Generic Array Programming in SAC

Clemens Grelck[1] and Sven-Bodo Scholz[2]

[1] University of Lübeck, Germany
Institute of Software Technology and Programming Languages
grelck@isp.uni-luebeck.de
[2] University of Hertfordshire, United Kingdom
Department of Computer Science
s.scholz@herts.ac.uk

Abstract. SaC is a purely functional array processing language designed with compute-intensive numerical applications in mind. The declarative, generic style of programming in SaC is demonstrated by means of a small case study: 3-dimensional complex fast-Fourier transforms. The impact of abstraction on expressiveness, readability, and maintainability of code as well as on clarity of underlying mathematical concepts is discussed and compared with other approaches. The associated impact on runtime performance is quantified both in uniprocessor and in multiprocessor environments.

1 Introduction

Functional languages are generally considered well-suited for parallelization. Program execution is based on the principle of context-free substitution of expressions. Programs are free of side-effects and adhere to the Church-Rosser property. Any two subexpressions without data dependencies can be executed in parallel without any further analysis.

Classical domains of parallel computing like image processing or computational sciences are characterized by large arrays of numerical data [1]. Unfortunately, almost all functional languages focus on lists and trees, not on arrays. Notational support for multi-dimensional arrays is often rudimentary. Even worse, sequential runtime performance in terms of memory consumption and execution times fails to meet the requirements of numerical applications [2–4].

SaC (Single Assignment C) [5] is a purely functional array language. Its design aims at combining generic, high-level array processing with a runtime performance that is competitive with low-level machine-oriented programs written in C or Fortran. The core syntax of SaC is a subset of C with a strict, purely functional semantics based on context-free substitution of expressions. Nevertheless, the meaning of functional SaC code coincides with the state-based semantics of literally identical C code. This design is meant to facilitate conversion to SaC for programmers with a background in imperative languages.

The language kernel of SaC is extended by multi-dimensional, stateless arrays. In contrast to other array languages, SaC provides only a very small set of

built-in operations on arrays, mostly primitives to retrieve data pertaining to the structure and contents of arrays. All aggregate array operations are specified in SaC itself using a versatile and powerful array comprehension construct, named WITH-*loop*. WITH-loops allow code to abstract not only from concrete shapes of argument arrays, but even from concrete ranks (number of axes or number of dimensions) . Moreover, such rank-invariant specifications can be embedded within functions, which are applicable to arrays of any rank and shape.

By these means, most built-in operations known from Fortran-95 or from interpreted array languages like Apl, J, or Nial can be implemented in SaC itself without loss of generality [6]. SaC provides a comprehensive selection of array operations in the standard library. In contrast to array support which is hard-wired into the compiler, our library-based solution is easier to maintain, to extend, and to customize for varying requirements.

SaC propagates a programming methodology based on the principles of abstraction and composition. Like in Apl, complex array operations and entire application programs are constructed by composition of simpler and more general operations in multiple layers of abstractions. Unlike Apl, the most basic building blocks of this hierarchy of abstractions are implemented by WITH-loops, not built-in. Whenever a basic operation is found to be missing during program development, it can easily be added to the repertoire and reused in future projects.

Various case studies have shown that despite a generic style of programming SaC code is able to achieve runtime performance figures that are competitive with low-level, machine-oriented languages [7, 8, 5, 9]. We achieve this runtime behaviour by the consequent application of standard compiler optimizations in conjunction with a number of tailor-made array optimizations. They restructure code from a representation amenable to programmers and maintenance towards a representation suitable for efficient execution by machines [10, 5, 9, 11]. Fully compiler-directed parallelization techniques for shared memory architectures [12–14] further enhance performance. Utilization of a few additional processing resources often allow SaC programs to outperform even hand-optimized imperative codes without any additional programming effort.

The rest of the paper is organized as follows. Section 2 gives a short introduction to SaC, while Section 3 further elaborates on programming methodology. Section 4 applies the techniques to a well-known benchmark: 3-dimensional complex FFT. Section 5 provides a quantitative analysis, while Section 6 draws conclusions and outlines directions of future work.

