WITH-Loop-Folding in SAC - Condensing Consecutive Array Operations

Sven-Bodo Scholz

Dept of Computer Science
University of Kiel
24105 Kiel, Germany
e-mail: sbs@informatik.uni-kiel.de

Abstract. This paper introduces a new compiler optimization called WITH-*loop-folding*. It is based on a special loop construct, the WITH-loop, which in the functional language SAC (for Single Assignment C) serves as a versatile vehicle to describe array operations on an element-wise basis. A general mechanism for combining two of these WITH-loops into a single loop construct is presented. This mechanism constitutes a powerful tool when it comes to generate efficiently executable code from high-level array specifications. By means of a few examples it is shown that even complex nestings of array operations similar to those available in APL can be transformed into single loop operations which are similar to hand-optimized WITH-loop specifications. As a consequence, the way a complex array operation is combined from primitive array operations does not affect the runtime performance of the compiled code, i.e., the programmer is liberated from the burden to take performance considerations into account when specifying complex array operations.

1 Introduction

The suitability of functional languages for numerical applications critically depends on the compilation of array operations into efficiently executable code. In the context of lazy functional languages, e.g. HASKELL[HAB$^+$95] or CLEAN[PvE95], it seems to turn out that this can only be achieved with strict arrays and if single threading for most of the array operations can be guaranteed statically [HSB97,Ser97]. Whereas the former restriction in most cases does not affect the style of programming, the latter requires the programmer to make opportunities for destructive array updates explicit to the compiler by the introduction of uniqueness types [SBvEP93] or state monads [LP94].

In contrast to lazy languages, SISAL[MSA$^+$85] demonstrates that strict languages can be compiled into efficiently executable code without forcing the programmer to think about single threading of array operations [Can92]. However, SISAL allows for low-level specifications only. Neither the expressive power of polymorphism, higher-order functions, and partial applications nor any high-level array operations are supported. As a consequence, despite a new syntax

SISAL offers few benefits in terms of expressiveness when compared with imperative languages, e.g. FORTRAN[Weh85] or C[KR90].

The development of the strict functional language SAC[Sch96] constitutes a different approach. Although SAC as well does not yet support higher-order functions nor partial applications it differs from SISAL in three respects. Most important, SAC offers substantial support for the specification of high-level array operations, i.e., it provides array operations similar to those available in APL[Ive62] as well as a special loop construct, the WITH-loop, which allows array operations to be specified element-wise. All of these language constructs are designed to facilitate the specification of array operations that can be applied to arrays of arbitrary dimensionalities and shapes. Furthermore, the syntax of SAC sticks as close as possible to that of C which does not only facilitate the compilation process but might also increase the acceptance of SAC by programmers that are used to program in C. Last but not least, SAC provides states, state-modifications, and I/O-operations [GS95] which are incorporated safely into the functional framework by means of uniqueness typing [SBvEP93].

The SAC compiler is supported by a type inference system based on a hierarchy of array types which allows dimension-independent SAC specifications to be specialized to shape-specific C programs. Although this concept leads to code which can be executed very efficiently in terms of runtime and space consumption [Sch97], the compilation of array operations in the current compiler version is still implemented straightforwardly. Each array operation is compiled into a separate piece of code, i.e., nestings of array operations are broken up into sequences of single array operations by the introduction of temporary arrays. Since there is no optimization between two or several consecutive array operations, the way a complex array operation is composed from the language constructs available has a strong influence on the code generated. Different specifications lead to the introduction of different temporary arrays and thus to different runtime overheads. Redundancies within consecutive array operations are not detected and thus have a direct impact on runtimes as well. As a consequence, the programmer has to know about the implementation when it comes to writing programs that can be compiled to efficiently executable code. Since this runs counter to the idea of functional programming, which is to liberate the programmer from low level concerns, an optimization is needed that tries to eliminate temporary arrays as well as redundancies of consecutive array operations whenever possible.

The basic idea is to use WITH-loops as a universal representation for array operations and to develop a general transformation scheme which allows to transform two consecutive WITH-loops into a single one. With this mechanism, called WITH-loop-folding, any nesting of primitive array operations can be transformed stepwise into a single loop construct which contains an element-wise specification of the resulting array.

The consequences of this optimization are far-reaching:

– The code generated for any complex array operation specified in SAC becomes invariant against the way it is composed from more primitive opera-

tions, i.e., the programmer is not concerned with the low level details any more.
– In combination with the application of other, well-known optimizations (e.g. constant-folding, loop unrolling, etc.) most of the redundancies which would result from a straightforward compilation can be eliminated.
– All primitive array operations of SAC that are similar to those available in APL can be defined through WITH-loops and are implemented via the standard library rather than by the compiler itself. The only array operation implemented by the compiler is the WITH-loop.
– Since the primitive array operations are defined in the standard library, they can be adjusted to the programmers needs easily.

The paper is organized as follows: While the following section gives a short introduction into the basic language concepts of SAC, Section 3 presents the basic WITH-loop-folding mechanism. An extended example is discussed in Section 4. It demonstrates nicely how even complex nestings of array operations can be folded into a simple loop construct. Some implementation issues are discussed in Section 5. These include some restrictions which are imposed on WITH-loop-folding in order to guarantee its statical applicability as well as some performance considerations. Section 6 points out the relationship of WITH-loop-folding to other optimization techniques. Some concluding remarks finally can be found in Section 7.

2 SAC - a Short Introduction

SAC is a strict, purely functional language whose syntax in large parts is identical to that of C. In fact, SAC may be considered a functional subset of C extended by high-level array operations which may be specified in a shape-invariant form. It differs from C proper mainly in that

– it rules out global variables and pointers to keep functions free of side-effects,
– it supports multiple return values for user-defined functions, as in many dataflow languages [AGP78,AD79,BCOF91],
– it supports high-level array operations, and
– programs need not to be fully typed.

