Accellerating APL Programs with SAC

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

Abstract

The paper investigates, how SAC, a purely functional language based on C syntax, relates to APL in terms of expressiveness and run-time behavior. To do so, three different excerpts of real world APL programs are examined. It is shown that after defining the required APL primitives in SAC, the example programs can be re-written in SAC with an almost one-to-one correspondence. Run-time comparisons between interpreting APL programs and compiled SAC programs show that speedups due to compilation vary between 2 and 500 for three representative benchmark programs.

Keywords: Language Comparison, SAC, Compilation, Runtime Performance.

1 Introduction

Array-oriented programming languages basically fall into two categories: languages that provide a high-level of abstraction for concise program design and fairly low-level languages aiming at utmost run-time efficiency.

APL [9], J [6], and NIAL [10] are typical representatives of the first category. The main objective in the design of these languages is to support means for specifying algorithms on arrays in a very concise and abstract manner. They provide array operations overloaded with many different combinations of array shapes, including scalars.

Although overloading improves conciseness, reusability, and elegance of programs, it causes difficulties when it comes to executing them efficiently with respect to raw run-times. It usually requires dynamic typing and execution in an interpreting environment. Much effort has been devoted to improving the run-time efficiency of such programs by application of sophisticated optimization techniques [4, 5] and by attempts to compile them [20, 7, 5, 3]. However, since complete shape and type inference generally requires rather sophisticated analysis methods, code efficiency in many cases is less than satisfactory.

Languages such as FORTRAN90 [1], HPF [8], or ZPL [12] fall into the second category. They derive from low-level (imperative) languages, e.g. FORTRAN or C, which have been primarily designed to facilitate compilation into efficiently executable code. Arrays in languages like FORTRAN90 are supported by so-called *intrinsic* array operations. Similar to their APL counterparts, these intrinsics are "hard-wired" into the compilers and can be applied to arrays of any rank.

Since the introduction of these array languages considerable efforts have been made to improve the code generated for programs that heavily use such intrinsic operations. After first attempts that were based on scalarizing the intrinsics and trying to apply conventional low-level optimization techniques, e.g. loop fusion, loop splitting, and forward substitution (for surveys see [22, 2, 21]), more recent papers suggest optimizations on higher levels of abstraction [11, 13, 14]. Although this work aims at combining high-level abstractions with run-time efficiency, there is still a significant difference relative to the APL approach of improving run-time performance. In a language like FORTRAN90, there is no way of defining new (non-intrinsic) array operations that are applicable to arrays of any rank. Although this may seem to be a minor restriction on first glance, it has a considerable impact on the programming style. Being able to define new array operations that are applicable to arrays of any rank allows the programmer to adjust the set of basic array operations to the needs of any given algorithm. Instead of problem-specific loop nestings, new more generally applicable operations may be defined, which subsequently may be combined to express the desired functionality. As a consequence, programs become more modular and easier to understand, which in turn increases code reusability and makes programs less error-prone.

The design of SAC [15, 16] tries to strike for a balance between high-level abstractions *and* compilation into efficiently executable code. It amalgamates a well-known syntax (that of C proper), functional abstractions that are inherently free of side-effects, and means to specify high-level array operations that are applicable to arrays of any rank. It has been shown that rank-invariant SAC programs can be compiled into code whose efficiency is competitive, both in terms of raw run-times and memory space consumptions, with equivalent hand-coded FORTRAN programs [16, 17, 18]. Although APL-like operators can be easily defined in SAC [19], it has not yet been tested how well SAC lends itself to writing programs completely in APL style, and which run-time behaviour can be expected of them.

The purpose of this paper is to find answers to these questions. To do so, we set out, in Section 2, with fragments of real-world APL programs and try to re-code them as close as possible in SAC. In Section 3, we compare the run-times

of the interpreted APL programs and of the compiled SAC programs. Section 4 contains some notes on the suitability of SAC as a target language for compiling APL. In Section 5 we summarize to which extend SAC can be used to emulate APL programs and things that need to be done in SAC to improve its expressiveness while maintaining the run-time edge.

2 Writing APL programs in SAC

In this section, we investigate the relationship between APL and SAC from a programmer's point of view or, more specifically, how APL programs may be re-coded in SAC. The answer to this question may also shed light on how much effort is required to compile APL programs into SAC code.

