John von Neumann Institute for Computing

Merging Compositions of Array Skeletons in SAC

C. Grelck, S.-B. Scholz

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 859-866, 2006.

http://www.fz-juelich.de/nic-series/volume33

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

In the array language SAC, implicitly parallel array skeletons can be defined from meta-skeletons that are part of the language itself. This offers a high potential for code reuse as well as for high-level program specifications. However, the downside of introducing parallelism on the level of relatively simple application building blocks is limited runtime performance due to a generally poor ratio between productive computation on the one side and communication and synchronization on the other side. To overcome this limitation we have developed compiler optimizations that identify different composition scenarios of skeletons each of which is addressed by a specific code transformation technique that merges individual skeletons into a single more powerful one. This paper demonstrates how these transformations can be combined to systematically restructure compositions of skeletons into more complex skeletons that, in turn, can be compiled into efficiently executable parallel code.

1. Introduction

SAC (Single Assignment C) [12] is an implicitly parallel, purely functional array language designed with numerical applications in mind. The language design of SAC aims at combining generic, high-level specifications of array-based algorithms with a runtime performance that is competitive with low-level, machine-oriented languages. Compiler-directed parallelization of SAC programs [9] allows shared memory multiprocessor machines to be utilized without any additional programming effort.

Parallelism in SAC programs stems from the use of array skeletons like maps of scalar operations, rotation and shifting operations, subarray selections, or reduction operations. In SAC, all these skeletons are defined in the language itself by means of meta-skeletons, named WITH-loops. As a consequence, SAC programs usually consist of various layers of skeleton composition. This programming style leads to highly generic code with good opportunities for reuse on each layer.

The downside of this approach is that it also leads to deeply nested compositions of WITH-loops that are computationally light-weight. With parallel program execution bound to individual WITH-loops, the ratio between productive computation and synchronization/communication overhead tends to be unfavorable. In order to overcome this limitation we have developed optimization techniques that systematically restructure compositions of WITH-loops from a representation that is amenable to code development and maintenance towards a representation that is suitable for efficient parallel execution.

We have identified three different types of composition: vertical, horizontal, and nested composition. Vertical composition describes computational pipelines where the result of one WITH-loop becomes the argument of another WITH-loop. In order to avoid this kind of composition, the computational pipelines need to be shifted from the level of entire arrays to the level of individual elements. This is achieved by an optimization technique called WITH-loop-folding [11]. In essence, it performs forward substitutions from one WITH-loop body of a computational pipeline into the body

of the subsequent WITH-loop until the first WITH-loop becomes obsolete.

Horizontal composition is characterized by multiple WITH-loops that compute separate results from the same or at least an overlapping set of arguments. Provided there are no data dependencies between the WITH-loops under consideration, these can be combined into a single WITH-loop that computes all results simultaneously. As this technique to some extent resembles classical loop fusion techniques it is called WITH-loop-fusion. When applied, it results in multi-operator WITH-loops. They represent multiple skeleton operations and compute several arrays in a single step.

Nested composition occurs whenever individual elements of a WITH-loop-defined array are defined by yet another WITH-loop. Under certain circumstances such nestings can be combined into single WITH-loops that operate one scalar values eliminating all micro-synchronizations that would result from a naive compilation otherwise. A transformation scheme to this effect is WITH-loop-scalarization [10].

While folding, fusion, and scalarization are orthogonal in the intermediate code scenarios they address, all three improve the quality of parallelized code by reducing the need for synchronization and communication. By systematically merging computationally light-weight array skeletons into more complex operations they also improve the quality and the efficiency of scheduling workload to processing units. In this paper we demonstrate their combined effect on the optimization and parallelization of intermediate SAC code.

The remainder of this paper is organized as follows. Section 2 gives a rough idea of the design of SAC and provides a brief introduction into WITH-loops. In Section 3, we introduce a running example. Sections 4, 5, and 6 sketch the three optimization techniques and discuss their effect on the running example. Related work is presented in Section 7 and Section 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 behavior (for details see [12]).

