Fusing Filters with Integer Linear Programming

Amos Robinson[†] Ben Lippmeier[†] Gabriele Keller[†]

[†]Computer Science and Engineering
University of New South Wales, Australia
{amosr,benl,keller}@cse.unsw.edu.au

Abstract

The key to compiling functional, collection oriented array programs into efficient code is to minimise memory traffic. Simply fusing subsequent array operations into a single computation is not sufficient; we also need to cluster *separate* traversals of the same array into a single traversal. Previous work demonstrated how Integer Linear Programming (ILP) can be used to cluster the operators in a general data-flow graph into subgraphs, which can be individually fused. However, these approaches can only handle operations which preserve the size of the array, thereby missing out on some optimisation opportunities. This paper addresses this shortcoming by extending the ILP approach with support for size-changing operations, using an external ILP solver to find good clusterings.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; Optimization

General Terms Languages, Performance

Keywords Arrays, Fusion, Haskell

1. Introduction

A collection-oriented programming model is ideal for expressing data-parallel computations, as it exposes the communication patterns to the compiler without requiring complicated loop analysis. It is essential, however, that the compiler combines these operations into a small number of efficient loops, as a naive translation leads to high memory traffic and poor performance. This is a well known problem, and techniques to fuse subsequent array operations together have been presented in [5, 9, 10], to name just a few. This type of fusion alone is not generally sufficient to minimise memory traffic. As shown in Megiddo [13] and Darte [7], Integer Linear Programming can be used to find good clusterings. Unfortunately, they cannot handle operations like filter, where the output size differs from the input size. We present a technique that can handle both multi-loop fragments as well as size-altering operations.

To compile the clusters found by our clustering technique into sequential loops, we use data flow fusion [12]. It improved on existing array fusion approaches [5, 10] as it guarantees fusion into a single loop for programs that operate on the same size input data and contain no fusion-preventing dependencies between operators.

To see the effect of clustering, consider the following program:

```
normalize2 :: Array Int -> (Array Int, Array Int)
normalize2 xs
 = let sum1 = fold   (+)  0   xs
       gts  = filter (>   0)  xs
       sum2 = fold   (+)  0   gts
       ys1  = map    (/ sum1) xs
       ys2  = map    (/ sum2) xs
   in (ys1, ys2)
```

The function `normalize2` computes two sums, one of all the elements of `xs`, the other of only elements greater than zero. Since we need to fully evalute the sums to proceed with the maps, it is clear that we need at least two separate loops. These folds are examples of fusion-preventing dependencies, as fold cannot be fused with subsequent operations. Figure 1 shows the data-flow graph of `normalize2`. In the leftmost diagram we can see the effect of applying stream fusion to the program: we end up with four loops (denoted by the dotted lines): only the filter operation is combined with the subsequent fold. The best existing ILP approach results in the rightmost graph: it combines the `sum1` fold and `sum2` filter in one loop, but requires an extra loop for the fold operation which consumes the filter output, since it cannot fuse filter operations. Our approach, in the middle, produces the optimal solution in this case: one loop for the sums, another for the maps.

Our contributions are as follows:

- We extend prior work by Megiddo [13] and Darte [7], with support for size changing operators. Size changing operators can be clustered with operations on both their source array and output array, and compiled naturally with data-flow fusion (§4).
- We present a simplification to constraint generation that is also applicable to some ILP formulations such as Megiddo's: constraints between two nodes need not be generated if there is a fusion-preventing path between the two (§4.5).
- Our constraint system encodes a total ordering on the cost of clusterings, expressed using weights on the integer linear program. For example, we encode that memory traffic is more expensive than loop overheads, so given a choice between the two, memory traffic will be reduced (§4.4).
- We present benchmarks of our algorithm applied to several common programming patterns. Our algorithm is complete and yields good results in practice, though an optimal solution is uncomputable in general (§5).

An implementation of our clustering algorithm is available at `https://github.com/amosr/clustering`.

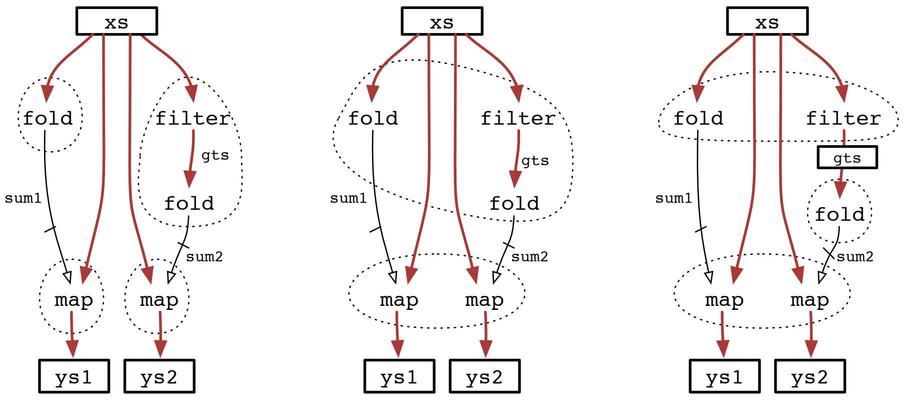

Figure 1. Clusterings for normalize2 example: with stream fusion; our system; best imperative system

2. Combinator Normal Form

Input programs are expressed in *Combinator Normal Form* (CNF), which is a textual description of the data flow graph. The grammar for CNF is given in Figure 2. The `normalize2` example on the previous page is in CNF, as is the matching data flow graph for `normalize2` in Figure 1. Our data flow graphs are similar to Loop Communication Graphs (LCGs) from related work in imperative array fusion [8]. We name edges after the corresponding variable from the CNF form, and edges which are fusion preventing are drawn with a dash through them (as per the edge labeled `sum1` in Figure 1). In data flow graphs, we tend to elide the worker functions to combinators when they are not important to the discussion — so we don't show the `(+)` operator on each use of `fold`.

Clusters of operators that are fused into single imperative loops are indicated by dotted lines, and we highlight materialized arrays by drawing them in boxes. In Figure 1, the variables `xs`, `ys1` and `ys2` are always in boxes, as these are the material input and output arrays of the program. However, in the graph on the far right hand side, `gts` has also been materialized because in this version, the producing and consuming operators (`filter` and `fold`) have not been fused. In Figure 2, note that the bindings have been split into those that produce scalar values (*sbind*), and those that produce array values (*abind*). These groupings are represented as open and closed arrow-heads in Figure 1.

Most of our array combinators are standard, and suggestive types are given at the bottom of Figure 2. The $\texttt{map}_n$ combinator takes a worker function, n arrays of the same length, and applies the worker function to all elements at the same index. As such, it is similar to Haskell's `zipWith`, with an added length restriction on the argument arrays. The `generate` combinator takes an array length and a worker function, and creates a new array by applying the worker to each index. The `gather` combinator takes an array of elements, an array of indices, and produces the array of elements that are positioned at each index. In Haskell, this would be `gather arr ixs = map (index arr) ixs`. The `cross` combinator returns the cartesian product of two arrays.

The exact form of the worker functions is left unspecified, as it is not important for the discussion. We assume workers are pure, can at least compute arithmetic functions of their scalar arguments, and index into arrays in the environment. We also assume that each CNF program considered for fusion is embedded in a larger host program which handles file IO and the like. Workers are additionally restricted so they can only directly reference the *scalar* variables bound by the local CNF program, though they may reference array variables bound by the host program. All access to locally

$$\begin{array}{lll}
\mathit{scalar} & \to & \text{(scalar variable)} \\
\mathit{array} & \to & \text{(array variable)} \\
f & \to & \text{(worker function)} \\
\mathit{fun} & \to & f\ \mathit{scalar}\ldots
\end{array}$$

$$\begin{array}{llll}
\mathit{bind} & ::= & \mathit{scalar} & = \mathit{sbind} \\
 & \mid & \mathit{array} & = \mathit{abind} \\
 & \mid & \mathit{scalar}\ldots,\mathit{array}\ldots & = \texttt{external}\ \mathit{scalar}\ldots\ \mathit{array}\ldots
\end{array}$$

