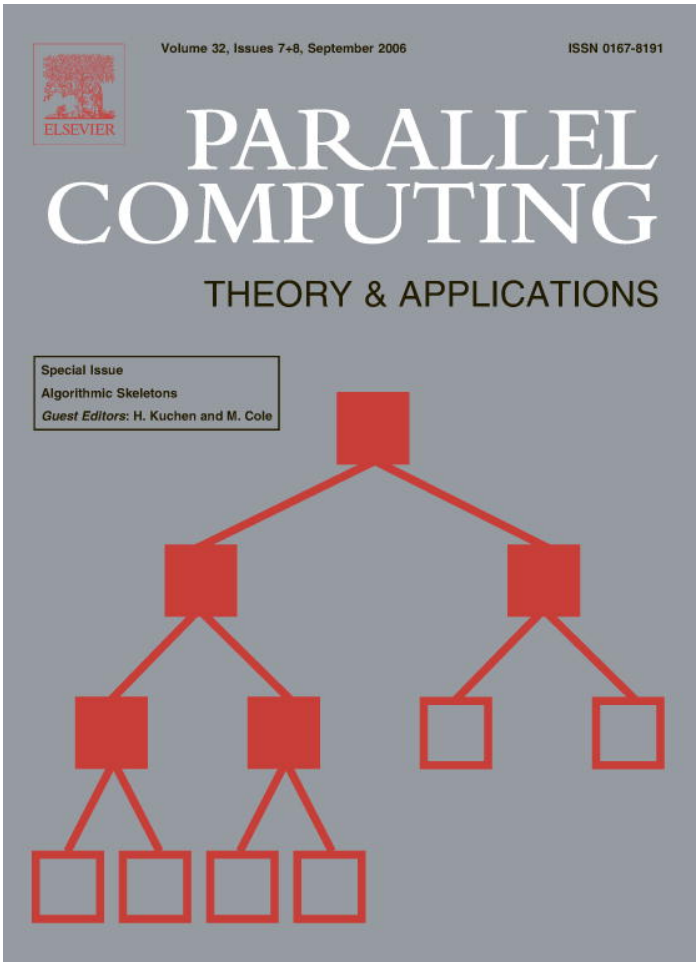

Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 32 (2006) 507–522
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Merging compositions of array skeletons in SaC

Clemens Grelck [a,*], Sven-Bodo Scholz [b]

[a] University of Lübeck, Institute of Software Technology and Programming Languages, Ratzeburger Allee 160, 23538 Lübeck, Germany
[b] University of Hertfordshire, Department of Computer Science, College Lane, Hatfield, Hertfordshire AL10 9AB, United Kingdom

Received 19 January 2006; received in revised form 11 August 2006; accepted 14 August 2006

Abstract

The design of skeletons for expressing concurrent computations usually faces a conflict between software engineering demands and performance issues. Whereas the former favour versatile fine-grain skeletons that can be successively combined into larger programs, coarse-grain skeletons are more desirable from a performance perspective.

We describe a way out of this dilemma for array skeletons. In the functional array language SaC we internally represent individual array skeletons by one or more meta skeletons, called with-loops. The design of with-loops is carefully chosen to be versatile enough to cope with a large variety of skeletons, yet to be simple enough to allow for compilation into efficiently executable (parallel) code. Furthermore, with-loops are closed with respect to three tailor-made optimisation techniques, that systematically transform compositions of simple, computationally light-weight skeletons into few complex and computationally heavier-weight skeletons.

Keywords: Single Assignment C; Algorithmic skeletons; Array programming; Program optimisation

1. Introduction

Array operators were first introduced as a mathematical notation in the context of Apl [1,2]. Simple array operators, for example, are extensions of scalar operators and functions to multi-dimensional arrays in an element-wise manner. Other array operators leave element values untouched, but manipulate the layout of arrays. Examples that preserve the shapes of argument arrays are rotate and shift operators. In contrast, take and drop operators yield subarrays whose shapes depend on additional arguments. Finally, there are reduction operators that allow us, for example, to compute sums and products or minimum and maximum values of argument arrays. More sophisticated operations and entire application programs are constructed mostly by composition of these array operators.

Array operators have rather straightforward data parallel implementations [3]. Exchanging sequential implementations by parallel implementations turns an array program into a parallel program. All aspects

* Corresponding author.
E-mail addresses: grelck@isp.uni-luebeck.de (C. Grelck), s.scholz@herts.ac.uk (S.-B. Scholz).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2006.08.003

of parallel program execution are confined in the array operators, which represent common patterns of data parallel execution. This analogy to algorithmic skeletons [4–8] motivates us to call array operators *array skeletons* if we want to put the emphasis on their parallel aspects.

Like the design of skeletal programming environments in general, the design of array skeletons faces a dilemma between good software engineering principles and the desire to achieve high runtime performance. Principles of good software engineering demand for a small number of orthogonal, general-purpose skeletons, which can easily be reused for different applications, and suggest to construct specific applications by composition. In contrast, high performance requires skeletons that are coarse-grain and application-specific, leaving little opportunities for code reuse.

One approach to have both, good design and good performance, could be an algebra of array skeletons, i.e., a set of reduction rules that replace some pattern of skeleton composition by a semantically equivalent though less expensive one. For example, we may replace the rotation of a vector by two elements to the left followed by a rotation by four elements to the right by a single rotation skeleton rotating the original vector by two elements to the right. Likewise, we may reduce taking the first ten elements of a vector followed by taking the first five elements of the intermediate vector to taking the first five elements of the original vector right away.

Unfortunately, such simple compositions rarely occur in practice. Worse, even simple compositions often require new or functionally extended skeletons for their proper representation. Consider as an example we would rotate a matrix by one element to the left and then by two elements downwards. In order to represent the combined operation by a single skeleton we need at least a rotation skeleton that is capable of simultaneously rotating arrays along multiple axes. Likewise, we may want to take the first ten elements of a vector and then drop the first five elements from the intermediate vector. The combined operation can only be expressed by a new skeleton that selects a subarray of some shape at some offset position.

In this paper we propose a different approach to achieve high runtime performance with a well designed collection of simple, general-purpose array skeletons and the principle of composition. We have developed this approach in the context of SaC (Single Assignment C) [9,10], a functional, implicitly parallel array skeleton language. On the programming language level SaC supports a similar collection of array skeletons as Apl or as sketched out in the beginning. However, implementation-wise SaC follows new roads.

If we analyse the algebraic approach, as sketched out above, we can identify as the main limitation the need to find suitable representations for increasingly application-specific compositions of array skeletons within the set of existing skeletons. This either leads to an ineffectively small number of reduction rules or to an inflationary increase in the number of skeletons and their degree of specialisation. In essence, the problem lies in the fact that the usual array skeletons are not closed with respect to composition.