On top of this language core SAC supports arrays as the only data structure. For manipulating these, several variants of meta-skeletons, the so called WITH-loops, are provided. As they are first class citizens of the language, i.e., they can be used in any expression position, all basic skeletons for array manipulation such as maps of scalar operations, rotation and shifting operations, can be defined in terms of WITH-loops.

For the purpose of this paper we consider WITH-loops program expressions that take the general form

with (*lower* <= *idx_vec* < *upper*) : *expr* ;
genarray(*shape*, *default*)

where *idx_vec* is an identifier, *lower*, *upper*, and *shape* denote expressions that should evaluate to vectors of identical length, and *expr* and *default* denote arbitrary expressions. Such a WITH-loop defines an array of shape *shape*, whose elements are either computed by the expression *expr* or by the default expression *default*. Which of these two values is chosen for an individual element depends on its location, i.e., it depends on its index position. If the index is within the range specified by

the lower bound *lower* and the upper bound *upper*, *expr* is chosen, otherwise *default* is taken. As a simple example, consider the WITH-loop

```
with ([1] <= iv < [4]) : 2;
genarray( [5], 0)
```

It computes the vector `[0, 2, 2, 2, 0]`. Note here, that the use of vectors for the shape of the result and the bounds of the index space (also referred to as the **generator**) allows WITH-loops to denote arrays of arbitrary rank. Furthermore, the **generator expression** *expr* may refer to the index position through the **generator variable** *idx_vec*. For example, the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1];
genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 4 & 0 \\ 0 & 3 & 4 & 5 & 0 \end{pmatrix}$.

It should be mentioned here, that these WITH-loops constitute a restricted form of the WITH-loops in SAC. Exact definitions of fully-fledged WITH-loops and the formal definitions of transformations on them can be found elsewhere [12,11,10].

3. Running example

We illustrate the combined effect of our three WITH-loop optimizations folding, fusion, and scalarization by a running example. In order to demonstrate complexity and versatility of the individual optimizations without making the example overly complicated, we use a rather artificial function `foo` as shown in Fig. 1. It takes a 9x9-element matrix of complex numbers as an argument and

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  B = take( [5,9], A) +0+ genarray( [4,9], [1.0, 0.0]);
  C = shift( [1,2],
             shift( [-1,-2],
                    A + shift( [1,1],
                               B,
                               [0.0, 0.0]),
                    [0.0, 0.0]),
             [0.0, 0.0]);
  D = take( [9,7], B) +1+ genarray( [9,2], [0.0, 0.0]);
  return( C, D);
}
```

B= C= D=

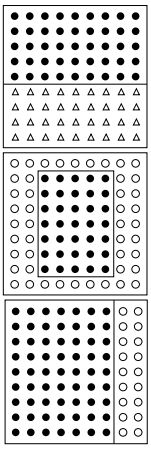


Figure 1. Running example.

yields two such matrices as results.

Both return arrays `C` and `D` are defined in terms of the argument array `A` and an intermediate array `B`. The latter is computed from `A` by taking the first five rows of `A` and by concatenating this 5×9 array with a 4×9 array of complex numbers of value 1. Essentially, the first result array `C` is defined as sums of the corresponding elements of `A` and of `B` where the latter elements are additionally shifted by one index position along both axes. The result of this addition is embedded

in two further shift operations which push zeros into the first and last row as well as into the first and last two columns. For result array D we take the corresponding elements of B for the leftmost seven columns and initialize the remaining two columns to complex zero.

Note here, that all array operations such as **take**, **shift**, **+0+** or + in fact are defined by WITH-loops. Inlining their definitions leads to more then 10 WITH-loops rendering a naive compilation prohibitive. Rather than explaining the entire merging process, we focus on the last few steps that need to be applied after the individual definitions of B, C and D have been merged. Fig. 2 shows a beautified version of that intermediate form. The concatenations with arrays of zeros are expressed

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  B = with ([0,0] <= iv < [5,9]) : A[iv]
      genarray( [9,9], [1.0,0.0]);
  C = with ([1,2] <= iv < [8,7]) : A[iv] + B[iv-1]
      genarray( [9,9], [0.0,0.0]);
  D = with ([0,0] <= iv < [9,7]) : B[iv]
      genarray( [9,9], [0.0,0.0]);
  return( C, D);
}
```

Figure 2. Running example as WITH-loops.

as WITH-loops with default element zero. Similarly, the outer two shift operations in the definition of C have been resolved into a zero default element, and the inner shift operation is expressed as an index offset when accessing the array B.