With these restrictions / enhancements of C a transformation of SAC programs into an applied λ-calculus can easily be defined. The basic idea for doing so is to map sequences of assignments that constitute function bodies into nestings of LET-expressions with the RETURN-expressions being transformed into the innermost goal expressions. Loops and IF-THEN-ELSE statements are transformed into (local) LETREC- expressions and conditionals respectively. For details see [Sch96].

An array in SAC is represented by a shape vector which specifies the number of elements per axis, and by a data vector which lists all entries of the array.

For instance, a 2×3 matrix $\begin{pmatrix} 1\ 2\ 3 \\ 4\ 5\ 6 \end{pmatrix}$ has shape vector $[2, 3]$ and data vector $[1, 2, 3, 4, 5, 6]$. The set of legitimate indices can be directly inferred from the shape vector as

$$\{[i_1, i_2] \quad | \quad 0 \leq i_1 < 2, \quad 0 \leq i_2 < 3\}$$

where $[i_1, i_2]$ refers to the position $(i_1 * 3 + i_2)$ of the data vector. Generally, arrays are specified as expressions of the form

`reshape(` *shape_vector*`,` *data_vector* `)`

where *shape_vector* and *data_vector* are specified as lists of elements enclosed in square-shaped brackets. Since 1-dimensional arrays are in fact vectors, they can be abbreviated as

$$[v_1, ..., v_n] \quad \equiv \quad \texttt{reshape([}n\texttt{], [}v_1, ..., v_n\texttt{])} \qquad .$$

Several primitive array operations similar to those in Apl, e.g. `dim` and `shape` for array inspection, `take, drop, cat`, and `rotate` for array manipulation, and `psi` for array element / subarray selection, are made available. For a formal definition of these see [Sch96,Sch97].

All these operations have in common that they, in one way or another, affect all elements of the argument array(s) in the same way. To have a more versatile language construct at hands which allows to specify array operations dimension independently on arbitrary index ranges, Sac also supports so called WITH-loops. They are similar to array comprehensions as known from Haskell or Clean as well as to the FOR-loops in Sisal.

The syntax of WITH-loops[1] is outlined in Fig. 1. Basically, they consist of

$$\begin{array}{lll}
\textit{WithExpr} & \Rightarrow & \texttt{with} \;\; (\;\; \textit{Generator} \;\;) \;\; \textit{Operation} \\[1ex]
\textit{Generator} & \Rightarrow & \textit{Expr} \;\; \texttt{<=} \;\; \textit{Identifier} \;\; \texttt{<=} \;\; \textit{Expr} \\[1ex]
\textit{Operation} & \Rightarrow & \left[\;\; \{ \;\; \textit{LocalDeclarations} \;\; \} \;\; \right] \textit{ConExpr} \\[1ex]
\textit{ConExpr} & \Rightarrow & \texttt{genarray} \;\; (\;\; \textit{Expr} \;\; , \;\; \textit{Expr} \;\;) \\
 & | & \texttt{modarray} \;\; (\;\; \textit{Expr} \;\; , \;\; \textit{Expr} \;\; , \;\; \textit{Expr} \;\;)
\end{array}$$

Fig. 1. WITH-loops in Sac.

two parts: a generator part and an operation part. The generator part defines lower and upper bounds for a set of index vectors and an 'index variable' which represents a vector of this set. The operation part specifies the operation to be performed on each element of the index vector set. Two different kinds of operation parts for the generation of arrays are available in Sac (see *ConExpr* in Fig. 1). Their functionality is defined as follows:

[1] Actually, only a restricted form of the WITH-loops in Sac is presented here which suffices to explain the WITH-loop-folding mechanism. An extension to the full-featured WITH-loops is straightforward and will not be further addressed.

Let *shp* and *idx* denote SAC-expressions that evaluate to vectors, let *array* denote a SAC-expression that evaluates to an array, and let *expr* denote an arbitrary SAC-expression. Then

– `genarray(` *shp*`,` *expr*`)` generates an array of shape *shp* whose elements are the values of *expr* for all index vectors from the specified set, and 0 otherwise;
– `modarray(` *array*`,` *idx*`,` *expr*`)` returns an array of shape `shape(` *array*`)` whose elements are the values of *expr* for all index vectors from the specified set, and the values of *array*`[` *idx*`]` at all other index positions.

To increase program readability, local variable declarations may precede the operation part of a WITH-loop. They allow for the abstraction of (complex) subexpressions from the operation part.

3 Folding WITH-Loops

The key idea of WITH-loop-folding is to provide a universal mechanism that transforms functional compositions of array operations into single array operations which realize the functional composition on an element-wise basis. This avoids the creation of temporary arrays as well as redundancies within consecutive operations. Since most of the primitive array operations of SAC can be specified as WITH-loops the folding scheme not only can be applied to user-defined WITH-loops but serves as tool for the optimization of nested primitive array operations as well.

The most basic situation for such a transformation is the composition of two WITH-loops which simply map functions `f` and `g` to all elements of an array. From the well known fact that `map f` $\circ$ `map g` $\equiv$ `map ( f` $\circ$ `g)` we directly obtain:

Let `A` be an array of arbitrary shape with elements of type τ and let `f` and `g` be functions of type $\tau \rightarrow \tau$. Then

```
{...
  B = with( 0*shape(A) <= i_vec <= shape(A)-1)
      modarray( A, i_vec, f( A[i_vec] ));
  C = with( 0*shape(B) <= j_vec <= shape(B)-1)
      modarray( B, j_vec, g( B[j_vec] ));
...}
```

can be substituted by

```
{...
  C = with( 0*shape(A) <= j_vec <= shape(A)-1)
      modarray( A, j_vec, g( f( A[j_vec] ) ));
...}
```

provided that `B` is not referenced anywhere else in the program which can be checked statically since the scope of `B` is well defined (for details see [Sch96]).

However, as soon as the WITH-loop is used in a more general fashion the transformation scheme becomes more complicated since it does not correspond to a simple mapping function anymore. In the following we want to generalize the above scheme in three respects:

1. the WITH-loops to be folded may have non-identical index sets in their generator parts;
2. the second WITH-loop may contain several references to the array defined by the first one;
3. the access(es) to the array defined by the first WITH-loop may be non-local, i.e., instead of `B[j_vec]` expressions of the form `B[I_op(j_vec)]` are allowed where `I_op` projects index vectors to index vectors.