Our approach to circumvent this dilemma lies in the introduction of a meta-level skeleton, named with-loop. Our with-loops are carefully chosen abstractions of array skeletons. On the one hand, they are sufficiently powerful and versatile to express a large variety of array skeletons without loosing genericity. On the other hand, they are restricted enough to allow us to generate efficiently executable (parallel) code from individual with-loops. Rather than hard-wiring a limited set of array skeletons both into our language definition and our compiler, we use these with-loops to implement standard array skeletons as user-defined functions and store them in a library. By inlining their definitions we establish an intermediate code representation that is exclusively made up of with-loops. They are carefully designed to be closed representations with respect to a collection of program transformations that systematically merge compositions of with-loops into single with-loops. Hence, with-loops form the basis of large-scale code transformations implemented as compiler optimisations in the SaC compiler.

We have identified three different types of composition: vertical, horizontal and nested composition. Each of them is addressed by a tailor-made optimisation technique. Vertical composition describes computational pipelines where the result of one with-loop becomes the argument of another with-loop. With-loop-folding [11] aims at shifting computational pipelines from the level of entire arrays to the level of individual elements. Horizontal composition is characterised by multiple with-loops that compute separate results from the same or at least an overlapping set of arguments. In the absence of data dependences with-loop-fusion [12] aims at

merging them into a single WITH-loop that computes all results simultaneously. Nested composition occurs whenever element-level operations within a WITH-loop are in fact WITH-loops themselves rather than scalar code. WITH-loop-scalar-isation [13] aims at merging nested WITH-loops into single ones operating on the scalar level.

In a complex optimisation process we step-by-step transform deeply nested compositions of rather simple, general-purpose and computationally light-weight WITH-loops into few complex, application-specific and computationally heavier-weight WITH-loops. With respect to parallel execution these program transformations have the effect of eliminating superfluous synchronisation and communication events and, generally, improve the ratio between productive computations and organisational overhead.

We have implemented all three code transformation techniques, WITH-loop-folding, WITH-loop-fusion and WITH-loop-scalar-isation, in our SAC compiler and thoroughly investigated their impact on runtime performance. Until now, we have addressed the individual optimisation techniques mostly in isolation [11,14,12,13]. The contribution of this paper is to show how a combination of these techniques serves as a general mechanism for merging array skeletons in SAC.

The remainder of this paper is organised as follows. Section 2 elaborates on the design of WITH-loops. In Section 3, we introduce our running example. Sections 4–6 introduce the three optimisation techniques and discuss their effect on the running example. Related work is presented in Sections 7 and 8 concludes.

2. With-loops in SAC

As the name suggests, SAC can be considered a functional variant of C. The basic idea is to restrict C in a way that guarantees a side-effect free setting. Essentially, this is achieved by eliminating global variables and pointers from C and by having a clear separation between expressions and assignments. All major language constructs from C such as conditionals, loops, assignments, function definitions, or function calls can be adopted without any change in their operational behaviour (for details see [9]).

On top of this language core SAC supports arrays as the only data structure. For manipulating these, several array skeletons are available. They resemble the functionality of the built-in operators in high-level array languages such as APL [1,2], J [15], or NIAL [16]. However, none of these skeletons is hard-wired into the compiler. Instead, all these skeletons are defined in terms of a meta-skeleton called WITH-loop. Its design is aimed at several goals:

(1) WITH-loops need to be expressive enough to define all array skeletons.
(2) It should also be possible to define rather complex problem-specific array operations in terms of one, or at least very few, WITH-loops.
(3) Their design must be such that nestings of these constructs can be systematically transformed into fewer, more complex ones.
(4) WITH-loops need to be suitable for compilation into concurrently executable code.

The most simple form of WITH-loops constitutes a mapping of an expression for computing an individual array element to all element positions of an array. It takes the general form

```
with
   (idx_vec): expr
genarray (shape)
```

where *idx_vec* is an identifier, *shape* denotes an expression that should evaluate to an integer vector and *expr* denotes an arbitrary expression. Such a WITH-loop defines exactly one array. Its dimensionality (also referred-to as *rank*) is defined by the length of the vector computed from *shape* and the extents of the individual axes are given by that vector's components. The elements of the result array are computed from the expression *expr*.

As an example, consider the following WITH-loop:

```
with
  (iv): 42
genarray([3, 5])
```

Here, for each legal index vector position `iv` of the resulting array of shape `[3,5]` we define the corresponding element to be `42`. Hence, we obtain as value for the above WITH-loop the matrix

$$\begin{pmatrix} 42 & 42 & 42 & 42 & 42 \\ 42 & 42 & 42 & 42 & 42 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}.$$

As these simple array creations occur rather often, the standard library of SAC defines an operation `mkarray` to that effect:

```
int[*] mkarray(int[.] shp, int value)
{
  res = with
        (iv): value
      genarray(shp);
  return(res);
}
```

Here, `int[.]` is a SAC type of integer vectors while the type `int[*]` denotes integer arrays of any shape, including any rank. Using this array skeleton, the above example can be specified as `mkarray( [3,5], 42)`.

In our first example the index variable `iv` is not referred to in the element definition. Therefore, the values of the resulting array do not depend on their position within the array and, hence, are all the same.

A simple example where the individual element values of the resulting array do depend on their index position is the APL operator `iota`, which, given a number n, yields the vector $[0,\ldots,n-1]$:

```
int[.] iota(int n)
{
  res = with
        (iv) : iv[0]
      genarray([n])
  return(res);
}
```

Here, the element values of the resulting vector are solely defined by their index position. Note here that the selection of the first (and only) element of `iv` is necessary as the index variable always constitutes a vector of indices rather than a scalar index.

With this facility, we can define extensions of scalar operations such as addition to multi-dimensional arrays of any shape as[1]:

```
int[*] (+)(int[*] a, int[*] b)
{
  res = with
        (iv) : a[iv] + b[iv]
      genarray(shape(a));
  return(res);
}
```

Since the index variable is a first class identifier in SAC, it constitutes an ordinary expression and can be used in arbitrary contexts. As an example, consider the definition of the array skeleton `drop`:

[1] For the sake of brevity we silently assume the arguments of + to have the same shape.

```
int[*] drop(int[.] offset, int[*] a)
{
  res = with
        (iv) : a[iv+offset]
       genarray(shape(a) - offset)
  return(res);
}
```

It expects two arguments: an offset vector `offset` specifies the number of elements to be dropped from the individual axes of the array `a`. The shape of the result array can be computed by subtracting the `offset` from the shape of `a` in an element-wise fashion. The individual elements are picked from `a` using the expression `iv+offset` as index into `a` for skipping the elements to be dropped. Thus, we obtain: `drop ([2], [0,1,2,3,4]) == [2,3,4]`.

