Index Vector Elimination — Making Index Vectors Affordable

Robert Bernecky[1], Stephan Herhut[2], Sven-Bodo Scholz[2], Kai Trojahner[3], Clemens Grelck[2], and Alex Shafarenko[2]

[1] University of Toronto, Canada
bernecky@acm.org
[2] University of Hertfordshire, UK
{s.a.herhut,s.scholz,c.grelck,a.shafarenko}@herts.ac.uk
[3] University of Lübeck, Germany
trojahner@isp.uni-luebeck.de

Abstract. Compiling indexing operations on n-dimensional arrays into efficiently executable code is a challenging task. This paper focuses on the reduction of offset computations as they typically occur when transforming index vectors into offsets for linearized representations of n-dimensional arrays. We present a high-level optimization to that effect which is generally applicable, even in the presence of statically unknown rank (n). Our experiments show run-time improvements between a factor of 2 and 16 on a set of real-world benchmarks.

1 Introduction

Languages that permit us to express algorithms in a terse, consistent manner enhance our thought processes, providing us with what Ken Iverson called "tools of thought" [1]. Data-parallel array languages, such as SaC, APL, and J, fall into this class of programming languages. They offer the programmer such benefits as shape-invariant programming, terse expression, and simpler control flow.

Some of these benefits arise from the use of *index sets* to specify data-parallel indexing operations on multi-dimensional arrays [2,3,4]. Index sets may be thought of as arrays of *index vectors*, in which each index vector specifies a single element or an entire sub-array to be selected from an array. For example, the index vector `[3,4]` in SaC could be used to select the element in row three and column four of a rank-2 matrix, or to select the matrix of shape `[7,6]` at hyperplane three and plane four from a tensor of shape `[9,8,7,6]`.

As powerful as index vectors are on the level of algorithmic specifications, they open a Pandora's Box of troubles when it comes to generating highly efficient executable code from them. If index vectors actually appear in code generated by an array-language compiler, run-time performance can be severely degraded. One source of performance degradation is memory management overhead arising from dynamic allocation and deallocation of all arrays, including index vectors. To avoid superfluous memory allocations and, even more importantly, to avoid superfluous array copying, reference counting is used as predominant garbage

Z. Horváth, V. Zsók, and A. Butterfield (Eds.): IFL 2006, LNCS 4449, pp. 19–36, 2007.

collection technique. Although quite some research went into optimizing this technique, such as the work described in [5,6,7], the dynamic creation of index vectors within the innermost loops often cannot be avoided.

Another major source of performance degradation due to index vectors appears whenever the n-dimensional arrays they select from are internally represented in a linearized fashion. This requires all index vectors within selections to be translated into offsets within the linearizations of the arrays they select from. Although this may seem an inexpensive operation – $n-1$ additions and $n-1$ multiplications per selection into an n-dimensional array – it turns out that indexing operations are usually heavily used within inner loops and, therefore, have a significant impact on the overall run-time.

In a setting with fixed array dimensionality (rank) or shape, the runtime impact can be alleviated by scalarizing indexing operations and consecutively applying standard optimization techniques. However, in generic array programming languages such as SaC, where the programmer is not bound to predefine the dimensionality of an array, scalarizing indexing operations often is not possible. In this paper, we describe INDEX-VECTOR-ELIMINATION (IVE), an optimization technique that independently of the static shape-knowledge is able to eliminate redundant offset computations.

The paper is organized as follows: the next section gives a brief introduction of a stripped-down version of SaC which serves as our model language. Section 3 presents an example which demonstrates the potential for code improvements due to array indexing within a typical loop kernel, and identifies the improvements that our optimization targets. A formalization of INDEX-VECTOR-ELIMINATION, presented in Section 4, provides the required transformation schemes for our model language SaC$_\lambda$. Section 5 presents some performance figures for a set of real-world benchmarks. We discuss related work in Section 6 and draw some conclusions in Section 7.

