Parallel Processing Letters

SAC — FROM HIGH-LEVEL PROGRAMMING WITH ARRAYS TO EFFICIENT PARALLEL EXECUTION

CLEMENS GRELCK
Institute of Software Technology and Programming Languages, University of Lübeck
Ratzeburger Allee 160, 23538 Lübeck, Germany

and

SVEN-BODO SCHOLZ
Institute of Computer Science and Applied Mathematics, University of Kiel
Olshausenstraße 40, 24098 Kiel, Germany

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

SaC is a purely functional array processing language designed with numerical applications in mind. It supports generic, high-level program specifications in the style of Apl. However, rather than providing a fixed set of built-in array operations, SaC provides means to specify such operations in the language itself in a way that still allows their application to arrays of any rank and size. This paper illustrates the major steps in compiling generic, rank- and shape-invariant SaC specifications into efficiently executable multithreaded code for parallel execution on shared memory multiprocessors. The effectiveness of the compilation techniques is demonstrated by means of a small case study on the PDE1 benchmark, which implements 3-dimensional red/black successive over-relaxation. Comparisons with Hpf and Zpl show that despite the genericity of code, SaC achieves highly competitive runtime performance characteristics.

1. Introduction

Programming language design essentially is about finding a suitable tradeoff between support for high-level program specifications and efficient runtime behavior. In the context of array programming, the data parallel approach seems to be well suited to meet this goal: using functions that operate on entire arrays rather than loop nestings and explicit indexing does not only improve program specification, but also provides opportunities for compilers to generate parallel code.

For example, intrinsic array operations in Fortran-90/Hpf allow for very concise specifications of algorithms that manipulate entire arrays in a homogeneous way. However, if operations depend on the structure of argument arrays, things become more difficult. One remedy to this problem, other than using conventional loop nestings, is the *triple notation* [1]. Unfortunately, it has some drawbacks as

well: program specifications become less readable, more error-prone, and triple-annotated assignments restrict the arrays involved to particular ranks.

The programming language ZPL [8] offers a more elegant solution for this problem by introducing *regions* [9]. Regions are either statically or dynamically defined sets of array indices. They can be used to map any scalar operation to all the elements referred to by a region. In order to enable more sophisticated mappings, e.g. stencil computations, ZPL provides *prepositions*. They specify mappings of the indices, e.g. linear projections or permutations. Owing to the dynamic scoping of regions, in ZPL entire procedures can be applied to arrays of varying ranks. Unfortunately, this concept precludes applications where the functionality of a procedure also depends on the shape of its argument arrays rather than solely on their element values because regions and prepositions are not first-class objects in ZPL.

The functional array language SAC (Single Assignment C, pronounced "sack") [23] takes the idea of high-level generic array programming considerably further than HPF or ZPL. SAC introduces arrays as abstract data objects with certain algebraic properties rather than merely as mappings into memory; in particular, all memory management for arrays is done implicitly by the runtime system. In contrast to FORTRAN-90/HPF, SAC does not provide compound array operations as intrinsics. Instead, so-called WITH-*loops* can be used to define such array operations in SAC itself. Still, they may be applied to arrays of any rank and size, a property which in other languages is usually restricted to built-in primitives.

Similar to the region concept of ZPL, WITH-loops can be used to map scalar operations to subsets of the elements of argument arrays. However, there are two main differences between WITH-loops and regions. First, WITH-loops are legitimate right-hand-side expressions that evaluate to complete arrays, and second, the set of array indices to which a scalar operation is to be applied as well as mappings of index vectors are specified by ordinary (first-class) expressions. The latter property allows for much more generic program specifications in the style of APL.

The functional side-effect free semantics in general and the WITH-loop construct in particular are amenable to implicit parallelization. For the time being the SAC compiler generates multithreaded code for parallel execution on shared memory multiprocessors [14]. In conjunction with rigorous type specialization and optimization schemes runtime performance characteristics have been achieved which despite the high-level generic programming methodology have been found competitive with low-level implementations [13].