2 SAC — Single Assignment C

Essentially, SaC is a functional subset of C extended by multi-dimensional stateless arrays as first class objects. Arrays in SaC are represented by two vectors. The *shape vector* specifies an array's rank and the number of elements along each axis. The *data vector* contains all elements of an array in row-major order. 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 encom-

passing both arrays of any rank and scalars: `int[*]`. The hierarchy of array types induces a subtype relationship. SAC supports function overloading both with respect to different base types and with respect to the subtype relationship.

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*`]`). Compound array operations are specified using WITH-loop expressions. As defined in Fig. 1, a WITH-loop basically consists of three parts: a *generator*, an *associated expression* and an *operation*.

WithLoopExpr	⇒	**with** *Generator* **:** *Expr Operation*
Generator	⇒	**(** *Expr Relop Identifier Relop Expr* [*Filter*] **)**
Relop	⇒	`<=` | `<`
Operation	⇒	**genarray** **(** *Expr* [**,** *Expr*] **)**
	|	**fold** **(** *FoldOp* **,** *Expr* **)**

Fig. 1. Syntax of with-loop expressions.

The operation determines the overall meaning of the WITH-loop. There are two variants: `genarray` and `fold`. With `genarray(` ***shp***`,` ***default*** `)` the WITH-loop creates a new array. The expression ***shp*** must evaluate to an integer vector, which defines the shape of the array to be created. With `fold(` ***foldop***`,` ***neutral*** `)` the WITH-loop specifies a reduction operation. In this case, ***foldop*** must be the name of an appropriate associative and commutative binary operation with neutral element specified by the expression ***neutral***.

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. An optional filter may be used to further restrict generators to various kinds of grids; for simplification we omit this detail in the following.

For each element of the set of index vectors defined by the generator the associated expression is evaluated. Depending on the variant of WITH-loop, the resulting value is either used to initialize the corresponding element position of the array to be created (`genarray`), or it is given as an argument to the fold operation (`fold`). In the case of a `genarray`-WITH-loop, elements of the result array that are not covered by the generator are initialized by the (optional) default expression in the operation part. For example, the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1]
genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0\,0\,0\,0\,0 \\ 0\,2\,3\,4\,0 \\ 0\,3\,4\,5\,0 \end{pmatrix}$ while the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1]
fold( +, 0)
```

evaluates to 21. More information on SAC is available at `www.sac-home.org`.

3 Programming methodology

As pointed out in the introduction, SAC propagates a programming methodology based on the principles of abstraction and composition. 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 us to implement generic compound array operations in SAC itself.

```
double[+] abs( double[+] a)
{
  res = with (. <= iv < shape(a)) : abs(a[iv])
        genarray( shape(a));
  return( res)
}

bool[+] (>=) ( double[+] a, double[+] b)
{
  res = with  (. <= iv <= .) : a[iv] >= b[iv]
        genarray( min( shape(a), shape(b)));
  return( res)
}

bool any( bool[+] a)
{
  res = with (0*shape(a) <= iv < shape(a)) : a[iv]
        fold( ||, false);
  return( res)
}
```

Fig. 2. Defining rank-invariant aggregate array operations in SAC.

Fig. 2 illustrates the principle of abstraction by rank-invariant definitions of three standard aggregate array operations. `abs` and `<=` extend the corresponding scalar functions to arrays of any rank and shape. The function `any` is a standard reduction operation, which yields `true` if any of the argument array elements is `true`, otherwise it yields `false`.

Some of the generators use the dot notation for lower or upper bounds. The dot represents the smallest or the largest legal index vector of the result array of a `genarray`-WITH-loop. The notation facilitates specification of frequent operations on all or on all inner elements of arrays.

In analogy to the examples in Fig. 2 most built-in operations known from other array languages can be implemented in SAC itself. The array module of the SAC standard library includes element-wise extensions of the usual arithmetic and relational operators, typical reduction operations like sum and product, various subarray selection facilities, as well as shift and rotate operations.

Basic array operations defined by WITH-loops lay the foundation to constructing more complex operations by means of composition, as illustrated in

