On Defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities

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

Abstract

Most of the existing high-level array processing languages support a fixed set of pre-defined array operations and a few higher-order functions for constructing new array operations from existing ones. In this paper, we discuss a more general approach made feasible by Sac (for **S**ingle **A**ssignement **C**), a functional variant of C.

Sac provides a meta-level language construct called with-loop which may be considered a sophisticated variant of the forall-loops in HPF or of array comprehensions in functional languages. It allows for the element-wise specification of high-level operations on arrays of any dimensionality: any set of high-level array operations can be specified by means of with-loops and be made available in a library. This does not only improve the flexibility of specifications, but also simplifies the compilation process.

By means of a few examples it is shown that the high-level operations that are typically available in array processing languages such as Apl or Fortran90 can be easily specified as with-loops in Sac. Furthermore, we briefly outline the most important optimization techniques used in the current Sac compiler for achieving efficiently executable code.

The paper finally presents a performance comparison between a high-level specification for the multigrid relaxation kernel of the NAS benchmarks in Sac on the one hand and low-level specifications in Sisal and in Fortran77 on the other hand. It shows that the Sac implementation, despite its higher level of abstraction, is competitive with the other two both in terms of program runtimes and memory consumption.

Keywords: High-Level Array Operations, Meta-Level Programming, Shape-Invariant Programming, Compilation, Performance Comparison.

1 Introduction

One of the key features of array processing languages such as Apl[Ive62] or Fortran90[ABM$^+$92] is that they provide a fixed set of high-level array operations that are applicable to arrays of any dimensionality. Although these pre-defined operations can be implemented by efficiently executable dimension-specific target code, this approach has some shortcomings.

Some numerical application problems require array operations which cannot be easily expressed by means of these primitives, often requiring rather complex compositions of them which are hard to comprehend. To express these operations more elegantly a feature is needed which allows to define new Apl-like primitives on an element-wise basis.

Moreover, the compilation of function compositions into efficiently executable code is far from straightforward as too many intermediate arrays may be created which in a low-level scalar-oriented specification could be avoided. Optimizations to this effect face different difficulties, depending on the level of abstraction on which they are being done:

Optimizations that operate more or less directly on source programs, e.g. *phrase recognition* or *array coordinate mapping* [Bro85, Bud88, Mul88] require many different optimization rules to be implemented. However, only a few of these rules can be implemented as *algebraic transformations* on the source level. Most of them require an implicit transformation into a lower level of abstraction.

Another way to avoid intermediate arrays is to compile the high-level array operations into programs (merely dimension-specific nestings of loops) of more basic languages, e.g. Fortran77 or C, and subsequently apply more general optimizations, e.g. *loop fusion*, *forward substitution*, or *scalar replacement* [PW86, ZC91, Can93, BGS94, Wol95]. Unfortunately, at this rather low level some of the high-level information about the structure of nestings of these loops and its interdependencies are not available anymore, which restricts the applicability of these optimizations.

A more promising approach is taken in the APEX compiler [Ber97b, Ber97a], where Sisal [MSA$^+$85] is chosen as intermediate language for the compilation of Apl programs. The advantage of this approach is that the for-loops in Sisal allow to preserve more information about the structure of the high-level operations. Furthermore, the Sisal compiler osc [Can93] offers excellent support for loop optimizations and compilation to shared-memory multiprocessor systems.

Unfortunately, the choice of Sisal as intermediate language has some drawbacks as well. In Sisal, all functions / array operations require the dimensionalities of the arguments to be specified explicitly. As a consequence, APEX has to be restricted to those Apl programs where the dimensionalities of all arrays can be inferred statically. Moreover, the dimension-dependence of SISAL a priori precludes language extensions that would allow to define new primitive array operations on an element-wise basis.

As a possible remedy for these problems we propose using SAC [Sch96, Sch97a] either as intermediate language or even as source language. SAC is a functional C-variant which truly supports dimension-invariant specifications of array operations. Besides providing a fixed set of primitives, SAC offers meta-level language constructs which on the one hand are versatile enough to specify arbitrary high-level array operations and on the other hand are restrictive enough to allow for the compilation into efficiently executable code. These constructs, called WITH-loops, may be considered sophisticated variants of the FOR-loops in SISAL, FORALL-loops in HPF, or of array comprehensions in functional languages. They do not only allow for the specification of operations on arbitrary subranges of arrays but may also be specified in a form that is completely invariant against the shapes of the arrays they are applied to.