To this end, we have selected three existing APL-benchmark programs: MCONV, LOGD, and TOMCATV [3]. Each of them is manually translated into an equivalent SAC-program. The emphasis in translation clearly is on the creation of SAC-code that remains as close as possible to the original APL-specification. As a consequence, the resulting SAC-programs are neither the most elegant nor the most efficient implementations of the underlying algorithms.

2.1 The MCONV benchmark

MCONV is a kernel routine adopted from a seismic signal processing application. It implements one-dimensional convolution, taking two double-precision floating point vectors as arguments: the trace vector `tr` and the (usually) much shorter wavelet vector `wv`. Fig. 1 shows the original APL-implementation and below that the translated SAC-code. Convolution is implemented as inner product of the wavelet vector `wv` with a skewed copy of the trace vector `tr` reshaped into a matrix.

```
r←wv conv tr;h;n;t
h←tr,(¯1+n←ρwv)ρ0
t←(ιn)φ(n,ρh)ρh
r←(ρtr)↑wv+.× t
```

```
double[] conv( double[] wv, double[] tr)
{
  h = cat( 0, tr, genarray( shape(wv)-1, 0d));
  t = rot( iota( shape(wv)),
           genarray( shape(wv), h));
  r = take( shape(tr), VxM( wv, t));

  return( r);
}
```

Figure 1: MCONV in APL and SAC

The SAC translation follows exactly this scheme. The SAC library function `genarray(shp,val)` is equivalent to "`shpρval`" The library function `cat(d,A,B)` is equivalent to "`A,[d]B`" while `shape(a)` just means "ρa". So, the first line of APL-code can be translated without any problems. However, the second line is not as simple as the first one. Although, the vector `h` may easily be expanded to a matrix using the `genarray` function, SAC, for the time being, provides no means to rotate each line of a matrix by a different offset, as does "φ" in APL. Here, additional specification effort is required. However, Fig. 2 shows how this lack of functionality may be overcome. Both, the "ι" and the "φ" operators of APL can be expressed in SAC using the SAC-specific WITH-loop construct and the SAC-version of `rotate` with constant rotation offset[1].

```
int[] iota( int[] shp)
{
  res = with( . <= [i] <= .)
        genarray( shp, i);
  return( res);
}

double[] rot( int[] v, double[] a)
{
  res = with( . <= iv <= .)
        genarray( shape(v),
                  rotate( 0, -v[iv], a[iv]));
  return( res);
}

double[] VxM( double[] v, double[] m)
{
  neutr = genarray( [shape(m)[1]], 0d);
  res = with( [0] <= iv < shape(v))
        fold( +, neutr, v[iv] * m[iv]);
  return(res);
}
```

Figure 2: Additional SAC functions for MCONV

The result of `conv` is computed as the inner product of the wavelet vector `wv` and the skewed matrix `t`. Again, SAC does not provide a similarly general mechanism as APL's "$\omega.\alpha$" operator. However, its concrete instantiation in the benchmark, i.e. "+.×" applied to a vector and a matrix, may well be expressed in SAC. One possible SAC implementation of this vector–matrix product is the function `VxM` given in Fig. 2.

2.2 The LOGD benchmark

Similar to MCONV, LOGD represents a kernel routine of a signal processing application; it transforms a huge vector of double precision floating point numbers. Two slightly different APL implementations of LOGD have been investigated; they are referred to as LOGD1 and LOGD2 from here on.

We start with LOGD1 whose original APL specification and the translated SAC code are shown in Fig. 3. A closer look reveals that the SAC code of the function `logd` is nothing but a literal translation of the corresponding APL operators into their counterpart functions from the SAC array library. Moreover, the translation of the function `diff` which is also used in LOGD1 is nearly as simple. However, instead of dropping the rightmost element of the `sig` vector and adding a zero on its left end, we may use the SAC function `shift(dim,offset,new,A)` which shifts an array `A` along the `dim` axis `offset` elements to the right and fills the leftmost positions of `A` with the scalar value `new`.