2 SaC$_\lambda$

We now describe a stripped-down version of SaC, comprising only the bare essentials of the language: its syntax has been modified to a λ-calculus style, in order to ease comprehension by a functional-programming audience.

Figure 1 shows the syntax of SaC$_\lambda$. A program consists of a set of mutually recursive function definitions and a designated main expression. Essentially, expressions are either constants, variables or function applications. Since SaC does not, at present, support higher-order functions nor nameless functions, all abstractions (function definitions) are explicitly user-defined. Function applications are written in C-style, *i.e.*, with parentheses around arguments, rather than around entire applications of functions. Constants are either scalars or vectors of expressions enclosed by square brackets. The reader may note here, that our formal description of SaC$_\lambda$ distinguishes between *LetExpr*, *Expr*, and *Val* where one would usually expect just *Expr*. This measure eases the formal specification of code transformations in later sections. Since a transformation of the general

$$
\begin{array}{ll}
Program & \Rightarrow \left[FunId \ = \lambda Id \left[\ , Id \right]^* . LetExpr \ ; \ \right]^* \\
 & \quad \textbf{main} \ = LetExpr \ ; \\
 & \\
LetExpr & \Rightarrow Val \\
 & \ \mid \ \textbf{let} \ Id = Expr \ \textbf{in} \ LetExpr \\
 & \\
Expr & \Rightarrow Val \\
 & \ \mid \ FunId \ (\ Val \left[\ , Val \right]^* \ ) \\
 & \ \mid \ Prf \ (\ Val \left[\ , Val \right]^* \ ) \\
 & \ \mid \ \textbf{if} \ Val \ \textbf{then} \ LetExpr \ \textbf{else} \ LetExpr \\
 & \ \mid \ \textbf{with}(\ Val <= Id < Val \ ) \ : LetExpr \\
 & \quad \textbf{genarray}(\ Val \ , Val \ ) \\
 & \\
Val & \Rightarrow Const \\
 & \ \mid \ [\left[Val \left[\ , Val \right]^* \right]] \\
 & \ \mid \ Id \\
 & \\
Prf & \Rightarrow \ \textbf{shape} \ \mid \ \textbf{dim} \ \mid \ \textbf{sel} \ \mid \ * \ \mid ...
\end{array}
$$

Fig. 1. The syntax of SAC$_\lambda$

case into this restricted form is rather straight-forward, we take the liberty to ignore some of these restrictions in our examples whenever this improves their readability.

SAC$_\lambda$ provides a few built-in array operators, referred to as primitive functions (Prf). Among these are **shape** and **dim** for computing an array's shape and dimensionality (rank), respectively. A selection operation, **sel**, is also provided; it takes two arguments: an index vector, specifying the element to be selected, and an array from which to select. These basic array operations are complemented by element-wise extensions of arithmetic and relational operations, such as *multiply* ($*$) and *greater-than-or-equal* ($>=$), respectively. For improved readability, we use the latter in infix notation throughout our examples.

On top of this language kernel, SAC provides the WITH-loop, a language construct for defining array operations in a generic way. In the interest of simplified exposition, we consider only a restricted form of the WITH-loop; fully-fledged WITH-loops are described in [3].

As can be seen from Figure 1, WITH-loops in SAC$_\lambda$ take the general form:

with (*lower* <= *iv* < *upper*) : *expr*
genarray(*shape*, *default*)

where *iv* is an identifier, *lower*, *upper*, and *shape* denote expressions that should evaluate to vectors of identical length, and *expr* and *default* denote arbitrary expressions that must evaluate to arrays of identical shape. Such a WITH-loop defines an array of shape *shape*, whose elements are either computed from the expression *expr* or from the default expression *default*. Which of these two values is chosen for an individual element depends on the element's location, *i.e.*,

it depends on its index position. If the index is within the range specified by the lower bound *lower* and the upper bound *upper*, *expr* is chosen, otherwise *default* is taken. As a simple example, consider this WITH-loop, which computes the vector `[0, 2, 2, 2, 0]`:

```
with ([1] <= iv < [4]) : 2
genarray( [5], 0)
```

Note that the use of vectors for the shape of the result and the bounds of the index space (also referred to as the "generator") allows WITH-loops to denote arrays of arbitrary rank. Furthermore, the "generator expression" *expr* may refer to the index position through the "generator variable" *iv*. For example, the WITH-loop

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

yields the matrix $\begin{pmatrix} 0\,0\,0\,0\,0 \\ 0\,2\,3\,4\,0 \\ 0\,3\,4\,5\,0 \end{pmatrix}$.