This paper discusses the pros and cons of using WITH-loops as meta-level language construct for defining high-level array operations similar to those available in APL. After giving a brief introduction to SAC in Section 2 the applicability of WITH-loops for defining APL-like primitives is investigated in Section 3. For a few examples it is demonstrated how such operations can be specified and how using WITH-loops for the specification of such primitives quite naturally leads to the introduction of yet other primitives and thus to a more modular programming style. Section 4 briefly sketches the most important optimizations of the current SAC compiler, and Section 5 compares some performance figures for a high-level SAC specification of the mgrid kernel of the NAS benchmarks [BBB+94] with those of equivalent FORTRAN and SISAL implementations.

2 SAC - a Short Introduction

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

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

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

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

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

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

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

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

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

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

A few high-level array operations are pre-defined in SAC, e.g. `dim` and `shape` for inspecting the dimensionality and the shape of an array, and a function `psi` for array element / subarray selection. Note here, that for reasons of convenience `psi( array, index_vector)` can be abbreviated as `array[ index_vector]`. For a formal definition of these array operations see [Sch96, Sch97b].

Any other high-level array operation has to be defined by means of so-called WITH-loops. They are similar to the FOR-loops in SISAL, but allow for the specification of index ranges within arrays of unknown dimensionality.

The syntax of WITH-loops is defined in Fig. 1. Basically,

$$\begin{array}{lcl}
WithExpr & \Rightarrow & \texttt{with} \;\; (\; Generator \; \big[Filter \; \big] \;) \;\; Operation \\
Generator & \Rightarrow & Expr \;\; \texttt{<=} \;\; Identifier \;\; \texttt{<=} \;\; Expr \\
Filter & \Rightarrow & \big[\; \texttt{step} \;\; Expr \; \big[\; \texttt{width} \;\; Expr \; \big] \big] \\
Operation & \Rightarrow & \big[\; \{ \;\; LocalDeclarations \;\; \} \; \big] ConExpr \\
ConExpr & \Rightarrow & \texttt{genarray} \;\; (\;\; Expr \;\; , \;\; Expr \;\;) \\
 & | & \texttt{modarray} \;\; (\;\; Expr \;\; , \;\; Expr \;\; , \;\; Expr \;\;) \\
 & | & \texttt{fold} \;\; (\; FoldFun \; , \;\; Expr \;\; , \;\; Expr \;\;) \\
FoldFun & \Rightarrow & + \;\; | \;\; * \;\; | \; Identifier
\end{array}$$

Figure 1: WITH-loops in SAC.

they consist of three parts: a generator part, a filter part, and an operation part. The generator part defines lower and upper bounds for a set of index vectors and an 'index variable' (*Identifier* in the second rule of Fig. 1) which represents an element of this set. The optional filter part may restrict the set of index vectors in the following way: let a, b, s, and w denote expressions that evaluate to vectors of the length n. Then

$$(\; a \; \texttt{<= i\_vec <=} \; b \; \texttt{step} \; s \; \texttt{width} \; w \;)$$

denotes the following set of index vectors:

$$\{i\_vec \mid \forall_{i \in \{0,...,n-1\}} : \quad a_i \le i\_vec_i \le b_i \quad \wedge \quad (i\_vec_i - a_i) \text{ modulo } s_i < w_i\}.$$

The operation part specifies the operation to be performed on each element of the index vector set defined by the generator / filter part. Three different kinds of operation parts for the generation of arrays are available in SAC (see *ConExpr* in Fig. 1). Their functionality is defined as follows:

Let *shp* and *idx* denote SAC-expressions that evaluate to vectors, let *array* denote a SAC-expression that evaluates to an array, and let *expr* denote an arbitrary SAC-expression. Furthermore, let *fold_op* be the name of a binary commutative and associative function (*FoldFun* in Fig. 1) with neutral element *neutral*. Then

- genarray(*shp*, *expr*) generates an array of shape *shp* whose elements are the values of *expr* for all index vectors from the specified set, and 0 otherwise;

- modarray(*array*, *idx*, *expr*) returns an array of shape shape(*array*) whose elements are the values of *expr* for all index vectors from the specified set, and the values of *array*[*idx*] at all other index positions.

