With-Loop Scalarization – Merging Nested Array Operations

Clemens Grelck[1], Sven-Bodo Scholz[2], and Kai Trojahner[1]

[1] University of Lübeck
Institute of Software Technology and Programming Languages
{grelck,trojahne}@isp.uni-luebeck.de
[2] University of Kiel
Institute of Computer Science and Applied Mathematics
sbs@informatik.uni-kiel.de

Abstract. Construction of complex array operations by composition of more basic ones allows for abstract and concise specifications of algorithms. Unfortunately, naïve compilation of such specifications leads to creation of many temporary arrays at runtime and, consequently, to poor performance characteristics.
This paper elaborates on a new compiler optimization, named WITH-LOOP-SCALARIZATION, which aims at eliminating temporary arrays in the context of nested array operations. It is based on WITH-loops, a versatile array comprehension construct used by the functional array language SAC both for specification as well as for internal representation of array operations.
The impact of WITH-LOOP-SCALARIZATION on the runtime performance of compiled SAC code is demonstrated by several experiments involving support for arithmetic on arrays of complex numbers and the application kernel FT from the NAS benchmark suite.

1 Introduction

Dedicated array languages like APL [19], J [20], or NIAL [21] allow for very abstract and concise specifications when processing large amounts of data homogeneously structured along multiple orthogonal axes. They provide large sets of built-in operations, which are universally applicable to arrays of any rank (number of axes) and of any shape (extent along individual axes). These basic operations form the building blocks for construction of entire application programs in a step-wise and layered process.

The advantages of this programming style are manifold. Arrays are treated as conceptual entities with certain algebraic properties rather than as loosely coupled collections of elements. Operations handle entire arrays in a homogeneous way; explicit indexing, which may be considered the most error-prone property of conventional array processing, is almost completely avoided.

However, this specificational advantage does not come for free. Compilation of such specifications turns out to be rather difficult as soon as runtime performance matters. Besides the challenge of compiling the basic array operations

P. Trinder, G. Michaelson, and R. Peña (Eds.): IFL 2003, LNCS 3145, pp. 118–134, 2004.

into efficiently executable code [1, 7, 4, 26] the main problem is the compositional nature of programs in general. Separate compilation of individual basic array operations requires all intermediate results of a complex operation to be explicitly created. While in a scalar language such values can be held in registers, in an array language entire arrays have to be created. As this incurs substantial overhead, one of the key challenges is to develop techniques which avoid actual creation of such intermediate arrays at runtime.

Different sources of intermediate arrays may be distinguished. The most prominent source are array operations that are defined as sequences of basic operations where, in a pipelined fashion, the result of one basic operation directly serves as argument of the subsequent one. As a simple example, consider the selection of the inner elements of an array. In most array languages this can be specified as an expression that takes all but the last elements of an (intermediate) array that itself is derived from the initial array by dropping the very first elements along each axis. Naïve compilation explicitly creates the intermediate array that contains all but the first elements. To avoid the associated overhead, several elaborate techniques have been developed. They reach from *drag-along and beating* [1] to WITH-LOOP-FOLDING [24].

A different source of intermediate arrays are nested operations on arrays. Often it often turns out to be convenient to consider an n-dimensional array to be an $n-m$-dimensional array of m-dimensional subarrays. Prominent examples are applications where individual array elements are arrays themselves, e.g. arrays of complex numbers or vectors of linear functions each being represented by a matrix. Further examples include operations that are to be applied to selected axes of an array only, i.e., an outer operation splits up a given array, applies an inner operation to individual subarrays, and recombines individual results into the overall result. In all these cases naïve compilation creates a temporary representation for each intermediate subarray.

This paper presents on optimization technique called WITH-LOOP-SCALARIZATION, which aims at avoiding this kind of intermediate arrays. It is based on a meta representation for high-level array operations called WITH-loop, as proposed in the context of the functional array programming language SaC (for Single Assignment C) [26]. The basic idea of WITH-LOOP-SCALARIZATION has been sketched out in the context of SaC's *axis control notation* [16], a WITH-loop-based technique for applying array operations to selected axes of an array. The particular contributions of this paper are

- a new optimization scheme which is based on a more flexible meta representation called multi-generator WITH-loop, rather than on ordinary WITH-loops. This allows WITH-LOOP-SCALARIZATION to interact with several other optimizations such as WITH-LOOP-FOLDING, which turns out to be mutually beneficial.
- extended auxiliary transformation schemes that further enhance the applicability of WITH-LOOP-SCALARIZATION in general.
- investigations on the performance impact of WITH-LOOP-SCALARIZATION in the current SaC compiler release.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction into the basic concepts of SAC for those readers who are not yet familiar with the language. Multi-generator WITH-loops as the basis for the definition of WITH-LOOP-SCALARIZATION are sketched out in Section 3. Section 4 introduces the basic compilation scheme realizing WITH-LOOP-SCALARIZATION, while Section 5 discusses the auxiliary transformation schemes. The impact of WITH-LOOP-SCALARIZATION on runtime performance is investigated in Section 6. After covering some related work in Section 7, Section 8 concludes and outlines directions of future work.

2 SAC – A Brief Introduction

The core language of SAC is a functional subset of C, extended by n-dimensional arrays as first class objects. Despite the different semantics, a rule of thumb for SAC code is that everything that looks like C also behaves as in C. Arrays are represented by two vectors, a shape vector that specifies an array's extent wrt. each of its axes, and a data vector that contains all its elements. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays with a fixed number of dimensions, e.g. `int[.,.]`, and arrays with any number of dimensions, i.e. `int[+]`.