In analogy to `drop` we can define the concatenation of two vectors as follows:

```
int[.] (++)(int[.] a, int[.] b)
{
  res =with
        (iv) : iv < shape(a) ? a[iv] : b[iv-shape(a)]
       genarray(shape(a) + shape(b))
  return(res);
}
```

With this example we reach the limits of our simple WITH-loop representation as outlined so far. The conditional expression which depends on values of the index variable `iv` has basically two disadvantages: Firstly, it is clumsy to read, in particular if we consider more complex array operations that distinguish several different areas within the array to be generated. Secondly, it requires rather complex compiler technology in order to avoid repetitive evaluations of the conditional in the WITH-loop-body at runtime. To avoid these difficulties of the restricted form of WITH-loops as presented so far, we introduce the concept of *multi-generator* WITH-loops. The basic idea is to replace the pair of an index variable and an expression by several pairs of these and to associate each of the pairs with a range of indices in a way that guarantees all ranges to constitute a partition of the entire index range. Syntactically, the range specification is annotated at the index variable by replacing (*idx_vec*) with (*lb*<=*idx_vec*<*ub*). Here, *lb* and *ub* denote expression which must evaluate to integer vectors of the same length as the *shape* expression. They define the element-wise minimum and maximum of the index range covered, respectively. In the sequel, we refer to these syntactical forms as *generators*.

With this extension at hand, we can rewrite the concatenation of two vectors as follows:

```
int[.] (++) (int[.] a, int[.] b)
{
  res_shp = shape(a) + shape(b);
  res = with
        ([0] <= iv < shape(a)) : a[iv]
        (shape(a) <= iv < res_shp) : b[iv-shape(a)]
       genarray(res_shp)
  return(res);
}
```

Which of the generator-associated expressions is evaluated to define a certain element of the resulting array now depends on the generator where its index position is located in.[2]

[2] For the purpose of this paper we assume that any user-defined multi-generator WITH-loop in fact constitutes a partition of the complete index range. In full SAC we have a slightly less restrictive semantics, which is based on first match. Whenever we cannot statically decide whether the ranges are pairwise disjoint, we transform these multi-generator WITH-loops into sequences of ordinary WITH-loops.

The flexibility of multi-generator WITH-loops allows us not only to define individual array operators such as `drop` or `++` as WITH-loops but it also enables us to specify more complex array operations as individual WITH-loops. This design constitutes the key prerequisite for a systematic, compiler driven transformation of nested array operations into individual WITH-loops. As an example, consider a nested application of some of the array operations defined above:

```
mkarray( [9,4], 0.0) ++++ drop( [0,4], B).
```

Assuming the array `B` has shape `[9,9]`, the expression `drop([0,4], B)` yields an array where the first 4 columns of `B` have been dropped, i.e., of shape `[9,5]`. Concatenation along the second axis (denoted by `++++`) with an array of zeros of shape `[9,4]` yields the desired result of shape `[9,9]` whose elements are 0 within the first 4 columns and copied from `B` within the last 5 columns.

The overall functionality of this expression can be specified by a single multi-generator WITH-loop as:

```
with
  ([0,0] <= iv < [9,4]) : [0,0]
  ([0,4] <= iv < [9,9]) : B[iv]
genarray([9,9]).
```

Comparing the two specifications we can observe that the former encourages code reuse, namely that of the basic operators, whereas the latter is more amenable for a compilation into efficiently executable parallel code. This is due to the fact that all generators define *sets* of index vectors, which implies that generator-associated expressions may be evaluated in any order without affecting the aggregate semantics.

In the remainder of the paper we describe three optimisation techniques that enable our compiler to combine the advantages of both representations: they transform the former into the latter. This is achieved by first inlining the definitions of the individual array operations and then systematically transforming the resulting nesting of WITH-loops into a single one.

3. Running example

We illustrate the combined effect of our three WITH-loop optimisations, folding, fusion and scalarisation, by a running example. In order to demonstrate complexity and versatility of the individual optimisations without making the example overly complicated, we use a rather artificial function `foo` as shown in Fig. 1.

Our function `foo` takes a 9×9-element matrix of complex numbers as an argument and yields two such matrices as results. The body of `foo` contains definitions of an intermediate array `B` as well as the result arrays `C` and `D`. The intermediate array `B` is defined to take the first five rows of the argument array `A` concatenated along the first axis with a 4×9-element array of complex ones (`[1,0]`). The infix operator `+++` extends the concatenation operator `++` on vectors, introduced in the previous section, to multi-dimensional arrays. The first result array `C` is defined as the element-wise sum of the argument array `A` and a version of the intermediate array `B` rotated by one row and by two columns towards increasing indices. At last, we define the second result array `D` as the column-wise concatenation (denoted by `++++`) of a 9×4-element array of complex zeros with the right-most five columns of array `B`.

Fig. 2 provides a graphical illustration of the running example. Frames represent index spaces with the origin being in the upper left corner. We symbolise the different expressions defining elements of arrays `B`, `C` and `D`

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  B = take( [5,9], A) +++ mkarray( [4,9], [1,0]);
  C = A + rotate([1,2], B);
  D = mkarray( [9,4], [0,0]) ++++ drop( [0,4], B);
  return( C, D);
}
```

Fig. 1. Running example using array skeletons.

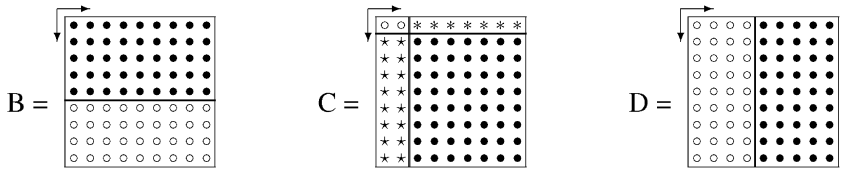

Fig. 2. Illustration of running example.

by circles, bullets, stars and asterisks. For example, in the illustration of array B bullets (•) represent selections into the argument array A (A[iv]) while circles (○) denote complex ones ([1,0]). The index spaces of both arrays B and D consist of two rectangular subranges divided row-wise and column-wise, respectively. The rotate operator yields an array C whose index space consists of four quadrants, as shown in Fig. 2. In each quadrant we must use different offsets for references into array B.

As pointed out in the previous section, in SaC we internally represent all array skeletons by means of with-loops. Hence, the internal representation of the code of Fig. 1 consists of eight with-loops. Since we have already demonstrated the definition of individual skeletons by with-loops in Section 2 and our space is limited, we refrain from showing this version of the running example. Instead, Fig. 3 contains an already partially optimised version, where each of the arrays B, C and D is defined by a single multi-generator with-loop. This intermediate version of the running example exactly coincides with the graphical representation in Fig. 2. For example, the elements of the generator shown in line 4 in Fig. 3 are represented by bullets (•) on the left hand side of Fig. 2. Likewise, we use stars (★) to represent the elements of the generator in line 10 in the illustration of array C in Fig. 2. We will use the code shown in Fig. 3 as a basis to illustrate our optimisation techniques in the following three sections.

4. With-loop folding

Our first optimisation technique, with-loop-folding, addresses vertical composition of with-loops, i.e., the result of one with-loop becomes the argument of another with-loop. In Fig. 3, we have two examples of vertical composition: array B defined by the first with-loop is referred to in both the second and the third with-loop. with-loop-folding aims at condensing two subsequent with-loops into a single one by systematically forward substituting computations associated with the first or *source* with-loop into the second or *target* with-loop. It is inspired by the well known map equivalence

$$(\texttt{map}\,\texttt{f}) \circ (\texttt{map}\,\texttt{g}) \Longleftrightarrow \texttt{map}\,(\texttt{f} \circ \texttt{g}).$$

