On Code Generation for Multi-Generator With-Loops in SAC

Clemens Grelck, Dietmar Kreye, and Sven-Bodo Scholz

University of Kiel
Department of Computer Science and Applied Mathematics
D-24098 Kiel, Germany
E-mail: {cg,dkr,sbs}@informatik.uni-kiel.de

Abstract. Most array operations in SAC are specified in terms of so-called with-loops, a SAC-specific form of array comprehension. Due to the map-like semantics of with-loops its loop instances can be computed in any order which provides considerable freedom when it comes to compiling them into nestings of for-loops in C. This paper discusses several different execution orders and their impact on compilation complexity, size of generated code, and execution runtimes. As a result, a multiply parameterized compilation scheme is proposed which achieves speedups of up to a factor of 16 when compared against a naïve compilation scheme.

1 Introduction

SAC is a functional C-variant that is particularly aimed at numerical applications involving complex array operations. To allow for a fairly high level of abstraction, SAC supports so-called shape-invariant programming, i.e., all operations/functions can be defined in a way that allows array arguments to have arbitrary extents in an arbitrary number of dimensions. The main language construct for specifying such array operations is the so-called with-loop, a form of array comprehension adjusted to the needs of shape-invariant programming.

In [20] it has been shown that the array concept of SAC is suitable for specifying reasonably complex array operations in a shape-invariant style. It has also been shown that such specifications can be compiled into code whose runtimes are competitive with those obtained from rather low-level specifications in other languages such as SISAL or FORTRAN.

However, the ability to specify a set of small but fairly general array operations in a shape-invariant form turns out to be a very powerful programming tool in itself, irrespective of whether an entire application is shape-invariant or not. It allows for the definition of basic array operations similar to the built-in operations of languages such as APL, J [2], NIAL [10], or FORTRAN-90 within SAC itself [9]. Placed into SAC libraries, these operations may serve as building blocks for real world applications, making their definitions more concise, less error-prone, and more comprehensible. From the language design perspective, this approach has a twofold benefit: the maintainability as well as the extensibility of SAC's array subsystem are improved while the language itself can be kept rather concise allowing for a lean compiler design.

Typically, such specifications introduce many intermediate arrays. To achieve competitive runtimes, powerful optimization techniques are required that avoid the actual creation of these intermediate arrays as far as possible. While in an imperative setting, e.g. in HPF, this task turns out to be difficult [17,13,18], it can be done more easily in SAC by applying so-called with-loop-folding. In [21] it is shown for several programs written in APL-style that with-loop-folding is able to eliminate large numbers of intermediate arrays. In fact, the

resulting code is almost identical to what can be accomplished for programs that directly implement the desired functionality in an element-wise manner rather than benefiting from an APL-like programming style.

While `with`-loops, as they are used in SAC programs, specify a single operation on a single set of index vectors, the result of `with`-loop-folding in general requires several different operations to be applied on different sets of index vectors. This more general form of `with`-loop is called multi-generator `with`-loop. In fact, their index vector sets constitute a partition of all legal indices. Like most array comprehensions in other functional languages, `with`-loops do have a map-like semantics, i.e., all instances of a `with`-loop can be computed in arbitrary order without affecting the overall result of the computation. Due to the underlying functional paradigm, it is guaranteed that `with`-loop-folding preserves this property. As a consequence, the SAC compiler may choose any execution order that seems suitable with respect to code complexity, loop overhead, data locality, and cache performance including non-sequential execution schemes [8].

The aim of this paper is to develop a scheme for compiling multi-generator `with`-loops into efficiently executable C code. Since the runtime performance of computations on large arrays critically depends on an efficient utilization of the target architecture's cache(s) [12,14,23,7,16], several different approaches are discussed with respect to their cache behavior. The idea of the so-called canonical order is introduced which requires a sophisticated compilation scheme that merges computations on intertwined grids into more linear memory accesses. To further improve cache locality, the proposed compilation scheme is parameterized by several pragmas. They control compiler-introduced tiling by explicitly annotating desired tile sizes which in a later stage may become compiler-inferred.

After a short introduction to `with`-loops and `with`-loop-folding in Section 2, Section 3 presents a straightforward compilation scheme for multi-generator `with`-loops. The re-ordering of operations on intertwined vector sets is described in Section 4. Section 5 discusses effects of this re-ordering on the size of the generated C code and introduces the idea of smaller units of compilation, so-called segments and cubes. Section 6 adds parameterizations to the compilation scheme which support tiling. Some preliminary performance figures are presented in Section 7. Section 8 tries to put the presented work into perspective with existing work on re-ordering array accesses for generating efficiently executable code. Section 9 concludes and sketches perspectives for future research.

2 `With`-Loops and `With`-Loop-Folding

SAC only provides a very small set of built-in array operations, basically primitives to retrieve data pertaining to the structure and contents of arrays, e.g. dimensionality, shape, or element selection. Other array operations can be specified using `with`-loops. A `with`-loop typically defines an entire array along with a specification of how to compute each array element depending on its index position. In this regard, `with`-loops are similar to array comprehensions in HASKELL or CLEAN and to the `for`-loops in SISAL. However, `with`-loops in SAC allow the specification of truely shape-invariant array operations, i.e., not only the extent of argument or result arrays may vary in some dimensions but also the number of dimensions itself.