In contrast to other array languages SAC provides only a very small set of built-in operations on arrays. Basically, they are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`). Aggregate array operations are specified in SAC itself using powerful array comprehensions, called WITH-loops. Their (simplified) syntax is outlined in Fig. 1.

WithLoopExpr ⇒ with (*Generator*) ⌈ *AssignBlock* ⌋ *Operation*
Generator ⇒ *Expr RelOp Id RelOp Expr* ⌈ *Filter* ⌋
RelOp ⇒ < | <=
Filter ⇒ step *Expr* ⌈ width *Expr* ⌋
Operation ⇒ genarray (*Expr* , *Expr*) | ...

Fig. 1. Syntax of with-loop expressions.

A WITH-loop basically consists of two parts: a *generator* and an *operation*. The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions, which must evaluate to vectors of equal length, define lower and upper bounds of a rectangular index vector range. An optional filter may further restrict this selection to grids of arbitrary width. Let a, b, s, and w denote expressions that evaluate to vectors of length n, then

`(` a `<= i_vec <` b `step` s `width` w `)`

defines the following set of index vectors:

$$\{i\_vec \mid \forall_{j \in \{0,\ldots,n-1\}} : a_j \leq i\_vec_j < b_j \ \wedge \ (i\_vec_j - a_j) \bmod s_j < w_j\} \ .$$

The operation specifies the computation to be performed for each element of the index vector set defined by the generator. Let *shp* denote a SAC expres-

sion that evaluates to a vector, and let *expr* denote any SaC expression. Then `genarray(` *shp*`,` *expr*`)` defines an array of shape *shp* whose elements are the values of *expr* for all index vectors from the generator-specified set and 0 otherwise. In order to simplify specification of complex expressions, the operation part may be preceded by a block of local variable definitions, and *expr* may be defined in terms of these variables.

Additional types of operation parts allow definition of various map- and fold-like operations. Since they are not needed in the scope of this paper, we omit their definition here and refer to [26], which provides a detailed introduction into SaC. A case study on a non-trivial problem investigating both the programming style and the resulting runtime performance is presented in [14]. Additional information on SaC is available at `http://www.sac-home.org/`.

3 Multi-generator With-Loops

As pointed out in the introduction, WITH-LOOP-FOLDING [25], a SaC-specific optimization technique plays a vital role in achieving high runtime performance. Its purpose is to avoid the creation of intermediate arrays by condensing consecutive WITH-loops into a single one according to the well-known equivalence

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

A simple WITH-LOOP-FOLDING example is shown in Fig. 2: selection of all inner elements of an array by a combination of `take` and `drop` For reasons of simplicity, we use constant boundary expressions in this example and expect the argument array `A` to be of shape `[100,100]`. While `take([99,99],A)` "takes" the first 99 rows and columns of the argument matrix `A`, the subsequent `drop([1,1],...)` "drops" the first row and the first column of the intermediate matrix. Inlining both `take` and `drop` yields two consecutive WITH-loops; subsequent WITH-LOOP-FOLDING transforms them into a single operation that selects all inner elements of `A` directly, i.e. without creating an intermediate array.

The example shown in Fig. 2 represents a trivial case of WITH-LOOP-FOLDING as the second generator defines a subset of index positions of the first generator.

```
res = drop( [1,1], take( [99,99], A));

            ⇓        FUNCTION INLINING

tmp = with ([0,0] <= iv < [99,99])
      genarray( [99,99], A[iv]);
res = with ([0,0] <= iv < [98,98])
      genarray( [98,98], tmp[iv+[1,1]]);

            ⇓        WITH-LOOP-FOLDING

res = with ([0,0] <= iv < [98,98])
      genarray( [98,98], A[iv+[1,1]]);
```

Fig. 2. WITH-LOOP-FOLDING example.

As a consequence, the entire operation can still be represented by a single generator. However, generators of subsequent WITH-loops may also define disjoint or overlapping sets of index vectors. In these cases, different elements of the target array must be computed according to different specifications, a property which is not supported by WITH-loops.

$$\begin{array}{lcl} \mathit{WithLoopExpr}' & \Rightarrow & \text{with } \left\lceil \mathit{Part} \right\rfloor^{+} \mathit{Operation} \\ \mathit{Part} & \Rightarrow & \mathit{Generator} \left\lceil \mathit{AssignBlock} \right\rfloor : \mathit{Expr} \end{array}$$

Fig. 3. Pseudo syntax of multi-generator WITH-loops.

To address this problem and to create a representation that is closed under WITH-LOOP-FOLDING, user-level WITH-loops are internally embedded into a more general representation called *multi-generator* WITH-*loop* [15]. Its pseudo syntax is defined in Fig. 3. The main difference between internal multi-generator WITH-loops and user-level WITH-loops is that the former consist of an entire sequence of *parts*. Each part is made up by an individual generator, an associated goal expression, and an optional block of local declarations. Being an internal format only allows to guarantee certain regularity properties, e.g., the index variables in the various generators are all the same, and the set of generators forms a partition of the target array's index space, i.e., each element of the target array is covered by exactly one generator.

The importance of multi-generator WITH-loops lies in the fact that all array operations in SAC are internally represented in this format. Hence, any optimization technique on array operations must be defined based on this representation. For additional information on multi-generator WITH-loops see [15] or [26].

4 With-Loop Scalarization – The Base Case

A convenient way of describing complex array operations is to map the basic operations defined in the SAC standard library to arrays of higher rank by means of WITH-loops. Since the library operations are themselves implemented by WITH-loops, this layered approach to software design results in nested WITH-loops in intermediate code, an example of which is shown in Fig. 4.

```
A = with ([0] <= iv < [4]) {
      B = with ([0] <= jv < [4])
          genarray( [4], iv[0] + 2 * jv[0]);
    }
    genarray( [4], B);
```

$$A = \begin{pmatrix} 0 & 2 & 4 & 6 \\ 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 3 & 5 & 7 & 9 \end{pmatrix}$$

Fig. 4. Array A is defined by two nested WITH-loops.