- fold(*fold_op*, *neutral*, *expr*) sets out with the neutral element *neutral* to fold with the binary operation *fold_op* the values of *expr* found in all index positions from the specified set. It is the responsibility of the programmer to make sure that the function *fold_op* is commutative and associative in order to guarantee deterministic results.

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

3 Defining dimension-invariant array operations in SAC

In this section the applicability of the WITH-loop constructs in SAC for defining APL-like array operations is investigated. As a first example, we try to define two functions take and drop which realize the ↑ and ↓ operators of APL, respectively.

take expects two arguments: a vector new_shape which specifies the size of the sub-array to be taken and an array A from which the elements are to be extracted of. The resulting array res is of shape new_shape and its elements are defined as follows: for each legal index vector i_vec of res we have res[i_vec] = A[i_vec]. This specification can directly be translated into a SAC function:

```
inline double[] take( int[] new_shp, double[] A)
{
  res = with ( . <= i_vec <= . )
        genarray( new_shp, A[i_vec]);
  return(res);
}
```

Note here, that double[] and int[] are type declarations in SAC that denote array types of unknown shape with basic type double and int, respectively. Furthermore, in the generator part of the WITH-loop a shorthand notation, the dot symbol, is used for the specification of the upper and lower bound of the index range to be processed. It denotes the minimum and maximum legal index vector of the resulting array.

A function drop can be defined similarly. However, in contrast to take, the elements of the resulting array are taken from the argument array by adding to the index vector i_vec an offset given as the first argument of drop:

```
inline double[] drop( int[] offset, double[] A)
{
  new_shp = shape(A)-offset;
  res = with ( . <= i_vec <= . )
        genarray( new_shp, A[i_vec+offset]);
  return(res);
}
```

The computation of the shape of the resulting array (new_shp) as well as the access to A require + and - to be applicable to arrays as well. This can easily be achieved by definitions of the following kind:

```
inline double[] +( double[] A, double[] B)
{
  res = with ( . <= i_vec <= . )
        modarray( A, A[i_vec] + B[i_vec]);
  return(res);
}
```

As a more sophisticated example consider the function rotate that rotates the elements of an array by a pre-specified number num of elements in a pre-specified dimension dim:

```
inline double[] rotate( int dim, int num, double[] A)
{
  max_rotate = shape(A)[dim];
  num = num % max_rotate;
  if( num < 0)
    num = num + max_rotate;

  offset = modarray( 0*shape(A), [dim], num);
  slice_shp = modarray( shape(A), [dim], num);
  res = with ( offset <= i_vec <= shape(A)-1)
        modarray( A, i_vec, A[i_vec-offset]);
  res = with ( 0*slice_shp <= i_vec <= slice_shp-1)
        modarray( res, i_vec, A[shape(A)-slice_shp+i_vec]);
  return(res);
}
```

After normalizing num to a positive integer between 0 and the maximum number of elements of A in dimension dim the computation of the rotated array is done in several steps. In order to avoid modulo operations for the computation of each element of the resulting array B, two different offsets to the index vectors i_vec are computed: one for those elements which have to be copied from positions with lower index vectors to positions with higher index vectors (offset) and another one (shape(A)-slice_shp) for those elements that have to be copied from positions with higher index vectors to positions with lower index vectors. Besides the introduction of an overloaded version of * for scalars and arrays, the computation of the vectors offset and slice_shp requires a function modarray which allows single elements of an array to be changed. As for the overloaded version of +, the implementation of these functions can be specified straight-forwardly by means of WITH-loops and therefore are omited here.

After offset and slice_shp are computed the rotation of array elements can be expressed by two consecutive WITH-loops, each of which copies a part of the array elements.

All primitive array operations that are available in languages such as APL or FORTRAN90 can be defined in SAC in a similar way. Moreover, defining array operations by means of WITH-loops may encourage programmers to introduce further abstractions (primitives) whenever deemed necessary. This may not only increase the readability of programs but may also lead to a more modular programming style and thus increase program re-usability.