The syntax of `with`-loops is outlined in Fig. 1. A `with`-loop basically consists of two parts: a generator part and an operation part. The generator part defines a set of index vectors along with an index variable representing elements of this set. Two expressions that must evaluate to vectors of equal length define the lower and the upper bound of a rectangular range of index vectors. This set of index vectors may be further restricted by an optional filter to

WithExpr ⇒ `with (` *Generator* `)` *Operation*

Generator ⇒ *Expr* `<=` *Id* `<` *Expr* [`step` *Expr* [`width` *Expr*]]

Operation ⇒ [`{` *LocalDeclarations* `}`] *ConExpr*

ConExpr ⇒ `genarray (` *Expr* `,` *Expr* `)`
| `modarray (` *Expr* `,` *Expr* `,` *Expr* `)`
| `fold (` *FoldFun* `,` *Expr* `,` *Expr* `)`

Fig. 1. The syntax of `with`-loops.

define grids of arbitrary stride and width. More precisely, let a, b, s, and w denote expressions that evaluate to vectors of length n, then

$$\texttt{(}\ a\ \texttt{<=}\ i\_vec\ \texttt{<}\ b\ \texttt{step}\ s\ \texttt{width}\ w\ \texttt{)}$$

denotes the following set of index vectors:

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

The operation part specifies the operation to be performed for each element of the index vector set. There are three different operation parts; their functionalities are defined as follows. Let *shp* and *i_vec* denote SAC-expressions that evaluate to vectors, *array* denote a SAC-expression that evaluates to an array, and *expr* denote an arbitrary SAC-expression. Moreover, let *fold_op* be the name of a binary commutative and associative function with neutral element *neutral*. Then

- `genarray(` *shp* `,` *expr*`)` generates an array of shape *shp* whose elements are the values of *expr* for all index vectors from the specified set, and 0 otherwise;
- `modarray(` *array* `,` *i_vec* `,` *expr*`)` defines an array of shape `shape(` *array*`)` whose elements are the values of *expr* for all index vectors from the specified set, and the values of *array*`[`*i_vec*`]` elsewhere;
- `fold(` *fold_op* `,` *neutral* `,` *expr*`)` specifies a reduction operation. Starting off with *neutral*, the value of *expr* is computed for each index vector from the specified set and these are subsequently folded using *fold_op*. Note here that the associativity and commutativity of *fold_op* guarantees deterministic results irrespective of a particular evaluation order.

The readability of complex goal expressions can be improved by adding a block of local declarations between the generator and the operation part and define the goal expression in terms of these variables.

`With`-loop-folding [21] is a SAC-specific optimization technique that is based on the well-known equivalence

$$\texttt{(map f)} \circ \texttt{(map g)} \quad \Longleftrightarrow \quad \texttt{map (f} \circ \texttt{g)} \qquad .$$

Its purpose is to avoid the creation of intermediate arrays by condensing consecutive `with`-loops into a single one. A simple `with`-loop-folding example is shown in Fig. 2, a function which computes the scalar product of two integer arrays of arbitrary shape. It is defined in terms of the functions `sum` and `*` from the SAC array library (a). Inlining these functions results in two consecutive `with`-loops (b); subsequent `with`-loop-folding transforms them into a single one (c).

The example in Fig. 2 represents only the most trivial case of `with`-loop-folding as the generators of both `with`-loops cover the entire array to be created. However, `with`-loop-folding in SAC is much more general than the MAP-equivalence in that it also allows

(a)
```
int[] scal_prod( int[] A, int[] B) {
  return( sum(A*B));
}
```

(b) $\Longrightarrow$ (inlining)
```
int[] scal_prod( int[] A, int[] B) {
  tmp = with (0*shape(A) <= iv < shape(A))
        genarray( shape(A), A[iv] * B[iv]);
  res = with (0*shape(A) <= iv < shape(A))
        fold( +, 0, tmp[iv]);
  return( res);
}
```

(c) $\Longrightarrow$ (wl-folding)
```
int[] scal_prod( int[] A, int[] B) {
  res = with (0*shape(A) <= iv < shape(A))
        fold( +, 0, A[iv] * B[iv]);
  return( res);
}
```

Fig. 2. Example for `with`-loop-folding.

to fold `with`-loops whose generators are restricted to subranges or grids. Moreover, the generators of two subsequent `with`-loops may even be different from each other. The latter case results in a situation where different array elements have to be computed according to different specifications. This cannot be expressed by a single `with`-loop, i.e., `with`-loops are not closed with respect to folding. To address this problem, user-level `with`-loops are internally embedded into a more general representation that allows arbitrary `with`-loop-folding: multi-generator `with`-loops.

```
with ([  0,  0] <= iv < [140,200]                        ): op1
     ([140,  0] <= iv < [320,200] step [1,2]             ): op1
     ([140,  1] <= iv < [320,200] step [1,2]             ): op2
     ([  0,200] <= iv < [320,400] step [9,1] width [2,1]): op2
     ([  2,200] <= iv < [320,400] step [9,1] width [7,1]): op1
```

Fig. 3. Example of a multi-generator `with`-loop.

Figure 3 shows an example of a multi-generator `with`-loop. Instead of a single generator it consists of a whole sequence of generators each associated with an individual goal expression ($\texttt{op}_n$). This may again be preceded by a block of local declarations. By definition, the index vector sets of the various generators are disjoint and completely cover the set of legal indices. In order to guarantee this property, multi-generator `with`-loops are not part of the language but compiler generated only.

As a consequence of `with`-loop-folding, a single multi-generator `with`-loop may represent a complex array operation that performs different computations on different parts of an array. This complexity is particularly increased by generators that use `step` or `width` specifications to define grids in addition to rectangular index vector subranges. Therefore, graphical representations as the one shown in Fig. 4 for the `with`-loop introduced in Fig. 3 are used in the sequel to illustrate 2-dimensional multi-generator `with`-loops.