```
1  complex[9,9], complex[9,9] foo (complex[9,9] A)
2  {
3    B = with
4          ([0,0] <= iv < [5,9]) : A[iv]
5          ([5,0] <= iv < [9,9]) : [1,0]
6        genarray( [9,9]);
7    C = with
8          ([0,0] <= iv < [1,2]) : A[iv] + B[iv-[-8,-7]]
9          ([0,2] <= iv < [1,9]) : A[iv] + B[iv-[-8, 2]]
10         ([1,0] <= iv < [9,2]) : A[iv] + B[iv-[ 1,-7]]
11         ([1,2] <= iv < [9,9]) : A[iv] + B[iv-[ 1, 2]]
12       genarray( [9,9]);
13   D = with
14         ([0,0] <= iv < [9,4]) : [0,0]
15         ([0,4] <= iv < [9,9]) : B[iv]
16       genarray( [9,9]);
17   return( C, D);
18 }
```

Fig. 3. Running example after conversion into with-loops.

What distinguishes WITH-loop-folding from the simple map equivalence is the fact that WITH-loops generally do not represent uniform operations. Unlike the functions `f` and `g` in the map equivalence, WITH-loops have access to the index position through its representation by the index identifier. Multi-generator WITH-loops have multiple defining expressions for pairwise disjoint index subsets.

Having a closer look at our running example in Fig. 3 shows some of the challenges. Let us begin with folding the first WITH-loop into the third WITH-loop. The latter has a single reference to array `B` (`B[iv]`). We cannot simply replace the expression `B[iv]` by some expression from the first WITH-loop because the first WITH-loop has two generators and each is associated with a different expression. Hence, we must first determine which of the two expressions to take. For this purpose we must relate the current generator (`[0,4] <= iv < [9,9]`) to each generator of the first WITH-loop. More precisely, we must compute pairwise intersections. The intersections between our current generator and both generators of the first WITH-loop are non-empty. Therefore, we split our current generator and replace it by the two intersection generators (`[0,4] <= iv < [4,9]`) and (`[4,4] <= iv < [9,9]`). Both generators are associated with the expression `B[iv]`. However, in the next step we can safely replace these references to array `B` by the corresponding expressions from the first WITH-loop. Fig. 4 shows the result of this folding step; Fig. 5 provides a graphical illustration in analogy to Fig. 2.

Folding the first WITH-loop of Fig. 3 into the second WITH-loop is further complicated by index computations. In order to identify the correct defining expression from the source WITH-loop, we must first translate the current generator with respect to the index computation. More precisely, we must identify the index computation as an affine function of the index vector and apply this affine function to the boundary vectors of the current generator. Taking the first generator of the second WITH-loop in as an example, the original generator (`[0,0] <= iv < [1,7]`) is translated into (`[8,2] <= iv < [9,9]`). After that we systematically compute the intersections between the translated generator and each generator of the referenced WITH-loop. The first intersection turns out to be empty and is ignored. Hence, all references to array `B` associated with the first generator of the target WITH-loop actually fall into the index set defined by the second generator of the source WITH-loop. Hence, we may safely replace `B[iv+[8,2]]` by `[1,0]` and continue with the remaining generators of the target WITH-loop.

In general, an expression in a source WITH-loop contains references to the index vector of the source WITH-loop (`iv` in our running example). In this case, all such references must be replaced by the translated reference to the target WITH-loop's index vector when the defining expression is moved from the scope of the source WITH-loop into the scope of the target WITH-loop. This can be observed for the third and the fourth generator of the resulting WITH-loop in Fig. 4.

Having a closer look at Fig. 5 shows a typical phenomenon: The first and the second generator of the WITH-loop defining array `C` could be coalesced since they are associated with identical expressions and the union of their index sets can be represented by a single generator. The same holds for the fifth and the sixth generator.

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  C = with
        ([0,0] <= iv < [1,2]) : A[iv] + [1,0]
        ([0,2] <= iv < [1,9]) : A[iv] + [1,0]
        ([1,0] <= iv < [6,2]) : A[iv] + A[iv-[ 1,-7]]
        ([1,2] <= iv < [6,9]) : A[iv] + A[iv-[ 1, 2]]
        ([6,0] <= iv < [9,2]) : A[iv] + [1,0]
        ([6,2] <= iv < [9,9]) : A[iv] + [1,0]
      genarray( [9,9]);
  D = with
        ([0,0] <= iv < [9,4]) : [0,0]
        ([0,4] <= iv < [5,9]) : A[iv]
        ([5,4] <= iv < [9,9]) : [1,0]
      genarray( [9,9]);
  return( C, D);
}
```

Fig. 4. Running example after WITH-loop-folding.

Fig. 5. Illustration of running example after WITH-loop-folding.

Accompanying WITH-loop-folding with a post-processing generator coalascing transformation has the benefit of fewer generators to be handled in subsequent applications of WITH-loop-folding and the other WITH-loop optimisations.

Proper WITH-loop-folding, as illustrated in the context of our running example, requires significant static knowledge about the array shapes and generators involved. Hence, WITH-loop-folding can only be as effective as static knowledge is available or can be inferred by means of other optimisations. For this purpose, we have incorporated a comprehensive collection of standard optimisation techniques into the SAC compiler, e.g., function inlining, function specialisation, loop unrolling, constant folding, or constant propagation to name just a few [9].

Comparing the original running example in Fig. 1 with the optimised representation in Fig. 4 we witness a reduction in the number of array skeletons or WITH-loops from 8 to 2. This comes with a substantial reduction in the number of intermediate arrays which in turn reduces the overhead for memory management and improves the locality of data. Taking into account that individual WITH-loops or array skeletons are the source of concurrency we aim to exploit for parallel execution, we achieve substantial reductions of communication and synchronisation overhead at the same time. Furthermore, we observe a systematic transformation from compositions of array skeletons with a well defined, application-unspecific meaning like `take` or `rotate` into fewer array skeletons that are highly application-specific.