4 Compiling WITH-loops into Efficiently Executable Code

The freedom gained by the WITH-loops with respect to specifying high-level array operations, which in fact introduces a meta language level, must of course be complemented by a much more generic compilation process than that of predefined array operations. This section outlines the most important optimizations used in the current SAC compiler to generate C code from such specifications that compares well with hand-optimized versions.

To examplify this optimization process we take the definition of the rotate function from the previous section as a running example and assume an application of the form rotate(1, n, A) where n is meant to refer to an integer that is not known at compile time and `A` refers to an array of shape `[100, 100]`.

As one of the first optimizations applied in the SAC compiler, all specifications of dimension-invariant array operations are specialized to dimension-specific versions as far as possible. For this purpose, the SAC compiler includes an elaborate type inference system which through a hierarchy of array types infers the most specific types of these parameters statically. This enables the compiler to translate SAC function definitions into function codes that are adapted exactly to the shapes of the arrays they are applied to. Similarly, WITH-loops can be compiled into nestings of FOR-loops in compliance with the shape of the indexing vectors specified in the generator parts. If necessary, the compiler may even generate several instances of function or WITH-loop codes to operate on arrays of changing dimensionalities and shapes.

With respect to the running example, a specialized version of `rotate` is built which makes use of the known shape of `A`:

```
inline double[100, 100] rotate_100_100( int dim,
                                        int num,
                                        double[100, 100] A)
{
  max_rotate = [100, 100][dim];
  num = num % max_rotate;
  if( num < 0)
    num = num + max_rotate;

  offset = modarray( 0*[100, 100], [dim], num);
  slice_shp = modarray( [100, 100], [dim], num);
  B = with ( offset <= i_vec <= [100, 100]-1)
      modarray( A, i_vec, A[i_vec-offset]);
  B = with ( 0*slice_shp <= i_vec <= slice_shp-1)
      modarray( B, i_vec, A[[100, 100]-slice_shp+i_vec]);
  return(B);
}
```

After applying standard optimizations, e.g. *inlining, constant folding,* and *constant propagation* [ASU86, BGS94], we obtain:

```
  ...
  num = num % 100;
  if( num < 0)
    num = num + 100;

  B = with ( [0, num] <= i_vec <= [99, 99])
      modarray( A, i_vec, A[i_vec-[0, num]]);
  B = with ( [0,0] <= i_vec <= [99, num-1])
      modarray( B, i_vec, A[[0, 100-num]+i_vec]);
  ...
```

Note here, that for reasons of simplicity we treat the overloaded versions of the arithmetic functions as well as the function `modarray` as if they would be primitives. In the actual compiler this is achieved by unrolling these WITH-loops and applying the standard optimizations mentioned above. At this stage of compilation a SAC-specific compiler optimization called WITH-loop-folding is applied which tries to fusion consecutive WITH-loops (for details see [Sch97c]). The result of this fusion process is an internal WITH-loop representation which may have multiple disjoint generator parts. In this particular case the compiler inferes two generator parts: one for the range from `[0, num]` to `[99, 99]` and the other one from `[0,0]` to `[99, num-1]`:

```
  ...
  num = num % 100;
  if( num < 0)
    num = num + 100;

  B = internal_with :
        [0, num] <= i_vec <= [99, 99]    : A[i_vec-[0, num]];
        [0,0] <= i_vec <= [99, num-1]    : A[[0, 100-num]+i_vec];
  ...
```

Finally, another SAC-specific compiler optimization called *index vector elimination* is applied (for details see [Sch96, Sch97b]). This optimization tries to eliminate vectors that are built for indexing purposes only. In our example these are `[0, num]`, `[0, 100-num]`, and even `i_vec`. Since the data vectors of all arrays in SAC are compiled into a flat C representation, the expressions `A[i_vec-[0, num]]` and `A[[0, 100-num]+i_vec]` can be compiled into C array accesses `A_data[ i_vec_offset - num]` and `A_data[100-num+i_vec_offset]`, respectively, where `A_data` refers to the data vector of `A` and `i_vec_offset` refers to an offset into `A_data` which is created implicitly during the compilation of the WITH-loop. Due to a sophisticated compilation scheme for the internal WITH-loop representation we finally obtain compiled C-code which looks quite similar to a hand-coded version:

```
  ...
  num = num % 100;
  if( num < 0)
    num = num + 100;

  ... // allocate memory for B_data

  i_vec_offset=0;
  for( tmp_0 = 0; tmp_0<=99; tmp_0++)
    for( tmp_1 = 0; tmp_0<=num-1; tmp_1++, i_vec_offset++)
      B_data[ i_vec_offset] = A_data[100-num+i_vec_offset];
    for( tmp_1 = 0; tmp_0<=num-1; tmp_1++, i_vec_offset++)
      B_data[ i_vec_offset] = A_data[ i_vec_offset - num];
  ...
```