$$\begin{array}{lllll}
\mathit{sbind} & ::= & \texttt{fold} & \mathit{fun}\ \mathit{array} & \\
\mathit{abind} & ::= & \texttt{map}_n & \mathit{fun}\ \mathit{array}^n & \mid\ \texttt{filter}\ \ \mathit{fun}\ \mathit{array} \\
 & \mid & \texttt{generate} & \mathit{scalar}\ \mathit{fun} & \mid\ \texttt{gather}\ \ \mathit{array}\ \mathit{array} \\
 & \mid & \texttt{cross} & \mathit{array}\ \mathit{array} &
\end{array}$$

$$\begin{array}{lll}
\mathit{program} & ::= & f\ \mathit{scalar}\ldots\ \mathit{array}\ldots\ = \\
 & & \texttt{let}\ \mathit{bind}\ldots \\
 & & \texttt{in}\ (\mathit{scalar}\ldots,\ \mathit{array}\ldots)
\end{array}$$

$$\begin{array}{ll}
\texttt{fold} & : (a \to a \to a) \to \texttt{Array}\ a \to a \\
\texttt{map}_n & : (\{a_i \to\}^{\,i \leftarrow 1\ldots n}\ b) \to \{\texttt{Array}\ a_i \to\}^{\,i \leftarrow 1\ldots n}\ \texttt{Array}\ b \\
\texttt{filter} & : (a \to \texttt{Bool}) \to \texttt{Array}\ a \to \texttt{Array}\ a \\
\texttt{generate} & : \texttt{Nat} \to (\texttt{Nat} \to a) \to \texttt{Array}\ a \\
\texttt{gather} & : \texttt{Array}\ a \to \texttt{Array}\ \texttt{Nat} \to \texttt{Array}\ a \\
\texttt{cross} & : \texttt{Array}\ a \to \texttt{Array}\ b \to \texttt{Array}\ (a,b)
\end{array}$$

Figure 2. Combinator normal form

bound array variables is via the formal parameters of array combinators, which ensures that all data dependencies we need to consider for fusion are explicit in the data flow graph.

The `external` binding invokes a host library function that can produce and consume arrays, but not be fused with other combinators. All arrays passed to and returned from host functions are fully materialised. External bindings are explicit *fusion barriers*, which force arrays and scalars to be fully computed before continuing.

Finally, note that `filter` is only one representative size changing operator. We can handle more complex functions such as `unfold` in our framework, but we stick with simple filtering to aid the discussion.

3. Size Inference

Before performing fusion proper, we must infer the relative sizes of each array in the program. We achieve this with a simple constraint based inference algorithm, which we discuss in this section. Size

Size Type	τ	$::=$	k	(size variable)
		$\mid$	$\tau \times \tau$	(cross product)
Size Constraint	C	$::=$	*true*	(trivially true)
		$\mid$	$k = \tau$	(equality constraint)
		$\mid$	$C \wedge C$	(conjunction)
Size Scheme	σ	$::=$	$\forall \overline{k}.\ \exists \overline{k}.\ (\overline{x:\tau}) \to (\overline{x:\tau})$	

Figure 3. Sizes, Constraints and Schemes

inference has been previously described in the context of array fusion by Chatterjee [4]. In constrast to our algorithm, [4] does not support size changing functions such as filter. If size inference fails, the programs may still be compiled, but fusion is not performed.

Although our constraint based formulation of size inference is reminiscent of type inference for HM(X) [14], there are important differences. Firstly, our type schemes include existential quantifiers, which express the fact that the sizes of arrays produced by filter operations are unknown in general. This is also the case for `generate`, where the result size is data dependent. HM(X) style type inferences use the $\exists$ quantifier to bind local type variables in constraints, and existential quantifiers do not appear in type schemes. Secondly, our types are first order only, as program graphs cannot take other program graphs as arguments. Provided we generate the constraints in the correct form, solving them is straightforward.

3.1 Size Types, Constraints and Schemes

Figure 3 shows the grammar for size types, constraints and schemes. A size scheme is like a type constraint from Hindley-Milner type systems, except that it only mentions the size of each input array, instead of the element types as well.

A size may either be a variable k or a cross product of two sizes. We use the latter to represent the result size of the `cross` operator discussed in the previous section. Constraints may either be trivially *true*, an equality $k = \tau$, or a conjunction of two constraints $C \wedge C$. We refer to the trivially true and equality constraints as *atomic constraints*. Size schemes relate the sizes of each input and output array. For example, the size scheme for the `normalize2` example from Figure 1 is as follows:

$$\texttt{normalize2} :_s \forall k.(xs : k) \to (ys_1 : k,\ ys_2 : k)$$

We write $:_s$ to distinguish size schemes from type schemes.

The existential quantifier appears in size schemes when the array produced by a filter or similar operator appears in the result. For example:

```
filterLeft :s ∀k1.∃k2.(xs : k1) → (ys1 : k1, ys2 : k2)
filterLeft xs
  = let ys1 = map (+ 1)   xs
        ys2 = filter even xs
    in (ys1, ys2)
```

The size scheme of `filterLeft` shows that it works for input arrays of all sizes. The first result array has the same size as the input, and the second has some unrelated size.

Finally, note that size schemes form but one aspect of the type information that would be expressible in a full dependently typed language. For example, in Coq or Agda we could write something like:

```
filterLeft : ∀k1:Nat.∃k2:Nat.
  Array k1 Float → (Array k1 Float, Array k2 Float)
```

However, the type inference systems for fully higher order dependently typed languages typically require quantified types to be provided by the user, and do not perform the type generalization process. In our situation, we need automatic type generalization, but for a first order language only.

3.2 Constraint Generation

The rules for constraint generation are shown in Figure 4. The top level judgment $program :_s \sigma$ assigns a size scheme to a program. It does this by extracting size constraints and then solving them. This rule, along with the constraint solving process is discussed in the next section. The judgment $\Gamma_1 \mid zs \vdash b \rightsquigarrow \Gamma_2 \vdash C$ reads: "Under environment Γ_1, array variable zs binds the result of b, producing a result environment Γ_2 and size constraints C". The remaining judgment that extracts constraints from a list of bindings is similar. The environment Γ has the following grammar:

$$\Gamma ::= \cdot \mid \Gamma, \Gamma \mid zs : k \mid k \mid \exists k$$

As usual, $\cdot$ represents the empty environment and Γ, Γ environment concatenation. The element $zs : k$ records the size k of some array variable zs. A plain k indicates that k can be unified with other size types when solving constraints, whereas $\exists k$ indicates a *rigid* size variable that cannot be unified with other sizes. We use the $\exists k$ syntax because this variable will also be existentially quantified if it appears in the size scheme of the overall program.

Note that the constraints are generated in a specific form, to facilitate the constraint solving process. For each array variable in the program, we generate a new size variable, like size k_{zs} for array variable zs. These new size variables always appear on the *left* of atomic equality constraints. For each array binding we may also introduce unification or rigid variables, and these appear on the *right* of atomic equality constraints.

For example, the final environment and constraints generated for the `normalize2` example from Section 1 are as follows:

$$\begin{array}{ll} & x : k_{xs},\ gts : k_{gts},\ \exists k_1,\ k_2,\ k_3 \\ \vdash & true \wedge k_{gts} = k_1 \wedge true \\ & \quad \wedge\ k_{xs} = k_2 \wedge k_{ys1} = k_2 \wedge k_{xs} = k_3 \wedge k_{ys2} = k_3 \end{array}$$

Rule (SProgram) also characterises the programs we accept: a program is *valid* if and only if $\exists \sigma.\ program :_s \sigma$.

3.3 Constraint Solving and Generalization

The top-level rule in Figure 4 assigns a size scheme to a program by first extracting size constraints, before solving them and generalizing the result. In the rule, the solving process is indicated by SOLVE, and takes an environment and a constraint set, and produces a solved environment and constraint set. As the constraint solving process is both standard and straightforward, we only describe it informally.

Recall from the previous section that in our generated constraints all the size variables named after program variables are on the left of atomic equality constraints, while all the unification and existential variables are on the right. To solve the constraints, we keep finding pairs of atomic equality constraints where the same variable appears on the left, unify the right of both of these constraints, and apply the resulting substitution to both the environment and original constraints. When there are no more pairs of constraints with the same variable on the left, the constraints are in solved form and we are finished.

