therefore at most one is allowed to be effectful. Each fset has a flag to indicate whether it already contains an effectful loop. If it doesn't, the CBF set doesn't even need to be checked. If it does, then no node in the CBF set can be effectful. While the fset has a convenient flag, the effect information needs to be computed for a CBF set. CBF information is stored in the nodes themselves, so the AST has to be traversed to find the statement encapsulating the definition, which is either wrapped in a Reflect and thus effectful or pure otherwise.

In future work, we would like to explore how to improve this step. The main problem comes from mirroring, because when a CBF node is mirrored, the new one is automatically added to the same set so that CBF information survives transformers. The set also contains pointers to all other nodes. Ideally these would be the symbols associated with the definitions, because there's a cache for looking up statements by symbol, which could give the definition and effect information in constant time. But during mirroring, only the new definition is available, the symbol is assigned later. It would be possible to assign a symbol in the CBF function, but that would change the system for a small class of nodes, which could cause confusion and have unexpected side effects. As a compromise, future work could add an optional function for loop transformers to set the symbol, so they can save it when they're mirroring CBF nodes or after they've looked it up.

The dependency check is more complicated. It needs to find out whether any of the nodes in the CBF set (transitively) depend on any nodes in the fset or vice versa. Both ways need to be checked, even though the CBF set comes after the fset in the schedule, because the fset might contain nodes that will come later in the schedule, but were part of an incorporated CBF set.

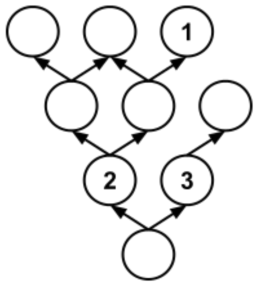
The following example shows such a case: `arr1` and `arr3` get vertically fused, resulting in the second snippet, where all three arrays have the same shape and 3 depends on 2.

```
val arr1 = array(100) { i => i + 1 }
val arr2 = array(100) { i => i + 2 }
val arr3 = array(arr1.length) { i => arr1.at(i) + arr2.at(0) }
// code using all three arrays
```

```
val arr1 = array(100) { i => i + 1 }
val arr2 = array(100) { i => i + 2 }
val arr3 = array(100) { i => i + 1 + arr2.at(0) }
// code using all three arrays
```

The horizontal fusion transformer first encounters `arr1`, which isn't part of any fusion set yet, so the transformer gets its CBF set (1,3). There's no existing fusion set to be merged with, so (1,3) becomes a new fset. Next is `arr2`. It isn't part of any fusion set yet, and its CBF set is just (2). Now the fset (1,3) has the correct range and neither is effectful, so it seems like a good candidate for merging. If we only checked that the CBF set (2) doesn't depend on the fset (1,3), we would catch the case where `arr2` depends on `arr1`, but here the two would be merged and code generation would fail with a `Recursive Schedule Error`: the expression computing `arr2` (the fat loop with all three bodies combined) depends on `arr2`, which is a cycle in the dependency graph and cannot be satisfied with a linear schedule. Therefore both directions need to be checked, because the fset can depend on the CBF set too.

The next question is how the dependency check works. For the dependency graph on the right, nodes 1 and 3 would pass the check, because neither depends on the other. 2 and 3 would work too, but 1 and 3 cannot be fused because the 2 depends on 1. Formulated as a graph problem, two nodes are independent if there is no directed path from either to the other one. So for the CBF set and the fset, no node in the fset can be reachable from the CBF set and inversely.



Horizontal fusion currently uses the existing implementation of Tarjan's algorithm, which does depth-first search (with an additional stack to compute the strongly connected components, which doesn't change the complexity, but increases the constant). DFS has complexity $V + E$, where V is the number of vertices and E the number of edges of the graph. So each time a CBF set is checked against an fset, the algorithm does two depth-first searches, first starting from one set and checking that no nodes of the other can be reached, and then the other way around. The DFS is stopped as soon as a node from the other set is reached.

The performance impact of this implementation is difficult to analyze. Each successful horizontal fusion that wasn't forced by previous CBF information costs at least those two depth-first searches. But the algorithm might redo a lot of work, since each time a candidate is rejected there is another full search from the same CBF set, but with different taboo fset nodes. Similarly, an fset might be checked against multiple CBF sets over the course of the algorithm. In any case, the performance of the checks doesn't change the runtime of the generated program, it just improves compile and code generation times.

5.6 Performance Improvement Ideas

Besides the obvious improvement of using pure DFS instead of strongly connected components, I have other ideas for speeding up this step. The first one would be to compute the full matrix of dependencies at the beginning. This can be done using the Floyd-Warshall algorithm in $O(V^3)$, or using a DFS from each node in $O(V(V+E))$. While theoretically $E \in O(V^2)$, most DSL nodes have a small number of dependencies, so the graph is sparse, with E likely of the order of V. The second alternative therefore has a better asymptotic behavior. Furthermore, fusion doesn't need the dependencies from all nodes to all nodes, but only from loops to loops and ifs to if. In the DFS approach this knowledge can be used to only run DFS from loop and if nodes, and then only store the found nodes of those types, considerably reducing the space needed.

But even then, the DFS will traverse many parts of the tree several times. If there was an ordering such that no part of the tree needed to be traversed several times, that would greatly speed things up. Again, the topological order is such a sequence, since each node can only depend on previous nodes whose dependencies will already have been computed. This leads to a design where the horizontal fusion transformer first runs an analysis pass over the full schedule, collects the dependency information and then does the transformation.

But this analysis still computes too much information. Since only nodes in the same exact scope can be fused, it is wasted time (and space) to go past the boundaries of the scope. Now the term scope should always raise a red light, because the scopes in the current schedule need to be combined according to CBF information. Luckily, the analysis pass doesn't need to combine CBF scopes, since nodes from different scopes are always independent. The analysis can therefore stop the DFS at the boundaries of the exact scope.

Note that the analysis cannot be piggybacked on the transformation, because the transformation needs dependency information about nodes in the current CBF set, which occur later in the schedule. But it can speed up the effects check from the last section by filling in the symbols of each CBF definition and pre-computing the effects flag as it traverses the schedule.

Another performance issue is that some of the nodes in each CBF set might be dead. In particular, vertical fusion creates the combined loop and marks it as fused with the producer loop, but the producer becomes dead if it was just an intermediary datastructure in a pipeline and isn't used anywhere else. And as explained before, each transformed node is added to the same CBF set when it's mirrored (otherwise each transformer would erase fusion information), and the old node probably (but not certainly) becomes dead. These dead nodes are still part of the AST, even though they will never get scheduled and therefore won't be transformed by the transformers. However, there's no good way of knowing when a node becomes dead (short of an analysis pass checking each statement in the AST against each statement in the current schedule), so the horizontal transformer currently has to check all nodes in the CBF set and will traverse dead parts of the AST. This waste of work would also be addressed by the new design, since only statements in the current schedule are analyzed and thus dependencies of dead nodes will never be computed.

