On Programming Scientific Applications in SAC - a Functional Language Extended by a Subsystem for High-Level Array Operations

Sven-Bodo Scholz

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

Abstract. This paper discusses some of the pros and cons of extending a simple functional language called SAC (for Single Assignment C) by array operations similar to those that are available in APL. The array operations in SAC are based on the ψ-calculus, an algebra of arrays which provides a formalism for specifying and simplifying array operations in terms of index set manipulations.
The programming techniques made possible by SAC are demonstrated by means of a functional program for the approximation of numerical solutions of partial differential equations by multigrid relaxation. This application is not only of practical relevance but also fully exposes the flavors of using high-level array operations. In contrast to specifications in other languages, e.g. in FORTRAN or SISAL, the SAC program is well structured, reasonably concise, and - what is most important - invariant against dimensionalities and shapes. However, sophisticated compilation techniques are necessary to avoid, whenever possible, the creation of temporary arrays and to eliminate redundant operations.
The paper also includes performance figures for a SAC implementation of the NAS-mgrid-benchmark which are competetive with those of a SISAL implementation.

1 Introduction

Scientific computations often require numerical approximations for solutions of partial differential equations, using sophisticated relaxation methods, which usually involve arrays of 10^6 .. 10^8 elements. Many of the applications such as fluid dynamic problems or weather forecasts must also be stepped through time, taking days or even weeks to run on todays supercomputers, and requiring enormous memory capacities. Clever algorithms and programming techniques must be complemented by optimizing compilers to obtain efficiently executable code which, whenever possible, avoids the creation of intermediate data structures and immediately re-claims memory that is no longer needed.

There is no doubt that the low-level programming style of imperative languages is highly suited for this purpose. Explicit control over the allocation (and de-allocation) of memory space is more or less directly placed in the hands of the programmer and can be adapted exactly to the needs of a given applications. The concept of multiple assignments allows to overwrite arrays no longer needed (and thus to re-use space that is already allocated), and iteration loops can be made to traverse, by properly chosen starts, stops and strides, exactly the array entries which contribute to the desired results.

Most of the efficiency of imperative programs derives from the rigorous exploitation of side-effects due to multiple assignments. Unfortunately, side-effects stand in the way of splitting large programs into concurrently executable parts. Since variables may be shared among them, it is the responsibility of either the programmer or of the compiler to organize the entire computation in a way that produces deterministic results irrespective of varying execution orders.

Functional languages are free of side-effects and therefore appear to be ideal candidates for non-sequential processing which in scientific computations is highly desirable to keep program runtimes within reasonable limits. The absence of side-effects, however, causes considerable problems with the efficient implementation of array operations. Conceptually, they must consume their operand arrays and create new result arrays, rather than overwriting existing ones, which generally is very costly in terms of both memory space and execution time expended. Inferring by static analysis which operations may overwrite their operands is not in all cases decidable [AP95], doing it at runtime inflicts considerable overhead for reference counting.

Functional languages such as MIRANDA [Tur86] or ML [QRM+87] provide little direct support for arrays. HASKELL [HAB+95] includes arrays as data types, using ZF-expressions (comprehensions) to generate array entries, but performance of the compiled code so far is not very competitive. SISAL [BCOF91] and all major data flow languages, e.g., ID and VAL [Nik88, AD79], provide control constructs for the traversal of array entries very similar to those of imperative languages, but strictly enforce the single assignment rule. Owing to very sophisticated compilation techniques, SISAL programs are known to outperform equivalent FORTRAN programs on multiprocessor systems[Can92]. However, SISAL does not offer substantial advantages in terms of programming techniques. Other than introducing another syntax, the programmer is still asked to specify array operations as iteration loops whose index variables and index ranges must be strictly adapted to array dimensionalities and shapes.

Integrating into functional languages an array processing concept similar to that of Iverson's APL [Ive62] considerably improves high-level array processing. Work to this effect has been reported in [Sch86, SBK92], which describes how array operations may be supported by graph reduction machinery, and in [Tu86, TP86], which introduces a functional language FAC based on APL syntax and on a lazy semantics.

Arrays in APL are treated as conceptual entities which can be operated upon by high-level structuring and value-transforming primitives. Explicit specifications of iteration loops (which often are the source of annoying errors due to incorrectly chosen starts, stops or strides) can be avoided in many cases.

Beyond these pragmatic advantages, the APL approach has also stimulated the development of an algebra of arrays, called the ψ-calculus [Mul88, Mul91, MJ91]. It is based on a small set of absolutely essential array operations which are solely defined in terms of dimensionalities, shapes and indexing functions. By application of the rules of this algebra, complex array expressions can be consequently simplified prior to actually compiling them to code, thus avoiding intermediate arrays whenever possible. Extending the ψ-calculus by a subset of high-level structuring and value transforming primitives yields a full-fledged subsystem for array processing which can be smoothly integrated into functional languages.

This paper is to investigate the pros and cons of programming real life scientific applications in the functional language SAC[Sch94, GS95] (for Single Assignment C) which includes high-level primitives for array operations as in the ψ-calculus. SAC is specifically designed to

- provide a functional language with a syntax very similar to that of C in order to ease, for a large community of programmers, the transition from an imperative to a functional programming style;
- support high-level array operations which are invariant against dimensionalities and shapes, liberate programming, whenever possible, from tedious and error-prone specifications of starts, stops and strides for array traversals, and also allow for term simplifications which avoid the creation of intermediate arrays;
- facilitate compilation to host machine code which can be efficiently executed both in terms of time and space demand.

