Polyhedral Expression Propagation

Johannes Doerfert
Saarland Informatics Campus
Saarland University, Germany
doerfert@cs.uni-saarland.de

Shrey Sharma
Saarland Informatics Campus
Saarland University, Germany
sharma@cs.uni-saarland.de

Sebastian Hack
Saarland Informatics Campus
Saarland University, Germany
hack@cs.uni-saarland.de

Abstract

Polyhedral techniques have proven to be powerful for various optimizations, from automatic parallelization to accelerator programming. At their core, these techniques compute accurate dependences among statement instances in order to apply complex program transformations. Such transformations comprise memory layout or program order modifications by optimizing memory access functions or scheduling functions. However, these approaches treat statements as opaque entities and do not consider changing the structure of the contained expressions or the memory accesses involved.

In this paper we present a technique that statically propagates expressions in order to avoid communicating their result via memory. While orthogonal to other polyhedral optimizations, this transformation can be used to enable them. Applied separately, expression propagation can increase parallelism, eliminate temporary arrays, create independent computations and improve cache utilization. It is especially useful for streaming codes that involve temporary arrays and scalar variables.

For multiple image processing pipelines we achieve portable speedups of up to 21.3× as well as a significant memory reduction compared to a naive parallel implementation. In 6 out of 7 cases, expression propagation outperforms a state-of-the-art polyhedral optimization especially designed for this kind of programs by a factor of up to 2.03×.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords polyhedral model, scalar removal, dependence removal, temporary memory elimination, recurrences

ACM Reference Format:
Johannes Doerfert, Shrey Sharma, and Sebastian Hack. 2018. Polyhedral Expression Propagation. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3178372.3179529

1 Introduction

Polyhedral-model-based analysis [11, 18, 58] and optimization techniques [4, 19] have been used for program optimization for decades. Their goal is to find a schedule of the statements of a loop that exhibits better locality [9, 20] and that allows to distribute the execution across different units such as vector lanes [33, 50], CPU cores [8, 21] or accelerators [2, 24, 41, 49, 54]. In addition to such schedule optimizations, polyhedral techniques have been successfully used to transform the memory layout either to eliminate false dependences [1, 17, 37] or to minimize the memory usage [6, 13, 35, 36, 55].

While all these approaches perform sophisticated changes to the execution order and the data layout, the structure of the program's expressions is generally not altered. Consequently, intermediate computations that are communicated via scalar variables or array cells to their users will be computed and communicated in the same way in the optimized program. However, distributing larger computations into various intermediate steps can limit the utilization of modern hardware features such as prefetchers and out-of-order execution. Additionally, the communication necessary for intermediate results can decrease the cache utilization and impose additional dependences that constrain scheduling.

```
for i = 0 to N do
 s = in[i] / 1.5;                              // S0
 if (i == 0)
   tmp[i] = (2 * in[i] + in[i+1]) / s;         // S1
 else if (i == N)
   tmp[i] = (in[i-1] + 2 * in[i]) / s;         // S2
 else
   tmp[i] = (in[i-1] + in[i] + in[i+1]) / s; // S3
 fi fi
done
for i = 1 to N-1 do
 out[i] = (tmp[i-1] + tmp[i] + tmp[i+1]);    // S4
done
```

Figure 1. Motivating example featuring a non-perfect loop nest, an input array `in`, an output array `out`, a temporary (non live-out) array `tmp` and a temporary scalar variable `s`.

The motivating example in Figure 1 features four statements (S0 to S3) that perform very simple computations to generate intermediate results before the final output is computed in statement S4. The overhead of a temporary array, the optimization loss due to the loop-carried write-after-write dependences caused by the scalar variable s and the nominal computation per statement can all cause severe performance degradation on a modern processing unit. While compiler optimizations [40, 51] and hardware features [48] can remedy some of the problems for certain programs, they will generally fail in the presence of multi-dimensional arrays [25] accessed in a sequence of non-perfect loop nests.

In this work we propose a polyhedral-model-based technique that statically propagates (intermediate) expressions to their users as an alternative to the communication of their result. If the communication was performed via non live-out memory, expression propagation can eliminate the need for the original computation and consequently the temporary memory used. Applied to the example in Figure 1 our technique will produce the code shown in Figure 2. As long as the resulting computation, here the one remaining loop body, stays within hardware specific bounds, e.g., all accesses can be prefetched, expression propagation can significantly improve performance and simultaneously decrease the memory requirement of an application.

```
out[1] = (     2 * in[0] + in[1]) / (in[0]/1.5)
       + (in[0] + in[1] + in[2]) / (in[1]/1.5)
       + (in[1] + in[2] + in[3]) / (in[2]/1.5);
for i = 2 to N-2 do
 out[i] = (in[i-2]+in[i-1]+in[i])/(in[i-1]/1.5)
        + (in[i-1]+in[i]+in[i+1])/(in[i  ]/1.5)
        + (in[i]+in[i+1]+in[i+2])/(in[i+1]/1.5);
done
out[N-1] = (in[N-3]+in[N-2]+in[N-1])/(in[N-2]/1.5)
         + (in[N-2]+in[N-1]+in[N  ])/(in[N-1]/1.5)
         + (        in[N-1]+2*in[N])/(in[N]/1.5);
```

Figure 2. Optimized version of the code in Figure 1. The scalar variable s and the temporary array tmp have been removed after the defining expressions have been propagated.

The remainder of this paper is organized as follows. In Section 2 we present necessary background information on all utilized techniques. Afterwards we present the four core contributions of this paper:

1. A formal criterion for sound expression propagation is given in Section 3 together with instructions on how to identify propagation opportunities as well as a complexity analysis for maximal propagation.
2. In Section 4 we discuss the practical application of expression propagation including different heuristics and a practical propagation algorithm.
3. Implementation details to maximize the performance gain of propagation are presented in Section 5.
4. An elaborate evaluation on seven different image processing pipelines is provided in Section 6. We show the portability of our approach using different architectures and measure the potential of our technique.

In Section 7 we will put our work into context and finish with a conclusion in Section 8.

2 Background

This section provides background on our program representation and the relevant aspects of the polyhedral model.

2.1 Input Language

We use a simple polyhedral input language as defined in Figure 3. It is similar to the ones used for other polyhedral approaches [14, 18] and consists of accesses, two kinds of expressions, and statements. Accesses $\langle acc\rangle$ can be *scalar* accesses or (multi-dimensional [25]) *array* accesses. Expressions are separated into affine expressions $\langle aexp\rangle$ and arbitrary expressions $\langle rexp\rangle$. Affine expressions are constant integers, parameters, loop induction variables $\langle iv\rangle$, additive and logical binary operations of affine expressions or a multiplicative binary operation where one operand is a constant. Regular expressions extend affine expressions with read accesses and arbitrary arithmetic or logical operations known from languages like C, except the assignment operator (=). Parameters are unknown but constant values that are defined outside the code region, e.g. function parameters. Statements $\langle stmt\rangle$ can be loops, conditionals or assignments. The body of the loop is executed until the induction variable, which is incremented by one, surpasses the affine upper bound. Thus both bounds are inclusive. Note that each assignment statement is identified by the single write access it contains.

$$
\begin{array}{lcl}
\langle acc\rangle & ::= & \langle ident\rangle\ (\textbf{[}\ \langle aexp\rangle\ \textbf{]})^{*} \\
\langle aexp\rangle & ::= & \langle cnst\rangle \mid \langle param\rangle \mid \textbf{(}\langle aexp\rangle\textbf{)} \mid \langle aexp\rangle\ (*|/)\ \langle cnst\rangle \\
 & \mid & \langle cnst\rangle * \langle aexp\rangle \mid \langle iv\rangle \mid \langle aexp\rangle\ (+|-|<=|==)\ \langle aexp\rangle \\
\langle rexp\rangle & ::= & \langle acc\rangle \mid \langle aexp\rangle \mid \textbf{(}\langle rexp\rangle\textbf{)} \mid \langle rexp\rangle \odot \langle rexp\rangle \\
\langle stmt\rangle & ::= & \textbf{for}\ \langle iv\rangle = \langle aexp\rangle\ \textbf{to}\ \langle aexp\rangle\ \textbf{do}\ \langle stmt\rangle\ \textbf{done} \\
 & \mid & \textbf{if}\ \langle aexp\rangle\ \textbf{then}\ \langle stmt\rangle\ [\textbf{else}\ \langle stmt\rangle]\ \textbf{fi} \\
 & \mid & \langle acc\rangle = \langle rexp\rangle\textbf{;} \mid \langle stmt\rangle\ \langle stmt\rangle \mid \epsilon
\end{array}
$$

Figure 3. Input language grammar consisting of accesses, affine expressions, arbitrary expressions and statements.

