With-Loop Fusion for Data Locality and Parallelism

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

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

Abstract. WITH-loops are versatile array comprehensions used in the functional array language SAC to implement aggregate array operations that are applicable to arrays of any rank and shape. We describe the fusion of WITH-loops as a novel optimisation technique to improve both the data locality of compiled code in general and the synchronisation behaviour of compiler-parallelised code in particular. Some experiments demonstrate the impact of WITH-loop-fusion on the runtime performance of compiled SAC code.

1 Introduction

SAC (Single Assignment C) [1] is a purely functional array processing language designed with numerical applications in mind. Image processing and computational sciences are two examples of potential application domains. The language design of SAC aims at combining generic functional array programming with a runtime performance that is competitive with low-level, machine-oriented languages both in terms of execution time and memory consumption.

The programming methodology of SAC essentially builds upon two principles: abstraction and composition [1,2,3]. In contrast to other array languages, e.g. APL [4], J [5], NIAL [6], or FORTRAN-90, SAC provides only a very small number of built-in operations on arrays. Basically, there are primitives to query for an array's shape, for its rank, and for individual elements. Aggregate array operations (e.g. subarray selection, element-wise extensions of scalar operations, rotation and shifting, or reductions) are defined in SAC itself. This is done with the help of WITH-loops, versatile multi-dimensional array comprehensions. SAC allows us to encapsulate these operations in abstractions that are universally applicable (i.e., they are applicable to arrays of any rank and shape). More complex array operations are not defined by WITH-loops, but by composition of simpler array operations. Again, they can be encapsulated in functions that may still abstract from concrete ranks and shapes of argument arrays.

Following this technique, entire application programs typically consist of various logical layers of abstraction and composition. This style of programming

A. Butterfield, C. Grelck, and F. Huch (Eds.): IFL 2005, LNCS 4015, pp. 178–195, 2006.

leads to highly generic implementations of algorithms and provides good opportunities for code reuse on each layer of abstraction. As a very simple example, consider a function `MinMaxVal` that yields both the least and the greatest element of an argument array. Rather than implementing this functionality directly using WITH-loops, our programming methodology suggests to define the function `MinMaxVal` by composition of two simpler functions `MinVal` and `MaxVal` that yield the least and the greatest element, respectively.

Direct compilation of programs designed on the principles of abstraction and composition generally leads to poor runtime performance. Excessive creation of temporary arrays as well as repeated traversals of the same array are the main reasons. Separately computing the minimum and the maximum value of an array `A` requires the processor to load each element of `A` into a register twice. If the array is sufficiently small, it may entirely be kept in the L1 cache and the second round of memory loads yields cache hits throughout. However, with growing array size elements are displaced from the cache before temporal reuse is exploited, and data must be re-fetched from off-chip L2 cache or even main memory. In fact, the time it takes to compute minimums and maximums of two values is completely negligible compared with the time it takes to load data from memory. Therefore, we must expect a performance penalty of a factor of two when computing the minimum and the maximum of an array in isolation rather than computing both in a single traversal through memory.

This example illustrates the classical trade-off between modular, reusable code design on the one hand and runtime performance on the other hand. Whereas in many application domains a performance degradation of a factor of 2 or more in exchange for improved development speed, maintainability, and code reuse opportunities may be acceptable, in numerical computing it is not. Hence, in our context abstraction and composition as software engineering principles are only useful to the extent to which corresponding compiler optimisation technology succeeds in avoiding a runtime performance penalty. What is needed is a systematic transformation of programs from a representation amenable to humans for development and maintenance into a representation that is suitable for efficient execution on computing machinery.

In the past, we have developed two complementary optimisation techniques that avoid the creation of temporary arrays at runtime: WITH-loop-folding [7] and WITH-loop-scalarisation [8]. In our current work we address the problem of repeated array traversals, as illustrated by the `MinMaxVal` example. We propose WITH-loop-fusion as a novel technique to avoid costly memory traversals at runtime. To make fusion of WITH-loops feasible, we extend the internal representation of WITH-loops in order to accomodate the computation of multiple values by a single WITH-loop, which we call *multi-operator* WITH-loop. We introduce WITH-loop-fusion as a high-level code transformation on intermediate SAC code. While the essence of fusion is formally defined in a very restricted setting, we introduce additional pre- and postprocessing techniques that broaden the applicability of fusion and improve the quality of fused code.

The remainder of this paper is organised as follows. Section 2 provides a brief introduction into WITH-loops. In Section 3 we extend the internal representation of WITH-loops to multi-operator WITH-loops. The base case for WITH-loop-fusion is described in Section 4. More complex cases are reduced to the base case using techniques described in Section 5 and post-fusion optimisations in Section 6. Section 7 illustrates the combined effect of the various measures on a small case study while Section 8 reports on a series of experiments. In Section 10 we draw conclusions and outline directions of future research.

2 With-Loops in SAC

As the name suggests, SAC is functional subset of C, extended by multi-dimensional arrays as first class citizens. We have adopted as much of the syntax of C as possible to ease adaptation for programmers with a background in imperative programming, the prevailing paradigm in our targeted application areas. Despite its C-like appearance, the semantics of SAC code is defined by context-free substitution of expressions. "Imperative" language features like assignment chains, branches, or loops are semantically explained and internally represented as nested `let`-expressions, conditional expressions, and tail-end recursive functions, respectively. Nevertheless, whenever SAC code is syntactically identical to C code, the functional semantics of SAC and the imperative semantics of C also coincide. Therefore, the programmer may keep his preferred model of thinking, while the SAC compiler may exploit the functional semantics for advanced optimisations. Space limitations prevent us from further elaborating on the design of SAC, but a rule of thumb is that everything that looks like C also behaves as in C. More detailed introductions to SAC and its programming methodology may be found in [1,2,3].

In contrast to other array languages SAC provides only a very small set of built-in operations on arrays. Basically, they are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`). Aggregate array operations are specified in SAC itself using powerful array comprehensions, called WITH-loops. Their syntax is defined in Fig. 1.