Section 2 gives a brief introduction into the basic language constructs of SAC, and Section 3 introduces the high-level array operations provided by SAC. As an application problem, a program for the approximation of numerical solutions of partial differential equations by multi-grid relaxation [Hac85, HT82, Bra84] is extensively studied in Section 4. This application is not only of practical relevance, but also of interest with respect to the programming techniques required for the array operations involved. Section 5 outlines how a compiler actually produces efficiently executable code from the the program specifications presented in Section 4. Section 6 compares the performance of a SAC implementation of the multigrid algorithm from the NAS-benchmarks [BBB$^+$94] with that of equivalent SISAL and FORTRAN programs.

2 SAC - Single Assignment C

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 usual in many dataflow languages[AGP78, AD79, BCOF91],
- it supports high-level array operations, and
- programs need not to be fully typed.

Fig. 1 illustrates the similarity to C by means of a Sac implementation of the Euclidian algorithm for computing the greatest common divisor of two integers, 22 and 27 in the particular case. It consists of three function definitions: a func-

```
int gcd( int high, int low)
{
  if (high < low) {
    mem = low;
    low = high;
    high = mem;
  };
  while( low != 0) {
    quotient, remainder = modulo( high, low);
    high = low;
    low  = remainder;
  }

  return(high);
}

int, int modulo( int x, int y)
{
  quot   = to_int( x/y);
  remain = x - quot*y;
  return( quot, remain);
}

int main ()
{
  return(gcd( 22, 27) );
}
```

Fig. 1. Sac program for computing the greatest common divisor.

tion `gcd` which implements the Euclidian algorithm, a function `modulo` which computes the quotient and the remainder of the division of two integer numbers, and the `main` function which specifies the goal expression to be computed.

Two differences to a C implementation can be observed: the absence of type declarations for local variables, e.g. for `mem, quotient,` and `remainder` in the definition of `gcd`, and the usage of two return values for the function `modulo`.

Type declarations for local variables are optional since Sac requires a sophisticated type inference system to deal with arrays of varying dimensionalities and shapes. However, type declarations for parameters and return values of functions are mandatory to aid the type system in resolving function overloading. Since Sac, in contrast to C, does not include pointers or records, there is no other way of returning multiple function values but to make them explicit.

Otherwise, this SAC program uses language constructs which syntactically are exactly the same as in C, i.e. assignments, conditionals, and loop constructs. SAC programs can be straightforwardly transformed into nestings of LET(REC)-expressions, conditionals and local function definitions as they typically occur in other functional languages, i.e. a functional semantics for SAC can be easily defined in terms of these constructs (see [Sch96]).

3 Array-Processing in SAC

The array concept supported by SAC is based on the ψ-calculus, an algebra of arrays [Mul88, Mul91] which provides a formal apparatus for specifying and simplifying array operations in terms of indexed memory accesses in a form that is independent of dimensionalities and shapes, treating arrays, whenever appropriate, as conceptual entities. An array 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 infered from the shape vector as

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

where $[i_1, i_2]$ refers to the position $(i_1 * 3 + i_2)$ of the data vector.

A small set of primitives suffices to express all structuring operations on arrays as modifications of their shapes. By introducing a function that converts array indices into offsets within data vectors, the translation of ψ-primitives into loops of data vector accesses can be specified in the ψ-calculus itself. In combination with dedicated transformation rules, this allows for a formal reduction of arbitrarily nested array operations to starts, stops, and strides of direct indexing schemes, from which efficiently executable code which avoids the creation of superfluous intermediate data structures can be directly generated.

In SAC, arrays are generally 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 1-dimensional arrays are in fact vectors, they can be abbreviated as

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

Most of the primitives of the ψ-calculus are made available and defined as primitive functions, using the following syntax[1]:

[1] Note, that wherever in these definitions there is a vector or an array as argument, there may be expressions that evaluate to these data structures.

Let `a`, `b` denote arrays, let `v`=`[`$v_0, ..., v_{k-1}$`]` denote a vector of k integers, then

`dim( a)` returns the dimensionality, i.e., the number of axes, of the array `a`;
`shape( a)` returns the shape vector of `a`;
`psi( v, a)` $\equiv$ **`a[ v ]`** returns the subarray of `a` selected by the index vector `v`, provided that $k \leq$ `dim(a)` and that `v` $\leq$ `shape(a)` component-wise over all indices $j \in \{[0], ..., [(k-1)]\}$, otherwise it is undefined;
`take( v, a)` returns the subarray of `a` with shape `v` from the front ends of the respective axes in `a`, provided that $0 \leq$ `v` $\leq$ `shape( a)` component-wise, otherwise it is undefined;
`drop( v, a)` returns the subarray of `a` with `shape( shape(a)-v)` from the back ends of the respective axes, provided that $0 \leq$ `v` $\leq$ `shape( a)` component-wise, otherwise it is undefined;
`cat( k, a, b)` catenates the arrays `a` and `b` along their k^{th} axis if the shapes along the other axes are same, otherwise it is undefined;

All binary operations defined on scalar values are extended to component-wise operations on pairs of arrays and scalar values, as well as on pairs of arrays of the same shapes. A few examples are given in fig. 2 to illustrate these operations.