Despite its beneficial role in our running example, WITH-loop-folding must be applied with care. As a matter of fact, WITH-loop-folding may duplicate computations, e.g., by folding computations of a source WITH-loop into several target WITH-loops or by replacing multiple references in a single target WITH-loop by the same expression from the source WITH-loop. Considering the continuously increasing discrepancy between processor and memory speeds, we may actually duplicate non-negligible amounts of computational work before an additional memory access pays off. The exact break-even point is both highly machine-dependent and difficult to predict. In the absence of a sufficiently accurate cost model, we take care not to increase the number of memory references or to duplicate applications of recursive functions. This rather simple control scheme has proved to be sufficient in practice.

A more thorough treatment of WITH-loop-folding as a compiler optimisation technique and its implementation in the SAC compiler can be found in [11]. We investigate the impact of WITH-loop-folding on the runtime performance of compiled SAC code and, as a consequence, on the style of programming in SAC in [14].

5. With-loop fusion

WITH-loop-fusion addresses horizontal compositions of WITH-loops. Horizontal composition is characterised by two a more WITH-loops without data dependences that iterate over the same index space or at least over similar index spaces. In the current representation of our running example, shown in Fig. 4, the two remaining WITH-loops form such a horizontal composition. The idea of WITH-loop-fusion is to fuse such WITH-loops into a single one that simultaneously defines both arrays. Simultaneous definition of multiple values requires an extension of our WITH-loops that goes beyond their introduction in Section 2. We call this extension *multi-operator* WITH-loop. Fig. 6 shows the effect of WITH-loop-fusion on the running example; a graphical illustration of the index space can be found in Fig. 7.

The multi-operator WITH-loop in Fig. 6 defines two arrays of 9×9 elements, as can be deduced from the repeated occurrence of the key word `genarray`. Two identifiers (`C` and `D`) are bound to these arrays by a simultaneous assignment. As we define two different arrays, each generator is actually associated with two expressions, separated by a comma. While the value of the first expression serves the initialisation of the first

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
 C,D = with
        ([0,0] <= iv < [1,4]) : A[iv] + [1,0],          [0,0]
        ([0,4] <= iv < [1,9]) : A[iv] + [1,0],          [1,0]
        ([1,0] <= iv < [6,2]) : A[iv] + A[iv-[1,-7]], [0,0]
        ([1,2] <= iv < [6,4]) : A[iv] + A[iv-[1,2]],   [0,0]
        ([1,4] <= iv < [5,9]) : A[iv] + A[iv-[1,2]],   [1,0]
        ([5,4] <= iv < [6,9]) : A[iv] + A[iv-[1,2]],   A[iv]
        ([6,0] <= iv < [9,4]) : A[iv] + [1,0],          [0,0]
        ([6,4] <= iv < [9,9]) : A[iv] + [1,0],          A[iv]
       genarray( [9,9])
       genarray( [9,9]);

 return( C, D);
}
```

Fig. 6. Running example after WITH-loop-fusion.

result array (C), the value of the second expression defines the element values of the second result array (D). Although not shown in the running example, multi-operator WITH-loops in general may freely combine genarray and fold operations, and the number of simultaneously defined values may well exceed two.

We have intentionally not introduced multi-operator WITH-loops in Section 2 as they are not necessary to define well designed array skeletons. Simultaneous definition of multiple different values runs counter the principles of good software engineering, that aim at identifying simple general-purpose abstractions. Any array skeleton that yields two values could easily be replaced by two simpler skeletons yielding one of the values each. However, as the result of a program transformation, the ability to simultaneously define multiple values is an important prerequisite for achieving high runtime performance.

Fusing the two WITH-loops of Fig. 4 to the single multi-operator WITH-loop of Fig. 6 requires us to make the index spaces defined by the various generators uniform. We achieve this by systematically computing the intersections between each generator of the first WITH-loop and each generator of the second WITH-loop. In the running example as well as in practice most of these intersections are in fact empty. Hence, rather than facing a quadratic explosion in the number of generators we usually observe only a modest increase.

WITH-loop-fusion reduces the loop overhead in the executable code, but more important is the effect of sharing information otherwise computed individually in both WITH-loops. For example, the expressions associated with generators 6 and 8 in Fig. 6 contain two references to argument array A (A[iv]). In the original non-fused representation of the code in Fig. 4 the iteration distance between two accesses to the same element of array A is proportional to the size of the array. Hence, for relevant problem sizes we may not expect the second reference to be a cache hit, and we effectively end up with two main memory accesses. WITH-loop-fusion manipulates the sequence of array references such that references to the same element of argument array A for computing arrays C and D occur one after the other. As a consequence the second reference is almost certainly a cache hit. Furthermore, minor additional compilation effort allows us to completely eliminate the second array reference and to reuse the value stored in a register as a result of the first array reference. Speaking in terms of parallel execution, WITH-loop-fusion reduces the overhead inflicted by communication and synchronisation. Both original WITH-loops share the argument array A, which must only be communicated once instead of twice. Furthermore, we eliminate the synchronisation barrier between the two original WITH-loops.

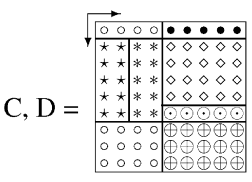

Fig. 7. Illustration of running example after WITH-loop-fusion.

Similar to WITH-loop-folding, we must take some care before applying WITH-loop-fusion. Although we have not observed this in practice, the systematic computation of intersections of index sets may lead to an explosion in the number of generators. The corresponding fragmentation of the index space hinders the generation of efficiently executable code. A separate issue is the growth of the memory footprint if the argument sets of two WITH-loops to be fused are not identical. If the intersection of argument sets is small and the reordering of memory references cannot be exploited to increase data locality to a significant extent, the increased memory footprint may in fact reduce the cache hit rate and, as a consequence, result in an overall performance degradation. While we have rarely found such cases in practice, WITH-loop-fusion must be avoided by a careful analysis of these cases.

A more thorough treatment of WITH-loop-fusion including runtime experiments with its implementation in the SAC compiler can be found in [12].

6. With-loop scalarisation

So far, we have not paid attention to the element types of the arrays involved. Both WITH-loop-folding and WITH-loop-fusion would have had exactly the same effect on arrays of integers or doubles as on arrays of complex numbers. However, a significant difference is that unlike integers and doubles complex numbers are not built-in in SAC. Instead, they are defined as 2-element vectors in SAC itself, and the type `complex` is actually user-defined. As a consequence, our 9×9-element arrays of complex numbers are in fact three-dimensional arrays of shape $9 \times 9 \times 2$. In this light, all operations on the "element level" are, in fact, again array skeletons. For example, in the definition of our addition skeleton:[3]