WithExpr	⇒	**with** *Generator* **:** *Expr Operation*
Generator	⇒	(*Expr* <= *Identifier* < *Expr* ⌈*Filter*⌋)
Filter	⇒	**step** *Expr* ⌈ **width** *Expr* ⌋
Operation	⇒	**genarray** (*Expr* ⌈ , *Expr* ⌋)
	|	**fold** (*FoldOp* , *Expr*)

Fig. 1. Syntax of with-loop expressions

A WITH-loop is a complex expression that consists of three parts: a *generator*, an *associated expression* and an *operation*. The operation determines the overall

meaning of the WITH-loop. There are two variants: `genarray` and `fold`. With `genarray( shp, default)` the WITH-loop creates a new array of shape *shp*. With `fold( foldop, neutral)` the WITH-loop specifies a reduction operation with *foldop* being the name of an appropriate associative and commutative binary operation with neutral element *neutral*.

The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions, which must evaluate to integer vectors of equal length, define lower and upper bounds of a rectangular index vector range. For each element of this set of index vectors the associated expression is evaluated. Depending on the variant of WITH-loop, the resulting value is either used to initialise the corresponding element position of the array to be created (`genarray`) or it is given as an argument to the fold operation (`fold`). In the case of a `genarray`-WITH-loop, elements of the result array that are not covered by the generator are initialised by the (optional) default expression in the operation part. For example, the WITH-loop

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

yields the matrix $\begin{pmatrix} 0&0&0&0&0\\ 0&2&3&4&0\\ 0&3&4&5&0 \end{pmatrix}$. The generator in this example WITH-loop defines the set of 2-element vectors in the range between `[1,1]` and `[3,4]`. The index variable `iv` represents elements from this set (i.e. 2-element vectors) in the associated expression `iv[0] + iv[1]`. Therefore, we compute each element of the result array as the sum of the two components of the index vector, whereas the remaining elements are initialised with the value of the default expression. The WITH-loop

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

sums up all non-zero elements of the above matrix and evaluates to 21. An optional filter may be used to further restrict generators to periodic grid-like patterns, e.g.,

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

yields the matrix $\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0\\ 0&1&1&0&1&1&0&1&0&0\\ 0&1&1&0&1&1&0&1&0&0 \end{pmatrix}$.

3 Multi-operator With-Loops

The aim of WITH-loop-fusion is to avoid the repeated traversal of argument arrays by computing multiple values in a single sweep. Hence, a major prerequisite for fusion is the ability to represent the computation of multiple values by a single WITH-loop. Regular WITH-loops, as described in the previous section, define either a single array or a single reduction value. To overcome this limitation we extend the internal representation of WITH-loops to *multi-operator* WITH-loops, as illustrated in Fig. 2.

MultiOpWith ⇒ **with** ⌈ *Generator* : *Expr* [, *Expr*]*]+⌈ *Operation*]+
Generator ⇒ (*Expr* <= *Identifier* < *Expr* [*Filter*])
Filter ⇒ **step** *Expr* [**width** *Expr*]
Operation ⇒ **genarray** (*Expr*)
| **fold** (*FoldOp* , *Expr*)

Fig. 2. Pseudo syntax of multi-operator with-loop expressions

Internal multi-operator WITH-loops differ from language-level WITH-loops in various aspects:

- They have a non-empty sequence of operations rather than exactly one.
- They have a non-empty sequence of generators rather than exactly one.
- Each generator is associated with a non-empty, comma-separated list of expressions rather than a single one.
- There is no default expression in `genarray` operations.

In the internal representation of WITH-loops, the default case is made explicit by creating a full partition of the index space. If necessary, additional generators are introduced that cover those indices not addressed by the original generator. These generators are explicitly associated with the default expression. A side condition not expressed in Fig. 2 is that all generators must be associated with the same number of expressions, and this number must match the number of operations. More precisely, the first operation corresponds to the first expression associated with each generator, the second operation corresponds to each second expression, etc. For example, the function `MinMaxVal` from the introduction can be specified by the following multi-operator WITH-loop for argument arrays of any rank and shape:

```
int, int MinMaxVal( int[*] A)
{
  Min, Max = with (0*shape(A) <= iv < shape(A)) : A[iv], A[iv]
             fold( min, MaxInt())
             fold( max, MinInt());
  return( Min, Max);
}
```

The multi-operator WITH-loop yields two values, which are bound to two variables using simultaneous assignment. While this simple example only uses `fold` operations, `fold` and `genarray` operations are generally mixed. We do not feature multi-operator WITH-loops on the language level because they run counter the idea of modular generic specifications. We consider the above representation of `MaxMinVal` the desired outcome of an optimisation process, not a desirable implementation.

4 With-Loop-Fusion — The Base Case

In the following we describe WITH-loop-fusion as a high-level code transformation. The base case for optimisation is characterised by two WITH-loops that

have the same sequence of generators and no data dependence (i.e., none of the variables bound to individual result values of the first WITH-loop is referred to within the second WITH-loop). A formalisation of WITH-loop-fusion for this base case is shown in Fig. 3. We define a transformation scheme

$$\mathcal{WLFS}[\![pattern]\!] \;=\; expr \mid guard$$

that denotes the context-free replacement of an intermediate SAC program fragment pattern *pattern* by the instantiated SAC expression *expr* provided that the guard expression *guard* evaluates to `true`.

WITH-loop-fusion systematically examines intermediate SAC code to identify pairs of suitable WITH-loops. The guard condition for applying $\mathcal{WLFS}$ is twofold. Firstly, all operation parts of type `genarray` must refer to the same shape. Secondly, the two WITH-loops under consideration must be free of data dependences. For the formalisation of this property we employ a function $\mathcal{FV}$ that yields the set of free variables of a given SAC expression. The third prerequisite (i.e. the equality of the generator sequences) is expressed by using the same identifiers in the pattern part of the transformation scheme. Here, we ignore the fact that generators actually form a set rather than a sequence in order to simplify our presentation. In the implementation we resolve the issue by keeping generators sorted in a systematic way.