```
bool cont( double[*] new, double[*] old, double eps)
{
  return( any( abs( new - old) >= eps))
}
```

Fig. 3. Defining array operations by composition.

Fig. 3. We define a generic convergence criterion for iterative algorithms of any kind purely by composition of basic array operations. Following this compositional style of programming, more and more complex operations and, eventually, entire application programs are built.

The strength of this generic rank-invariant programming style is the ability to specify array operations that are universally applicable to arrays of any shape, a property that is usually limited to built-in primitives in other languages.

4 Case study: NAS benchmark FT

In this section, we apply the generic programming techniques of SaC to a small but representative case study: 3-dimensional complex FFT. As part of the NAS benchmark suite [15] this numerical kernel has previously been used to assess the suitability of languages and compilers. Formal benchmarking rules and existing implementations in many languages ensure comparability of results. The NAS benchmark FT implements a solver for a class of partial differential equations by means of repeated 3-dimensional forward and inverse complex fast-Fourier transforms. They are implemented by consecutive collections of 1-dimensional FFTs on vectors along the three dimensions., i.e., an array of shape `[X,Y,Z]` is consecutively interpreted as a `ZY` matrix of vectors of length `X`, as a `ZX` matrix of vectors of length `Y`, and as a `XY` matrix of vectors of length `Z`.

```
complex[.,.,.] FFT( complex[.,.,.] a, complex[.] rofu)
{
  b = { [.,y,z] -> FFT( a[.,y,z], rofu) };
  c = { [x,.,z] -> FFT( b[x,.,z], rofu) };
  d = { [x,y,.] -> FFT( c[x,y,.], rofu) };

  return( d);
}
```

Fig. 4. SaC implementation of 3-dimensional FFT.

As shown in Fig. 4, the algorithm can be carried over into a SaC specification almost literally. The function `FFT` takes a 3-dimensional array of complex numbers (`complex[.,.,.]`) and consecutively applies 1-dimensional FFTs to all subvectors along the x-axis, the y-axis, and the z-axis. The SaC code takes advantage of the *axis control notation*. This notation facilitates specification of operations along one or multiple whole axes of argument arrays. Applications of this notation are transformed into WITH-loops in a pre-processing step. A

detailed introduction to both usage and compilation can be found in [16]. The additional parameter `rofu` provides a pre-computed vector of complex roots of unity, which is used for 1-dimensional FFTs.

```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even      = condense(2, v);
  odd       = condense(2, rotate( [-1], v));
  rofu_even = condense(2, rofu);

  fft_even  = FFT1d( even, rofu_even);
  fft_odd   = FFT1d( odd,  rofu_even);

  left      = fft_even + fft_odd * rofu;
  right     = fft_even - fft_odd * rofu;

  return( left ++ right);
}

complex[2] FFT(complex[2] v, complex[1] rofu)
{
  return( [v[[0]] + v[[1]] , v[[0]] - v[[1]]]);
}
```

Fig. 5. SaC implementation of 1-dimensional FFT.

The overloaded function `FFT` on vectors of complex numbers (`complex[.]`) almost literally implements the Danielson-Lanczos algorithm [17]. It is based on the recursive decomposition of the argument vector `v` into elements at even and at odd index positions. The vector `even` can be created by means of the library function `condense(n,v)`, which selects every `n`-th element of `v`. The vector `odd` is generated in the same way after first rotating `v` by one index position to the left. `FFT` is then recursively applied to even and to odd elements, and the results are combined by a sequence of element-wise arithmetic operations on vectors of

```
typedef double[2] complex

complex (*) (complex a, complex b)
{
  return( [ a[0] * b[0] - a[1] * b[1],
            a[0] * b[1] + a[1] * b[0] ]);
}

complex[+] (*) (complex[+] a, complex[+] b)
{
  res = with (. <= iv <= .) : a[iv] * b[iv]
        genarray( min( shape(a), shape(b)));

  return( res);
}
```

Fig. 6. Complex numbers in SaC.

complex numbers and a final vector concatenation (++). A direct implementation of FFT on 2-element vectors (`complex[2]`) terminates the recursion.

Note that unlike FORTRAN neither the data type `complex` nor any of the operations used to define `FFT` are built-in in SAC. Fig.6 shows an excerpt from the complex numbers module of the SAC standard library.