Let a be a 2×3 matrix with a $= \begin{pmatrix} 1\,2\,3 \\ 4\,5\,6 \end{pmatrix}$, then the following holds:

```
reshape( [2,3], [1,2,3,4,5,6]) == a
shape(a) == [2,3]
dim(a)   == 2
psi( [1,2], a) == a[[1,2]] == 6
psi( [1], a)   == a[[1]]   == reshape( [3], [4,5,6]) == [4,5,6]
dim( psi( [1], a)) == 1
take( [2,1], a) == reshape( [2,1], [1,4]) != [1,4]
take( [1,1], a) == reshape( [1,1], [1])   != 1
drop( [1,1], a) == reshape( [1,2], [5,6]) != [5,6]
dim( take([2,1],a)) == dim( take([1,1],a)) == dim( drop([1,1],a)) == 2
dim( [1,4]) == dim( [5,6]) == 1
shape( [1,4]) == shape( [5,6]) == [2]
cat( 1, a, reshape( [2,1], [7,8])) == reshape( [2,4], [1,2,3,7,4,5,6,8])
2*a == a+a == reshape( [2,3], [2,4,6,8,10,12])                          .
```

Fig. 2. Example applications of the primitive array operations of SAC.

The primitive functions introduced so far, in one way or another, affect all elements of the argument array(s) in the same way. Unfortunately, this may lead to rather awkward programs if only subarrays need to be operated on, or different operations need to be carried out on non-overlapping subarrays.

As a simple example, consider the problem of adding some constant value, say `1`, to the inner elements of an array `a` of arbitrary shape. These elements are identified by index vectors `i_vec` from the interval `0*shape(a)+1 <= i_vec <= shape(a)-2`. Expressing this computation in terms of an addition function which uniformly applies to all array elements requires that the array first be dismantled of all subarrays specified by index vectors with at least one zero or one maximal component, then the addition be performed on the remaining array, and finally the subarrays that have been taken off be attached again by catenation.

What needs to be cut off before and glued on after the addition of 1 in the special case of a two-dimensional array are the rows and columns with the lowest and highest indices. This leads to the following piece of SAC-program:

```
{ ...
  m = psi([0], shape(a));
  n = psi([1], shape(a));
  upper_row = take( [1,n], a);
  lower_row = drop( [m-1,0], a);
  left_col  = drop( [1,0], take( [m-1,1], a));
  right_col = take( [m-2,1], drop( [1,n-1], a));
  inner = take( [m-2,n-2], drop( [1,1], a));
  middle_section = cat( 1, left, cat( 1, inner+1, right));
  result         = cat( 0, upper, cat( 0, middle_section, lower));
... }
```

The disadvantages of this solution are obvious: The program is quite complicated and compilation to efficiently executable code is difficult, and last not least, it is dimension-specific: an adaptation to other dimensionalities requires extensive re-writing.

To overcome these programming problems (and the ensuing compilation problems as well), a more versatile construct for array operations is essential. For this purpose, SAC provides a variant of ZF-expressions called WITH-loops by which operations over pre-specified index ranges can be specified in a shape-independent form.

The syntax of WITH-loops is defined in fig. 3. 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' which represents a vector of this set. The filter part consists of boolean expressions that usually depend on the index variable. They restrict the set of index vectors to those for which all filter expressions evaluate to true. The operation part finally specifies the operation to be performed on each element of the index vector set. Basically, three different kinds of operation parts are available (see $ConExpr$ in fig. 3). 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 3) with neutral element $neutral$. Then

WithExpr	$\Rightarrow$ `with` (*Generator* $\left[\ ,\ \textit{Filter}\ \right]^{*}$) *Operation*
Generator	$\Rightarrow$ *Expr* `<=` *Identifier* `<=` *Expr*
Filter	$\Rightarrow$ *Expr*
Operation	$\Rightarrow$ [{ *LocalDeclarations* }] *ConExpr*
ConExpr	$\Rightarrow$ `genarray` (*Expr* , *Expr*)
	$\vert$ `modarray` (*Expr* , *Expr* , *Expr*)
	$\vert$ `fold` (*FoldFun* , *Expr* , *Expr*)
FoldFun	$\Rightarrow$ + $\vert$ $*$ $\vert$ *Identifier*

Fig. 3. WITH-loops in SAC.

- `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.
Using these WITH-loops, the above example problem can be specified as

```
{ ...
  result = with( 0*shape(a)+1 <= i_vec <= shape(a)-2)
           modarray( a, i_vec, a[i_vec]+1);
... }                                                .
```

Apart from the fact that this specification is more concise and easier to understand, it is also invariant against the shape and the dimensionality of `a`.

4 Programming numerical Applications in SAC: an Example

To illustrate how SAC programs for real world application problems look like, we consider, as an example, approximations of numerical solutions for Poisson equations, i.e., for partial differential equations (PDEs) of the general form

$$\Delta\, u\,(x_0,\ \dots\ ,x_{p-1}) = f\,(x_0,\ \dots\ ,x_{p-1}) \ \ \vert\ (x_0,\ \dots\ ,x_{p-1})\ \in\ \Omega\ \ ,$$

where Δ denotes the Laplace-operator and Ω denotes the domain within which solutions for u are defined, given some specific boundary values. For the sake of simplicity, these boundary values are assumed to be

$$u\ (x_0,\ \dots\ , x_q^{min},\ \dots\ , x_{p-1}) = u\ (x_0,\ \dots\ , x_q^{max},\ \dots\ , x_{p-1}) = 0$$

for all $q \in \{0, \dots, p-1\}$ throughout the example program, though in real world applications the boundary conditions may be more complicated.

Numerical solutions for Poisson equations are based on Gauss-Seidel or Jacobi relaxation algorithms. Both use discretizations of the PDEs on grids of some fixed mesh size h. They take the form

$$L\ u\ (i_0, \dots, i_{p-1}) = h^2 * f\ (i_0, \dots, i_{p-1})$$

in which all $x_q \in \{x_0, \dots, x_{p-1}\}$ are replaced by indices $i_q = \lceil x_q - x_q^{min}/h \rceil$ and the values of u and f are represented as p-dimensional arrays, with $i_q \in \{0, \dots, n_q - 1\}$ for all $q \in \{0, \dots, p-1\}$. The discretizised Laplace operator L adds up, in every inner grid point $(i_0, \dots, i_{p-1}) \in I_0 \times \dots \times I_{p-1}$ with $I_q \in \{1, \dots, n_q - 2\}$, weighted values of u in all adjacent points and in the point itself, to compute a new p-dimensional array u'.

