International Journal of Parallel Programming, Vol. 34, No. 4, August 2006 (© 2006)
DOI: 10.1007/s10766-006-0018-x

SAC — A Functional Array Language for Efficient Multi-threaded Execution

Clemens Grelck[1,3] and Sven-Bodo Scholz[2]

Received: September 1, 2005 / Accepted: June 1, 2006

We give an in-depth introduction to the design of our functional array programming language SaC, the main aspects of its compilation into host machine code, and its parallelisation based on multi-threading. The language design of SaC aims at combining high-level, compositional array programming with fully automatic resource management for highly productive code development and maintenance. We outline the compilation process that maps SaC programs to computing machinery. Here, our focus is on optimisation techniques that aim at restructuring entire applications from nested compositions of general fine-grained operations into specialised coarse-grained operations. We present our implicit parallelisation technology for shared memory architectures based on multi-threading and discuss further optimisation opportunities on this level of code generation. Both optimisation and parallelisation rigorously exploit the absence of side-effects and the explicit data flow characteristic of a functional setting.

KEY WORDS: Compiler optimisation; data parallel programming; multi-threading; Single Assignment C.

1. INTRODUCTION

Programming concurrent systems is known to be a challenging task. Several organisational problems need to be solved without spending too much computational overhead for organisational measures at runtime.

[1]Institute of Software Technology and Programming Languages, University of Lübeck, Ratzeburger Allee 160, 23538 Lübeck, Germany.
[2]Department of Computer Science, University of Hertfordshire, College Lane, Hatfield AL109AB, United Kingdom.
[3]To whom correspondence should be addressed. Email: grelck@isp.uni-luebeck.de

0885-7458/06/0800-0383/0 © 2006 Springer Science+Business Media, Inc.

Among these are: finding a suitable workload distribution, making the required data readily available for individual processors, and getting the required synchronisations right. As the effectiveness of possible organisational measures are highly sensitive to the problem at hand, most applications in the area of high-performance computation rely on implementations with tailor-made organisations of concurrent executions, i.e., concurrency is explicitly introduced by programmers using, for instance, message passing libraries like Mpi or Pvm.

The drawbacks of this approach are manifold. First of all, making the right design decisions for such a system requires a wealth of expertise in concurrency issues, which is not necessarily combined with the expertise in a given application area. In other words, writing a correct parallel program is substantially harder than writing a correct sequential program. Furthermore, different hardware systems may require different measures to be taken in order to achieve favourable runtimes. A hard-wired solution therefore usually limits the portability of applications to other than the initially intended hardware platform.

However, deriving and exploiting concurrency from program specifications without any explicit concurrency annotations by the programmer, in general, is considered an overly ambitious goal. There have been several attempts into that direction most of which proved less than optimal with respect to runtime behaviour. Nevertheless, for certain applications, namely those that primarily perform operations on large arrays, the situation is different. For this kind of applications, languages such as Hpf,[1] Zpl[2] or Sisal,[3] as well as the more recent developments in OpenMP[4] have demonstrated that reasonable performance gains can be achieved through smart compilation techniques or by providing compiler annotations only.

Nevertheless, these languages require programmers to organise their programs carefully in order to facilitate compilation into efficiently executable parallel code. In order to reduce the number of synchronisations the programmer usually needs to enlarge the scope of individual data parallel operations as far as possible. Furthermore, the programmer has to carefully plan the use of individual data structures as superfluous memory usage in array-intensive applications may adversely affect runtime performance to quite some extent.

All these requirements are at odds with good software engineering practice. Well designed programs build on a compositional style of programming, where a rich set of generalised operations is shared among several applications. The intended behaviour of a given application then is achieved by a combination of appropriate basic functionalities. This way,

programs can make better use of existing code, applications can be more easily maintained or adapted to changing requirements.

The downside of following established principles of software engineering like abstraction and modularisation in the field of performance-aware array programming is non-competitive runtime behaviour. There is a trade-off between the wish to efficiently develop and maintain software and the wish to run this software efficiently on parallel computers. A lot of scientific work has been devoted to overcome this dilemma by smart compilation techniques.[5–10] While they often turn out to be effective on a small scale, code-restructuring on a larger scale quickly suffers from the explicit control flow inherent to imperative programs. Any variation from the prescribed control flow requires a formal proof that the program semantics is still preserved. To do so, exact knowledge about the data flow is mandatory. However, the data flow is typically obfuscated by imperative language features. Hence, compilers must make conservative assumptions, and their effectiveness in restructuring the code is severely limited.

As an alternative programming concept, functional languages are characterised by the absence of any control flow. Program execution is solely driven by data flow, which is easily identifiable in a program's source code. Functional programs specify the extensionally observable behaviour only. The organisation of the individual computations is fully under the regime of the compiler. As a consequence, assignments to variables do not necessarily require resource allocations at runtime; computations can be shifted around in the program without any restrictions other than data dependencies. These liberties provide the grounds for radical code transformations that transform a specified nesting of general purpose operations into tailor-made operations with better runtime characteristics.

Unfortunately, almost all existing functional programming languages focus on algebraic data types like lists and trees rather than on arrays as primary data structures. Languages like HASKELL[11] or CLEAN[12] do support arrays,[13–16] but despite some effort to improve efficiency they are far from meeting the performance expectations of classical application domains of parallel processing.[17–19]

Reasons for the unsatisfactory runtime performance of general-purpose functional languages in processing multi-dimensional arrays are manifold. Certainly, all high-level language features like polymorphism, higher-order functions, partial applications, or lazy evaluation contribute some share of the overhead, but array-specific pitfalls exist as well. Conceptually, functions consume argument values and create result values from scratch. For small data items, such as list cells, this can be implemented fairly efficiently. However, operations which "change" just a few elements of a large monolithic array run into the *aggregate update*

problem.[20] They need linear time to create a copy of the whole array, whereas imperative languages accomplish the same task in constant time by destructively writing into the argument array.

Investigations show that the only way to achieve reasonably competitive performance characteristics with array processing in functional languages is to artificially re-introduce a partial control flow, e.g. by the use of uniqueness types[21] or state monads.[22] However, this approach incidentally sacrifices both the benefits we are seeking on the programming level and the ease of program restructuring. Likewise, functional languages that introduce arrays as impure features, e.g. ML,[23] suffer from the same deficiencies.

A notable exception in the development of functional languages is SISAL,[3] which first focussed on purely functional arrays. While SISAL has a reputation for highly efficient code, the level of abstraction is rather low. All array operations need to be defined for statically fixed levels of vector nestings. This restriction renders definitions of generic array operations similar to those in array languages such as APL[24] impossible. As a consequence, a compositional programming style based on generic array operations cannot be achieved.

With SAC (Single Assingment C)[25] we aim at supporting and encouraging such a high-level compositional programming style on arrays without sacrificing the potential for efficient execution. The language design of SAC aims at combining three essential objectives:

- support for a high-level, compositional programming style with arrays,
- potential for optimisations that transform compositions of fine-grained data parallel operations into coarse-grained ones,
- feasibility of compiler-directed generation of parallel host machine code.

By already taking optimisation and parallelisation opportunities into account in the language design we aim at achieving a runtime performance which is competitive with programs written in machine-oriented style.

The syntax of SAC is adopted as far as possible from that of C, hence the name. This measure is meant to facilitate adaptation of SAC for programmers with a background in imperative programming languages. Via its module system and the foreign language interface almost all familiar functions from the standard C library can be used in SAC programs without any difference. This includes functions that interact with the execution environment, e.g. the file system or a terminal. Despite the imperative-looking syntax of SAC, the underlying semantics is purely functional, i.e., program execution is based on the principle of context-free substitution of expressions

rather than the step-wise modification of a global state. One may think that adopting a well-known syntax, but giving it a different semantics leads to confusion, but the opposite is true. In practice, the functional semantics of SAC code and the imperative semantics of literally identical C code coincide. This property allows programmers to stick to their preferred model of reasoning, either functional or imperative. At the same time, the compiler may take full advantage of the functional, side-effect-free semantics for advanced optimisations that are not feasible in an imperative context.

On top of the C-like language kernel of SAC, an array model similar to that of array languages such as APL,(24) NIAL,(26) or J(27) is adopted in SAC. Arrays are first class citizens of the language, i.e., they are considered state-less entities that are passed to and returned from functions by value rather then by reference. This liberates the programmer from the burden of explicit memory mangement for arrays and allows them to treat arrays in the same way as scalars in conventional languages.

In contrast to the classical array languages as well as in contrast to the high-level extensions of imperative languages such as the more recent FORTRAN dialects, in SAC, the generic array operations are not built-in operators. Instead, they are defined in the language itself. This unique feature allows complex generic array operations to be successively defined from more basic ones.

Whereas such a compositional style of array programming is attractive in terms of programming efficiency, code reuse, and maintenance costs to mention just a few, straightforward compilation into host machine code is unlikely to yield acceptable performance levels. Separation of concerns on the specification level prevents efficient execution on the machine level. In practice, the compositional style of programming leads to the creation of numerous temporary arrays and to the repeated traversal of existing arrays at runtime. Both is prohibitively expensive on modern hardware designs where cpu speed by far exceeds memory speed and the effective utilisation of memory hierarchies is crucial for overall performance. Likewise, parallel performance suffers compositional programming as well. While we can typically parallelise individual array operations straightforwardly following a data parallel approach, the usually extremely low computational complexity per argument array element incurs a poor ratio between productive computation and coordination overhead.

In order to bridge the gap between high-level programming and attractive runtime behaviour our compiler systematically restructures code from a representation that is amenable to humans into a representation that is amenable to efficient execution on (parallel) machines. Step-by-step deeply nested compositions of simple, fine-grained, and general-purpose array operations are transformed into complex, coarse-grained, and

application-specific operations. The key to this compilation process is a versatile SaC-specific array comprehension and reduction construct, named WITH-loop. It is versatile enough to define primitive, highly generic array operations similar to the built-in array operations in languages such as Fortran90 on the one hand side, and to specify rather complex, problem-specific array manipulation on the other side. In fact, With-loops form the major intermediate layer of representation during the compilation process and constitute the foundation of our code-restructuring optimisations. As such, their design aims at striking the balance between the expressive power to represent complex operations on the one hand side and the ability to be compiled into efficient machine code on the other hand side. During the code-restructuring process large numbers of originally simple and computationally light-weight WITH-loops are systematically aggregated into few complex and computationally heavy-weight ones. Successful aggregation of WITH-loops essentially relies on the functional, semantics of SaC, which guarantees the absence of side-effects and exact compile time knowledge about data flow.

With-loops are inherently data parallel. Consequently, they also form the natural basis for multi-threading. Preceding optimisations typically result in WITH-loops whose computational complexity per element is much more attractive for parallelisation than source-level array operations. Furthermore, aggregation of WITH-loops incidentally eliminates many of the synchronisation and communication requirements typical for naive parallelisation on the basis of source-level array operations. However, aggregation of WITH-loops is always constrained by structural properties of the operations involved, not only by data dependence considerations. Separation of concerns between the computational task associated with a WITH-loop and the corresponding coordination behaviour opens up a range of additional optimisation opportunities that further aim at reducing the need for costly synchronisation and communication events at runtime. Once again the functional semantics pays off. In the absence of a control flow the sequence in which operations are executed at runtime is solely limited by data flow constraints. This allows us to systematically rearrange the code and derive a particularly well-suited sequence of operations from the data flow graph, i.e. a sequence that reduces synchronisation requirements.