```
subroutine  cffts1 ( is,d,x,xout,y)

include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(1), 2)
integer i, j, k, jj

do i = 1, 3
 logd(i) = ilog2(d(i))
end do

do k = 1, d(3)
 do jj = 0, d(2)-fftblock, fftblock
  do j = 1, fftblock
   do i = 1, d(1)
    y(j,i,1) = x(i,j+jj,k)
   enddo
  enddo

  call cfftz (is, logd(1),
              d(1), y, y(1,1,2))

  do j = 1, fftblock
   do i = 1, d(1)
    xout(i,j+jj,k) = y(j,i,1)
   enddo
  enddo
 enddo
enddo

return
end
```

```
subroutine fftz2 ( is,l,m,n,ny,ny1,u,x,y)

integer is,k,l,m,n,ny,ny1,n1,li,lj
integer lk,ku,i,j,i11,i12,i21,i22
double complex u,x,y,u1,x11,x21
dimension u(n), x(ny1,n), y(ny1,n)

n1 = n / 2
lk = 2 ** (l - 1)
li = 2 ** (m - 1)
lj = 2 * lk
ku = li + 1

do i = 0, li - 1
 i11 = i * lk + 1
 i12 = i11 + n1
 i21 = i * lj + 1
 i22 = i21 + lk
 if (is .ge. 1) then
  u1 = u(ku+i)
 else
  u1 = dconjg (u(ku+i))
 endif
 do k = 0, lk - 1
  do j = 1, ny
   x11 = x(j,i11+k)
   x21 = x(j,i12+k)
   y(j,i21+k) = x11 + x21
   y(j,i22+k) = u1 * (x11 - x21)
  enddo
 enddo
enddo