This `with`-loop creates a 2-dimensional array of shape `[320,400]`. All array elements in the range `[0,0]` $\rightarrow$ `[140,200]` are computed by `op1` (1st generator from Fig. 3). Columns

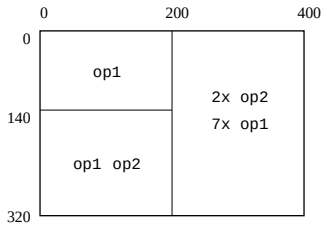

Fig. 4. Graphical representation of multi-generator `with`-loop.

of array elements in the lower left corner (range `[140,0]` → `[320,200]`) are alternately computed by `op1` and `op2`, respectively (generators 2 and 3 from Fig. 3). On the right hand side of the array (range `[0,200]` → `[320,400]`), alternately, two rows of array elements are computed by `op2` followed by seven rows computed using `op1` (generators 4 and 5).

3 Naïve Compilation

The definition of multi-generator `with`-loops presented in the previous section can be compiled into C code straightforwardly by transforming each generator into a perfect nesting of `for`-loops. This approach in the sequel is referred to as naïve compilation.

```
for (iv_0 = 0; iv_0 < 140; iv_0++) {
    for (iv_1 = 0; iv_1 < 200; iv_1++) {
      res[iv] = op1( iv);
}   }
for (iv_0 = 140; iv_0 < 320; iv_0++) {
    for (iv_1 = 0; iv_1 < 200; iv_1 += 2) {
      res[iv] = op1( iv);
}   }
for (iv_0 = 140; iv_0 < 320; iv_0++) {
    for (iv_1 = 1; iv_1 < 200; iv_1 += 2) {
      res[iv] = op2( iv);
}   }
for (iv_0 = 0; iv_0 < 320; iv_0 += 7) {
  for (stop_0 = iv_0+2; iv_0 < stop_0; iv_0++) {
    for (iv_1 = 200; iv_1 < 400; iv_1++) {
      res[iv] = op2( iv);
} } }
for (iv_0 = 2; iv_0 < 320; iv_0 += 2) {
  for (stop_0 = iv_0+7; iv_0 < stop_0; iv_0++) {
    for (iv_1 = 200; iv_1 < 400; iv_1++) {
      res[iv] = op1( iv);
} } }
```

first generator (G_1)

second generator (G_2)

third generator (G_3)

fourth generator (G_4)

fifth generator (G_5)

Fig. 5. Naïvely compiled code.

Figure 5 shows the C code which results from applying naïve compilation to the multi-generator `with`-loop of the previous section (Fig. 3). It consists of five loop nestings each of which can be generated separately from a single generator as indicated on the right hand side of the figure. For dense generators such as G_1 and strided generators with `width` $= 1$

(G_2 and G_3), the nestings of `for`-loops directly correspond to the boundary vectors and the `step` specifications (cf. Fig. 3). However, for each dimension where `width` > 1 holds, a further `for`-loop is needed to address all elements between 0 and `width` $- 1$. In our example this leads to the creation of a third `for`-loop for G_4 and G_5, respectively.

Unfortunately, naïve compilation has two major disadvantages. First, compiling the generators separately often introduces a considerable amount of loop overhead. One source of loop overhead are adjacent generators that perform identical operations (e.g. G_1 and G_5 in Fig. 5). Another source of such overhead are intertwined generators (e.g. G_2 and G_3). They lead to loop nestings with almost identical boundaries which could be reused. Although elaborate C compilers provide optimizations to that effect, e.g. loop fusion and loop splitting [1,24,25], in most cases they fail to improve naïvely compiled code. The major problem compilers for imperative languages like C or FORTRAN have to deal with, is to actually infer whether the operations within the loop nestings can be re-ordered accordingly [15,17]. The reason for that shortcoming is that in these languages any function call or any reference to an array might cause a side-effect which in turn requires the order of operations to be kept unchanged. Therefore, rather than relying on the C compiler this kind of optimizations has to be done on the SAC level, where due to the functional paradigm referential transparency is guaranteed, and the order of computations can be changed arbitrarily.

The second disadvantage of naïve compilation results from the intracacies of the executing hardware, in particular from the usage of caches. Caches are generally organized by cache lines which hold a fixed number of adjacent bytes (typically 16, 32, or 64) from the main memory. Whenever a value from memory has to be loaded into the processor, an entire cache line will be loaded unless the required data is already present. This property favors memory accesses to adjacent addresses since the values of the subsequent addresses will be available in the cache after the first address was accessed (so-called spatial reuse [14]).

Whenever a non-dense generator (e.g. G_2 from Fig. 5) is compiled naïvely, all values that are in between the addressed elements will be loaded into the cache but they will not be used immediately. Instead, they will be addressed within another loop nesting (e.g. that of G_3) when their values very likely have been flushed out of the cache, which leads to another load from the main memory. To avoid these superfluous cache misses, the evaluation order of the array elements has to be completely re-arranged. To do so, a more sophisticated compilation scheme is required.

4 Canonical Order

In this section a compilation scheme is proposed which tries to optimize the generated C code with respect to loop overhead and potential spatial reuse. Optimal spatial reuse is achieved if all read and write accesses are done in strictly ascending order and the different accesses do not interfere with each other. Although in general the memory access patterns cannot be statically determined, in most real world applications `with`-loops compute array elements by applying some function to elements of (other) arrays at the same index position or at a position with a constant offset to the actual position. This observation straightforwardly leads to the idea of the so-called canonical order. Computing the array elements in canonical order means that the addresses of the resulting array elements are sorted in strictly ascending order irrespective of the form the involved generators have. This guarantees good spatial reuse for the write accesses to the resulting array and in many applications leads to good spatial reuse of the read accesses as well.