Before applying any of the three WITH-loop optimizations, all WITH-loops are transformed into a slightly more complex representation that makes the default elements explicit by adding further generators associated with the default expression. Fig. 3 shows the result of this preprocessing step for the second WITH-loop.

```
C = with ([0,0] <= iv < [1,9]) : [0.0,0.0]
         ([1,0] <= iv < [8,2]) : [0.0,0.0]
         ([1,2] <= iv < [8,7]) : A[iv] + B[iv-1]
         ([1,7] <= iv < [8,9]) : [0.0,0.0]
         ([8,0] <= iv < [9,9]) : [0.0,0.0]
    genarray( [9,9]);
```

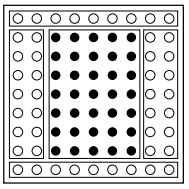
C= 

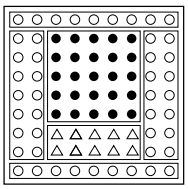
Figure 3. Creating a full partition for the second WITH-loop of the running example.

4. With-loop folding

Our first optimization technique, WITH-loop-folding, addresses vertical compositions of WITH-loops. In the running example introduced in the previous section, we have vertical compositions between the first and the second WITH-loop and again between the first and the third WITH-loop.

Technically spoken, WITH-loop-folding aims at identifying array references within the generator-associated expressions in WITH-loops. If the index expression is an affine function of the WITH-loop's index variable and if the referenced array is itself defined by another WITH-loop, the array reference is replaced by the corresponding element computation. Instead of storing an intermediate result in a temporary data structure and taking the data from there when needed, we forward-substitute the computation of the intermediate value to the place where it is actually needed.

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  C = with ([0,0] <= iv < [1,9]) : [0.0,0.0]
           ([1,0] <= iv < [8,2]) : [0.0,0.0]
           ([1,2] <= iv < [6,7]) : A[iv] + A[iv-1]
           ([1,7] <= iv < [8,9]) : [0.0,0.0]
           ([6,2] <= iv < [8,7]) : A[iv] + [1.0,0.0]
           ([8,0] <= iv < [9,9]) : [0.0,0.0]
      genarray( [9,9]);
  D = with ([0,0] <= iv < [5,7]) : A[iv]
           ([0,7] <= iv < [5,9]) : [0.0,0.0]
           ([5,0] <= iv < [9,7]) : [1.0,0.0]
           ([5,7] <= iv < [9,9]) : [0.0,0.0]
      genarray( [9,9]);
  return( C, d);
}
```

C= 

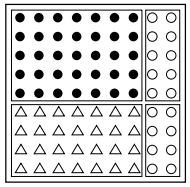
D= 

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

The challenge of WITH-loop-folding lies in the identification of the correct expression which is to be forward-substituted. Usually, the referenced WITH-loop has multiple generators each being associated with a different expression. Hence, we must decide which of the index sets defined by the generators is actually referenced. To make this decision we must take into account the entire generator sequence of the referenced WITH-loop, the generator of the referencing WITH-loop that is associated with the expression which contains the array reference under consideration, and the affine function defining the index. As demonstrated by the example in Fig. 4, this process generally involves intersection of generators. For example, folding the first WITH-loop into the second one results in splitting the index range of the addition into two separate ones.

5. With-loop fusion

WITH-loop-fusion addresses horizontal compositions of WITH-loops. Horizontal composition is characterized by two a more WITH-loops without data dependences that iterate over the same index space. In our running example the WITH-loops defining the result arrays `C` and `D` form such a horizontal composition. The idea of WITH-loop-fusion is to combine horizontally composed WITH-loops into a more versatile internal representation named multi-operator WITH-loop. The major characteristic of multi-operator WITH-loops is their ability to define multiple array comprehensions and multiple reduction operations as well as mixtures thereof.

Fig. 5 shows the effect of WITH-loop-fusion on the running example. As a consequence of the code transformation both result arrays `C` and `D` are computed in a single sweep. This allows us to share the overhead inflicted by the multi-dimensional loop nest among computing both `C` and `D`.