This new design will be explored in future work. First we need to integrate the loop fusion work into Delite, and if we see that the dependency check is a hot spot for larger programs, we can gather characteristic information to decide on the algorithm. In particular, if exact scopes are pretty small in practice it might be well feasible to store their n^2 matrix of dependencies, whereas for big scopes it might be better to only store direct dependencies and recompute the transitive closure lazily.

6. Vertical Fusion

This chapter will explain the design and implementation of the first transformer, which does the vertical loop fusion. Vertical fusion combines a producer with a consumer into a fused loop that computes the consumer directly. If the producer isn't used anywhere else it is DCE'd, otherwise horizontal fusion fuses the producer with the combined loop since they're marked as fused in their CanBeFused information and have the same shape. The scheduler extension then creates a fat TTP that computes both the producer and the consumer with a single loop. The conditions for vertical fusion are:

V1. The consumer iterates over the length of the producer
V2. The consumer body only uses the producer at the current index, but is otherwise independent
V3. Their effects can be interleaved and the producer value is pure
V4. They are in the same exact scope

6.1 Real vs. Reconstructed Producers

These conditions seem simple enough, but already the first one needs some examination. In the following example, `cons1` clearly iterates over the length of the producer, but so does `cons2`:

```
val prod  = array(100) { i => i + 1 }
val cons1 = array(prod.length) { i => prod.at(i) * 2 }
val cons2 = array(100) { i => prod.at(i) * 2 }
```

It's important to recognize both, because many DSLs do the optimization of replacing the length of a collection with a constant if it is known. In the first case, `prod` is called the *real producer* of `cons1`, whereas it is a *reconstructed producer* for `cons2`. There can be at most one real producer for each loop, but there might be many reconstructed producers. Recognizing them is a prerequisite for fusing with multiple producers as explained later on.

It isn't enough to look at the range of the loop however, since if the producer was computed by a filter expression, its output length might be smaller than the input length. Similarly for a flatMap, the output could be smaller or bigger than the input. This brings us to our first two *extractors*. Loop fusion isn't tied to a particular DSL, it is designed to work on all DSLs that use loops and define the extractors. The names of the extractors are mostly the same as in the old loop fusion, since DSL authors are already familiar with them.

The first extractor is called `SimpleDomain`, and is intended to match the `prod.length` expression above. While an array DSL might call this IR node `ArrayLength`, another one might call it `VectorSize`. Each DSL can override the functions in the trait `LoopFusionExtractors`, in the case of my example DSL it looks as follows:

```
override def unapplySimpleDomain(d: Def[Any]): Option[Exp[Any]] = d match {
  case ArrayLength(array) => Some(array)
  case _ => super.unapplySimpleDomain(d)
}
```

Now loop fusion doesn't have to match on `ArrayLength` (and be changed for every new DSL), but can just use the following extractor object defined in `LoopFusionCore`:

```
object SimpleDomain {
  def unapply(d: Def[Any]): Option[Exp[Any]] = unapplySimpleDomain(d)
}
// Pattern match uses SimpleDomain instead of ArrayLength:
statement match {
  case TP(sym, SimpleDomain(array)) => // do something with the array
}
```

This extractor handles the case of real producers. But for the reconstructed producers, loop fusion needs to know whether an expression defines a producer with a fixed output length. The second extractor is called `FixedDomain` and returns an expression of type `Int` (either a constant or a symbol) if the output length of the collection being matched is fixed.

The name domain might be misleading, since it commonly designates the input set, and image or range is the output set. But domain doesn't refer to the expression defining the producer here (whose domain is 100, but the transformer needs to know its output size), but to the producer collection itself, which can be seen as a function from its index values to its elements.

With these extractors in place, the transformer stores all loops with fixed domains in a map when they are processed. When it then encounters a consumer loop with a potential producer, condition V1 passes if one of the following holds:

- the consumer's range expression matches SimpleDomain(producer)
- the consumer's range and the producer's stored length are the same constant
- the consumer's range and the producer's stored length are the same symbol and the symbol doesn't represent a mutable variable (isWritableSym is false)

As this happens at the same time as vertical fusion, all symbols are also run through substitution, because the new combined loops might have fixed lengths even though the original loops didn't. Furthermore, the transformer uses the fixed length information to replace SimpleDomain expressions with constants whenever possible. Ideally the DSL doesn't do this, because then the transformer can prioritize fusing real producers over reconstructed ones if they conflict. Otherwise that information is lost and the first loop is picked.

6.2 Dependency Check

The second condition specifies how the consumer depends on the producer:

> V2. The consumer body only uses the producer at the current index, but is otherwise independent

Thus an extractor is needed to match the random accesses of the producer. It is called `SimpleIndex` as in the old fusion, because it covers operations that index into a DSL datastructure. In the array DSL used in the examples it will match the `prod.at(i)` expression which is represented by the `ArrayIndex` node:

```
override def unapplySimpleIndex(d: Def[Any]) = d match {
  case ArrayIndex(array, index) => Some((array, index))
  case _ => super.unapplySimpleIndex(d)
}
object SimpleIndex {
  def unapply(d: Def[Any]): Option[(Exp[Any], Exp[Int])] =
    unapplySimpleIndex(d)
}
// Pattern match uses SimpleIndex instead of ArrayIndex:
statement match {
  case TP(sym, SimpleIndex(array, index)) => // do something
}
```

The dependency check is then the same as for the horizontal transformer, except that `SimpleIndex(producer, consumerIndex)` is not counted as a dependency on the producer. The check also pulls in the set of fused nodes for each node, and the same optimization ideas as shown in section 5.6 apply here. Furthermore, the analysis pass suggested there could also speed up the search for producers, which currently traverses the full consumer loop body.

6.3 Effects Check

The third condition specifies what effects are allowed:

> V3. Their effects can be interleaved and the producer value is pure

Once again, since there is currently no way of determining whether two effects are compatible, the first part means that at most one loop can be effectful. But this isn't enough for vertical fusion. Let's consider an example language with the instruction `printX(x: Int)`, which prints `x` on the console and then returns it. If the block result of the producer is the statement `TP(symP, Reflect(PrintX(symX), …)`, then the producer access will be replaced by symP in the consumer body. But as explained in the LMS section, the consumer body can only reify effects that were mirrored during the `reflectBlock` call, so the fused loop wouldn't

be wrapped in the required `Reflect` and the scheduler might reorder it.
But mirroring it again also isn't the solution, because then the final combined loop will have two `printX` statements, one coming from the producer and one from the fused loop. Pure statements are deduplicated through CSE, but effectful statements are assigned a new symbol every time they are mirrored. So the transformer cannot pull in any effects from the producer into the consumer, but it can still fuse an effectful producer with a pure consumer if the *value* of the producer body is pure. The block result of an effectful producer body is a `Reify` node encapsulating the value expression and the effects of the block. If the value and its dependencies are pure, then the producer access can be replaced by the value only, ignoring the `Reify` and the effects. Since effectful loops aren't DCE'd, the producer will survive and eventually be fused with the combined loop. The fat loop will have the effectful statements only once, since they're contributed only from the producer loop, whereas the pure parts of the value computation come from both loops, but are deduplicated through CSE.