An example that covers all these aspects and therefore will be referred to throughout the whole section is the piece of a SAC program given in the upper part of Fig.2. It consists of two WITH-loops which successively compute vectors `B`

```
{...
  B = with( [0]<= i_vec <= [39])
      modarray( A, i_vec, A[i_vec] + 3);
  C = with( [20]<= j_vec <= [79])
      modarray( B, j_vec, B[j_vec] + B[j_vec - [10]]);
... }
```

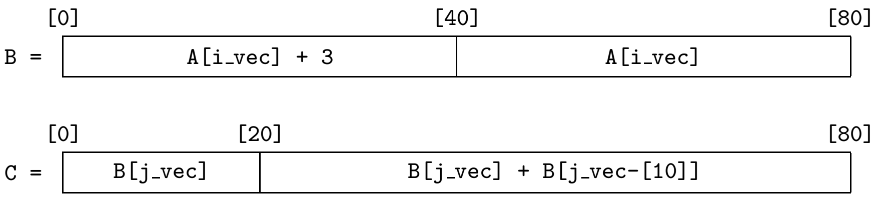

Fig. 2. Two successive WITH-loops with overlaping index ranges and multiple references.

and `C` from a given vector `A`. Each of these vectors consists of `80` integer numbers. While the first WITH-loop defines `B` to differ from `A` in that the first `40` elements are increased by `3`, the second WITH-loop defines `C` to differ from `B` in that the last `60` elements of `C` are computed as the sum of two elements of `B`, the actual one and the one that is located at the actual index position minus `10`.

A graphical representation of these WITH-loops is given in the lower part of Fig.2: Each horizontal bar represents all elements of the vector named to the left of it. The index vectors on top of the bars indicate the positions of the respective elements within the bars. The SAC expressions annotated in the bars define how the vector elements are computed from the elements of other vectors. Since different computations are required in different index vector ranges the bars are divided up by vertical lines accordingly.

Instead of first computing `B` from `A` then `C` from `B`, the array `C` can be computed from the array `A` directly. This operation requires four index ranges of `C` to be treated differently, as depicted in Fig.3.

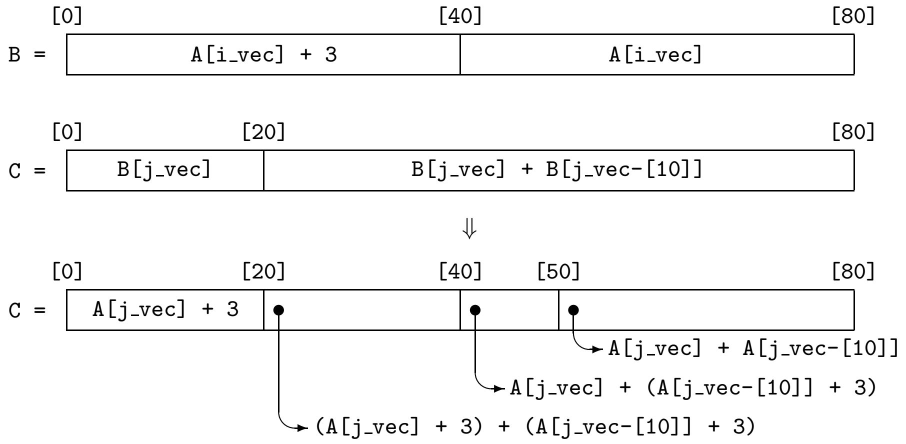

Fig. 3. Substituting two successive array modifications by a single one.

Due to the four index ranges whose array elements have to be treated differently the resulting operation cannot be expressed by a single WITH-loop. Therefore, at least at the level of compilation, a more general version of WITH-loops has to be introduced. It generalizes the possibility to specify a single operation on a subset of array indices in that it allows different operations on arbitrary partitions of array indices to be specified. Throughout this paper the following notation will be used to denote such internal WITH-loop representations:

Let S denote the set of legal indices for an array of shape $[s_1, \ldots, s_n]$, i.e., $S := \{ \; [i_1, \ldots, i_n] \mid \forall j \in \{1,...,n\} \; 0 \leq i_j < s_j \; \}$. Furthermore, let $IV_1, \ldots, IV_m$ be a partition of S and let $Op_1(\text{ i\_vec}), \ldots, Op_m(\text{ i\_vec})$ be expressions that evaluate to data structures of type τ if $\texttt{i\_vec} \in S$. Then

$$
\begin{array}{l}
A = \texttt{internal\_with(i\_vec)} \; \{ \\
\qquad IV_1 \; : Op_1(\texttt{ i\_vec}) \\
\qquad \;\vdots \qquad\quad \vdots \\
\qquad IV_m : Op_m(\texttt{ i\_vec}) \\
\quad \}
\end{array}
$$

defines an array `A` of shape $[s_1, \ldots, s_n]$ with element type τ where

$$\texttt{A[i\_vec]} := Op_j(\texttt{ i\_vec}) \Leftrightarrow \texttt{i\_vec} \in IV_j.$$ [2]

With this notation at hands, our example problem can be specified as follows:

[2] Note, that `A` is well defined since $IV_1, \ldots, IV_m$ is a partition of S

Find a transformation scheme which transforms

```
B = internal_with( i_vec) {
      [0]→[39]  : A[i_vec] + 3
      [40]→[79] : A[i_vec]
    }
C = internal_with( j_vec) {
      [0]→[19]  : B[j_vec]
      [20]→[79] : B[j_vec] + B[j_vec - [10]]
    }
```

into

```
C = internal_with( j_vec) {
      [0]→[19]  : A[j_vec] + 3
      [20]→[39] : (A[j_vec] + 3) + (A[j_vec - [10]] + 3)
      [40]→[49] : A[j_vec] + (A[j_vec - [10]] + 3)
      [50]→[79] : A[j_vec] + A[j_vec - [10]]
    }
```

where `[from`$_1$`, ..., from`$_n$`]→[to`$_1$`, ..., to`$_n$`]` denotes the set of index vectors $\{$`[i`$_1$`, ..., i`$_n$`]` $\mid \forall$`j`$\in\{$`1, ..., n`$\}$ `: from`$_j \leq$ `i`$_j \leq$ `to`$_j\}$.