So far, we have published on the language design and programming methodology of SaC[25] as well as on various individual measures taken to compile highly generic SaC programs into efficient code both for sequential[28–30] and for parallel[31–33] execution. The main contribution of this paper is to give a comprehensive account of the interplay between language design, programming methodology, compiler optimisation and

generation of multi-threaded code. It is this careful interplay that allows us to combine highly generic, declarative programming with competitive runtime behaviour, as demonstrated in a number of case studies.[34–36]

The remainder of this paper is organised into three larger sections. We present the language design of SaC and the associated programming methodology in more detail in Section 2. Section 3 outlines the compilation process with particular emphasis on the code restructuring array optimisations. Parallelisation into multi-threaded code as well as optimisations on this level are the subject of Section 4. Section 5 draws some conclusions.

2. SAC—SINGLE ASSIGNMENT C

2.1. Core Language Design

Single Assignment C (SaC) is a functional language whose design targets array-intensive applications as they typically appear in areas like computational sciences or image processing. The fundamental idea in the design of the language is to keep the language as close as possible to C, but to nevertheless base the language on the principle of context-free substitutions. While the former is intended to attract application programmers with an imperative background, the latter ensures the Church-Rosser property, which is crucial for extensive compile-time optimisations as well as for non-sequential execution. The second key objective in the design of SaC is to provide support for high-level declarative array programming in a way similar to interpreted array languages like Apl or J.

Figure 1 illustrates the overall design of SaC. As can be seen in the middle of the figure, a large part of standard C, e.g. basic types and operators as well as the way of defining and applying functions, is adopted by SaC without alteration. Only a few language constructs of C such as pointers and global variables need to be excluded in order to be able to guarantee a side-effect-free setting. Instead, some new language constructs are added pertaining to array programming.

The most fundamental addition to the language kernel in SaC is genuine support for n-dimensional arrays. As scalar values are considered 0-dimensional arrays, SaC does not support any data structures apart from arrays.

Besides the integration of multi-dimensional arrays the most important addition to the language kernel are with-*loops*. They are data parallel skeleton operations suitable for defining various generic array operations including element-wise generalisations of the standard scalar operations as well as more complex operations similar to those available in languages such as Apl. In principle, with-loops bear some similarity with

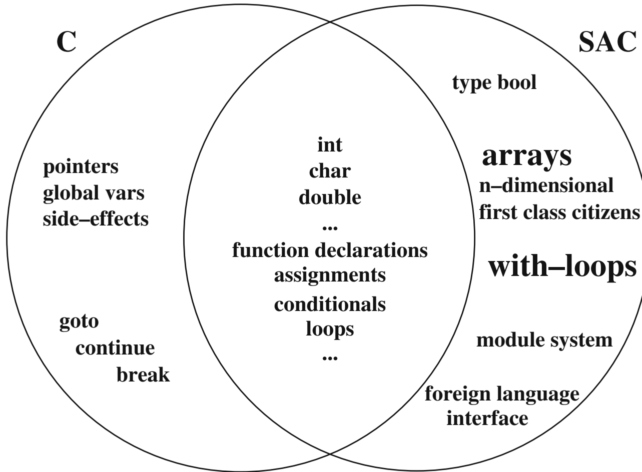


Fig. 1. The overall design of SAC.

array comprehensions as found in functional languages such as HASKELL or CLEAN, and they can also be considered a variant of FORALL-loops in FORTRAN. However, the distinguishing feature of WITH-loops is their capability to be specified shape-invariantly. A shape-invariant operation can be applied to argument arrays of statically unknown extents as well as statically unknown dimensionality (rank). As far as we are aware, this is a unique feature of SAC. In languages such as modern FORTRAN-dialects or dedicated array languages such as APL shape-invariance is restricted to the fixed set of built-in operators.

2.1.1. To be and not to be Functional

The incorporation of most of the fundamental language constructs of C such as loops, conditionals, and assignments into the functional setting of SAC allows the programmer to stick to his preferred model of computation. To illustrate this effect, let us consider the following SAC function `foo`

```
int foo(int v, int w)
{
  r = v + w;
  r = r + 1;
  return(r);
}
```

which takes two arguments `v` and `w` and computes the sum, which is stored in the local variable `r`. Then, `r` is incremented by `1` and returned as result.

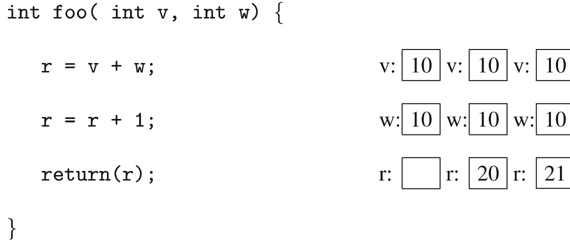


Fig. 2. An imperative look on `foo`.

An imperative interpretation of `foo` is shown in Fig. 2. In the imperative world, `v`, `w`, and `r` constitute names for box variables. During the execution of the body of `foo` the content of these box variables is successively changed. Assuming an application to arguments `10` and `10` these variable manipulations are indicated on the right hand side of Fig. 2. Eventually, the final value of variable `r`, i.e. `21`, is returned as the overall result of the function call to `foo`.

However, the definition of the function `foo` equally well can be interpreted as syntactic sugar for a `let`-based function definition, as shown on the left hand side of Fig. 3. With this interpretation, `v`, `w`, and `r` become variables in a λ-calculus[37] sense. As we can see, the two successive assignments to `r` have turned into two nested `let`-expressions each binding `r` to a different value. In fact, these `let`-expressions refer to two distinct variables `r` which accidentally have the same name. The scoping rules ensure that the variable `r` in the defining expression of the second `let`-expression refers to the variable `r` bound to the value of `v+w` in the first `let`-expression. In contrast, the variable `r` in the goal expression refers to binding of the second `let`-expression, which effectively shadows the first binding of `r`.

A further transformation into an applied λ-calculus, as shown on the right-hand-side of Fig. 3, identifies the potential for independent

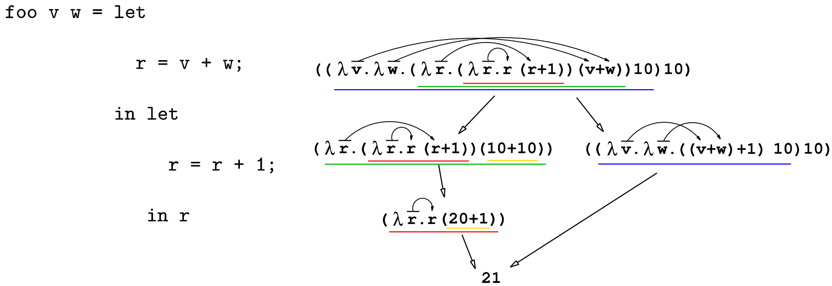


Fig. 3. A functional look on `foo`.

evaluation of subexpressions. The arrows on top of the λ-expressions indicate the static scoping of the individual variables. The lines under the expressions indicate the β-redices that are present. As indicated by the different reduction sequences, the λ-calculus representation thus eases the identification of legal program transformations, part of which may be performed at compile-time. The static availability of this information together with the formal guarantee that the result of the computation remains invariant for any chosen (terminating) reduction order forms the basis for many program optimisations in SAC. These are essential when it comes to compiling programs for highly efficient concurrent program execution.

We achieve this duality in program interpretation by the choice of a subset of C that we can easily map to an applied λ-calculus whose semantics reflects that of the corresponding C program. A formal definition of the semantics of SAC is beyond the scope of this paper; the interested reader is referred to Ref. 25. In the sequel, it suffices to expect all language constructs adopted from C to adhere to their operational behaviour in C.

2.1.2. Stateless Arrays

In a purely functional world all data are state-less values. This very property is essential for achieving the freedom in choosing any reduction order, as explained in the previous section. If any data would have a state then all state manipulating operations would impose a fixed order within the evaluation of these.

While almost all programming languages have a notion of state-less scalars, for arrays this is usually not the case. The reason for that design choice is rather simple: as soon as an array is considered state-less it cannot be modified after it has been defined. As a consequence a modification of a single element of an array, at least in principle, does require the entire array to be copied. A naive implementation of this property is prohibitive in terms of both, runtime and space efficiency. However, these inefficiencies can be overcome by using reference counting techniques and by applying several optimisation techniques as developed in the context of SISAL[38] and later refined in the context of SAC.[39]

Having state-less arrays does not only facilitate radical program optimisations as explained before, but it also benefits the programmer. It liberates the programmer from the burden to think about memory issues entirely. No memory needs to be allocated, arrays passed as arguments to functions can never be affected by such calls, and the programmer does not need to copy arrays in order to preserve a certain state. Instead, the programmer defines various arrays and relies on the compiler and the runtime system to ensure that memory is being allocated and reused as often as possible.

2.1.3. Shape-Invariant Programming

Another distinguishing feature in the design of SAC is its support for shape-invariant programming. The ability to define operations that can be applied to arrays of arbitrary rank may seem overkill at first glance. Most real world examples require fixed rank arrays only and the rank usually does not exceed 4.

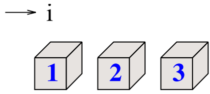
Looking at array programming languages such as APL one can observe that the productivity of programmers derives from the availability of a rich set of shape-invariant built-in operators. Their generic applicability allows programs to be specified rather concisely. Complex loop nestings for mapping operations over certain dimensions are usually not required. The only weakness of such a built-in approach is its inflexibility. If the set of operations provided by a chosen language does not match well a given application task, program specifications often become a bit cumbersome. As a consequence of this, many APL dialects have evolved all of which provide slightly different sets of built-in operations.

Providing shape-invariant programming on a meta level such as the WITH-loops in SAC enables the programmer to design a set of shape-invariant operators to the particular requirements of a given set of applications. These operations can be stored within a library, which can be shared for entire sets of applications.

Another advantage of shape-invariant programming stems from the fact that $(n + m)$-dimensional arrays can be considered a homogeneous n-dimensional nesting of m-dimensional arrays. Provided that all operations are consistently defined, programs written for n-dimensional arrays can be applied without modification to $(n + m)$-dimensional arrays. All "inner arrays" are simply applied element-wise. The consistency of such a set of operations manifests itself in several fundamental universal equations which, once, they are chosen, dictate the behaviour of most operations in the non-scalar case. A discussion of the design choices and their consequences are beyond the scope of this paper and can be found elsewhere (e.g. in Ref. 40). In order to give the reader a flavour of such a set of universal equations, in the next section, we provide the most important equations that underly the standard library of SAC.

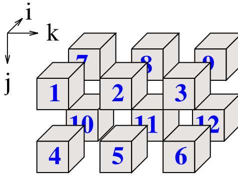
2.2. Arrays at a Glance

All arrays in SAC are represented by two vectors: the *data vector* contains all elements in linear order while the *shape vector* defines the structural properties of the array. Figure 4 shows a few example arrays. As can be seen from the examples, the length of the shape vector corresponds to



shape vector: [3]

data vector: [1, 2, 3]



shape vector: [2, 2, 3]

data vector: [1, 2, 3, ..., 11, 12]

42

shape vector: [ ]

data vector: [42]

Fig. 4. Array representations.

the rank (number of axes) of the array while the individual elements of the shape vector define the array's extent along each axis. The data vector enumerates all elements with lexicographically increasing indices. From this relation between data and shape vector we obtain:

Lemma 1. *Let* $[d_0, \ldots, d_{q-1}]$ *denote the data vector of an array and let* $[s_0, \ldots, s_{n-1}]$ *denote its shape vector. Then we have* $q = \prod_{i=0}^{n-1} s_i$.

The bottom of Fig. 4 shows that scalar values can be considered 0-dimensional arrays with empty shape vector. Note here, that Lemma 1 still holds for scalars.