2.2 Polyhedral Model

The polyhedral model is a mathematical representation for static control parts (SCoPs) [18, 23] that is based on an implementation of Presburger arithmetic [52]. All programs derived from our input language form valid SCoPs, thus can be described and optimized in the model. The representation centers around the iteration instances of statements and the dependences that exist between them [18, 42]. All relevant parts of the polyhedral model are introduced in the following:

Iteration Domain The *iteration domain* $\mathcal{I}_S$ of a statement S describes all dynamically executed instances in terms of the values of the surrounding loop induction variables. Each element of $\mathcal{I}_S$ is an iteration vector $\mathbf{i}$ of the same dimensionality as the loop depth of S. The iteration domain of statement S4 in Figure 1 is for example: $\mathcal{I}_{S4} := \{(i_0) \mid 1 \leq i_0 \leq N-1\}$.

Schedule The *schedule* θ_S of a statement S maps all iteration instances of S to multidimensional timestamps which are executed in lexicographic order [19, 20]. This instance-wise lexicographic happens-before order is denoted as $\ll_{lex}$.

Instance-wise Dependences Dependence polyhedra [18, 29], or *instance-wise dependences*, relate dependent iteration instances of two statements if there exists a read-after-write *(RAW)*, write-after-read *(WAR)* or write-after-write *(WAW)* dependence between them. In contrast to other polyhedral approaches we require dependences to identify which read access of a statement is dependent on a write. Thus, dependences have access instance granularity not statement instance granularity [15]. We will denote dependences between a source access s and a target access t as $s\longrightarrow t$. Hence $w\longrightarrow r$ is a RAW dependence between a read r and a write w. Similarly, $r\longrightarrow *$ are all WAR dependences of a read r.

Notation To simplify the notation we omit statement names that can be inferred from the context. Instead, we use only the iterations vector, e.g., $\mathbf{i}$, $\mathbf{j}$ or $\mathbf{l}$, to identify statement iterations. We use r and w to refer to read and write accesses and e for source expressions. Finally, we denote the evaluation of an expression e in a specific iteration $\mathbf{i}$ as $[\![e]\!]_\mathbf{i}$.

3 Expression Propagation

The core of our approach is *expression propagation*, that is the evaluation of an expression at the program point where it is needed as an alternative to communicating the expression result through a scalar or memory location. To this end, we introduce the notion of *propagation dependences* that relates the original program point of the expression with one where it is both needed and "recomputable". Additionally, we introduce *propagation expressions* which are the actual expressions evaluated at the later program point instead of the original expression defined in the program.
An example for expression propagation is given in Figure 4. The propagation dependences shown in Figure 4c (left) are equal to the RAW dependences from statement S to T, though we omitted the iteration domain information. The propagation expressions (right) are derived from the original source expression `(i+1)*i` in statement S. They have been adjusted to compute the same value that would have been read in statement T otherwise. After propagation (ref. Figure 4b), the accesses in statement S might be obsolete if they are not otherwise read. At the same time the expression in statement T can be simplified to `4*j` by common scalar optimizations.

Definition 3.1 (Propagation Dependence & Expression)**.** A *propagation dependence* $w\rightharpoonup r$ is a subset of a *RAW* dependence $w\longrightarrow r$ for which a *propagation expression* $\overrightarrow{e_w}$ exists. The propagation expression has to evaluate at the target statement instance $\mathbf{j}$ to the same value that was written in the dependent source statement instance $\mathbf{i}$. Thus, if we denote the expression written by w as e_w:

$$w\rightharpoonup r \subseteq w\longrightarrow r \tag{1}$$

$$\forall(\mathbf{i},\mathbf{j}) \in w\rightharpoonup r: \ [\![e_w]\!]_\mathbf{i} = [\![\overrightarrow{e_w}]\!]_\mathbf{j} \tag{2}$$

```
for i = 1 to 2*N do
  A[i] = (i+1)*i; // S
done
for j = 1 to N do
  B[j] = A[2*j]    // T
       - A[2*j-1];
done
```

(a) Simple input program.

```
for i = 1 to 2*N do
  A[i] = (i+1)*i;    // S
done
for j = 1 to N do
  B[j] = (2*j+1)*2*j // T
       - 2*j*(2*j-1);
done
```

(b) Program after propagation.

$$w\rightharpoonup r_0 := \{(i,j) \mid i = 2*j\} \qquad \overrightarrow{e_w}_0 := \texttt{(2*j+1)*2*j}$$
$$w\rightharpoonup r_1 := \{(i,j) \mid i = 2*j-1\} \qquad \overrightarrow{e_w}_1 := \texttt{2*j*(2*j-1)}$$

(c) Propagation dependences $w\rightharpoonup r$ and expressions $\overrightarrow{e_w}$ for the two read accesses in Figure 4a. Note that we show propagation dependences without domain information to improve readability.

Figure 4. Propagation example for the simple input program in Figure 4a. The propagation dependences and expressions are shown in Figure 4c. The result is given in Figure 4b.

Propagation Expression Equivalence Expression propagation preserves the semantics if the evaluation of the propagation expression $\overrightarrow{e_w}$ is equal to the value loaded by r for all iterations in the range of the propagation dependence $w\rightharpoonup r$:

$$\forall(\mathbf{i},\mathbf{j}) \in w\rightharpoonup r: \ [\![\overrightarrow{e_w}]\!]_\mathbf{j} = [\![r]\!]_\mathbf{j}$$

To prove this property we apply the defining equation (2). Afterwards we need to show that the value read in the target statement instance is equal to the value written in the source statement instance for iterations related by $w\rightharpoonup r$:

$$\forall(\mathbf{i},\mathbf{j}) \in w\rightharpoonup r: \ [\![e_w]\!]_\mathbf{i} = [\![r]\!]_\mathbf{j}$$

Using the subset relation (1) and the definition of a RAW dependence [18] we know that the value of e_w in iteration $\mathbf{i}$, is in fact the value read by r in iteration $\mathbf{j}$ for $(\mathbf{i},\mathbf{j}) \in w\rightharpoonup r$.

Syntactic Read Replacement We now established that the read access of a propagation dependence can be replaced by the propagation expression for all iterations in the range of the propagation dependence. In order to allow syntactic replacement of the access we additionally require the propagation dependence to be surjective with regards to the iteration domain of the target statement, thus $range(w\rightharpoonup r) = \mathcal{I}_r$.

