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
Olshausenstr. 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 is basically about finding the best possible trade-off between support for high-level program specifications and efficient runtime behavior. In the context of array programming, the approach taken by the so-called *data parallel languages* seems to be well suited to meet this goal: using functions that operate on entire arrays rather than loop nestings on individual elements does not only improve program specification but also provides opportunities for compilers to generate parallel code.

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 so-called *triple-notation*. Unfortunately, it has some drawbacks as well: pro-

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

The programming language ZPL [4] offers a more elegant solution for this problem by introducing *regions* [5]. 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. Due 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 (for Single Assignment C) [14] takes the idea of high-level generic array programming even one step further. 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-*loop expressions* (or WITH-loops for short) allow 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 on 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. At the time being the SAC compiler on demand generates multithreaded code for parallel execution on shared memory multiprocessors [6,9]. 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 to be highly competitive with other approaches [12,8].

This paper outlines the major steps in compiling rank- and shape-invariant high-level SAC programs into efficiently executable multithreaded code. For illustration purposes, we refrain from giving exact compilation schemes and employ a rather simple example instead: rotation of an array along multiple axes. Starting out from a high-level, generic rank- and shape-invariant SAC specification taken from the standard array library, effects of major compilation phases towards efficiently executable C code are identified in a step-by-step process.

Advantages of the generic programming model encouraged by SAC and the effectiveness of the associated compilation technique in order to still achieve competitive performance characteristics are demonstrated by means of a small case study. The PDE1 benchmark realizing 3-dimensional red/black successive over-relaxation

is implemented as two layes on top of multi-axis rotation. While the first layer consists of a completely benchmark-independent generic relaxation kernel, all code specific to the PDE1 benchmark is added as a second layer. Despite this generic approach experiments show that SAC manages to substantially outperform more conventional benchmark implementations in HPF and in ZPL both sequentially as well as in parallel.

The paper is organized as follows. Following a brief introduction to SAC in Section , the compilation process is sketched out in Section . Section describes the small case study involving the PDE1 benchmark while Section 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 techniques. 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, last but not least, 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 dimension (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`).

WithLoopExpr	⇒	**with** (*Generator*) *Operation*
Generator	⇒	*Expr Relop Identifier Relop Expr* [*Filter*]
Relop	⇒	< | <=
Filter	⇒	**step** *Expr* [**width** *Expr*]
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 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 vectors of equal length, define lower and upper bounds of a rectangular index vector range. An optional filter may further restrict this selection to grids of arbitrary width. Let a, b, s, and w denote expressions that evaluate to vectors of length n, then

`(` a `<= i_vec <` b `step` s `width` w `)`

defines the following set of index vectors:

$$\{i\_vec \mid \forall_{j \in \{0,\ldots,n-1\}} : a_j \leq i\_vec_j < b_j \ \wedge \ (i\_vec_j - a_j) \mod s_j < w_j\}$$

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 a 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 dimension and size allows for implementing APL-like compound array operations in SAC itself. This feature is exploited by the SAC array 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 program specifications into efficiently executable multithreaded code. For illustration purposes, we focus on a single example rather than providing complete compilation schemes. Consider for example the function

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

,

which rotates the given integer array `A` in each dimension by as many elements as are specified by the corresponding element of the vector `offsets`. Note that the function `rotate` is applicable to arrays of any rank and, hence, the first argument `offsets` may be a vector of varying length.

The implementation of `rotate`, as it may be found in the SAC array library, is shown in Fig. 2. Rotation along multiple axes is realized as a sequence of rotations along a single axis. 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 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

```
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.

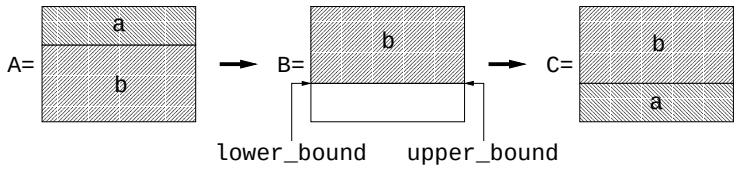

Figure 3: Illustration of rotation along single axis.

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*.

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. Fig. 4 shows a specialized version for rotating 2-dimensional arrays with 50 rows and 80 columns (`int[50,80]`).

After type inference, various conventional optimization techniques [1] such as function inlining, constant folding, constant propagation, loop unrolling, or dead code removal are repeatedly applied to the type- and shape-annotated code. Assuming an application of the specialized version of `rotate` to a constant offset