During constraint solving, all unification variables occuring in the environment can have other sizes substituted for them. In contrast, the rigid variables marked by the $\exists$ symbol cannot. For example, consider the constraints for `normalize2` mentioned before:

$$\begin{array}{ll} & x : k_{xs},\ gts : k_{gts},\ \exists k_1,\ k_2,\ k_3 \\ \vdash & true \wedge k_{gts} = k_1 \wedge true \\ & \quad \wedge\ k_{xs} = k_2 \wedge k_{ys1} = k_2 \wedge k_{xs} = k_3 \wedge k_{ys2} = k_3 \end{array}$$

Note that k_{xs} is mentioned twice on the right of an atomic equality constraint, so we can substitute k_2 for k_3. Eliminating the duplicates, as well as the trivially *true* terms then yields:

$$\begin{array}{ll} & x : k_{xs},\ gts : k_{gts},\ \exists k_1,\ k_2 \\ \vdash & k_{gts} = k_1 \wedge k_{xs} = k_2 \wedge k_{ys1} = k_2 \wedge k_{ys2} = k_2 \end{array}$$

To produce the final size scheme, we look up the sizes of the input and output variables of the original program from the solved constraints and generalize appropriately. This process is determined by the top-level rule in Figure 4. In this case, no rigid size variables appear in the result, so we can universally quantify all size variables.

$$\texttt{normalize2} :_s \forall k.(xs : k) \to (ys_1 : k,\ ys_2 : k)$$

3.4 Rigid Sizes

When the environment of our size constraints contains rigid variables (indicated by $\exists k$), we introduce existential quantifiers instead of universal quantifiers into the size scheme. Consider the `filterLeft` program from Section 3.1

```
filterLeft xs
  = let ys1 = map (+ 1)   xs
        ys2 = filter even xs
    in (ys1, ys2)
```

The size constraints for this program, already in solved form, are as follows.

$$\begin{array}{ll} & xs : k_{xs},\ ys_1 : k_{ys1},\ \exists k_1,\ ys_2 : k_{ys2},\ k_2 \\ \vdash & k_{ys_1} = k_1 \wedge k_{ys_2} = k_2 \wedge k_{xs} = k_2 \end{array}$$

As variable k_2 is marked as rigid, we introduce an existential quantifier for it, producing the size scheme stated earlier:

$$\texttt{filterLeft} :_s \forall k_1.\exists k_2.(xs : k_1) \to (ys_1 : k_1, ys_2 : k_2)$$

Note that, although Rule (SProgram) from Figure 4 performs a *generalisation* process, there is no corresponding instantiation rule. The size inference process works on the entire graph at a time, and there is no mechanism for one operator to invoke another. To say this another way, all subgraphs are fully inlined. Recall from §2, that we assume our operator graphs are embedded in a larger host program. We use size information to guide the clustering process, and although the host program can certainly call the operator graph, static size information does not flow across this boundary.

When producing size schemes, we do not permit the arguments of an operator graph to have existentially quantified sizes. This restriction is necessary to reject programs that we cannot statically guarantee will be well sized. For example:

```
bad1 xs  = let flt   = filter p xs
               ys    = map2   f flt xs
           in  ys
```

The above program filters its input array, and then applies `map2` to the filtered version as well as the original array. As the `map2` operators requires both of its arguments to have the same size, `bad1` would only be valid when the predicate `p` is always true. The size constraints are as follows:

$$\begin{array}{ll} & xs : k_{xs},\ flt : k_{flt},\ \exists k_1,\ ys : k_{ys},\ k_2 \\ \vdash & k_{flt} = k_1 \wedge k_{flt} = k_2 \wedge k_{xs} = k_2 \wedge k_{ys} = k_2 \end{array}$$

Solving this then yields:

$$\begin{array}{ll} & xs : k_{xs},\ flt : k_{flt},\ \exists k_1,\ ys : k_{ys},\ k_1 \\ \vdash & k_{flt} = k_1 \wedge k_{xs} = k_1 \wedge k_{ys} = k_1 \end{array}$$

In this case, Rule (SProgram) does not apply, because the parameter variable xs has size k_1, but k_1 is marked as rigid in the environment (with $\exists k_1$).

As a final example, the following program is ill-sized, because the two filter operators are not guaranteed to produce the same number of elements.

```
bad2 xs = let as  = filter p1 xs
              bs  = filter p2 xs
              ys  = map2   f  as bs
          in  ys
```

The initial size constraints for this program are:

$$\begin{array}{ll} & xs : k_{xs},\ as : k_{as},\ \exists k_1,\ bs : k_{bs},\ \exists k_2,\ ys : k_{ys},\ k_3 \\ \vdash & k_{as} = k_1 \wedge k_{bs} = k_2 \wedge k_{as} = k_3 \wedge k_{bs} = k_3 \wedge k_{ys} = k_3 \end{array}$$

To solve these, we note that k_{as} is used twice on the left of an atomic equality constraint, so we substitute k_1 for k_3:

$$\begin{array}{ll} & xs : k_{xs},\ as : k_{as},\ \exists k_1,\ bs : k_{bs},\ \exists k_2,\ ys : k_{ys},\ k_1 \\ \vdash & k_{as} = k_1 \wedge k_{bs} = k_2 \wedge k_{bs} = k_1 \wedge k_{ys} = k_1 \end{array}$$

At this stage we are stuck, because the constraints are not yet in solved form, and we cannot simplify them further. Both k_1 and k_2 are marked as rigid, so we cannot substitute one for the other and produce a single atomic constraint for k_{bs}.

3.5 Iteration Size

After inferring the size of each array variable, each operator is assigned an *iteration size*, which is the number of iterations needed in the loop which evaluates that operator. For `filter` and other size changing operators, the iteration and result sizes are in general different. For such an operator, we say that the result size is a *descendant* of the iteration size. Conversely, the iteration size is a *parent* of the result size.

This descendant–parent size relation is transitive, so if we filter an array, then filter it again, the size of the result is a descendant of the iteration size of the initial filter. This relation arises naturally from Data Flow Fusion [12], as such an operation would be compiled into a single loop — with an iteration size identical to the size of the input array, and containing two nested if-expressions to perform the two layers of filtering.

Iteration sizes are used to decide which operators can be fused with each other. As in prior work, operators with the same iteration size can be fused. However, in our system we also allow operators of different iteration sizes to be fused, provided those sizes are descendants of the same parent size.

We use T to range over iteration sizes, and write $\perp$ for the case where the iteration size is unknown. The $\perp$ size is needed to handle the `external` operator, as we cannot statically infer its true iteration size, and it cannot be fused with any other operator.

$$\begin{array}{llll} \textbf{Iteration Size} & T & ::= \tau & \text{(known size)} \\ & & \mid\ \ \perp & \text{(unknown size)} \end{array}$$

Once the size constraints have been solved, we can use the following function to compute the iteration size of each binding. In the definition, we use the syntax $\Gamma(xs)$ to find the $xs : k$ element in the environment Γ and return the associated size k. Similarly, we use the syntax $C(k)$ to find the corresponding $k = \tau$ constraint in C and return the associated size type τ.

$$\begin{array}{llll} iter_{\Gamma,C} & : & bind \to T & \\ iter_{\Gamma,C} & \mid & (z\ = \texttt{fold}\ f\ xs) & = C(\Gamma(xs)) \\ & \mid & (ys = \texttt{map}_n\ f\ \overline{xs}) & = C(\Gamma(ys)) \\ & \mid & (ys = \texttt{filter}\ f\ xs) & = C(\Gamma(xs)) \\ & \mid & (ys = \texttt{generate}\ s\ f) & = C(\Gamma(ys)) \\ & \mid & (ys = \texttt{gather}\ is\ xs) & = C(\Gamma(is)) \\ & \mid & (ys = \texttt{cross}\ as\ bs) & = C(\Gamma(as)) \times C(\Gamma(bs)) \\ & \mid & (ys = \texttt{external}\ \overline{xs}) & = \perp \end{array}$$