Using a p-dimensional array D of elements $D[i_0, \dots, i_{p-1}]$ with $i_j \in \{0, 1, 2\}$ for all $j \in \{0, \dots, p-1\}$, this relaxation step is formally specified as:

$$\begin{array}{c} \forall i_0 \in \{1, ..., n_0 - 2\} ... \forall i_{p-1} \in \{1, ..., n_{p-1} - 2\} : \\ u'[i_0, ..., i_{p-1}] = \sum\limits_{j_0=0}^{2} ... \sum\limits_{j_{p-1}=0}^{2} D[j_0, ..., j_{p-1}] * u[(i_0 + j_0 - 1), ..., (i_{p-1} + j_{p-1} - 1)] \end{array}.$$

Both relaxation algorithms require that this computation be repeated several times until u' approximates the solution reasonably well, i.e. until a relaxation step changes the values in all grid points by less than some pre-specified threshold value.

With `i_vec=[`$i_0, \dots, i_{p-1}$`]` and `j_vec=[`$j_0, \dots, j_{p-1}$`]` denoting index vectors and `a` denoting an array , relaxation steps as above may be programmed in SAC in a completely shape-independent form as

```
{ ...
  new_u = with( shape(u)*0+1 <= i_vec <= shape(u)-2) {
            val = with( shape(D)*0 <= j_vec <= shape(D)-1)
                  fold( +, 0, D[j_vec] * u[ i_vec+j_vec-1 ]);
        } modarray( u, i_vec, val);
... }                                                          .
```

The inner WITH-loop of this SAC-statement computes a new value `val` in grid point `[`$i_0, \dots, i_{p-1}$`]` by forming, with `fold`, the weighted sum of the values in all adjacent points (and in the point itself). The outer WITH-loop steps through all but the boundary grid points to update the values of `u` by `val`, and to assign the entire array thus computed to `new_u`.

Note that the number of components of the index vectors is solely determined by the shapes of the arrays whose elements must be traversed, i.e., by `shape(u)` for the outer loop, and by `shape(D)` for the inner loop. The term `shape(u)*0+1` first multiplies all components of `shape(u)` by `0`, and then adds `1`, returning as the lower bound for `i_vec` a vector of p elements `[1,...,1]`, and the term `shape(u)-2` subtracts from all components $n_q \mid q \in \{0, \ldots, p-1\}$ of the vector `shape(u)` the value `2`, returning as the upper bound for `i_vec` a p-dimensional vector `[`$n_0 - 2, \ldots, n_{p-1} - 2$`]`. Since shape vectors are part of the array specifications, this piece of program can be applied to arrays of any given shape.

Jacobi or Gauss-Seidel relaxation is known to reduce fairly quickly high-frequency error components but does poorly on low-frequency errors. This is due to the slow point-to-point propagation of corrected values through the entire grid, which in real life applications may have up to 10^4 points in each dimension. A well-established remedy for this problem are so-called multigrid methods which embed relaxation steps into a recursive fine-to-coarse grid approximation, followed by a coarse-to-fine grid correction [Hac85, HT82, Bra84].

Roughly speaking, multigrid relaxation applies the Jacobi or Gauss-Seidel relaxation algorithm recursively to grids of mesh sizes h_1, ... , h_k , $h_{(k+1)}$, ..., h_m with $h_{(k+1)} = 2 * h_k$. It usually sets out with the finest grid and recursively works through some finite sequence of error approximations on increasingly coarser grids (which propagate errors in increasingly larger strides over the points of the original grid), followed by the passage of error corrections in the opposite direction (from coarser to finer grids). The mappings from finer to coarser grids and vice versa are done by calculating weighted averages of the values in adjacent grid points. To illustrate this, fig. 4 depicts two arrays of dimensionality two. The black dots are to represent grid points belonging to a coarse grid of

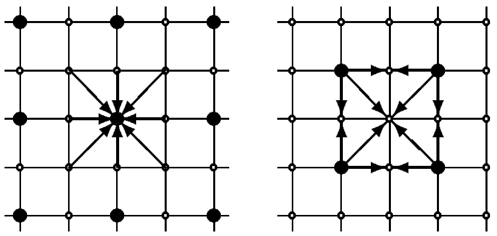


Fig. 4. The fine-to-coarse and coarse-to-fine grid mapping problem

mesh size $2 * h$, whereas both the blank and black dots represent the grid points that belong to a finer grid of mesh size h. The spider in the center of the left grid indicates how the elements of the coarser grid are calculated from those of the finer grid during a fine-to-coarse mapping. Conversely, the spider in the right grid shows from which elements of the coarser grid the elements of the finer grid are interpolated during a coarse-to-fine mapping.