The transformer implements this check by traversing all statements in the producer body, starting from the value of the block result and stopping as soon as it finds an effectful statement. If all statements reachable are pure and the consumer (and its set) is pure as well, then the producer and the consumer can be fused. The condition is also satisfied if the producer (and its set) is pure.

6.4 One Producer, One Consumer

The last condition is:

V4. They are in the same exact scope

Note that while the producer is always in scope for the consumer, the consumer might be in a nested scope. If it is nested in a loop, the fused loop would also be there and the producer would get computed over and over again in the loop. If the consumer is in an `IfThenElse`, the fused loop would be scheduled outside because the producer is used outside of it, so the consumer would be computed regardless of the condition. This doesn't only cause computational overhead, an effectful producer in the first case and an effectful consumer in the second case would result in a wrong program, where effectful statements are executed that shouldn't be.

To check this condition, the transformer maintains a list of loops already seen in the current exact scope, and fusion is only possible if the producer is in the list. The scopes don't have to be combined according to the CBF information, because it's not possible that a producer is in one scope and the consumer in the parallel one, since the name of the producer would then be out of scope for the consumer.

Now that the infrastructure is in place to check all conditions, let's look at the general steps necessary to fuse one producer with one consumer. The easiest fusion case is where the consumer maps over the producer. The producer is inlined into the consumer, and the fused

loop uses the index variable of the producer. It can't use the index of the consumer because then the producer body would have to be changed, but it was already transformed and it wouldn't be a forward transformer if it was mirrored twice. Even more importantly, when the producer is fused with multiple consumers, each fusion would result in a new copy of the producer body. The horizontal pass would then have to change all those bodies again so that the combined loop uses a single index variable. If the producer index is used instead, all that work and blowup of the AST is avoided.

Let's illustrate the general vertical fusion transformer algorithm using this example:

```
val prod = array(10)({ i => i * 3 + 4 })
val cons = array(prod.length) { i => prod.at(i) * 5 + 6 }

// generated code without fusion:
val x5 = new Array[Int](10)
for (x1 <- 0 until 10) {                    // x1 is the producer index
  val x2 = x1 * 3
  val x3 = x2 + 4
  x5(x1) = x3                               // x3 is the producer result
}
val x6 = x5.length                          // matches SimpleDomain(x5)
val x12 = new Array[Int](x6)
for (x7 <- 0 until x6) {                    // x7 is the consumer index
  val x8 = x5.apply(x7)                     // matches SimpleIndex(x5, x7)
  val x9 = x8 * 5
  val x10 = x9 + 6
  x12(x7) = x10
}
```

1. The producer is mirrored, its fixed length is recorded, and the loop is recorded as seen in the current exact scope
2. The consumer body is traversed in search for producer candidates and `x5` is found
3. The producer and consumer satisfy the fusion conditions:
    V1. `prod (x5)` is a real producer because the shape of the consumer matches `SimpleDomain(x5)`
    V2. `cons` is independent of the producer apart from `SimpleIndex(x5, x7)`
    V3. neither is effectful
    V4. x5 is in the same exact scope
4. The transformer registers the following substitutions:
    - replace consumer index with producer index: `x7 -> x1`
    - replace producer access with producer block result: `x8 -> x3`
5. The transformer mirrors the loop, which causes the body to be mirrored too, and thus the index and the access are substituted. The transformer automatically adds a substitution from the old to the new consumer loop.

The generated code after vertical fusion:

if prod is used after, both loops survive and we need a fat loop computing both at once	if prod isn't used after, it is DCE'd

`val x21 = new Array[Int](10)`
`for (x1 <- 0 until 10) {  // fused`
`  val x2 = x1 * 3`
`  val x3 = x2 + 4`
`  val x18 = x3 * 5`
`  val x19 = x18 + 6`
`  x21(x1) = x19`
`}`
`val x5 = new Array[Int](10)`
`for (x1 <- 0 until 10) {  // prod`
`  val x2 = x1 * 3`
`  val x3 = x2 + 4`
`  x5(x1) = x3`
`}`
	`val x19 = new Array[Int](10)`
	`for (x1 <- 0 until 10) {`
	`  val x2 = x1 * 3`
	`  val x3 = x2 + 4`
	`  val x16 = x3 * 5`
	`  val x17 = x16 + 6`
	`  x19(x1) = x17`
	`}`

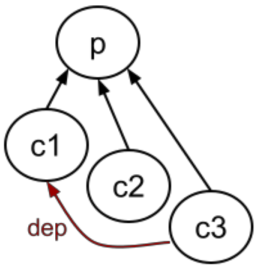
after horizontal and CombineTTPScheduling	

`val x21 = new Array[Int](10)`
`val x5 = new Array[Int](10)`
`for (x1 <- 0 until 10) {`
`  val x2 = x1 * 3`
`  val x3 = x2 + 4`
`  val x18 = x3 * 5`
`  val x19 = x18 + 6`
`  x21(x1) = x19`
`  x5(x1) = x3`
`}`
	no changes

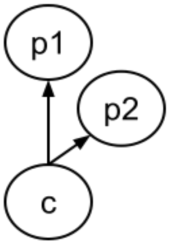
The statements highlighted come from the producer body. The transformer just substituted `x3` for `x8`, and then the scheduler puts them into the fused loop. The producer and the fused loop need to be combined, otherwise the statements are computed twice. This example shows a case where horizontal fusion doesn't need to change any symbols, because the producer and the combined loop already use the same index symbol and there are no effects that require mirroring.

6.5 Multiple Producers, Multiple Consumers

While it seems quite simple to fuse one producer with one consumer, there are cases where one producer is consumed by multiple consumers, or multiple producers are used to compute one consumer.



The first case is illustrated on the right. The algorithm starts by fusing the producer p with the consumer c1, after having done the effects and the dependency check. Then it reaches c2, and instead of just verifying p, the two checks are run between the set (p, c1) and c2, which pass successfully in this case. For c3 however, even though it is a consumer of p, it will not be fused because it depends on c1, so the final TTP created for all four loops would have a circular dependency. In general, p could itself have been a consumer and already fused with other loops, and as the next paragraph shows, any of the consumers might also be fused with other producers. This is why the dependency check runs between sets of loops instead of just single nodes.



The second case with multiple producers and one consumer is more interesting. This is the configuration of a zip, or it could occur in user code such as the following:

```
val p1 = array(100)        { i => i + 1 }
val p2 = array(100)        { j => j + 2 }
val c  = array(p1.length) { k => p1.at(k) + p2.at(k) + k }
val c' = array(100)        { k => p1.at(k) + p2.at(k) + k }
```

The two formulations of the consumer show that this case cannot be fused without considering reconstructed producers, because there can be at most one real producer. In the first case, `p1` is real and `p2` reconstructed, whereas in the second case both are reconstructed.