$$\boxed{\mathit{program}\ :_s\ \sigma}$$

$$\frac{\begin{array}{l}\{k_i,\ xs_i : k_i\}^{i\leftarrow 1..n}\ \vdash\ \texttt{let}\ bs\ \texttt{in}\ \{ys_j\}^{j\leftarrow 1..m}\ \leadsto\ \Gamma[ys_j : k'_j]^{j\leftarrow 1..m}\ \vdash\ C \\ (\Gamma',\ C') = \mathrm{SOLVE}(\Gamma,\ C) \quad \{k_i = s_i\}^{i\leftarrow 1..n} \in C' \quad \{k'_j = t_j\}^{j\leftarrow 1..m} \in C' \\ \overline{k_a} = \{k \mid k \in \Gamma'\} \cap (\bigcup_{i\leftarrow 1..n} \mathrm{fv}(s_i)) \quad \overline{k_e} = \{k \mid \exists k \in \Gamma'\} \cap (\bigcup_{j\leftarrow 1..m} \mathrm{fv}(t_j)) \quad \{\exists k \notin \Gamma \mid \bigcup_{i\leftarrow 1..n} \mathrm{fv}(s_i)\}\end{array}}{f\ \{xs\}^{i\leftarrow 1..n} = \texttt{let}\ bs\ \texttt{in}\ \{ys\}^{j\leftarrow 1..m}\ :_s \forall \overline{k_a}.\ \exists \overline{k_e}.\ (\{xs_i : s_i\}^{i\leftarrow 1..n}) \to (\{ys_j : t_j\}^{j\leftarrow 1..m})} \quad \text{(SProgram)}$$

$$\boxed{\Gamma\ \vdash\ \mathit{lets}\ \leadsto\ \Gamma\ \vdash\ C}$$

$$\Gamma\ \vdash\ \texttt{let}\ \cdot\ \texttt{in}\ \mathit{exp}\ \leadsto\ \Gamma\ \vdash\ \mathit{true} \quad \text{(SNil)} \qquad \frac{\Gamma_1 \mid zs\ \vdash\ b\ \leadsto\ \Gamma_2\ \vdash\ C_1 \quad \Gamma_2\ \vdash\ \texttt{let}\ bs\ \texttt{in}\ \mathit{exp}\ \leadsto\ \Gamma_3\ \vdash\ C_2}{\Gamma_1\ \vdash\ \texttt{let}\ zs\ = b\ ;\ bs\ \texttt{in}\ \mathit{exp}\ \leadsto\ \Gamma_3\ \vdash\ C_1 \wedge C_2} \quad \text{(SCons)}$$

$$\boxed{\Gamma \mid z\ \vdash\ \mathit{bind}\ \leadsto\ \Gamma\ \vdash\ C}$$

$$\begin{array}{llll}
\Gamma[xs_i : k_i]^{i\leftarrow 1..n} & \mid zs\ \vdash\ \texttt{map}_n\ f\ \{xs_i\}^{i\leftarrow 1..n} & \leadsto\ \Gamma,\ zs : k_{zs},\ k' & \vdash\ \bigwedge_{i\leftarrow 1..n}\{k_i = k'\}\ \wedge\ k_{zs} = k' \\
\Gamma & \mid zs\ \vdash\ \texttt{filter}\ f\ xs & \leadsto\ \Gamma,\ zs : k_{zs},\ \exists k' & \vdash\ k_{zs} = k' \\
\Gamma & \mid x\ \ \vdash\ \texttt{fold}\ f\ xs & \leadsto\ \Gamma & \vdash\ \mathit{true} \\
\Gamma & \mid zs\ \vdash\ \texttt{generate}\ s\ f & \leadsto\ \Gamma,\ zs : k_{zs},\ \exists k' & \vdash\ k_{zs} = k' \\
\Gamma[is : k_{is}] & \mid zs\ \vdash\ \texttt{gather}\ xs\ is & \leadsto\ \Gamma,\ zs : k_{zs},\ k' & \vdash\ k_{zs} = k',\ k_{is} = k' \\
\Gamma[xs : k_{xs},\ ys : k_{ys}] & \mid zs\ \vdash\ \texttt{cross}\ xs\ ys & \leadsto\ \Gamma,\ zs : k_{zs},\ k',\ k'' & \vdash\ k_{zs} = k' \times k''\ \wedge\ k_{xs} = k'\ \wedge\ k_{ys} = k'' \\
\Gamma & \mid zs\ \vdash\ \texttt{external}\ \{xs\}^{i\leftarrow 1..n} & \leadsto\ \Gamma,\ zs : k_{zs},\ \exists k' & \vdash\ k_{zs} = k'
\end{array}$$

Figure 4. Constraint Generation

3.6 Transducers

We define the concept of *transducers* as combinators having a different output size to their iteration size. As with any other combinator, a transducer may fuse with other nodes of the same iteration size, but transducers may also fuse with nodes having iteration size the same as the transducer's output size. For our set of combinators, the only transducer is `filter`.

Looking back at the `normalize2` example, the iteration sizes of the combinators of `gts` and `sum1` are both k_{xs}. The iteration size of `sum2` is k_{gts}, and the filter combinator which produces `gts` is a transducer from k_{xs} to k_{gts}. Even though k_{gts} is not equal to k_{xs}, the three nodes `gts`, `sum1` and `sum2` can all be fused together.

We now define a function *trans*, to find the parent transducer of a combinator application. Since each name is bound to at most one combinator, we abuse terminology here slightly and write *combinator n* when refering to the combinator occuring in the binding of the name n. The parent transducer $\mathit{trans}(bs, n)$ of a combinator n has the same output size as n's iteration size, but the two have different iteration sizes.

$$\begin{array}{lllll}
\mathit{trans} & : & \mathit{binds} \to \mathit{name} \to \{\mathit{name}\} & & \\
\mathit{trans}(bs, o) & & & & \\
 & \mid & o = \texttt{filter}\ f\ n & \in bs & = \mathit{trans}'(bs, n) \\
 & \mid & \text{otherwise} & & = \mathit{trans}'(bs, o)
\end{array}$$

$$\begin{array}{llll}
\mathit{trans}'(bs, o) & & & \\
\mid & o = \texttt{fold}\ f\ n & \in bs & = \emptyset \\
\mid & o = \texttt{map}_n\ f\ ns & \in bs & = \bigcup_{x \in ns} \mathit{trans}(bs, x) \\
\mid & o = \texttt{filter}\ f\ n & \in bs & = \{o\} \\
\mid & o = \texttt{generate}\ s\ f & \in bs & = \emptyset \\
\mid & o = \texttt{gather}\ i\ d & \in bs & = \mathit{trans}(bs, i) \\
\mid & o = \texttt{cross}\ a\ b & \in bs & = \emptyset \\
\mid & o = \texttt{external}\ \mathit{ins} & \in bs & = \emptyset
\end{array}$$

To determine whether two combinators of different iteration sizes may be fused together, we first find parent or ancestor transducers of the same size, if they exist:

$$\begin{array}{lll}
\mathit{parents} & : & \mathit{binds} \to \mathit{name} \to \mathit{name} \to \{\mathit{name} \times \mathit{name}\} \\
\mathit{parents}(bs, a, b) & & \\
 & \mid & \mathit{iter}_{\Gamma,C}(bs(a)) == \mathit{iter}_{\Gamma,C}(bs(b)) \\
 & & \quad = \{(a, b)\} \\
 & \mid & \text{otherwise} \\
 & & \quad = \{\mathit{parents}(bs, a', b) \mid a' \in \mathit{trans}(bs, a)\} \\
 & & \quad \cup\ \{\mathit{parents}(bs, a, b') \mid b' \in \mathit{trans}(bs, b)\}
\end{array}$$

Two combinators a and b of different size may be fused together only if they have parents $(c, d) \in \mathit{parents}(a, b)$, and the combinators and their parents are also fused together. That is, in order for a and b to be fused together, c and d must be fused, a and c must be fused, and d and b must be fused. In the previous example, `sum1` and `sum2` have different iteration size, and their parents are $\mathit{parents}(\texttt{sum1}, \texttt{sum2}) = \{(\texttt{sum1}, \texttt{gts})\}$. In order for `sum1` and `sum2` to be fused together, `sum1` and `gts` must be fused, `sum1` and `sum1` must be fused, and `gts` and `sum2` must be fused. Now we can express the restriction on programs we view as valid for our transformation more formally:

Lemma: sole transducers. If a program p is *valid*, then its bindings will have at most one transducer:

$$\forall p, \sigma, n.\ p :_s \sigma \implies |\mathit{trans}(\mathit{binds}(p), n)| \leq 1$$

Lemma: sole parents. For some program p with valid constraints, each pair of names a and b will have at most one pair of parents $\mathit{parents}(a, b)$.

$$\forall p, \sigma, a, b.\ p :_s \sigma \implies |\mathit{parents}(\mathit{binds}(p), a, b)| \leq 1$$

These two lemmas are used in the integer linear programming formulation, when generating the constraints. When fusing two nodes of different iteration size, at most one pair of parents will need to be checked.