Fine-to-coarse mapping, in fact, is very similar to relaxation: the value in a point of the coarser grid must be computed as a weighted sum over the values in all surrounding points, including the actual value in the point itself, of the finer grid. Let `u_c`, `u_f` and `W` respectively denote the arrays by which the coarser grid, the finer grid and the weight coefficients are represented, then the fine-to-coarse mapping may be specified in SAC in a shape-invariant form as

```
{ ...
  u_c = with( 0*shape(u_f)+1 <= i_vec <= shape(u_f)/2-1) {
          val = with( 0*shape(W) <= j_vec <= shape(W)-1)
                fold( +, 0, W[j_vec] * u_f[ 2*i_vec+j_vec-1]);
      } genarray( shape(u_f)/2+1, val);
... }
```

with `shape(u_c)`=`shape(u_f)/2+1`, and with `shape(W)`=`[3,...,3]` and `dim(W)`= p. For the two-dimensional case, a typical array of weight coefficients is

$$\mathtt{W} = 1/4 * \mathtt{C} \qquad \text{with} \qquad \mathtt{C} = \begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{pmatrix} \qquad ,$$

of which the value 1 in the center coincides with a point of the finer grid which maps directly into a point of the coarser grid (or coincides with the center of the spider in the left picture of fig. 4). This weight array lets the value in each point of the finer grid contribute with different weights to adjacent values of the coarser grid: 1/4 if it is in the center of the spider, 1/8 each to two points on each side if it is on a horizontal or vertical axis of the grid, and 1/16 each to four points if it is on the intersection between two diagonals.

The basic idea of specifying coarse-to-fine grid mapping in a shape-invariant form is to first initialize the array for the finer grid, which must have twice the shape of the coarser grid, in every other point along each axis with the respective values of the coarser grid, and all points in between with zero values. Then all values of the finer grid are computed in the same way as for the fine-to-coarse mapping, i.e., by adding up, in each point of the finer grid, weighted values of all points that are adjacent to it. This may be done with the same weight array as above, except that all its entities have to be multiplied by 4, i.e. we can use `C` as weight array for the coarse-to-fine grid mapping.

Thus, if the center of the weight array `C` coincides with a point of the coarser grid (see also the spider in the right picture of fig. 4), it reproduces the value in that point since it is multiplied by 1, and all surrounding points contribute zero values. Similarly, the values in points between two non-zero elements on a horizontal or vertical axis are computed as the sum of their values, multiplied by 1/2, and the values of points on the intersection between two diagonals are computed as the sum of four non-zero values, multiplied by 1/4. This algorithm can be implemented in SAC as follows:

```
{ ...
  u_f = with( 0*shape(u_c) <= i_vec <= shape(u_c)*2-3)
        genarray( shape(u_c)*2-2, 0);

  u_f = with( 0*shape(u_c) <= i_vec <= shape(u_c)-1)
        modarray( u_f, 2*i_vec, u_c[i_vec]);

  u_f = with( 0*shape(u_f)+1 <= i_vec <= shape(u_f)-2) {
          val = with( 0*shape(C) <= j_vec <= shape(C)-1)
                fold( +, 0, C[j_vec] * u_f[ i_vec+j_vec-1]);
      } modarray( u_f, i_vec, val);
... }                                                      .
```

This piece of program, exept for the coefficients of the weight array, again remains invariant against changing dimensionalities and shapes. However, there is a simple algorithm by which the coefficients of the weight array may be computed dependent on the dimensionality p of the grid. Given that all entries of the weight array have p-dimensional index vectors within range

$$[0,\ldots,0] <= \texttt{i\_vec} <= [2,\ldots,2]$$

and that the entry in the center has index vector $\texttt{i\_vec}_C = [1,\ldots,1]$, all there is to do is to compute $\texttt{i\_vec - i\_vec}_C = \texttt{i}_D$, count the number n of non-zero components in the vector $\texttt{i}_D$, and use the value 2^{-n} as entry `C[i_vec]`:

```
{ ...
  C = with( shape(u)*0 <= i_vec <= shape(u)*0+2) {
        n=0;
        for( i=0; i<dim(u); i++) {
          if( i_vec[[i]] != 1)
            n++;
        }
        val = pow(2,-n);
      } genarray( shape(u)*0+3, val);
... }                                            .
```

Apart from the WITH-loops introduced sofar, the full multigrid program primarily consists of WHILE-loops which iterate through relaxation steps embedded in successive fine-to-coarse grid mappings followed by successions of coarse-to-fine grid mappings. As these iterations are quite straightforward and not directly related to the shapes of the arrays involved, they are omitted here.

5 A Note on Compilation

The program fragments specified in the preceding section are essential components of a SAC implementation of multigrid relaxation which can be uniformly applied to argument arrays (grids) of varying dimensionalities and shapes. However, this generality may have to be paid for by some penalty on runtime performance, unless compilation to executable code can be parameterized at least by dimensionalities, if not shapes, of actual argument arrays. For this purpose, the SAC compiler includes an elaborate type inference system to infer through a hierarchy of array types the most specific of these parameters statically. This enables the compiler to translate SAC function definitions into function codes or WITH-loops into nestings of FOR-loops that are exactly adapted to the array parameters which actually have to be dealt with. If necessary, the compiler may even generate several instances of function or WITH-loop codes to operate on arrays of changing dimensionalities and shapes.

To convey the basic idea of how the SAC compiler goes about converting WITH-loops into executable code, we consider, as an example, the program fragment which implements single relaxation steps.

For two-dimensional arrays with a shape vector of the form `[n,n]`, it can be specialized at the SAC-level as

```
{ ...
  new_u = with( [1,1] <= i_vec <= [n-2,n-2]) {
            val = with( [0,0] <= j_vec <= [2,2])
                  fold( +, 0, D[j_vec] * u[ i_vec+j_vec-1 ]);
        } modarray( u, i_vec, val);
... }
```

simply by applying constant folding to the specifications of loop boundaries, which in the particular case are two-component vectors. Assuming

$$\mathtt{D} = \begin{pmatrix} 0 & 1/4 & 0 \\ 1/4 & -1 & 1/4 \\ 0 & 1/4 & 0 \end{pmatrix}$$

to be the array of weight coefficients, the inner WITH-loop can be further specialized as

```
{ ...
  new_u = with( [1,1] <= i_vec <= [n-2,n-2]) {
            val = 0.25 * u[i_vec+[-1,0]] + 0.25 * u[i_vec+[0,-1]]
                  - a[i_vec]
                  + 0.25 * u[i_vec+[0,1] ] + 0.25 * u[i_vec+[1,0] ];
        } modarray( u, i_vec, val);
... }
```

by means of loop unrolling in combination with another constant folding step.