Due to the flexibility of multi-generator `with`-loops, compilation into canonical order may require sophisticated nestings of `for`-loops to be generated. Therefore, the compilation

process is divided into several subsequent transformation steps which finally lead to a representation which can be translated into C code easily. Figure 6 demonstrates that process for the example used in the previous sections.

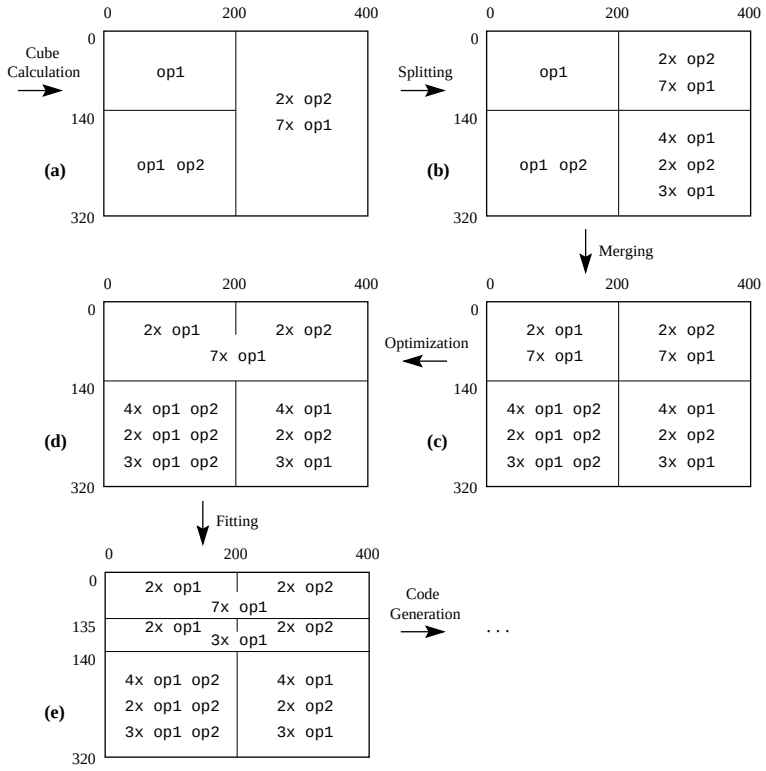

Fig. 6. Compilation into canonical order.

In a first step, intertwined index vector sets are identified and combined into so-called cubes. A cube is a dense rectangular index vector set whose elements are exhaustively described by one or several disjoint generators with identical upper bounds. Although this in general may require generators to be split, in the given example three cubes can be identified without any modification: G_1 constitutes a cube by itself, G_2 and G_3 as well as G_4 and G_5 form cubes with two generators each. This situation is illustrated in Fig. 6(a).

After the cubes have been identified, cubes that are adjacent with respect to inner dimensions are adjusted according to their extent on outer dimensions. The purpose of this so-called splitting operation can be illustrated at the running example in Fig. 6. Along the outermost axis, i.e. on the left side of the array (the current SAC compiler stores arrays in row-major order), the expressions to be computed at index 140 change from `op1` to `(op1 op2)`. In order to ease the final code generation step, the cube on the right hand side is split accordingly (cf. Fig. 6(b)). Since 140 is not a multiple of 9 (the period of the cube on the right hand side) the generators of the freshly created cube consists of three generators which actually result from "shifting" the generators $(140 \bmod 9) = 5$ rows down.

After the cubes have been adjusted to each other, the next step adjusts the generators of adjacent cubes. This so-called merging step requires the periods of generators that are

strided on outer dimensions to be hoisted in periods of the least common multiple (lcm) of all neighboring generators. For the given example this leads to the creation of five new generators for the cubes on the left hand side as depicted in Fig. 6(c).

At this stage of transformation, C code that obeys the canonical order could be generated straightforwardly. However, a closer look to the actual example shows opportunities for further improvements. Whenever two generators with identical operation parts are adjacent with respect to the innermost dimension(s) they can be combined into a single one avoiding superfluous loop overhead. Therefore, the next step (so-called optimization step) tries to combine generators whenever possible. In our example this step combines the second generators of the two upper cubes from Fig. 6(c) into a single one as depicted in Fig. 6(d).

Another source of inefficiency are strided cubes where the size of the cube is not a multiple of the period. For instance, a straightforward compilation of the upper cube in Fig. 6(d) might lead to the following C code:

```
(1)    for (iv_0 = 0; iv_0 < 140; ) {
(2)      stop_0 = MIN( iv_0+2, 140);
(3)      for ( ; iv_0 < stop_0; iv_0++) ...
(4)      stop_0 = MIN( iv_0+7, 140);
(5)      for ( ; iv_0 < stop_0; iv_0++) ...
```

The critical part of this code is the usage of the minimum function (`MIN`) in lines (2) and (4). Because of ($140 \bmod 9 \neq 0$) the last iteration cycle of the outer loop is incomplete. Therefore, the computation of the upper bounds for the indices (`stop_0`) in the loop body must prevent `iv_0` to exceed 140.[1] Unfortunately, this leads to inefficiently executable code because the minimum is calculated in every loop cycle although it is of use only in the last one. This problem can be avoided by an additional transformation called fitting. All generators with incomplete iteration cycles are split into two separate generators, a large one covering all periods but the incomplete last one, and a very small generator for the remaining elements. Since the periods of both generators now comply with their sizes, no further minimum computations are required. For the given example the fitting leads to the new array layout shown in Fig. 6(e).

In a final code generation phase this new array layout can be converted into efficiently executable C code. A formal description of the compilation scheme for `with`-loops and its implementation can be found in [11].