```
complex[*] (+) (complex[*] A, complex[*] B)
{
  R = with
      (iv) : A[iv] + B[iv]
     genarray(shape( A));
  return(R);
}
```

the application of the infix operator + in the body of the WITH-loop in fact refers to our addition skeleton on double arrays:

```
double[*] (+) (double[*] A, double[*] B)
{
  R = with
      (iv) : A[iv] + B[iv]
     genarray(shape(A));
  return(R);
}
```

Only the application of the infix operator + in the body of the latter WITH-loop actually refers to the built-in arithmetic operator. Combining both skeletons yields two nested WITH-loops:

```
R = with
    (iv) : with
              (kv) : (A[iv])[kv] + (B[iv])[kv]
           genarray(shape(A[iv]));
   genarray(shape(A));
```

[3] To keep the example simple, we silently assume argument arrays to be of identical shape.

Naive compilation of these WITH-loops yields inefficient code. For each index of the outer WITH-loop we effectively create a 2-element array. Later we must copy the values from this intermediate array into the result array R. References to the argument arrays A and B happen in two steps: first we create another intermediate 2-element array holding a single complex number, then we successively extract real and imaginary parts before we actually compute the desired sums. Considering the role of WITH-loops as representations of parallel execution yields substantial further overhead. Generally, we would prefer to exploit concurrency in the outer coarse-grain WITH-loop rather than in the inner fine-grain WITH-loop.

WITH-loop-scalar-isation aims at eliminating all these sources of overhead by merging nested WITH-loops into a single one. The name "scalarisation" reflects our goal to create WITH-loops that effectively operate on the level of scalar values. Fig. 8 demonstrates the effect of WITH-loop-scalar-isation on the running example.

The scalarised WITH-loop contains two generators for each generator of the original outer WITH-loop, one for the corresponding real parts and one for the corresponding imaginary parts of the complex numbers involved. There are no more two-element vectors which results in less memory management overhead. Furthermore, the individual values are directly written into the result arrays without any copying from temporary vectors. The fine-grain skeletons for the additions of complex numbers have been absorbed within the coarse-grain skeleton that constitutes the entire function body now. Last but not least, we can replace aggregate complex constants one and zero by scalar numbers.

Technically spoken, WITH-loop-scalar-isation aims at identifying nested WITH-loops where the generators of the inner WITH-loop do not depend on the index vector of the outer WITH-loop or such a data dependence can be eliminated by other measures. If the outer WITH-loop is a multi-generator WITH-loop this prerequisite must be met by all generator-associated expressions. Likewise, all generator-associated expressions of the outer WITH-loop must contain a suitable inner WITH-loop.

If all conditions are met, we take each generator of the outer WITH-loop and combine it with each generator of the associated inner WITH-loop to form several new generators of the new, scalarised WITH-loop. This is done by concatenating the generator and shape vectors of the WITH-loops involved and by adjusting the index variables accordingly. Subsequently, we replace the typical cascading references into argument arrays like `(A[iv])[kv]` in the code examples above by single references to scalar elements making use of the new index vector of the scalarised WITH-loop.

```
double[9,9,2], double[9,9,2] foo (double[9,9,2] A)
{
 C,D =
  with
    ([0,0,0] <= iv < [1,4,1]) : A[iv] + 1,              0
    ([0,0,1] <= iv < [1,4,2]) : A[iv] + 0,              0
    ([0,4,0] <= iv < [1,9,1]) : A[iv] + 1,              1
    ([0,4,1] <= iv < [1,9,2]) : A[iv] + 0,              0
    ([1,0,0] <= iv < [6,2,1]) : A[iv] + A[iv-[1,-7,0]], 0
    ([1,0,1] <= iv < [6,2,2]) : A[iv] + A[iv-[1,-7,0]], 0
    ([1,2,0] <= iv < [6,4,1]) : A[iv] + A[iv-[1, 2,0]], 0
    ([1,2,1] <= iv < [6,4,2]) : A[iv] + A[iv-[1, 2,0]], 0
    ([1,4,0] <= iv < [4,9,1]) : A[iv] + A[iv-[1, 2,0]], 1
    ([1,4,1] <= iv < [4,9,2]) : A[iv] + A[iv-[1, 2,0]], 0
    ([4,4,0] <= iv < [6,9,1]) : A[iv] + A[iv-[1, 2,0]], A[iv]
    ([4,4,1] <= iv < [6,9,2]) : A[iv] + A[iv-[1, 2,0]], A[iv]
    ([6,0,0] <= iv < [9,4,1]) : A[iv] + 1,              0
    ([6,0,1] <= iv < [9,4,2]) : A[iv] + 0,              0
    ([6,4,0] <= iv < [9,9,1]) : A[iv] + 1,              A[iv]
    ([6,4,1] <= iv < [9,9,2]) : A[iv] + 0,              A[iv]
  genarray( [9,9,2])
  genarray( [9,9,2]);

  return( C, D);
}
```

Fig. 8. Running example after WITH-loop-scalarisation.

A tricky issue in WITH-loop-scalar-isation is the potential duplication of code. For example, in the code fragment