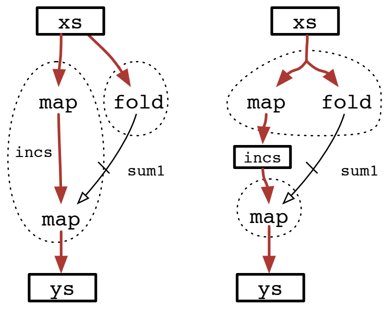

Figure 5. Possible clusterings for `normalizeInc`

4. Integer Linear Programming

It is usually possible to cluster a program graph in multiple ways. For example, consider the following simple function:

```
normalizeInc :: Array Int -> Array Int
normalizeInc xs
 = let incs = map  (+1)      us
       sum1 = fold (+) 0     us
       ys   = map  (/ sum1) incs
   in  ys
```

Two possible clusterings are shown in Figure 5. One option is to compute `sum1` first and fuse the computation of `incs` and `ys`. Another option is to fuse the computation of `incs` and `sum1` into a single loop, then compute `ys` separately. A third option (not shown) is to compute all results separately, and not perform any fusion.

Which option is better? On current hardware we generally expect the cost of memory access to dominate runtime. The first clustering in Figure 5 requires two reads from array `xs` and one write to array `ys`. The second requires a single fused read from `xs`, one write to `incs`, a read back from `incs` and a final write to `ys`. From the size constraints of the program we know that all intermediate arrays have the same size, so we expect the first clustering will peform better as it only needs three array accesses instead of four.

For small programs such as `normalizeInc` it is possible to naively enumerate all possible clusterings, select just those that are *valid* with respect to fusion preventing edges, and choose the one that maximises a cost metric such as the number of array accesses needed. However, as the program size increases the number of possible clusterings becomes too large to naively enumerate. For example, Pouchet et al [15] present a fusion system using the polyhedral model [16] and report that some simple numeric programs have over 40,000 possible clusterings, with one particular example having 10^{12}.

To deal with the combinatorial explosion in the number of potential clusterings, we instead use an Integer Linear Programming (ILP) formulation. ILP problems are defined as a set of variables, an objective linear function and a set of linear constraints. The integer linear solver finds an assignment to the variables that minimises the objective function, while satisfying all constraints. For the clustering problem we express our constraints regarding fusion preventing edges as linear constraints on the ILP variables, then use the objective function to encode our cost metric. This general approach was first fully described by Megiddo and Sarkar [13], and our main contribution is to extend it to work with size changing operators such as `filter`.

$$\begin{array}{lll} nodes & : & program \to V \\ edges & : & program \to E \\ edge & : & \{bind\} \times bind \to E \\ inedge & : & \{bind\} \times name \times name \to E \end{array}$$

$$nodes(bs) \ = \{(name(b), iter_{\Gamma,C}(b)) | b \in bs\}$$

$$edges(bs) \ = \textstyle\bigcup_{b \in bs} edge(bs, b)$$

$$\begin{array}{l}
edge(bs, out = \texttt{fold}\ f\ in) \\
\quad = \{inedge(bs, out, s) | s \in fv(f)\} \cup \{inedge(bs, out, in)\} \\
edge(bs, out = \texttt{map}\ f\ in) \\
\quad = \{inedge(bs, out, s) | s \in fv(f)\} \cup \{inedge(bs, out, in)\} \\
edge(bs, out = \texttt{filter}\ f\ in) \\
\quad = \{inedge(bs, out, s) | s \in fv(f)\} \cup \{inedge(bs, out, in)\} \\
edge(bs, out = \texttt{gather}\ data\ indices) \\
\quad = \{(out, data, \text{fusion-preventing})\} \cup \{inedge(bs, out, indices)\} \\
edge(bs, out = \texttt{cross}\ a\ b) \\
\quad = \{inedge(bs, out, a)\} \cup \{(out, b, \text{fusion-preventing})\} \\
edge(bs, outs = \texttt{external}\ ins) \\
\quad = \{(outs, i, \text{fusion-preventing}) | i \in ins\}
\end{array}$$

$$\begin{array}{l}
inedge(bs, to, from) \\
\quad | \ (from = \texttt{fold}\ f\ s) \in bs \\
\quad = (to, from, \text{fusion-preventing}) \\
\quad | \ (outs = \texttt{external} \ldots) \in bs \land from \in outs \\
\quad = (to, outs, \text{fusion-preventing}) \\
\quad | \ otherwise \\
\quad = (to, from, \text{fusible})
\end{array}$$

Figure 6. Dependency Graphs from Programs

4.1 Dependency Graphs

A dependency graph represents the data dependencies of the program to be fused, and we use it as an intermediate stage when producing linear constraints for the ILP problem. The dependency graph contains enough information to determine the possible clusterings of the input program, while abstracting away from the exact operators used to compute each intermediate array. The rules for producing dependency graphs are in Figure 6.

Each binding in the source program becomes a node in the dependency graph. For each intermediate variable, we add a directed edge from the binding that produces a value to all bindings that consume it. Each edge is also marked as either *fusible* or *fusion preventing*. Fusion preventing edges are used when the producer must finish its execution before the consumer node can start. For example, a `fold` operation must complete execution before it can produce the scalar value needed by its consumers. Conversely, the `map` operation produces an output value for each value it consumes, so is marked as fusible.

The `gather` operation is a hybrid: it takes an indices array and an elements array, and for each element in the indices array returns the corresponding data element. This means that gather can be fused with the operation that produces its indices, but not the operation that produces its elements — because those are accessed in a random-access manner.

4.2 ILP Variables

After generating the dependency graph, the next step is to produce a set of linear constraints from this graph. The variables involved in these constraints are split into three groups:

$$x \ : \ node \times node \ \to \ \mathbb{B}$$

For each pair of nodes with indices i and j we use a boolean variable $x_{i,j}$ which indicates whether those two nodes are fused. We use $x_{i,j} = 0$ when the nodes are fused and $x_{i,j} = 1$ when they are not. Using 0 for the fused case means that the objective function can be a weighted function of the $x_{i,j}$ variables, and minimizing it tends to increase the number of nodes that are fused. The values of these variables are used to construct the final clustering, such that $\forall i,j.\, x_{i,j} = 0 \iff cluster(i) = cluster(j)$.

$\pi \;:\quad node \quad \to \quad \mathbb{R}$

The second group of variables is used to ensure that the clustering is acyclic. This means that for each node in the graph, the dependencies of that node can be executed before the node itself. For each node i, we associate a real π_i such that every node j that depends on i we have $\pi_j > \pi_i$. Our linear constraints will ensure that if two nodes are fused into the same cluster then their π values will be identical — though nodes in different clusters can also have the same π value. Here is an example of a cyclic clustering:

```
cycle xs  = let ys  = map (+1) xs     (C1)
                sum = fold ys         (C2)
                zs  = map (+sum) ys   (C1)
            in  zs
```

There is no fusion-preventing edge directly between the `xs` and `zs` bindings, but there is a fusion-preventing edge between `sum` and `zs`. If the `xs` and `zs` bindings were in the same cluster `C1` and `sum` was in cluster `C2`, there would be a dependency cycle between `C1` and `C2`, and neither could be executed before the other.

$c \;:\quad node \quad \to \quad \mathbb{B}$

The final group of variables is used to help define the cost model encoded by the objective function. Each node is assigned a variable c_i that indicates whether the array the associated binding produces is *fully contracted*. When an array is fully contracted it means that all consumers of that array are fused into the same cluster, so we have $c_i = 0 \iff \forall (i', j) \in E.\, i = i' \implies x_{i,j} = 0$. In the final program, each successive element of a fully contracted array can be stored in a scalar register, rather than requiring an array register or memory storage.

4.3 Linear Constraints

The constraints we place on the ILP variables are split into four groups: constraints that ensure the clustering is acyclic; constraints that encode fusion preventing edges; constraints on nodes with different iteration sizes, and constraints involving array contraction.