Since WITH-loop-fusion can be applied repeatedly, we define the transformation scheme $\mathcal{WLFS}$ on multi-operator WITH-loops. Hence, $Ids^{(a)}$ matches a non-empty, comma-separated list of identifiers rather than a single identifier. No special treatment of language-level WITH-loops is required. If all conditions are met, $\mathcal{WLFS}$ takes two assignments with WITH-loops on their right hand sides and concatenates

1. the sequences of assigned identifiers,
2. the sequences of expressions associated with each generator, and
3. the sequences of operations.

Intermediate SAC code is represented in a variant of static single assignment form [9]. Therefore, index variables used in the two WITH-loops to be fused have different names. In the transformation scheme $\mathcal{WLFS}$ we address this issue by keeping the index variable of the first WITH-loop. All associated expressions that originally stem from the second WITH-loop are systematically α-converted to use the index variable of the first WITH-loop, too. In Fig. 3, this is denoted by $[expr]^{iv^{(a)}}_{iv^{(b)}}$ meaning that all free occurrences of $iv^{(b)}$ in *expr* are replaced by $iv^{(a)}$.

The transformation scheme $\mathcal{WLFS}$ as presented in Fig. 3 is meaning-preserving as it preserves the one-to-one correspondence between associated expressions, WITH-loop operations, and bound variables. In the absence of data dependences, the associated expressions and operations of the second WITH-loop may safely be moved out of the scope of the identifiers bound by the first WITH-loop without penetrating the static binding structure.

$$
\mathcal{WLFS}\left[\!\!\left[
\begin{array}{l}
Ids^{(a)} \texttt{ = with} \\
\quad \texttt{(} lb_1 \texttt{ <= } iv^{(a)} \texttt{ < } ub_1 \texttt{ step } s_1 \texttt{ width } w_1 \texttt{) :} \\
\quad\quad expr^{(a)}_{1,1}, \ldots, expr^{(a)}_{1,m} \\
\quad\quad \ldots \\
\quad \texttt{(} lb_k \texttt{ <= } iv^{(a)} \texttt{ < } ub_k \texttt{ step } s_k \texttt{ width } w_k \texttt{) :} \\
\quad\quad expr^{(a)}_{k,1}, \ldots, expr^{(a)}_{k,m} \\
\quad operation^{(a)}_1 \;\; \ldots \;\; operation^{(a)}_m \texttt{ ;} \\
Ids^{(b)} \texttt{ = with} \\
\quad \texttt{(} lb_1 \texttt{ <= } iv^{(b)} \texttt{ < } ub_1 \texttt{ step } s_1 \texttt{ width } w_1 \texttt{) :} \\
\quad\quad expr^{(b)}_{1,1}, \ldots, expr^{(b)}_{1,n} \\
\quad\quad \ldots \\
\quad \texttt{(} lb_k \texttt{ <= } iv^{(b)} \texttt{ < } ub_k \texttt{ step } s_k \texttt{ width } w_k \texttt{) :} \\
\quad\quad expr^{(b)}_{k,1}, \ldots, expr^{(b)}_{k,n} \\
\quad operation^{(b)}_1 \;\; \ldots \;\; operation^{(b)}_n \texttt{ ;}
\end{array}
\right]\!\!\right]
$$

$$
= \left\{
\begin{array}{l}
Ids^{(a)},\; Ids^{(b)} \texttt{ = with} \\
\quad \texttt{(} lb_1 \texttt{ <= } iv^{(a)} \texttt{ < } ub_1 \texttt{ step } s_1 \texttt{ width } w_1 \texttt{) :} \\
\quad\quad expr^{(a)}_{1,1}, \ldots, expr^{(a)}_{1,m}, \\
\quad\quad [expr^{(b)}_{1,1}]^{iv^{(a)}}_{iv^{(b)}}, \ldots, [expr^{(b)}_{1,n}]^{iv^{(a)}}_{iv^{(b)}} \\
\quad\quad \ldots \\
\quad \texttt{(} lb_k \texttt{ <= } iv^{(a)} \texttt{ < } ub_k \texttt{ step } s_k \texttt{ width } w_k \texttt{) :} \\
\quad\quad expr^{(a)}_{k,1}, \ldots, expr^{(a)}_{k,m}, \\
\quad\quad [expr^{(b)}_{k,1}]^{iv^{(a)}}_{iv^{(b)}}, \ldots, [expr^{(b)}_{k,n}]^{iv^{(a)}}_{iv^{(b)}} \\
\quad operation^{(a)}_1 \;\; \ldots \;\; operation^{(a)}_m \\
\quad operation^{(b)}_1 \;\; \ldots \;\; operation^{(b)}_n
\end{array}
\right.
$$

$$
\left|
\begin{array}{l}
\forall i \in \{1, \ldots, m\} \;:\; \forall j \in \{1, \ldots, n\} \;: \\
\quad operation^{(a)}_i \equiv \texttt{genarray(}shape^{(a)}_i\texttt{)} \\
\quad \wedge\; operation^{(b)}_j \equiv \texttt{genarray(}shape^{(b)}_j\texttt{)} \\
\quad\quad \Longrightarrow \quad shape^{(a)}_i = shape^{(b)}_j \\
\\
Ids^{(a)} \;\cap\; \bigcup_{i=1}^{k} \bigcup_{j=1}^{n} \mathcal{FV}(expr^{(b)}_{i,j}) \;=\; \emptyset
\end{array}
\right.
$$

Fig. 3. Basic WITH-loop-fusion scheme

5 Enabling With-Loop Fusion

The transformation scheme $\mathcal{WLFS}$, as outlined in the previous section, is only applicable in a very restricted setting. In particular, adjacency in intermediate code and the need for identical generator sets are difficult to meet in practice. Instead of extending our existing transformation scheme to cover a wider range of settings, we accompany $\mathcal{WLFS}$ by a set of preprocessing code transformations that create application scenarios.