```
int[*] rotate (int[2] offsets, int[50,80] A)
{
  int[50,80] B, C;
  int[2]     iv, upper_bound, lower_bound;
  int        i, offset, max_rotate;

  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 4: Function `rotate` after type specialization.

vector `[-1,2]` and an array `A`, i.e, a rotation of `A` upwards by one row and to the right by two columns, these optimizations yield a code fragment similar to the one shown in Fig. 5. To maintain some resemblance with preceding versions, dead code is displayed in the shaded grey areas rather than being removed.

Exact shape information allows the compiler to replace applications of the built-in functions `dim` and `shape` by the corresponding values. This creates many additional opportunities for constant propagation and folding. Moreover, knowledge of the rank of `A` and of the shape of `offsets` determines the number of iterations of the FOR-loop and, hence, allows its unrolling. As a consequence, the loop variable `i` becomes constant in both iterations, enabling further optimization. Eventually, all but the four WITH-loops can be eliminated as dead code.

Although this implementation avoids expensive modulo operations in index calculations, it is still far from being efficient. A naive compilation would lead to the creation of three temporary arrays before the function result itself is computed. The creation of many temporary arrays has several serious drawbacks. The predominant problem is the increase in memory accesses that are required. As for most modern computer architectures intensive memory accesses are by far the most performance limiting operations, the ratio between memory accesses and value transforming operations should be kept as low as possible.

Other problems occur at the compilation into concurrently executable code. The data dependencies between the various temporary arrays increase the demand for synchronization. As a consequence, the granularity of tasks is decreased which has a direct impact on the overall performance achieved.

```
...

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 5: `rotate( [-1,2], A)` after general optimizations.

To avoid all these problems, a SAC-specific optimization technique called WITH-LOOP-FOLDING [13] aims at eliminating costly creation of temporary arrays by condensing subsequent WITH-loops into a single though more general variant. Fig. 6 illustrates the effect of WITH-LOOP-FOLDING on the running example.

```
...

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 6: `rotate( [-1,2], A)` after WITH-LOOP-FOLDING.

WITH-LOOP-FOLDING results in a special (compiler internal) form of WITH-loops called *multigenerator* WITH-loop. Rather than specifying a single generator and a single associated element definition with an implicit default specification for all elements not covered by the generator, multigenerator WITH-loops consist of a complete partition of the target array defined by a set of generators each with associated

explicit element specification. For the 2-dimensional rotation example, the four ordinary WITH-loops can be condensed into a single multigenerator WITH-loop with four sets of index vectors. All four operations attached to these index ranges are selections into the original array `A`. They only differ by the offsets added to the running index `iv`.

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. This flexibility allows cache or scheduling considerations to be taken into account (cf. [10]).

Fig. 7 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. Code shown in Fig. 7 is simplified in various aspects to maintain readability, e.g., reference counting instructions for implicit memory management are omitted, and array references are still written in SAC style.

```
int[50,80] rotate (int[2] offsets=[-1,2], int[50,80] A)
{
  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+48];
    }
    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+48];
    }
    for( iv1=2; iv1<80; iv1++) {
      C[iv0,iv1] = A[iv0-49, iv1-2];
    }
  }
  return( C);
}
```

Figure 7: `rotate( [-1,2], A)` after code generation.

As pointed out earlier, the SAC compiler also supports generation of multi-threaded code for parallel execution on shared memory systems. The code generated for the `rotate` example is outlined in Fig. 8. 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`.

Since parallelization of code and transformation of multigenerator WITH-loops into potentially complex nestings of FOR-loops are to some extent orthogonal issues,

```
int[50,80] rotate (int[2] offsets=[-1,2], int[50,80] A)
{
  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+48];
        }
        for( iv1=max(2,sb1); iv1<min(80,se1); iv1++) {
          C[iv0,iv1] = A[iv0+1, iv1-2];
        }
      }
      for( iv0=max(49; iv0<min(50,se0); iv0++) {
        for( iv1=max(0,sb1); iv1<min(2,se1); iv1++) {
          C[iv0,iv1] = A[iv0-49, iv1+48];
        }
        for( iv1=max(2,sb1); iv1<min(80,se1); iv1++) {
          C[iv0,iv1] = A[iv0-49, iv1-2];
        }
      }
    }
    while (cont);
  }
  return(C);
}
```

Figure 8: `rotate( [-1,2], A)` after multithreading.