Following these high-level optimizations, the SAC-to-C compiler takes over to compile the remaining WITH-loop into two nested C-FOR-loops. Since SAC represents arrays by shape and data vectors, the index vectors `i_vec` which are to select array elements must be converted, by means of a function `idx_to_off(i_vec,` `shape`) (with `shape` = `[n,n]` in the particular case), into offsets into the data vector. Taking `offset` as a variable that carries actual values of `idx_to_off(i_vec, [n,n])` the C-code for the WITH-loop looks like this:

```
{ ...
  offset = 0;

  /* copy non-indexed elems of dim 0 */
  for(tmp=0; tmp <=n; tmp++)
    new_u_data[offset++] = u_data[offset];

  for(i_vec_data[0]=1; i_vec_data[0]<=n-2; i_vec_data[0]++) {
    /* copy non-indexed elem of dim 1  */
    new_u_data[offset++] = u_data[offset];
    for(i_vec_data[1]=1; i_vec_data[1]<=n-2; i_vec_data[1]++) {
      val = ... ;           /* compiled code for the weighted  */
                            /* summation of neighbors          */
      new_u_data[offset++] = val;
    }
    /* copy non-indexed elem of dim 1  */
    new_u_data[offset++] = u_data[offset];
  }

  /* copy non-indexed elems of dim 0 */
  for(tmp=0; tmp <=n; tmp++)
    new_u_data[offset++] = u_data[offset];
... }
```

In order to generate efficiently executable code for the expression that computes `val` for each index vector `i_vec`, it is critically important to simplify as much as possible accesses in the data vector representation `u_data` of `u` to the four elements adjacent to `i_vec` (whose values have to added up). To do so, we make use of an optimization called index-vector-elimination which is based on the fact that `idx_to_off(i_vec + j_vec,` *shp*)=`idx_to_off(i_vec,` *shp*)+`idx_to_off(j_vec,` *shp*). This renders it possible to transform a selector term of the form, say, `u[ i_vec + [0,-1]]` into an access to `u_data` which is given as:

```
  u_data[idx_to_off(i_vec + [0,-1], [n,n]) ]
= u_data[idx_to_off(i_vec, [n,n]) + idx_to_off([0,-1], [n,n]) ]
= u_data[idx_to_off(i_vec, [n,n]) - n]
```

Since the variable `offset` in the above piece of program already holds actual values of `idx_to_off(i_vec, [n,n])`, the entire statement can, by similar transformation of the other four terms, be compiled to

```
{ ...
      val = 0.25 * u_data[offset-n] + 0.25 * u_data[offset-1]
          - u_data[offset]
          + 0.25 * u_data[offset+1] + 0.25 * u_data[offset+n];
      new_u_data[offset++] = val;
... }
```

as the body of the innermost FOR-loop. This optimization reduces the indexing arithmetic for accesses into the data vector to what is absolutely necessary.

Beyond that, the SAC-to-C compiler, of course, performs other optimizations which belong to the standard repertoire and, therefore, will not be outlined here.

6 Performance Figures

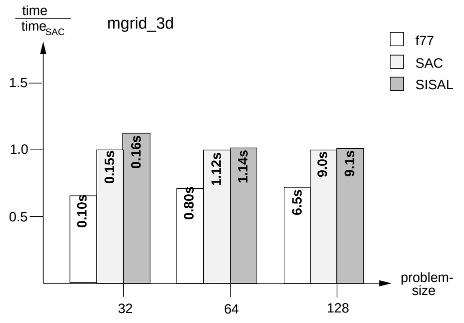
In this section we present some comparative performance measurements which show to which extend the SAC approach is competitive, in terms of program runtimes and memory space consumption, with FORTRAN and SISAL implementations. This comparison is based on the multigrid kernel MG of the NAS-benchmarks [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 4x4x4 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 uses the shape-invariant specifications of relaxation steps and of mappings between finer and coarser grids as outlined in Section 4.

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

Fig.5 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 the left 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 execution time of the SAC program is marginally shorter than that of the SISAL program, whereas the FORTRAN pro-

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



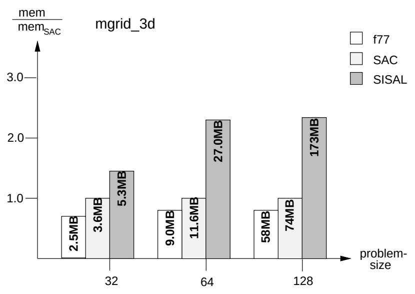


Fig. 5. Time and Space Demand for Multigrid Relaxation on 3 Dimensional Arrays

gram, on average, takes only 70% of the time for the SAC program. The reasons for the gap between the FORTRAN- and the SAC implementation are manifold.

Part of it may be attributed to the shape-independent specification of the coarse-to-fine mapping. As a comparison with a dimension-specific specification shows, the overhead due to additional additions/ multiplications causes about 10% of the slowdown. Also, the SAC compiler, as of now, uses the UNIX commands `malloc` and `free` to allocate and de-allocate heap space. Experiences from the implementation of the functional language KIR[Klu94] suggest that managing some program-specific heap from within the code can be expected to improve performance by another 10%. Last but not least, the C-code produced by the current compiler version is not yet fully optimized. Experiments with hand-coded improvements suggest that program execution times close to that of the FORTRAN implementation are within reach by integrating more elaborate optimizations.