We always achieve surjective propagation dependences by splitting the target statement in two parts as illustrated in Figure 5. One part is completely reached by the dependence and one is not reached at all. Our approach will perform this splitting automatically, including the duplication of statements, accesses and dependences as well as the restriction of the iteration domains to their respective parts.

```
for i = 0 to N do
 if (i/2)*2 == i then
  A[i] = f(i);
 fi
done
for j = 0 to N do

  B[j] = A[j]; // T




done
```

(a) Propagation opportunity for even elements of A.

```
for i = 0 to N do
 if (i/2)*2 == i then
  A[i] = f(i);
 fi
done
for j = 0 to N do
 if (j/2)*2 == j then
  B[j] = f(j);  // T
 else
  B[j] = A[j];  // T'
 fi
done
```

(b) Propagation of A[i] to statement T after split into T and T'.

Figure 5. Statement splitting and syntactic read replacement for a non surjective propagation dependence.

3.1 Propagation Expressions

One task for expression propagation is to identify a valid propagation expression for a given (subset of a) RAW dependence. If a propagation expression was found the (part of the) RAW dependence becomes the propagation dependence and expression propagation can be applied. However, there are indefinitely many different syntactic expressions. In order to effectively identify propagation expressions we therefore limit ourselves to the ones that can be constructed from the original expression via induction variable adjustment. This is usually a necessary step as their scope is limited and their values change when the loop progresses.

Expression Rewriting To derive the propagation expressions we traverse e_w recursively and rewrite all induction variables $iv \in e_w$ according to relation $w{\longrightarrow}r$. The goal is to construct an expression $\overrightarrow{iv}$ that depend on the iteration vector of the target and that evaluates to the same value there as iv evaluated to in the source for dependent iterations.

Without loss of generality we assume iv is the induction variable of the k-th out of n loops surrounding the source statement. The affine function that provides the *value of iv* for an iteration $\mathbf{i}$ of the source statement is defined as

$$v(iv) := \big\{(\mathbf{i}, iv)\big\} = \big\{((i_1, \dots, i_n), iv)\big\} = \big\{((i_1, \dots, i_n), i_k)\big\}.$$

Since $w{\longrightarrow}r$ is an affine relation between instances of the statements surrounding w and r it can be written as

$$\big\{(\mathbf{i}, \mathbf{j}) \mid f\big((\mathbf{i}, \mathbf{j})\big)\big\} = \big\{((i_1, \dots, i_n), \mathbf{j}) \mid f\big((i_1, \dots, i_n), \mathbf{j}\big)\big\}$$

where f is a Presburger formula that defines the constrains under which the dependence exists. To obtain the function $v(\overrightarrow{iv})$ we apply the dependence $w{\longrightarrow}r$ to the domain of $v(iv)$:

$$v(\overrightarrow{iv}) = \big\{\big(w{\longrightarrow}r(\mathbf{i}), iv\big) \mid f(\mathbf{i}, \mathbf{j})\big\} = \big\{(\mathbf{j}, iv) \mid f'(\mathbf{j}, iv)\big\} \quad (3)$$

The Presburger formula f' relates instances of the target statement $\mathbf{j}$ to the value of $iv = i_k$ of the source statement. Using common polyhedral code generation techniques [26] we can generate the expression $\overrightarrow{iv}$ from $v(\overrightarrow{iv})$ which can then be evaluated in the target statement.

Algorithm 1 Expression rewrite algorithm

```
1:  procedure REWRITE( $e : \langle rexp \rangle,\ w{\longrightarrow}r$ )
2:    switch $e$ do
3:      case $c : \langle constant \rangle$:  return $c$
4:      case $p : \langle param \rangle$:  return $p$
5:      case $iv$: $\langle iv \rangle$:  return $\overrightarrow{iv}$        see equation (3)
6:      case $l \odot r : \langle rexp \rangle \times \langle rexp \rangle$:    recurses for $\langle aexp \rangle$
7:        return REWRITE($l, w{\longrightarrow}r$) $\odot$ REWRITE($r, w{\longrightarrow}r$)
8:      case $A[e_1][\dots][e_n] : \langle acc \rangle$:
9:        $f_k \leftarrow$ REWRITE($e_k,\ w{\longrightarrow}r$)        $1 \le k \le n$
10:       return $A[f_1][\dots][f_n]$
11:   end switch
```

To derive the complete propagation expressions $\overrightarrow{e_w}$ we use the REWRITE procedure presented in Algorithm 1 on the original expression e_w and the RAW dependence $w{\longrightarrow}r$. Note that rewriting with the RAW dependence $w{\longrightarrow}r$ will generate a correct but potentially not minimal propagation expression. Instead, our implementation uses the propagation dependences $w{\rightharpoonup}r \in w{\longrightarrow}r$ defined below.

3.2 Propagation Dependences

Depending on the propagation expressions and the program, any subset of a RAW dependence, including the empty one, can be a maximal propagation dependence. While propagation expressions are sensitive to induction variables in the original expression, propagation dependences are sensitive to contained read accesses. The value read by an access can change if there are intermediate writes between the source and target iteration of the propagation dependence.

To identify conditions under which a read access will yield the same result in a different statement instance we perform a *reload test*. The resulting conditions restrict the propagation dependence to a subset of the *RAW* dependences.

Reload Test The *reload test* determines the subset of the RAW dependence $w{\longrightarrow}r$ that is a valid propagation dependence $w{\rightharpoonup}r$. For this subset each rewritten read access $\overrightarrow{r_{e_w}}$ will result in the same value in the target as the original read r_{e_w} contained in the written expression e_w does in the source:

$$\forall r_{e_w} \in e_w : \forall (\mathbf{i}, \mathbf{j}) \in w{\rightharpoonup}r : [\![r_{e_w}]\!]_{\mathbf{i}} = [\![\overrightarrow{r_{e_w}}]\!]_{\mathbf{j}}$$

This equation holds if there are no intermediate writes to the read location which are located between the original access in iteration $\mathbf{i}$ and the potential reload in iteration $\mathbf{j}$.

In order to determine the intermediate writes we first compute a mapping from instances of *potential intermediate writes* to target statement instances they might precede. This mapping, $PIW(r_{e_w})$, is computed with regards to the RAW

```
UNSAT = c₁ ⊕ ... ⊕ cₙ;
c₁ = 0; ...; cₙ = 0;
SAT = !UNSAT;
```

(a) Encoding of the conjunction $c_1 \wedge \ldots \wedge c_n$. UNSAT can only be propagated if all disjunctions have been replaced. If that is the case, the formula is satisfiable, otherwise it is not.

```
cᵢ = l₁ ⊕ ... ⊕ lₖ;
if (witᵢ == 1)
  False[γ(l₁)] = 0; l₁ = 0;
...
else if (witᵢ == k)
  False[γ(lₖ)] = 0; lₖ = 0;
```

(b) Encoding for a disjunction $c_i = l_1 \vee \ldots \vee l_k$ that can only be propagated if the (negated) literal at position wit_i does not access the False array.

```
x⁺ = 1; x⁻ = 1;
if (Aₓ)
  x⁻ = False[γ(x⁻)];
else
  x⁺ = False[γ(x⁺)];
```

(c) Encoding for a literal x in positive ($x^+ := x$) and negative ($x^- := \neg x$) form. The condition A_x is a parameter that determines the assignment of x.