```
with
  (iv) : fun(iv) + with
                     (kv) : ....
                   genarray(...)
genarray(...)
```

the subexpression `fun(iv)` is loop-dependent with respect to the outer WITH-loop, but loop-invariant with respect to the inner WITH-loop. A single scalarised WITH-loop offers no opportunities to express the fact that the computation of `fun` is invariant to the inner axes of the result array. As a consequence, computations of `fun` are effectively duplicated. Whether or not this multiplication of work effectively outweighs the gains of WITH-loop-scalar-isation, depends on the individual case and is often difficult to predict. Therefore, we currently support two variants of WITH-loop-scalar-isation: a conservative variant that only scalarises WITH-loops without duplication of work and an aggressive variant that trades duplication of work for reduced runtime overhead otherwise.

Further details on WITH-loop-scalar-isation including experiments with its implementation in the SAC compiler can be found in [13].

7. Related work

Algorithmic skeletons are a popular approach to high-level parallel programming [4,8,7,17]. In this context frameworks of meaning-preserving transformation rules have been developed that aim at replacing multiple related instances of skeletons by a presumably more efficient combination, sometimes based on a cost model [18–20]. More specific to high-level array processing is the psi-calculus [21] that offers transformation rules on APL-like array operations. Similar optimisations have been proposed on the level of collective operations in MPI [22]. All these approaches have in common that code transformations always remain within the given set of existing skeletons.

In main-stream functional languages such as HASKELL, CLEAN, or ML, separate parts of a program are typically glued together using intermediate data structures other than arrays. Hence, a considerable amount of research effort went into the development of techniques for their detection and elimination. They are generally referred to as *deforestation* or *fusion* techniques [23–27]. Although they are similar in spirit to our optimisations, their setting differs from our's as they are based on linked lists rather than multi-dimensional arrays. Array related research in the area of functional programming has mostly focused on achieving reasonable efficiency in general: [28–30] discuss issues such as strictness, unboxing and the aggregate update problem. A variant of deforestation for arrays is described in [31]; it is similar to WITH-loop-folding adapted to the context of HASKELL arrays.

An optimisation that bears some resemblance to WITH-loop-scalar-isation is the flattening transformation [32] of NESL. In contrast to SAC, arrays in NESL are irregular, e.g., each row of a matrix may have a different size. This format is particularly amenable to the representation of irregular problems or sparse data structures. The flattening operation aims at transforming a multi-dimensional irregular array into a flat data vector and an auxiliary vector encapsulating all structural information.

There is a plethora of work on fusion of FORTRAN-style `do`-loops [33–36]. While the intentions are similar to the objectives of our optimisations, the setting is fairly different. Despite their name, our WITH-loops represent potentially complex array comprehensions with abstract descriptions of multi-dimensional index spaces rather than conventional loops. Whereas WITH-loops define the computation of an aggregate value in an abstract way, `do`-loops merely define a control flow that leads to a specific sequence of read and write operations. Since the fusion of `do`-loops changes this sequence, a compiler must prove that both the old and the new sequence are semantically equivalent. Moreover, the new sequence should be beneficial with respect to some metric. Since both require thorough code analysis, much of the work on loop fusion in FORTRAN is devoted to identification of dependences and anti-dependences on a scalar or elementary level. In contrast, the functional setting of SAC rules out anti-dependences and discloses the data flow. Rather than reasoning on the level of scalar elements, our optimisations address the issue on the level of abstract representations of index spaces.

8. Conclusions and future work

The design of skeletons for expressing concurrent computations usually faces a conflict between software engineering demands and performance issues. Good principles of software engineering call for general-purpose fine-grain skeletons that provide ample opportunities for code reuse and suggest their successive composition into larger programs. In order to achieve high runtime performance, however, coarse-grain, application-specific skeletons are more beneficial. In essence, we face the trade-off between a design that suits the human programmer and a design that suits execution on conventional computing machinery.

In this paper we describe an approach to overcome this dilemma, that we have developed in the context of skeletal generic array programming and implemented as well as evaluated in the context of the functional array language SaC. In SaC all array skeletons are internally mapped to a more general meta skeleton, named WITH-loop. Three tailor-made program transformations, namely WITH-loop-folding, WITH-loop-fusion and WITH-loop-scalar-isation, aim at merging vertical, horizontal and nested compositions of WITH-loops, respectively, into single, more complex WITH-loops. They effectively eliminate temporary, intermediate arrays, reduce the amount of communication and synchronisation overhead, and improve the ratio between productive computation and overhead inflicted by the organisation of parallel program execution.

In conjunction, folding, fusion and scalarisation restructure entire application programs from a representation that is amenable to code development and maintenance towards a representation that is suitable for efficient parallel execution. Having them implemented as compiler optimisations allows programmers to enjoy the benefits of code reuse and improved programming productivity in general that come with the use of a generic, high-level programming environment. Nevertheless, the compiler, by tacitly though aggressively employing code restructuring optimisation techniques, generates executable code that often is similar to low-level hand-coded solutions, both in style and runtime performance achieved.

Space limitations prevent us from providing empirical evidence for our claims. As far as the individual code transformations are concerned, such information may be found in specific papers [14,12,13]. We have also conducted various case studies on evaluating our approach in general for solving non-trivial problems [37–39]. They show that our merging techniques for array skeletons are crucial for high runtime performance both in the sequential and in the parallel case. We achieved sequential runtimes competitive with hand-coded FORTRAN code for the numerical application kernels we examined. Nevertheless, implicit parallelisation showed near-linear speedups on shared memory multiprocessors.

Of course, our approach is not without limitations. Effective application of folding, fusion and scalarisation depends on certain degrees of static knowledge about the code. However, this information is inferred by the compiler from generic specifications like those used in Sections 2 and 3. To this effect we heavily exploit the functional semantics of SaC for data flow analysis and partial evaluation. While the presented code transformations, at least in principle, can be done in an imperative setting as well, their effective application requires code analysis that is generally more difficult than in the functional setting. If for some reason the static knowledge is insufficient for the compiler to effectively apply code transformations with the necessary rigour, runtime performance is likely to be inferior to low-level hand-coded solutions.

We currently work on improving the effective applicability of folding, fusion and scalarisation to a wider range of programs. For the time being, computing intersections between generators, for example, relies on static knowledge of the range boundaries. In the future we aim at covering cases with incomplete static knowledge through symbolic analysis. Another area of future work is the extension of WITH-loop-fusion to fuse WITH-loops whose index spaces are similar, though not identical. In such cases we may extend fusion candidates to the convex hull of their individual index spaces by introducing dummy generators that are not associated with any computation and fuse them thereafter.