Intermediate code between two WITH-loops under consideration for fusion must be moved ahead of the first WITH-loop if it does not reference any of the variables bound by it. The remaining code must be moved below the second WITH-loop if it does not bind variables referenced within the second WITH-loop. Any remaining code constitutes an indirect data dependence between the two WITH-loops and prevents their fusion. The referential transparency of a single assignment language like SAC substantially facilitates code reorganisation and is one prerequisite to make WITH-loop-fusion effective in practice.

$$\mathcal{IS}\left[\!\!\left[\begin{array}{l}\texttt{with}\\ \quad gen_1^{(a)}\ :\ exprs_1^{(a)}\\ \quad\quad\cdots\\ \quad gen_k^{(a)}\ :\ exprs_k^{(a)}\\ operator_1^{(a)}\\ \quad\cdots\\ operator_p^{(a)}\end{array}\right]\!\!\right]\left[\!\!\left[\begin{array}{l}\texttt{with}\\ \quad gen_1^{(b)}\ :\ exprs_1^{(b)}\\ \quad\quad\cdots\\ \quad gen_l^{(b)}\ :\ exprs_l^{(b)}\\ operator_1^{(b)}\\ \quad\cdots\\ operator_q^{(b)}\end{array}\right]\!\!\right]$$

$$= \left\{\begin{array}{l}\texttt{with}\\ \quad \mathcal{GEN}\left[\!\!\left[gen_1^{(a)}\ :\ exprs_1^{(a)}\right]\!\!\right]\left[\!\!\left[gen_1^{(b)}\ \cdots\ gen_l^{(b)}\right]\!\!\right]\\ \quad\quad\cdots\\ \quad \mathcal{GEN}\left[\!\!\left[gen_k^{(a)}\ :\ exprs_k^{(a)}\right]\!\!\right]\left[\!\!\left[gen_1^{(b)}\ \cdots\ gen_l^{(b)}\right]\!\!\right]\\ operator_1^{(a)}\\ \quad\cdots\\ operator_p^{(a)}\end{array}\right.$$

$$\mathcal{GEN}\left[\!\!\left[gen^{(a)} : exprs\right]\!\!\right]\left[\!\!\left[gen_1^{(b)}\ \cdots\ gen_l^{(b)}\right]\!\!\right]$$

$$= \left\{\begin{array}{l}\mathcal{CUT}\left[\!\!\left[gen^{(a)} : exprs\right]\!\!\right]\left[\!\!\left[gen_1^{(b)}\right]\!\!\right]\\ \quad\cdots\\ \mathcal{CUT}\left[\!\!\left[gen^{(a)} : exprs\right]\!\!\right]\left[\!\!\left[gen_l^{(b)}\right]\!\!\right]\end{array}\right.$$

$$\mathcal{CUT}\left[\!\!\left[(lb^{(a)}\ \texttt{<=}\ iv^{(a)}\ \texttt{<}\ ub^{(a)})\ :\ exprs\right]\!\!\right]\left[\!\!\left[(lb^{(b)}\ \texttt{<=}\ iv^{(b)}\ \texttt{<}\ ub^{(b)})\right]\!\!\right]$$

$$= \mathcal{ELIM}\left[\!\!\left[\texttt{(max(}lb^{(a)}\texttt{,}lb^{(b)}\texttt{)}\ \texttt{<=}\ iv^{(a)}\ \texttt{<}\ \texttt{min(}lb^{(a)}\texttt{,}lb^{(b)}\texttt{))}\ :\ exprs\right]\!\!\right]$$

$$\mathcal{ELIM}[\![\texttt{([}lb_1,\ldots,lb_n\texttt{]}\ \texttt{<=}\ iv\ \texttt{<}\ \texttt{[}ub_1,\ldots,ub_n\texttt{])}\ :\ exprs]\!] = \left\{\begin{array}{ll}./. & \mid\ \exists\, i \in \{1,\ldots n\} : lb_i \geq ub_i\\ \texttt{([}lb_1,\ldots,lb_n\texttt{]}\ \texttt{<=}\ iv\ \texttt{<}\ \texttt{[}ub_1,\ldots,ub_n\texttt{])}\ :\ exprs & \mid\ otherwise\end{array}\right.$$

Fig. 4. Intersection of generators

We unify generator sets of two WITH-loops by systematically computing intersections of each pair of generators from the first and from the second WITH-loop. This code transformation is formalised by the compilation scheme $\mathcal{IS}$, defined in

Fig. 4. $\mathcal{IS}$ takes two arguments: firstly, the WITH-loop whose generator set is to be refined and, secondly, the WITH-loop which is under consideration for later fusion. Each generator of the first WITH-loop is associated with the entire sequence of generators of the second WITH-loop. The auxiliary scheme $\mathcal{GEN}$ effectively maps the generator/expression pair that originates from the first WITH-loop to each generator originating from the second WITH-loop. Finally, the auxiliary scheme $\mathcal{CUT}$ defines the intersection between two individual generators. For the sake of clarity we restrict our presentation to generators without step and width specifications and refer to [10] for more details.

The resulting number of generators equals the product of the numbers of generators of the individual WITH-loops. However, in practice many of the potential generators refer to empty index sets. Therefore, we add another auxiliary scheme $\mathcal{ELIM}$ that identifies and eliminates these generators. In addition, we use a compile time threshold on the number of generators in fused WITH-loops to prevent accidental code explosion in rare cases. We illustrate the unification of generator sets in Fig. 5. We start with two language-level WITH-loops and as a first step introduce additional generators that make each WITH-loop's default rule explicit. In a second step, we unify the two generator sets by computing all pairwise intersections between generators, and, eventually, we apply WITH-loop-fusion itself.

Another common obstacle to WITH-loop-fusion are data dependences between WITH-loops. If the sets of generators are sufficiently simple or similar to make fusion feasible, it is often beneficial to eliminate the data dependence by a forward substitution of associated expressions of the first WITH-loop into the second WITH-loop. More precisely, we analyse the second WITH-loop and replace every reference to an element of an array defined by the first WITH-loop with the corresponding defining expression.

Technically, the forward substitution of expressions from one WITH-loop into another resembles WITH-loop-folding. However, it is of little help here as WITH-loop-folding only performs the forward substitution of an associated expression if the original WITH-loop eventually becomes obsolete in order to avoid duplication of work. Exactly this prerequisite is not met in a fusion scenario because the values defined by both WITH-loops under consideration are necessarily needed in subsequent computations. However, if we are sure to apply WITH-loop-fusion as well and if the second WITH-loop solely references elements of the first WITH-loop at the position of the index variable, we can guarantee that the duplication of work introduced by forward substitution will be undone by subsequent transformations. This is demonstrated by means of a more realistic example, which we discuss in Section 7.