5 Segmentation and Cubes

In general, using the canonical iteration order seems to be a good idea as it minimizes loop overhead and improves the locality of array references. Unfortunately, this technique turns out to have a serious drawback in some rare cases where generators define very inhomogeneous strides. Let us consider a slight variation of the example used so far. The dense generator for the upper left corner of the array is replaced by two strided generators as shown on the left hand side of Fig. 7. These two generators alternately define 15 rows of elements to be computed by `op1` and the next two rows to be computed by `op2`.

While the first two compilation steps of cube calculation and splitting are hardly affected by this modification, the merging phase becomes rather complex. In order to adapt the strides of the upper left cube with the strides of the right cube, new generators with a stride of $9 \times 17 = 153$ have to be generated. As a consequence, for almost each of the topmost 140

[1] Actually only the second occurrence of `MIN` is needed. The first one can be eliminated since 140 is always greater or equal `iv_0+2`.

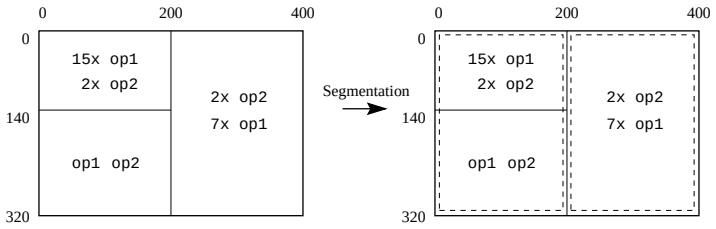

Fig. 7. Segmentation of a multi-generator `with`-loop with inhomogeneous strides.

rows an individual generator has to be created. This code explosion does not only lead to a substantial increase of object code, but also to a negative performance impact due to instruction cache overflows which easily outweigh the benefits achieved by the canonical iteration order.

To avoid such situations, a mechanism is required that penetrates the canonical iteration order whenever this concept is inappropriate due to significantly inhomogeneous strides. The idea is to subdivide the iteration space into a set of pairwise disjoint rectangular subspaces and compute one after the other. This is called a segmentation, the subspaces are called segments. Within each segment, a canonical iteration order is established, i.e., the compilation scheme described in Section 4 actually is applied to each segment. A suitable segmentation for the modified example is shown on the right hand side of Fig. 7. It treats the loss of some spatial locality at the border of the two segments for avoiding a code explosion due to inhomogeneous strides in the upper part of the array.

In principle, any rectangular subarray may form a segment. However, for performance reasons, it is recommended to keep the number of segments as small as possible and use segmentation only to handle inhomogeneous strides. Therefore, the cubes defined in Section 4 constitute the natural basic building blocks for segments. Whenever the stride patterns of two adjacent cubes are too different from each other, the cubes should be placed in different segments. This strategy straightforwardly leads to two specific segmentations: the trivial segmentation where all cubes together form a single segment (default) and the segmentation where each cube represents a segment on its own.

For the time being, little is known about the consequences of different segmentations on runtime performance. To alter this, the SAC compiler is supplied with a versatile interface that allows to experiment with varying segmentations. A special pragma allows programmers to choose any segmentation either on a local scope (one `with`-loop) or on a global scope (all `with`-loops up to next pragma). The syntax of this pragma is

`#pragma wlcomp` *Conf*

where *Conf* denotes either a concrete segmentation or a segmentation strategy; *Conf* may either be `All()` to indicate the trivial segmentation, `Cubes()` to select the cube segmentation, or `ConstSegs(`S`)` where S specifies a list of segments by concrete index ranges.

Once sufficient experience with different segmentation strategies has been made, the pragmas may to some extent be replaced by an inference scheme that implicitly selects a suitable segmentation based on the individual properties of each `with`-loop.

6 Tiling `With`-Loops

The canonical order results in optimal spatial reuse for write accesses to the result array of a `with`-loop as well as for all those read accesses to argument arrays whose array indices differ

from the write index only by a constant offset. Therefore, it provides a reasonable cache performance in many cases. However, the cache performance can often be further improved if the canonical order is not established on the entire iteration space, but on rather small rectangular subspaces which then are computed one after the other again in a canonical order as illustrated in Fig. 8.

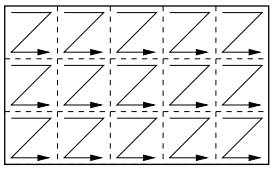

Fig. 8. The principle of tiling.

This subdivision of the iteration space in order to exploit locality and improve the cache performance is called tiling, the subspaces themselves are called tiles. Tiling, often also referred to as blocking, is a well-known optimization technique for scientific numerical codes which has proven to be a critical factor to achieve a good runtime performance over a wide range of applications [12,14,23,7,16].

The problem concerning tiling is that the resulting code even for small problems becomes extremely complicated as the number of loops is doubled. Writing tiled code from scratch is a very time-consuming and error-prone venture. Compiling tiled code from conventional loop specifications ends up with the problem that changing the iteration order requires a compiler to prove the legality of this transformation beforehand. In the context of low-level languages like C or FORTRAN this often precludes tiling because of potential side-effects.

Due to the functional semantics of SAC and in particular the semantics of the `with`-loop, there is no restriction on the iteration order of `with`-loops. This benefit of the functional paradigm is exploited in the SAC compiler in that it allows to generate tiled code. Since today's caches are usually organized in hierarchies with different sizes and technical properties on each level of the hierarchy, the SAC compiler supports hierarchical tiling with up to three levels.

Similar to segmentation, tiling is controlled by means of a pragma

`#pragma wlcomp TvL`n `[`t_0`,...,`t_m`]`

where $n \in \{1, 2, 3\}$ specifies the tiling level and the vector `[`t_0`,...,`t_m`]` defines the desired tile size on that level.