Vertical fusion usually replaces the SimpleIndex expression with the producer block and changes the consumer to use the producer index, such that the producer body will be scheduled into the combined loop. But which index variable should be used? Each producer has a different index variable. First I considered doing a horizontal fusion pass before vertical fusion, which would leave both producers with the same index variable and would also already check that they're independent. However, some horizontal fusions could prevent vertical fusions, as the following example shows:

```
val prod = array(100) { i => i + 1 }
val other = array(100) { i => i + 2 }
val cons = array(prod.length) { i => other.at(i + 1) + prod.at(i) }
```

The horizontal pass would fuse `prod` and `other`, but then `cons` couldn't be fused with `prod` because it depends on `other`, but without being a consumer. Vertical fusion needs to have the priority, since it potentially removes intermediate datastructures and is crucial for getting high performance when translating high-level functional code into low-level C-style code. So we would need to have a vertical fusion simulation pass, then a slimmer version of horizontal fusion that fuses only the loops determined by the simulation, and then finally run the vertical fusion, keeping track of what was pre-fused. That seems too complicated, and each additional transformer increases compilation times considerably.

What if we could horizontally fuse the two producers during the vertical transformation? The problem here is that they have already been mirrored, so if the transformer changed them again (even just to change the index variable), they would get a new symbol and the transformer would have to go back and re-mirror all statements that depend on them. Furthermore, there would need to be a way to replace the symbol in all internal datastructures of the transformer, which is not trivial in complexity.

Therefore, each loop has to be brought into its final shape when it is being transformed. So this means that those two producers need to have the same indices. The only conditions are that they are in the same exact scope and that they have the same fixed length. The transformer has a per-scope datastructure that maps from lengths to index variables and looks up the index entry when processing fixed-length loops. If there is already an index entry the transformer adds a substitution to that index, otherwise the index of the loop is entered into the map.

Note that there is no dependency check here to prevent loops from being assigned to the wrong set. If two loops depend on each other there would be two different sets, but then a third loop that doesn't depend on either might pick the wrong set, preventing further fusions. In fact, it isn't even necessary to check independence between multiple producers when fusing them with a consumer, because the transformer already checks that the consumer doesn't depend on anything in each producer set except for the producers themselves. This is sufficient, because if any loop in the set of producer A depended on a node b in the set of producer B, then the consumer would depend on b through A, and thus B wouldn't be a producer.

But couldn't there be a conflict if the loop is also a consumer and already has an index substitution? The answer is no, because if the loop has a fixed length and is a consumer, it needs to be a map over a producer with a fixed length. The producer is in the same scope by property V4. Therefore the producer has been mirrored to use the same fixed-length index symbol, and so the substitution from consumer index -> producer index is the same as from consumer index -> fixed-length index.

With this mechanism, a consumer can be fused with multiple producers, since all producers with the same fixed length now use the same index variable. The transformer substitutes the consumer index with any producer index, and then substitutes all accesses with the block results of the respective producers.

As a result, there might be multiple loops with the same index at the end of the vertical transformer. This isn't a problem, since the scheduler decision of what ends up in a loop body is based on the block result of the loop, as well as the index. It would only be a problem if a nested loop was assigned the same index as its outer loop, but that can't happen since loops only take indices from loops in the same exact scope. Loops might be scheduled differently after fusion, but only because an inner loop L previously depended on the index of an outer loop O, but fusion removed the dependency and L is now invariant and thus moved outside of O by code motion. So loops can travel to outer scopes, but they can't become nested in other loops. However, horizontal fusion might re-uniquify index variables in the future, so that only loops that are really fused have the same index. Again, this doesn't change the correctness of the program, but it will make it more readable, and will make life easier for anybody who is looking at the AST and trying to figure out which statement belongs to which loop.

In conclusion, vertical fusion can now handle fusing multiple consumers with multiple producers, where each consumer might itself be a producer and vice versa. This effectively partitions the set of loops, and the partition is communicated to horizontal fusion through CanBeFused.

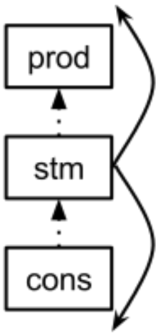
6.6 Transformer Structure

The skeleton of the vertical transformer looks similar to the horizontal transformer, even though the transformation is much more complex. It is also a preserving forward substitution transformer. The per-scope datastructures are the set of loops already seen in the current exact scope, and the map from fixed lengths to index symbols that is used to handle multiple producers. As seen before, the per-scope datastructure doesn't need to be combined according to CBF, each exact scope in the current schedule gets a new one. The fusion sets are now tracked globally since there is no searching through the list for candidates, a consumer can either be fused with the producer and its set or not, so a simple HashMap from loops to sets is sufficient.

`ReflectBlock` is responsible for saving the old per-scope datastructure object and creates a new one for each block, and upon returning resets it to the old one corresponding to the outer scope. The vertical transformer sometimes needs to substitute whole blocks, so `reflectBlock` first checks the substitution map and only traverses the block if it doesn't already have a replacement.

The next function is `transformStm`. It currently ignores loops that were already fused by SOA, but this will be changed in future work, once the transformer runs in a fixpoint fashion (see section 6.11). `TransformStm` replaces `SimpleDomain` nodes with fixed lengths if possible, and matches on `SimpleIndex` nodes to substitute them with producer block results whenever applicable. Lastly, it matches on loops and calls the vertical transformation function `transformLoop`.

All other statements are just mirrored. But couldn't there be statements in between a producer and a consumer that would prevent fusion? No, it is sufficient to do the dependency and the effects check. Remember that the effect system adds dependencies, but not all effects are serialized. When there is no dependency between two effectful nodes, then the scheduler is free to reorder them. If a producer and a consumer pass the two tests, then the scheduler will be able to move all statements in between them either before or after the fused loop.



No pure statement in between a producer and a consumer can prevent fusion:

- no dependency -> move before or after
- statement depends on prod, but cons doesn't depend on it -> move after
- consumer depends on it, but it doesn't depend on prod -> move before
- otherwise consumer depends on producer -> fail dep check

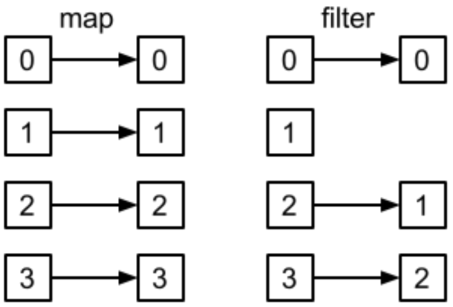
No effectful statement in between producer and consumer can prevent fusion:

- prod & cons have effects -> fail effect check
- consumer is pure
    - no dependency -> move after
    - statement depends on prod, but consumer doesn't depend on it -> move after
    - cons depends on statement, but it doesn't depend on producer -> move before
        - note that if the statement had same type of effect as prod, it depended on prod and thus cons depended on prod -> fail dep check
    - otherwise consumer depends on producer -> fail dep check
- producer is pure
    - no dependency -> move before
    - statement depends on prod, but consumer doesn't depend on it -> move after
        - note that if the statement had same type of effect as cons, then cons depended on it and thus cons depended on prod -> fail dep check
    - cons depends on statement, but it doesn't depend on producer -> move before
    - otherwise consumer depends on producer -> fail dep check

I structured the loop transformation itself into three parts: first is `findProducers`, which computes all possible producers for the given loop. Then comes `combineProducers`, which handles all the various types of loops and collects all the information necessary to do the fusion. This involves a lot of lookups and pattern matches that can fail, and some combinations of loops that cannot be fused. So the transformer prepares all the pieces at this stage, and since nothing has been fused yet it can still abort the fusion, print a debug message

and just mirror the loop. The next step then doesn't have to check anything. It is called `doFusion` and adds the necessary substitutions and effects depending on the type of loop. It also does the bookkeeping to record the fusion and the new fused loop symbol.