Acyclic and precedence-preserving The first group of constraints ensures that the clustering is acyclic:

$$\begin{array}{rl} x_{i,j} \le \pi_j - \pi_i \le N \cdot x_{i,j} & \text{(with an edge from } i \text{ to } j) \\ -N \cdot x_{i,j} \le \pi_j - \pi_i \le N \cdot x_{i,j} & \text{(with no edge from } i \text{ to } j) \end{array}$$

As per Megiddo [13] the form of these constraints is determined by whether there is an dependency between nodes i and j. The N value is set to the total number of nodes in the graph.

If there is an edge from node i to j we use the first constraint form shown above. If the two nodes are fused into the same cluster then we have $x_{i,j} = 0$. In this case the constraint simplifies to $0 \le \pi_j - \pi_i \le 0$, which forces $\pi_i = \pi_j$. If the two nodes are in *different* clusters then the constraint instead simplifies to $1 \le \pi_j - \pi_i \le N$. This means that the difference between the two πs must be at least 1, and less than N. Since there are N nodes, the maximum difference between any two πs would be at most N, so the upper bound of N is large enough to be safely ignored. This means the constraint can roughly be translated to $\pi_i < \pi_j$, which enforces the acyclicity constraint.

If instead there is no edge from node i to j then we use the second constraint form above. As before, if the two nodes are fused into the same cluster then we have $x_{i,j} = 0$, which forces $\pi_i = \pi_j$. If the nodes are in different clusters then the constraint simplifies to $-N \le \pi_j - \pi_i \le N$, which effectively puts no constraint on the π values.

Fusion-preventing edges As per Megiddo [13], if there is a fusion preventing edge between two nodes we add a constraint to ensure the nodes will be placed in different clusters.

$$\begin{array}{l} x_{i,j} \quad = 1 \\ \text{(for fusion-preventing edges from } i \text{ to } j) \end{array}$$

When combined with the precedence-preserving constraints earlier, setting $x_{i,j} = 1$ also forces $\pi_i < \pi_j$.

Fusion between different iteration sizes This group of constraints restricts which nodes can be placed in the same cluster based on their iteration size. The group has three parts. Firstly, either of the two nodes connected by an edge have an unknown ($\perp$) iteration size then they cannot be fused and we set $x_{i,j} = 1$:

$$\begin{array}{l} x_{i,j} \quad = 1 \\ \text{(if } iter_{\Gamma,C}(i) = \perp \vee iter_{\Gamma,C}(j) = \perp) \end{array}$$

Secondly, if the two nodes have different iteration sizes and no common parent then they also cannot be fused and we set $x_{i,j} = 1$:

$$\begin{array}{l} x_{i,j} \quad = 1 \\ \text{(if } iter_{\Gamma,C}(i) \ne iter_{\Gamma,C}(j) \wedge parents(i,j) = \emptyset) \end{array}$$

Finally, if the two nodes had different iteration sizes but *do* have parent transducers of the same size, then the two nodes can be fused if they are fused with their respective parents, and the parents themselves are fused:

$$\begin{array}{l} x_{a,A} \quad \le x_{a,b} \\ x_{b,B} \quad \le x_{a,b} \\ x_{A,B} \quad \le x_{a,b} \\ \text{(if } iter_{\Gamma,C}(a) \ne iter_{\Gamma,C}(b) \wedge parents(a,b) = \{(A,B)\}) \end{array}$$

This last part is the main difference to existing ILP solutions: we allow nodes with different iteration sizes to be fused when their parent transducers are fused. The actual constraints encode a "no more fused than" relationship. For example $x_{a,A} \le x_{a,b}$ means that nodes a and b can be no more fused than nodes a and A.

As a simple example, consider fusing an operation on filtered data with its generating filter:

```
sum1 = fold (+) 0  xs
gts  = filter (>0) xs
sum2 = fold (+) 0  gts
```

Here $sum1$ and $sum2$ have different iteration sizes and we have that $parents(sum1, sum2) = \{(sum1, gts)\}$. This means that $sum1$ and $sum2$ may only be fused if $sum1$ is fused with $sum1$ (trivial), $sum2$ is fused with gts, and $sum1$ is fused with gts.

Array contraction The final group gives meaning to the c variables, which represent whether an array is fully contracted:

$$\begin{array}{l} x_{i,j} \quad \le c_i \\ \text{(for all edges from i)} \end{array}$$

Recall that an array is fully contracted when all of the consumers are fused with the nodes that produces it, which means that the array does not need to be fully materialized in memory. As per Darte's work on array contraction [7], we define a variable c_i for each array, and the constraint above ensures that $c_i = 0$ only if $\forall (i', j) \in E.\, i = i' \implies x_{i,j} = 0$. By minimizing c_i in the objective function, we favor solutions that reduce the number of intermediate arrays.

4.4 Objective Function

The objective function defines the cost model of the program, and the ILP solver will find the clustering that minimizes this function while satisfying the constraints defined in the previous section. The cost model we use in this paper has three components:

- the number of array reads and writes — an abstraction of the amount of memory bandwidth needed by the program;
- the number of intermediate arrays — an abstraction of the amount of intermediate memory needed;
- the number of distinct clusters — an abstraction of the cost of loop management instructions, which maintain loop counters and the like.

The three components of the cost model are a heuristic abstraction of the true cost of executing the program on current hardware. They are ranked in order of importance — so we prefer to minimize the number of array reads and writes over the number of intermediate arrays, and to minimize the number of intermediate arrays over the number of clusters. However, minimizing one component does not necessarily minimize any other. For example, as the fused program executes multiple array operations at the same time, in some cases the clustering that requires the least number of array reads and writes uses more intermediate arrays than strictly necessary.

We encode the ordering of the components of the cost model as different weights in the objective function. First, note that if the program graph contains N combinators (nodes) then there are at most N opportunities for fusion. We then encode the relative cost of loop overhead as weight 1, the cost of an intermediate array as weight N, and the cost of an array read or write as weight N^2. This ensures that no amount of loop overhead reduction can outweigh the benefit of removing an intermediate array, and likewise no number of removed intermediate arrays can outweigh a reduction in the number of array reads or writes. The integer linear program including the objective function is as follows:

$$\begin{array}{lll}
\text{Minimise} & \Sigma_{(i,j)\in E} W_{i,j}\cdot x_{i,j} & \text{(memory traffic and loop overhead)}\\
\quad + & \Sigma_{i\in V} N\cdot c_i & \text{(removing intermediate arrays)}
\end{array}$$

$$\begin{array}{lll}
\text{Subject to} & \ldots \text{ constraints from §4.3} \ \ldots & \\
\text{Where} & W_{i,j} = N^2 \,| & (i,j)\in E \\
 & & \text{(fusing } i \text{ and } j \text{ will reduce memory traffic)}\\
 & W_{i,j} = N^2 \,| & \exists k.(k,i)\in E \wedge (k,j)\in E \\
 & & (i \text{ and } j \text{ share an input array})\\
 & W_{i,j} = 1 \,\,\,| & \texttt{otherwise}\\
 & & \text{(the only benefit is loop overhead)}\\
 & N = |V| &
\end{array}$$

4.5 Fusion-preventing Path Optimisation

The integer linear program defined in the previous section includes more constraints than strictly necessary to define the valid clusterings. If two nodes have a path between them which includes a fusion preventing edge, then we know up front that they must be placed in different clusters. The following function $possible(a,b)$ determines whether there is any possibility that the two nodes a and b can be fused. Similarly the function $possible'(a,b)$ checks whether there is any possibility that the parents of a and b may be fused.

$$\begin{array}{l}
possible\ :\ \ name\times name \to \mathbb{B}\\
possible(a,b)\ =\ \forall p\in path(a,b)\cup path(b,a).\ \text{fusion-preventing} \notin p
\end{array}$$

$$\begin{array}{l}
possible'\!:\ \ name\times name \to \mathbb{B}\\
possible'(a,b)\ =\ \exists A,B.\ parents(a,b)=\{A,B\}\wedge possible(a,b)\\
\qquad\qquad\quad \wedge\ possible(A,a)\wedge possible(B,b)\wedge possible(A,B)
\end{array}$$