Unfortunately naïve compilation does not translate nested WITH-loops into efficient programs. Like in the case of consecutive array operations addressed by

WITH-LOOP-FOLDING creation of temporary arrays forms the main obstacle for achieving competitive runtime performance. Even worse, the number of intermediate arrays is not proportionate to the number of consecutive operations, but to the size of the index range defined by the outer generator.

WITH-LOOP-SCALARIZATION is a high-level program transformation that approaches this problem by merging nested WITH-loops into single ones. For example, application of WITH-LOOP-SCALARIZATION to the WITH-loops in Fig. 4 would replace the nesting with the equivalent WITH-loop shown in Fig. 5.

```
A = with ([0,0] <= iv < [4,4])
    genarray( [4,4], iv[0] + 2 * iv[1]);
```

$$A = \begin{pmatrix} 0\,2\,4\,6 \\ 1\,3\,5\,7 \\ 2\,4\,6\,8 \\ 3\,5\,7\,9 \end{pmatrix}$$

Fig. 5. Array A is generated by a single scalar WITH-loop.

The code transformation applied can be generalized to the compilation scheme shown in Fig. 6. For illustrative purposes we define WITH-LOOP-SCALARIZATION on ordinary WITH-loops first and extend this scheme to multi-generator WITH-loops later in this section. WITH-LOOP-SCALARIZATION replaces two nested WITH-loops with a single one, which is defined as follows:

- The new generator's boundary, step, and width vectors result from the concatenation (denoted by ++) of the original outer WITH-loop's vectors with the corresponding vectors of the inner WITH-loop.
- The shape vector is also defined by the concatenation of the two original shape vectors.
- The body equals that of the inner WITH-loop prepended with a reconstruction of the two former index vectors. To maintain dimension invariance, these are defined by `take` and `drop` operations performed on the new index vector.

$$\mathcal{SWLS}\left[\!\!\left[\begin{array}{l} \texttt{with (} lb_1 \texttt{ <= } iv_1 \texttt{ < } ub_1 \texttt{ step } s_1 \texttt{ width } w_1\texttt{) \{} \\ \quad val_{outer} \texttt{ = with (} lb_2 \texttt{ <= } iv_2 \texttt{ < } ub_2 \texttt{ step } s_2 \texttt{ width } w_2\texttt{)} \\ \qquad \texttt{\{} \\ \qquad\quad val_{inner} \texttt{ = } expr\texttt{(} iv_1, iv_2\texttt{);} \\ \qquad \texttt{\} genarray(} shape_{inner}\texttt{, } val_{inner}\texttt{);} \\ \texttt{\} genarray(} shape_{outer}\texttt{, } val_{outer}\texttt{)} \end{array}\right]\!\!\right]$$

$$= \left\{\begin{array}{l} \texttt{with (} lb_1\texttt{++}lb_2 \texttt{ <= } iv \texttt{ < } ub_1\texttt{++}ub_2 \texttt{ step } s_1\texttt{++}s_2 \texttt{ width } w_1\texttt{++}w_2\texttt{) \{} \\ \quad iv_1 \texttt{ = take(shape(} lb_1\texttt{), } iv \texttt{);} \\ \quad iv_2 \texttt{ = drop(shape(} lb_1\texttt{), } iv \texttt{);} \\ \quad val_{inner} \texttt{ = } expr\texttt{(} iv_1, iv_2\texttt{);} \\ \texttt{\} genarray(} shape_{outer}\texttt{++}shape_{inner}\texttt{, } val_{inner}\texttt{)} \end{array}\right.$$

if $iv_1 \notin FV(lb_2) \ \wedge\ iv_1 \notin FV(ub_2) \wedge\ iv_1 \notin FV(s_2) \wedge\ iv_1 \notin FV(w_2)$

Fig. 6. A simplified compilation scheme for WITH-LOOP-SCALARIZATION.

– The result expression of the new WITH-loop is the same as that of the original inner WITH-loop.

This scheme can be applied to most of the cases in which the elements of an outer WITH-loop are defined by an inner WITH-loop. However, WITH-LOOP-SCALARIZATION cannot be applied if lb_2, ub_2, s_2, or w_2 depend on iv_1, since no reference must be lifted outside the binding scope of the variable it refers to.

As pointed out in Section 3, user-level WITH-loops are internally embedded into multi-generator WITH-loops. Hence, the compilation scheme illustrates the working principle of WITH-LOOP-SCALARIZATION, but further generalization is required to adapt it to multi-generator WITH-loops. To reuse elements of the simplified compilation scheme, multi-generator WITH-LOOP-SCALARIZATION is split into two consecutive phases. The first phase, called *distribution phase*, deals with multiple generators occurring in inner WITH-loops. It distributes the generator of the surrounding part over all inner parts. This is done by creating an outer part for each of the inner parts, as depicted in Fig. 7. Step and width vectors are omitted for reasons of simplicity. They are treated in the same way as boundary vectors.