The slightly poorer performance of the SISAL vs. the SAC implementation, despite the highly optimizing SISAL compiler [SW85, Can89, CE95, FO95], is presumably caused by the representation of arrays as vectors of vectors of data elements which inflicts a considerable overhead when accessing array entries that are adjacent to each other along other than the major axis.[3]

The diagram on the right of fig. 5 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 up by the SAC program. The additional space demand of the latter can be tracked down, by careful analysis of how both programs actually execute, to the coarse-to-fine grid mapping. Whereas the FORTRAN implementation needs to allocate only one instance of the finer array since it can do the interpolation of elements of the finer grid from those of the coarser grid directly, the SAC compiler version faithfully creates a second array into which the updated values are placed, as it is not yet equipped to fold WITH-loops whenever there is an opportunity to do so.

[3] For the next SISAL release (version 2.0)[BCOF91] an implementation of multi-dimensional arrays as continuous memory-blocks is intended [Old92].

The memory demand of the Sisal program significantly exceeds that of the Sac program, in the case of the largest problem size by more than a factor of two. We have not yet been able to identify the likely cause of this problem.

In order to dig a little deeper into the differences between the Sac implementation and the Sisal implementation, we also compared multigrid relaxation on two- and four-dimensional arrays. Whereas the Sac program could be used without changes, the Sisal program had to be re-written to adapt the for-loops to the actual dimensionalities of the arrays modified.

The performance measurements shown in fig. 6 nicely expose the overhead

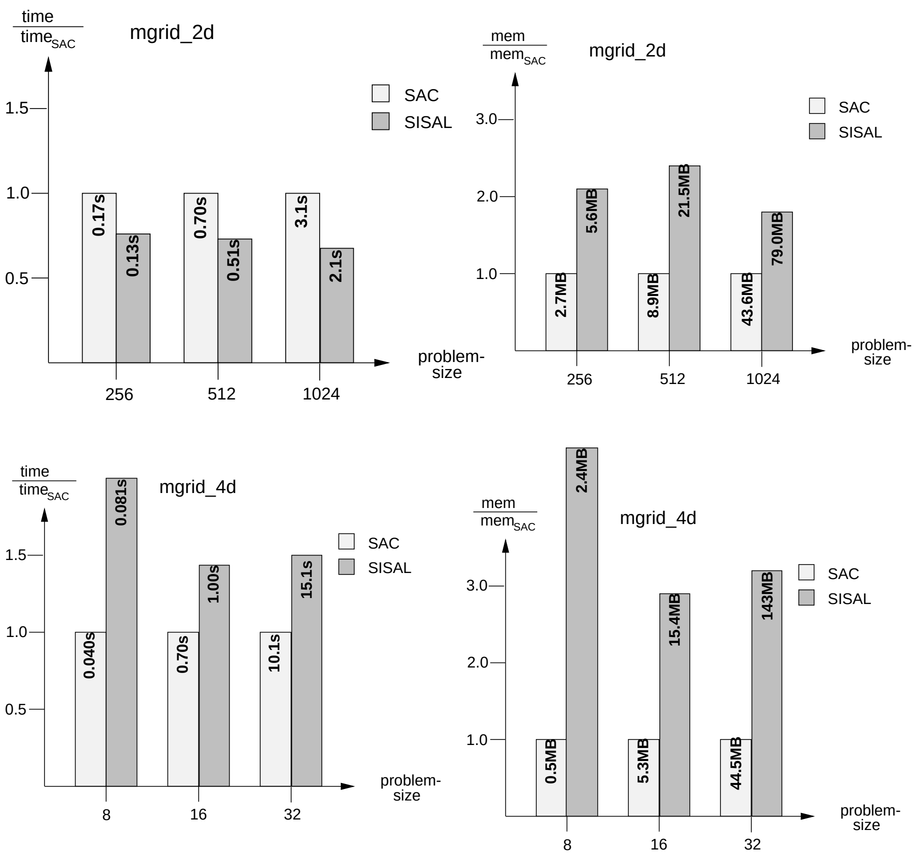


Fig. 6. Time and Space-Demand for Multigrid Relaxations on Different Dimensionalities

inflicted by the array representations of Sisal which grows with the dimensionality of the grid. While the two-dimensional Sisal program requires only 70% of the execution time of the Sac program, the situation is completely reverse in the four-dimensional case. Likewise, the space demand of the Sisal program increases relative to that of the Sac program with increasing grid dimensionality (from a factor of 2 for the two-dimensional case to a factor of 3 for the four-dimensional case).

7 Conclusion

This paper was to investigate, by means of a reasonably sophisticated array processing program (multigrid relaxation of PDEs), to which extent the integration of high-level array programming techniques into functional languages enhances program design and abstraction from problem-specific parameters, without paying too much of a performance penalty. The investigation is based on a performance comparison between a functional variant of C called Sac, whose design is focussed on the efficient support of high-level array operations, and Sisal, which is widely accepted to be the most efficient functional language for numerical applications.

Sac defines a set of structuring and value-transforming array operations which are invariant against dimensionalities and shapes, and apply uniformly to all elements (or subarrays) of arrays. They liberate programming from tedious and error-prone specifications of starts, stops and strides of iteration loops which prescribe traversals of array entries in a particular order, and also render programs applicable to arrays of different dimensionalities and shapes.

