A Case Study: Effects of WITH-Loop-Folding on the NAS Benchmark MG in SAC

Sven-Bodo Scholz

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

Abstract. SAC is a functional C variant with efficient support for high-level array operations. This paper investigates the applicability of a SAC specific optimization technique called WITH-loop-folding to real world applications. As an example program which originates from the Numerical Aerodynamic Simulation (NAS) Program developed at NASA Ames Research Center, the so-called NAS benchmark MG is chosen. It comprises a kernel from the NAS Program which implements 3-dimensional multigrid relaxation.
Several run-time measurements exploit two different benefits of WITH-loop-folding: First, an overall speed-up of about 20% can be observed. Second, a comparison between the run-times of a hand-optimized specification and of APL-like specifications yields identical run-times, although a naive compilation that does not apply WITH-loop-folding leads to slowdowns of more than an order of magnitude. Furthermore, WITH-loop-folding makes a slight variation of the algorithm feasible which substantially simplifies the program specification and requires less memory during execution.
Finally, the optimized run-times are compared against run-times gained from the original FORTRAN program, which shows that for different problem sizes, the code generated from the SAC program does not only reach the execution times of the code generated from the FORTRAN program but even outperforms them by about 10%.

1 Introduction

SAC[21] is a functional programming language aimed at numerical applications. Basically, it can be considered a functional subset of C augmented with an array concept that allows for the specification of array operations that are applicable to arrays of any dimensionality. The central language construct for such high-level array operations is a dimension-invariant form of array comprehensions called WITH-loops. They allow for the definition of basic array operations similar to those available in array processing languages, such as APL[14], NIAL[15], or J[7], which subsequently can be combined to more sophisticated array operations [24]. Assuming a straightforward compilation scheme, this style of programming inherently leads to the creation of many superfluous intermediate array structures.

To avoid this overhead, a high-level optimization called WITH-loop-folding has been proposed in [23].

This paper investigates the effects of WITH-loop-folding on real world applications. For several reasons the NAS multigrid relaxation benchmark MG [3] is chosen as example: first of all, the benchmark is a suitable representative for many numerical applications. Furthermore, since the benchmark is designed for exploiting the capabilities of FORTRAN compilers a reasonable FORTRAN version is commonly available. This allows for an easy inter-language run-time comparison. Another motivation for the choice of the benchmark MG is the fact that the suitability of SAC for the dimension-invariant specification of multigrid relaxation algorithms in general is studied in [22]. Therefore, this paper can focus on the new aspects introduced by WITH-loop-folding.

In particular, the paper addresses the following questions:

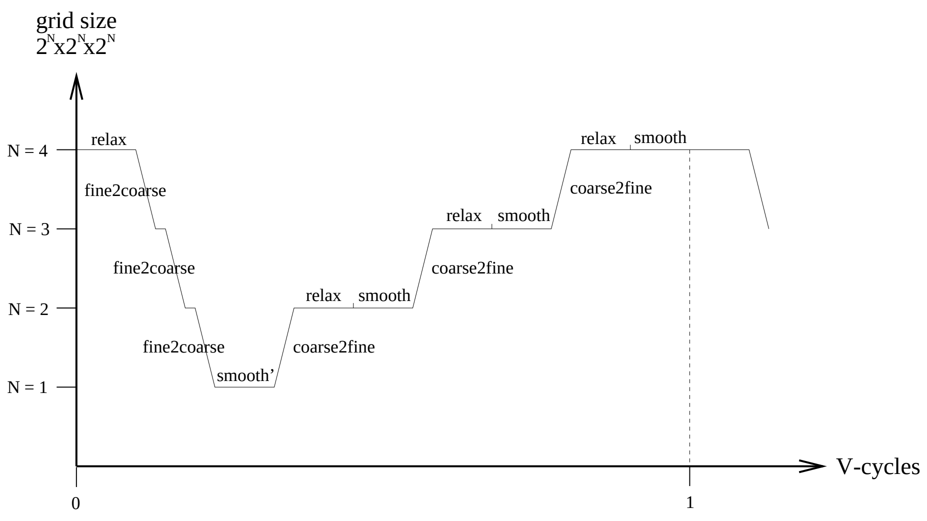
- Does WITH-loop-folding yield an overall run-time improvement? How does that compare against implementation in other languages, such as FORTRAN?
- Does WITH-loop-folding allow for more specificational freedom without the loss of run-time efficiency? If so, does that have an impact on the programming style?