The basic idea to a general solution is to formulate a scheme which stepwise replaces all references to "temporary arrays" (the array `B` in our example) by their definitions. Once all references to a temporary array are replaced, the WITH-loop by which it is defined does not contribute to the program's result anymore and thus can be eliminated.

The central rule which defines the replacement of array references by their definitions is specified in Fig.4. For an application of the rule, it is anticipated that all WITH-loops have been replaced by equivalent internal WITH-loop constructs beforehand. The upper part of Fig.4 shows the most general situation in which a replacement can be made: two array definitions are given within one scope (e.g. a function body) whose second one refers to (an) element(s) of the first one. For the sake of generality it is assumed that the first array, named `A`, is defined by m expressions `Op`$_{1,1}$`(i_vec)`, ..., `Op`$_{1,m}$`(i_vec)` on m disjoint sets of index vectors `IV`$_{1,1}$, ..., `IV`$_{1,m}$, and that the second array `B` is defined by n expressions `Op`$_{2,1}$`(j_vec)`, ..., `Op`$_{2,n}$`(j_vec)` on sets of index vectors `IV`$_{2,1}$, ..., `IV`$_{2,n}$. Furthermore, we assume that `Op`$_{2,i}$`(j_vec)` is an expression that contains a subexpression `A[ I_op( j_vec)]` as indicated by $\dashv$ ...`A[ I_op( j_vec)]`... $\vdash$. If `A[ I_op( j_vec)]` is to be replaced by parts of the definition of `A`, it has to be determined to which element of `A` the index vector `I_op(j_vec)` refers. Since these index vectors, in general, may be spread over the whole array `A`, the set of index vectors `IV`$_{2,i}$ has to be divided up into sets `IV`$_{2,i,1}$, ..., `IV`$_{2,i,m}$ with respective expressions `Op`$_{2,i,1}$`(j_vec)`, ..., `Op`$_{2,i,m}$`(j_vec)` where each expression `Op`$_{2,i,j}$`(j_vec)` is derived from `Op`$_{2,i}$`(j_vec)` by replacing `A[ I_op( j_vec)]` by `Op`$_{1,j}$`(I_op( j_vec))` as specified in Fig.4.

Applying this transformation rule to our example problem we get a sequence of program transformations as shown in Fig.5. Starting out from the two given

```
{ ...
  A = internal_with( i_vec) {
        $IV_{1,1}$  : $Op_{1,1}$( i_vec)
           ⋮           ⋮
        $IV_{1,m}$  : $Op_{1,m}$( i_vec)
      }
  ...
  B = internal_with( j_vec) {
        $IV_{2,1}$  : $Op_{2,1}$( j_vec)
           ⋮           ⋮
        $IV_{2,i}$  : $Op_{2,i}$( j_vec)=$\dashv$ ...A[ I_op( j_vec)]... $\vdash$
           ⋮           ⋮
        $IV_{2,n}$  : $Op_{2,n}$( j_vec)
      }
  ...}
```

$$\Downarrow$$

```
{ ...
  A = internal_with( i_vec) {
        $IV_{1,1}$  : $Op_{1,1}$( i_vec)
           ⋮           ⋮
        $IV_{1,m}$  : $Op_{1,m}$( i_vec)
      }
  ...
  B = internal_with( j_vec) {
        $IV_{2,1}$    : $Op_{2,1}$( j_vec)
           ⋮             ⋮
        $IV_{2,i,1}$  : $Op_{2,i,1}$( j_vec)=$\dashv$ ...$Op_{1,1}$( I_op( j_vec))... $\vdash$
           ⋮             ⋮
        $IV_{2,i,m}$  : $Op_{2,i,m}$( j_vec)=$\dashv$ ...$Op_{1,m}$( I_op( j_vec))... $\vdash$
           ⋮             ⋮
        $IV_{2,n}$    : $Op_{2,n}$( j_vec)
      }
  ...}

    with $IV_{2,i,1}$ := { j_vec | j_vec $\in$ $IV_{2,i}$ $\wedge$ I_op( j_vec) $\in$ $IV_{1,1}$}
              ⋮            ⋮
         $IV_{2,i,m}$ := { j_vec | j_vec $\in$ $IV_{2,i}$ $\wedge$ I_op( j_vec) $\in$ $IV_{1,m}$}
```

Fig. 4. Single WITH-loop-folding step.

```
B = internal_with( i_vec) {
      [0]→[39]  : A[i_vec] + 3
      [40]→[79] : A[i_vec]
    }
C = internal_with( j_vec) {                                            (a)
      [0]→[19]  : B[j_vec]
      [20]→[79] : B[j_vec] + B[j_vec - [10]]
    }

                              ⇓

C = internal_with( j_vec) {                                            (b)
      [0]→[19]  : A[j_vec] + 3
      [20]→[79] : B[j_vec] + B[j_vec - [10]]
    }

                              ⇓

C = internal_with( j_vec) {                                            (c)
      [0]→[19]  : A[j_vec] + 3
      [20]→[39] : (A[j_vec] + 3) + B[j_vec - [10]]
      [40]→[79] : A[j_vec] + B[j_vec - [10]]
    }

                              ⇓

C = internal_with( j_vec) {                                            (d)
      [0]→[19]  : A[j_vec] + 3
      [20]→[39] : (A[j_vec] + 3) + (A[j_vec - [10]] + 3)
      [40]→[79] : A[j_vec] + B[j_vec - [10]]
    }

                              ⇓

C = internal_with( j_vec) {                                            (e)
      [0]→[19]  : A[j_vec] + 3
      [20]→[39] : (A[j_vec] + 3) + (A[j_vec - [10]] + 3)
      [40]→[49] : A[j_vec] + (A[j_vec - [10]] + 3)
      [50]→[79] : A[j_vec] + A[j_vec - [10]]
    }
```

Fig. 5. Stepwise WITH-loop-folding at the example presented in Fig.3

WITH-loops in internal representation (Fig.5(a)), a stepwise transformation of the second WITH-loop construct is presented until the final version in Fig.5(e) is reached which does not contain any references to the array B anymore. Each of these steps results from a single application of the WITH-loop-folding rule from Fig.4; the references to B which are to be replaced in the next transformation step in each of the intermediate forms Fig.5(a) to Fig.5(d) are marked by boxes which surround them.