Whenever tiling is enabled, a new transformation step between the splitting phase and the merging phase is added as shown for the running example in Fig. 9, where one-level tiling with a tile size vector of `[100,80]` is applied. Although this tile size is not representative for real problems, it well illustrates the effect of tiling. In particular, it shows that at the edge of the iteration space typically incomplete tiles occur as the size of the iteration space in a certain dimension not necessarily is a multiple of the tile size in this dimension.

7 A Performance Comparison

In this section the optimized compilation scheme for `with`-loops as presented in Section 4 is compared to the naïve one given in Section 3 with respect to runtime efficiency of the generated code. The comparison is based on two examples: The first example is an artificial multi-generator `with`-loop and the second one is a part of a real world application.

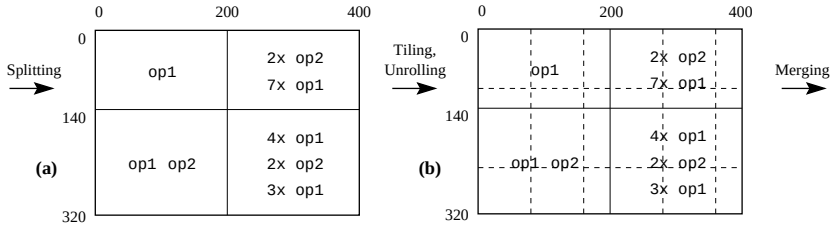

Fig. 9. Tiling with a tile size vector of `[100,80]`.

The hardware platform used for the measurements is a Sun Ultra-2 with 128 MB of main memory running under Solaris 7. The GNU C compiler (gcc, egcs) Version 2.91.66 is used to compile the C code generated by the Sac compiler into native machine code.

The multi-generator `with`-loop shown in Fig. 10 is particularly designed to explore the maximal benefits of the canonical execution order. It consists of `k` generators, each of which defines a grid of complete columns with stride `k` and width 1, i.e., every `k`-th column belongs to the same generator.

```
with ([0,  0] <= iv < [1000,1000] step [1,k]): 1
     ([0,  1] <= iv < [1000,1000] step [1,k]): 2
     ([0,  2] <= iv < [1000,1000] step [1,k]): 3
     ...
     ([0,k-1] <= iv < [1000,1000] step [1,k]): k
```

Fig. 10. `With`-loop with `k` generators forming intertwined columns.

Figure 11 shows the runtimes of the naïvely compiled code (naïve version) as well as of the code obtained by using the optimized compilation scheme (optimized version). For both versions six different numbers of generators are used, these being $k \in \{1, 2, 4, 8, 16, 32\}$. The bars in the diagram depict runtime relative to that of the optimized version with absolute runtimes annotated inside the bars.

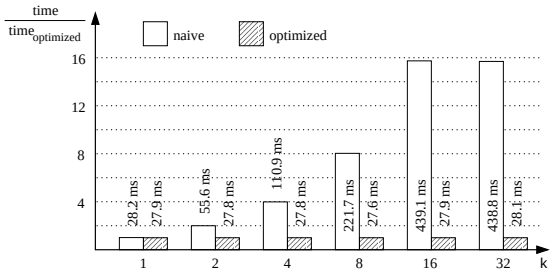

Fig. 11. Time demand for the `with`-loop with `k` intertwined generators.

The performance figures indicate that the absolute time demand of the optimized version is almost independent from the number of generators. Although the complexity of the `with`-loop grows with increasing values of `k`, the runtime remains unchanged. In contrast, the

number of generators has a significant effect on the time demand of the naïve version. In case of `k`$=1$ the array elements are also computed in canonical order. Consequently, the performance is the same as for the optimized version. But for `k`$=2$ a loss of spatial reuse occurs. Both generators address the array elements with a stride of 2. Because the whole array is too large to fit into the cache, the values in between are flushed out of the cache before they are addressed in the next loop nesting. Therefore, the number of memory loads is doubled. Since the `with`-loop contains no computations besides the write accesses this leads to a slowdown of a factor of 2. In analogy the runtime increases proportional to `k` until the number of generators is greater than 16. With `k`≥ 16 no spatial reuse is possible anymore since a single cache line of size 64 byte can hold 16 integer values of 4 bytes each. Thus the worst case for this architecture is reached and no further slowdown due to naïve compilation can be observed.

In the example mentioned so far only a trivial computation is done. In order to demonstrate that the canonical order pays even in real world applications with a higher workload, in the following the mapping from coarse to fine grids as part of a multi-grid algorithm is analyzed. The shape-invariant SAC implementation of this mapping, as shown in Fig. 12, consists of two steps. First the elements from a given coarse grid are copied into every other

```
double[] Coarse2Fine( double[] coarse, double[] weights)
{
  sh = 2 * (shape( coarse) - 1);
  fine = with( 0*sh <= iv < sh step 0*sh+2)
         genarray( sh, coarse[ iv/2]);
  fine = with( 0*sh+1 <= iv < sh-1) {
           val = sum( weights * tile( shape( weights), iv-1, fine));
         } modarray( fine, iv, val);
  return (fine);
```

Fig. 12. SAC implementation of the coarse to fine mapping.

position of a new array of double the size, and the elements in between are initialized with 0. Subsequently, the elements of the new array are re-computed as weighted sum of their neighbor elements. This is done by means of two library functions `tile` and `sum`. The function `tile(sh,iv,A)` returns the subarray of `A` with shape `sh` and upper left index position `iv` whereas `sum(A)` computes the sum of all elements of the given array `A`.