The alternative APL specification LOGD2 differs from LOGD1 only w.r.t. the implementation of the `diff` function which here uses an n-ary "$\alpha/$" operator. Fig. 4 shows both the alternative APL specification of `diff` and the equivalent

[1] Note that upon a positive rotation offset, SAC rotates a vector to the right rather than to the left, as APL does.

```
RES←DIFF SIG
RES←SIG-0,¯1↓SIG

L←LOGD WV;RR
RR←.01
L←¯50⌈50⌊50×(DIFF WV)÷RR+WV
```

```
double[] diff( double[] sig)
{
  res = ( sig - shift( 0, 1, 0d, sig));
  return( res);
}

double[] logd( double[] wv)
{
  res = max( -50d, min( 50d,
                        50d * (diff( wv) / (0.01+wv))));
  return( res);
}
```

Figure 3: LOGD1 in APL and SAC

```
RES←DIFF SIG
RES←¯2-/0.0E0,SIG

double[] minus_2_reduce( int n, double[] v)
{
  res = with ( . <= iv < .) {
          tl = tile( [n], iv, v);
          val = tl[1] - tl[0];
        }
        genarray( shape(v)-n+1, val);
  return( res);
}

double[] diff( double[] x)
{
  res = minus_2_reduce( 2, cat( 0, [0d], x));
  return( res);
}
```

Figure 4: LOGD2 in APL and SAC

SAC code. Since for the time being SAC does not support higher-order functions, the functionality of the n-ary " $\alpha/$ " operator cannot be expressed in its general form. However, in conjunction with a specific operator to be applied, e.g. subtraction as in the benchmark example, it is well possible to emulate it in SAC, as is shown by the definition of the function `minus_2_reduce` in Fig. 4. The library function `tile(shp,offset,a)` selects a subarray of `a` with the shape `shp`, starting at index position `offset`. Actually, `tile` is a combination of `take` and `drop`. After having selected the appropriate 2-element subvector of `v`, its first element is subtracted from the second one as is specified in APL by the negative left operand to the n-ary " $\alpha/$ " operator.

2.3 The TOMCATV benchmark

The benchmark TOMCATV is a vectorized mesh generation program. Implemented in FORTRAN, TOMCATV is part of both the SPEC CFP'92 and the SPEC CFP'95 benchmark suites. Our APL version is directly based on the original FORTRAN source code. However, some minor modifications have been made insofar as always a fixed number of iterations is performed, and the code which computes the termination condition is manually lifted from the iteration loop. This is done to assure fair run-time comparisons between SAC and APL, as the SAC compiler would automatically move the corresponding statements outside the iteration loop, but the APL interpreter has no opportunity to do so. Since this benchmark is decidedly larger than MCONV and LOGD, we restrict the comparison between APL and SAC code to a representative set of selected subroutines.

```
r←x compmesh y;xx;yx;xy;yy;a;b;c;pxx;pxy;qxx;qxy;pyy;qyy
xx←1 ¯2↓¯1 0↓(2⌽x)-x
yx←1 ¯2↓¯1 0↓(2⌽y)-y
xy←¯2 1↓0 ¯1↓(2⊖x)-x
yy←¯2 1↓0 ¯1↓(2⊖y)-y

a←0.250×(xy×xy)+yy×yy
b←0.250×(yx×yx)+xx×xx
c←0.125×(xy×xx)+yx×yy

pxx←(1 2↓¯1 0↓x)-(2.0×1 1↓¯1 ¯1↓x)-1 0↓¯1 ¯2↓x
qxx←(1 2↓¯1 0↓y)-(2.0×1 1↓¯1 ¯1↓y)-1 0↓¯1 ¯2↓y
pyy←(2 1↓0 ¯1↓x)-(2.0×1 1↓¯1 ¯1↓x)-0 1↓¯2 ¯1↓x
qyy←(2 1↓0 ¯1↓y)-(2.0×1 1↓¯1 ¯1↓y)-0 1↓¯2 ¯1↓y

pxy←((2 2↓x)-¯2 2↓x)+(¯2 ¯2↓x)-2 ¯2↓x
qxy←((2 2↓y)-¯2 2↓y)+(¯2 ¯2↓y)-2 ¯2↓y

aa←-b
dd←b+b+(2.0÷0.98)×a
rx←(a×pxx)+(b×pyy)-c×pxy
ry←(a×qxx)+(b×qyy)-c×qxy
r←0
```

Figure 5: TOMCATV: function `compmesh` in APL

A major routine of TOMCATV is `compmesh`. This function recieves two square matrices `x` and `y` of size `N` as arguments and creates a total of four square matrices of size `N-2`. The APL implementation of `compmesh` is shown in Fig. 5, its translation to SAC in Fig. 6. We note that there are two different signatures for the two versions of `compmesh`. Since APL is limited to monadic and dyadic operators, additional parameters and return values can only be realized by global variables. The APL version of `compmesh` returns all four result matrices via the global variables `aa`, `dd`, `rx`, and `ry`; the actual return value of `compmesh` is not used. This contrasts with SAC which provides the required flexibility by supporting functions with any number of formal parameters and return values.