2.2.1. Specifying Arrays

We specify vectors using the common square bracket notation introduced in Fig. 4. Multi-dimensional arrays can be defined either by nesting vectors, or we use the following notation that explicates the representation of arrays by shape and data vector:

$$\texttt{reshape}\ ([s_0, ..., s_{n-1}], [d_0, ..., d_{q-1}])$$

where $q = \prod_{i=0}^{n-1} s_i$. In fact, we can consider `reshape` the only array constructing function apart from WITH-loops in SAC. This observation leads to the following generalisations:

$$s \equiv \texttt{reshape([], [}s\texttt{])}, \text{and}$$
$$\texttt{[}v_0, ..., v_{n-1}\texttt{]} \equiv \texttt{reshape([}n\texttt{], [}v_0, ..., v_{n-1}\texttt{])}.$$

Provided that Lemma 1 holds for the resulting array, the following property holds for `reshape`:

$$\begin{aligned}&\texttt{reshape(shp_vec, reshape(shp_vec_2, data_vec))}\\ &\quad == \texttt{reshape(shp_vec, data_vec)}\end{aligned} \tag{1}$$

2.2.2. Inspecting Arrays

Alongside the array constructing operator `reshape`, functions for extracting shape and data information are required. We introduce two operations for retrieving shape information:

`shape` returns an array's shape vector, and
`dim` returns an array's rank (dimensionality)

For example, we have:

```
shape( 42) == []
dim( 42) == 0
shape( [1, 2, 3]) == [3]
dim( [1, 2, 3]) == 1
shape( reshape( [2, 3], [1, 2, 3, 4, 5, 6])) == [2, 3]
dim( reshape( [2, 3], [1, 2, 3, 4, 5, 6])) == 2
```

Formally, `shape` is defined by

$$\texttt{shape(reshape(shp_vec, data_vec))} == \texttt{shp_vec} \tag{2}$$

and `dim` is defined by

$$\texttt{dim(reshape(shp_vec, data_vec))} == \texttt{shape(shp_vec)[0]} \tag{3}$$

where the square brackets denote element selection. From these definitions, we can derive

$$\forall \texttt{a} : \texttt{dim(a)} == \texttt{shape(shape(a))[0]} \tag{4}$$

as

$$\begin{aligned}&\texttt{dim(reshape(shp_vec, data_vec))}\\ &\stackrel{(3)}{==}\ \texttt{shape(shp_vec)[0]}\\ &\stackrel{(2)}{==}\ \texttt{shape(shape(reshape(shp_vec, data_vec)))[0]}\end{aligned} \qquad \square$$

So far, we have used square brackets to denote selection within vectors. However, we want to introduce a more versatile definition for array selections. It is supposed to work for n-dimensional arrays in general. As one index per axis is required, such a definition requires an n-element vector as index argument rather than n separate scalar index arguments. Hence, we define an operation

$$\texttt{sel(idx_vect, array)}$$

which selects that element of `array` that is located at the index position `index_vect`. For example:

```
sel( [1], [1, 2, 3]) == 2
sel( [1, 0], reshape( [2, 3], [1, 2, 3, 4, 5, 6])) == 4
```

As we can see from the examples, we always have `shape( idx_vect)[0] == dim(array)`. This leads to the formal definition

$$\begin{array}{c}\texttt{shape(sel(idx_vec, reshape(shp_vec, data_vec))) == 0}\\ \text{provided that}\quad \texttt{shape(idx_vec) == shape(shp_vec)}\end{array} \tag{5}$$

From it, we obtain for scalars `s`:

$$\texttt{sel([], s) == s}$$

In order to extend this property to non-scalar arrays (5) is generalised into

$$\begin{array}{c}\texttt{shape(sel(idx_vec, reshape(shp_vec, data_vec)))}\\ \texttt{== shape(shp_vec) - shape(idx_vec)}\\ \text{provided that}\quad \texttt{shape(idx_vec) <= shape(shp_vec)}\end{array} \tag{6}$$

This extension enables the selection of entire subarrays whenever the index vector is shorter than the rank of the array to be selected from. For example, we have

```
sel( [1, 0], reshape( [2, 3], [1, 2, 3, 4, 5, 6]))
  == 4
sel( [1], reshape( [2, 3], [1, 2, 3, 4, 5, 6]))
  == [4, 5, 6]
sel( [], reshape( [2, 3], [1, 2, 3, 4, 5, 6]))
  == reshape( [2, 3], [1, 2, 3, 4, 5, 6]).
```

2.3. WITH-Loops

Besides basic support for *n*-dimensional arrays, as described in the previous section, all aggregate operations on arrays in SAC are defined in terms of a language construct called WITH-loop. Despite the term *loop* in the name, WITH-loops are more similar to array comprehensions in other functional languages than to control structures in imperative language. In particular, WITH-loops always appear in expression position and yield a value.

The most simple form of WITH-loops constitutes a mapping of an expression for computing an individual array element to all element positions of an array; it takes the general form

with
  generator : *expr*
genarray(*shape*)

Such a WITH-loop defines an array (key word `genarray`), whose shape is determined by the expression *shape*, which must evaluate to an integer vector. The elements of the resulting array are defined by a colon-separated pair of a *generator* and an associated expression *expr*. The generator defines a set of indices; for each element of this set the value of the associated expression defines the corresponding element value of the result array. As an example, consider the following WITH-loop

```
with
  default : 42
genarray( [3,5])
```

which computes the matrix $\begin{pmatrix} 42 & 42 & 42 & 42 & 42 \\ 42 & 42 & 42 & 42 & 42 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$. As specified by the *shape* expression, the WITH-loop defines a matrix with 3 rows and 5 columns. The generator `default` defines the set of all legal indices with respect to the shape of the result array. Hence, each element of the result array has the value `42`.

For all but the most trivial aggregate array operations we need access to the current index position within generator-associated expressions. This can be accomplished as in the following example:

```
with
  (iv) : iv[0]
genarray( [5])
```

Here, the generator consists of an identifier enclosed in parentheses. Like in the case of the generator `default`, this notation refers to the set of all legal indices. However, it allows the associated expression to be specified in terms of the position where the element is located. Hence, the above example defines the vector `[0,1,2,3,4]`. Note here, that we still need to select the first component from `iv` as the *generator variable* always refers to an index vector of the same length as that of the shape vector.

In many cases not only the concrete values of array elements depend on the index position, but even the expressions that define them are different in different parts of the array. Whereas this could in principle be accomplished by conditional expressions that evaluate to different subexpressions depending on some predicate on the index position, the resulting code is clumsy and the performance of compiled code typically poor. To avoid these drawbacks, we introduce the concept of *multi-generator* WITH-loops. The basic idea is to replace the pair of a generator and an expression by several pairs of these and to associate each of the pairs with a range of indices in a way that guarantees all ranges to constitute a partition of the entire index range. Syntactically, the range specification is annotated at the index variable by replacing `(` *idx_vec* `)` with `(` *idx_vec* `<` *ub* `)` or with `(` *lb* `<=` *idx_vec* `<` *ub* `)`. Here, *lb* and *ub* denote expressions which must evaluate to integer vectors of the same length as the *shape* expression. They define the element-wise minimum and maximum of the index range covered, respectively. If the explicit specification of a lower bound is omitted, the lower bound defaults to a vector of zeros of appropriate length. This choice is motivated by the fact that the origin of index spaces in SaC always starts at 0 and, hence, this lower bound is found frequently in generators.

With these extensions at hand, we can for example write the concatenation of two vectors `a` and `b` as follows:

```
with
  ([0]      <= iv < shape(a)) : a[iv]
  (shape(a) <= iv < shape(b)) : b[iv-shape(a)]
genarray( shape(a) + shape(b))
```

Often, it is convenient to combine range generators with default generators, e.g. the WITH-loop

```
with
  ([2,1] <= iv < [8,11]): a[iv]
  default: 0
genarray([10,13)
```

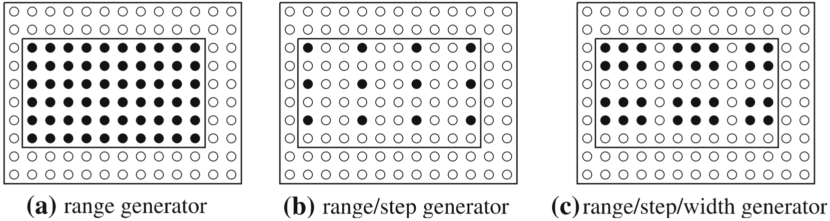


Fig. 5. Illustration of generator-defined index sets.

defines a 10×13-element matrix whose inner elements are taken from an existing array `a` while its boundary elements are uniformly set to zero, as illustrated in Fig. 5a. The `default` generator now specifies the set of indices not contained in any of the other generator-defined index sets.

The concept of multi-generator WITH-loops introduces generators that define dense rectangular ranges of indices. We further refine this concept to regular grids by adding a *step vector* to generators.

```
with
  ([2,1] <= iv < [8,11] step [2,3]): a[iv]
  default: 0
genarray([10,13])
```

The additional step specification, an expression that must evaluate to an integer vector of the same length as the lower and the upper bound vector, results in a periodic, grid-like index set, as illustrated in Fig. 5b.

Our last generator extension adds an additional *width vector* to a grid generator:

```
with
  ([2,1] <= iv < [8,11] step [3,4] width [2,3]): a[iv]
  default: 0
genarray( [10,13])
```

As illustrated in Fig. 5c, the width specification allows us to specify index sets that repeat rectangular blocks of indices in a periodic manner. While the width vector determines the shape of these blocks, the step vector defines the periodicity.

The `genarray`-WITH-loops as outlined so far are accompanied by two additional variants of WITH-loops: `modarray`-WITH-loops and `fold`-WITH-loops. The `modarray`-WITH-loop addresses the frequent case in which a new array is derived from an existing array such that the new array has the same shape as the existing one and the values of some of its elements are directly taken from the corresponding elements of the existing array. For example, the WITH-loop

```
with
  ([2,1] <= iv < [8,11]): 0
modarray( a)
```

is in a sense inverse to the one illustrated in Fig. 5a. It sets the inner elements of the resulting array to zero while it implicitly copies the values of the boundary elements from the existing array `a`, which is referenced following the key word `modarray`. The `modarray`-WITH-loop combines two aspects: Both the shape and the default generator are explicitly derived from an existing array. The syntactic position in which this array is referenced is that of an arbitrary expression.

In contrast, to the rather small extension of `modarray`-WITH-loops, `fold`-WITH-loops are characterised by a quite different operational semantics. Rather than array comprehensions, `fold`-WITH-loops are abstract representations of reduction-like computations. Following the key word `fold` we specify the name of a binary folding operation and its neutral element, which may be defined by an arbitrary expression. For example, the WITH-loop

```
with
  ([0] <= iv < shape(vec)): vec[iv]
fold( +, 0)
```

defines the sum of all elements of a vector `vec`. More precisely, it computes the following sum:

$$0 + \texttt{vec}[0] + \texttt{vec}[1] + \texttt{vec}[2] + \cdots + \texttt{vec}[\texttt{shape}(\texttt{vec}) - 1]$$

Since the generator defines a *set* of indices rather than a sequence of indices and we intend to exploit this property for code transformation, legal fold operations must be associative and commutative to guarantee deterministic results. Whereas `fold`-WITH-loops like the other WITH-loop variants may have multiple generator/expression pairs, only range generators are allowed. This is due to the principle lack of a suitable reference shape that would define the bounds of the overall index space.