We can formalize the semantics of SAC$_\lambda$ by a standard big-step operational semantics for λ-calculus-based applicative languages as defined in several textbooks, *e.g.*, [8]. The core relations, *i.e.*, those for conditionals, abstractions, and function applications can be used in their standard form. Hence, only those relations pertaining to the array specific features of SAC$_\lambda$ are shown in Figure 2.

As a unified representation for n-dimensional arrays we use pairs of vectors $< [\ shp_1, \ldots, shp_n], [\ data_1, \ldots, data_m] >$ where the vector $[\ shp_1, \ldots, shp_n]$ denotes the shape of the array, *i.e.*, its extent with respect to the n individual axes, and the vector $[\ data_1, \ldots, data_m]$ contains all elements of the array in a linearized form. Since the number of elements within an array equals the product of the number of elements per individual axis, we have $m = \prod_{i=1}^{n} shp_i$. The linearization we choose is row-major, *i.e.*, elements that correspond to variations in the rightmost index only are consecutive in the vector of elements.

The first two evaluation rules of Figure 2 show how scalars as well as vectors are transformed into the internal representation. The rule VECT requires that all elements need to be of the same shape, thereby ensuring shape consistency in the overall result.

The next three rules formalize the semantics of the main primitive operations on arrays: `dim`, `shape`, and `sel`. There are two aspects of the SEL rule to be observed: Firstly, we require the selection index to be of the same length as the shape of the array to be selected from. This ensures scalar values as results. If a more versatile selection is required, *i.e.*, a selection that may return entire subarrays, this can be achieved by embedding the selection operation into a WITH-loop. Secondly, the selection requires a transformation of the index vector into a scalar offset l into the linearized form of the array. The sum of products used here reflects the row-major linearization we have chosen.

$$\text{CONST}: \frac{}{n \rightarrow < [], [n] >}$$

$$\text{VECT}: \frac{\forall i \in \{1,\ldots,n\} : e_i \rightarrow < [\ s_1,\ \ldots,\ s_m], [\ d_1^i,\ \ldots,\ d_p^i] >}{[\ e_1,\ \ldots,\ e_n] \rightarrow < [\ \mathrm{n},\ s_1,\ \ldots,\ s_m], [\ d_1^1,\ \ldots,\ d_p^1,\ \ldots,\ d_1^n,\ \ldots,\ d_p^n] >}$$

$$\text{DIM}: \frac{e \rightarrow < [\ s_1,\ \ldots,\ s_n], [\ d_1,\ \ldots,\ d_m] >}{\texttt{dim(}\ e\texttt{)} \rightarrow < [], [n] >}$$

$$\text{SHAPE}: \frac{e \rightarrow < [\ s_1,\ \ldots,\ s_n], [\ d_1,\ \ldots,\ d_m] >}{\texttt{shape(}\ e\texttt{)} \rightarrow < [\ n], [\ s_1,\ \ldots,\ s_n] >}$$

$$\text{SEL}: \frac{\begin{array}{c} iv \rightarrow < [\ n], [\ i_1,\ \ldots,\ i_n] > \\ e \rightarrow < [\ s_1,\ \ldots,\ s_n], [\ d_1,\ \ldots,\ d_m] > \end{array}}{\texttt{sel(}\ iv\texttt{,}\ e\texttt{)} \rightarrow < [], [\ d_{l+1}] >}$$