4 Simplifying Nestings of APL-like Operations

As mentioned in the previous section, WITH-loop-folding is not solely ment for the optimization of user defined WITH-loops but as a universal tool for the optimization of nested primitive array operations as well. This requires all, or at least most, of the primitive array operations to be defined in terms of WITH-loops.

At this point, one of the major design principles for WITH-loops in SAC, namely to support the specification of shape-invariant array operations, pays off. It turns out that all primitive array operations of SAC can be defined as WITH-loops within a standard library rather than being implemented as part of the compiler. As an example, the definitions for `take`, and `rotate` are given in Fig.6.

```
inline double[] take( int[] new_shp, double[] A)
{
  B = with ( 0*new_shp <= i_vec <= new_shp-1)
      genarray( new_shp, A[i_vec]);
  return(B);
}

inline double[] rotate( int dim, int num, double[] A)
{
  max_rotate = shape(A)[[dim]];
  num = num % max_rotate;
  if( num < 0)
    num = num + max_rotate;
  offset = modarray( 0*shape(A), [dim], num);
  slice_shp = modarray( shape(A), [dim], num);
  B = with ( offset <= i_vec <= shape(A)-1)
      modarray( A, i_vec, A[i_vec-offset]);
  B = with ( 0*slice_shp <= i_vec <= slice_shp-1)
      modarray( B, i_vec, A[shape(A)-slice_shp+i_vec]);
  return(B);
}
```

Fig. 6. Library definitions of `take` and `rotate`.

`take` expects two arguments, a vector `new_shape` and an array `A`, where `new_shape` indicates the number of elements to be selected from the "upper left corner" of `A`. While this operation can easily be specified as a single WITH-loop that simply copies the requested elements from `A`, the definition of `rotate` is a bit more complex.

`rotate` expects three arguments: two integer numbers `dim` and `num`, and an array `A`, whose elements are to be rotated `num` positions along the axis `dim`. In order to avoid modulo operations for the computation of each element of the resulting array `B`, two different offsets to the index vectors `i_vec` are computed: one for those elements which have to be copied from positions with lower index vectors to positions with higher index vectors (`offset`) and another one (`shape(A)-slice_shp`) for those elements that have to be copied from positions with higher index vectors to positions with lower index vectors. The usage of two different offsets on two disjoint ranges of index vectors leads to the specification of two consecutive WITH-loops.

With these definitions at hands, even rather complex nestings of primitive array operations can be transformed into sequences of WITH-loops which subsequently can be folded into single loop constructs. To illustrate this, the transformation of a simplified version of two-dimensional Jacobi relaxation [Bra96] is outlined in the remainder of this section. Given a two-dimensional array `A` with shape `[m,n]` all inner elements, i.e., all those elements with non-maximal and non-minimal index components, have to be replaced by the sums of their respective neighbor elements. Although this algorithm could be specified elegantly by a single WITH-loop, we want to examine an artificially complex specification that uses the primitive array operations of SAC only in order to illustrate the strengths of WITH-loop-folding in the context of these operations:

```
double[] relax( double[] A)
{
  m = psi( [0], shape(A));
  n = psi( [1], shape(A));

  B = rotate( 0, 1, A) + rotate( 0, -1, A)
      + rotate( 1, 1, A) + rotate( 1, -1, A);

  upper_A = take( [1,n], A);
  lower_A = drop( [m-1,0], A);
  left_A  = drop( [1,0], take( [m-1,1], A));
  right_A = take( [m-2,1], drop( [1,n-1], A));
  inner_B = take( [m-2,n-2], drop( [1,1], B));

  middle = cat( 1, left_A, cat( 1, inner_B, right_A));
  result = cat( 0, upper_A, cat( 0, middle, lower_A));
  return(result);
}                                                                     .
```

The key idea of this piece of SAC-program is to specify the summation of neighbor elements as the summation of rotated arrays. Since in the resulting array `B` the boundary elements are modified as well, these elements have to be replaced by those of the initial array `A`. This is done by first cutting off the first row of `A` (`upper_A`), the last row of `A` (`lower_A`), the first and last column of the inner rows of `A` (`left_A` and `right_A` respectively), as well as the inner elements of `B`. Subsequently, these vectors/arrays are re-combined by successive catenation operations.

Due to the complexity of this example, we want to focus on the transformation of the nesting of array operations that specifies the computation of the array `B` first. Before an inlining of the function applications of `rotate` takes place, the nesting of array additions is transformed into a sequence of assignments whose right hand sides only consist of one array operation each, i.e.,

```
{ ...
  B = rotate( 0, 1, A) + rotate( 0, -1, A)
      + rotate( 1, 1, A) + rotate( 1, -1, A);
  ...  }
```

is transformed into

```
{ ...
  tmp0 = rotate( 0, 1, A);
  tmp1 = rotate( 0, -1, A);
  tmp2 = tmp0 + tmp1;
  tmp3 = rotate( 1, 1, A);
  tmp4 = tmp2 + tmp3;
  tmp5 = rotate( 1, -1, A);
  B    = tmp4 + tmp5;
  ...  }                                        .
```