Figure 6. Encoding rules for a *k-CNF-SAT* formula as a program that allows maximal polyhedral expression propagation to solve satisfiability. All variables except UNSAT are live-out. The $\gamma(\cdot)$ function is a constant valued, injective enumeration for literals l_i which can be positive x^+ or negative x^- uses of a variable x, thus $\gamma(\cdot)$ is not present in the final program.

dependence $w \longrightarrow r$ and all WAR dependences emanating from r_{e_w} denoted as $r_{e_w} \longrightarrow *$. $PIW(r_{e_w})$ maps instances of writes, here $\mathbf{l}$, to the instances of $\overrightarrow{r_{e_w}}$ in the target statement if the write instance overwrites r_{e_w} in the source statement, thus:

$$PIW(r_{e_w}) := \{(\mathbf{l}, \mathbf{j}) \mid \forall\, \mathbf{i}.\, (\mathbf{i}, \mathbf{l}) \in r_{e_w} \longrightarrow * \wedge (\mathbf{i}, \mathbf{j}) \in w \longrightarrow r\}.$$

An intermediate write exists if and only if the write, hence the first component of $PIW(r_{e_w})$, precedes the reload, thus the second component. Using $PIW(r_{e_w})$, the schedule θ and the lexicographic ordering $\ll_{lex}$ of instances we define the set of overwritten reload access instances $OR(r_{e_w})$ as follows:

$$OR(r_{e_w}) := \{\, \mathbf{j} \mid \forall\, \mathbf{l}.\, (\mathbf{l}, \mathbf{j}) \in PIW(r_{e_w}) \wedge \theta(\mathbf{l}) \ll_{lex} \theta(\mathbf{j})\}.$$

It is now possible to compute the propagation dependence $w \rightharpoonup r$ from the *RAW* dependence $w \longrightarrow r$ by eliminating all iteration pairs that contain an overwritten reload:

$$w \rightharpoonup r := \{(\mathbf{i}, \mathbf{j}) \in w \longrightarrow r \mid \forall r_{e_w} \in e : \mathbf{j} \notin OR(r_{e_w})\}.$$

Self-WAR Dependences In contrast to other polyhedral optimizations we have to include same iteration self-overwrites such as A[i]=A[i]+1 in the set of WAR dependences. The write of the propagation dependence is at the same time an intermediate write for a contained read access, thus propagation is generally impossible. However, it is possible that propagation allows to eliminate all propagated write instances if the write itself is later overwritten, thus if it is not live-out.

3.3 Limitations & Extensions

While propagation dependences (ref. Equation 1) and expressions (ref. Equation 2) in Definition 3.1 are kept general, we limit their identification in two main regards:

1. Propagation expressions will read exactly the same memory locations as the original source expression. We especially do not introduce new accesses to communicate overwritten values.
2. Propagation dependences are determined using the given schedule which is not altered during the process.

Both limitations can hinder propagation as illustrated in Figure 7. However, for streaming codes that do not reuse temporary memory locations but merely store intermediate results once, these limitations will not prohibit propagation.

```
tmp[0] = 1; tmp[1] = 1;
for i = 2 to N do
  tmp[i] = tmp[i-1]
         + tmp[i-2];
done
out = tmp[N];
```

(a) Naive Fibonacci computation that requires two new scalar variables to hold intermediate results in order to allow propagation and elimination of the temporary array tmp.

```
for i = 0 to N do
  tmp[i] = A[i];
done
for j = 0 to N do
  A[j] = 0;       // S
  B[j] = tmp[j];  // T
done
```

(b) Propagation prohibiting overwrite in statement S that could be avoided by an interchange of statement S and T.

Figure 7. Examples illustrating the limitations of the propagation expression and dependence construction as described in Section 3.1 and Section 3.2.

3.4 Complexity Analysis

Maximal expression propagation can be used as a solver for *k-CNF-SAT* formulae, thus maximal expression propagation is *NP-hard*. The polynomial encoding for a *CNF* formula into a program in our input language is sketched in Figure 6. The outer conjunction is translated to an (arbitrary) operation referencing the results of all disjunctive clauses $c_1, \ldots, c_n$ as illustrated in Figure 6a. The expression result is stored in the *only non live-out variable* named UNSAT. Afterwards all clause results are overwritten. Consequently, propagation of UNSAT is possible if and only if prior propagation happened for all $c_1, \ldots, c_n$. Each clause result c_i is defined as an (arbitrary) operation on the contained (negated) literals $l_1, \ldots, l_k$ as shown in Figure 6b. Propagation of a clause result is only possible if the use of the witness literal l_{wit_i} was replaced by its definition and that definition does not access the False array at position $\gamma(l_{wit_i})$. Note that wit_i is a parameter and $\gamma(\cdot)$ a constant valued, injective enumeration function for literals that is not present in the code but used only for the construction. We define a literal l_i as x^+ if it is a positive use of x and as x^- if it is a negated use. In Figure 6c the positive and negative occurrences of a literal x are encoded as a constant value and an access to the False array depending on the parameter A_x which determines the *assignment of x*. Both literal forms can be propagated to the definition of a clause but only the form that was assigned a constant can justify the propagation of the clause.

Maximal expression propagation will explore all possible assignment combinations by splitting statements based on the values of the parameters $\mathtt{A}_x$ and $\mathtt{wit}_i$. Each fulfilling variable assignment is determined by the parameter values of $\mathtt{A}_x$ in the iteration domains of the statement splits in which UNSAT was made obsolete after propagation. To this end, we assume the propagation algorithm will record all iteration domains for which the non live-out variable UNSAT can be eliminated and thereby also record all fulfilling assignments.

Note that the encoding has to happen first for all literals (Figure 6c), then for all disjunctions (Figure 6b) and then for the conjunction (Figure 6a).

4 Heuristics

While expression propagation, as presented in Section 3, will never increase the required memory or introduce new scheduling constraints, it can easily decrease performance. An increased cache miss rate and additional computation are the most common causes, but statement splitting can also result in complex loop structures that are hard to execute efficiently. In order to determine if propagation is beneficial we devised several heuristics to guide our propagation algorithm that are explained in the following.

4.1 Non Live-Out Memory Heuristic

A main source of optimization potential for expression propagation stems from the elimination of all RAW dependences for non live-out, e.g., temporary or overwritten, memory. If all RAW dependences of such locations have been eliminated the original computation as well as the write accesses become obsolete. Elimination of write accesses can lead to less WAR and WAW dependences, decrease the overall memory requirement and the cache contention. While propagation via live-out memory is not harder to do, there is generally less performance to gain.

4.2 Cache Miss Heuristic

The goal of the *cache miss heuristic* is to limit the number of required cache lines per iteration. In order to approximate the number of required cache lines we will assume that each access is located in the middle of a cache line as illustrated in Figure 8. Consequently, we can assume that accesses to close-by memory locations will cause cache hits. For the example shown in Figure 8, all accesses between A[i-3] and A[i+3] will be assumed to cause cache hits after A[i] was accessed. The heuristic will iterate over all memory accesses contained in the target statement and those that would be after propagation. Note that the new accesses have to be adjusted according to the propagation dependence and that the accesses replaced by propagation should not be considered. The number of cache lines needed per iteration is equal to the number of accesses that do not hit a line that is assumed to be already in the cache. If an access is considered a cache miss the locations surrounding the accessed one will be assumed to be cached for all accesses to come.

The number of allowed cache misses per iteration is hardware dependent and determined for each architecture using measurements similar to the ones shown in Section 6.1.

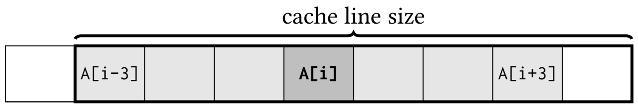

Figure 8. Memory locations that are assumed to be cached (in gray) after A[i] (dark gray, center) was accessed. In the picture the cache line size is 8 times the element size of A.

4.3 Code Complexity Heuristic

A key property of our propagation scheme is the ability to propagate to a part of a statement. Syntactic read replacement requires this part to be split off as described in Section 3. This statement splitting can grow the number of statements, and thereby dependences, exponentially. Additionally, splits can severely increase the complexity of the generated code. In particular, it might cause loops to be duplicated with a single specialized statement instance in-between. Such loops are less often vectorized, due to a smaller trip-count, and can increase the synchronization overhead if they are separately parallelized. To limit code complexity we restrict single instance specialization in the following way: Instances are only specialized for a single iteration if the fixed dimension is not in-between two non-fixed, thus loop dimensions.