5 A Performance Comparison

This section briefly presents some comparative performance figures which show to which extent the SAC approach is competitive with FORTRAN77 and SISAL implementations in terms of runtime efficiency and memory space consumption. An exhaustive discussion of these results can be found in [Sch97b][1]. The comparison is based on the multigrid kernel MG of the NAS-benchmarks [BBB+94] which performs some prespecified number of complete multigrid cycles on a three-dimensional array of $2^n, n \in \{3, 4, ...\}$ entries per axis in the finest grid. Each cycle moves through a sequence of mappings from the finest to the coarsest grid of $4 * 4 * 4$ entries, followed by a sequence of alternatingly doing relaxations and coarse-to-fine grid mappings back to the finest grid.

The FORTRAN implementation of this algorithm was directly taken from the benchmark[2], the SISAL program was hand-coded to perform the same elementary computations in the same order as the FORTRAN benchmark, whereas the SAC program heavily utilizes WITH-loops to define a dimension-invariant solution.

The hardware platform used for this contest was a SUN ULTRASPARC-170 with 192MB of main memory. The FORTRAN program was compiled by the SUN FORTRAN compiler f77 version sc3.0.1 which generates native code directly. The SISAL and SAC programs were compiled by the SISAL compiler osc, version 13.0.2, and by the SAC compiler sac2c, respectively, both of which produce C-code as output. The GNU-C-compiler gcc version 2.6.3 was used to compile the C-code to native machine code. Program execution times

[1]The runtime figures presented in this paper are different from those in [Sch97b] since the back-end of the actual SAC compiler incorporates some of the optimizations mentioned in that paper as future work.

[2]We only simplified the initial array generation and modified the problem-size.

and space demands were measured by the operating system timer and process status commands, respectively.

Fig.2 shows the time and space demands of all three multigrid implementations for three different problem-sizes, these being 32, 64, and 128 elements per axis. The bars in

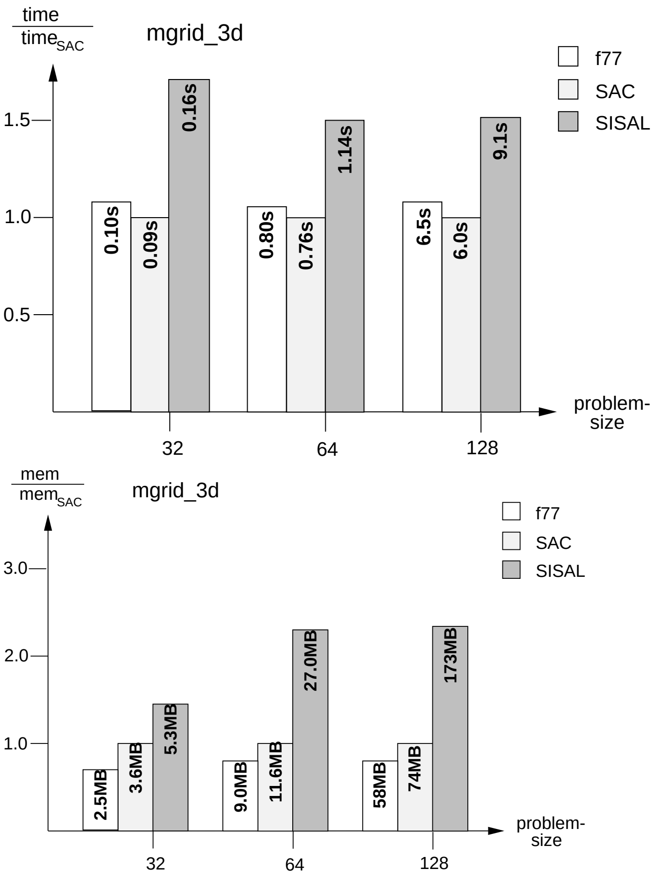

Figure 2: Time and Space Demand for Multigrid Relaxation on 3 Dimensional Arrays

the upper diagram depict execution times relative to that of the SAC program, with absolute times for one full multigrid cycle annotated inside the bars. For all three problem sizes the code compiled from the SAC program requires the shortest execution time. The FORTRAN and SISAL programs, respectively, require about 8% and about 50% more execution time.