After the complete program is transformed in a similar way, standard optimizations e.g. function inlining, constant folding, constant propagation or variable propagation (for surveys see [ASU86,BGS94,PW86,Wol95,ZC91]) are applied, which introduce several WITH-loops into our example. For the inlined version of the first application of the function `rotate` we obtain:

```
{ ...
  tmp0_B = with ( [1,0] <= i_vec <= shape(A)-1)
             modarray( A, i_vec, A[i_vec-[1,0]]);
  tmp0 = with ( [0,0] <= i_vec <= [0,n-1])
           modarray( tmp0_B, i_vec, A[[m-1,0]+i_vec]);
  ...  }                                                          .
```

Transforming the WITH-loops into internal representations yields:

```
{ ...
  tmp0_B = internal_with( i_vec) {
             [0,0]→[0,n-1]   : A[i_vec]
             [1,0]→[m-1,n-1] : A[i_vec-[1,0]]
           }
  tmp0 = internal_with( i_vec) {
           [0,0]→[0,n-1]   : A[[m-1,0]+i_vec]
           [1,0]→[m-1,n-1] : tmp0_B[i_vec]
         }
  ...  }                                                  .
```

To these two WITH-loops the folding mechanism from Section 3 can be applied, which leads to the elimination of array `tmp0_B`:

```
{ ...
  tmp0 = internal_with( i_vec) {
           [0,0]→[0,n-1]   : A[[m-1,0]+i_vec]
           [1,0]→[m-1,n-1] : A[i_vec-[1,0]]
         }
  ...  }                                              .
```

Likewise, we obtain for the next application of `rotate` as well as for the application of `+` to the two intermediates `tmp0` and `tmp1`:

```
{ ...
  tmp0 = internal_with( i_vec) {
           [0,0]→[0,n-1]   : A[[m-1,0]+i_vec]
           [1,0]→[m-1,n-1] : A[i_vec-[1,0]]
         }
  tmp1 = internal_with( i_vec) {
           [0,0]→[m-2,n-1]   : A[[1,0]+i_vec]
           [m-1,0]→[m-1,n-1] : A[i_vec-[m-1,0]]
         }
  tmp2 = internal_with( i_vec) {
           [0,0]→[m-1,n-1] : tmp0[i_vec]+tmp1[i_vec]
         }
  ...  }                                                        .
```

This sequence of WITH-loops again allows for the application of WITH-loop-folding. As a consequence, the temporary arrays `tmp0` and `tmp1` are of no further use and the specification of the loop construct which defines `tmp2` is split up into three disjoint ranges of index vectors: the left column, all inner columns , and the right column.

```
{ ...
     tmp2 = internal_with( i_vec) {
              [0,0]→[0,n-1]     : A[[m-1,0]+i_vec] + A[[1,0]+i_vec]
              [1,0]→[m-2,n-1]   : A[i_vec-[1,0]] + A[[1,0]+i_vec]
              [m-1,0]→[m-1,n-1] : A[i_vec-[1,0]] + A[i_vec-[m-1,0]]
            }
  ...  }
```

Applying these optimizations further, we obtain for the generation of `B` a single loop construct with nine disjoint sets of array elements to be computed differently.

Similarly, the other temporary arrays specified explicitly in the function `relax` can be folded into single WITH-loops, e.g.

```
{ ...
  inner_B = take( [m-2,n-2], drop( [1,1], B));
  ...  }
```

is transformed into

```
{ ...
  inner_B = internal_with( i_vec) {
              [0,0]→[m-3,n-3] : B[i_vec+[1,1]]
            }
  ...  }                                                   .
```

which finally leads to a single WITH-loop that implements the entire relaxation step:

```
double[] relax( double[] A)
{
  m = psi( [0], shape(A));
  n = psi( [1], shape(A));


  result = internal_with( i_vec) {
             [0,0]→[0,n-1]     : A[i_vec]
             [1,0]→[m-2,0]     : A[i_vec]
             [1,1]→[m-2,n-2]   : A[i_vec-[1,0]] + A[[1,0]+i_vec]
                                + A[i_vec-[0,1]] + A[[0,1]+i_vec]
             [1,n-1]→[m-2,n-1] : A[i_vec]
             [m-1,0]→[m-1,n-1] : A[i_vec]
           }

  return(result);
}                                                          .
```

5 Implementing WITH-Loop-Folding

After a formal description of WITH-loop-folding in Section 3 and a case study of its applicability in the context of APL-like operations in the last section a few implementation issues have to be discussed. Since this is work in progress only two questions will be addressed here:

– Which impact does WITH-loop-folding have on the runtime performance of compiled SAC programs?
– How can WITH-loop-folding be implemented as a (statical) compiler optimization?

The general problem concerning runtimes is that we have performance gains due to the elimination of "temporary arrays" on one side, and potential performance losses due to the loss of sharing of the computations for individual elements of these arrays on the other side. Therefore, an exact prediction of the runtime impact of WITH-loop-folding would require a cost analysis for all operations involved, which in case of element computations that depend on the evaluation of conditionals is not statically decidable.

To guarantee speedups, as a first conservative approach, we restrict WITH-loop-folding to situations where it can be guaranteed that no sharing is lost at all. A simple criterion which is sufficient to exclude a loss of sharing is to demand that the "temporary arrays" may only be referenced once within the body of the second WITH-loop, and that the index vector projection used for the selection of elements of the "temporary array" (`I_op` from Fig.4) is surjective. Despite being quite restrictive these two conditions are met for the relaxation example from the last section. However, practical experiences from the application of WITH-loop-folding to real world examples will have to show whether this restriction is acceptable, or more elaborate criteria have to be developed.

The implementation of WITH-loop-folding as part of the compilation process imposes other restrictions on its applicability. The central problem in this context is that the index vector sets involved have to be known statically. Although this for the general case is not possible, in most situations it nevertheless can be done. The reason for this situation is that most generator parts of WITH-loops are specified relative to the shape of their argument arrays. Since the type inference system of SAC infers the exact shapes of all argument arrays, most of these index vector sets can be computed statically by simply applying constant folding. But infering the index vector sets of the initial WITH-loops does not suffice to do WITH-loop-folding statically. Another important prerequisite is to be able to compute intersections of such index vector sets as well as their projection by index vector mapping functions (`I_op` from Fig.4). Therefore, the projection functions have to be restricted to a set of functions which are "simple enough" to render these computations possible.