4.4 Propagation Algorithm

Expression propagation was introduced as an optimization performed per RAW, or propagation, dependence. However, if it is guided by heuristics, some outgoing RAW dependences of a non live-out location might get propagated while others might be considered not beneficial. Since the elimination of non live-out memory is a major performance factor we want to avoid such partial propagations. To this end, we applied propagation, as well as the heuristics, not to a single dependence but to the set of all RAW dependences emanating from a non live-out array. Propagation is therefore only performed if it is *possible* and deemed *beneficial* for *all* dependences of this location. Consequently, if a non live-out array is propagated, all writes to the location can be removed.

Our algorithm tries to propagate each array only once. The order in which arrays are visited is described now.

Propagation Order The order in which expressions are propagated is important due to the effects on the heuristic results as well as for legality. In Figure 9, both effects are illustrated. For our implementation we chose to propagate the temporary location first that will minimize the number of different arrays accessed in the target statements. If there are multiple locations, the one with the least amount of RAW dependences is chosen. If there is still no unique location, we pick the one with less: source statements, read accesses, dimensions and then number of incoming RAW dependences. The final tie breaker is the syntactic ordering in the source.

```
s0 = A[i];
s1 = s0 + B[i];
A[i] = ...;
C[i] = s1;
```

(a) Propagation of one scalar, `s0` or `s1`, prohibits propagation of the other one.

```
t0 = A[i] + C[i];
t1 = B[i] + D[i];
Out[i] = t0 + t1;
```

(b) Propagation of one temporary location, `t0` or `t1`, prevents beneficial propagation for the other one if only 4 cache misses are allowed.

Figure 9. Examples illustrating the impact of propagation order on propagation legality (9a) and benefit heuristics (9b).

5 Implementation Details

Expression propagation is implemented in LLVM's [34] polyhedral optimizer Polly [23]. To maximize performance we augmented both as described below.

5.1 Live-Out Access Analysis

We implemented a simple, intra-procedural live-out access analysis to identify non live-out locations. The analysis can deal with scalars, stack locations, internal globals and heap locations for which the deallocation is in the current function and immediately post-dominating the analyzed region.

5.2 Loop Parallelization

Polly is capable of parallelizing loops using OpenMP [7]. By default, the outermost parallel loop is chosen even if the loop trip count prevents full utilization of the machine. To avoid undersubscription we will not parallelize outermost loops if there is a more suitable one nested inside. In contrast to the default setting we also parallelize innermost loops as well as loops with non-affine write accesses. The latter is justified by the annotations present in the benchmark sources.

5.3 Scheduling and Tiling

Polly does perform polyhedral scheduling [9] and tiling [23]. However, by default it will not perform smart loop fusion [8], choose suitable tile sizes [3, 27, 59] or do a combination thereof [40]. In our experiments we noticed performance regressions when tiling with a fixed tile size. Additionally, the default scheduling algorithm did not perform well after statements were split. Consequently, we disabled loop tiling altogether and modified the scheduling objective. While Polly tries to create independent loop nests for each statement (split), we merge all write accesses to the same array into one loop nest. This scheduling choice is similar to the naive inputs. However, our scheduling and tiling choices are far from optimal and need to be revisited. The evaluation in Section 6.3 shows that they perform better than the defaults.

5.4 Higher-Order Recurrences

Recurrences are scalar variables that communicate values from one iteration to the next [30]. We augmented code generation in order to generate recurrences for consecutive accesses that have been replaced by propagation expressions. While remaining consecutive read accesses can also be communicated this way, it is often better to keep them. Such accesses will cause low-level cache hits while recurrences will inevitably increase register pressure. Higher-Order recurrences, i.e., recurrences that communicate values based on other recurrences, are especially useful to avoid recomputation. An example is shown in Figure 10. Since the LLVM loop vectorizer is limited to single-level recurrences we had to extend it to handle higher-Order recurrences as well.

```
for i = 0 to N do
  tmp[i] = f(i);
done
for j = 1 to N-1 do
  Out[j] = tmp[j-1]
   + tmp[j] + tmp[j+1];
done
```

(a) Consecutive accesses that can be eliminated by propagation.

```
t0 = f(0); t1 = f(1);
for j = 1 to N-1 do
  t2 = f(2);
  Out[j] = t0 + t1 + t2;
  t0 = t1;
  t1 = t2;
done
```

(b) Recurrences used to communicate intermediate values.

Figure 10. Higher-order recurrences reduce computation overhead after expression propagation. Naive code generation would triple the number of instances of f, but recurrences allow propagation with only one evaluation of f.

5.5 Language Extensions

Our input language is restricted to single-write statements to simplify the implementation and to allow fine-grained scheduling. Such statements can be created from multi-write statements by a single reverse traversal that splits the statement after each contained write access. The reverse order is important as each split can induce new scalar "writes" in the preceding part that communicate values via fresh temporaries to the split-off remainder.

Polly offers approximations for non-affine access functions and control regions [38]. While enabled, we do not propagate into or out of such accesses or regions.

Aliasing, integer overflows and other problems that arise in real programs are already handled by Polly [14].

6 Evaluation

The evaluation compares the following optimization schemes and is performed for the 7 benchmarks listed in Table 1. These benchmarks have also been used to evaluate the PolyMage tool and are available online [45].

- (vanilla[1]) Polly [23], the polyhedral optimizer available for LLVM [34] that is also the basis for our approach,

[1]Modifications were necessary to represent, and optimize the benchmarks.

Table 1. Benchmark details [40] including the number of arrays, loops, statements and accesses for the *Naive* (N), *Polly* (P), *PolyMage* (PM) and *Expression Propagation* (EP) version. The lines of code (LOC) are measured after code formatting.

Benchmark			*# Arrays*			*# Loops*				*# Statements*				*# Accesses*			
#	Name	LOC	N/P	PM	EP	N	P	PM	EP	N	P	PM	EP	N	P	PM	EP
(1)	unsharp mask	96	5	2	2	12	12	14	3	4	4	4	1	20	21	20	28
(2)	harris	153	12	2	2	22	22	8	2	11	11	3	1	65	61	159	26
(3)	bilateral grid	502	7	5	4	25	25	29	18	8	7	8	6	90	39	90	57
(4)	camera pipe	1114	30	6	5	67	67	84	4	41	35	41	13	288	146	276	675
(5)	pyramid blending	1501	43	11	14	123	123	184	31	58	52	58	23	272	268	272	967
(6)	interpolate	1801	39	27	19	186	186	292	71	182	182	182	53	294	294	294	299
(7)	local laplacian	7018	69	44	37	461	461	687	168	814	626	537	441	1533	1212	1070	3846

- optimized code versions generated by PolyMage [40], the state-of-the-art polyhedral optimization tool for concatenated stencil computations, and Halide [44],
- our Polly fork that was modified as described in Section 5 but without expression propagation, and
- our Polly fork with automatic expression propagation as explained in Section 4.4, and guided by the heuristics presented in Section 4.1– 4.3.

To showcase both scalability (4 – 20 threads) as well as portability with regards to the cache size, ISA, and execution model (in-order vs. out-of-order) we performed tests on 4 different CPU architectures presented in Table 2.

For the Polly based approaches as well as PolyMage we used the same LLVM/clang version (close to v4.0.1)[2]. We choose this setup to compare the effects caused by the specific optimizations rather than artifacts that arise due to different vectorization or register allocation schemes employed by the compilers. Additionally, we present PolyMage results compiled with gcc (v7.2.0) and icc (v18.0.1 20171018) on the Xeon E3-1225v3 architecture in Table 4. LLVM/clang and gcc are invoked with "-fopenmp -ffast-math -march=native -O3", icc with "-qopenmp -fp-model fast=1 -ftz -xhost -O3".