The paper is organized as follows: In the next section a brief overview on the NAS benchmark MG is given. Section 3 investigates the effect of WITH-loop-folding on the overall run-time of the benchmark and compares those figures against a FORTRAN77 and a SISAL implementation. Section 4 compares run-times obtained from different specifications on varying levels of abstraction for one part of the benchmark, the so-called relaxation kernel. After exploiting the effects of WITH-loop-folding for the given multigrid algorithm, Section 5 proposes a slight variant of the algorithm which allows for a far more elegant specification of the given approximation problem and furthermore improves the space consumption of the program. Section 6 puts the work presented in this paper into the context of other research done on the fusion of operations on large data structures. Finally, a conclusion is given in Section 7.

2 An Introduction to the NAS Benchmark MG

The NAS benchmark MG implements the V-cycle multigrid algorithm [4,5] to approximate a solution u of the discrete Poisson problem $\nabla^2 u = v$ on a 3-dimensional grid with periodic boundary conditions. The V-cycle algorithm consists of a recursive nesting of relaxation steps and smoothing steps on grids of different granularity as well as mappings between these grids. The upper part of Fig. 1 for a single V-cycle on a 64x64x64 grid depicts the order in which these transformations are applied (horizontal axis) and which grid sizes are involved (vertical axis).

This sequence of operations can easily be described by means of a recursive function `v_cyc` as specified in the lower part of Fig. 1. The function `mgrid` given



```
double[] mgrid( double[] u,
                double[] v,
                  int iter)
{
  for( i=0; i<iter; i++) {
    r = relax( u, v);
    u  = u + v_cyc( r);
  }
  return(u);
}
```

```
double[] v_cyc( double[] r)
{
  if( coarsest( r)) {
    z = smooth_prime( r);
  }
  else {
    rn = fine2coarse( r);
    zn = v_cyc( rn);
    z  = coarse2fine( zn);
    r  = relax( z, r);
    z  = smooth( z, r);
  }

  return(z);
}
```

Fig. 1. An Outline of the V-cycle.

there as well, initiates `iter` V-cycles on a given 3-dimensional grid `v` and an initial approximation of the solution `u`.[1]

The functions `relax` and `smooth` merely re-compute the elements of an argument grid as weighted sums of their neighbor elements. Since the benchmark requires periodic boundary conditions the missing neighbors of border elements have to be taken from the "opposite side" of the grid. Fig. 2 depicts the situation in the 1-dimensional case. While all inner elements are re-computed using their direct neighbors, each of the two border elements has to be computed differently. Carrying over this principle to problems of higher dimensionalities, the sets of elements which require special treatment increase. In the 2-dimensional

[1] Note, that in real world applications the number of V-cycles applied depends on the convergence properties of the problem.

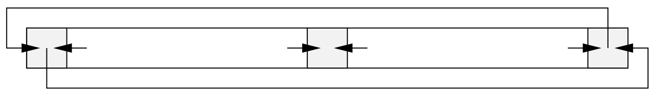


Fig. 2. Relaxation on 1-Dimensional Arrays.

case 9 different operations have to be performed, in case of three dimensions 27 operations are required.