The internal representation of multi-generator WITH-loops as a result of the distribution phase is characterized by overlapping generators. However, this deficiency is addressed by the subsequent *scalarization phase*, which completes WITH-LOOP-SCALARIZATION. Basically, this is achieved by mapping the simplified compilation scheme from Fig. 6 to all parts of the outer WITH-loop, as shown in Fig. 8. Finally, the composition of both compilation phases yields a scheme for multi-generator WITH-LOOP-SCALARIZATION:

$$\mathcal{WLS} = \mathcal{DIST} \circ \mathcal{SCAL} \quad .$$

It remains to be pointed out that the same restrictions apply for multi-generator WITH-LOOP-SCALARIZATION as mentioned for the simplified case. If some inner generator depends on the index vector of an outer generator, the compound operation cannot be expressed by a single multi-generator WITH-loop and, hence, there is no opportunity for WITH-LOOP-SCALARIZATION.

5 With-Loop Scalarization – Enhancing Applicability

Unfortunately, WITH-LOOP-SCALARIZATION as defined in the previous section is insufficient to handle all cases of nested array operations. The compilation scheme for multi-generator WITH-loops requires each part of an outer WITH-loop to contain exactly one nested WITH-loop. However, in many cases intermediate code which would benefit from WITH-LOOP-SCALARIZATION does not comply to this restricted format, e.g., an inner WITH-loop may be accompanied by additional code or the inner non-scalar expression may not be given as a WITH-loop at all.

This section is about making WITH-LOOP-SCALARIZATION applicable in a broader range of optimization cases. Three auxiliary transformation schemes are presented which tackle a specific code pattern each and result in nested WITH-loops, thus enabling WITH-LOOP-SCALARIZATION.

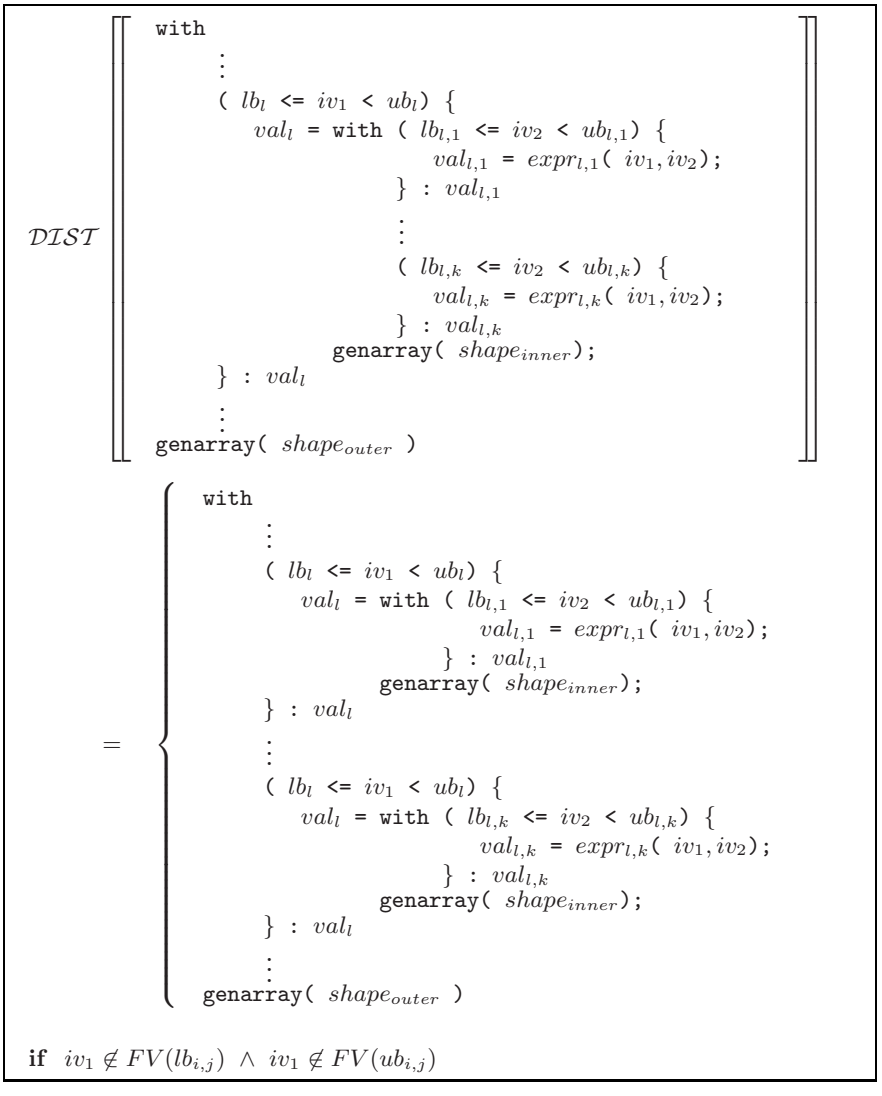

Fig. 7. The distribution phase of multi-generator WITH-LOOP-SCALARIZATION.

5.1 Vectors

The first auxiliary transformation deals with the case of the non-scalar element of a WITH-loop being given in vector notation. As can be seen in Fig. 9, it suffices to replace the reference to the vector with a WITH-loop that contains one part for each of the vector's elements. Hence, each generator describes an index space containing exactly one element. The resulting WITH-loop nesting can then be scalarized using multi-generator WITH-LOOP-SCALARIZATION.