```
complex[9,9], complex[9,9] foo (complex[9,9] A)
{
  C,D = with ([0,0] <= iv < [1,7]) : [0.0,0.0], A[iv]
             ([0,7] <= iv < [1,9]) : [0.0,0.0], [0.0,0.0]
             ([1,0] <= iv < [5,2]) : [0.0,0.0], A[iv]
             ([5,0] <= iv < [8,2]) : [0.0,0.0], [1.0,0.0]
             ([1,2] <= iv < [5,7]) : A[iv] + A[iv-1], A[iv]
             ([5,2] <= iv < [6,7]) : A[iv] + A[iv-1], [1.0,0.0]
             ([6,2] <= iv < [8,7]) : A[iv] + [1.0,0.0], [1.0,0.0]
             ([1,7] <= iv < [5,9]) : [0.0,0.0], [0.0,0.0]
             ([5,7] <= iv < [8,9]) : [0.0,0.0], [0.0,0.0]
             ([8,0] <= iv < [9,7]) : [0.0,0.0], [1.0,0.0]
             ([8,7] <= iv < [9,9]) : [0.0,0.0], [0.0,0.0]
        genarray( [9,9])
        genarray( [9,9]);
  return( C, D);
}
```

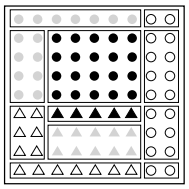


Figure 5. Running example after WITH-loop-fusion.

Furthermore, we change the order of array references at runtime. The intermediate code as shown in Fig. 4 accesses large parts of array A in both WITH-loops. Assuming array sizes typical for numerical computing, elements of A are extremely likely not to reside in cache memory any more when they are needed for execution of the second WITH-loop. With the fused code in Fig. 5 both array references `A[iv]` occur in the same WITH-loop iteration and, hence, the second one always results in a cache hit.

Technically, WITH-loop-fusion requires systematically computing intersections of generators in a way similar to WITH-loop-folding. After identification of suitable WITH-loops, we compute the intersections of all pairs of generators. Whereas this leads to a quadratic increase in the number of generators for the worst case, many of the new generators turn out to be empty in practice as can be seen for our example.

6. With-loop scalarization

So far, we have not paid any attention to the element types of the arrays involved. In SAC, complex numbers are not built-in, but they are defined as vectors of two elements of type double. As a consequence, our 9×9 arrays of complex numbers are in fact three-dimensional arrays of shape `[9,9,2]` and the addition operation on complex numbers in fact is defined by a WITH-loop over vectors of two elements. The idea of WITH-loop-scalarization is to get rid of these nestings of withloops and to transform them into WITH-loops that operate on scalar values. This is achieved by concatenating the bound and shape expressions of the WITH-loops involved and by adjusting the generator variables accordingly. For our example we obtain code equivalent to the code shown in Fig. 6. When comparing this code against the code of Fig. 5, we can observe several benefits. There are no more two-element vectors which results in less memory allocations and deallocations at runtime. 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.