Our Polly fork, expression propagation code and evaluation scripts are available online at https://github.com/cdl-saarland/PolyhedralExpressionPropagation.

Table 2. Architecture details including the CPU name, number of threads (#T), vector size in bits (Vec), first (L1) and last level cache size (LLC) as well as the execution model (Exec).

CPU	#T	Vec	L1	LLC	Exec
Cortex A53	1×4	128	16 KiB	0.5 MiB	In Order
Cortex A57	1×4	128	24 KiB	2 MiB	OoO
Xeon E3-1225v3	1×4	256	128 KiB	8 MiB	OoO
Core i9-7900X	2×10	512	320 KiB	13 MiB	OoO

6.1 Evaluation of the Heuristics

Before we compare the performance across the optimization approaches we discuss the impact of our heuristics.

[2]Our LLVM/clang is a slightly modified version based on git commit: 1aa4ba7ed969272250c7647ff14305f5b2f32c26 (ref. Section 5.4).

Figure 11 shows the effects of different cache miss limits (ref. Section 4.2) for two target architectures. While the two smallest benchmarks (1) and (2), see numbers on the right of the plot, will yield the best performance after propagation of all temporary arrays, the others show different behaviour. Especially number (3) and (5) are interesting as they are not restricted by the code complexity heuristic (ref. Section 4.3). Instead, the performance improves or regresses if an increased cache miss limit allows more propagations. Depending on the architecture, the performance can regress compared to the baseline. This happens if more resources, e.g., registers, issuing ports or prefetcher streams, are used than available. Later improvements are possible since array elimination generally decreases the runtime while the resource contention does not necessarily increase.

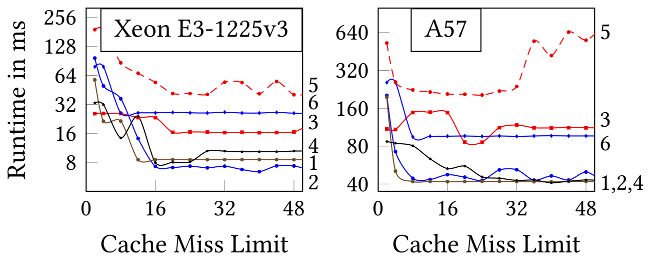

Figure 11. Effect of cache miss limits (Section 4.2) on the runtime for the Intel Xeon E3-1225v3 and ARM Cortex A57.

Propagation without the code complexity heuristic (ref. Section 4.3) has a similar negative effect as a very large cache miss limit. However, mainly benchmarks (6) and (7) are affected. Without the complexity restriction, the performance of interpolate (6) varies on Xeon from a speedup of up to 3.6× to a slowdown of up to 17.8×. This shows that the impact of expression propagation can vary significantly even for a fixed benchmark and architecture. Consequently, the heuristics, including the propagation order (ref. Section 4.4), have to be chosen carefully and tuned to the architecture to prevent regressions. At the same time it is crucial to account for other factors, e.g., tiling and scheduling (ref. Section 5.3) or the use of recurrences (ref. Section 5.4), since they have a non-trivial impact on the profitability of a propagation.

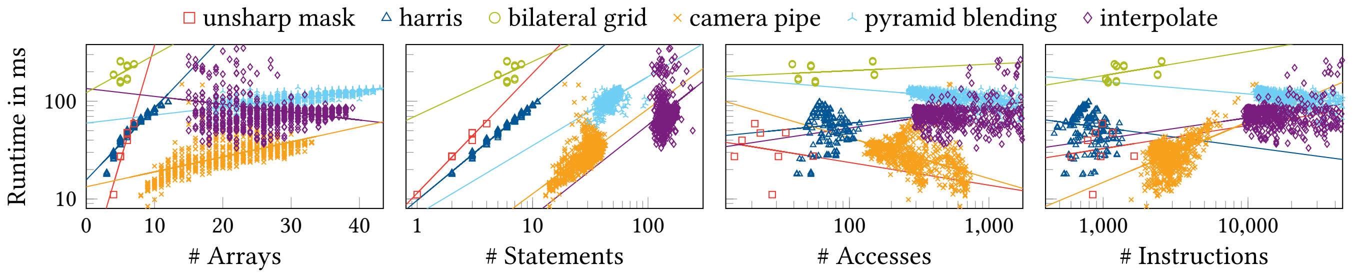

Figure 12. Correlation between runtime and the number of arrays, statements, accesses, and instructions for the Xeon E3-1225v3 architecture. A positive regression line slope indicates that minimizing the number generally improves performance. However, note that the absolute value of the slope is highly dependent on the scaling and the shown range.

In addition to the heuristics discussed so far, we experimented with other strategies including the approximated number of required registers or the number of accesses. We also altered the propagation orders based on these and similar factors. While we did observe up to 20% better performance, the overall results were always inconclusive as illustrated in Figure 12. The four plots show measurements taken after expression propagation of randomly chosen combinations of temporary arrays up to a combination size of 31. For each size we generated up to 31 unique sets of temporary arrays that correspond to the data points in the plot. The reported runtime is the median observed in 31 runs for one combination. The regression lines in the first two graphs indicate that propagation and the consequent array and statement elimination has a positive impact on the runtime. The only exception is interpolate (6) for which propagation can easily cause too many statement splits. If it does not, the number of statements is minimized and propagation is beneficial (ref. second graph). The two graphs on the right show that neither the number of accesses nor the number of instructions have a clear impact on the runtime. Three of the benchmarks show speedups while the number of accesses increases and the same holds true for two benchmarks with regards to the number of instructions. In Table 3 we provide the performance results of the naive, the heuristic guided as well as the best and worst observed version in this experiment.

Table 3. Performance results (in ms) including the limits observed in the random propagation tests shown in Figure 12.

Version \ Bench.	(1)	(2)	(3)	(4)	(5)	(6)
Naive	47.8	82.1	24.5	40.4	114.0	72.1
Expression Prop.	8.5	7.2	16.3	10.5	53.7	26.5
rnd. prop. best	8.5	7.2	16.3	8.4	55.2	33.2
rnd. prop. worst	59.0	98.6	25.7	149.4	138.6	1285

We believe that a good performance model for expression propagation is a challenging task and a research topic on its own. To this end, we choose the simple, conservative combination of heuristics explained in Section 4.1–4.3 to demonstrate the potential of expression propagation. In order to maximize performance it is crucial to use a more elaborate cost model and to revisit the scheduling and tiling choices.

6.2 Memory Requirement and Code Complexity

Expression propagation is not only a performance optimization but also capable of eliminating temporary arrays and thereby reducing the memory requirements of an application. Table 1 lists the number of parametric sized arrays remaining after the optimizations, together with other static properties of the optimized program. Both PolyMage and expression propagation decrease the number of arrays for each benchmark. However, expression propagation completely eliminates temporary arrays while PolyMage often replaces them with smaller, constant-sizes, ones that hold the intermediate results for one tile at a time.

As noted earlier, two of the benchmarks perform best if all temporary arrays have been removed. While this is certainly not true for the three largest benchmarks, it already does not hold for the rather small bilateral grid (3) implementation. This benchmark performs best if two of the three affine arrays are eliminated and one is kept. This trade-off is not the only problem when minimal memory usage is the only propagation goal. In our experiments there was a significant increase in the number of accesses (per statement). While the cache miss heuristic does indirectly limit this number, we already observe a notable growth that causes an even bigger increase in the number of dependences. Consequently, propagation purely guided by memory elimination does not scale above the size and complexity of programs like pyramid blending (5) or interpolate (6).