To avoid such complicated specifications, the FORTRAN program given in the benchmark (see ⟨http://www.nas.nasa.gov/NAS/NPB/⟩) represents the grids by arrays which have 2 more elements in each dimension. These hold copies of the values of the missing neighbor fields. Fig. 3 depicts a relaxation step for the 1-dimensional case using such extra elements. Since the border elements of

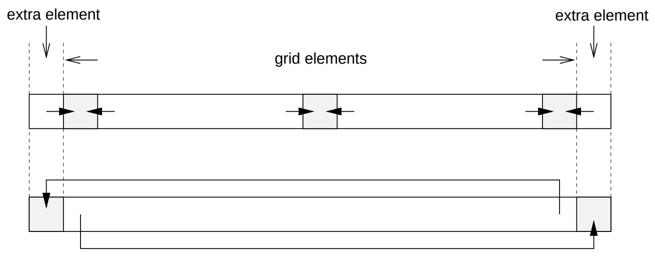


Fig. 3. Relaxation on 1-Dimensional Arrays Using Border Elements.

the arrays solely provide missing neighbor elements, the relaxation step itself becomes a unique operation for all inner elements of the array as shown in the upper part of Fig. 3. Subsequently, the border elements of the array have to be updated accordingly so that they hold the correct values from the "opposite side" of the grid (lower part of Fig. 3). For the 1-dimensional case as depicted here, this extended grid representation does not offer any benefits; still three different operations are required: the unique relaxation step, the updating of the leftmost element, and the updating of the rightmost element. However, for problems of higher dimensionalities this grid representation is advantageous since the number of different operations required does not grow exponentially but linearly, i.e., for the 2-dimensional case 5 operations and for the 3-dimensional case 7 operations are needed.

A dimension-invariant realization of `relax` and `smooth` based on such extended grids in SAC can be deduced straightforwardly. Let `A` and `S` be program constants that hold the arrays of weights needed for the computation of weighted sums of neighbor elements in `relax` and `smooth`, respectively. Then these functions can be specified as:

```
double[] relax( double[] u, double[] v)
{
  r = with( 0*shape(u)+1 <= x <= shape(u)-2)
      modarray( u, x, v[x] - weighted_sum( u, x, A));
  r = setup_periodic_border(r);
  return(r);
}


double[] smooth( double[] z, double[] r)
{
  z = with( 0*shape(r)+1 <= x <= shape(r)-2)
      modarray( r, x, z[x] + weighted_sum( r, x, S));
  z = setup_periodic_border( z);
  return( z);
}

inline double weighted_sum( double[] u, int[] x, double[] w)
{
  res = with( 0*shape(w) <= dx < shape(w) )
        fold( +, u[x+dx-1] * w[dx]);
  return(res);
}
```

where `setup_periodic_border` for each dimension copies those elements into the border elements that are needed for the next relaxation/smoothing step.

Re-using `weighted_sum` the mapping from fine grids to coarse grids can be specified in a similar way:

```
double[] fine2coarse( double[] r)
{
  rn = with( 0*shape(r)+1 <= x<= shape(r) / 2 -1)
       genarray( shape(r) / 2 + 1, weighted_sum( r, 2*x, P));
  rn = setup_periodic_border(rn);
  return(rn);
}
```

The specification of mappings from coarse to fine grids is more complicated. As explained in detail in [22], a dimension-invariant specification of that operation requires two consecutive WITH-loops:

```
double[] coarse2fine( double[] rn)
{
  r = with( 0*shape(rn) <= iv <= 2*shape(rn)-3  step 0*shape(rn)+2 )
      genarray( 2*shape(rn)-2, rn[iv/2] );

  r = with( 0*shape(r) < iv < shape(r)-1 ) {
        val = relaxkernel( r, iv, Q);
      } modarray( r, iv, val);

  r = setup_periodic_border(r);
  return(r);
}
```

This two-step process for 1-dimensional grids is depicted in Fig. 4. In the first step, the elements from the coarse grid are copied into every other position of a new array of double the size. The elements in between are initialized with zeros. Subsequently, a relaxation step is performed whose array of weights determines the interpolation of the values initialized with zero. For the 1-dimensional case, `[ 0.5, 1, 0.5]` serves as array of weights. Although in principle all elements are computed by the same scheme, the placement of zeros forces several values to be neglected as indicated by the dotted lines in Fig. 4. As a consequence, the elements of the resulting finer grid are computed from the elements of the coarser grid basically by two different operations: all elements with even indices (starting by index [0]) are simply copied from the coarser grid, whereas the other elements are averages of two adjacent elements of the coarser grid. Note here, that for problems of dimensionality n a choice of appropriate arrays of weights implicitly generates the required 2^n different operations.