As this paper shows, elegant and concise programming can be achieved by the introduction of a high-level array concept similar to that of Apl, extended by array comprehension constructs (with-loops) which, whenever necessary or appropriate, apply operations (functions) only to selected subsets of array entries which are specified in terms of index ranges and filter expressions. Of course, with-loop constructs, which have been frequently used in the multigrid program, re-introduce, as index ranges in conjunction with filter expressions, starts, stops and strides into programming through the back door. However, they may be specified in a shape-invariant form, and the filters may select entries from within the pre-specified index ranges by criteria other than equidistant index positions as they can be traversed with `DO`-loops.

As has been demonstrated for the example program, compilation to efficiently executable code of high-level programs which use a mix of array comprehensions and primitive array operations poses no major problems. Although the Sac compiler used for the performance measurements still lacks some advanced optimizations as they are integrated into the Sisal compiler [SW85, Can89, CE95, FO95], competitive performance is achieved through a type-inference system which generates dimension dependent specializations of function bodies and through a special optimization technique for the elimination of temporary index vectors.

Further improvements of the C-code generated can be expected by the application of the transformation rules of the ψ-calculus, which serves as a formal basis for the array concept of Sac. This does not only include simple optimizations of nested applications of primitive operations, e.g. `take( v, (take( w, a))` $=$ `take( v, a)` but the systematic simplification of complex expressions such as the folding of two with-loops which, in the example examined in this paper, can be expected to reduce the space-demand of the Sac implementation to that of the Fortran implementation.

References

[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.

[AP95] P. Achten and R. Plasmeijer: *The ins and outs of Clean I/O.* Journal of Functional Programming, Vol. 5(1), 1995, pp. 81–110.

[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.

[Bra84] A. Brandt: *Multigrid Methods: 1984 Guide.* Dept of applied mathematics, The Weizmann Institute of Science, Rehovot/Israel, 1984.

[Can89] D.C. Cann: *Compilation Techniques for High Performance Applicative Computation.* Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1989.

[Can92] D.C. Cann: *Retire Fortran? A Debate Rekindled.* Communications of the ACM, Vol. 35(8), 1992, pp. 81–89.

[CE95] D.C. Cann and P. Evripidou: *Advanced Array Optimizations for High Performance Functional Languages.* IEEE Transactions on Parallel and Distributed Systems, Vol. 6(3), 1995, pp. 229–239.

[FO95] S.M. Fitzgerald and R.R. Oldehoeft: *Update-in-place Analysis for True Multidimensional Arrays.* In A.P.W. Böhm and J.T. Feo (Eds.): High Performance Functional Computing, 1995, pp. 105–118.

[GS95] C. Grelck and S.B. Scholz: *Classes and Objects as Basis for I/O in SAC.* In T. Johnsson (Ed.): Proceedings of the Workshop on the Implementation of Functional Languages'95. Chalmers University, 1995, pp. 30–44.

[HAB+95] K. Hammond, L. Augustsson, B. Boutel, et al.: *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language.* University of Glasgow, 1995. Version 1.3.

[Hac85] W. Hackbusch: *Multi-grid Methods and Applications.* Springer, 1985.

[HT82] W. Hackbusch and U. Trottenberg: *Multigrid Methods.* LNM, Vol. 960. Springer, 1982.

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

[Klu94] W. Kluge: *A User's Guide for the Reduction System* π-RED. Internal Report 9419, University of Kiel, 1994.

[MJ91] L.M. Restifo Mullin and M. Jenkins: *A Comparison of Array Theory and a Mathematics of Arrays.* In Arrays, Functional Languages and Parallel Systems. Kluwer Academic Publishers, 1991, pp. 237–269.

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

[Mul91] L.M. Restifo Mullin: *The Ψ-Function: A Basis for FFP with Arrays.* In L.M. Restifo Mullin (Ed.): Arrays, Functional Languages and Parallel Systems. Kluwer Academic Publishers, 1991, pp. 185–201.

[Nik88] R.S. Nikhil: *ID Version 88.1, Reference Manual.* CSG Memo 284, MIT, Laboratory for Computer Science, Cambridge, MA, 1988.

[Old92] R.R. Oldehoeft: *Implementing Arrays in SISAL 2.0.* In Proceedings of the Second SISAL Users' Conference, 1992, pp. 209–222.

[QRM$^+$87] D. Mac Queen, R.Harper, R. Milner, et al.: *Functional Programming in ML.* Lfcs education, University of Edinburgh, 1987.

[SBK92] C. Schmittgen, H. Blödorn, and W.E. Kluge: *π-RED* - a Graph Reducer for Full-Fledged λ-Calculus.* New Generation Computing, Vol. 10(2), 1992, pp. 173–195.

[Sch86] C. Schmittgen: *A Datatype Architecture for Reduction Machines.* In 19th Hawaii International Conference on System Sciences, Vol. I, 1986, pp. 78–87.

[Sch94] S.-B. Scholz: ***S**ingle **A**ssignment **C** – Functional Programming Using Imperative Style.* In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia, 1994.

[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.

[SW85] S. Skedzielewski and M.L. Welcome: *Data Flow Graph Optimization in IF1.* In FPCA '85, Nancy, LNCS, Vol. 201. Springer, 1985, pp. 17–34.

[TP86] H-C. Tu and A.J. Perlis: *FAC: A Functional APL Language.* IEEE Software, Vol. 3(1), 1986, pp. 36–45.

[Tu86] H-C. Tu: *FAC: Functional Array Calculator and its Application to APL and Functional Programming.* PhD thesis, Yale University, 1986.

[Tur86] D.A. Turner: *An Overview of Miranda.* SIGPLAN Notices, Vol. 21(12), 1986, pp. 158–166.