```
double[9,9,2], double[9,9,2] foo (double[9,9,2] A)
{
  C,D = with ...
             ([1,2,0] <= iv < [5,7,1]) : A[iv] + A[iv-1], A[iv]
             ([1,2,1] <= iv < [5,7,2]) : A[iv] + A[iv-1], A[iv]
             ...
             ([6,2,0] <= iv < [8,7,1]) : A[iv] + 1.0, 1.0
             ([6,2,1] <= iv < [8,7,2]) : A[iv] + 0.0, 0.0
             ...
        genarray( [9,9,2])
        genarray( [9,9,2]);
  return( C, D);
}
```

Figure 6. Running example after WITH-loop-scalarization.

7. Related work

For imperative array languages, such as various FORTRAN dialects or ZPL [6], optimizations for combining loop constructs typically concentrate on classical fusion techniques [1] that are applied on the level of individual loops rather than entire loop nestings. Forward-substitutions from the body of one loop nesting to another one, as done by WITH-loop-folding, usually are not made due to the lack of a side-effect free setting. Optimizations like WITH-loop-scalarization as well have not been pursued because in an imperative context operational aspects are decoupled from data layout aspects: memory representations of arrays are defined through explicit declaration, not by the operations that incrementally initialize their elements.

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. However, 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 [15,7,8,13]. Although being similar in spirit, they completely differ from our setting as they are based on linked lists while we operate on multidimensional arrays. Array related research in the area of functional programming has mostly focused on achieving reasonable efficiency in general. [2,14,5] discuss issues such as strictness, unboxing, and the aggregate update problem. A variant of deforestation for arrays is described in [4]; it is similar in spirit to WITH-LOOP-FOLDING adapted to the context of HASKELL arrays.

A notable exception from the main-stream of functional programming that puts the emphasis on arrays rather than on lists is SISAL. However, according to [3] the loop optimizations in the context of SISAL are merley restricted to the conventional setting, i.e., to the fusion of individual loops.

8. Conclusions

The design of skeletons for expressing concurrent computations usually faces a conflict between software-engeneering demands and performance issues. The former favor versatile small-grain skeletons that can be successively combined into larger programs, whereas from a performance perspective, coarse grain skeletons are better suitable.

This paper demonstrates in the context SAC how WITH-loop-folding, WITH-loop-fusion, and WITH-loop-scalarization can be used to combine the benefits of both approaches. SAC programs are typically specified as combinations of APL-like array operations that are defined in terms of

fine-grain WITH-loops. After inlining these definitions, the three optimizations, when used jointly, in most cases suffice to merge rather complex compositions of trivial WITH-loops into fewer, more complex WITH-loops that are better suited for an efficent concurrent execution.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[2] S. Anderson and 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, USA*, volume 25 of *SIGPLAN Notices*, pages 137–149. ACM Press, 1990.

[3] D.C. Cann. *The Optimizing SISAL Compiler: Version 12.0*. Lawrence Livermore National Laboratory, Livermore, California, 1993. part of the SISAL distribution.

[4] Manuel M.T. Chakravarty and Gabriele Keller. Functional Array Fusion. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01), Florence, Italy*, pages 205–216. ACM Press, 2001.

[5] Manuel M.T. Chakravarty and Gabriele Keller. An Approach to Fast Arrays in Haskell. In Johan Jeuring and Simon Peyton Jones, editors, *Summer School and Workshop on Advanced Functional Programming, Oxford, England, UK, 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer-Verlag, Berlin, Germany, 2003.

[6] B.L. Chamberlain, S.-E. Choi, C. Lewis, L. Snyder, W.D. Weathersby, and C. Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3), 1998.

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

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

[9] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[10] C. Grelck, S.-B. Scholz, and K. Trojahner. With-Loop Scalarization: Merging Nested Array Operations. In P. Trinder and G. Michaelson, editors, *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers*, volume 3145 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2004.

[11] S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 72–92. Springer-Verlag, Berlin, Germany, 1998.

[12] S.-B. Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[13] D. van Arkel, J. van Groningen, and S. Smetsers. Fusion in Practice. In R. Peña and T. Arts, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, Berlin, Germany, 2003.

[14] J. van Groningen. The Implementation and Efficiency of Arrays in Clean 1.1. In W. Kluge, editor, *Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany, Selected Papers*, volume 1268 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, Berlin, Germany, 1997.

[15] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990.