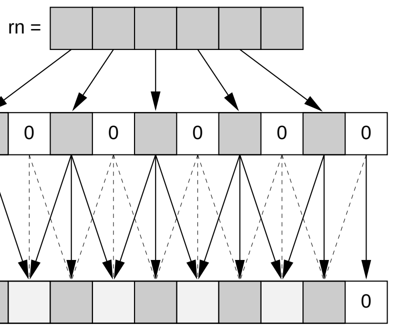


Fig. 4. Coarse-to-Fine-Mapping on 1-Dimensional Arrays.

3 Applying WITH-Loop-Folding to Mgrid

In this section the effect of WITH-loop-folding to the SAC specification of the benchmark outlined in the previous section is examined. Furthermore, the run-times are compared against those obtained from running compiled FORTRAN and SISAL solutions.

All those measurements are done in the same setting as in [22], i.e., a SUN ULTRASPARC-170 with 192MB of main memory serves as hardware platform. The FORTRAN program is compiled by the SUN FORTRAN compiler f77 version 4.2 which generates native code directly. The SISAL program is compiled by OSC version 13.0.2 which generates C code that subsequently is compiled into native code by GCC version 2.7.2.1. The optimization flags used are "-O4" for the FORTRAN compiler and "-O -nobounds -CC=gcc -cc=-O3 -seq" for the SISAL compiler.

The SAC program is compiled by the new SAC2C compiler version 0.7 which in comparison to the version used in [22] does not only include WITH-loop-folding but has an improved "back-end" for the generation of C code from WITH-loops. GCC version 2.7.2.1 with optimization level 3 is used again as the compiler for the C-code generated by SAC2c.

Fig. 5 shows the run-times relative to the time needed by the compiled FORTRAN program for three different problem sizes[2]. The problem-sizes investigated

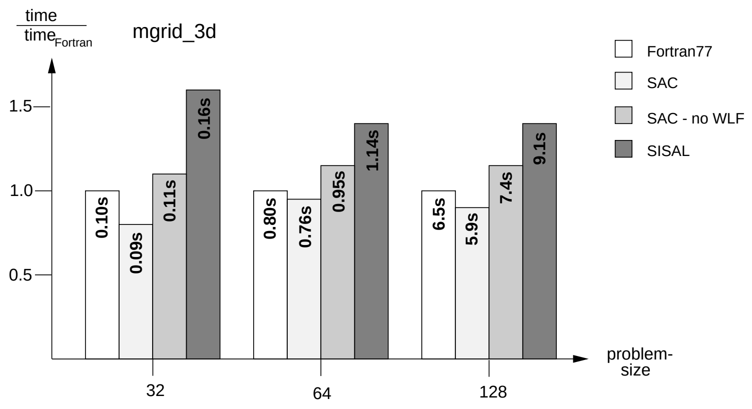


Fig. 5. Run-times for 3-Dimensional Multigrid Relaxation.

are 3-dimensional grids with 32, 64, and 128 elements per axis.

The SISAL implementation turns out to be the slowest solution. It runs about 40% slower than the FORTRAN program and about 50% slower than the optimized SAC version. While the SAC program compiled without WITH-loop-folding (SAC -noWLF) is about 10% slower than the FORTRAN solution, the version gen-

[2] The absolute run-times for one V-cycle are denoted inside the bars.

erated from the Sac program using WITH-loop-folding (Sac) is about 10% faster than the FORTRAN program.