This paper outlines the major steps in compiling rank- and shape-invariant high-level SAC programs into efficiently executable multithreaded C code. For illustration purposes we refrain from giving exact transformation schemes and employ a rather simple example instead: compilation of multi-axis array rotation is sketched out in a step-by-step process. Advantages of the generic programming methodology supported by SAC and the effectiveness of the compilation techniques described are demonstrated by means of a small case study: red/black successive over-relaxation is realized by only five lines of SAC code based on generic multi-axis rotation.

The paper is organized as follows. After a brief introduction to SAC in Section 2, the compilation process is sketched out in Section 3 and the case study in Section 4. Section 5 summarizes some more related work while Section 6 concludes.

2. SAC — Single Assignment C

The core language of SAC is a functional subset of C, a design which aims at simplifying adaptation for programmers with a background in imperative programming languages. This kernel is extended by n-dimensional arrays as first class objects. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, arrays of any rank, e.g. `int[+]`, and a most general type encompassing both arrays and scalars: `int[*]`. SAC provides a small set of built-in array operations, basically primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`).

WithLoopExpr	⇒	**with** (*Generator*) *Operation*
Generator	⇒	*Expr Relop Identifier Relop Expr*
Relop	⇒	< | <=
Operation	⇒	**genarray** (*Expr* , *Expr*)
	|	**modarray** (*Expr* , *Expr*)
	|	**fold** (*FoldOp* , *Expr* , *Expr*)

Figure 1: Syntax of with-loop expressions.

Compound array operations are specified using WITH-loop expressions, whose core syntax is outlined in Fig. 1. A WITH-loop basically consists of two parts: a *generator* and an *operation*. The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions, which must evaluate to integer vectors of equal length, define lower and upper bounds of a rectangular index vector range. Let a and b denote such vectors of length n, then a generator of the form `( `a` <= i_vec < `b` )` refers to the following set of index vectors: $\{i\_vec \mid \forall_{j \in \{0,\ldots,n-1\}} : a_j \leq i\_vec_j < b_j\}$.

The operation specifies the computation to be performed for each element of the index vector set defined by the generator. Let *shp* denote a SAC expression that evaluates to an integer vector, let *i_vec* denote the index variable defined by the generator, let *array* denote a SAC expression that evaluates to an array, and let *expr* denote any SAC expression. Moreover, let $fold\_op$ be the name of a binary commutative and associative function with neutral element *neutral*. Then

- `genarray(` *shp*`,` *expr*`)` creates 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*`,` *expr*`)` defines 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*`[`*i_vec*`]` at all other index positions;
- `fold(` $fold\_op$`,` *neutral*`,` *expr*`)` specifies a reduction operation; starting out with *neutral*, the value of *expr* is computed for each index vector from the specified set and these are subsequently folded using $fold\_op$.

The usage of vectors in WITH-loop generators as well as in the selection of array elements along with the ability to define functions which are applicable to arrays of

any rank and size allows for implementing APL-like compound array operations in SAC itself. This feature is exploited by the SAC standard library, which provides, among others, element-wise extensions of arithmetic and relational operators, typical reduction operations like sum and product, various subarray selection facilities, as well as shift, rotate, and transpose operations. More information on SAC is available at http://www.sac-home.org/ .

3. The compilation process

This section sketches out the major steps in compiling rank- and shape-invariant high-level SAC specifications into efficiently executable multithreaded code. As pointed out before, we focus on multi-axis array rotation as an example rather than providing complete compilation schemes. The function

```
int[*] rotate (int[.]  offsets, int[*] A)
```

from the SAC standard library rotates the given integer array `A` in each dimension by as many elements as specified by the corresponding element of the vector `offsets`. Note that the function `rotate` is applicable to arrays of any rank (`int[*]`) and, hence, the first argument `offsets` may be a vector of varying length (`int[.]`).

```
int[*] rotate (int[.] offsets, int[*] A)
{
  for (i=0; i < min( shape(offsets)[[0]], dim(A)); i+=1)
  {
    max_rotate = shape(A)[[i]];
    offset = offsets[[i]] % max_rotate;
    if (offset < 0) offset += max_rotate;

    lower_bound = modarray( 0 * shape(A), [i], offset);
    upper_bound = modarray(     shape(A), [i], offset);

    B = with (. <= iv < upper_bound)
        genarray( shape(A), A[ iv + shape(A) - upper_bound]);

    C = with (lower_bound <= iv <= .)
        modarray( B, A[ iv - lower_bound]);

    A = C;
  }

  return( A);
}
```