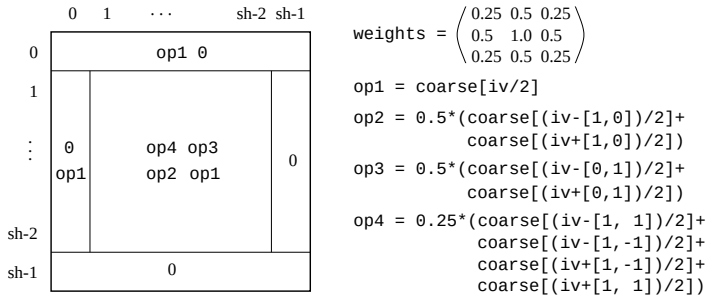

Fig. 13. Layout of the array with the fine grid.

During the compilation process both `with`-loops are folded to a single multi-generator `with`-loop. Its layout in case of two dimensions is depicted in Fig. 13. Neglecting the border elements, this `with`-loop consists of four generators: two different operations are alternately applied in each dimension. Therefore, by establishing the canonical order the number of memory loads due to write accesses is halved in comparison with the naïve version. In analogy to the discussion of the first example this potentially leads to a speedup of a factor of 2, but due to the higher workload per element computation a smaller effect on the overall runtime can be expected. However, the runtime figures presented in Fig. 14 indicate that nevertheless with increasing array sizes the speedup asymptotically approaches 2. This performance gain can be attributed to improved spatial reuse of read accesses as well as to an overall reduction of loop overhead.

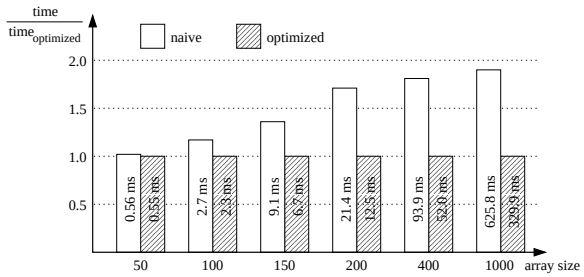

Fig. 14. Time demand for coarse to fine mapping.

8 Related Work

Due to the fact that lists lend themselves very well to the recursive nature of functional programming, comparably few work has been spent on efficient support for arrays. However, some functional languages such as ML and its derivatives Caml and OCaml achieve reasonable array performance by including arrays as impure features. Compilers for these languages typically map these structures directly to their imperative counterparts and rely on standard optimizations of the compiler back-end.

Probably the most notable effort for efficient support of purely functional arrays was done in the Sisal project. In that context, several optimizations for array operations have been developed, most of which are either aimed at the reduction of loop overhead [4] or at the minimization of memory allocation overhead [3,5]. Re-ordering of loop iterations for improving cache utilization, to our knowledge, was not investigated. The reason for this probably can be attributed to the way multi-dimensional arrays are represented in Sisal, i.e., as vectors of vectors. As a consequence of this storage format, only limited benefit with respect to cache behavior can be expected from changing the order in which the array elements are computed.

More recent efforts for efficient array support in functional languages were made in the context of Clean. Due to the lazy regime of Clean, these efforts focus on the update-in-place mechanism for avoiding superfluous memory allocations [22]. Although for several benchmarks runtimes comparable to those of C can be achieved no particular back-end optimizations for improving cache locality in the context of multi-dimensional arrays are installed. Similar to Sisal, the non-flat representation of multi-dimensional arrays probably constitutes a major obstacle for such optimizations.

More closely related work can be found in the context of highly optimizing compilers for imperative languages such as HPF. For these compilers several so-called unimodular loop transformations have been proposed [1,19,6], particularly loop permutation, also called loop interchange, and loop reversal. Loop interchange permutes the loops of a perfect loop nesting; loop reversal reverses the iteration order of a single loop. Both are particularly applied to adjust the iteration order to the array storage format and hence to exploit spatial locality in the same way as through the introduction of the canonical order in SAC. Tiling is a well-known optimization technique for improved utilization of caches in high-performance computing. In terms of FORTRAN loop transformations, it is a combination of loop skewing and loop interchange [23].

The essential difference between these loop transformations and our work is the level of abstraction on which the optimizations are applied. Whereas in the imperative setting much information, e.g. data dependencies between loop incarnations or relations between loop boundaries, has to be gathered from several nested/subsequent loops, the multi-generator `with`-loops of SAC inherently provide this information. Due to the side-effect free setting in SAC, it can be guaranteed without any statical analysis that multi-generator `with`-loops can be computed in any order. As a consequence, loop (re-)organization can be applied more often and it can take into account the structure of the entire sequence of loop nestings needed for a single multi-generator `with`-loop, which in turn allows for better optimization results.

9 Conclusion and Future Work

This paper describes compilation techniques for multi-generator `with`-loops in SAC. One of the basic properties of these `with`-loops is the absence of any specification of a concrete iteration order. Any iteration order is guaranteed to yield the same result. However, the opposite is true with respect to runtime performance. Speedups in execution time of up to a factor of 16 have been measured for the same `with`-loop when using sophisticated compilation schemes as compared to straightforward transformation into C loops. Speedups of up to a factor of 2 are achieved for an essential part of a real world application, i.e. the coarse to fine mapping of a multi-grid relaxation. This is mostly due to the effects of different iteration orders on the utilization of caches. Consequently, it is worthwhile to adjust the iteration order to improve cache performance.

The compilation scheme for multi-generator `with`-loops basically follows two approaches to achieve this. The so-called canonical iteration order is established as far as possible, i.e., array elements are accessed in exactly the same order as they are stored in memory. This exploits spatial locality. A segmentation scheme addresses the potential problem of code explosion in the presence of grids with inhomogeneous strides. Moreover, tiling is introduced in order to reduce the iteration distance between subsequent accesses to the same array element to improve temporal locality.