The reason for the speed-up of about 20% gained by WITH-loop-folding can be attributed to the mapping from coarse to fine grids `coarse2fine`. As explained in the previous section, it has to be specified as a two-step process in order to allow for a dimension-invariant program. Together with a specialization of `coarse2fine` to 3-dimensional arguments of specific shapes as done by the type inference system of Sac, WITH-loop-folding converts this operation into a direct computation of fine grids from coarse grids.

Fig. 6 depicts the effect of WITH-loop-folding for 1-dimensional grids. Whereas

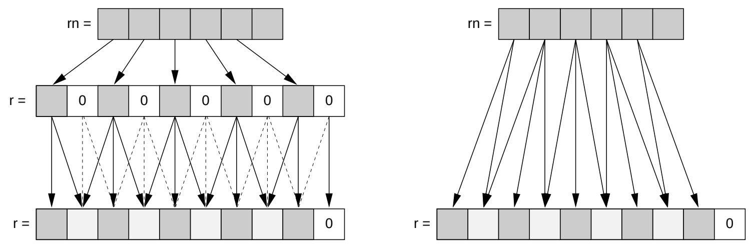


Fig. 6. Applying WITH-Loop-Folding to `coarse2fine`.

the version using two WITH-loops (left part of Fig. 6) subsequently applies two unique operations on the elements of the array representing the coarse grid, the version obtained by WITH-loop-folding (right part of Fig. 6) directly computes the resulting fine grid from the coarse grid by using two different operations: elements at odd index positions (marked in light grey in Fig. 6) are computed as average of two adjacent elements of the coarse grid, and elements at even index positions (marked in dark grey) are simply copied from the coarse grid. As a consequence, the intermediate data structure as well as any superfluous computation (dotted lines in Fig. 6) can be avoided resulting in a speed-up of about 20%.

4 WITH-Loop-Folding and Specificational Freedom

One of the aims of WITH-loop-folding is to provide a uniform optimization scheme which does not only allow for more modular specifications of array operations without substantial loss of run-time efficiency, but encourages the programmer to do so. This section investigates that effect in the context of the benchmark. Instead of re-coding the whole benchmark, only a part of it, namely the function `relax` (cf. Section 2), is examined more closely. This on the one side allows for a smaller scope during testing and on the other side carries over to the major part of the benchmark since most kernel routines of the benchmark (`smooth` and `fine2coarse`) are only slight variations of `relax`.

The similarity of these routines leads to the first variant of `relax` which allows for more code re-use. The central idea is to abstract the WITH-loop in the body of `relax` into a new function `relax_kernel`. As a consequence, the difference of `v` and a weighted sum of some elements in `u` can be specified as an array operation rather than element-wise:

```
double[] relax( double[] u, double[] v)
{
  res= v - relax_kernel( A, u);
  res = setup_periodic_border( res);
  return( res);
}


inline double[] relax_kernel( double[] w, double[] u)
{
  res = with( 0*shape(u)+1 <= x <= shape(u)-2) {
          val = weighted_sum( u, x, w));
        } modarray( u, x, val);
  return( res);
}
```

A more sophisticated variant of `relax` is based on the idea of replacing the element-wise specification of the re-computation of inner elements by operations on entire arrays. This turns the explicit selection and summation of neighbor elements in the body of `weighted_sum` into rotations and additions of entire arrays, respectively. Thus the function `weighted_sum` is not needed anymore and `relax_kernel` can be specified as:

```
inline double[] relax_kernel( double[] w, double[] u)
{
  res = with( 0*shape(w) <= dx < shape(w) )
        fold( +, rotate_vec( dx-1, u) * w[dx]);
  return(res);
}
```

where the function `rotate_vec` rotates the array `u` along all axes according to the rotation vector given by `dx-1`. In turn, `rotate_vec` can be defined in terms of `rotate` which is defined in the standard array library and rotates a given array `a` by `num` elements along a pre-specified axis `dimen`:

```
inline double[] rotate_vec( int[] rv, double[] a)
{
  for( i=0; i<shape(rv)[0]; i=i+1)
    a = rotate( i, rv[[i]], a);
  return(a);
}
```