Figure 2: SAC implementation of multi-axis array rotation.

The implementation of `rotate` is shown in Fig. 2. Rotation along multiple axes is realized as a sequence of rotations along individual axes. If the length of the vector of offsets does not match the rank of the given array, either the array remains unrotated along rightmost axes or surplus rotation offsets are ignored. Rotation along a single axis starts with extracting the respective offset from the vector `offsets` and its normalization to the range between `0` and the extent of `A` along this axis.

Rotation itself is realized by dividing the set of legitimate index vectors of `A` into two partitions `a` and `b`, as illustrated in Fig. 3 for the 2-dimensional case. The `genarray`-WITH-loop defines an intermediate array `B` of the same shape as `A`, which contains all elements belonging to partition `b` of array `A` being correctly rotated to their new locations. According to the semantics of the WITH-loop, the remaining

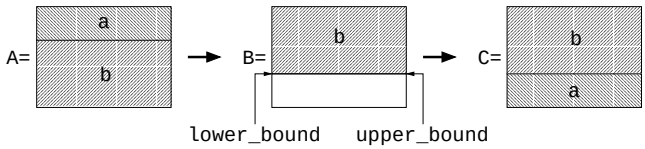

Figure 3: Illustration of rotation along single axis.

elements of `B` (the white box in Fig. 3) are set to zero. The `modarray`-WITH-loop defines the result array `C` with partition `a` from array `A` being rotated to its new location and with the remaining elements being implicitly taken from the corresponding index positions of the array `B`. The boundary vectors of the WITH-loop generators are defined using the SAC function `modarray(`*`array,index,value`*`)`, which yields a new array identical to *`array`* except for position *`index`* which is set to *`value`*. Dots in boundary positions form a convenient syntactic shortcut: they represent the least or the greatest legal index vector, respectively. Eventually, `C` is re-assigned to `A` for the next FOR-loop iteration. This seeming contradiction in a single assignment language is resolved by the semantics of SAC, which introduces conventional loops as syntactic sugar for recursive functions.

Note that we have intentionally defined `rotate` on the most general type `int[*]`, which includes zero-dimensional arrays aka scalars. In this case, the FOR-loop is executed zero times, and `rotate` degenerates to the identity function.

The first major step in the compilation of SAC programs is a rigorous type inference and specialization scheme. Type specifications are assigned to each expression, and type declarations for local variables are inserted where missing. As for the hierarchy of array types, the inference scheme aims at identifying types as specific as possible, preferably ones with complete shape information. Step by step, rank- and shape-invariant program specifications are transformed into collections of functions tailor-made for specific problem sizes.

After type inference, various conventional optimization techniques [2] such as function inlining, constant folding, constant propagation, loop unrolling, and dead code removal are repeatedly applied to the type- and shape-annotated code. Fig. 4 shows a code fragment of an application of the given `rotate` function to a constant offset vector `[-1,2]` and a 2-dimensional array `A` with 50 rows and 80 columns (`int[50,80]`) as it looks like after conventional optimizations. To maintain some resemblance to the function definition, dead code is displayed in the shaded grey areas rather than being removed. The FOR-loop in the original definition is unrolled, and boundary vectors are replaced by constants computed at compile time. The sequence of four WITH-loops is all that remains after dead code removal.