The body of `compmesh` essentially uses three different categories of operations: rotation along both axes using "ϕ" and "$\ominus$", selection of submatrices using "$\downarrow$" and various arithmetic operations. The first and the third categories pose no problems as far as translation into SAC code is concerned. However, since the SAC library functions `take` and `drop` do not support take or drop vectors with negative elements, applications of them cannot literally be translated into SAC. Instead, any APL "$\downarrow$" operation in `compmesh` must be expressed as a nested `take` and `drop` operation in the SAC version.

Most of the other functions of the TOMCATV benchmark can be translated more or less straightforwardly into equivalent SAC code, as demonstrated for `compmesh`, an exception being the function `fma`. It searches for the matrix element with the highest absolute value. The value of this element

```
double[], double[],
double[], double[] compmesh( double[] x, double[] y)
{
  xx = take( shape(x)-2,
             drop( [1, 0], (rotate( 1, -2, x) - x)));
  yx = take( shape(x)-2,
             drop( [1, 0], (rotate( 1, -2, y) - y)));
  xy = take( shape(x)-2,
             drop( [0, 1], (rotate( 0, -2, x) - x)));
  yy = take( shape(x)-2,
             drop( [0, 1], (rotate( 0, -2, y) - y)));

  a = 0.250 * (xy*xy + yy*yy);
  b = 0.250 * (yx*yx + xx*xx);
  c = 0.125 * (xy*xx + yx*yy);

  pxx = take( shape(x)-2, drop( [1,2], x))
        - (2.0 * take( shape(x)-2, drop( [1,1], x))
        - take( shape(x)-2, drop( [1,0], x)));
  qxx = take( shape(x)-2, drop( [1,2], y))
        - (2.0 * take( shape(x)-2, drop( [1,1], y))
        - take( shape(x)-2, drop( [1,0], y)));
  pyy = take( shape(x)-2, drop( [2,1], x))
        - (2.0 * take( shape(x)-2, drop( [1,1], x))
        - take( shape(x)-2, drop( [0,1], x)));
  qyy = take( shape(x)-2, drop( [2,1], y))
        - (2.0 * take( shape(x)-2, drop( [1,1], y))
        - take( shape(x)-2, drop( [0,1], y)));

  pxy = (drop( [2,2], x)
           - take( shape(x)-2, drop( [0,2], x)))
        + (take( shape(x)-2, x)
           - take( shape(x)-2, drop( [2,0], x)));
  qxy = (drop( [2,2], y)
           - take( shape(x)-2, drop( [0,2], y)))
        + (take( shape(x)-2, y)
           - take( shape(x)-2, drop( [2,0], y)));

  aa = -1d * b;
  dd = b + b + (2.0/0.98) * a;
  rx = a * pxx + (b*pyy - c*pxy);
  ry = a * qxx + (b*qyy - c*qxy);

  return( aa, dd, rx, ry);
}
```

Figure 6: TOMCATV: function `compmesh` in SAC

is returned along with the coordinates of its first occurrence in the row-wise unrolling of the matrix. Notwithstanding the problem that additional parameters have to be communicated to and from the function via global variables, the APL implementation of `fma` is short and concise as can be seen in Fig. 7.

The problem with translating `fma` to SAC is that neither the functionality of the dyadic "ι" nor reduction along a single coordinate are directly supported by SAC. In order to keep the translated SAC code as close as possible to the original APL specification, we have implemented both as SAC functions, as can be seen in Fig. 7. Since SAC, unlike APL, strictly distinguishes between arrays and scalars, two versions of `max_reduce` are required, one that operates on vectors and returns a scalar, and one that operates on arrays with at least rank two and returns an array whose rank is reduced by one. The first version may simply be implemented as a call to the SAC library function `maxval` which returns the maximum value of all elements of a given array. The second version, in contrast, may only be specified by a

```
z←fma y;t;ay;v
ay←|y
v←⌈/ay
t←⌈/v
i←v⍳t
j←ay[i;]⍳t
z←y[i;j]

double[] max_reduce(double[] array)
{
  res = with (. <= iv <= .)
        genarray( take( [dim(array)-1], shape(array)),
                  maxval( array[iv]));
  return(res);
}

double max_reduce(double[.] array)
{
  return(maxval(array));
}

int iota(double[.] vect, double val)
{
  pos = 0;
  while ((pos < shape(vect)[0]) && (vect[pos] != val)) {
    pos++;
  }
  return(pos);
}

double, int, int fma( double[] y)
{
  ay = abs(y);
  v  = max_reduce(ay);
  t  = max_reduce(v);
  i  = iota(v, t);
  j  = iota(ay[i], t);
  z  = y[i,j];
  return(z, i, j);
}
```