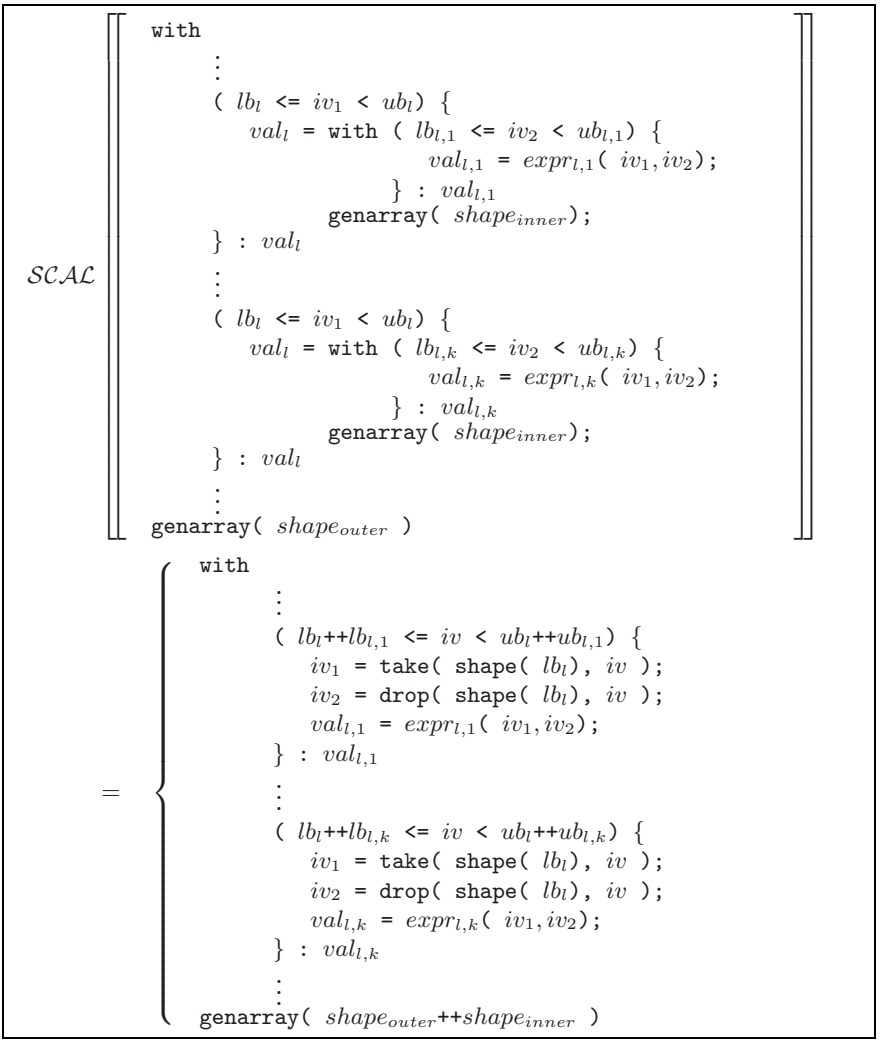

Fig. 8. The scalarization phase of the multi-generator WITH-LOOP-SCALARIZATION.

5.2 Arbitrary Arrays

In general, non-scalar elements of a WITH-loop may not only be given by nested WITH-loops or by vectors, but may also result from WITH-loops defined outside of the WITH-loop or even from function applications. In these cases the arrays' contents are hidden from the view of any local optimization strategy.

As illustrated in Fig. 10, WITH-LOOP-SCALARIZATION can handle this problem by inserting an *identity* WITH-*loop* whose index space covers the entire array, and its operation is just a selection. Hence, the identity WITH-loop replaces a reference to an array with a definition of it and thereby enables WITH-LOOP-SCALARIZATION.

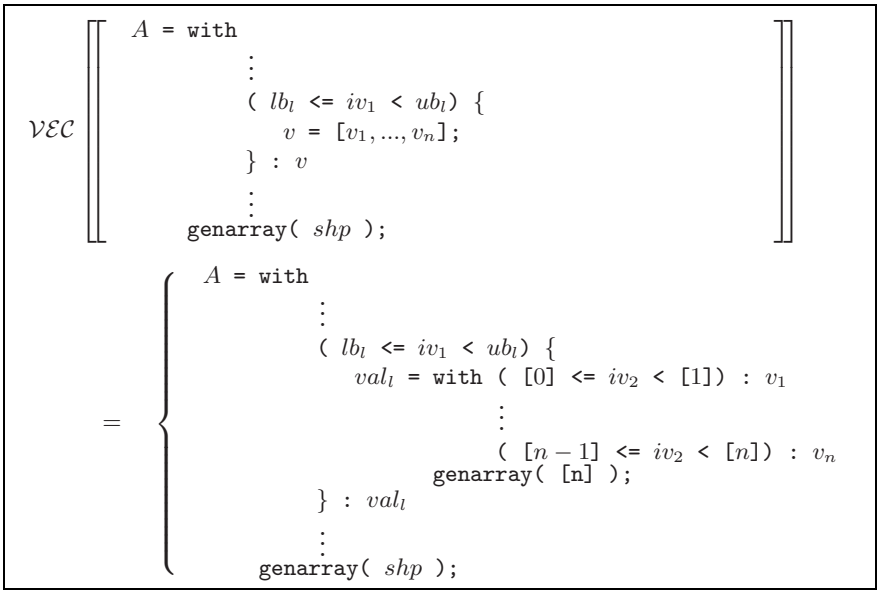

Fig. 9. Auxiliary transformation scheme for arrays given in vector notation.