Despite this substantial simplification the derived code is still rather inefficient. A naive compilation would create three temporary arrays before eventually generating the function result. As a consequence, individual elements are copied up to four times before reaching the final index position. This situation may be considered typical for intermediate code derived from APL-like specifications. To address this issue, a SAC-specific optimization technique called WITH-LOOP-FOLDING [22] aims at eliminating costly creation of temporary arrays by condensing subsequent

```
...
max_rotate = 50;
offset = -1;
offset = 49;

lower_bound = [49, 0];
upper_bound = [49,80];

B = with ([0,0] <= iv < [49,80])
    genarray( [50,80], A[ iv + [1,0]]);

C = with ([49,0] <= iv < [50,80])
    modarray( B, A[ iv - [49,0]]);

A = C;

max_rotate = 80;
offset = 2;

lower_bound = [ 0,2];
upper_bound = [50,2];

B = with ([0,0] <= iv < [50,2])
    genarray( [50,80], C[ iv + [0,78]]);

C = with ([0,2] <= iv < [50,80])
    modarray( B, C[ iv - [0,2]]);
...
```

Figure 4: `rotate( [-1,2], int[50,80] A)` after general optimizations.

WITH-loops into a single though more general variant. Fig. 5 illustrates the effect of WITH-LOOP-FOLDING on the running example.

WITH-LOOP-FOLDING results in a compiler internal variant of WITH-loops called *multigenerator* WITH-loop. They provide a complete partition of the target array defined by a set of generators each associated with an explicit element specification. In the running example, the four ordinary WITH-loops can be condensed into a single multigenerator WITH-loop with four generators, as shown in Fig. 5. All four operations attached to these index ranges are selections into the original array `A`. They differ only by the offsets added to the running index `iv`.

```
...
C = with (iv)
      [ 0, 0] <= iv < [49, 2] : A[ iv + [  1, 78]];
      [ 0, 2] <= iv < [49,80] : A[ iv + [  1, -2]];
      [49, 0] <= iv < [50, 2] : A[ iv + [-49, 78]];
      [49, 2] <= iv < [50,80] : A[ iv + [-49, -2]];
    genarray( [50,80]);
...
```

Figure 5: `rotate( [-1,2], int[50,80] A)` after WITH-LOOP-FOLDING.

As the compiler ensures that all index sets are mutually disjoint, in the final code generation phase arbitrary traversal orders through the array elements can be chosen. Fig. 6 sketches out the nestings of FOR-loops that are generated for the running example. A closer examination of the loops shows that the generators are not compiled individually. Instead, loop nestings are generated that linearly traverse the associated memory for improved locality of array references.

As pointed out before, the SAC compiler supports generation of multithreaded code for parallel execution on shared memory systems. The code derived for the

```
...
C = ALLOCATE_ARRAY( [50,80], int);

for( iv0=0; iv0<49; iv0++) {
  for( iv1=0; iv1< 2; iv1++)  C[iv0,iv1] = A[iv0+1, iv1+78];
  for( iv1=2; iv1<80; iv1++)  C[iv0,iv1] = A[iv0+1, iv1- 2];
}

for( iv0=49; iv0<50; iv0++) {
  for( iv1=0; iv1< 2; iv1++)  C[iv0,iv1] = A[iv0-49, iv1+78];
  for( iv1=2; iv1<80; iv1++)  C[iv0,iv1] = A[iv0-49, iv1- 2];
}
...
```

Figure 6: `rotate( [-1,2], int[50,80] A)` after code generation.