Figure 7: TOMCATV: function `fma` in APL and SAC

WITH-loop which first derives the result shape and then applies `maxval` to each subvector of the innermost rank. The dyadic "ι" which determines the index position of the leftmost occurrence of a value in a vector, can best be specified in a C-like fashion as shown in Fig. 7. However, once the functions `max_reduce` and `iota` have been implemented, the SAC version of `fma` itself becomes a rather literal translation of the original APL source.

3 Performance Comparison: Interpreted APL vs Compiled SAC

In this section, we compare the performance of the APL and the SAC implementations of the three benchmarks MCONV, LOGD, and TOMCATV, as described in Section 2. Our test system is a Pentium-II with 266MHz clock frequency and 64MB of main memory. For the interpretation of the APL programs we use APL+WIN 3.0 running under Windows95. The manually translated SAC programs are compiled by the current SAC prototype compiler SAC2C v0.8 running under LINUX, kernel revision 2.0.35. Gnu gcc 2.7.2.1 is used as a backend compiler to generate host machine code. All APL run-times given below are determined by the "$\square$`ts`" system function; the SAC run-times are user CPU times measured by the LINUX shell command `time`. They all are the minimal

run-times of ten independent runs of the respective benchmark programs.

3.1 Performance of MCONV

MCONV ρ tr	ρ wv	$time_{\text{APL}}$	$time_{\text{SAC}}$	$\frac{time_{\text{APL}}}{time_{\text{SAC}}}$
10,000	50	0.22 sec	0.09 sec	2.4
10,000	100	0.44 sec	0.20 sec	2.2
10,000	150	0.71 sec	0.32 sec	2.2
10,000	200	0.99 sec	0.42 sec	2.4
5,000	200	0.49 sec	0.20 sec	2.5
15,000	200	1.48 sec	0.71 sec	2.1
30,000	200	—	2.29 sec	—
45,000	200	—	3.78 sec	—
60,000	200	—	5.80 sec	—
150,000	200	—	15.17 sec	—

Figure 8: Performance of MCONV

Fig. 8 shows the run-times of the MCONV benchmark for various problem sizes determined by the lengths of the trace vector and the (shorter) wavelet vector. The compiled SAC code achieves speedups relative to interpreted APL programs by factors of 2 and more although the SAC implementation exactly follows the same algorithm as the APL program. Moreover, the problem sizes are varied within a range in which the interpreter should perform quite well. There is no explicit iteration and the array sizes are large enough to amortize setup costs over a large number of array elements.

A closer look at the two implementations in Fig. 1 reveals that by far the largest part of the execution time is spent on creating, transforming, and reducing two huge intermediate matrices of shape "`(ρwv),¯1+(ρtr)+ρwv`". However, whereas the APL interpreter more or less executes the program as it is specified, the SAC compiler transforms the original specification in a way that completely avoids intermediate matrices. In fact, SAC owes its performance edge over APL to compiled code from which intermediate matrices are completely eliminated in this particular case. Even more important than speeding up the execution times is the massive reduction in memory consumption. Whereas 15,000 by 200 is roughly the largest problem size that can be dealt with by the APL interpreter, requiring about 48MB of memory to hold the two intermediate matrices alone, SAC may easily handle a problem size that is ten times larger, as shown in Fig. 8. This may also be seen as evidence that SAC actually succeeds in avoiding intermediate matrices.