return
end
```

Fig. 7. Excerpts from the FORTRAN-77 implementation of NAS-FT.

In order to help assessing the differences in programming style and abstraction, Fig. 7 shows excerpts from about 150 lines of corresponding FORTRAN-77 code. Three slightly different functions, i.e. `cffts1`, `cffts2`, and `cffts3`, intertwine the three transposition operations with a block-wise realization of a 1-dimensional FFT. The iteration is blocked along the middle dimension to improve cache performance. Extents of arrays are specified indirectly to allow reuse of the same set of buffers for all orientations of the problem. Function `fftz2` is part of the 1-dimensional FFT. It must be noted that this excerpt represents high quality code, which is well organized and well structured. It was written by expert programmers in the field and has undergone several revisions. Everyday legacy FORTRAN-77 code is likely to be less "intuitive".

5 Experimental evaluation

This section investigates the runtime performance achieved by code compiled from the SAC specification of NAS-FT, as outlined in the previous section. It is

compared with that of the serial FORTRAN-77 reference implementation coming with the NAS benchmark suite 2.3[1], with a C implementation derived from the FORTRAN-77 code and extended by OPENMP directives[2] by Real World Computing Partnership (RWCP), and last but not least with the fastest HASKELL implementation proposed in [4]. All experiments were made on a 12-processor SUN Ultra Enterprise 4000 shared memory multiprocessor using SUN Workshop compilers.

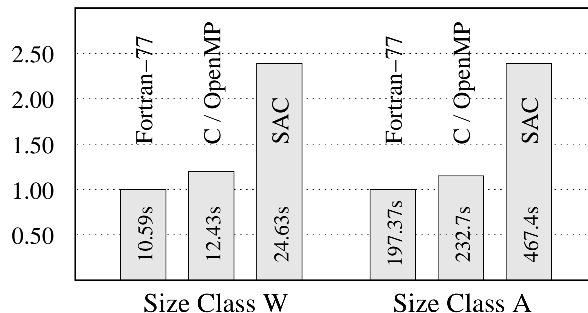

Fig. 8. Single processor performance of NAS-FT.

Fig. 8 shows sequential execution times for FORTRAN, C, and SAC. For both size classes investigated, FORTRAN-77 outperforms SAC by less than a factor of 2.4 while C outperforms SAC by less than a factor of 2.0. The performance of a few lines of highly generic SAC code is in reach of hand-optimized imperative implementations of the benchmark. The remaining performance gap must to a large extent be attributed to dynamic memory management overhead caused by the recursive decomposition of argument vectors when computing 1-dimensional FFTs. Unlike SAC, both imperative implementations use a static memory layout. HASKELL runtimes are omitted in Fig. 8 because with more than 27 minutes runtime for size class W it is more than 2 orders of magnitude slower than the other candidates. Furthermore, HASKELL fails altogether to compute size class A in a 32-bit address space. Therefore, we have excluded HASKELL from further experiments.

Fig. 9 shows the scalability achieved by the 3 candidates, i.e. parallel execution times divided by each candidate's best serial runtime. Whereas hardly any performance gain can be observed for automatic parallelization of the FORTRAN-77 code, SAC achieves speedups of up to 5.5 and up to 6.0 for size classes W and A, respectively. With these figures SAC even slightly outperforms OPENMP in terms of scalability.

Fig. 10 shows absolute runtimes using ten processors. Due to its superior sequential performance the C/OPENMP combination achieves the best abso-

[1] The source code is available at `http://www.nas.nasa.gov/Software/NPB/` .
[2] The source code is available at `http://phase.etl.go.jp/Omni/` .

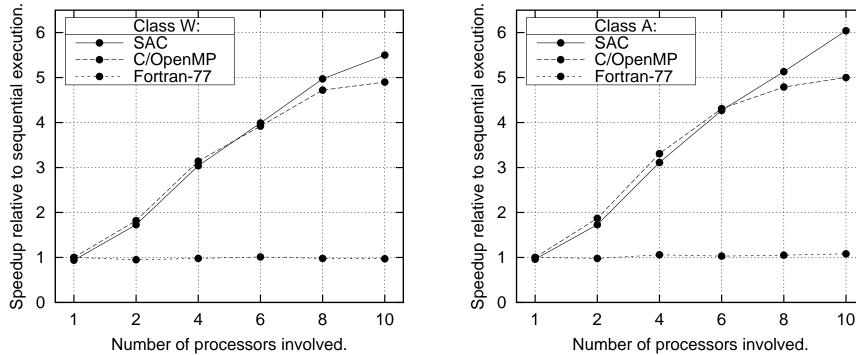

Fig. 9. Speedups achieved by multithreaded execution.

lute runtimes. However, this comes at the expense of 25 compiler directives for guiding parallelization. While parallelization of the SaC code is completely implicit like a compiler optimization, the resulting performance is still in reach of explicit approaches. It clearly outperforms automatic parallelization of the original Fortran-77 code.

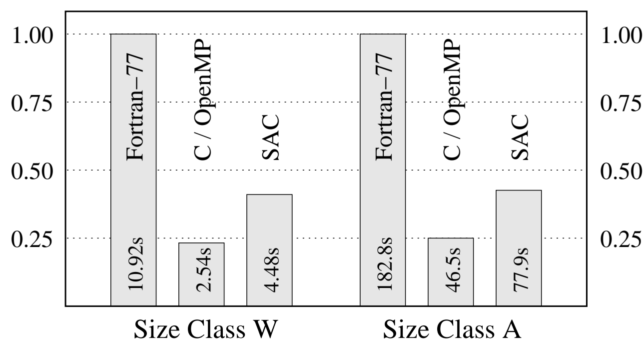

Fig. 10. 10-processor performance of NAS-FT.

6 Conclusions and future work

SaC aims at combining high-level, generic array programming with competitive runtime performance. The paper evaluates this approach based on the NAS benchmark FT. It is shown how 3-dimensional FFTs can be assembled by about 15 lines of SaC code as opposed to about 150 lines of fine-tuned Fortran-77 or C code. Due to its conciseness and high level of abstraction the SaC code clearly exhibits underlying mathematical algorithms, which are completely disguised by performance-related coding tricks in the case of Fortran-77 or C.