`rotate` example is outlined in Fig. 7. Whereas the initial allocation of memory for the result array may be adopted from the sequential code generation scheme, the iteration space traversed by the following loop nestings needs to be partitioned into several disjoint subspaces, one for each thread. The pseudo statement `MT_EXECUTION` specifies that the following code block may be executed by multiple threads. Their exact number is given by the runtime constant `#THREADS`; individual threads are identified by the variable `tid`.

```
...
C = ALLOCATE_ARRAY( [50,80], int);

MT_EXECUTION( 0 <= tid < #THREADS) {
  do {
    sb0, se0, sb1, se1, cont
      = SCHEDULE( tid, #THREADS, shape(C), STRATEGY);

    for( iv0=max(0,sb0); iv0<min(49,se0); iv0++) {
      for( iv1=max(0,sb1); iv1<min(2,se1); iv1++)
        C[iv0,iv1] = A[iv0+1, iv1+78];
      for( iv1=max(2,sb1); iv1<min(80,se1); iv1++)
        C[iv0,iv1] = A[iv0+1, iv1-2];
    }

    for( iv0=max(49,sb0); iv0<min(50,se0); iv0++) {
      for( iv1=max(0,sb1); iv1<min(2,se1); iv1++)
        C[iv0,iv1] = A[iv0-49, iv1+78];
      for( iv1=max(2,sb1); iv1<min(80,se1); iv1++)
        C[iv0,iv1] = A[iv0-49, iv1-2];
    }
  } while (cont);
}
...
```

Figure 7: `rotate( [-1,2], int[50,80] A)` after multithreading.

Since parallelization of code and transformation of multigenerator WITH-loops into potentially complex nestings of FOR-loops are to some extent orthogonal issues, we aim at reusing existing sequential compilation technology as far as possible. This is achieved by introducing a separate loop scheduler `SCHEDULE`, which based on the total number of threads and individual thread identifiers computes disjoint rectangular subranges of the iteration space. The original loop nesting is modified only insofar as each loop is restricted to the intersection between its original range and the current iteration subspace defined by the scheduler. As indicated by the

additional scheduler argument `STRATEGY`, this design offers the opportunity to plug-in different scheduling techniques including dynamic load balancing schemes.

4. Case study: PDE1 benchmark

This section illustrates the highly generic programming style encouraged by SaC and quantifies the effectiveness of the compilation techniques introduced in the previous section. The PDE1 benchmark approximates solutions to 3-dimensional discrete Poisson equations by means of red/black successive over-relaxation. It has previously been studied in compiler performance comparisons in the context of Hpf and Zpl [20,3]; reference implementations are available for various languages.

```
double[+] RelaxKernel( double[+] u, double[+] weights)
{
  res = with ( 0*shape(weights) <= iv < shape(weights))
        fold( +, weights[iv] * rotate( 1-iv, u));

  return( res);
}

double[+] PDE1( int iter, double hsq, double[+] f, double[+] u,
                bool[+] red, bool[+] black, double[+] weights)
{
  for (nrel=1; nrel<=iter; nrel+=1) {
    u = where(   red, hsq * f + RelaxKernel( u, weights), u);
    u = where( black, hsq * f + RelaxKernel( u, weights), u);
  }

  return( u);
}
```

Figure 8: SaC implementation of PDE1 benchmark.

Fig. 8 shows a highly generic SaC implementation of PDE1, built in two layers on top of multi-axis rotation, as discussed in the previous section. On the first layer a completely benchmark-independent auxiliary function `RelaxKernel` realizes a single relaxation step on the argument array `u`, which may be of any rank and size. Relaxation is parameterized over a stencil description given by an array of weights. The key idea is to sum up entire arrays that are derived from the argument array `u` by rotation into as many directions as given by the rank of the array of weights and multiplication with the corresponding weight coefficient.

In the case of PDE1, relaxation is actually performed on 3-dimensional arrays using a 6-point stencil with identical weights for all six direct neighbors. Therefore, the (constant) array of weights is chosen as follows:

```
[ [ [ 0d,    0d, 0d], [    0d, 1d/6d,    0d], [ 0d,    0d, 0d] ],
  [ [ 0d, 1d/6d, 0d], [ 1d/6d,    0d, 1d/6d], [ 0d, 1d/6d, 0d] ],
  [ [ 0d,    0d, 0d], [    0d, 1d/6d,    0d], [ 0d,    0d, 0d] ] ]    .
```

Governed by the shape of this array, i.e. `[3,3,3]`, the argument array is rotated into all 27 possible directions and multiplied by the corresponding elements. However, as most of the weights turn out to be zero, only six multiplications and five additions are actually performed after optimization.