Thus, it turns out that the original APL implementation of MCONV, though being elegant and concise, is only of limited practical use due to its enormous memory demands. This led to the idea of re-implementing MCONV in APL in a less memory-consuming way. This program is shown in Fig. 9 along with a SAC translation which is kept to it as close as possible. While the value of the temporary `h` in the function `conv` is computed just as before, the result of `conv` is specified as a map operation of `VxV_at` over the indices of the trace vector `tr`. Note that `w` and `h` are global variables, so they can be used in the specification of `VxV_at`. The function `VxV_at` ignores its first operand which is just a dummy. It computes a vector product on the wavelet vector `w` and

```
r←dummy VxVat n
r←((ρw)↑n↓h)+.×w

r←wv conv tr
w←wv
h←tr,(¯1+n←ρwv)ρ0
r←0∘.VxVat ιρtr
```

```
double VxV( double[] v1, double[] v2)
{
  return( sum( v1 * v2));
}

double VxV_at( int dummy, int[] iv,
               double[] h, double[] wv)
{
  res = VxV( wv, tile( shape(wv), iv, h));
  return( res);
}

double[] OP_VxV_at( int scal, int[] vect,
                    double[] h, double[] wv)
{
  res =  with( . <= iv <= .) {
           val = VxV_at( scal, iv, h, wv);
         }
         genarray( shape( vect), val)
  return( res);
}

double[] conv( double[] wv, double[] tr)
{
  h   = cat( 0, tr, genarray( shape(wv)-1, 0d));
  res = OP_VxV_at( 0, iota( shape(tr)), h, wv);
  return( res);
}
```

Figure 9: MCONVopt: improved implementation of MCONV

the corresponding subvector of `h` starting at index position `n`. The SAC implementation exactly imitates the APL program in order to allow for a fair performance comparison. The two major differences are that `wv` and `h` are explicitly passed as arguments in function applications rather than being global variables and that outer and inner products have to be specialized to the specific operations since SAC does not support higher-order functions.

Run-time measurements for both the APL and the SAC implementations of MCONVopt are taken for the same problem sizes as those for MCONV; the results are summarized in Fig. 10. The memory requirements of the APL implementation are successfully reduced and do now allow for larger problem sizes. However, for problem sizes that MCONV can succesfully deal with, a severe slowdown by factors between 3.8 and 15 is encountered. For the SAC implementation of MCONVopt it is just the opposite: MCONVopt is faster by factors between 3 and 7 compared to MCONV. Thus, although largely imitating the APL specification, MCONVopt turns out to be an almost ideal SAC implementation of the MCONV benchmark in terms of run-time behaviour.

However, this immediately raises the question why is the APL version that slow? Considering the particular implementation, one would expect run-times to scale linearly both

MCONVopt ρ tr	ρ wv	$time_{\text{APL}}$	$time_{\text{SAC}}$	$\frac{time_{\text{APL}}}{time_{\text{SAC}}}$
10,000	50	3.40 sec	0.03 sec	113.3
10,000	100	3.52 sec	0.04 sec	88.0
10,000	150	3.62 sec	0.05 sec	72.4
10,000	200	3.79 sec	0.06 sec	63.2
5,000	200	1.26 sec	0.03 sec	42.0
15,000	200	5.52 sec	0.09 sec	61.3
30,000	200	27.19 sec	0.17 sec	159.9
45,000	200	83.98 sec	0.28 sec	299.9
60,000	200	185.70 sec	0.35 sec	530.6
150,000	200	$>$ 30 min	0.92 sec	—

Figure 10: Performance of MCONVopt

with the length of the trace vector and with the length of the wavelet vector. This is exactly the case with the SAC implementation, but not with the APL implementation. The latter is extremely sensitive against increasing lengths of the trace vector whereas the effect of the length of the wavelet vector on run-times is almost negligible. It is reasonable to assume that the APL implementation of the outer product with a user-defined function is the performance killer in the APL version of MCONVopt. Unfortunately, we haven't managed to find an APL implementation of MCONVopt with better performance figures. A much more elegant and presumably also faster implementation could be had with the CUT-operator which, however, is not available in our APL interpreter.

3.2 Performance of LOGD

LOGD1 ρ wv	$time_{\text{APL}}$	$time_{\text{SAC}}$	$\frac{time_{\text{APL}}}{time_{\text{SAC}}}$
500,000	0.99 sec	0.23 sec	4.3
1,000,000	2.31 sec	0.47 sec	4.9
1,500,000	3.57 sec	0.71 sec	5.0
LOGD2 ρ wv	$time_{\text{APL}}$	$time_{\text{SAC}}$	$\frac{time_{\text{APL}}}{time_{\text{SAC}}}$
500,000	1.38 sec	0.23 sec	6.0
1,000,000	2.85 sec	0.51 sec	5.6
1,500,000	4.56 sec	0.74 sec	6.2