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

Fig. 7 compares the run-times for the three different versions of `relax` introduced so far. The problem size examined here are 15 relaxation steps on a 2-dimensional array with 1000 elements per axis. All run-times are measured on the same architecture as the previous examples. Whereas the left column shows

	WLF	noWLF
Direct specification of `relax`	4.9s	4.9s
`relax` using `relax_kernel`	4.9s	5.5s
`relax` using `rotate`	4.9s	77.1s

Fig. 7. Run-times With and Without WITH-Loop-Folding.

the run-times of the three versions using WITH-loop-folding, the right column contains those obtained without. The direct solution as explained in Section 2 is not affected by WITH-loop-folding since that version does not contain any consecutive WITH-loops at all. In the second version, the subtraction operation without applying WITH-loop-folding leads to a single superfluous array which causes a slowdown of about 10%. In contrast, for the high-level specification which completely forgos any explicit indexing a non-folding compilation leads to a slowdown of about 1500%!

Despite these slowdowns introduced by a naive compilation, the run-times for the optimized versions are identical. Analyzing the generated C-code yields that the WITH-loops eventually generated by WITH-loop-folding in all three cases are almost identical.

5 A Variant Without Borders

The results of the previous section show that a specification based on the summation of rotated arrays does not lead to any performance losses in terms of run-time. The main difference of that solution in comparison to the others considered is that the border elements are re-computed as well. These computations are superfluous since `setup_periodic_border` copies these elements from inner elements of the array anyway. Having a closer look at the operations performed on the border elements yields that they are computed as weighted sums of their "neighbor elements" as well. Since `rotate` shifts the elements of an array cyclicly, missing neighbors implicitly are taken from the "opposite side" of the array. In fact, this algorithm performs an operation on the complete array that satisfies the original problem specification (cf. Fig. 2 in Section 2).

Therefore, the data layout for the arrays holding the grid elements can be simplified throughout the entire program by cutting off the border elements. As a result, the function `setup_periodic_border` becomes redundant and most other functions can be further simplified, for example, `relax` can be specified as:

```
double[] relax( double[] u, double[] v)
{
  return( v - relax_kernel( A, u));
}
```

This exactly resembles the mathematical specification given in [3]. Besides the specificational advantages of that solution it decreases the overall memory consumption and thus improves the overall performance.

6 Related Work

The effects of fusion techniques have been studied in various contexts.

In the area of functional programming, several variants of so-called *deforestation* have been proposed and examined [25,9,11,19]. Since these techniques are tailor-made for the elimination of temporary lists, they implicitly assume that the length of the list(s) involved is statically unknown and that each function will be applied on all elements of the list(s).

Since these assumptions for array computations in general do not hold, other approaches in the context of functional programming have been proposed which are based on the idea of representing arrays as functions from indices to values [6,13]. As a consequence, array operations can be folded by simply β-reducing them. Although this approach conceptually is very promising it still lacks a proof that an efficient implementation is possible [12].

Closer related to the work in this paper are the evaluations of fusion techniques in the context of high performance array languages, such as SISAL [18], FORTRAN90 [1], HPF [10], or ZPL [17]. Whereas earlier approaches in that field are based on traditional loop optimizations [27,8,2,26] which are applied to scalarized versions of the high-level operations, more recent publications [20,16] point out the importance of fusion operations that are applied to high-level operations. However, specificational benefits comparable to those presented in this paper are not possible in these languages, since they do not allow for the specification of dimension-invariant array operations, e.g. `rotate` or `rotate_vec` from Section 4.

7 Conclusion

This paper was to investigate, by means of a case study, the effects of WITH-loop-folding on a program kernel which originates from a real world application. The example chosen is the multigrid relaxation kernel from the NAS benchmarks. Applying WITH-loop-folding to a dimension-invariant SAC specification derived from the FORTRAN program given in the benchmark does not only yield the same run-times, it even outperforms them by about 10%. These improvements can be tracked down to some redundancies in the mapping from coarse to fine grids caused by the dimension-independent specification.