6.3 Automatic Expression Propagation

Figure 13 shows the performance of all five optimizations on the different platforms. The graphs are normalized to the results of a naive parallel implementation which is also the input for the Polly-based optimization schemes[3]. Measurements were taken 51 times and the median result is reported. The PolyMage version [45] was optimized for a processor similar to the Xeon E3-1225v3 and it was not specialized for the other architectures. However, it is important to note that Polly does not employ target dependent tile sizes and tiling is disabled for our forked Polly and expression propagation.

[3]OpenMP pragmas are stripped from the inputs of Polly-based schemes.

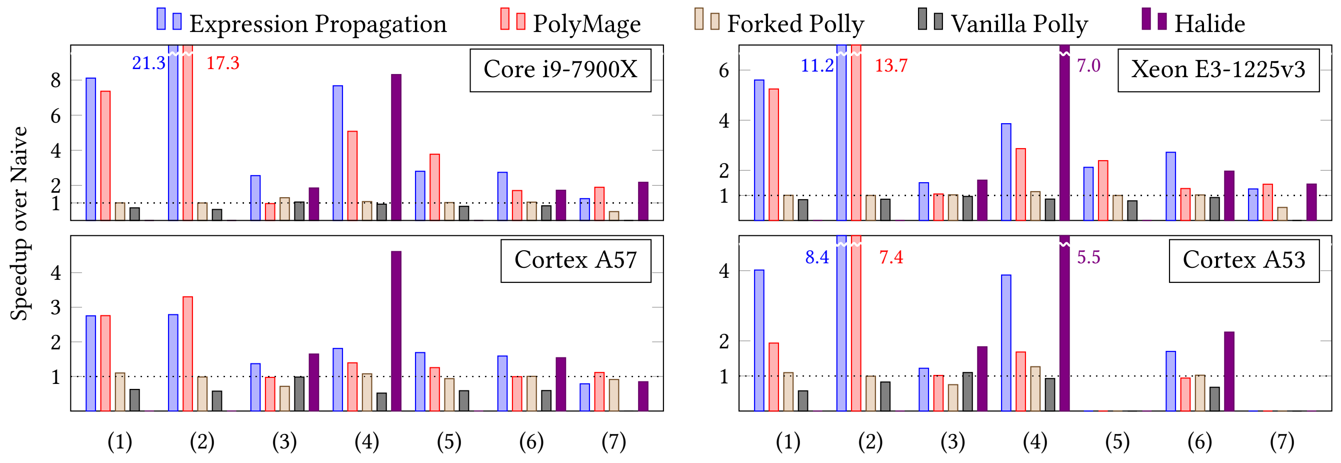

Figure 13. Performance of different schemes (ref. Section 6) normalized to a naive parallel version. The Vanilla Polly bar is missing for benchmark (7) due to a code generation issue. The bars for Cortex A53 are missing due to insufficient main memory. For Halide only the results of available benchmarks (https://github.com/halide/Halide/tree/release_2017_05_03/apps) are reported.

Our Polly fork (ref. Section 5) outperforms the vanilla version except for one benchmark ((3) on A53) which benefits from tiling. At the same time there is only one case ((7) on A57) where it performs better than expression propagation.

PolyMage [40] is faster in 7 cases and consistently better for local laplacian (7). Expression propagation generates the faster version in 18 cases and is, except for local laplacian on A57, consistently faster than the naive implementation. The highest speedups are achieved for the rather simple harris benchmark (2). Though, depending on the system, either the tiled PolyMage code with two constant sized temporary arrays or our fully propagated version performs best.

Halide [43] results are available for the benchmarks contained in the 2017/05/03 release[4]. While the performance for camera pipe (4) is consistently the best, our approach comes close on the Core i9 architecture. The performance of the remaining benchmarks is comparable to expression propagation and it depends on the platform which one performs best. However, it is important to consider that Halide and PolyMage perform more complex optimizations, including tiling and (non-trivial) scheduling which have not been explored in this work. Note that all expression propagation speedups are only due to the elimination of temporary arrays. There is especially no increase in parallelism, e.g., due to propagated scalars, in these benchmarks.

Real World Application Expression propagation is often limited by the live-out information available. In real world code only scalars are easy to spot non live-out values. Nevertheless, propagation of scalars is important to eliminate false dependences and increase scheduling freedom. In the SPEC2000/2006 benchmark suites our technique eliminates 63%/60% of all scalar read accesses in loop nests analyzable by our Polly fork.

7 Related Work

Iteration domain splitting is performed by multiple polyhedral optimization approaches [22, 38]. However, we do it to perform syntactic read replacement (ref. Section 3) and did not further investigate the effects on scheduling.

Predictive commoning [30] is an optimization that employs higher-order recurrences to avoid recomputation of expressions in loops (ref. Section 5.4). Our polyhedral version is especially powerful as it is only applied to propagated accesses, thus large expressions.

The elimination of memory writes to non live-out locations is a form of dead assignment elimination that has been investigated for both scalars [32] and arrays [53]. In contrast to such elaborate approaches we do not iterate the detection of dead assignments in order to identify transitively dead ones. While this might become necessary for more complex programs it was not needed for our evaluation.

Dependence elimination is an important research topic in the context of polyhedral optimization. Various techniques are concerned with the removal of false dependences (WAR and WAW) in order to enable optimizations that were otherwise prohibited [1, 10, 37]. While expression propagation explicitly removes true (RAW) dependences, it implicitly eliminates false dependences as well. However, only if they are caused by non live-out locations that can be propagated. An example that is similar to the uses cases of other approaches was already provided in Figure 1.

Programs in dynamic single assignment form (DSA) [17, 31] write every memory location at most once and are consequently free of false dependences. In contrast to other propagation techniques [28, 39, 43, 51], we do not require the program to be in DSA form. Instead we construct propagation dependences with regards to possible intermediate writes as described in Section 3.2. Nevertheless, expression

[4]Online: https://github.com/halide/Halide/tree/release_2017_05_03/apps

Table 4. Raw performance results for the Xeon E3-1225v3. The best performer is highlighted in bold. The Vanilla Polly result is missing for benchmark (7) due to a code generation issue. For Halide only the results of available benchmarks (https://github.com/halide/Halide/tree/release_2017_05_03/apps) are reported. Note that PolyMage is listed as (PM).

#	Name	Naive	Vanilla Polly	Forked Polly	Expr. Prop.	PM clang	PM gcc	PM icc	Halide
(1)	unsharp mask	47.79 ms	57.55 ms	47.55 ms	**8.53 ms**	9.11 ms	9.01 ms	9.00 ms	n/a
(2)	harris	82.12 ms	96.82 ms	82.23 ms	7.29 ms	**5.99 ms**	6.59 ms	9.61 ms	n/a
(3)	bilateral grid	24.53 ms	25.55 ms	24.00 ms	16.29 ms	23.21 ms	28.52 ms	26.20 ms	**15.25 ms**
(4)	camera pipe	40.40 ms	47.38 ms	35.13 ms	10.46 ms	14.09 ms	13.95 ms	11.68 ms	**5.76 ms**
(5)	pyramid blending	113.97 ms	145.554 ms	114.19 ms	53.73 ms	47.76 ms	45.94 ms	**41.40 ms**	n/a
(6)	interpolate	72.14 ms	78.83 ms	70.77 ms	**26.51 ms**	56.62 ms	40.99 ms	41.08 ms	36.64 ms
(7)	local laplacian	140.46 ms	n/a	270.30 ms	111.62 ms	97.11 ms	86.29 ms	**82.39 ms**	96.71 ms

propagation is generally more effective in the absence of false dependences as illustrated by Figure 7b in Section 3.3.

Transformation into DSA form will increase the memory requirement of a program. To alleviate this increase, memory optimization techniques [6, 12, 13, 35, 55] are used after the DSA based optimizations. These techniques perform scheduling on the memory access functions in order to reuse memory locations for different intermediate results. In contrast, expression propagation never changes the accessed memory but the “time and place” where intermediate results are computed, thereby eliminating the need for storage locations altogether. To this end, expression propagation can be used as a pre-processing step to reduce the number of stored intermediate results while keeping the computations manageable for the target hardware. Additionally, it is interesting to look into the combination of memory scheduling techniques and expression propagation to overcome the limitation illustrated by Figure 7a in Section 3.3.