The `findProducer` function starts by traversing all statements in the body of the current loop (which is the consumer), matching on `SimpleIndex` nodes that access collections only at the loop index. Those potential producers are then checked against the seen loops to filter out those that are not in the same exact scope. Next, the range of the consumer loop is considered, and some candidates are discarded because the consumer doesn't iterate over their full range.



Then comes the dependency check explained in section 6.2. Depending on the type of the producer loop, it also verifies that the consumer doesn't use its index variable other than in the SimpleIndex expression. If the producer is a map function, then both the producer loop and the consumer loop iterate over the same range (from 0 to 3 in the diagram on the left). However, if the producer is a filter, flatMap or any other loop, then the consumer will not necessarily iterate over the same range. Since the combined loop iterates over the range of the producer, the consumer can only be fused if it doesn't depend on the values of the index, but only on the value of the producer at that index.

Last is the effects check. For multiple producers the real producer is moved to the head of the list, and `combineProducers` will choose as many of the valid producers as is possible without violating the effects condition (a pure consumer can only be fused with one effectful producer). The following sections will explain the extractors and the structure of the DSLs covered, before detailing the fusion rules implemented in `combineProducers`.

6.7 Filters, Yields and MultiCollects

The old loop fusion recognized two types of producers, `SimpleCollect` and `SimpleCollectIf`. The first one defines the collection through a function of the loop index, and the extractor returns the block result. `SimpleCollect` matches expressions such as creating a new array from a function or map operations, for example:

```
val arr1  = array(10) { i => i + 3 }                    // create new array
val arr2  = array(arr1.length) { i => arr1.at(i) + 4 }  // map
val arr2' = arr1.map({ x => x + 4 })                    // DSL with map op
```

The `SimpleCollectIf` extractor returns the block result representing the function and an additional condition to encode filter-like operations:

```
val arr1 = arrayIf(arr0.length)(
  { i => i % 2 == 0 },                          // condition
  { i => arr0.at(i) + 4 })                      // value
```

All operations that should be considered as consumers then had to contain a list of conditions as well as a collect-elem (a map function) that accessed the producer. Fusion would then call `applyAddCondition` to fuse with a `SimpleCollectIf` producer. For both types of producers it would replace the producer access with its block result in the consumer elem (similar to the new fusion). Here is an example:

```
val prod = arrayIf(10)({ i => i != 2 }, { i => i - 2 })
val cons = prod.fold(0.0, { (x,y) => x + 1.0/y })
```

The DSL has to encode the fold instruction in a loop with a node that has a collect-elem as well as a list of conditions. The consumer would conceptually look similar to this:

```
SimpleLoop(prod.length, indexVar,
  ArrayFoldElem(0, cons, elemVal,
    { indexVar => prod.at(indexVar) },              // the map function
    { indexVar => cons + 1.0/elemVal },             // the reduce functior
    Nil                                             // no conditions yet
))
```

The combined loop after fusion:

```
SimpleLoop(10, indexVar,
  ArrayFoldElem(0, cons, elemVal,
    { indexVar => indexVar - 2 },                   // fused map
    { indexVar => cons + 1.0/elemVal },
    List(indexVar != 2)                             // added condition
))
```

The problem with this approach becomes apparent in the generated code:

```
var cons = 0
for (indexVar <- 0 until 10) {
  val elemVal = indexVar - 2
  val res = cons + 1.0/elemVal                  // ouch, division by zero
  val condition = (indexVar != 2)
  if (condition) cons = res
}
```

`ElemVal` and `res` are computed regardless of the condition, which is inefficient in the best case and wrong in the worst case, because the consumer computation (`res`) might not be possible with the values filtered out by the producer. The example shows that at index `2` there will be a division by zero, which wouldn't happen in the non-fused version. And while a DSL could choose to generate filter nodes in this way and specify that the computation is executed regardless of the condition, through fusion this behavior is transmitted to consumers, and their consumers and so on. For effectful consumers in particular, this means that their effects are executed regardless of the condition, which certainly isn't the expected behavior. This bug caused a lot of problems with the old loop fusion, which were usually solved by turning off loop fusion altogether.

Why doesn't code generation emit the if before the computation? Unfortunately it's not that simple. The code generation can emit the condition and the if, but then when emitting `elemVal` and `res` the scheduler can't know that the condition is in scope. So there won't be any CSE between the condition computation and the rest. This is a problem, particularly when the consumer is itself a filter that uses the producer value, because the fused loop would then compute the producer twice for every index that passes the filter - once for the condition and once for `elemVal` - still not the desired behavior.

The underlying issue is that the if is only emitted by codegen, but isn't a part of the AST and thus the scheduler cannot do its usual magic on it. Materializing the `IfThenElse` node in the AST would solve the issue, but what goes in the else-branch? It can't be left out in a sea of nodes representation, because having no else-branch in a sequential program means "continue with the next instruction", but there is no next instruction in the AST. The `IfThenElse` is only alive if something alive depends on it. As seen before, this is either the case when its value is used (so both branches need to return a value) or when it is effectful (still needs a value, but it can be the Unit constant).

These two options suggest two very different DSL strategies: The effectful strategy is to expose yields in the AST, the pure strategy is to have a flatMap-based IR (intermediate representation). The yield approach has been described abstractly by Tiark Rompf in [5]. To implement it, a DSL would have to create effectful nodes initializing and then yielding into the output collection. Fusion would then match on the yields and replace them with a combined yield according to some simple rules found in the paper. We did not choose this approach for these reasons:

- It adds effects, whereas the previous representation was pure: introducing effects places additional constraints on the scheduler and makes it harder to separate an effectful loop from a pure loop that only has the yield-effect. Adding a new effect type could solve these problems, but would require widespread changes in the codebase. This is particularly important for the main use case of LMS, the parallelization and heterogeneous execution framework Delite. The element computations of parallel Delite loops are inherently unordered, so Delite would have to introduce a special treatment for the yield effects.