Figure 11: Performance of LOGD

Fig. 11 includes the performance data for the two variants of the LOGD benchmark. In the case of LOGD1, SAC achieves a speedup of a factor of about five over the interpreted APL implementation. Whereas the APL version of LOGD2 is slightly slower than that of LOGD1, the equivalent SAC version takes about the same time, resulting in a speedup by a factor of about 6. Here again the question must be raised why is SAC so much faster than APL for this benchmark? The APL version of LOGD1 is non-iterative, and represents good quality coding style. The entire program does not contain more than 8 built-in APL operators which step by step transform a relatively large data vector, i.e., the overhead inflicted by the APL interpreter should be negligible relative to useful computations. So, compilation per sé cannot be expected to improve performance significantly beyond what can be accomplished with interpretation. However, in SAC compilation is the key to large-scale program restructuring based on detailed program analysis. For both LOGD1 as well as LOGD2, the SAC compiler is able to generate target code which completely avoids the creation of all intermediate arrays produced by the individual operators. In fact, the original SAC specification is internally transformed into a single, complex array operation which is then applied to the argument vector. This large-scale code-restructuring optimization technique, called WITH-loop folding, is described in detail in [19, 18]. For both versions of the LOGD benchmark, WITH-loop folding is the key to the superior performance of SAC relative to APL.

3.3 Performance of TOMCATV

TOMCATV ρ mat	$time_{\text{APL}}$	$time_{\text{SAC}}$	$\frac{time_{\text{APL}}}{time_{\text{SAC}}}$
20, 20	1.59 sec	0.11 sec	14.5
40, 40	4.45 sec	0.38 sec	11.7
60, 60	8.68 sec	0.79 sec	11.0
80, 80	14.23 sec	1.50 sec	9.5
100, 100	22.41 sec	2.47 sec	9.1
120, 120	32.54 sec	4.05 sec	8.0

Figure 12: Performance of TOMCATV

Run-time figures for the third benchmark, TOMCATV, are shown in Fig. 12 for various sizes of the square matrices being transformed. Similar to LOGD, the SAC implementation of TOMCATV is significantly faster than the APL version. Speedup factors are between 8 for the largest problem size under consideration and more than 14 for the smallest problem size. This speedup degression with increasing problem sizes can simply be explained by the decline of the interpretive overhead relative to the overall computation performed.

Similar to LOGD, the WITH-loop folding optimization technique generates target code that avoids large numbers of superfluous intermediate arrays. The APL implementation of the function `compmesh` (cf. Fig. 5) alone creates almost 90 large intermediate matrices provided that no specific optimizations apply what seems difficult in an interpreting environment. Here, the SAC strategy of implementing all array operations by the more general WITH-loop construct combined with the capability of folding subsequent WITH-loops to form a single more complex one provides an enormous optimization potential. In fact, the SAC compiler succeeds in transforming the implementation of `compmesh` into four complex WITH-loops, each specifying how to compute one result matrix. However, this folding cannot be had without problems. Whereas the APL interpreter computes intermediate results which are shared in the computation of more than one result matrix, WITH-loop folding results in computing each result matrix from scratch. As the run-time figures show, this does not at all outweigh the positive performance impact of WITH-loop folding. However, an additional opportunity for an optimization becomes apparent. Sharing of intermediate results may well be re-introduced by fusion of two or more WITH-loops to a single compound WITH-loop

that computes several result arrays simultaneously. With-loop fusion, however, has not yet been implemented into the SAC compiler, but remains subject to future work. Nevertheless, it makes clear that in particular the more complex TOMCATV benchmark provides various additional opportunities for optimization that presumably change the relative run-time performance further in favor of SAC.

4 A Note on Compiling APL to SAC

Since manual translation of APL programs into SAC can obviously be done more or less directly and the benchmarks show speedups between 2 and 500, compiling APL to SAC appears to be a worthwhile undertaking. The main problem is to specify transformation rules for *all* legitimate APL constructs or at least for some large subset of them. This section is to identify to which extend SAC and its compiler have to be extended and what kind of restrictions may possibly have to be imposed on APL programs in order to facilitate a smooth compilation of APL to SAC.

Many of the pre-defined APL functions are available in SAC either as built-in (so-called *intrinsic*) operations, or as part of the SAC standard library. Including into the SAC standard library most of the APL operators that are not yet supported poses no major problem. Exceptions are the APL operators "⊂" and "⊃" that manipulate nested arrays, the problem being that SAC does not support arrays that contain subarrays of different shapes. Including them would require drastic changes of both the current type system and of the internal array representation from straight data vectors, say, into nested vector representations.