5.3 Imperfect Nestings

All optimization strategies presented so far can only be applied if WITH-loops form perfect nestings, i.e., there must be no code before inner WITH-loops. To handle imperfect nestings as well, WITH-LOOP-SCALARIZATION is accompanied by the auxiliary transformation scheme shown in Fig. 11. The scheme pushes

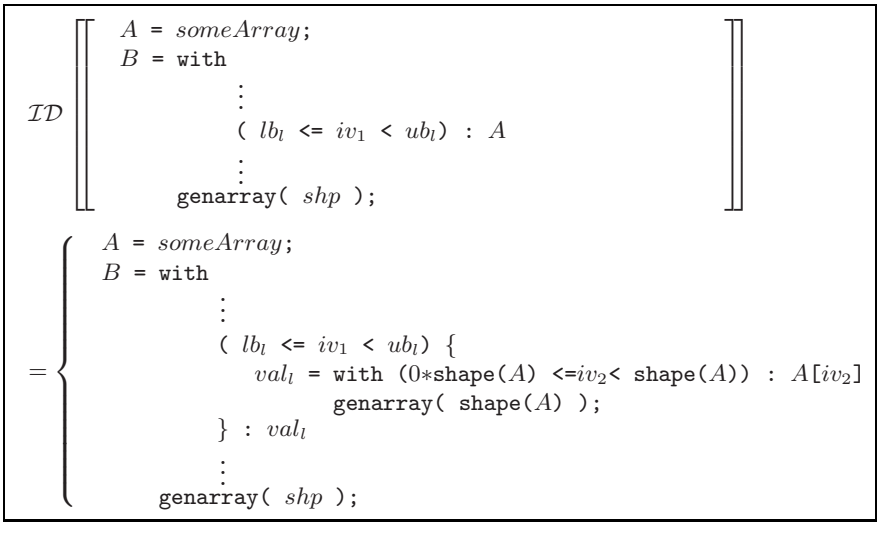

Fig. 10. Inserting an identity WITH-loop enables WITH-LOOP-SCALARIZATION.

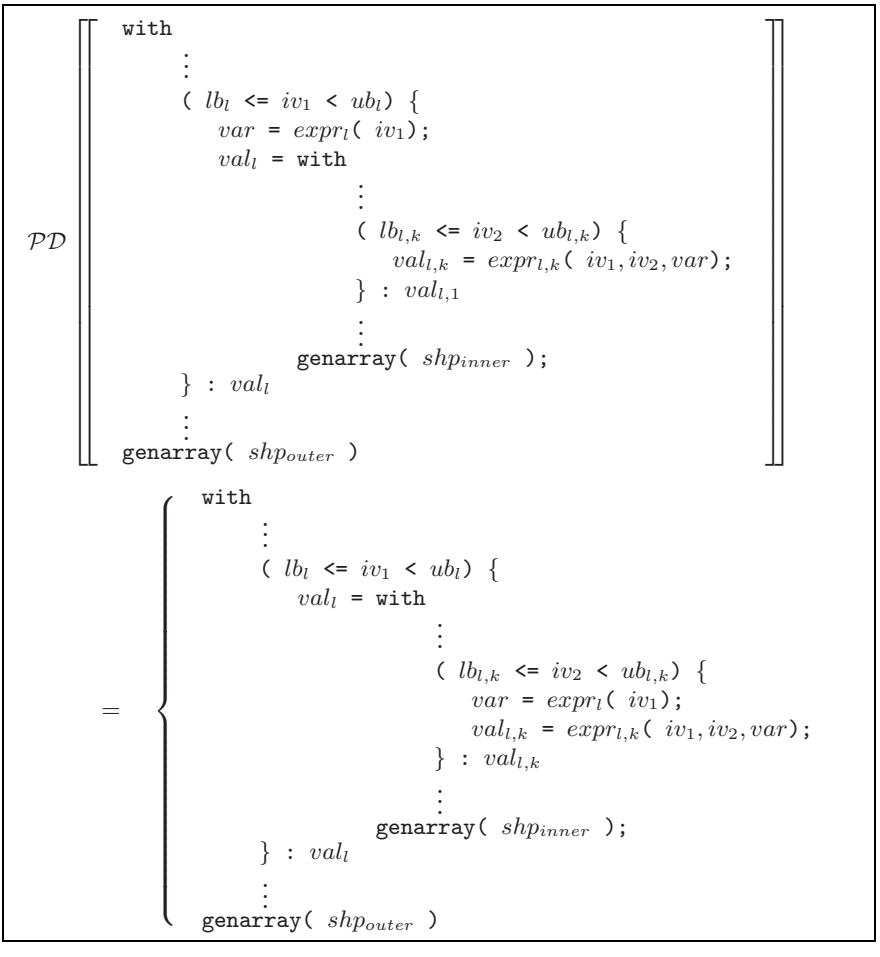

Fig. 11. Auxiliary transformation scheme for imperfect WITH-loop nestings.

down assignments into the inner WITH-loop and this way creates perfect nestings. While this transformation enables WITH-LOOP-SCALARIZATION, a serious drawback is that the moved expressions may be evaluated repeatedly. However, in many situations the performance increase achieved by WITH-LOOP-SCALARIZATION outweighs the cost of redundant code execution. Still, this transformation has speculative character and should be used carefully.

6 Performance Evaluation

This section reports on some experiments evaluating the performance impact of WITH-LOOP-SCALARIZATION. All reported tests have been made on a SUN Ultra 1 workstation using SUN Workshop 5.0 compilers for code generation. Additional experiments on an Intel Pentium III based PC running LINUX and gcc 3.2 confirmed the figures.

6.1 A Customized Benchmark

Fig. 12 shows a SaC micro benchmark tailor-made for estimating the performance impact of WITH-LOOP-SCALARIZATION. It defines a matrix in a row-wise manner, i.e. by creating one intermediate vector per matrix row. The constants `SIZE` and `INNER` allow for specific investigations on the impact of the intermediate vector's size while retaining the overall problem size.

```
A = with ([0] <= iv < [SIZE/INNER])
    {
      B = with ([0] <= jv < [INNER])
          genarray( [INNER], iv[0] + 2 * jv[0]);
    }
    genarray( [SIZE/INNER], B);
```

Fig. 12. Computational kernel of simple WITH-LOOP-SCALARIZATION test.