With *possible* and *possible'* defined, we refine our formulation to only generate constraints between two nodes if there is a chance they may be fused together. Doing this reduces the total number of constraints, and makes the job of the ILP solver easier. The final formulation of the integer linear program follows.

$$\begin{array}{ll}
\text{Minimise} & \Sigma_{(i,j)\in E} W_{i,j}\cdot x_{i,j} + \Sigma_{i\in V} N\cdot c_i\\
 & (\text{if } possible(i,j))\\
\text{Subject to} & -N\cdot x_{i,j} \ \le\ \pi_j - \pi_i \ \le\ N\cdot x_{i,j}\\
 & (\text{if } possible(i,j)\wedge (i,j)\notin E \wedge (j,i)\notin E)\\
 & x_{i,j} \qquad\ \le\ \pi_j - \pi_i \ \le\ N\cdot x_{i,j}\\
 & (\text{if } possible(i,j)\wedge (i,j,\text{fusible})\in E)\\
 & \qquad\qquad\quad \pi_i < \pi_j\\
 & (\text{if } (i,j,\text{fusion-preventing})\in E)\\
 & x_{i,j} \qquad\ \le\ c_i\\
 & (\text{if } (i,j,\text{fusible})\in E)\\
 & c_i \qquad\ \ =\ 1\\
 & (\text{if } (i,j,\text{fusion-preventing})\in E)\\
 & x_{i,j} \qquad\ =\ 1\\
 & (\text{if } \bot\in\{iter_{\Gamma,C}(i), iter_{\Gamma,C}(j)\})\\
 & x_{i',i} \qquad\ \le\ x_{i,j}\\
 & x_{j',j} \qquad\ \le\ x_{i,j}\\
 & x_{i',j'} \qquad \le\ x_{i,j}\\
 & (\text{if } iter_{\Gamma,C}(i)\ne iter_{\Gamma,C}(j)\wedge possible'(i,j)\\
 & \qquad\qquad \wedge\ parents(i,j)=\{(i',j')\})\\
 & x_{i,j} \qquad\ =\ 1\\
 & (\text{if } iter_{\Gamma,C}(i)\ne iter_{\Gamma,C}(j)\wedge\neg possible'(i,j))\\
\text{Where} & W_{ij}=N^2 \ |\ \ (i,j)\in E\\
 & \qquad\qquad\quad\ \ \text{(fusing } i \text{ and } j \text{ will reduce memory traffic)}\\
 & W_{ij}=N^2 \ |\ \ \exists k.(k,i)\in E\wedge (k,j)\in E\\
 & \qquad\qquad\quad\ \ (i \text{ and } j \text{ share an input array})\\
 & W_{ij}=1 \quad\ |\ \ \texttt{otherwise}\\
 & \qquad\qquad\quad\ \ \text{(the only benefit is loop overhead)}\\
 & N=|V|
\end{array}$$

5. Benchmarks

This section discusses three representative benchmarks, and gives the full ILP program of the first. These benchmarks highlight the main differences between our fusion mechanism and related work. The runtimes of each benchmark are summarized in Figure 7. We report times for: the unfused case where each operator is assigned to its own cluster; the clustering implied by stream fusion [5]; the clustering chosen by Megiddo [13], and the clustering chosen by our system.

For each benchmark we report the runtimes of hand-fused C code based on the clustering determined by each algorithm. Although we also have an implementation of our Data Flow Fusion system in terms of a GHC plugin [12], we report on hand-fused C code to provide a fair comparison to related work. As mentioned in [12], the current Haskell stream fusion mechanism introduces overhead in terms of a large number of duplicate loop counters, which increases register pressure unnecessarily. Hand fusing all code and compiling it with the same compiler (GCC) isolates the true cost of the various clusterings from low level differences in code generation.

We have used both GLPK and CPLEX as external ILP solvers. For small programs such as `normalizeInc`, both solvers produce solutions in under 100ms. For a larger randomly generated example with twenty-five combinators, GLPK took over twenty minutes to produce a solution while the commercial CPLEX solver was able to produce a solution in under one second — which is still quite usable. We will investigate the reason for this wide range in performance in future work.

The benchmark programs are at `https://github.com/amosr/papers/tree/master/2014betterfusionforfilters/benches`.

	Unfused		Stream		Megiddo		**Ours**	
	Time	Loops	Time	Loops	Time	Loops	Time	Loops
Normalize2	1.88s	5	1.64s	4	1.82s	3	**1.59s**	**2**
Closest points	3.83s	6	3.33s	5	2.92s	3	**2.92s**	**3**
QuadTree	5.22s	8	5.22s	8	4.72s	2	**4.72s**	**2**

Figure 7. Benchmark results

5.1 Normalize2

To demonstrate the ILP formulation we will use the `normalize2` example from §1, repeated here:

```
normalize2 :: Array Int -> Array Int
normalize2 xs
 = let sum1 = fold   (+)  0   xs
       gts  = filter (>   0)  xs
       sum2 = fold   (+)  0   gts
       ys1  = map    (/ sum1) xs
       ys2  = map    (/ sum2) xs
   in (ys1, ys2)
```

We use the final ILP formulation from §4.5. First, we calculate *possible* – that is, the nodes which have no fusion-preventing path between them.

$$\{\{sum1, gts, sum2\}, \{sum1, ys2\}, \{gts, sum2, ys1\}, \{ys1, ys2\}\}$$

The complete ILP program is shown below. Note that in the objective function the weights for $x_{sum1,sum2}$ and $x_{sum2,ys1}$ are both only 1, because they do not share any input arrays.

$$\begin{array}{llll}
\text{Minimise} & \multicolumn{3}{l}{25 \cdot x_{sum1,gts} + 1 \cdot x_{sum1,sum2} + 25 \cdot x_{sum1,ys2} +} \\
& \multicolumn{3}{l}{25 \cdot x_{gts,sum2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} +} \\
& \multicolumn{3}{l}{25 \cdot x_{ys1,ys2} + 5 \cdot c_{gts} + 5 \cdot c_{ys1} + 5 \cdot c_{ys2}} \\
\text{Subject to} & -5 \cdot x_{sum1,gts} & \leq \pi_{gts} - \pi_{sum1} & \leq 5 \cdot x_{sum1,gts} \\
& -5 \cdot x_{sum1,sum2} & \leq \pi_{sum2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,sum2} \\
& -5 \cdot x_{sum1,ys2} & \leq \pi_{ys2} - \pi_{sum1} & \leq 5 \cdot x_{sum1,ys2} \\
& -5 \cdot x_{gts,ys1} & \leq \pi_{ys1} - \pi_{gts} & \leq 5 \cdot x_{gts,ys1} \\
& -5 \cdot x_{sum2,ys1} & \leq \pi_{ys1} - \pi_{sum2} & \leq 5 \cdot x_{sum2,ys1} \\
& -5 \cdot x_{ys1,ys2} & \leq \pi_{ys2} - \pi_{ys1} & \leq 5 \cdot x_{ys1,ys2} \\
& x_{gts,sum2} & \leq \pi_{sum2} - \pi_{gts} & \leq 5 \cdot x_{gts,sum2} \\
& & \pi_{sum1} < \pi_{ys1} & \\
& & \pi_{sum2} < \pi_{ys2} & \\
& x_{gts,sum2} & \leq c_{gts} & \\
& x_{gts,sum2} & \leq x_{sum1,sum2} & \\
& x_{sum1,sum1} & \leq x_{sum1,sum2} & \\
& x_{sum1,gts} & \leq x_{sum1,sum2} &
\end{array}$$

One minimal solution to this is:

$$\begin{array}{ll}
x_{sum1,gts},\ x_{sum1,sum1},\ x_{sum1,sum2},\ x_{gts,sum2},\ x_{ys1,ys2} & = 0 \\
x_{sum1,ys2},\ x_{gts,ys1},\ x_{sum2,ys1} & = 1 \\
\pi_{sum1},\ \pi_{gts},\ \pi_{sum2} & = 0 \\
\pi_{ys1},\ \pi_{ys2} & = 1 \\
c_{gts},\ c_{ys1},\ c_{ys2} & = 0
\end{array}$$

This minimal solution is not unique, though in this case the only other minimal solutions use different π values, and denote the same clustering. Looking at just the non-zero variables in the objective function, the value is $25 \cdot x_{sum1,ys2} + 25 \cdot x_{gts,ys1} + 1 \cdot x_{sum2,ys1} = 51$. For illustrative purposes, note that objective function could be reduced by setting $x_{sum1,ys2} = 0$ (fusing *sum*1 and *ys*1), but this conflicts with other constraints. Since $x_{sum1,sum2} = 0$, we require $\pi_{sum1} = \pi_{sum2}$, but also $\pi_{sum2} < \pi_{ys2}$. These constraints cannot be satisfied, so a clustering that fused *sum*1 and *ys*2 would not also permit *sum*1 and *sum*2 to be fused.