The alert reader may observe that using `rotate` as basis for `RelaxKernel` implicitly implements periodic boundary conditions. As PDE1 requires fixed boundary conditions, the application of `RelaxKernel` must be restricted to the inner elements

of the array. This is achieved by embedding the relaxation step into the `where` function from the SAC standard library. It takes a boolean mask and two arrays of equal shapes as arguments and yields an identically-shaped array as result. Depending on individual mask elements it either selects the corresponding element of the second argument array (`true`) or that of the third one (`false`). This flexibility allows the restriction to inner elements to be combined with the benchmark requirement to restrict relaxation alternatingly to two different sets of elements, the `red` set and the `black` set. These two applications of `where` are finally embedded into a simple `for`-loop realizing the desired number of iterations, which makes up the entire body of the function `PDE1`.

Experiments with respect to the runtime performance of this SAC implementation of PDE1 have been made on a 12-processor SUN Ultra Enterprise 4000. They are compared with the outcomes of similar experiments involving an HPF implementation compiled by the ADAPTOR HPF-compiler v7.0 [6] using PVM as communication layer and a ZPL implementation using `zc` v1.15.7 and MPICH. Both the HPF as well as the ZPL implementation of the PDE1 benchmark are taken from the demo codes that come with the corresponding compiler distributions. In order to compare all three codes on a reasonably fair basis startup overhead is eliminated by statistical measures.

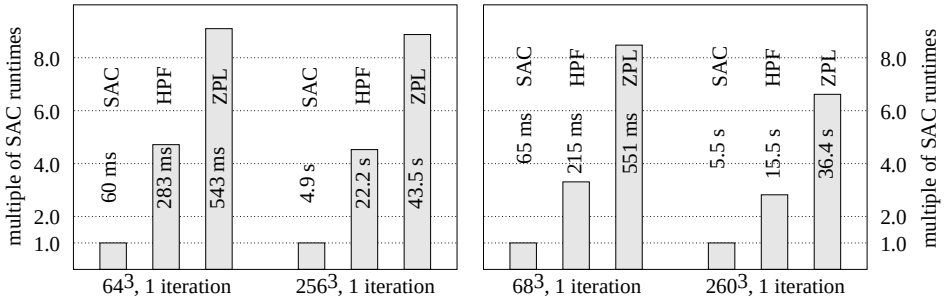

Figure 9: Single processor runtime performance.

Fig. 9 (left) shows sequential runtimes for two different problem sizes: 64^3 and 256^3 grid elements. It turns out that despite the high-level generic approach characterizing the SAC implementation it clearly outperforms both the HPF and the ZPL implementation. Multiprocessor performance achieved by all three candidates is shown in Fig. 10 (top) relative to sequential execution of SAC code. While SAC achieves a maximum speedup of about eight using ten processors for both problem sizes, HPF and ZPL also achieve substantial gains by parallel execution, but suffer from their inferior single processor performance.

Being powers of two, both benchmarking problem sizes may produce cache effects which could render the findings unrepresentative in general. To eliminate the potential for such effects we have repeated all experiments with two slightly different problem sizes which are unlikely to be subject to cache thrashing. Sequential and parallel performance figures for grid sizes of 68^3 and 260^3 elements are given in Fig. 9 (right) and in Fig. 10 (bottom), respectively.