6 Post-fusion Optimisations

After successful fusion of WITH-loops, generators are associated with multiple expressions. The expressions themselves, however, are left unmodified. Taking the definition of the function `MinMaxVal` introduced in Section 3 as an example,

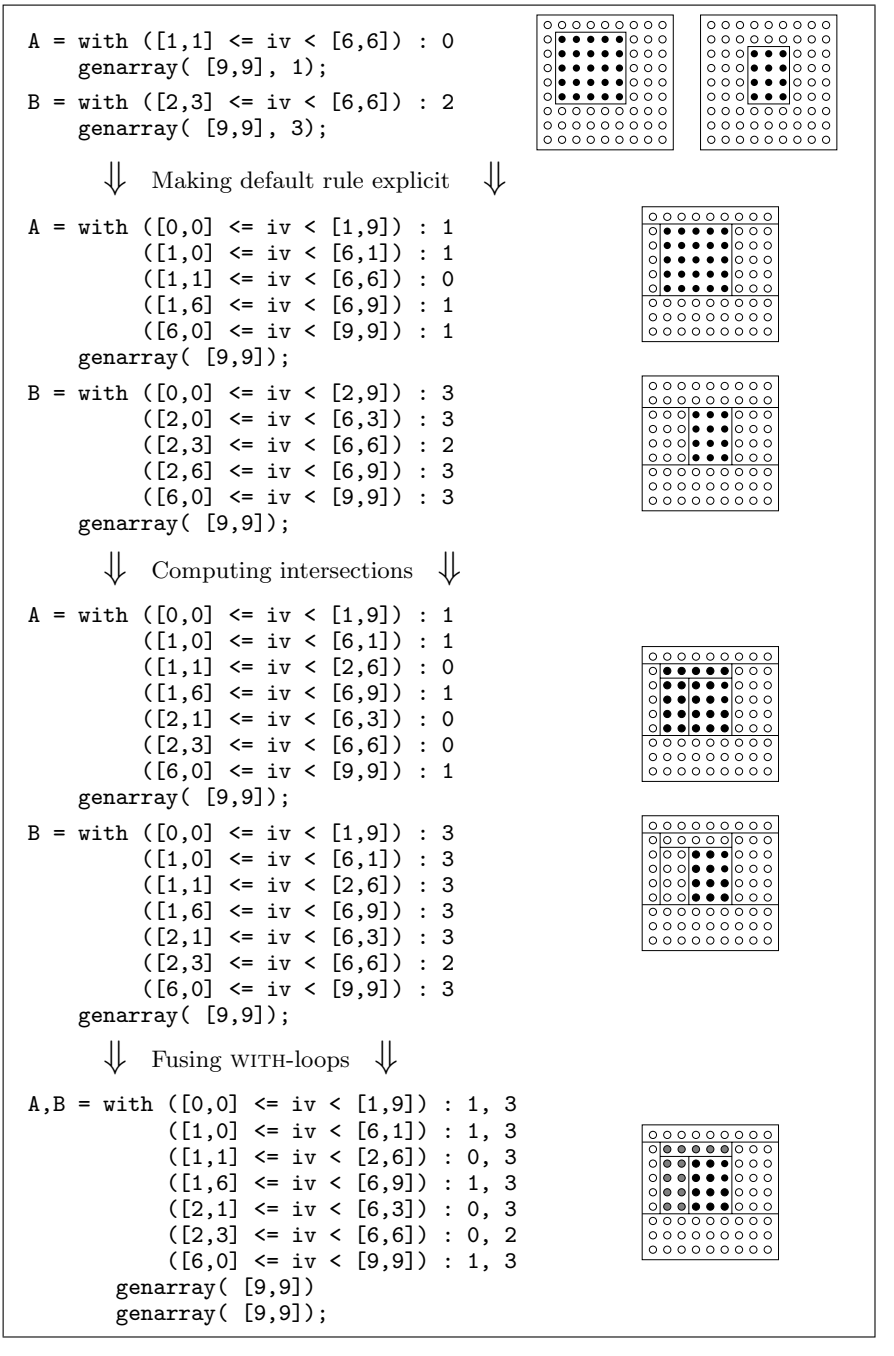

Fig. 5. Example illustrating the systematic intersection of generators

fusion has changed the order in which elements of the argument array `A` are accessed, but the number of accesses is still the same. This change in the order of memory accesses improves temporal locality. In fact, every second access is guaranteed to be an L1 cache hit. Nevertheless, it would be even more desirable to avoid the second memory access at all and to directly take the value from the destination register of the first memory load.

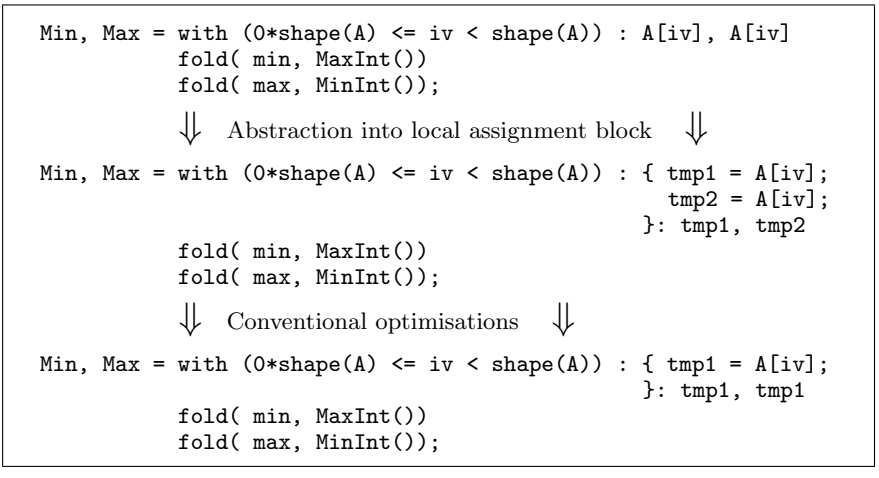

Fig. 6. Illustration of post-fusion optimisation

Unfortunately, our current representation, which associates a sequence of unrelated expressions with each generator, effectively hinders our standard optimisations to further improve the code. Therefore, we introduce a block of local variable bindings between each generator and its associated expressions. At the same time, we restrict these expressions to be identifiers bound in that block. Fig. 6 illustrates this transformation by means of the `MinMaxVal` example. Here, we assume `tmp1` and `tmp2` to be fresh, previously unused identifiers. This rather simple postprocessing step allows us to apply the full range of optimisation techniques available in SaC. In the example common subexpression elimination and variable propagation succeed in reducing the effective number of memory references by one half.