At the time being, the segmentation scheme as well as the tiling mechanism are controlled through the annotation of pragmas. Although this gives experienced programmers a high degree of freedom in program specification and constitutes a versatile tool for experimentation, it contradicts the ideals of high-level declarative programming. Consequently, future work will focus on implicit control strategies for both segmentation and tiling. Inhomogeneous grids have to be identified leading to a segmentation that prevents code explosion. Favorable tile sizes need to be determined taking into account a specification of the cache parameters, the sizes of result and argument arrays, as well as a thorough analysis of the array access patterns.

References

1. D. F. Bacon, S. L. Graham, O. J. Sharp: *Compiler Transformations for High-Performance Computing*. ACM Computing Surveys, 26(4), pp. 345–420, 1994.
2. C. Burke: *J and APL*. Iverson Software Inc., Toronto, Canada, 1996.
3. D. C. Cann: *Compilation Techniques for High Performance Applicative Computation*. Technical Report CS-89-108, Lawrence Livermore National Laboratory, Livermore, California, USA, 1989.
4. D. C. Cann: *The Optimizing Sisal Compiler (Version 12.0)*. Lawrence Livermore National Laboratory, Livermore, California, USA, 1993. Part of the Sisal distribution.
5. D. C. Cann, P. Evripidou: *Advanced Array Optimizations for High Performance Functional Languages*. IEEE Transactions on Parallel and Distributed Systems, 6(3), pp. 229–239, 1995.
6. M. Cerniak: *Optimizing Programs by Data and Control Transformations*. PhD Thesis, University of Rochester, New York, USA, 1997.
7. S. Coleman, K. McKinley: *Tile Size Selection Using Cache Organization and Data Layout*. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95), La Jolla, California, USA. ACM Press, 1995, pp. 279–290.
8. C. Grelck: *Shared Memory Multiprocessor Support for SAC*. In C. Clack, T. Davie, K. Hammond (Eds.): Implementation of Functional Languages, 10th International Workshop (IFL '98), London, England, UK, Selected Papers. Vol. 1595 of: LNCS. Springer, 1999, pp. 38–53. ISBN 3-540-66229-4.
9. C. Grelck, S.-B. Scholz: *Accelerating APL Programs with SAC*. In O. Lefèvre (Ed.): Proceedings of the Array Processing Language Conference (APL '99), Scranton, Pennsylvania, USA. Vol. 29(2) of: APL Quote Quad. ACM Press, 1999, pp. 50–57.
10. M. A. Jenkins, W. H. Jenkins: *The Q'Nial Language and Reference Manuals*. Nial Systems Ltd., Ottawa, Canada, 1993.
11. D. Kreye: *Zur Generierung von effizient ausführbarem Code aus SAC-spezifischen Schleifenkonstrukten*. Diploma Thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1998.
12. M. S. Lam, E. E. Rothberg, M. E. Wolf: *The Cache Performance and Optimizations of Blocked Algorithms*. In: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, California, USA. 1991, pp. 63–74.
13. E. C. Lewis, C. Lin, L. Snyder: *The Implementation and Evaluation of Fusion and Contraction in Array Languages*. In J. W. Davidson (Ed.): Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98), Montreal, Canada. ACM Press, 1998, pp. 50–59.
14. N. Manjikian, T. S. Abdelrahman: *Array Data Layout for the Reduction of Cache Conflicts*. In: Proceedings of the International Conference on Parallel and Distributed Computing Systems. 1995.
15. D. E. Maydan: *Accurate Analysis of Array References*. PhD Thesis, Stanford University, California, USA, 1992.
16. K. McKinley, S. Carr, C.-W. Tseng: *Improving Data Locality with Loop Transformations*. ACM Transactions on Programming Languages and Systems, 18(4), pp. 424–453, 1996.
17. G. Roth, K. Kennedy: *Dependence Analysis of Fortran-90 Array Syntax*. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. 1996.
18. G. Roth, K. Kennedy: *Loop Fusion in High Performance Fortran*. CRPC TR98745, Rice University, Houston, Texas, 1998.
19. V. Sarkar, R. Thekkath: *A General Framework for Iteration-Reordering Loop Transformations*. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '92), San Francisco, California, USA. ACM Press, 1992, pp. 175–187.
20. S.-B. Scholz: *Single Assignment C — Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. PhD Thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996. ISBN 3-8265-3138-8.

21. S.-B. Scholz: *A Case Study: Effects of WITH-Loop-Folding on the NAS Benchmark MG in SAC.* In C. Clack, T. Davie, K. Hammond (Eds.): Implementation of Functional Languages, 10th International Workshop (IFL '98), London, England, UK, Selected Papers. Vol. 1595 of: LNCS. Springer, 1998, pp. 216–228. ISBN 3-540-66229-4.
22. J. van Groningen: *The Implementation and Efficiency of Arrays in Clean 1.1.* In W. E. Kluge (Ed.): Implementation of Functional Languages, 8th International Workshop (IFL '96), Bad Godesberg, Germany, Selected Papers. Vol. 1268 of: LNCS. Springer, 1997, pp. 105–124. ISBN 3-540-63237-9.
23. M. E. Wolf, M. S. Lam: *A Data Locality Optimizing Algorithm.* In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91), Toronto, Ontario, Canada. ACM Press, 1991, pp. 30–44.
24. M. J. Wolfe: *High-Performance Compilers for Parallel Computing.* Addison-Wesley, 1995. ISBN 0-8053-2730-4.
25. H. P. Zima, B. Chapman: *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.