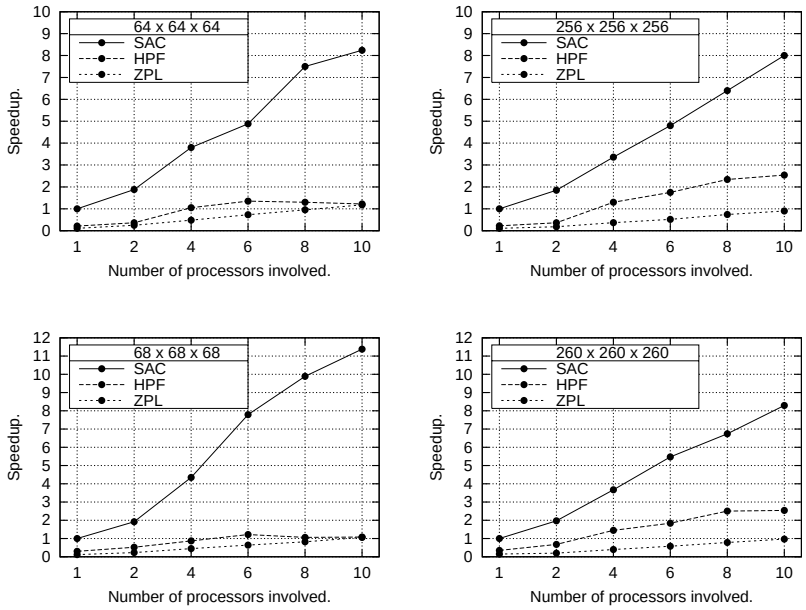

Figure 10: Multiprocessor runtime performance.

5. Related work

Although throughout this paper comparisons with Fortran-90/Hpf and Zpl served both as a motivation for the design of SaC and as a baseline for performance evaluation, there is more related work to mention than this. For example, interpreted array languages, notably Apl and J, also provide a very generic approach to array programming. However, their runtime performance characteristics are usually considered prohibitive as soon as performance really matters. Although their suitability for parallelization has been discussed in principle [4], parallel implementations have not evolved.

In the field of functional programming Sisal [7] used to be the most prominent array language. It offers high-level array handling free of side-effects as well as implicit memory management. Aggregate array operations are defined by means of Sisal-specific array comprehensions. However, Sisal supports only vectors, while higher-dimensional arrays must be represented as nested vectors of equal length. Moreover, Sisal neither provides built-in high-level aggregate operations nor means to define such general abstractions in the language, as SaC does. The design of Sisal90 [12] promises improvements, but has not been implemented.

Another approach to high-level array programming is FISh [17]. Characteristic for FISh is the distinction between two categories of expressions: *shapely expressions* which define shapes of arrays and regular expressions that define element values. While the latter are evaluated at runtime, shapely expressions must be evaluated at compile time to ensure static knowledge of all array shapes. This distinction restricts FISh to *uniform functions* [16], i.e. functions whose result shapes

can entirely be derived from the shapes of their arguments. This restriction rules out definition of many basic array operations, e.g. `take` or `drop`. Moreover, to the best knowledge of the authors no parallel version of FISH has been realized yet.

A popular approach to high-level parallel programming can be found in *algorithmic skeletons* [10]. They aim at encapsulating common patterns of parallel execution in a fixed set of abstractions, usually higher-order functions. Many different flavors of this approach have been proposed, reaching from tailor-made (imperative) languages like P3L[11] or SKIL [5] to libraries both for functional [15] as well as for imperative host languages [18,21,19]. Although WITH-loops in SAC share some similarities with data-parallel skeletons, their intention is quite different. Skeletons provide abstract means to specify the parallel aspects of program execution; they do not increase the level of abstraction in array processing in general. In contrast, WITH-loops are abstractions that provide the basis for high-level array processing regardless of whether program execution is sequential or parallel.

6. Conclusion

The major design goal of SAC is to combine highly generic specifications of array operations with compilation techniques for generating efficiently executable multi-threaded code. This paper illustrates the major steps in the compilation process by means of a basic SAC-implemented array operation from the standard library: a rank- and shape-invariant specification of multi-axis rotation. The effectiveness of the measures described is investigated by means of a highly generic SAC implementation of the PDE1 benchmark based on multi-axis rotation. Despite its high level of abstraction the SAC implementation substantially outperforms benchmark implementations in the data parallel languages HPF and ZPL both in sequential and in parallel execution. This shows that high-level generic program specifications and good runtime performance — even relative to conventional low-level approaches — not necessarily exclude each other.