2.4. The Type System of SAC

As mentioned in Section 2.1, the elementary types of C are also available in SaC. However, they constitute only a small subset of the types of SaC. For each elementary type in C there exists an entire hierarchy of array types in SaC. As an example, Fig 6 shows the hierarchy for integer arrays. It consists of three layers of array types which differ wrt. the level of shape restrictions that is imposed on their constituents. On the top

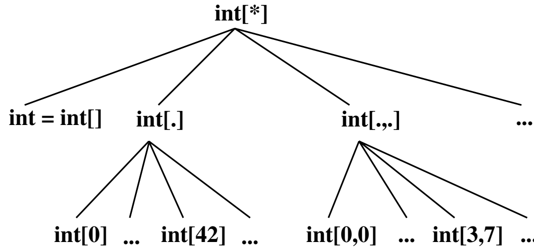


Fig. 6. A hierarchy of shapely types.

layer, we find `int[*]` which comprises all possible integer arrays. The second layer of types distinguishes between arrays of different rank (number of axes). This layer comprises the standard type `int`, which still denotes scalar integers only. All other types on this level are denoted by the elementary type followed by a vector of *dot*-symbols. The number of *dot*-symbols determines the rank of the arrays contained. On the bottom layer, the types are shape-specific. They are denoted by the elementary type followed by the shape. For example, `int[3,2]` denotes all integer matrices with three rows and two columns.

Although a generic array programming style suggests a predominant use of the top layer types, the availability of the type hierarchy provides the programmer with additional expressiveness. Domain restrictions wrt. rank or shape of the arguments can be made explicit, and support for function overloading eases rank/shape-specific implementations. Figure 7 shows an example for such an overloading.

Let us consider an implementation of a general solver for a set of linear equations `Ax = b`, as indicated in the top of Fig. 7. For arrays of a certain shape it may be desirable to apply a different algorithm which

```
double[.] solve( double[.,.] A, double[.] b) {

  /* general solver */   ...

}



double[3] solve( double[3,3] A, double[3] b) {

  /* direct solver */   ...

}
```

Fig. 7. Overloading and function dispatch.

has different runtime properties. The bottom of Fig. 7 shows how this functionality can be added in SaC by specifying a further instance of the function solve which is defined on a more restricted domain.

Besides the hierarchy of array types and function overloading it should be mentioned here that SaC does not require the programmer to declare array types for variables. Type specifications are only mandatory for argument and return types of functions.

2.5. Programming Methodology

SaC propagates a programming methodology based on the principles of abstraction and composition. Rather than building entire application programs by means of WITH-loops, we use WITH-loops merely to realise small functions, abstractions with a well defined and easily comprehendable meaning. They represent the basic building blocks for the composition of full application programs.

Figure 8 illustrates the principle of abstraction by rank-invariant definitions of three standard aggregate array operations. The overloaded definitions of the function abs and the infix operator >= extend the

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

  return( res)
}

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

  return( res)
}

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

  return( res)
}
```

Fig. 8. Defining rank-invariant aggregate array operations in SaC.

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`.

In analogy to the examples in Fig. 8 we have implemented most aggregate array operations known from other languages 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 for constructing more complex operations by means of composition. As illustrated in Fig. 9, we may 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 real power of shape-invariant programming can be observed at this very simple example. Although one may read the definition of the function `continue` as if it was applied to scalar arguments, and in fact it can be applied to scalars, the shape-invariance of the individual operations makes the whole function applicable to arrays of arbitrary rank. This does not only liberate the programmer from nestings of loops that obfuscate the core functionality of the function, but it also makes the function more generally applicable and more easily maintainable.

Another advantage to be observed materialises when it comes to debugging and testing programs. Real world applications tend to manipulate large multi-dimensional arrays. In a non-shape-invariant setting, an inspection of partial results for testing or debugging is rather difficult. It usually requires some extra code to be inserted that extracts a small enough subset of data suitable for human inspection. Even if some subfunctionality operates on outermost dimensions only, it usually cannot be applied to arrays of a smaller rank as the loop nesting fixes the expected rank of arguments.

In a shape-invariant setting as in SAC, most of these problems vanish. Individual subfunctionalities can be developed, debugged, and tested on arrays of smaller rank. After debugging these functions can simply be applied to arrays of higher rank as required by potential applications.

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

Fig. 9. Defining array operations by composition.

Considering our example again, it suffices to get the scalar case right. Subsequently, the function `continue` can be applied to arrays of arbitrary rank as indicated by the argument types specified.

As case studies such as Ref. 36 show, this particular feature proves very useful when it comes to specifying real world applications in SAC.

3. COMPILATION

3.1. Overview

Compiling high-level SAC programs following the design principles of abstraction and composition into efficiently executable machine code is a challenging undertaking. Figure 10 shows the major phases of the compilation process. After scanning and parsing the SAC-program to be compiled, its internal representation is simplified by a transformation called *functionalisation*. In addition to a general desugaring into a core language and the systematic flattening of complex expressions, we bring our internal representation of the code in line with its functional semantics. For example, we explicitly replace loops by tail-end recursive functions and eliminate multiple assignments to variables by systematic renaming of identifiers. As a result of this phase, we achieve a variant of static single assignment form Ref. 41, which is much more amenable to further code transformations than the initial language-level representation.

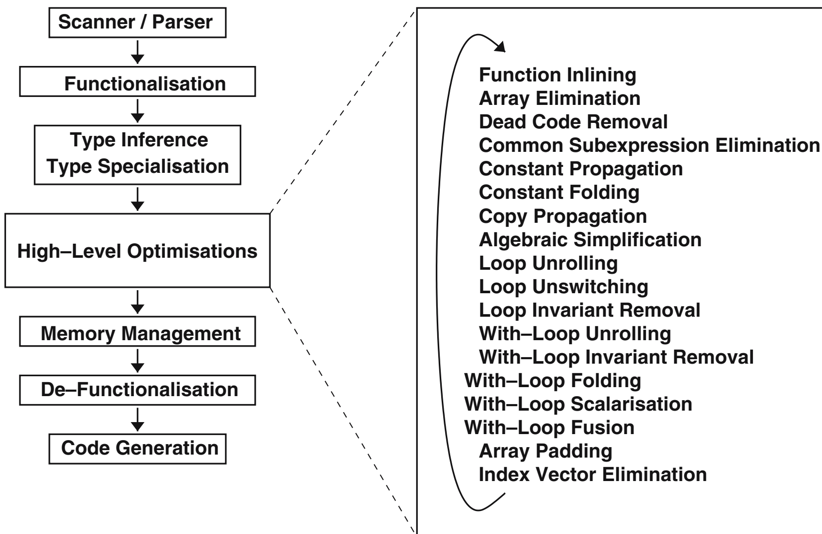


Fig. 10. Compilation of SAC programs.

The next compilation phase implements a type inference algorithm based on the hierarchy of array types described in Section 2. For each expression in the code we aim at inferring as concrete as possible information concerning its array shape. If feasible, functions with more general argument types than the concrete values they are applied to are specialised.

The central and by far most complex part of the compiler is the high-level code optimiser. It consists of an extensive range of machine-independent optimisations both of general nature and SAC-specific. General optimisations such as constant folding, common subexpression elimination, dead code removal, or variable and copy propagation are well known in literature. However, their applicability in practice substantially benefits from the absence of side-effects in our setting. In fact, they can be applied much more rigorously than in state-of-the-art compilers for imperative languages with an often opaque data flow. Since successful application of one optimisation often triggers several others, all optimisations are arranged in a cycle. Optimisation terminates as soon as a either a fixed point or a pre-specified number of cycles is reached.

At the heart of the optimiser are the SAC-specific array optimisations, namely WITH-LOOP-FOLDING, WITH-LOOP-FUSION, and WITH-LOOP-SCALARISATION. The compositional style of high-level array programming typically leads to a large number of WITH-loops in intermediate SAC code. Each individual WITH-loop represents a fairly simple and — per element of arrays involved — computationally light-weight operation. Without additional optimisation typical intermediate SAC code would inflict the creation of numerous temporary arrays and the repeated traversal of existing arrays at runtime. Furthermore, taking individual WITH-loops as basis for parallelisation would lead to frequent synchronisations and a generally poor ratio between coordination overhead and productive computation. As a consequence, neither sequential nor parallel performance would be satisfying. WITH-LOOP-FOLDING, WITH-LOOP-FUSION, and WITH-LOOP-SCALARISATION aim at systematically condensing compositions of simple and computationally light-weight WITH-loops into more complex but also more heavy-weight WITH-loops. Step by step, they transform intermediate SAC code from a representation that is amenable to programmers into a representation that is amenable to efficient execution on computing machinery; The following sections introduce these transformations in more detail.

With the *memory management* compiler phase we leave the state-free, functional world of SAC. Here, we explicitly introduce the notion of memory and augment the code with primitives to allocate and de-allocate memory resources to store arrays at runtime. De-allocation of memory is based on systematic counting of active references to arrays at runtime.

The corresponding instructions are also inserted into the intermediate code during this phase.

Eventually, the code is *de-functionalised*, i.e., recursive functions are again replaced by more efficient loops and multiple assignments are re-introduced to reduce the variable pressure. In the very last stage, we generate C code. Compilation to C rather than to machine code liberates us from hardware-specific, low-level optimisations such as delay-slot utilisation or register allocation. It also allows us to support a wider range of target architectures and operating systems with limited manpower.

3.2. WITH-Loop Folding

WITH-LOOP-FOLDING addresses vertical compositions of WITH-loops, i.e., the result of one WITH-loop is referenced as an argument in another WITH-loop. Consider, for example, a definition

$$\texttt{res} = (\texttt{a} + \texttt{b}) + \texttt{c};$$

where a, b, and c are arrays of identical shape. Inlining the definition of + leads to two subsequent WITH-loops of the form

```
tmp = with
        (iv) : a[iv] + b[iv]
      genarray( shape( a)));

res = with
        (iv) : tmp[iv] + c[iv]
      genarray( shape( tmp));
```

WITH-LOOP-FOLDING combines the two WITH-loops into a single one that performs both additions on the scalar level of array elements:

```
res = with
        (iv) : (a[iv] + b[iv]) + c[iv]
      genarray( shape( a));
```

The advantage of the folded code over the original code is basically two-fold. Firstly, we eliminate the temporary array tmp. This saves on runtime overhead incurred by memory management, namely one cycle of memory allocation and de-allocation. At the same time, we also eliminate one cycle of writing values into the temporary array and reading them from the array shortly thereafter. In particular if the size of the arrays involved exceeds the cache size requiring slow main memory interaction, the savings are usually significant. Secondly, the folded code is also advantageous

with respect to parallelisation. Considering that individual WITH-loops are the basis for multi-threaded data parallel execution each WITH-loop inflicts an expensive barrier synchronisation. Hence, in the above example WITH-LOOP-FOLDING also eliminates the need for one out of two barrier synchronisations.

WITH-LOOP-FOLDING is one prerequisite to make the compositional programming style advocated in Section 2.5 feasible in practice with respect to runtime performance. For the example of the compositional specification of a convergence criterion in Fig. 9 WITH-LOOP-FOLDING has the effect of condensing all four operations, subtraction, absolute value, greater equal, and any, into a single WITH-loop. Despite the compositional specification of `continue`, the compiled code computes the convergence criterion in a single step without introducing intermediate arrays or synchronisation barriers in the case of multi-threaded execution.

Technically spoken, WITH-LOOP-FOLDING aims at identifying array references within the generator-associated expressions in WITH-loops. If the index expression is an affine function of the WITH-loop's index variable and if the referenced array is itself defined by another WITH-loop, the array reference is replaced by the corresponding element computation. Instead of storing an intermediate result in a temporary data structure and taking the data from there when needed, we forward-substitute the computation of the intermediate value to the place where it is actually needed.

The challenge of WITH-LOOP-FOLDING lies in the identification of the correct expression which is to be forward-substituted. Usually, the referenced WITH-loop has multiple generators each being associated with a different expression. Hence, we must decide which of the index sets defined by the generators is actually referenced. To make this decision we must take into account the entire generator sequence of the referenced WITH-loop, the generator of the referencing WITH-loop that is associated with the expression which contains the array reference under consideration, and the affine function defining the index.

The top of Fig. 11 shows an example for a more general situation. The generator ranges of both WITH-loops do not cover the entire array. Instead, they overlap without one being included within the other. As a consequence, the result of the folding step requires the computation of the intersection of the generators. In order to be able to do this in a systematic way, we first default generators by a full partition of range generators. The result of this extension is shown in the middle part of Fig. 11. In case of the first WITH-loop, we obtain four additional generators. Similarly, we extend the second WITH-loop by two additional generators. At the same time, we turn the `modarray`-WITH-loop into a `genarray`-WITH-loop. Having replaced the implicit rule of the `modarray`-WITH-loop by

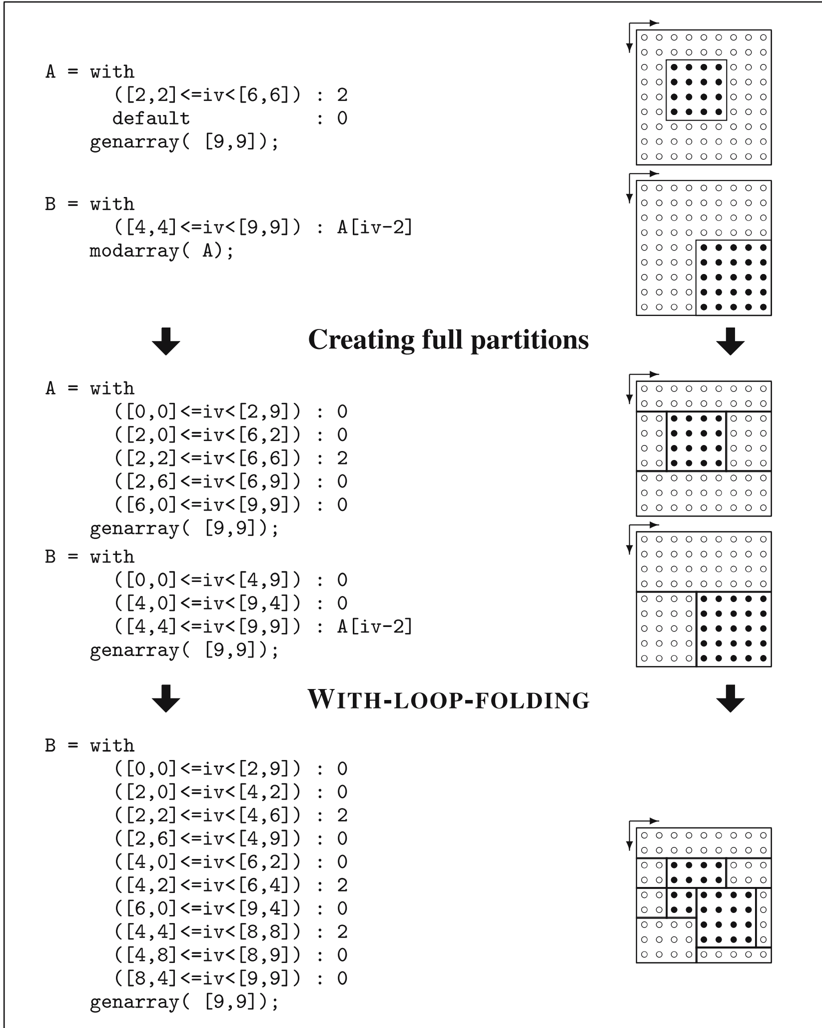


Fig. 11. WITH-LOOP-FOLDING in the general case.

explicit range generators, there is no need any more to distinguish between the two variants.

As pointed out before, WITH-LOOP-FOLDING identifies references to elements of WITH-loop-defined arrays within generator-associated expressions of other WITH-loops. In the example of Fig. 11 there is one potential optimisation case in each generator-associated expression of

the second WITH-loop. Let us focus on the first one. We need to figure out how `A[iv]` is defined for each element of the generator-defined index set. To do so systematically, we compute the intersection between the generator `([0,0]<=iv<[ 4, 9])` and each generator of the first WITH-loop. It turns out that the intersection with generator 5 of the first WITH-loop in fact is empty and can be ignored. However, the other four intersections are non-empty and lead to generators 1–4 of the folded WITH-loop at the bottom of Fig. 11. Proceeding with the second generator `([4,0]<=iv<[9,4])`, we find out that intersections with generators 2,3, and 5 are non-empty. These intersections lead to the new generators 5–7 of the folded WITH-loop.

An additional level of complexity arises as soon as the index expression is not the WITH-loop variable itself, but an affine function of `iv`, as in the third generator (`A[iv-2]`). In this case, we must adjust the generator under consideration by the inverse of the affine function prior to computing any intersections with the generators of the referenced WITH-loop. In the example, this leads to the remaining three generators of the folded WITH-loop.

As in the example of Fig. 11 WITH-LOOP-FOLDING may lead to quite complex index spaces, although in practice such examples are rather rare. Nevertheless, generation of efficiently executable code from complex generator sets is a challenging task, which is further complicated as soon as generators use `step` and `width` specifications for periodic grids. Therefore, we have developed code generation techniques to efficiently handle complex generator sets.(42) Their presentation, however, is beyond the scope of this paper. Likewise, a more thorough technical description of WITH-LOOP-FOLDING along with an investigation on runtime performance impact can be found in Ref. 28.

3.3. WITH-Loop Fusion

WITH-LOOP-FUSION addresses horizontal compositions of WITH-loops, i.e. WITH-loops without data dependences which have similar generator sets and preferably share references to the same argument arrays. Consider for example a function body where both, the maximum element and the minimum element of a given array `A` is computed. This can be specified as

```
minv = minval(A);
maxv = maxval(A);
```

Inlining the WITH-loop-based definitions for `minval` and `maxval` leads to

```
minv = with
         (iv < shape( A)): A[iv]
        fold( min, MaxInt());
```

```
maxv = with
       (iv < shape( A)): A[iv]
       fold( max, MinInt());
```

The idea of WITH-LOOP-FUSION is to combine such WITH-loops into a more versatile internal representation named *multi-operator* WITH-loop. The major characteristic of multi-operator WITH-loops is their ability to define multiple array comprehensions and multiple reduction operations as well as mixtures thereof. For the above example, we obtain:

```
minv, maxv = with
              (iv < shape( A)) : A[iv], A[iv]
             fold( min, MaxInt())
             fold( max, MinInt());
```

Prominent differences to WITH-loops as used so far are that each generator is associated with a sequence of expressions rather than a single one, Likewise, the WITH-loop features a sequence of operation parts. We assume that the number of generator-associated expressions is the same for each generator and coincides with the number of operation parts. In fact, each first generator-associated expression is connected to the first operation part, etc. Last but not least, the WITH-loop yields a sequence of values, one for each operation part, that need to be bound to variables collectively.

Whereas the original code is desirable from a software engineering perspective, the fused code has several advantages in terms of execution speed. Both values `minv` and `maxv` are computed in a single sweep. This allows us to share the overhead inflicted by the multi-dimensional loop nest. Furthermore, we change the order of array references at runtime. The intermediate code as shown above accesses the same array `A` in both WITH-loops. Assuming array sizes typical for numerical computing, elements of `A` are extremely likely not to reside in cache memory any more when they are needed for execution of the second WITH-loop. With the fused code both array references `A[iv]` occur in the same WITH-loop iteration and, hence, the second one always results in a cache hit. Onward optimisations, here common subexpression elimination, may remove the second memory access entirely and reuse the value that already resides in a register. With respect to parallelisation the elimination of one WITH-loop also eliminates the need for one barrier synchronisation at runtime.

Similar to WITH-LOOP-FOLDING WITH-LOOP-FUSION is one prerequisite to make the compositional programming style advocated in Section 2.5 feasible in practice with respect to runtime performance. Consider the convergence criterion example discussed in Section 2.5. In practice, the convergence criterion is used in conjunction with an iterative algorithm, each step of which is likely to be represented by one or more WITH-loops.

In this scenario WITH-LOOP-FUSION has the effect of integrating the code for computing one iteration step with the code for determining convergence in a single WITH-loop.

Similar to WITH-LOOP-FOLDING, the challenge of WITH-LOOP-FUSION lies in non-identical generator sets, which typically arise from preceding WITH-LOOP-FOLDING steps. Figure 12 shows a non-trivial example. In contrast to WITH-LOOP-FOLDING, the first subtask of WITH-LOOP-FUSION is the identification of suitable fusion candidates since they are unrelated in the data flow graph. Two WITH-loops are considered for fusion if they are data independent and their *aggregate index sets* are identical. The aggregate index set is the union of all generator-defined index sets of a WITH-loop. In the case of a `genarray`-WITH-loop, the aggregate index set is identical with the set of legal indices of the array to be created.

Having found suitable folding candidates, we systematically compute the pairwise intersections between each generator of the first WITH-loop and each generator of the second WITH-loop. In the worst case the number of generators in the fused WITH-loop equals the product of the numbers of generators in the original WITH-loops. However, in practice this case is rather rare since many intersections in fact turn out to be empty. Nevertheless, we take care to avoid fusion of WITH-loops that would lead to an explosion in the number of generators.

Fusion of WITH-loops whose aggregate index sets differ is also feasible; Fig. 13 shows an example. An additional preprocessing step is required that adjusts a WITH-loop's aggregate index space to the smallest rectangular hull of the union of its original aggregate index set and that of the fusion candidate. This preprocessing step introduces additional generators that are associated with the special (internal) expression `nothing`. It represents a dummy value since for the corresponding elements of the generator-defined index set there is no computation required. Although any computation is avoided for dummy indices, some loop overhead at runtime cannot. Hence, this form of WITH-LOOP-FUSION only makes sense if the aggregate index sets are sufficiently similar. As shown in Fig. 13, this form of WITH-LOOP-FUSION typically leads to index spaces, as represented by the set of generators, that differ from the shapes of the arrays defined. A more thorough technical description of WITH-LOOP-FUSION along with an investigation on runtime performance impact can be found in Ref. 30.

3.4. WITH-Loop Scalarisation

Since the generator-associated expressions of WITH-loops are in fact arbitrary SAC expressions, they may in particular again be WITH-loops. WITH-LOOP-SCALARISATION, our third high-level code transformation

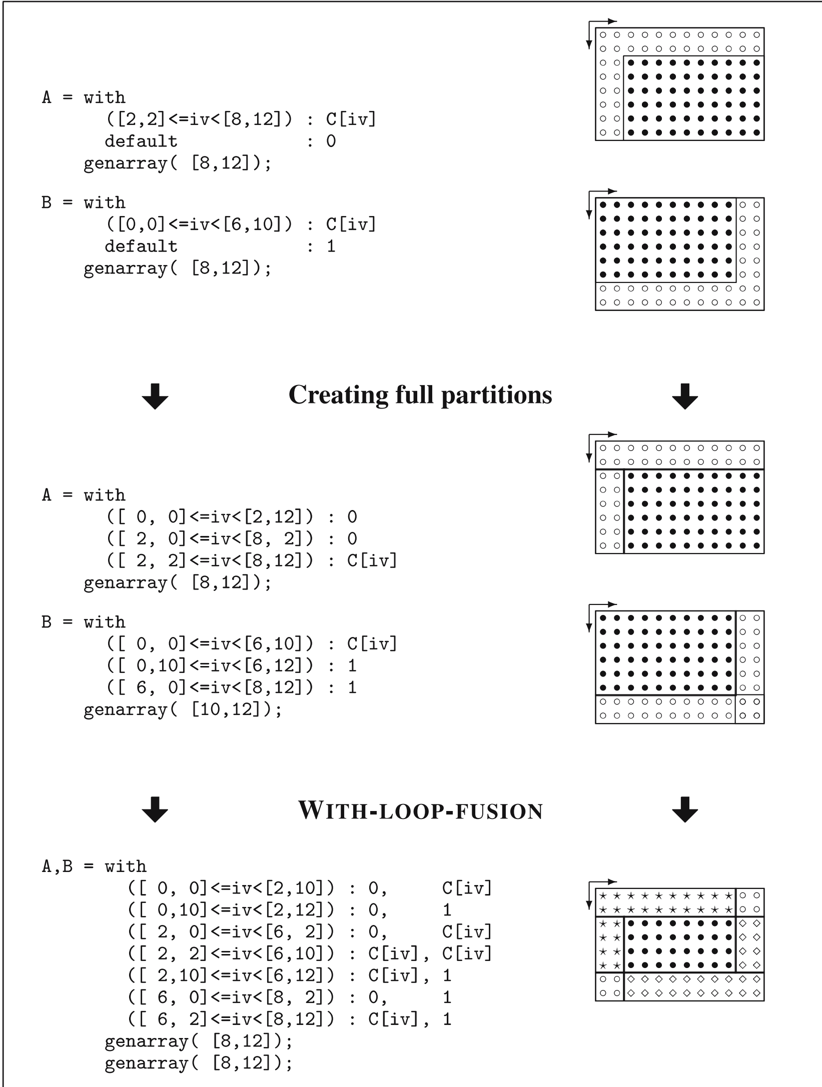


Fig. 12. WITH-LOOP-FUSION in the general case.

addresses nested compositions of WITH-loops. Such nestings occur frequently in practice as soon as elementary types of an array operation are not scalar, but turn out to be arrays themselves. For example, we may write the element-wise sum of two 10-element vectors `a` and `b` of complex numbers as

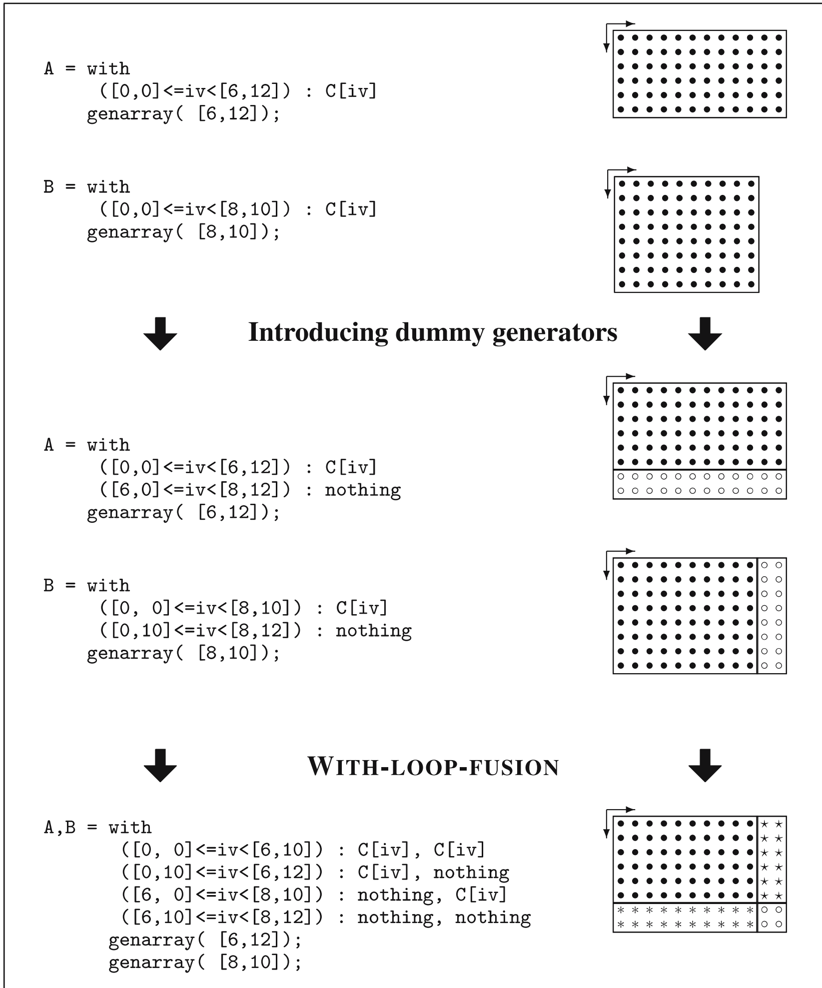


Fig. 13. WITH-LOOP-FUSION with non-identical aggregate index sets.