As a conservative approach we restrict these projection functions to linear projections, i.e., we allow element-wise multiplications and element-wise additions with n-ary vectors. For this restriction we can easily define a simple notation for index vector sets which is closed under intersection building and linear projection:

Let $l = [l_1, \dots, l_n]$, $u = [u_1, \dots, u_n]$, and $s = [s_1, \dots, s_n]$ denote vectors of length $n \in \mathbb{N}$. Then $l \overset{s}{\rightarrow} u$ defines the set of Index vectors

$$\{[i_1, \dots, i_n] \mid \forall_{j \in \{1,\dots,n\}} : (l_j \leq i_j \leq u_j \wedge \exists_{k_j \in \mathbb{N}_0} : i_j = l_j + k_j s_j)\} \qquad .$$

With this notation of index vector sets, intersections can be computed as follows: Let $A = l_A \overset{s_A}{\rightarrow} u_A$ and $B = l_B \overset{s_B}{\rightarrow} u_B$ denote two index vector sets of index vectors of length $n \in \mathbb{N}$. Then we have

$$A \cap B = \begin{cases} l \overset{lcd(s_A,s_B)}{\rightarrow} min(u_A, u_B) \text{ iff } \exists_{x,y \in \mathbb{N}_0^n} : & \begin{array}{l} l = l_A + x * s_A = l_B + y * s_B \\ \wedge\, (max(l_A, l_B) \leq l) \\ \wedge\, (l < max(l_A, l_B) + lcd(s_A, s_B)) \end{array} \\ \emptyset \qquad\qquad\qquad \text{otherwise} & \end{cases}$$

where $lcd(a, b)$ denotes the element-wise least common denominator of the n-ary vectors a and b, and max, and min denote the element-wise maxima and minima respectively.

Furthermore, linear projections of such index vector sets can directly be expressed by other index vector sets of this kind, i.e., with $m = [m_1, \dots, m_n]$ denoting a vector of n integers we have

$$m * (l \overset{s}{\rightarrow} u) = (m * l) \overset{(m*s)}{\rightarrow} (m * u)$$

and

$$(l \overset{s}{\rightarrow} u) + m = (l + m) \overset{s}{\rightarrow} (u + m)$$

where `*` and `+` are understood as element-wise extensions of multiplication and summation for sets and vectors.

Similar to the restrictions imposed in order to guarantee runtime improvements empiric tests on real world examples will have to show whether these restrictions are appropriate or more complex projection functions have to be included.

6 Related Work

Although WITH-loop-folding is tailor-made for the WITH-loops provided by SAC and makes use of the particular properties of that language construct, some relations to other compiler optimizations can be observed.

From the functional point of view, it can be considered a special case of *deforestation* [Wad90,Chi94,Gil96]. Both optimizations, WITH-loop-folding and the deforestation approach, aim at the elimination of intermediate data structures that are used only once. The basic idea of deforestation is to identify nestings of functions that recursively consume data structures with other functions that recursively produce these structures. Once such a nesting can be identified, the two recursions can be fused into a single one that directly computes the result of the consuming function.

Considering the basic case of WITH-loop-folding, where both WITH-loops apply a uniform operation to all array elements the first WITH-loop corresponds to a producing function whereas the second WITH-loop corresponds to a consuming function. For this case the only difference between the two optimizations is that WITH-loops operate on a single data structure whose size is statically known whereas deforestation deals with a recursive nesting of data structures of unknown depth.

Turning to the general case of WITH-loop-folding, the situation becomes more difficult since it allows the WITH-loops to apply different operations on disjoint sets of index vectors. In fact, such WITH-loops represent several different functions each of which computes a part of the elements of the resulting array, namely those that are characterized by a single set of index vectors. In order to be able to apply deforestation, a one-to-one correspondence between the producing and consuming functions is needed, i.e., the index vector sets have to be split up accordingly, which is exactly what WITH-loop-folding does. For the deforestation approach these particular situations cannot be detected as easily since that approach applies to the more general case where neither the complete data structure nor the producing and consuming functions are known explicitly as for WITH-loops in SAC.

Besides the relationship to deforestation, WITH-loop-folding can also be seen as a special combination of *loop fusion*, *loop splitting* and *forward substitution*, all of which are well-known optimization techniques in the community of high-performance computing (for surveys see [BGS94,PW86,Wol95,ZC91]). Although it is possible to draw the connection between these optimizations and WITH-loop-folding in the same manner as done for the deforestation approach, again the traditional optimization techniques suffer from their generality. In contrast, WITH-loop-folding can utilize several properties which for WITH-loops per definition hold, but for other loop constructs are often undecidable: There are no dependencies between the computations for different elements of the same array, and there is no other effect of these loops but the creation of a single array; any temporary variable used within the "loop body" cannot be referenced anywhere else.

7 Conclusions

Sac is specifically designed for the compilation of high-level array operations into efficiently executable code. One of the major design principles of Sac is to support the specification of shape-invariant array operations. As a consequence, the programmer can stepwise abstract high-level array operations from primitive array operations by the definition of Sac functions. This includes the definition of functions, which are similar to the array operations available in Apl or other languages that focus on the manipulation of arrays (e.g. Nesl[Ble94], or Nial[JJ93]). The basic language construct for the definition of such functions is the with-loop, a special loop construct for element-wise specifications of array manipulations.

This paper introduces a new compiler optimization, called with-loop-folding, for the transformation of two consecutive with-loops into a single one. As an extended example, a simplified version of Jacobi relaxation, specified by means of high-level array operations similar to those available in Apl, is examined. It is shown that with-loop-folding in combination with standard optimizations, e.g. function inlining, constant folding, etc., allows the relaxation algorithm to be folded into a single loop construct that directly implements one relaxation step. A runtime comparison shows that this loop compiles to code which executes about 5 times faster than code which is compiled directly from the initial nesting of Apl-constructs.