User-defined APL functions can be more or less one-to-one translated into equivalent SAC functions. Exceptions relate to the usage of global variables. Since SAC is a purely functional language, there is no concept of side-effects. Instead, results of function applications have to be passed explicitly as function values to the calling context. As a consequence, compilation of APL functions requires an analysis to detect and subsequently eliminate side-effects. Techniques for doing so can be found in the APEX compiler [3] which compiles APL to SISAL.

The compilation of the higher-order APL functions such as "$\omega.\alpha$" or "$\circ.\alpha$" is more difficult since SAC does not (yet) support higher-order functions. Instead, each combination of higher-order APL operators and functions has to be replaced by a specialized SAC version, as is done in the APEX compiler. Exceptions are "$\alpha/$" and "$\alpha\neq$" which in some contexts can be realized in SAC using so-called *fold* WITH-*loops*. However, this is only possible if the reduction function α is associative and commutative since the folding order in SAC is non-deterministic.

Though it is possible in principle to compile APL into SAC programs, a compiler implementation would have to take care of another problem. In contrast to APL, SAC requires programs to be statically typed as an essential prerequisite for compilation into highly efficient code. This could be considered a conceptual advantage for APL programs as well since typing, beyond efficiency considerations, also improves confidence in program correctness. Unfortunately, the type system of SAC also rejects programs whose APL counterparts may be type-correct, but whose array shapes cannot be inferred statically. However, in a follow-up version of the SAC compiler the type system will be relaxed so that only the ranks but not the exact shapes need to be known statically, which likely covers most APL programs.

5 Conclusion

The objective of this paper was to find out how much effort it takes to translate APL programs into SAC programs and how both versions compare with respect to run-time performance. Our findings are based on three benchmark programs which also gives some insight into the expressiveness of both languages.

The means for defining rank-invariant functions in SAC considerably facilitate the translation of APL programs into SAC programs. Supporting in SAC higher-order functions would not only improve the elegance of SAC specifications but also the compilation of APL programs which make use of this feature. Moreover, adding new data structures to SAC would help to implement nested arrays, and the type system would have to be relaxed in order to facilitate compilation of programs in which the exact shapes cannot be inferred statically.

The run-time figures presented in this paper show that the compiled SAC programs outperform the interpreted APL programs by factors of 2 to 500. These improvements can be attributed to the various optimizations implemented in the current SAC compiler. In particular, WITH-loop-folding has shown to be essential for avoiding superfluous temporary arrays. Closer examination of the code generated by the SAC compiler for the given benchmarks exposed further opportunities of program optimization, e.g. WITH-loop fusion or memory pre-allocation analysis, both of which are not yet included in the SAC compiler.

Acknowledgements: We would like to thank Robert Bernecky for the APL source code of the benchmark programs presented in this paper and his overall support in APL questions. We also would like to thank APL2000 Inc. for their support concerning the APL interpreter.

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

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

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

[6] C. Burke. *J and APL*. Iverson Software Inc., Toronto, Canada, 1996.

[7] G.C. Driscoll and D.L. Orth. Compiling APL: The Yorktown APL Translator. *IBM Journal of Research and Development*, 30(6):583–593, 1986.

[8] High Performance Fortran Forum. *High Performance Fortran language specification V1.1*, 1994.

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

[10] M.A. Jenkins and W.H. Jenkins. *The Q'Nial Language and Reference Manuals*. Nial Systems Ltd., Ottawa, Canada, 1993.

[11] 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.

[12] C. Lin. ZPL Language Reference Manual. UW-CSE-TR 94-10-06, University of Washington, 1996.

[13] G. Roth and K. Kennedy. Dependence Analysis of Fortran90 Array Syntax. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Apllications*, 1996.

[14] G. Roth and K. Kennedy. Loop Fusion in High Performance Fortran. CRPC TR98745, Rice University, Houston, Texas, 1998.

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

[16] 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.

[17] 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.

[18] S.-B. Scholz. A Case Study: Effects of WITH-Loop-Folding on the NAS Benchmark MG in SAC. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages*. University College, London, 1998.

[19] 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.

[20] J. Weigang. An Introduction to STSC's apl compiler. In *APL89 Conference Proceedings*, pages 231–238. ACM SIGAPL Quota Quad, volume 15, 1989.

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

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