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 identifyers computes disjoint rectangular subranges of the iteration space. The original loop nesting is only modified insofar as each loop is restricted to the intersection of its original range and the 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. A more detailed description of compilation to multithreaded target code as well as the associated runtime system may be found in [6,9].

4. Performance evaluation

This section is to investigate and to quantify the performance impact of the highly generic programming style encouraged by SaC in general and the effectiveness of the compilation techniques introduced in the previous section in particular. Since a simple `rotate` operation, as used in the previous section for illustration purposes can hardly be considered a representative operation by itself, we choose the PDE1 benchmark instead, which 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

[11,2], and 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 9: SAC implementation of PDE1 benchmark.

Fig. 9 shows a SAC implementation of PDE1. It is based on a highly generic auxiliary function `RelaxKernel`, which is designed to be applicable in various contexts where relaxation is required. It realizes a single relaxation step on the argument array u, which may be of any rank and of any size. The operation is parameterized over a stencil description specified by an array of weights which implicitly steers the summation of weighted neighbor elements. Using the rotate function discussed in the previous section this operation can be specified elegantly without any explicit element selections. The key idea is to sum up entire arrays, that are derived from the original array by rotation and multiplication with the respective component of the array of weights.

In case of PDE1, relaxation is performed on a 3-dimensional array using a 6-point stencil with identical weights for all six direct neighbors. Therefore, the (constant) array of weights has to be 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 in all 27 possible directions and multiplied by the corresponding weight coefficient. However, as most of the weights turn out to be zero, after optimization, only 5 additions remain.

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` has to 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 identical-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 finaly 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 [3] using PVM as communication layer and 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 corresponding compiler's demo codes. In order to compare all three codes on a reasonably fair basis despite the usage of different underlying communication layers startup overhead is eliminated by statistical measures.

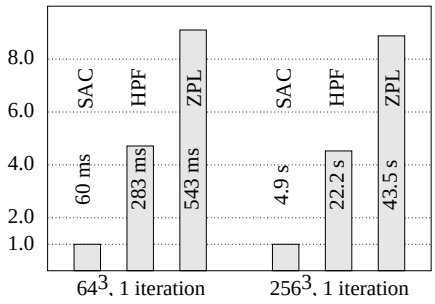

Figure 10: Single processor runtime performance.

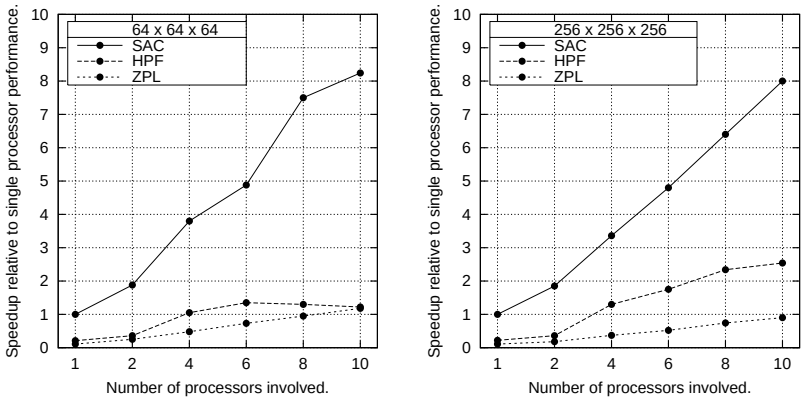

Figure 11: Multiprocessor runtime performance.

Fig. 10 shows sequential runtimes for two different problem sizes: 64^3 and 256^3 grid elements, respectively. 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. 11 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 substantially 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. Whereas the SAC compiler addresses the issue of cache effects by specific code optimizations [7,6], the other compilers may or may not have similar capabilities. To quantify these 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. 12 and in Fig. 13, respectively.