Fig. 13 shows program runtimes for systematic variations of the intermediate vector's size. Prior to WITH-LOOP-SCALARIZATION this manipulation has an enormous impact on overall performance due to memory management costs, loop overhead, and various cache effects. WITH-LOOP-SCALARIZATION not only accelerates program execution by between 33% and a factor of 5, it also eliminates any dependence between result matrix shape and runtime performance.

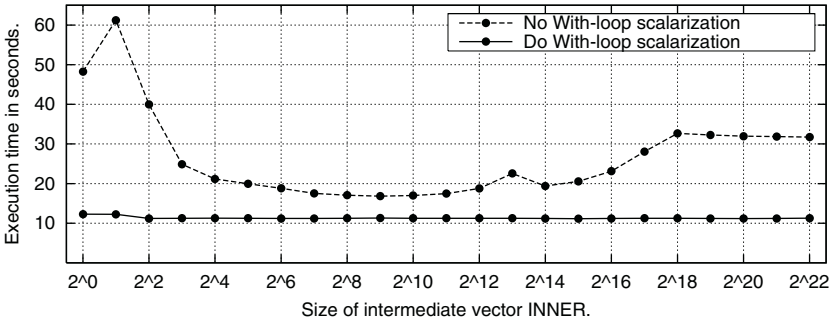

Fig. 13. Performance evaluation of simple WITH-LOOP-SCALARIZATION test.

6.2 Arithmetic on Complex Numbers

Arithmetic on arrays of complex numbers is a particularly prominent example of nested array operations and, hence, a good motivation for WITH-LOOP-SCALARIZATION. Fig. 14 shows an excerpt from the SaC standard library, which defines complex numbers as 2-element vectors and provides overloaded versions of the usual arithmetic operators. In a second step, these overloaded operators a mapped to arrays of any rank and shape. Dots as boundary expressions in

```
typedef double[2] complex;

complex (+) (complex a, complex b)
{
  return( [ a[0] + b[0], a[1] + b[1]]);
}

complex[+] (+) (complex[+] a, complex[+] b)
{
  res = with (. <= iv <= .)
        genarray( shape(a), a[iv] + b[iv]);

  return( res);
}
```

Fig. 14. Complex numbers in SAC.

WITH-loop generators are syntactic sugar referring to the least and the greatest legal index vector of the array to be created. The impact of WITH-LOOP-SCALARIZATION on the runtime performance achieved by this 2-level implementation is shown in Fig. 15; program execution times are given for 100 additions/multiplications of matrices of 1000 by 1000 complex numbers.

WITH-LOOP-SCALARIZATION reduces runtimes by 60% and by 50% for addition and for multiplication, respectively. To provide readers with an impression of absolute performance values achieved by SAC, Fig. 15 also presents corresponding runtimes of equivalent FORTRAN-90 and C programs. Whereas the FORTRAN-90 code benefits from built-in support for complex numbers including built-in arithmetic operators on arrays of complex numbers, straightforward C implementations do not achieve the same performance levels. In the case of multiplication SAC even outperforms C. Due to the functional semantics of SAC the compiler manages to identify multiple references to identical array elements when computing each complex product. Whereas the SAC compiler avoids these superfluous memory accesses, a C compiler must make conservative assumptions, which result in lower performance or require hand optimization.

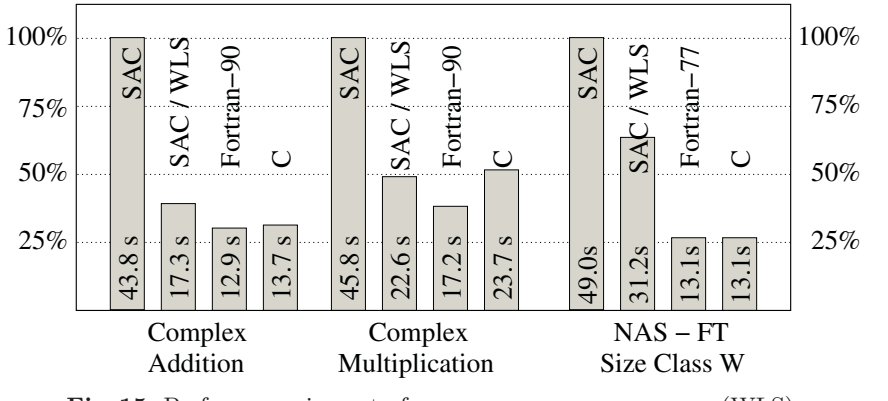

Fig. 15. Performance impact of WITH-LOOP-SCALARIZATION (WLS).

6.3 NAS Benchmark FT

To evaluate the performance impact of WITH-LOOP-SCALARIZATION on a larger application we have chosen the NAS benchmark FT [3]. It implements a solver for a class of partial differential equations by means of repeated 3-dimensional forward and inverse complex fast-Fourier transforms. This benchmark has previously been used for evaluating the suitability of functional languages for numerical computing [18]. A high-level SaC implementation is described in [17].

Fig. 15 shows that WITH-LOOP-SCALARIZATION reduces the total benchmark execution time by as much as one third. Comparing SaC runtimes with highly hand-optimized FORTRAN-77 and C implementations of the benchmark makes clear that WITH-LOOP-SCALARIZATION reduces the performance penalty of high-level programming from a factor of 4 down to less than a factor of 2.4, numbers which are significantly better than those reported in [18].