7 Case Study

We illustrate the various code transformation steps involved in WITH-loop-fusion by means of a small case study. Fig. 7 shows a dimension-invariant SaC implementation of a simple convolution algorithm with periodic boundary conditions and convergence test. Within the function `convolution` we iteratively compute a single relaxation step (`relax`) and evaluate a convergence criterion (`continue`). Relaxation with periodic boundary conditions is realised by rotating the argument array one element clockwise and one element counterclockwise in each

dimension. The convergence criterion `continue` yields `true` iff there is an index position for which the absolute difference of the corresponding values in argument arrays `A` and `B` exceeds the given threshold `eps`. All array operations are imported from the SaC standard array library and are themselves implemented in SaC by means of WITH-loops.

```
double[+] relax (double[+] A)
{
  for (i=0; i<dim(A); i+=1) {
    R = R + rotate( i, 1, A) + rotate( i, -1, A);
  }
  return( R / (2 * dim(A) + 1));
}

bool continue (double[+] A, double[+] B, double eps)
{
  return( any( abs( A - B) > eps));
}

double[+] convolution (double[+] A, double eps)
{
  do {
    B = A;
    A = relax( B);
  }
  while (continue( A, B, eps));
  return( A);
}
```

Fig. 7. Dimension-invariant specification of convolution

Specialisation of the dimension-invariant code to a concrete shape of argument arrays and preceding optimisations, mostly function inlining and WITH-loop folding, lead to the intermediate SaC code shown on top of Fig. 8. In each iteration of the convolution algorithm we essentially compute the relaxation step by a single `genarray`-WITH-loop and the convergence test by a single `fold`-WITH-loop. For illustrative purposes we assume a specialisation to 9-element vectors.

Fig. 8 illustrates the various steps required for even this simple example to achieve successful fusion of the two WITH-loops. The first step is the unification of the two generator sequences. The single generator of the `fold`-WITH-loop is split into three parts and the associated expression is duplicated accordingly, as described in Section 5. Unfortunately, the result of the relaxation step is required for evaluating the convergence test. This data dependence still prevents WITH-loop fusion. We eliminate it by replacing the reference to array `A` in the `fold`-WITH-loop by the corresponding expression that defines the value of this element of `A` in the `genarray`-WITH-loop. As the corresponding generators from both WITH-loops are treated in the same way from here on, we show only one.

```
A = with ([0]<=iv<[1]): (B[iv+8]+B[iv]+B[iv+1])/3
         ([1]<=iv<[8]): (B[iv-1]+B[iv]+B[iv+1])/3
         ([8]<=iv<[9]): (B[iv-1]+B[iv]+B[iv-8])/3
    genarray( [9]);
c = with ([0]<=jv<[9]): abs(A[jv]-B[jv])>=eps
    fold( ||, false);
```

⇓ Generator unification ⇓

```
A = with ([0]<=iv<[1]): (B[iv+8]+B[iv]+B[iv+1])/3
         ([1]<=iv<[8]): (B[iv-1]+B[iv]+B[iv+1])/3
         ([8]<=iv<[9]): (B[iv-1]+B[iv]+B[iv-8])/3
    genarray( [9]);
c = with ([0]<=jv<[1]): abs(A[jv]-B[jv])>=eps
         ([1]<=jv<[8]): abs(A[jv]-B[jv])>=eps
         ([8]<=jv<[9]): abs(A[jv]-B[jv])>=eps
    fold( ||, false);
```

⇓ Data dependence elimination ⇓

```
A = with ...
         ([1]<=iv<[8]):(B[iv-1]+B[iv]+B[iv+1])/3
    genarray( [9]);
c = with ...
         ([1]<=jv<[8]): abs(((B[jv-1]+B[jv]+B[jv+1])/3)-B[jv])>=eps
    fold( ||, false);
```

⇓ With-loop fusion ⇓

```
A,c = with ...
           ([1]<=iv<[8]): (B[iv-1]+B[iv]+B[iv+1])/3,
                          abs(((B[iv-1]+B[iv]+B[iv+1])/3)-B[iv])>=eps
      genarray( [9])
      fold( ||, false);
```

⇓ Abstraction into local assignment block ⇓

```
A,c = with ...
           ([1]<=iv<[8]):
                { tmp1 = (B[iv-1]+B[iv]+B[iv+1])/3;
                  tmp2 = abs(((B[iv-1]+B[iv]+B[iv+1])/3)-B[iv])>=eps
                }: tmp1, tmp2
      genarray( [9])
      fold( ||, false);
```

⇓ Conventional optimisations ⇓

```
A,c = with ...
           ([1]<=iv<[8]):
                { tmp0 = B[iv];
                  tmp1 = (B[iv-1]+tmp0+B[iv+1])/3;
                  tmp2 = abs(tmp1-tmp0)>=eps;
                }: tmp1, tmp2
      genarray( [9])
      fold( ||, false);
```

Fig. 8. Illustration of fusion steps for convolution example

We now apply WITH-loop fusion as defined in Section 4. Abstraction of subexpressions into a joint block of local variable bindings, as described in Section 6, follows next. This opens up a plethora of further optimisation opportunities. Most notable, common subexpression elimination avoids the repeated computation of the relaxation step introduced when eliminating the data dependence between the two initial WITH-loops. The overall outcome of this sequence of code transformations is an intermediate code representation that computes both the relaxation step and the convergence test in a single sweep.