- While there is mutability in the generated code (a filter appends to an ArrayBuilder, a fold reads&writes to a variable), it was only introduced by codegen, under the responsibility of each DSL. But with a mutable collection IR, it might be tempting (and difficult to prevent) to use yield nodes outside of loops. This is a problem for Delite, which targets many different platforms without exposing the different architectures to the user. So the example of the ArrayBuilder doesn't make sense for a C-like execution environment, where Delite has to allocate and manage static buffers. A more restrictive IR is much easier to translate to different execution targets than the yield-based IR, where yields could appear anywhere in the code. Delite would have to do a lot of pattern matching to recognize the different operations.
- The third reason is that the yield-fusion requires TTPs, but section 4.2 already explained that transformers can't operate on TTPs without rearchitecting mirroring and changing most of the codebase. TTPs are required because for vertical fusion, the two loop bodies aren't just concatenated, but the consumer needs to be inlined into the producer at the place where the yields are happening. It's unclear how to do this with a separate producer and a combined loop, because we'd have the problem of the duplicate consumer effect explained in section 6.3. But even more importantly, the combined loop would have different symbols than the producer since there is a new chunk of code (a new dependency) in the middle of the producer code, which would prevent CSE from removing the producer in the final TPP.
- The last reason is that this approach was already taken and then abandoned by Vojin Jovanovic.

I found the alternative strategy of the flatMap-based IR, which gives a much more literal answer to the question of what to put in the else-branch: Nil. The new extractor is called a `MultiCollect`, because it collects multiple values per index. It matches flatMap-like operations. A flatMap with a singleton at each index is a map, a flatMap with an `IfThenElse` node with a singleton and an empty collection is a filter, and with another flatMap inside it is a flatMap. Thus `MultiCollect`, `Singleton` and `Empty` are the only extractors needed to cover map, filter and flatMap. This model seems general enough to capture most collect-like operations that could be represented with yields. It furthermore suggests a nice recursive decomposition of fusion.

This strategy requires some changes to the IR, but they are limited to the loop DSLs actually using fusion, instead of requiring changes to the whole codebase. And changing all nodes to contain a `MultiCollect` instead of a `SimpleCollect` is a prerequisite for fusing flatMap producers anyway. On the other hand, the list of conditions and implementation of `applyAddCondition` become obsolete. Code generation might be slightly more complex, because the map and filter operations that were translated into flatMap nodes probably shouldn't use intermediate datastructures in the generated code, so code generation needs to translate back and emit map and filter code. But overall, this design solves the problems with filter nodes without introducing effects or TTPs and with minimal effort for the DSL author.

6.8 List of Extractors

Here is a list of all the extractors currently known to fusion:

Name	Defs matched:	Returns
`SimpleIndex`	random access of collection at index	(collection, index)
`SimpleDomain`	length of collection	collection
`FixedDomain`	collection with fixed length	length
`EmptyColl`	empty collection	
`SingletonColl`	singleton collection of the value	block result (value)
`MultiCollect`	flatMap	block result (collection)
`ForLike`	for/foreach	block result (valFunc)
`ReduceLike`	reduce	block result (valFunc)

The last three lines match loop bodies. A reduce is defined by a value function and the reduce function. The value function computes a collection for each index, and the reduce function reduces the concatenation of these collections. A for/foreach loop doesn't return anything, so its value function directly processes each element and is always effectful. The value function of both nodes is the part that accesses the producer and needs to be fusable with `MultiCollect` producers, which is why it returns a collection rather than a single element.

In the example array DSL, a typical snippet with `MultiCollects` could look as follows:

```
1:  val arr = flatten(4) { i =>
2:    if (i > 5)
3:      emptyArray[Int]()
4:    else
5:      array(3) { j => i + j }
6:  }
```

```
2  TP(Sym(2),OrderingGT(Sym(1),Const(5)))
3  TP(Sym(4),EmptyArray(Sym(3)))
5  TP(Sym(6),IntPlus(Sym(1),Sym(5)))
5  TP(Sym(7),SingletonInLoop(Block(Sym(6)),Sym(5)))
5  TP(Sym(8),SimpleLoop(Const(3),Sym(5),MultiArrayElem(Block(Sym(7)))))
2  TP(Sym(9),IfThenElse(Sym(2),Block(Sym(4)),Block(Sym(8))))
1  TP(Sym(10),SimpleLoop(Const(4),Sym(1),MultiArrayElem(Block(Sym(9)))))
```

The extractors match in the following statements:

```
TP(Sym(4),EmptyArray(Sym(3))) match {
  case TP(_, EmptyColl) => ...

TP(Sym(7),SingletonInLoop(Block(Sym(6)),Sym(5))) match {
  case TP(_, SingletonColl(Sym(6))) => ...

TP(Sym(8),SimpleLoop(Const(3),Sym(5),MultiArrayElem(Block(Sym(7))))) match
  case TP(_, SimpleLoop(_,_,MultiCollect(Sym(7)))) => ...

TP(Sym(10),SimpleLoop(Const(4),Sym(1),MultiArrayElem(Block(Sym(9))))) match
  case TP(_, SimpleLoop(_,_,MultiCollect(Sym(9)))) => ...
```

The extractors can be separated into producer and consumer types. The producer types are expressions that return some kind of collection that can be iterated over, depending on the DSL this might be Arrays, Vectors, Buffers etc. Here it will just be denoted by `Coll[T]`. The producers can furthermore be separated into base and composed producers, the latter building collections from other collections. The composed producers are fused by recursively fusing their components. `IfThenElse` is also listed as a composed producer, because it is now part of the AST instead of being implicitly contained in the `SimpleCollectIf` node.

Base producers:

- `EmptyColl: Coll[T]`
- `SingletonColl(elem: T): Coll[T]`

Composed producers:

- `IfThenElse(cond: Boolean, thenP: Coll[T], elseP: Coll[T]): Coll[T]`
- `MultiCollect(body: Int => Coll[T]): Coll[T]`

Consumers:

- `MultiCollect(body: Int => Coll[T]): Coll[T]`
- `ForLike(valFunc: Int => Unit): Unit`
- `ReduceLike(valFunc: Int => Coll[V], reduceFunc: (V,V) => V) : V`

6.9 Fusion Rules

Not all producers can be fused with all consumers. The following shows how the different types are fused, provided all the conditions explained previously hold. The rules are implemented in `combineProducers`. The separation between `combineProducers` and the actual fusion in `doFusion` allows for the recursive deconstruction, since `combineProducers` can compute the inner fusion information and still abort if there's a problem along the way. ⊕ denotes fusion, p is the producer, c is the consumer, i is the loop index, s is the loop range, p(i) is the producer access at index i, p.l is the length of p (the range of the consumer loop).

Empty:

```
p=EmptyColl: Coll[T]  ⊕  c=Loop(p.l, i, MultiCollect(…)): Coll[V]
```

```
new empty collection of different type Coll[V], created by extractor
EmptyCollNewEmpty that DSL overrides to allow creation
```

```
p=EmptyColl: Coll[T]  ⊕  c=Loop(p.l, i, ForLike(…)): Unit
```

```
(): Unit
```

Note: `EmptyColl` is not fused with `ReduceLike` because the behavior is DSL-specific and should be preserved (throw exception, return zero element, return None).

Singleton:

```
p=SingletonColl(elem: T): Coll[T]
    ⊕  c=Loop(p.l, i, MultiCollect(f(i,p(i)): Coll[V])): Coll[V]
```

```
f(0, elem): Coll[V]
```

```
p=SingletonColl(elem: T): Coll[T]
    ⊕  c=Loop(p.l, i, ForLike(f(i,p(i)): Unit): Unit
```

```
f(0, elem): Unit
```

Note: `SingletonColl` is currently not fused with `ReduceLike` because it's unclear what the result should be. Some DSLs actually use a zero element, while others strip the first element to initialize the accumulator variable or use Option types. Depending on the valueFunc, a single producer element might still result in a collection that needs to be reduced.