7 Related Work

In functional languages separate parts of a program are typically glued together using intermediate data structures. Their detection and elimination is crucial for achieving good runtime performance. Optimizations to this effect are generally referred to as *deforestation* or *fusion* techniques [29, 13, 11, 12, 27]. Although being similar in spirit, they completely differ from WITH-LOOP-SCALARIZATION in the concrete setting. Whereas they are based on linked lists, WITH-LOOP-SCALARIZATION acts on multidimensional arrays. Moreover, deforestation connects one producer to one consumer, whereas WITH-LOOP-SCALARIZATION combines the creation of a single array with the creation of many subarrays, one for each element position.

Since main-stream functional programming is based on algebraic data types, research on functional arrays has mostly been focused on achieving reasonable efficiency under less than optimal side conditions discussing such issues as strictness, unboxing, or the aggregate update problem [2, 28, 9]. A variant of deforestation for arrays is described in [8]; it is similar in spirit to WITH-LOOP-FOLDING [24] 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 [22]. However, with a vector-of-vectors representation of multidimensional arrays, SISAL avoids the need for an optimization like WITH-LOOP-SCALARIZATION, but pays with inefficient array accesses in general [23, 25].

An optimization bearing some resemblance to WITH-LOOP-SCALARIZATION is the flattening transformation [6] developed in the context of NESL [5]. 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, but incurs substantial overhead in the case of regular arrays. The flattening operation aims at transforming a multidimensional irregular array into a flat data vector and an auxiliary vector encapsulating all structural information.

In imperative array languages, e.g. FORTRAN-90 or ZPL [10], optimizations like WITH-LOOP-SCALARIZATION have not been pursued because in their 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 contrast, high-level array processing, as in the case of SAC, combines memory layout definition and monolithic initialization in a single conceptual step. As memory layout generally follows the initializing operation, optimizations like WITH-LOOP-SCALARIZATION must ensure that stepwise initializations do not incur costly intermediate data layouts.

8 Conclusion and Future Work

Creation of large numbers of temporary arrays at runtime and, hence, a mediocre runtime performance is the price which typically must be paid for a high-level coding style. To make this way of programming reasonable in areas where performance matters requires powerful optimization schemes that eliminate temporary arrays by meaning-preserving code transformations. This paper discusses WITH-LOOP-SCALARIZATION, a new optimization technique based on WITH-loops. It focuses on intermediate arrays arising from nested array operations. Several experiments show that WITH-LOOP-SCALARIZATION may have a tremendous impact on the runtime performance of compiled code. It turns out to be one key technique to achieve levels of performance competitive to FORTRAN.

WITH-LOOP-SCALARIZATION requires intermediate code to follow a quite specific pattern. To improve its applicability in practice it is accompanied by auxiliary transformation schemes which rewrite intermediate code accordingly. One of these transformations intentionally moves code into the body of a nested WITH-loop. This runs counter traditional optimization strategies, which aim at removing loop-invariant code, and may lead to repeated evaluation of expressions. Experience shows that in many cases other optimizations effectively solve this problem and eliminate repeated evaluations by means beyond the scope of this paper. Nevertheless, future work is needed to gain any guarantees to this effect. Our current approach is to wait until the final code generation phase when multidimensional WITH-loops are eventually transformed into complex nestings of FOR-loops. As soon as binding levels for individual loop variables are once again discriminated, an additional backend loop invariant removal phase would solve the problem.

References

1. P.S. Abrams. An APL Machine. SLAC 114, Stanford Linear Accelerator Center, 1970.
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.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, T.A. Schreiber, R.S. Simon, V. Venkatakrishnam, and S.K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.
4. R. Bernecky. APEX: The APL Parallel Executor. Master's thesis, University of Toronto, Toronto, Canada, 1997.
5. G.E. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1995.
6. G.E. Blelloch and G.W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.
7. T. Budd. *An APL Compiler*. Springer-Verlag, Berlin, Germany, 1988.
8. M.M.T. Chakravarty and G. 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.
9. M.M.T. Chakravarty and G. Keller. An Approach to Fast Arrays in Haskell. In J. Jeuring and S. 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.
10. 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.
11. W.N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. *Journal of Functional Programming*, 4(4):515–550, 1994.
12. A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, Glasgow, Scotland, UK, 1996.
13. A. Gill, J. Launchbury, and S.L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark*, pages 223–232. ACM Press, 1993.
14. C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.
15. C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, selected papers*, volume 1868 of *Lecture Notes in Computer Science*, pages 77–94. Springer-Verlag, Berlin, Germany, 2000.
16. C. Grelck and S.-B. Scholz. Axis Control in SAC. 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 182–198. Springer-Verlag, Berlin, Germany, 2003.
17. C. Grelck and S.-B. Scholz. Towards an Efficient Functional Implementation of the NAS Benchmark FT. In V. Malyshkin, editor, *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia*, volume 2763 of *Lecture Notes in Computer Science*, pages 230–235. Springer-Verlag, Berlin, Germany, 2003.

18. J. Hammes, S. Sur, and W. Böhm. On the Effectiveness of Functional Language Features: NAS Benchmark FT. *Journal of Functional Programming*, 7(1):103–123, 1997.
19. International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.
20. K.E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.
21. M.A. Jenkins and W.H. Jenkins. *The Q'Nial Language and Reference Manual*. Nial Systems Ltd., Ottawa, Canada, 1993.
22. 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, Livermore, California, USA, 1985.
23. R.R. Oldehoeft. Implementing Arrays in SISAL 2.0. In *Proceedings of the 2nd SISAL Users Conference, San Diego, California, USA*, pages 209–222. Lawrence Livermore National Laboratory, 1992.
24. 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.
25. S.-B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, selected papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 216–228. Springer-Verlag, Berlin, Germany, 1999.
26. 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.
27. 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.
28. 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.
29. P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990.