From a theoretical point of view, any nesting of primitive array operations can be folded into a single loop which defines the nested array operation as an element-wise direct computation of the resulting array from the argument array(s). As a consequence, the code compiled from an array operation becomes invariant against the number and kind of temporary arrays specified, i.e., the programmer is liberated from low level concerns.

However, with respect to the implementation of with-loop-folding, a couple of questions have not been addressed yet: Since the Sac compiler heavily specializes the source programs in order to be able to infer the exact shapes of all arrays statically, for real world examples, it may be necessary to apply with-loop-folding very often. Due to the complexity of the optimization this may lead to large compiler runtimes. Therefore, a sophisticated scheme for the execution order of with-loop-folding and standard optimizations may be required.

Another problem not yet addressed is the influence of caches on the relationship between with-loop-folding and the runtime behavior of compiled Sac programs. A trivial example for such a situation is a multiple access to the elements of an array in column major order. Since the arrays are stored row major order, such accesses lead to bad cache behavior. This can be improved significantly by the explicit creation of a transposed array. To avoid slowdowns due to such problems, rules have to be established that determine which folding operations are favorable and which are unfavorable. For a more general solution, the development of other transformation schemes may be needed which analyze the array access schemes and after the application of WITH-loop-folding (re-)introduce temporary arrays whenever appropriate.

References

[AD79] W.B. Ackerman and J.B. Dennis: *VAL-A Value-Oriented Algorithmic Language: Preliminary Reference Manual.* TR 218, MIT, Cambridge, MA, 1979.

[AGP78] Arvind, K.P. Gostelow, and W. Plouffe: *The ID-Report: An asynchronous Programming Language and Computing Machine.* Technical Report 114, University of California at Irvine, 1978.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman: *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986. ISBN 0-201-10194-7.

[BCOF91] A.P.W. Böhm, D.C. Cann, R.R. Oldehoeft, and J.T. Feo: *SISAL Reference Manual Language Version 2.0.* CS 91-118, Colorado State University, Fort Collins, Colorado, 1991.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp: *Compiler Transformations for High-Performance Computing.* ACM Computing Surveys, Vol. 26(4), 1994, pp. 345–420.

[Ble94] G.E. Blelloch: *NESL: A Nested Data-Parallel Language (Version 3.0).* Carnegie Mellon University, 1994.

[Bra96] D. Braess: *Finite Elemente.* Springer, 1996. ISBN 3-540-61905-4.

[Can92] D.C. Cann: *Retire Fortran? A Debate Rekindled.* Communications of the ACM, Vol. 35(8), 1992, pp. 81–89.

[Chi94] W.-N. Chin: *Safe Fusion of Functional Expressions II: Further Improvements.* Journal of Functional Programming, Vol. 4(4), 1994, pp. 515–550.

[Gil96] A. Gill: *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Glasgow University, 1996.

[GS95] C. Grelck and S.B. Scholz: *Classes and Objects as Basis for I/O in SAC.* In T. Johnsson (Ed.): Proceedings of the Workshop on the Implementation of Functional Languages'95. Chalmers University, 1995, pp. 30–44.

[HAB$^+$95] K. Hammond, L. Augustsson, B. Boutel, et al.: *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language.* University of Glasgow, 1995. Version 1.3.

[HSB97] J. Hammes, S. Sur, and W. Böhm: *On the effectiveness of functional language features: NAS benchmark FT.* Journal of Functional Programming, Vol. 7(1), 1997, pp. 103–123.

[Ive62] K.E. Iverson: *A Programming Language.* Wiley, New York, 1962.

[JJ93] M.A. Jenkins and W.H. Jenkins: *The Q'Nial Language and Reference Manuals.* Nial Systems Ltd., Ottawa, Canada, 1993.

[KR90] B.W. Kernighan and D.M. Ritchie: *Programmieren in C.* PC professionell. Hanser, 1990. ISBN 3-446-15497-3.

[LP94] J. Launchbury and S. Peyton Jones: *Lazy Functional State Threads.* In Programming Languages Design and Implementation. ACM Press, 1994.

[MSA$^+$85] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al.: Sisal*: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* M 146, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1985.

[PvE95] M.J. Plasmeijer and M. van Eckelen: *Concurrent Clean 1.0 Language Report.* University of Nijmegen, 1995.

[PW86] D.A. Padua and M.J. Wolfe: *Advanced Compiler Optimizations for Supercomputers.* Comm. ACM, Vol. 29(12), 1986, pp. 1184–1201.

[SBvEP93] S. Smetsers, E. Barendsen, M. van Eeklen, and R. Plasmeijer: *Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs.* Technical report, University of Nijmegen, 1993.

[Sch96] S.-B. Scholz: **S***ingle* **A***ssignment* **C** *– Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen.* PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.

[Sch97] S.-B. Scholz: *On Programming Scientific Applications in* Sac *- A Functional Language Extended by a Subsystem for High-Level Array Operations.* In Werner Kluge (Ed.): Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers, LNCS, Vol. 1268. Springer, 1997, pp. 85–104.

[Ser97] P.R. Serrarens: *Implementing the Conjugate Gradient Algorithm in a Functional Language.* In Werner Kluge (Ed.): Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers, LNCS, Vol. 1268. Springer, 1997, pp. 125–140.

[Wad90] P.L. Wadler: *Deforestation: transforming programs to eliminate trees.* Theoretical Computer Science, Vol. 73(2), 1990, pp. 231–248.

[Weh85] H. Wehnes: *FORTRAN-77: Strukturierte Programmierung mit FORTRAN-77.* Carl Hanser Verlag, 1985.

[Wol95] M.J. Wolfe: *High-Performance Compilers for Parallel Computing.* Addison-Wesley, 1995. ISBN 0-8053-2730-4.

[ZC91] H. Zima and B. Chapman: *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.