```
with
  ([0] <= iv < [10]): a[iv] + b[iv]
genarray( [10])
```

Whereas this definition looks very much like the element-wise sum on arrays of built-in scalar values, the selections `a[iv]` and `b[iv]` actually yield 2-element vectors of values of type `double`. Consequently, the operator + actually refers to the element-wise sum of arrays again rather than

to the built-in operation on scalars. Hence, inlining the inner application of + yields

```
with
 ([0] <= iv < [10]):
  with
   ([0] <= iv < [2]): (a[iv])[jv] + (b[iv])[jv]
  genarray( [2])
genarray( [10])
```

and, hence, a case of nested composition of WITH-loops. Assuming a straightforward compilation into machine code, we must allocate (and deallocate) memory to accommodate the inner vector for each element of the outer WITH-loop's aggregate index set. Furthermore, we must copy the values of the inner vector into the overall result array in each iteration of the outer WITH-loop, i.e., the result of the inner WITH-loop is a classical example of a temporary data structure.

The idea of WITH-LOOP-SCALARISATION is to eliminate nestings of WITH-loops and to transform them into single WITH-loops that operate on scalar values. We achieve this by concatenating the bound and shape expressions of the WITH-loops involved and by adjusting the generator variables accordingly. For our example we obtain

```
with
  ([0,0] <= iv < [10,2]) : a[iv] + b[iv]
genarray( [10])
```

When comparing this code against the non-scalarised version above we can observe several benefits. There are no more two-element vectors which results in less memory allocations and deallocations at runtime. Furthermore, the individual values are directly written into the result arrays without any copying from temporary vectors. With respect to multi-threaded execution the scalarised WITH-loop provides a larger index space right away that generally renders scheduling of workload to threads more effective.

After WITH-LOOP-FOLDING and WITH-LOOP-FUSION WITH-LOOP-SCALARISATION is the third prerequisite to make the compositional programming style advocated in Section 2.5 feasible in practice with respect to runtime performance. It allows us to use exactly the same compositional specification of, for instance, the convergence criterion on arrays of scalar elements as on arrays of arrays without paying a performance penalty for the layered specification.

Similar to WITH-LOOP-FOLDING and WITH-LOOP-FUSION the general case is more complicated than an introductory example. For example, the

presence of multi-generator WITH-loops requires us to define WITH-LOOP-SCALARISATION general enough to cope with multiple outer generators each being associated with different inner WITH-loop that again have multiple generators. Furthermore, arrays are not necessarily defined by means of WITH-loops. Many interesting optimisation cases, like the one sketched out above, are likely to use simple vector composition rather than a complex WITH-loop to define nested arrays. Such cases need a preprocessing transformation into a semantically equivalent WITH-loop before WITH-LOOP-SCALARISATION can be used to optimise the code.

A more thorough technical description of WITH-LOOP-SCALARISATION along with an investigation on runtime performance impact can be found in Ref. 29.

4. MULTI-THREADING

4.1. Parallelisation at a Glance

With generators defining sets rather than sequences of indices and with associated expressions that may be evaluated independently of each other and in any order, WITH-loops specify prototypical data parallel operations. Furthermore, the language design of SAC, which is characterised by implementing any aggregate array operation in SAC itself by means of WITH-loops, leads to an omnipresence of WITH-loops in intermediate SAC code. Last but not least, the various optimisation techniques sketched out in the previous section systematically improve the computational workload per array element during the compilation process. In conjunction, these properties make WITH-loops ideal candidates for parallelisation both with distributed memory as well as with shared memory architectures in mind. Notwithstanding the opportunity for genuine support of both architectural models or even mixtures thereof, we have so far focussed on compiler-directed parallelisation techniques for shared memory architectures based on multi-threading.

When contemplating the multi-threaded execution of SAC programs, it turns out that some program parts require sequential execution by a dedicated thread. For example, I/O operations must be performed in a single-threaded way in order to preserve the sequential I/O behaviour. I/O operations and state modifications in general are properly integrated into the purely functional world of SAC by a variant of uniqueness types,[43] named *classes*.[44] Enforcing the uniqueness property bans all state modifications from bodies of WITH-loops. Hence, the easiest way to preserve sequential I/O behaviour is to restrict parallel program execution to individual WITH-loops.

Following this idea, a *master thread* executes most parts of a SaC program in a single-threaded way, similar to sequential execution. Only when it comes to computing a WITH-loop, the master thread creates the desired number of additional *worker threads*. Subsequently, all worker threads concurrently, but cooperatively, execute the single WITH-loop. Meanwhile, the master thread merely awaits the termination of the worker threads. After all worker threads have finished their individual shares of work, they terminate, and the master thread resumes sequential execution.

4.2. Generating Multi-Threaded Code for Individual WITH-Loops

We illustrate the generation of multi-threaded code by means of a prototypical WITH-loop of the form

$$
\begin{array}{lclcl}
a,b,c & = & \texttt{with} & gen_1 & : \; g_1, m_1, f_1 \\
 & & & \vdots & \\
 & & & gen_n & : \; g_n, m_n, f_n \\
 & & \multicolumn{3}{l}{\texttt{genarray(}\; shp,\, def \;\texttt{)}} \\
 & & \multicolumn{3}{l}{\texttt{modarray(}\; old \;\texttt{)}} \\
 & & \multicolumn{3}{l}{\texttt{fold(}\; fop,\, neutr \;\texttt{);}}
\end{array}
$$

In order to demonstrate the different requirements of `genarray`-, `modarray`-, and `fold`-WITH-loops for parallelisation our prototypical WITH-loop is in fact a multi-operator WITH-loop with three different operation parts, one of each kind. While we abstract from a concrete number of generator/expression pairs, we use exactly these three operation parts in order to avoid an overly complicated representation. The extension of our scheme to arbitrary numbers of operation parts and arbitrary mixtures of the three kinds thereof is rather straightforward. We assume *shp, def, old,* and *neutr* to denote simple identifiers rather than complex expressions. In the case of SaC this is guaranteed by preceding compilation steps, namely the *functionalisation* phase, as described in Section 3.1. In contrast, g_i, m_i, f_i denote potentially complex expressions, which correspond to the three operation parts, `genarray`, `modarray`, and `fold`, respectively. Likewise, the three variables a, b, and c are bound to the values computed by the `genarray`-, `modarray`-, and `fold` operation parts, respectively.

Figure 14 shows the multi-threaded pseudocodes generated from this prototypical WITH-loop for the master thread (left hand side) and for worker threads (right hand side). We indicate the interaction between master thread and worker threads by horizontal arrows. Whenever execution of the master thread reaches a WITH-loop, the master thread first

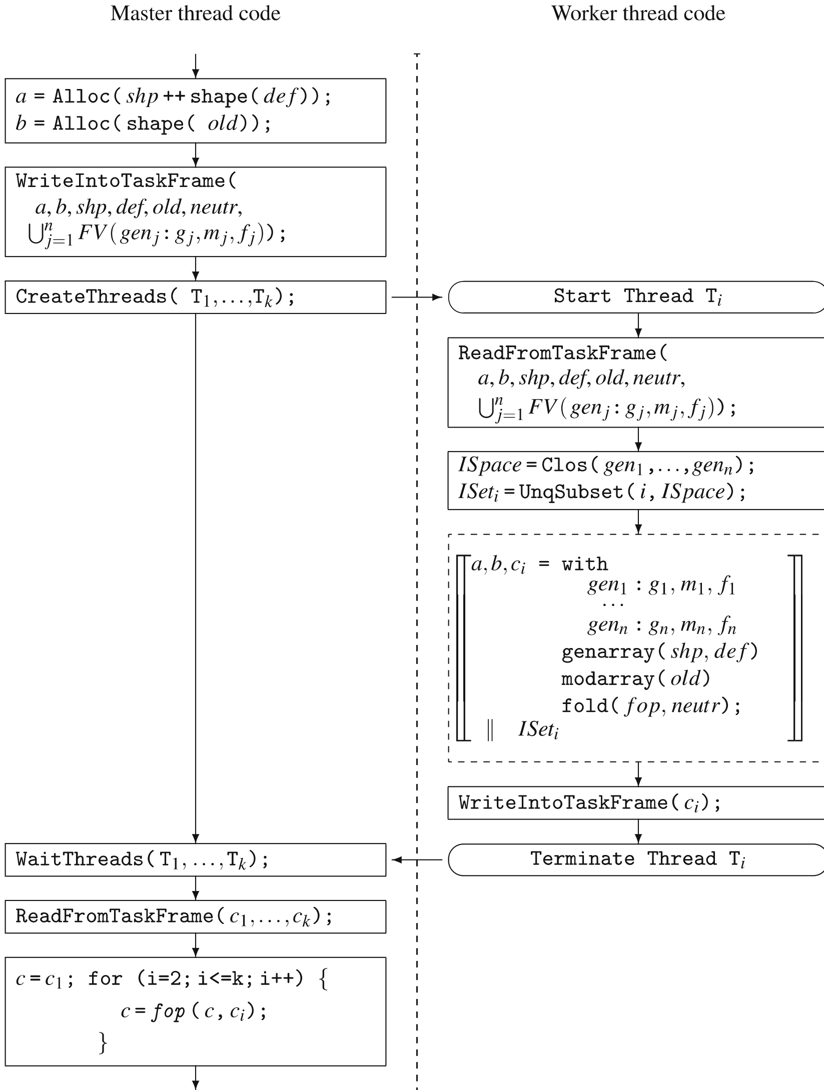


Fig. 14. Compilation of WITH-loops into multi-threaded pseudo code.

allocates memory appropriate for storing the result arrays, i.e. the result values associated with `genarray` and `modarray` operation parts. While in the `modarray` case the shape of the result array can simply be adopted from the referenced array, the `genarray` case requires us to concatenate the given result shape with the shape of the elements. Since we enforce

uniformity of subarrays, we can use the shape of the default element as a representative. Due to the commitment to shared memory architectures, explicit decomposition of arrays is obsolete. Hence, implicit dynamic memory management for arrays can be adopted from the existing sequential implementation with only minor adaptations. Actually, arrays are still stored in a single contiguous address space.

To enable worker threads to cooperatively participate in the computation of a WITH-loop we must first set-up an execution environment equivalent to that of the master thread. More precisely, all those variables referred to within the body of the WITH-loop, but defined before, must also exist in the worker thread. Moreover, they must be initialised to the current values in the master thread to ensure proper execution of the WITH-loop by each worker thread. The required information must be communicated to the worker threads before they start execution of productive code.

In a shared memory environment communication between threads can be realised by writing to and reading from global data structures. Therefore, each WITH-loop in a SAC program is associated with a global buffer, called *task frame*. The task frame is tailor-made for communication between master thread and worker threads for that individual WITH-loop. The master thread stores the values of all free variables of the various generator-expression pairs in the task frame. Additionally, we store some more information specific to the operation parts: the base addresses of memory allocated to store the result arrays of `genarray` and `modarray` operations, the result shapes and the default values of `genarray` operations, the referenced arrays of `modarray` operations, and the neutral elements of `fold` operations. It is important to note here that due to the shared memory environment we only communicate array descriptors, rather than the arrays themselves. We do not store the fold operation in the task frame because for the time being SAC does not support higher-order functions in general. Hence, the fold operation is not a computable expression, but literally the name of an existing function or operator.

Eventually, the desired number of worker threads is created. The actual number may either be fixed at compile time, or it is determined by a command line parameter or by an environment variable upon program start. In any case, the number of threads remains constant throughout an entire program run. All worker threads uniformly execute the code shown on the right hand side of Fig. 14, but each thread may identify itself by means of a unique ID. As a first step, a worker thread sets up its execution environment by extracting the required information from the task frame. Access to the task frame can always be granted without costly synchronisation of threads via critical regions or similar. As long as the