IfThenElse:

```
p=IfThenElse(cond: Boolean, thenP: Coll[T], elseP: Coll[T]): Coll[T]
    ⊕  c=Loop(p.l, i, body): Coll[V]
```

```
IfThenElse(cond: Boolean, thenP ⊕ c, elseP ⊕ c): Coll[V]
```

Note: Currently, this fusion is only executed when at least one of the branches is `EmptyColl` and can be fused with the consumer (and thus the consumer disappears), because otherwise it can cause exponential code blowup. However, since only one branch is executed, runtime performance would benefit from fusion. Future work might investigate heuristics/ callbacks/annotations to fuse regardless.

MultiCollect:

```
p=Loop(s, i_p, MultiCollect(f_p(i_p): Coll[T])): Coll[T]
    ⊕  c=Loop(p.l, i_c, MultiCollect(f_c(p(i_c)): Coll[V])): Coll[V]
```

```
Loop(s, i_p, MultiCollect(
  x=f_p(i_p): Coll[T];
  x ⊕ Loop(x.l, i_c, MultiCollect(f_c(x(i_c))))))): Coll[V]
```

```
p=Loop(s, i_p, MultiCollect(f_p(i_p): Coll[T])): Coll[T]
    ⊕  c=Loop(p.l, i_c, ForLike(f_c(p(i_c)): Unit)): Unit
```

```
Loop(s, i_p, ForLike({
    x=f_p(i_p): Coll[T];
    x ⊕ Loop(x.l, i_c, ForLike(f_c(x(i_c))))})): Unit
```

```
p=Loop(s, i_p, MultiCollect(f_p(i_p): Coll[T])): Coll[T]
    ⊕  c=Loop(p.l, i_c, ReduceLike(valF(p(i_c)): Coll[V], redF: V)): V
```

```
Loop(s, i_p, ReduceLike({
    x=f_p(i_p): Coll[T];
    x ⊕ Loop(x.l, i_c, MultiCollect(valF(x(i_c))))
  }, redF: V)): V
```

The implementation of these rules required defining case classes to capture the information specific to each rule, with nested members for the recursive rules. `DoFusion` then adds the necessary substitutions, mirrors the correct definition and records the fusion.

6.10 If-Fusion

The need for fusing ifs horizontally is a direct consequence of materializing the `IfThenElse` nodes of filters in the AST and treating ifs as producers in vertical fusion. The following example of a filter consumed by a map shows that without if-fusion, the producer will be calculated twice in the generated code:

```
val arr0 = arrayIf(100)(
  { i => i > 10 },
  { i => i + 1 })
val arr1 = array(arr0.length)(
  { i => arr0.at(i) + 2 })
```

without if-fusion	with if-fusion

`for (x1 <- 0 until 100) {`
`  val x2 = x1 > 10`
`  if (x2) {`
`  val x3 = x1 + 1  // duplicate!`
`  x8_builder += x3`
`  }`
`  if (x2) {`
`  val x3 = x1 + 1  // duplicate!`
`  val x23 = x3 + 2`
`  x26_builder += x23`
`  }`
`}`
	`for (x1 <- 0 until 100) {`
	`  val x2 = x1 > 10`
	`  if (x2) {`
	`  val x3 = x1 + 1`
	`  val x23 = x3 + 2`
	`  x8_builder += x3`
	`  x26_builder += x23`
	`  }`
	`}`

6.11 Multi-Pass or Fixpoint

Vertical fusion achieves most fusions in the first pass. Since optimizations that combine scopes like horizontal fusion cannot create more vertical fusion opportunities, it is unnecessary to rerun vertical fusion after other passes. However, in a few rare cases there needs to be more than one pass of vertical fusion. This happens when successive SimpleIndex operations could be fused, or when a loop changes scope as a consequence of fusion and now satisfies condition V4 as a consumer.

In the following example, the first fusion pass will substitute the first `SimpleIndex`, and the second pass could fuse the inner arrays:

```
val range = array(100) { i => array(i) { j => j + 1 } }
val range2 = array(100) { k => array(k) { l => range.at(k).at(l) } }
```

↓

```
val range2 = array(100) { i =>
  val x = array(i) { j => j + 1 }
  array(i) { l => x.at(l) }
}
```

↓

```
val range2 = array(100) { i => array(i) { j => j + 1 } }
```

In the second case, a nested loop moves out of a loop when it doesn't depend on the outer loop variable anymore. Each vertical fusion removes a dependency on the loop index (namely the access of the producer at the index). The following example shows how fusing `a1` and `a3` causes the inner array to become independent of `i`, so it could now be fused with `a2`:

```
val a1 = array(100) { i => 1 }
val a2 = array(100) { i => i + 2 }
val a3 = array(a1.length) { i => array(100) { j => a1.at(i) + a2.at(j) } }
```

↓

```
val a2 = array(100) { i => i + 2 }
val x  = array(100) { j => 1 + a2.at(j) }  // inner loop fused and moved ou
val a3 = array(100) { i => x }
```

↓

```
val x  = array(100) { i => i + 2 + 1 }
val a3 = array(100) { i => x }
```

The current implementation only does a single pass, but future work will investigate whether it's possible to reliably detect when to re-run vertical fusion. In theory, everything could be fused in the first pass. For the first case, the transformer would have to do the second fusion based on the information computed by `combineProducers` for the first one. The second case requires a stack of outer loop indices and seen loops, so that when `combineProducers` removes an index dependency the transformer could check whether it was the last one and which scope the loop would move to. But since these are corner cases, the handful of times it would result in shorter compile times probably isn't worth the effort (and increased code complexity) to implement one-pass vertical fusion, and future work will just implement the detection. The transformer could also have a flag to run in a fixpoint fashion until nothing can be fused anymore, for the few programs where these edge cases matter.

7. Future Work

The next step is to integrate the new loop fusion into Delite, which requires changing the parallel Delite loops to the `MultiCollect` design and will allow us to test the new loop fusion on all the existing applications. I wrote over 60 test cases as part of the development process, but they are all short snippets, whereas the Delite applications will have much bigger ASTs and will give us more insights into performance hotspots and priorities for future work.

The previous chapters already showed the following ideas for future work:

- Improve CBF set management: symbol information instead of only Def (5.5)
- Scoped dependency analysis pass for both fusion transformers (5.6, 6.2)
- Multi-pass or fixpoint vertical fusion (6.11)

Loop fusion currently does not fuse if any of execution time, memory space or code size would increase. However apps should be able to choose to make these trade offs, either statically through annotations, through callback functions, hooks exposed in the performance debugger or through an autotuning tool. In the simplest case it would mark loops that should or shouldn't be fused (5.4). Section 6.9 mentioned pushing consumers into both branches of an `IfThenElse` producer. This duplicates consumer code, leading to an exponential growth in the pathological case. But it does not incur any time or space penalty and might enable further optimizations, in particular the producer will not be allocated if it is not used other than by the consumer, saving memory space.