Besides these overall run-time benefits a gain in specificational freedom without any loss of run-time efficiency can be observed for a central part of the multigrid benchmark, the relaxation kernel. This allows for a variation of the algorithm based on rotations of entire arrays with a couple of advantages: the algorithm is more concise; it resembles the mathematical specification more directly; it is based entirely on standard array operations and thus encourages code re-use; and it requires less memory at run-time.

References

1. J.C. Adams, W.S. Brainerd, J.T. Martin, et al. *Fortran90 Handbook - Complete ANSI/ISO Reference.* McGraw-Hill, 1992. ISBN 0-07-000406-4.
2. D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
3. D. Bailey, E. Barszcz, J. Barton, et al. The NAS Parallel Benchmarks. RNR 94-007, NASA Ames Research Center, 1994.
4. D. Braess. *Finite Elemente.* Springer, 1996. ISBN 3-540-61905-4.
5. A. Brandt. Multigrid Methods: 1984 Guide. Dept of applied mathematics, The Weizmann Institute of Science, Rehovot/Israel, 1984.
6. T. Budd. Composition and Compilation in Functional Programming Languages. Technical Report 88-60-14, Oregon State University, 1988.
7. C. Burke. *J and APL.* Iverson Software Inc., Toronto, Canada, 1996.
8. D.C. Cann. *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory, LLNL, Livermore California, 1993. part of the SISAL distribution.
9. W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. *Journal of Functional Programming*, 4(4):515–550, 1994.
10. High Performance Fortran Forum. *High Performance Fortran language specification V1.1*, 1994.
11. A. Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Glasgow University, 1996.
12. J. Halen, P. Hammarlund, and B. Lisper. An Eperimental Implementation of a Highly Abstract Model of Data Parallel Programming. TRITA-IT 97:2, Dept. of Teleinformatics, KTH, Stockholm, 1997.
13. P. Hammarlund and B. Lisper. On the Relation between Functional and Data Parallel Programming Languages. In *FPCA '93*, pages 210–222. ACM Press, 1993.
14. K.E. Iverson. *A Programming Language.* Wiley, New York, 1962.

15. M.A. Jenkins and W.H. Jenkins. *The Q'Nial Language and Reference Manuals.* Nial Systems Ltd., Ottawa, Canada, 1993.
16. E.C. Lewis, C. Lin, and L. Snyder. The Implementation and Evaluation of Fusion and Contraction in Array Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.* ACM, 1998.
17. C. Lin. ZPL Language Reference Manual. UW-CSE-TR 94-10-06, University of Washington, 1996.
18. J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1985.
19. L. Nemeth and S. Peyton Jones. A Design for Warm Fusion. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages*, pages 381–393. University College, London, 1998.
20. G. Roth and K. Kennedy. Loop Fusion in High Performance Fortran. CRPC TR98745, Rice University, Houston, Texas, 1998.
21. S.-B. Scholz. **S***ingle* **A***ssignment* **C** – *Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen.* PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.
22. S.-B. Scholz. On Programming Scientific Applications in SAC - A Functional Language Extended by a Subsystem for High-Level Array Operations. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, volume 1268 of *LNCS*, pages 85–104. Springer, 1997.
23. 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*, pages 225–242. University of St. Andrews, 1997.
24. S.-B. Scholz. On Defining Application-Specific High-Level Operations by Means of Shape-Invariant Programming Facilities. In S. Picchi and M. Micocci, editors, *Proceedings of the Array Processing Language Conference 98*, pages 40–45. ACM-SIGAPL, 1998.
25. P.L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.
26. M.J. Wolfe. *High-Performance Compilers for Parallel Computing.* Addison-Wesley, 1995. ISBN 0-8053-2730-4.
27. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.