master thread writes to the task frame, it is known to be the only thread in the process. As soon as multiple threads exist, the task frame is used as a read-only buffer.

Although all worker threads execute exactly the same code, they must address pairwise disjoint subsets of the overall index space for parallel execution to make sense. This additional side condition must be taken into account when eventually compiling a WITH-loop into nestings of FOR-loops in C. Even in the sequential case, generation of efficient code from WITH-loops with multiple interleaved generators has proved to be an extremely complicated and challenging task.[25,42] Therefore, we want to reuse the existing sequential code generation scheme for WITH-loops as far as possible. However, we need to associate each element of the result array with exactly one thread. The *scheduling* of array elements to threads directly affects workload distribution among processors and is vital for performance.

The solution adopted in Fig. 14 keeps code generation and scheduling as separate as possible. As a first step, each worker thread determines the overall index space of the WITH-loop by means of a system function `Clos`, which yields the smallest rectangular hull of the transitive closure of the generators. Then, each worker thread calls the system function `UnqSubset` to identify a rectangular index subspace $IdxSet_i$ based on its unique ID and the index space $ISpace$. Proper implementations of `UnqSubset` guarantee that each index is covered by exactly one such index subspace. Different implementations of `UnqSubset` allow us to realise various different scheduling techniques without affecting the compilation scheme otherwise. Based on programmer annotations or on compiler heuristics, the most appropriate scheduler implementation may be selected.

After initialisation of the execution environment and allocation of a rectangular index subspace by the loop scheduler, execution of the WITH-loop by an individual worker thread proceeds almost as in the sequential case. In Fig. 14, this is indicated by the recursive application of code generation. The main difference between generation of multi-threaded and generation of sequential code here is that we restrict the range of any FOR-loop in compiled code to the intersection between its original range and the index subspace $IdxSet_i$ allocated by the WITH-loop scheduler.

In the case of `genarray` and `modarray` operations our scheduling mechanism suffices to let the worker threads compute and initialise the resulting array allocated by the master thread. For `fold` operations, however, each worker thread initialises a local accumulation variable, denoted c_i in Fig. 14, by the neutral element of the fold operation. Consequently, each worker thread computes a partial fold result, which upon completion it writes back into the task frame. In order to avoid costly synchronisation

upon concurrent access to the task frame by worker threads, the task frame provides a dedicated entry for each worker thread.

While worker threads terminate after having completed their task, the master thread awaits termination of all worker threads. Only then, it extracts the partial fold results from the task frame, and combines them to sequentially generate the overall fold result. Eventually, the master thread resumes sequential execution of the program.

4.3. Enhancing the Execution Model

The compilation scheme defined in Section 4.2, addresses individual WITH-loops. In compiled code, there is no connection between multi-threaded execution of consecutive WITH-loops. This limited scope facilitates the definition of a basic compilation scheme, yet it excludes any optimisation across multiple WITH-loops. Following the multi-threaded execution of one WITH-loop, all worker threads terminate during synchronisation. The same number of worker threads is again created for the multi-threaded execution of the following WITH-loop. Program execution is a sequence of steps alternatingly performed in single-threaded and in multi-threaded mode, as illustrated on the left hand side of Fig. 15.

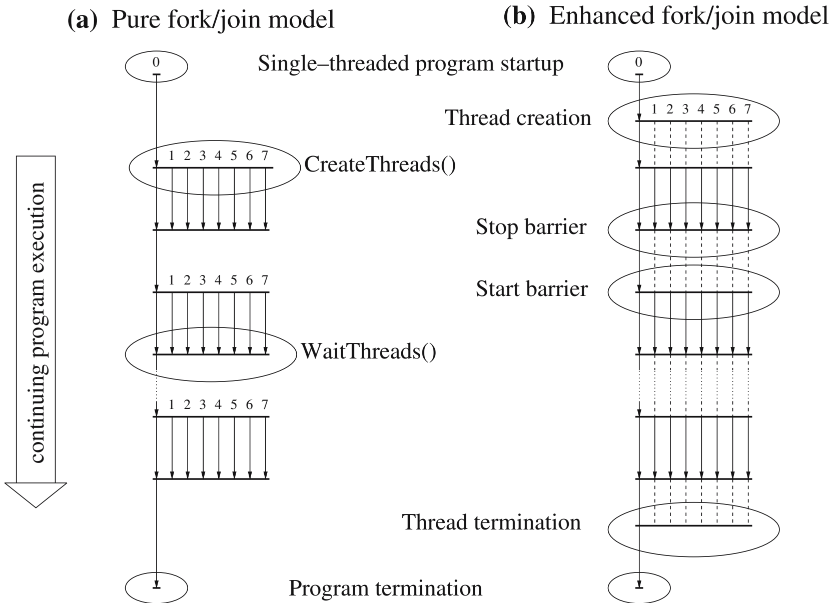


Fig. 15. Comparison of pure and enhanced fork/join execution models.

This fork/join execution model is conceptually simple. Synchronisation and communication events are confined to thread creation and thread termination. Worker threads do not interact with each other at all. However, the price for simplicity is excessive runtime overhead due to frequent creation and termination of threads. Although the associated costs are much smaller than those for process creation and termination, they are still prohibitive for successful parallelisation.

There are basically two approaches to reduce runtime overhead without leaving the overall model of organisation. Firstly, we may improve the implementation of thread creation and termination, e.g. by caching worker threads during periods of sequential execution and by employing efficiently implemented synchronisation and communication constructs instead.(33) Secondly, we may combine multiple data independent instances of skeletons into a single instance of some more versatile meta skeleton.(45) While the first approach reduces the costs associated with each individual such event, the second approach reduces the number of costly synchronisation and communication events. We proceed with discussing the former approach while the latter is addressed in Section 4.4.

A solution which combines the conceptual simplicity of the fork/join approach with an efficient execution scheme is shown on the right hand side of Fig. 15. In the *enhanced fork/join model*, the desired number of worker threads is created once at program start, and all threads remain active until the whole program terminates. Two tailor-made barriers, the *start barrier* and the *stop barrier*, realise all necessary synchronisation among threads.

After creation, worker threads immediately hit a start barrier. As soon as the master thread encounters the first WITH-loop, the start barrier is lifted. The worker threads thereupon activated share the computation of the WITH-loop, exactly as in the pure fork/join model. Unlike in the pure fork/join model, the master thread temporarily turns itself into a worker thread and joins the other threads in the cooperative execution of the WITH-loop. Regular worker threads that have completed their individual computations pass the following stop barrier and, with nothing else to do, immediately move on to the next start barrier. After having finished its own assignment of work, the master thread waits at the stop barrier for the longest-running worker thread to arrive. Only then the master thread proceeds with subsequent (sequential) computations.

In the absence of particular hardware support for efficient multi-threading (i.e. on standard shared memory multi-processor systems) thread creation and termination are costly operations as they typically require operating system interaction. The enhanced fork/join model may be thought of as caching threads while they are not productively computing.

The applications we consider typical for SaC spend the overwhelming part of their execution time computing WITH-loops. As a consequence, the time spent executing sequential code in between two WITH-loopsis negligible. This justifies the continuous allocation of resources even when they are not effectively used for very short periods of time.

4.4. SPMD Optimisation

The enhanced fork/join execution model significantly reduces synchronisation costs by replacing thread creation and thread termination by less expensive start and stop barriers, respectively. Nevertheless, program execution stalls at each stop barrier until arrival of the longest-running worker thread, and start and stop barriers are still major sources of overhead. An orthogonal approach to optimising multi-threaded program execution is to reduce the number of synchronisations performed at runtime. Our aim is to create larger program sections of multi-threaded execution without intercepting synchronisation and communication events. Besides avoiding the cost immediately associated with the execution of the barrier code and the need to wait for the longest-running worker thread, larger regions of parallel execution also render loop scheduling techniques more effective.

In principle, WITH-LOOP-FOLDING and WITH-LOOP-FUSION have exactly this effect. However, their applicability is restricted by various side conditions, which often could be neglected when solely having synchronisation requirements in mind. The main problem here is the fact that WITH-loops describe both a computational task and a coordination behaviour, i.e. the organisation of the parallel execution of the given task by multiple threads. Creating regions of parallel execution that stretch over several WITH-loops which neither permit WITH-LOOP-FOLDING nor WITH-LOOP-FUSION requires explicit separation of these two concerns.

Therefore, we introduce *SPMD skeletons* as intermediate representations of the coordination behaviour of regions of parallel execution. Within such regions, which may contain multiple WITH-loops, program execution follows the "Single Program, Multiple Data" approach, hence the name. As a starting point, each WITH-loop that is to be executed in parallel is internally embedded within an SPMD skeleton. Then, we systematically aim at identifying SPMD skeletons without data dependences. Any two such SPMD skeletons are merged into a single one containing all WITH-loops embedded in the former individual skeletons. This optimisation step is only restricted by data dependences and not by additional properties of the generator sets as is the case with both WITH-LOOP-FOLDING and WITH-LOOP-FUSION. In the further compilation process all code responsible for coordination of

threads and communication between threads is derived from the SPMD skeletons.

5. CONCLUSION

We have presented the design and sketched out the implementation of the functional array programming language SaC. The paper gives a comprehensive account of the interplay between language design, compiler optimisation, and generation of multi-threaded code. It is this careful interplay that allows us to compile high-level declarative SaC programs into executable code whose runtime performance is competitive with that of low-level machine-oriented solutions.

A SaC-specific language construct called WITH-loop takes a key role in this task. Its carefully chosen design provides the grounds for specificational power combined with suitability for program reorganisation as well as code generation for multi-threaded execution. Besides an extensive introduction into the expressive power of WITH-loop the paper explains the three most important WITH-loop optimisations. They transform compositions of many simple WITH-loops, as they result from a compositional programming style, into few though significantly more complex WITH-loops. Furthermore, we outline the most important organisational measures that are required to compile such complex WITH-loops into multi-threaded code.