Fusion of larger read stencils is another example. When a consumer uses prod.at(i) and prod.at(i+1), naively replacing both with the producer computation causes each producer element to be computed twice, while possibly saving the memory space of the producer. Some applications might want to make this tradeoff, for example when the collection is large and garbage collection expensive, but the computation quick. Since this changes cache behavior, it will be difficult to define an accurate cost-model to make the decision automatically, but it would be interesting to offer this flexibility. The redundant computation could also be avoided at the cost of adding a mutable variable and effects to memoize the previously computed element.

A more elaborate stencil analysis could also enable concatMap-fusion:

```
(a ++ b).map(f)
// in loop notation:
for (i <- 0 until a.size + b.size) {
  val x = if (i < a.size) a(i) else b(i-a.size)
  append(f(x))
}
```

In general, fusing producers that are accessed at a different index requires mirroring the producer to the other index, which will prevent horizontal fusion. But it could still be done if the producer isn't used anywhere else or the duplication is ok. The loop range check could also be ignored if horizontal fusion isn't required and the producer is pure. Section 6.6 explained that a consumer isn't allowed to use its index variable (apart from the producer access) when the producer isn't a map function. This restriction could also be lifted if a variable was used to track the consumer index.

Another relaxation of the conditions would be to vertically fuse loops from different scopes (6.4), which can cause performance overhead if a producer is suddenly nested in a loop and computed repeatedly, or a consumer is moved out of its conditional. But the benefits of removing the intermediate datastructure and other fusion-enabled optimizations might be worth it in some cases, as long as the program stays correct.

In terms of supported operations, future work will implement groupBy-like operations, which are called *bucket* operations in the fusion paper [5]. The general idea is that there is a way to put elements into buckets according to a key function, and then either return them in a nested collection (bucket-collect) or reduce each bucket to a value and return the collection of values (bucket-reduce). The fusion rules will look similar to flatMap and reduce.

The Delite team is also interested in a way to transform between different bucket types, in particular for this case:

```
bucketCollect(keyFunc, valFunc).flatMap({ arr => foo(arr.reduce(+)) })
```

↓

```
bucketReduce(keyFunc, valFunc, +).flatMap(foo)
```

This will probably be implemented as a transformer pass specific to Delite rather than a generic loop fusion transformation, since it needs to create a new `bucketReduce` node of the right kind. Ideally it would be easy to add more rules, one idea is to express everything in a fission-fusion model.

A large part of the difficulty of this project was to completely understand mirroring and the current effect system, which are poorly documented apart from the papers where the ideas were originally proposed. In this present thesis I've attempted to capture the knowledge I gained and needed to implement the project, and I would like to make it accessible to the community, probably in the form of a tutorial and better documentation.

8. Conclusion

The initial reasons for implementing another loop fusion algorithm were to cover more operations (in particular flatMap) and to use the transformer architecture. My first implementation used the same extractors as the old fusion, but when the problems with effects and filters/ifs became apparent, it was clear that a new design was necessary. The new loop fusion has the flatMap/MultiCollect as basic unit, whereas the old one was built around map/SimpleCollect. This design is flexible enough to express all the operations and fusion rules required, without the need to expose effectful yields as originally proposed.

It can now fuse flatMap operations, and correctly handles effects and filters by manifesting the `IfThenElse` node in the AST. It is implemented as transformers as far as possible, and only the very final combination into TTPs is handled by a scheduler extension, but no symbols are changed in the last step. The CanBeFused trait is a clean interface that allows other transformers and analysis passes to take fused loops and ifs into account, and do their own fusion. The decomposition into vertical, horizontal and TTP fusion allows for a clear separation of concerns. The transformers use the mirroring functionality instead of switching out nodes in the scheduler.

While the old fusion ran in a fixpoint fashion, only the vertical transformer might need several passes, and only in rare edge cases. It can fuse multiple producers and multiple consumers in one pass. The horizontal transformer runs in a single pass, taking into account the fusion requirements from vertical fusion, SOA and possibly future optimizations requiring fusion. The old loop fusion required multiple passes if loops could be horizontally fused as a result of fusing two outer loops and thus combining their scopes.

This project required all aspects of software engineering. I had to gain a deep understanding of the LMS framework and codebase in order to handle effects, use the full power of mirroring and structure the algorithms as forward transformers. I evaluated the different strategies shown in the loop fusion paper [5], the existing implementation and Vojin Jovanovic's prior attempts at exposing yields. Through many discussions with the Delite team I gathered requirements and priorities.

The quote "Plan to throw one away; you will, anyhow" from Fred Brooks' The Mythical Man Month also applied to this project. The first implementation used the same extractors and DSL structure as the old loop fusion, while using transformers instead of being baked into the scheduler. Through countless tests I slowly gained an understanding of what went wrong with effects and filters and came up with the idea of the MultiCollect fusion. I discussed the pros and cons of MultiCollects vs. yields with the Delite team, and we decided on the MultiCollect solution since it didn't introduce effects and didn't expose mutable collections to the user. Much of the effort required from a DSL author is necessary anyway to fuse flatMap producers.

I added a planning phase before starting the second implementation and went through all the fusion cases, determining how each could be solved. This lead to the recursive deconstruction algorithm and the architecture of the vertical transformer with the separation between computing information for fusion and actually adding the substitutions and mirroring the loops. Thanks to the separation, there was no need to implement a rollback mechanism in case inner fusions failed. After this design phase came the final implementation of the `MultiCollect` fusion. It reused some parts like the dependency and effects check, but with new fusion rules and fusion sets.

In conclusion, this project required a variety of engineering skills and allowed me to develop a complex compiler optimization. I learned a lot about the LMS ecosystem, code generation, domain specific languages and loop fusion itself. I would like to thank the Delite group at Stanford and in particular Kevin Brown and HyoukJoong Lee for the great design discussions we had and their patient explanations of the LMS and Delite worlds. Many thanks also go to Prof. Kunle Olukotun, Stanford University, the PPL (pervasive parallelism lab) group, EPFL and Darlene Hadding for making my stay as a visiting student researcher at Stanford possible. And last but not least, I would like to thank Prof. Martin Odersky for being my EPFL supervisor and Tiark Rompf for the discussions and answers to LMS and administrative questions.

9. References

[1] T. Rompf, M. Odersky: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs (GPCE 2010)

[2] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, K. Olukotun: Implementing Domain-Specific Languages for Heterogeneous Parallel Computing (IEEE 2011)

[3] D. Coutts, R. Leshchinskiy, D. Stewart: Stream Fusion: From Lists to Streams to Nothing at All (ICFP 2007)

[4] O. Kiselyov: Iteratees

[5] T. Rompf, A. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, M. Odersky: Optimizing Data Structures in High-Level Programs (POPL 2013)

[6] Chvátal, Václav (1984), "Perfectly orderable graphs", in Berge, Claude; Chvátal, Václav, Topics in Perfect Graphs, Annals of Discrete Mathematics 21, Amsterdam: North-Holland, pp. 63–68