We will now compare the clustering produced by our system, with the one implied by stream fusion. As described in [12], stream fusion cannot fuse a produced array into multiple consumers, or fuse operators that are not in a producer-consumer relationship. The corresponding values of the x_{ij} variables are:

$$\begin{array}{ll}
x_{gts,sum2} & = 0 \\
x_{sum1,gts}, x_{sum1,sum2}, x_{ys1,ys2}, x_{sum1,ys2}, x_{gts,ys1}, x_{sum2,ys1} & = 1
\end{array}$$

We can force this clustering to be applied in our integer linear program by adding the above equations as new constraints. Solving the resulting program then yields:

$$\begin{array}{ll}
\pi_{sum1}, \pi_{gts}, \pi_{sum2} & = 0 \\
\pi_{ys1}, \pi_{ys2} & = 1 \\
c_{gts}, c_{ys1}, c_{ys2} & = 0
\end{array}$$

Note that although nodes *sum*1 and *sum*2 have equal π values, they are not fused. Conversely, if two nodes have different π values then they are never fused.

5.2 Closest Points

The closest points benchmark is a divide-and-conquer algorithm that finds the closest pair of 2-dimensional points in an array. We first find the midpoint along the Y-axis, and filter the remaining points to those above and below the midpoint. We then recursively find the closest pair of points in the two halves, and merge the results. As the filtered points are passed directly to the recursive call, there is no further opportunity to fuse them, and our clustering is the same as returned by Megiddo's algorithm. However, our clustering generates both filtered arrays in a single loop, unlike stream fusion that requires a separate loop for each.

5.3 QuadTree

The QuadTree benchmark recursively builds a 2-dimensional space partitioning tree from an array of points. At each step the array of points is filtered into four 2-dimensional boxes. As with the closest points algorith, there are no further opportunities for fusing the filtered results, and our clustering is the same as Megiddo's. However, our clustering produces all four filtered results in a single loop, whereas stream fusion requires four loops.

5.4 QuickHull

The core of the QuickHull algorithm is shown below: given a line and an array of points, we filter the points to those above the line, and also find the point farthest from that line.

```
hull :: (Point,Point) -> Array Point -> Array Point
hull line@(l,r) pts
 = let pts' = filter (above   line) pts
       ma   = fold   (maxFrom line) pts'
   in (hull (l, ma) pts') ++ (hull (ma, r) pts')
```

Stream fusion cannot fuse the `pts'` and `ma` bindings because `pts'` is referred to multiple times and thus cannot be inlined. Megiddo's algorithm also cannot fuse the two bindings because their iteration sizes are different. If the `ma` binding was rewritten to operate over the `pts` array instead of `pts'`, Megiddo's formulation would be able to fuse the two, and the overall program would give the same result. However, this performance behavior is counter intuitive because `pts'` is likely to be smaller than `pts`, so in an unfused program the original version would be faster. Our system fuses both versions.

6. Related Work

The idea of using integer linear programming to cluster an operator graph for array fusion was first fully described by Megiddo and Sarkar [13] (1999). A simpler formulation, supporting only loops of the same iteration size, but optimizing for array contraction, was then described by Darte and Huard [7] (2002). Both algorithms were developed in the context of imperative languages (Fortran) and are based around a Loop Dependence Graph (LDG). In a LDG the nodes represent imperative loops, and the edges indicate which loops may or may not be fused. Although this work was developed in a context of imperative programming, the conceptual framework and algorithms are language agnostic. In earlier work, Chatterjee [3] (1991) mentioned that ILP can be used to schedule a data flow graph, though did not give a complete formulation. Our system extends the prior ILP approaches with support for size changing operators such as `filter`.

In the loop fusion literature, the ILP approach is considered "optimal" because it can find the clustering that minimizes a global cost metric. In our case the metric is defined by the objective function of §4.4. Besides optimal algorithms, there are also heuristic approaches. For example, Gao, Olsen and Sarkar [8] use the maxflow-mincut algorithm to try to maximize the number of fused edges in the LDG. Kennedy [11] describes another greedy approach which tries to maximize the reuse of intermediate arrays, and Song [18] tries to reduce memory references.

Greedy and heuristic approaches that operate on lists of bindings rather than the graph, such as Rompf [17], can find optimal clusterings in some cases, but are subject to changes in the order of bindings. In these cases, reordering bindings can produce a different clustering, leading to unpredictable runtime performance.

Darte [6] formalizes the algorithmic complexity of various loop fusion problems and shows that globally minimizing most useful cost metrics is NP-complete. Our ILP formulation itself is NP-hard, though in practice we have not yet found this to be a problem.

Recent literature on array fusion for imperative languages largely focuses on the polyhedral model. This is an algebraic representation imperative loop nests and transformations on them, including fusion transformations. Polyhedral systems [16] are able to express *all possible* distinct loop transformations where the array indices, conditionals and loop bounds are affine functions of the surrounding loop indices. However, the polyhedral model is not applicable to (or intended for) one dimensional filter-like operations where the size of the result array depends on the source data. Recent work extends the polyhedral model to support arbitrary indexing [19], as well as conditional control flow that is predicated on arbitrary (ie, non-affine) functions of the loop indices [2]. However, the indices used to write into the destination array must still be computed with affine functions.

Ultimately, the job of an array fusion system is to make the program go as fast as possible on the available hardware. Although the cost metrics of "optimal" fusion systems try to model the performance behavior of this hardware, it is not practical to encode the intricacies of all available hardware in a single compiler implementation. Iterative compilation approaches such as [1] instead enumerate many possible clusterings, use a cost metric to rank them, and perform benchmark runs to identify which clustering actually performs the best. An ILP formulation like ours naturally supports this model, as the integer constraints define the available clusterings, and the objective function can be used to rank them.

Acknowledgements

Many thanks are due to Manuel Chakravarty, Robert Clifton-Everest, Erik de Castro Lopo, Kai Engelhardt, Bill Kroon, Frederik M. Madsen, Abdallah Saffidine, Carter Schonwald, and Jingling Xue for enlightening discussions relating to this work.

References

[1] Thomas J. Ashby and Michael F. P. O'Boyle. Iterative collective loop fusion. In *CC: Compiler Construction*, 2006.

[2] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC: Compiler Construction*, 2010.

[3] Siddhartha Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *TOPLAS: Transactions on Programming Languages and Systems*, 15(3), 1993.

[4] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *PLDI: Programming Language Design and Implementation*, 1991.

[5] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*, 2007.

[6] Alain Darte. On the complexity of loop fusion. In *PACT: Parallel Architectures and Compilation Techniques*, 1999.

[7] Alain Darte and Guillaume Huard. New results on array contraction. In *ASAP*, 2002.

[8] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *LCPC: Languages and Compilers for Parallel Computing*, 1992.

[9] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*. ACM, 1993.

[10] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *ICFP: International Conference on Functional Programming*, 2010.

[11] Ken Kennedy. Fast greedy weighted fusion. *International Journal of Parallel Programming*, 29(5), 2001.

[12] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Amos Robinson. Data flow fusion with series expressions in Haskell. In *Haskell Symposium*, 2013.

[13] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *SPAA: Symposium on Parallel Algorithms and Architectures*, 1997.

[14] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[15] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC: High Performance Computing, Networking, Storage and Analysis*, 2010.

[16] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL: Principles of Programming Languages*, 2011.

[17] Tiark Rompf, Arvind K Sujeeth, Nada Amin, Kevin J Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *ACM SIGPLAN Notices*, volume 48, pages 497–510. ACM, 2013.

[18] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Improving data locality by array contraction. *IEEE Transactions on Computers*, 53(9), 2004.

[19] Anand Venkat, Manu Shantharam, Mary W. Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *CGO: Code Generation and Optimization*, 2014.