Development and maintenance of these codes require deep knowledge about computer architecture and corresponding optimization techniques, e.g. padding, tiling, buffering, or iteration ordering.

Nevertheless, the SaC runtime is within a factor of 2.4 of the Fortran-77 code and within a factor of 2.0 of the C code. In contrast, using the general-purpose functional language Haskell leads to a performance degradation of more than two orders of magnitude and prohibitive memory demands for non-trivial problem sizes. Furthermore, SaC by simple recompilation outperforms both low-level imperative implementations with only 4 processors of an SMP system. In contrast, only annotation with 25 OpenMP directives succeeded in exploiting multiple processors, whereas implicit parallelization of the Fortran-77 code failed to achieve any performance improvements.

Future work is basically twofold. First, various inefficiencies in the intermediate SaC code should be overcome by additional symbolic program transformations, which may allow us to further close the performance gap between SaC and low-level solutions. Second, we would like to extend the comparative study to other benchmark implementations, e.g. Mpi-based parallelization of Fortran-77 and C codes, a data parallel Hpf implementation, or a presumably faster Haskell implementation based on strict and unboxed arrays.

References

1. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, T., Simon, R., Venkatakrishnam, V., Weeratunga, S.: The NAS Parallel Benchmarks. International Journal of Supercomputer Applications **5** (1991) 63–73
2. Hartel, P., Langendoen, K.: Benchmarking Implementations of Lazy Functional Languages. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, ACM Press (1993) 341–349
3. Hartel, P., et al.: Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. Journal of Functional Programming **6** (1996)
4. Hammes, J., Sur, S., Böhm, W.: On the Effectiveness of Functional Language Features: NAS Benchmark FT. Journal of Functional Programming **7** (1997) 103–123
5. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming **13** (2003) 1005–1059
6. Grelck, C., Scholz, S.B.: Accelerating APL Programs with SAC. In Lefèvre, O., ed.: Proceedings of the International Conference on Array Processing Languages (APL'99), Scranton, Pennsylvania, USA. Volume 29 of APL Quote Quad. ACM Press (1999) 50–57
7. Grelck, C., Scholz, S.B.: HPF vs. SAC — A Case Study. In Bode, A., Ludwig, T., Karl, W., Wismüller, R., eds.: Proceedings of the 6th European Conference on Parallel Processing (Euro-Par'00), Munich, Germany. Volume 1900 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (2000) 620–624

8. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In Prasanna, V.K., Westrom, G., eds.: Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA, IEEE Computer Society Press (2002)
9. Grelck, C., Scholz, S.B.: SAC — From High-level Programming with Arrays to Efficient Parallel Execution. Parallel Processing Letters **13** (2003) 401–412
10. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In Clack, C., Davie, T., Hammond, K., eds.: Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers. Volume 1467 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (1998) 72–92
11. Grelck, C., Scholz, S.B., Trojahner, K.: With-Loop Scalarization: Merging Nested Array Operations. In Trinder, P., Michaelson, G., eds.: Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers. Volume 3145 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (2004)
12. Grelck, C.: Shared Memory Multiprocessor Support for SAC. In Hammond, K., Davie, T., Clack, C., eds.: Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, Selected Papers. Volume 1595 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (1999) 38–54
13. Grelck, C.: Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany (2001). Logos Verlag, Berlin, 2001.
14. Grelck, C.: A Multithreaded Compiler Backend for High-Level Array Programming. In Hamza, M., 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), Innsbruck, Austria, ACTA Press, Anaheim, California, USA (2003) 478–484
15. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Research Center, Moffet Field, California, USA (1995)
16. Grelck, C., Scholz, S.B.: Axis Control in SAC. In Peña, R., Arts, T., eds.: Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Revised Selected Papers. Volume 2670 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (2003) 182–198
17. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C. Cambridge University Press, Cambridge, UK (1993)