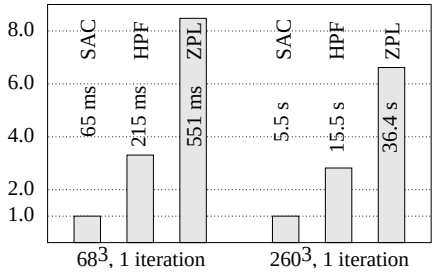

Figure 12: Single processor runtime performance.

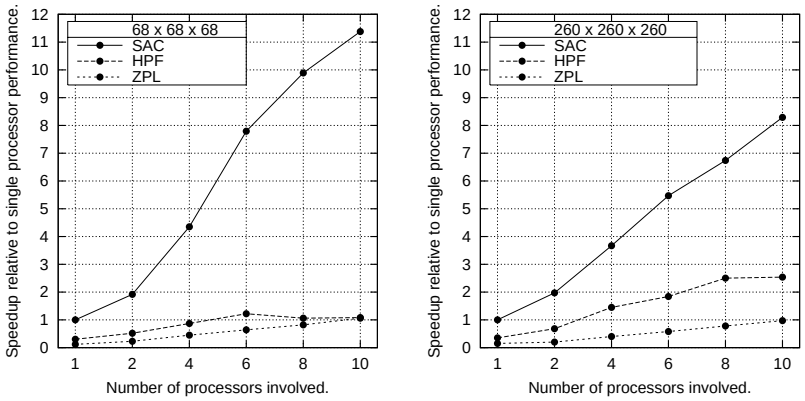

Figure 13: Multiprocessor runtime performance.

Whereas sequential SAC runtimes increase more or less proportionally to the increase in grid elements, substantial performance gains can in fact be observed for HPF and for ZPL. However, SAC still outperforms both of them by factors. Hence, cache effects play some role in the initial experiments but not the dominant one.

With respect to parallel performance, as shown in Fig. 11, it can be observed that

SAC even achieves superlinear speedups for the problem size 68^3 while speedups for 260^3 grid elements are identical to what is observed for 256^3 elements. In contrast, scalability of both the HPF and the ZPL implementation is less than before. Although they start from a better sequential base line relative to SAC maximum speedups are about the same. This observation can be explained by the fact that unlike SAC they both use distributed memory models as underlying communication layers. Consequently, memory layouts are different for each number of processors and, hence, some of the cache problems are solved incidentally by employing increasing numbers of processors. This effect improves scalability in the initial experiments. In contrast, the SAC implementation uses the same memory layout for any number of processors and, thus, does not show similar effects.

5. 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 implementation 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 not necessarily exclude each other.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0.

[2] Inc. Applied Parallel Research. xhpf benchmark results. Technical report, Applied Parallel Research, Inc., Roseville, CA, 1995.

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

[4] B.L. Chamberlain, S.-E. Choi, E.C. Lewis, C. Lin, L. Snyder, and W.D. Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000. Special Issue on Architecture-Independent Languages and Software Tools for Parallel Processing.

[5] B.L. Chamberlain, E.C. Lewis, C. Lin, and L. Snyder. Regions: An abstraction for expressing array computation. In O. Levefre, editor, *Prceedings of the International Conference on Array Processing Languages (APL'99), Scranton, Pennsylvania, USA*, volume 29-1 of *APL Quote Quad*, pages 41–49. ACM Press, 1999.

[6] C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC - Single Assignment C*. PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2001.

[7] C. Grelck. Improving Cache Effectiveness through Array Data Layout in SAC. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL'00), Aachen, Germany,*

selected papers, volume 2011 of *Lecture Notes in Computer Science*, pages 231–248. Springer-Verlag, Berlin, Germany, 2001.

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

[9] C. Grelck. A Multithreaded Compiler Backend for High-Level Array Programming. In M.H. Hamza, editor, *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), Innsbruck, Austria*. ACTA Press, 2003.

[10] C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proc. of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, Selected Papers*, volume 1868 of *LNCS*, pages 77–95. Springer, 2000.

[11] C. Lin, L. Snyder, R.E. Anderson, et al. ZPL vs HPF: A Comparison of Performance and Programming Style. Technical Report 95-11-5, University of Washington, 1995.

[12] 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.

[13] S.-B. Scholz. With-loop-folding in SAC–Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.

[14] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 2002. Accepted for publication.