The diagram in the lower part of Fig. 2 compares the relative space-demands of the three implementations. It shows that the FORTRAN program is the most space-efficient one, requiring on average only 80% of the space taken by the SAC program. The memory demand of the SISAL program significantly exceeds that of both of the others, in the case of the largest problem size by more than a factor of two.

6 Conclusion

In this paper, we discuss the pros and cons of using SAC for specifying high-level array operations. Instead of providing a fixed set of primitives, SAC offers meta-level language constructs which, on the one hand, are versatile enough to specify arbitrary high-level array operations, and on the other hand are restrictive enough to allow for the compilation into efficiently executable code. These constructs, called WITH-loops, may be considered sophisticated variants of the FORALL-loops in HPF or of array comprehensions in functional languages. They do not only allow for the specification of operations on arbitrary subranges of arrays but may also be specified in a form that is completely invariant against the shapes of the arrays they are applied to.

By means of simple examples it is shown that the high-level array operations that are typically available in array processing languages such as APL or Fortran 90 can be easily specified as WITH-loops in SAC. Thus, different sets of predefined high-level array operations may be provided through different versions of the standard library. As a consequence, the user may not only choose the appropriate set of high-level primitives but may even adapt the set of functions to each application's individual needs.

In addition to the gains made in terms of specificational flexibility, the meta-level approach proposed for SAC offers also advantages when it comes to compiling compositions of high-level array operations into efficiently executable code. Instead of requiring many specific optimization rules, a general scheme for folding WITH-loops suffices to avoid superfluous intermediate arrays. By means of an example it is shown that these operations are transformed into code that is identical to its hand-coded counterparts.

Finally, the paper compares some performance figures of high-level SAC specifications of the multigrid relaxation kernel from the NAS benchmarks against low-level specifications in SISAL and FORTRAN77. Despite its higher level of abstraction, the SAC implementation is compiled into code which is competitive with the other implementations both in terms of program runtimes and memory consumption.

References

[ABM+92] 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.

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

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

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

[BBB+94] D. Bailey, E. Barszcz, J. Barton, et al.: *The NAS Parallel Benchmarks.* RNR 94-007, NASA Ames Research Center, 1994.

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

[Ber97a] R. Bernecky: *An Overview of the APEX Compiler.* Technical Report 305/97, University of Toronto, 1997.

[Ber97b] R. Bernecky: *APEX: The APL Parallel Executor.* Master's thesis, University of Toronto, 1997.

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

[Bro85] J. Brown: *Inside the APL2 Workspace.* SIGAPL Quote Quad, Vol. 15, 1985, pp. 277–282.

[Bud88] T. Budd: *An APL Compiler.* Springer, 1988.

[Can93] D.C. Cann: *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory, LLNL, Livermore California, 1993. part of the SISAL distribution.

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

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

[Mul88] L.M. Restifo Mullin: *A Mathematics of Arrays.* PhD thesis, Syracuse University, 1988.

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

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

[Sch97a] S.-B. Scholz: *An Overview of* SAC *– a Functional Language for Numerical Applications.* In R. Berghammer and F. Simon (Eds.): Programming Languages and Fundamentals of Programming, Technical Report 9717. Institut für Informatik und Praktische Mathematik, Universität Kiel, 1997.

[Sch97b] S.-B. Scholz: *On Programming Scientific Applications in* SAC *- A Functional Language Extended by a Subsystem for High-Level Array Operations.* In Werner Kluge (Ed.): Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers, LNCS, Vol. 1268. Springer, 1997, pp. 85–104.

[Sch97c] S.-B. Scholz: *With-loop-folding in* SAC*–Condensing Consecutive Array Operations.* In C. Clack, T. Davie, and K. Hammond (Eds.): Proceedings of the 9th International Workshop on Implementation of Functional Languages. University of St. Andrews, 1997, pp. 225–242.

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

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