References

[1] J.C. Adams, W.S. Brainerd, et al. *Fortran-95 Handbook — Complete ANSI/ISO Reference*. Scientific and Engineering Computation. MIT Press, 1997.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[3] Applied Parallel Research, Inc. xHPF Benchmark Results. Technical report, 1995.

[4] R. Bernecky. The Role of APL and J in High-Performance Computation. *APL Quote Quad*, vol. 24(1), pp. 17–32, 1993.

[5] G.H. Botorog and H. Kuchen. Efficient High-Level Parallel Programming. *Theoretical Computer Science*, vol. 196(1-2), pp. 71–107, 1998.

[6] T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In *Programming Environments for Massively Parallel Distributed Systems*, pp. 91–96. Birkhäuser Verlag, 1994.

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

[8] B.L. Chamberlain, S.-E. Choi, et al. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Transactions on Software Engineering*,

vol. 26(3), pp. 197–211, 2000.

[9] B.L. Chamberlain et al. Regions: An Abstraction for Expressing Array Computation. In O. Levefre, ed., *Proceedings of the International Conference on Array Processing Languages (APL'99)*, *APL Quote Quad*, vol. 29(1), pp. 41–49, 1999.

[10] M.I. Cole. Algorithmic Skeletons. In K. Hammond and G. Michaelson, eds., *Research Directions in Parallel Functional Programming*, pp. 289–303. Springer-Verlag, 1999.

[11] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P3L. In C. Lengauer, M. Griebl, and S. Gorlatch, eds., *Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97)*, *LNCS*, vol. 1300, pp. 619–628. Springer-Verlag, 1997.

[12] J.T. Feo, P.J. Miller, et al. Sisal 90. In A.P.W. Böhm and J.T. Feo, eds., *Proceedings of the Conference on High Performance Functional Computing (HPFC'95)*, pp. 35–47. Lawrence Livermore National Laboratory, 1995.

[13] C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Computer Society Press, 2002.

[14] C. Grelck. A Multithreaded Compiler Backend for High-Level Array Programming. In M.H. Hamza, ed., *Proceedings of the 21st International Multi-Conference on Applied Informatics (AI'03), Part II: International Conference on Parallel and Distributed Computing and Networks (PDCN'03)*, pp. 478–484. ACTA Press, 2003.

[15] K. Hammond and A.J. Rebón Portillo. HaskSkel: Algorithmic Skeletons in Haskell. In C. Clack and P. Koopman, eds., *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'00), selected papers*, *LNCS*, vol. 1868, pp. 181–198. Springer-Verlag, 2000.

[16] R.K.W. Hui. Rank and Uniformity. In *Proceedings of the International Conference on Array Processing Languages (APL'95)*, pp. 83–90. ACM Press, 1995.

[17] C.B. Jay and P.A. Steckler. The Functional Imperative: Shape! In C. Hankin, ed., *Proceedings of the 7th European Symposium on Programming (ESOP'98)*, *LNCS*, vol. 1381, pp. 139–53. Springer-Verlag, 1998.

[18] E. Johnson and D. Gannon. Programming with the HPC++ Parallel Standard Template Library. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (PP'97), Minneapolis, Minnesota, USA*, 1997.

[19] H. Kuchen. A Skeleton Library. In B. Monien and R. Feldmann, eds., *Proceedings of the 8th European Conference on Parallel Processing (Euro-Par'02)*, *LNCS*, vol. 2400, pp. 620–629. Springer-Verlag, 2002.

[20] C. Lin, L. Snyder, et al. ZPL vs. HPF: A Comparison of Performance and Programming Style. Technical Report TR-95-11-05, Department of Computer Science and Engineering, University of Washington, 1995.

[21] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Library. In D.R. O'Hallaron, ed., *Proceedings of the 4th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'98), selected papers*, *LNCS*, vol. 1511, pp. 402–410. Springer-Verlag, 1998.

[22] 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 (IFL'97), selected papers*, *LNCS*, vol. 1467, pp. 72–92. Springer-Verlag, 1998.

[23] S.-B. Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*, accepted for publication.