8 Experimental Evaluation

We have conducted several experiments in order to quantify the impact of WITH-loop-fusion on the runtime performance of compiled SAC code. Our test system is a 1100MHz Pentium III based PC running SuSE LINUX, and we used gcc 3.3.1 as backend compiler to generate native code.

The first experiment involves our initial motivating example: computing minimum and maximum values of an array. Fig. 9a shows runtimes for three different problem sizes with and without application of WITH-loop-fusion. As expected, there is almost no improvement for very small arrays. The benefits of fusion in this example are two-fold. We do save some loop overhead, but our experiments show this to be marginal. Therefore, the main advantage of fusion in this example is that we can avoid one out of two memory accesses. However, as long as an argument array easily fits into the L1 cache of the processor, the penalty turns out to be negligible. As Fig. 9a shows, this situation changes in steps as the array size exceeds L1 and later L2 cache capacities. In the latter case, WITH-loop-fusion reduces program execution time by almost 50%.

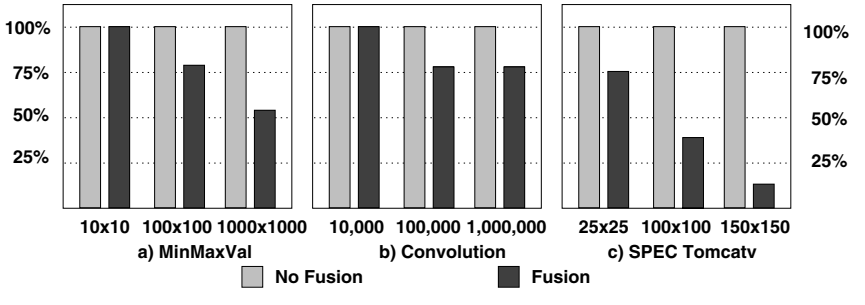

Fig. 9. Impact of WITH-loop-fusion on program execution times for computing minimum and maximum element values (left), convolution with periodic boundaries and convergence test (centre), and the SPEC benchmark `tomcatv` (right) for varying problem sizes

Our second benchmark is the convolution algorithm used as a case study in Section 7. Fig. 9b shows our measurements. For a small problem size fusion again

has no visible impact on performance, but with growing problem size a nearly 25% reduction can be observed. While this is truly a substantial performance gain, we had anticipated more. For the given example, WITH-loop-fusion should reduce the number of memory load operations in the inner loop from 5 (3 in the relaxation kernel and 2 in the convergence criterion) to only 3. Keeping in mind that this numerical kernel is fully memory bound, one would expect a speedup of 40% rather than only 25%. However, a closer look at the generated assembly code revealed that both short and simple loop kernels in the non-fused case exclusively operate on registers, whereas the larger and more complex loop kernel derived from the fused code partially operates on the stack. This explains the sub-optimal performance gain observed.

The last experiment is based on a SAC implementation of the SPEC benchmark `tomcatv`. As shown in Fig. 9c, substantial performance gains can be observed for this benchmark with growing problem size. Improvements of up to 80% must be attributed to the fact that unlike in the previous examples more than two WITH-loops are fused at a time.

Having demonstrated the significant performance impact of WITH-loop-fusion, it would be similarly interesting to see how many application cases exist across a representative suite of programs. However, the answer critically depends on programming style. In fact, we consider WITH-loop-fusion, as WITH-loop-folding and WITH-loop-scalarisation, an enabling technology to make our propagated programming methodology based on the principles of abstraction and composition feasible in practice. Rather than generally improving the runtime behaviour of existing programs, WITH-loop-fusion eliminates the performance penalty of compositional specifications and, thus, enables us to write code that is easier to maintain and to reuse without sacrificing performance.

9 Related Work

WITH-loop-fusion is the third and last missing optimisation technique to systematically transform generic SAC programs into efficiently executable code. It is orthogonal to our previously proposed optimisations, WITH-loop-folding [7] and WITH-loop-scalarisation [8], in the sense that each of the three optimisations addresses a specific type of composition. WITH-loop-folding resolves vertical compositions of WITH-loops, where the result of one array operation becomes the argument of subsequent array operations (i.e., program organisation follows a producer-consumer pattern). WITH-loop-scalarisation addresses nested compositions of WITH-loops, where for each element of the set of indices of an outer WITH-loop a complete inner WITH-loop is evaluated (i.e., in each iteration of the outer WITH-loop a temporary array is created). Thus, both WITH-loop-folding and WITH-loop-scalarisation aim at avoiding the actual creation of temporary arrays at runtime. In contrast, WITH-loop-fusion addresses WITH-loops that are unrelated in the data flow graph or that can be made so by preprocessing techniques. In this case, fusion of WITH-loops does not change the number of data structures created at runtime, but it reduces some loop overhead and — most

important — it changes the order of references into existing arrays in a way that improves data locality in memory hierarchies.

The wish to avoid the repeated traversal of large data structures is neither specific to generic array programming in general nor to SaC in particular. In the context of algebraic data types tupling [11] has been proposed to avoid repeated traversals of list- and tree-like data structures. Rather than gathering two values from the same data structure one after the other, tupling aims at gathering the tuple of values in a single traversal, hence the name. Whereas, the underlying idea essentially is the same in tupling and WITH-loop-fusion the different settings make their concrete appearances fairly different.

Fusion techniques have a long tradition in research on implementation of functional languages [12,13,14,15]. The growing popularity of generic programming techniques [16,17] has created additional optimisation demand [18]. All these approaches follow the mainstream of functional languages in that they focus on lists or on algebraic data types. Much less work has been devoted to arrays, one exception being *functional array fusion* [19]. All these techniques aim at identifying and eliminating computational pipelines, where a potentially complex intermediate data structure is synthesised by one function for the sole purpose to be analysed by another function later on. In contrast, the objective of WITH-loop-fusion is not the elimination of intermediate data structures, which in SaC is taken care of by WITH-loop-folding [7]. The essence of WITH-loop-fusion is more similar to the aims of traditional loop fusion in high performance computing in reducing loop overhead and the size of memory footprints.