PolyMage [40] and Halide [39] perform, among other things, a limited form of expression propagation. However, in contrast to our technique, both will need to fuse the producer and consumer loop, which contain the definition and respectively the *single* user of a temporary value, in the process. Additionally, PolyMage did require both access functions to be equal. If it is possible to generate a schedule to equalize the access functions, loop fusion is applied and expression propagation is reduced to a redundant load optimization limited to a single loop iteration. Alternative techniques [16, 43] perform similar redundant load optimizations across different iterations of a single loop. The Julia language [5] offers syntax to force loop fusion but relies on the compiler backend to perform redundant load optimizations afterwards.

For non-surjective propagation dependences statement splitting was already introduced [51, 57].

Existing techniques that propagate array locations are limited to DAG pipelines [28, 39, 44] and they do not employ a similarly precise dependence analysis. Others handle special expressions e.g., constants [46, 47, 56] or single array reads [51, 57]. In the case of constants or array reads in DSA form programs [28, 39, 44, 51], propagation dependences are trivially equal to the RAW dependences. An alternative approach to detect unsound propagations is to compare initial and resulting flow dependences [57].

8 Conclusion

In this work we present a propagation technique for arbitrary expressions that are communicated via affine-indexed, multi-dimensional arrays in a sequence of non-perfect loop nests. In principle, expression propagation allows for removing temporary arrays, reduce the pressure on the memory system, and create additional parallelism.

We define expression propagation formally and show that maximal expression propagation is NP-hard. Therefore, we present a set of heuristics to guide the propagation.

Our experimental evaluation on a set of image processing pipelines and four different architectures show that expression propagation can significantly outperform existing polyhedral techniques but is sensitive to various parameters that guide its application. Consequently, we need further research on an appropriate performance model to apply expression propagation dependably.

Acknowledgments

We would like to thank Ravi Teja Mullapudi, Vinay Vasista and Uday Bondhugula for their publicly available PolyMage benchmarks [45]. We also appreciate the feedback we got from Uday Bondhugula, Abhinav Jangda and our reviewers.

References

[1] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. 2013. Improved Loop Tiling Based on the Removal of Spurious False Dependences. *ACM Trans. Archit. Code Optim.* (2013).
[2] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. 2010. Putting Automatic Polyhedral Compilation for GPGPU to Work *(CPC'10)*.
[3] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism *(ICS'12)*.
[4] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. 2003. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing*.
[5] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. 2012. Julia: A fast dynamic language for technical computing. *arXiv* (2012).
[6] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization *(POPL'16)*.

[7] OpenMP Architecture Review Boards. 2017. The OpenMP API specification for parallel programming. http://www.openmp.org/. (2017).
[8] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler *(PACT'10)*.
[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer *(PLDI'08)*.
[10] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. 1997. Plugging anti and output dependence removal techniques into loop parallelization algorithm. *Parallel Comput.* (1997).
[11] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program *(POPL'78)*.
[12] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. 2016. Extended Lattice-based Memory Allocation *(CC'16)*.
[13] Alain Darte, Robert Schreiber, and Gilles Villard. 2005. Lattice-Based Memory Allocation. *IEEE Trans. Comput.* (2005).
[14] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic Loop Optimization *(CGO'17)*.
[15] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's Polyhedral Scheduling in the Presence of Reductions *(IMPACT'15)*.
[16] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1993. A practical data flow framework for array reference analysis and its use in optimizations. *ACM Sigplan Notices* (1993).
[17] P. Feautrier. 1988. Array Expansion *(ICS'88)*.
[18] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. of Parallel Programming* (1991).
[19] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *Int. J. Parallel Program.* (1992).
[20] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.* (1992).
[21] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*.
[22] Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index Set Splitting. *Int. J. Parallel Program.* (2000).
[23] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* (2012).
[24] T. Grosser and T. Hoefler. 2016. Polly-ACC: Transparent compilation to heterogeneous hardware *(ICS'16)*.
[25] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. 2015. Optimistic Delinearization of Parametrically Sized Arrays *(ICS'15)*.
[26] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* (2015).
[27] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2017. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. *Computing Research Repository (CoRR)* (2017).
[28] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* (2014).
[29] F. Irigoin and R. Triolet. 1987. *Computing dependence direction vectors and dependence cones with linear systems.* Tech. Rep.
[30] Kevin O'Brien. 1990. *Predictive Commoning: A method of optimizing loops containing references to consecutive array elements.* Technical Report. IBM Thomas J. Watson Research Center.
[31] Bart Kienhuis. 2000. *MatParser: An array dataflow analysis compiler.*
[32] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination *(PLDI'94)*.
[33] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation *(PLDI'13)*.
[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation *(CGO'04)*.
[35] Vincent Lefebvre and Paul Feautrier. 1997. Optimizing Storage Size for Static Control Programs in Automatic Parallelizers *(Euro-Par'97)*.
[36] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. 1993. Array-data Flow Analysis and Its Use in Array Privatization *(POPL'93)*.
[37] Sanyam Mehta and Pen-Chung Yew. 2016. Variable Liberalization. *TACO* (2016).
[38] Simon Moll, Johannes Doerfert, and Sebastian Hack. 2016. Input Space Splitting for OpenCL *(CC'16)*.
[39] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* (2016).
[40] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines *(ASPLOS'15)*.
[41] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based Data Reuse Optimization for Configurable Computing *(FPGA'13)*.
[42] William Pugh and David Wonnacott. 1993. An Exact Method for Analysis of Value-based Array Data Dependences *(LCPC'93)*.
[43] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* (2012).
[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines *(PLDI '13)*.
[45] Uday Bondhugula Ravi Teja Mullapudi, Vinay Vasista. 2017. PolyMage benchmarks. http://mcl.csa.iisc.ac.in/polymage.html. (2017).
[46] Silvius Rus, Guobin He, and Lawrence Rauchwerger. 2005. Scalable Array SSA and Array Data Flow Analysis *(LCPC'05)*.
[47] Vivek Sarkar and Kathleen Knobe. 1998. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form *(SAS'98)*.
[48] Tingting Sha, M. M. K. Martin, and A. Roth. 2005. Scalable store-load forwarding via store queue index prediction *(MICRO'05)*.
[49] Jun Shirako, Akihiro Hayashi, and Vivek Sarkar. 2017. Optimized Two-level Parallelization for GPU Accelerators Using the Polyhedral Model *(CC'17)*.
[50] Konrad Trifunović, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-Model Guided Loop-Nest Auto-Vectorization *(PACT'09)*.
[51] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Henk Corporaal, and Francky Catthoor. 2003. Advanced Copy Propagation for Arrays *(LCTES'03)*.
[52] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model *(ICMS '10)*.
[53] Sven Verdoolaege. 2015. *PENCIL support in pet and PPCG.* Tech. Rep.
[54] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *TACO* (2013).
[55] Doran Wilde and Sanjay V. Rajopadhye. 1996. Memory Reuse Analysis in the Polyhedral Model *(Euro-Par'96)*.
[56] David Wonnacott. 1999. *Constant Propagation Through Array Variables.*
[57] David Wonnacott. 2000. Extending Scalar Optimizations for Arrays *(LCPC'00)*.
[58] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. 2013. Array Dataflow Analysis for Polyhedral X10 Programs *(PPoPP)*.
[59] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E Eichenberger, and Kevin O'Brien. 2010. Automatic creation of tile size selection models *(CGO'10)*.