High-level program transformations and generation of multi-threaded code are no isolated issues. In fact, they depend on each other to achieve the overall goal of high runtime performance. On the one hand, the program transformations are mandatory to achieve high sequential performance. Without them parallelisation would start from uncompetitive performance levels and, hence, would hardly justify the effort. In fact, high-level program transformations account for many optimisations that in other approaches are dealt with when generating parallel code. They (silently) eliminate the need for synchronisation and communication and identify coarser-grained parallelisation candidates. On the other hand, fully implicit, compiler-directed parallelisation gives the extra competitive edge in terms of performance, despite the high-level declarative style of programming.

Although the proposed techniques — in principle — can be applied in an imperative setting as well, it is doubtful whether the same overall effectiveness can be achieved. As can be seen from the transformations presented in the paper, it is crucial for all these measures that expressions can be moved rather freely within functions if not the entire program. While the opportunities for such movements can be easily detected statically in the

functional setting, an imperative setting is far more restrictive. There, slight program modifications may introduce potential side-effects that inhibit further optimisations. Furthermore, explicit storage declarations often prevent optimisations which, in the functional context, due to the implicit memory management can be done.

Due to space limitations we have neither discussed any larger example of a SaC program nor have we shown any runtime performance figures. In previous publications we have investigated several case studies in-depth substantiating our above claims to achieve competitive runtimes despite a declarative style of programming. The interested reader is referred to Refs. 34–36 for additional information of this kind.

REFERENCES

1. High Performance Fortran Forum: High Performance Fortran Language Specification V2.0. (1997).
2. B. Chamberlain, S. E. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby, ZPL: A Machine Independent Programming Language for Parallel Computers, *IEEE Transactions on Software Engineering* **26**:197–211 (2000).
3. D. Cann, Retire Fortran? A Debate Rekindled, *Communications of the ACM,* **35**:81–89 (1992).
4. L. Dagum, and R. Menon, OpenMP: An Industry-Standard API for Shared-Memory Programming, *IEEE Transactions on Computational Science and Engineering,* **5**:46–55 (1998).
5. H. Zima, and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley, Reading, Massachusetts, USA (1991).
6. M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison-Wesley, Reading, Massachusetts, USA (1995).
7. R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon, The Combined Effectiveness of Unimodular Transformations, Tiling, and Software Prefetching, In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, Hawaii, USA, pp. 39–45 (1996).
8. B. Chamberlain, E. Lewis, C. Lin, and L. Snyder, Regions: An Abstraction for Expressing Array Computation, In O. Levefre, (ed.), *Proceedings of the International Conference on Array Processing Languages (APL'99)*, Scranton, Pennsylvania, USA. Volume 29 of APL Quote Quad., ACM Press pp. 41–49 (1999).
9. R. Allen, and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers (2001).
10. H. Zima, High-Level Programming Support for HPC – The Tradeoff between Elegance and Performance, In *Proceedings of the 19th International Supercomputer Conference (ISC'01)*, Heidelberg, Germany (2001).
11. S. P. Jones, *Haskell 98 Language and Libraries*, Cambridge University Press, Cambridge, UK (2003).
12. M. Plasmeijer, and M. van Eekelen, Concurrent Clean 2.0 Language Report. University of Nijmegen, The Netherlands (2001).
13. J. van Groningen, The Implementation and Efficiency of Arrays in Clean 1.1, In W. Kluge (ed.), *Proceedings of the 8th International Workshop on Implementation of*

Functional Languages (IFL'96), Bonn, Germany, Selected Papers, Volume 1268 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany pp. 105–124 (1997).
14. P. Serrarens, Implementing the Conjugate Gradient Algorithm in a Functional Language, In W. Kluge (ed.), *Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL'96)*, Bonn, Germany, Selected Papers, Volume 1268 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany pp. 125–140 (1997).
15. T. Zörner, Numerical Analysis and Functional Programming, In K. Hammond, T. Davie, and C. Clack (eds.), *Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98)*, London, UK, University College, London, UK, pp. 27–48 (1998).
16. M. M. Chakravarty, and G. Keller, An Approach to Fast Arrays in Haskell, In J. Jeuring, and S. P. Jones (eds.), *Summer School and Workshop on Advanced Functional Programming*, Oxford, England, UK, (2002), Volume 2638 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany pp. 27–58 (2003).
17. P. Hartel, and K. Langendoen, Benchmarking Implementations of Lazy Functional Languages, In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, Copenhagen, Denmark, ACM Press, New york, pp. 341–349 (1993).
18. P. Hartel et al., Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark, *Journal of Functional Programming* **6**:621–655 (1996).
19. J. Hammes, S. Sur, and W. Böhm, On the Effectiveness of Functional Language Features: NAS Benchmark FT, *Journal of Functional Programming* **7**:103–123 (1997).
20. P. Hudak, and A. Bloss, The Aggregate Update Problem in Functional Programming Systems, In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, New Orleans, Louisiana, USA, ACM Press, New York, pp. 300–313 (1985).
21. E. Barendsen, and S. Smetsers, Uniqueness Type Inference, In M. Hermenegildo, and S. Swierstra (eds.), *Proceedings of the 7th International Symposium on Programming Language Implementation and Logic Programming (PLILP'95)*, Utrecht, The Netherlands, Volume 982 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany, pp. 189–206 (1995).
22. P. Wadler, The Essence of Functional Programming, In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico, USA, ACM Press, New York, pp. 1–14 (1992).
23. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts, USA (1990).
24. A. Falkoff, and K. Iverson, The Design of APL. *IBM Journal of Research and Development* **17**:324–334 (1973).
25. S. B. Scholz, Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting, *Journal of Functional Programming* **13**:1005–1059 (2003).
26. M. Jenkins, Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial, *Software Practice and Experience* **19**:111–126 (1989).
27. K. Iverson, Programming in J. Iverson Software Inc., Toronto, Canada (1991).
28. S. B. Scholz, A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in SAC, In K. Hammond, T. Davie, and C. Clack (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, pp. 216–228 (1999).

29. C. Grelck, S. B. Scholz, and K. Trojahner, With-Loop Scalarization: Merging Nested Array Operations, In P. Trinder, G. Michaelson (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).
30. C. Grelck, K. Hinckfuß, S. B. Scholz, With-Loop Fusion for Data Locality and Parallelism, In A. Butterfield (ed.), *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05)*, Dublin, Ireland. Volume 4015 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany (2006).
31. C. Grelck, Shared Memory Multiprocessor Support for SAC, In K. Hammond, T. Davie, and C. Clack (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, pp. 38–54 (1999).
32. C. Grelck, A Multithreaded Compiler Backend for High-Level Array Programming, In M. 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)*, Innsbruck, Austria, ACTA Press, Anaheim, California, USA, pp. 478–484 (2003).
33. C. Grelck, Shared Memory Multiprocessor Support for Functional Array Processing in SAC, *Journal of Functional Programming* **15**:353–401 (2005).
34. C. Grelck, and S. B. Scholz, HPF vs. SAC — A Case Study, In A. Bode, T. Ludwig, W. Karl, and R. Wismüller (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, pp. 620–624 (2000).
35. C. Grelck, and S. B. Scholz, Towards an Efficient Functional Implementation of the NAS Benchmark FT, In V. Malyshkin (ed.), *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03)*, Nizhni Novgorod, Russia. Volume 2763 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany pp. 230–235 (2003).
36. A. Shafarenko, S. B. Scholz, S. Herhut, C. Grelck, and K. Trojahner, Implementing a Numerical Solution for the KPI Equation using Single Assignment C: Lessons and Experience, In A. Butterfield, (ed.), *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05)*, Dublin, Ireland. Volume 4015 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany (2006).
37. H. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, Volume 103 of Studies in Logics and the Foundations of Mathematics. North Holland, Amsterdam, The Netherlands (1981).
38. D. C. Cann, Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, Livermore, California, USA (1989).
39. C. Grelck, and K. Trojahner, Implicit Memory Management for SaC, In C. Grelck, and F. Huch (eds.), *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, University of Kiel, pp. 335–348 (2004).
40. M. Jenkins, and P. Falster, Array Theory and NIAL. Technical Report 157, Technical University of Denmark, ELTEK, Lyngby, Denmark (1999).
41. A. Appel, SSA is Functional Programming, *ACM SIGPLAN Notices* **33**:17–20 (1998).
42. C. Grelck, D. Kreye, and S. B. Scholz, On Code Generation for Multi-Generator WITH-Loops in SAC, In P. Koopman, C. Clack, (eds.), *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'99)*, Lochem,

The Netherlands, Selected Papers. Volume 1868 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 77–94 (2000).

43. P. Achten, and M. Plasmeijer, The Ins and Outs of Clean I/O, *Journal of Functional Programming* **5**:81–110 (1995).
44. C. Grelck, and S. B. Scholz, Classes and Objects as Basis for I/O in SAC, In T. Johnsson (ed.), *Proceedings of the 7th International Workshop on Implementation of Functional Languages (IFL'95)*, Båstad, Sweden, Chalmers University of Technology, Gothenburg, Sweden, pp. 30–44 (1995).
45. C. Grelck, Optimizations on Array Skeletons in a Shared Memory Environment, In T. Arts, and M. Mohnen (eds.), *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01)*, Stockholm, Sweden, Selected Papers. Volume 2312 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, pp. 36–54 (2002).