There is a plethora of work on fusion of FORTRAN-style `do`-loops [20,21,22,23]. While the intentions are similar to the objectives of WITH-loop-fusion, the setting is fairly different. Despite their name, our WITH-loops represent potentially complex array comprehensions with abstract descriptions of multi-dimensional index spaces rather than conventional loops. Whereas WITH-loops define the computation of an aggregate value in an abstract way, `do`-loops merely define a control flow that leads to a specific sequence of read and write operations. Since the fusion of `do`-loops changes this sequence a compiler must be sure that both the old and the new sequence are semantically equivalent and that the new sequence is beneficial with respect to some metric. Both require the compiler to develop a deeper understanding of the programmer's intentions. Consequently, much of the work on loop fusion in FORTRAN is devoted to identification of dependences and anti-dependences on a scalar or elementary level. In contrast, the functional setting of SaC rules out anti-dependences and discloses the data flow. Rather than reasoning on the level of scalar elements, WITH-loop-fusion addresses the issue on the level of abstract representations of index spaces.

10 Conclusion and Future Work

Engineering application programs based on the principles of abstraction and composition as in SaC leads to well-structured and easily maintainable software. However, the downside of this approach is that it requires non-trivial compilation

techniques which systematically restructure entire application programs into a form that allows for efficient execution on real computing machinery.

In the current work, we have described WITH-loop-fusion as one mosaic stone of this code restructuring compiler technology. WITH-loop-fusion takes two WITH-loops and transforms them into a single generalised variant named multi-operator WITH-loop, which we have introduced as a compiler internal intermediate code representation for exactly this purpose. The positive effect of WITH-loop fusion is to avoid repeated traversals of the same array and replace memory load and store operations by equivalent but much faster register accesses. In several experiments we have demonstrated the potential of WITH-loop-fusion to achieve substantial reductions of execution times. In fact, it has proved to be a major prerequisite to make the modular programming style of SAC feasible in practice.

Individual WITH-loops also form the basis of compiler-directed parallelisation of SAC programs following a data parallel approach [24]. Like folding and scalarisation WITH-loop-fusion has the effect to concentrate computational workload scattered throughout multiple WITH-loops within a single one. Therefore, WITH-loop-fusion also improves the quality of parallelised code by reducing the number of synchronisation barriers and the need for communication. Furthermore, dealing with larger computational workload improves both the quality and the efficiency of scheduling workload to processing units.

In the future, we plan to extend WITH-loop-fusion to handle `genarray`-WITH-loops that define arrays of non-identical shape. The idea is to create a joint WITH-loop whose generators cover the convex hull of the individual WITH-loop's index spaces. Index positions not existing in one or another result array would be associated with a special value `none` and ignored by compiled code. Another area of future research is the selection of WITH-loops for fusion. As fusion of two WITH-loops may prevent further fusion with a third WITH-loop, we may want to identify the most rewarding optimisation cases on the basis of heuristics.

References

1. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. J. Functional Programming **13** (2003) 1005–1059
2. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA, IEEE Computer Society Press (2002)
3. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia. LNCS 2763, Springer-Verlag (2003) 230–235
4. Iverson, K.: A Programming Language. John Wiley, New York, USA (1962)
5. Iverson, K.: Programming in J. Iverson Software Inc., Toronto, Canada. (1991)
6. Jenkins, M.: Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. Software Practice and Experience **19** (1989) 111–126

7. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In: Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers. LNCS 1467, Springer-Verlag (1998) 72–92
8. Grelck, C., Scholz, S.B., Trojahner, K.: With-Loop Scalarization: Merging Nested Array Operations. In: Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, UK, Revised Selected Papers. LNCS 3145, Springer-Verlag (2004)
9. Appel, A.: SSA is Functional Programming. ACM SIGPLAN Notices **33** (1998)
10. Hinckfuß, K.: With-Loop Fusion für die Funktionale Programmiersprache SAC. Master's thesis, University of Lübeck, Institute of Software Technology and Programming Languages, Lübeck, Germany (2005)
11. Chin, W.: Towards an Automated Tupling Strategy. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'97), Copenhagen, Denmark, ACM Press (1993) 119–132
12. Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theoretical Computer Science **73** (1990) 231–248
13. Gill, A., Launchbury, J., Peyton Jones, S.: A Short Cut to Deforestation. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, ACM Press (1993) 223–232
14. Chin, W.: Safe Fusion of Functional Expressions II: Further Improvements. J. Functional Programming **4** (1994) 515–550
15. van Arkel, D., van Groningen, J., Smetsers, S.: Fusion in Practice. In: Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL'02), Madrid, Spain, Selected Papers. LNCS 2670, Springer-Verlag (2003)
16. Alimarine, A., Plasmeijer, R.: A Generic Programming Extension for Clean. In: Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, Sweden, Selected Papers. LNCS 2312, Springer-Verlag (2002) 168–186
17. Löh, A., Clarke, D., Jeuring, J.: Dependency-style Generic Haskell. In: Proceedings of the 8th ACM SIGPLAN Conference on Functional Programming (ICFP'03), Uppsala, Sweden, ACM Press (2003) 141–152
18. Alimarine, A., Smetsers, S.: Improved Fusion for Optimizing Generics. In: Proceedings on the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'05), Long Beach, USA. LNCS 3350, Springer-Verlag (2005)
19. Chakravarty, M.M., Keller, G.: Functional Array Fusion. In: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01), Florence, Italy, ACM Press (2001) 205–216
20. McKinley, K., Carr, S., Tseng, C.W.: Improving Data Locality with Loop Transformations. ACM Transactions on Programming Languages and Systems **18** (1996)
21. Manjikian, N., Abdelrahman, T.: Fusion of Loops for Parallelism and Locality. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 193–209
22. Roth, G., Kennedy, K.: Loop Fusion in High Performance Fortran. In: Proceedings of the 12th ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia, ACM Press (1998) 125–132
23. Xue, J.: Aggressive Loop Fusion for Improving Locality and Parallelism. In: Parallel and Distributed Processing and Applications: 3rd International Symposium, ISPA 2005, Nanjing, China. LNCS 3758, Springer-Verlag (2005) 224–238
24. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. J. Functional Programming **15** (2005) 353–401