References

[1] K. Iverson, A Programming Language, John Wiley, New York City, New York, USA, 1962.
[2] A. Falkoff, K. Iverson, The design of APL, IBM Journal of Research and Development 17 (4) (1973) 324–334.
[3] R. Bernecky, The role of APL and J in high-performance computation, APL Quote Quad 24 (1) (1993) 17–32.
[4] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, Research Monographs in Parallel and Distributed Computing, Pitman, London, UK, 1989.

[5] J. Darlington, Y. Guo, H. To, Y. Jing, Skeletons for Structured Parallel Composition, in: Proceedings on the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95), Santa Barbara, California, USASIGPLAN Notices, vol. 32, ACM Press, 1995, pp. 19–28.
[6] M. Danelutto, F. Pasqualetti, S. Pelagatti, Skeletons for data parallelism in P3L, in: C. Lengauer, M. Griebl, S. Gorlatch (Eds.), Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Passau, Germany, Lecture Notes in Computer Science, vol. 1300, Springer-Verlag, Berlin, Germany, 1997, pp. 619–628.
[7] H. Kuchen, A skeleton library, in: B. Monien, R. Feldmann (Eds.), Proceedings of the 8th European Conference on Parallel Processing (Euro-Par'02), Paderborn, Germany, Lecture Notes in Computer Science, vol. 2400, Springer-Verlag, Berlin, Germany, 2002, pp. 620–629.
[8] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.
[9] S.-B. Scholz, Single assignment C – efficient support for high-level array operations in a functional setting, Journal of Functional Programming 13 (6) (2003) 1005–1059.
[10] C. Grelck, Shared memory multiprocessor support for functional array processing in SAC, Journal of Functional Programming 15 (3) (2005) 353–401.
[11] S.-B. Scholz, With-loop-folding in SAC – condensing consecutive array operations, in: C. Clack, T. Davie, K. Hammond (Eds.), Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers, Lecture Notes in Computer Science, vol. 1467, Springer-Verlag, Berlin, Germany, 1998, pp. 72–92.
[12] C. Grelck, K. Hinckfuß, S.-B. Scholz, With-loop fusion for data locality and parallelism, in: A. Butterfield (Ed.), Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4015, Springer-Verlag, Berlin, Heidelberg, New York, 2006.
[13] C. Grelck, S.-B. Scholz, K. Trojahner, With-loop scalarization: merging nested array operations, in: P. Trinder, G. Michaelson (Eds.), Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers, Lecture Notes in Computer Science, vol. 3145, Springer-Verlag, Berlin, Germany, 2004.
[14] S.-B. Scholz, A case study: effects of WITH-loop folding on the NAS benchmark MG in SAC, in: K. Hammond, T. Davie, C. Clack (Eds.), Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, Selected Papers, Lecture Notes in Computer Science, vol. 1595, Springer-Verlag, Berlin, Germany, 1999, pp. 216–228.
[15] K. Iverson, J Introduction and Dictionary, Iverson Software Inc., Toronto, Canada, 1995.
[16] M. Jenkins, Q'Nial: a portable interpreter for the nested interactive array language Nial, Software Practice and Experience 19 (2) (1989) 111–126.
[17] B. Bacci, S. Gorlatch, C. Lengauer, S. Pelagatti, Skeletons and transformations in an integrated parallel programming environment, in: V. Malyshkin (Ed.), Parallel Computing Technologies (PaCT-99), Lecture Notes in Computer Science, vol. 1662, Springer-Verlag, 1999, pp. 13–27.
[18] S. Gorlatch, C. Lengauer, (De)composition rules for parallel scan and reduction, in: Proceedings of the 3rd International Working Conference on Massively Parallel Programming Models (MPPM'97), London, UK, IEEE Computer Society Press, 1997, pp. 23–32.
[19] S. Gorlatch, S. Pelagatti, A transformational framework for skeletal programs: overview and case study, in: J. Rohlim et al. (Eds.), Parallel and Distributed Processing. IPPS/SPDP'99 Workshops Proceedings, Lecture Notes in Computer Science, vol. 1586, Springer-Verlag, 1999, pp. 123–137.
[20] H. Kuchen, Optimizing sequences of skeleton calls, in: C. Lengauer, D. Batory, C. Consel (Eds.), Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23–28, 2003. Revised Papers, Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, Berlin, Germany, 2004, pp. 254–273.
[21] L.R. Mullin, M. Jenkins, Effective data parallel computation using the psi calculus, Concurrency Practice and Experience 8 (7) (1996) 499–515.
[22] S. Gorlatch, C. Wedler, C. Lengauer, Optimization rules for programming with collective operations, in: M. Atallah (Ed.), Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99), San Juan, Puerto Rico, 1999, pp. 492–499.
[23] P. Wadler, Deforestation: transforming programs to eliminate trees, Theoretical Computer Science 73 (2) (1990) 231–248.
[24] W. Chin, Fusion and tupling transformations: synergies and conflicts, in: Proceedings of the Fuji International Workshop on Functional and Logic Programming, Susono, Japan, World Scientific Publishing, 1995, pp. 106–125.
[25] A. Gill, Cheap Deforestation for Non-strict Functional Languages, Ph.D. thesis, Glasgow University, Glasgow, Scotland, UK, 1996.
[26] H. Seidl, M. Sørensen, Constraints to stop higher-order deforestation, in: Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France, ACM Press, 1997.
[27] D. van Arkel, J. van Groningen, S. Smetsers, Fusion in practice, in: R. Peña, T. Arts (Eds.), Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Selected Papers, Lecture Notes in Computer Science, vol. 2670, Springer-Verlag, Berlin, Germany, 2003, pp. 51–67.
[28] S. Anderson, P. Hudak, Compilation of Haskell array comprehensions for scientific computing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), White Plains, New York, USASIGPLAN Notices, vol. 25, ACM Press, 1990, pp. 137–149.
[29] J. van Groningen, The implementation and efficiency of arrays in Clean 1.1, in: W. Kluge (Ed.), Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany, Selected Papers, Lecture Notes in Computer Science, vol. 1268, Springer-Verlag, Berlin, Germany, 1997, pp. 105–124.

[30] M.M. Chakravarty, G. Keller, An approach to fast arrays in Haskell, in: J. Jeuring, S.P. Jones (Eds.), Summer School and Workshop on Advanced Functional Programming, Oxford, England, UK, 2002, Lecture Notes in Computer Science, vol. 2638, Springer-Verlag, Berlin, Germany, 2003, pp. 27–58.
[31] M.M. Chakravarty, G. Keller, Functional array fusion, in: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01), Florence, Italy, ACM Press, 2001, pp. 205–216.
[32] G. Blelloch, G. Sabot, Compiling collection-oriented languages onto massively parallel computers, Journal of Parallel and Distributed Computing 8 (2) (1990) 119–134.
[33] K. McKinley, S. Carr, C.-W. Tseng, Improving data locality with loop transformations, ACM Transactions on Programming Languages and Systems 18 (4) (1996) 424–453.
[34] N. Manjikian, T. Abdelrahman, Fusion of loops for parallelism and locality, IEEE Transactions on Parallel and Distributed Systems 8 (2) (1997) 193–209.
[35] G. Roth, K. Kennedy, Loop fusion in high performance Fortran, in: Proceedings of the 12th ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia, ACM Press, 1998, pp. 125–132.
[36] J. Xue, Aggressive loop fusion for improving locality and parallelism, in: M. Hermenegildo, D. Cabeza (Eds.), Parallel and Distributed Processing and Applications: Third International Symposium, ISPA 2005, Nanjing, China, Lecture Notes in Computer Science, vol. 3758, Springer-Verlag, 2005, pp. 224–238.
[37] C. Grelck, Implementing the NAS benchmark MG in SAC, in: V.K. Prasanna, G. Westrom (Eds.), Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA, IEEE Computer Society Press, 2002.
[38] C. Grelck, S.-B. Scholz, Towards an efficient functional implementation of the NAS benchmark FT, in: V. Malyshkin (Ed.), Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia, Lecture Notes in Computer Science, vol. 2763, Springer-Verlag, Berlin, Germany, 2003, pp. 230–235.
[39] A. Shafarenko, S.-B. Scholz, S. Herhut, C. Grelck, K. Trojahner, Implementing a numerical solution of the KPI equation using single assignment C: lessons and experiences, in: A. Butterfield (Ed.), Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, September 19–21, 2005, Revised Selected Papers, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, 2006.