$$\text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k)$$

$$\iff \forall k \in \{1,\ldots,n\} : 0 \leq i_k < s_k$$

$$* : \frac{\begin{array}{c} e_1 \rightarrow < [\ s_1,\ \ldots,\ s_n], [\ d_1^1,\ \ldots,\ d_m^1] > \\ e_2 \rightarrow < [\ s_1,\ \ldots,\ s_n], [\ d_1^2,\ \ldots,\ d_m^2] > \end{array}}{\texttt{*(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\ s_1,\ \ldots,\ s_n], [\ d_1^1 * d_1^2,\ \ldots,\ d_m^1 * d_m^2] >}$$

$$\text{WITH}: \frac{\begin{array}{c} e_l \rightarrow < [\ n], [\ l_1,\ \ldots,\ l_n] > \\ e_u \rightarrow < [\ n], [\ u_1,\ \ldots,\ u_n] > \\ e_{shp} \rightarrow < [\ n], [\ shp_1,\ \ldots,\ shp_n] > \\ e_{def} \rightarrow < [], [\ d] > \\ \forall i_1 \in \{l_1, ..., u_1 - 1\}\ ...\ \forall i_n \in \{l_n, ..., u_n - 1\} : \\ (\lambda\, Id\texttt{.}\, e_b\ \texttt{[}\ i_1\texttt{,}\ \ldots\texttt{,}\ i_n\texttt{]}) \rightarrow < [], d^{[i_1,\ldots,i_n]} > \end{array}}{\begin{array}{c} \texttt{with(}\ e_l\ \texttt{<=}\ Id\ \texttt{<}\ e_u\texttt{)}\ \texttt{:}\ e_b\ \texttt{genarray(}\ e_{shp}\texttt{,}\ e_{def}\texttt{)} \\ \rightarrow < [\ shp_1,\ \ldots,\ shp_n], [\ d^{[0,\ldots,0]},\ \ldots,\ d^{[shp_1-1,\ldots,shp_n-1]}] > \\ \text{where } d^{[x_1,\ldots,x_n]} = d \\ \text{iff } \exists j \in \{1, ..., n\} : x_j \in \{0, ..., l_j - 1\} \cup \{u_j, ..., shp_j - 1\} \end{array}}$$

Fig. 2. An operational semantics for SAC$_\lambda$

Element-wise extensions of standard operations such as the arithmetic and relational operations are demonstrated by the example of the rule for multiplication (*).

The last rule gives the formal semantics of the WITH-loop in SAC$_\lambda$. The first three conditions require the lower bound, the upper bound and the shape expression to evaluate to vectors of identical length. The next two conditions relate to the default expression e_{def} and the generator expression e_b, respectively. They ensure that both the default expression and generator expression evaluate to scalar values. Since the generator expression may refer to the index variable, this is formalized by transforming the generator expression into an anonymous function and by evaluating a pseudo-application of this function to all indices specified in the generator. The lower part of the WITH-loop-rule shows how the values from

the individual generator expression evaluations and the value of the default expression are combined into the overall result. The result shape vector, which stems from the shape expression, comprises a concatenation of the data vectors from the individual generator expression evaluations. Since the generator does not necessarily cover the entire index space, the default expression values need to be inserted whenever at least one element of the index vector $[i_1, \ldots, i_n]$ is outside the generator range, *i.e.*, $\exists j \in \{1, ..., n\} : x_j \in \{0, ..., l_j - 1\} \cup \{u_j, ..., shp_j - 1\}$. Formally, this is achieved by the "where clause" of the rule WITH.

3 A Motivating Example

Let us consider the following definition of an n-dimensional array `R`:

```
...let
     R = A + shift( cv, A + B)
   in ...
```

where `A`, `B`, and `cv` are variables that are defined by some surrounding context indicated by the three dots. We assume here that `A` and `B` denote n-dimensional arrays of identical shape and that `cv` is an identifier that denotes an n-element vector. Let us, furthermore, assume that `+` is a user-defined function that extends scalar addition to n-dimensional arrays in an element-wise fashion, and that `shift` implements an n-dimensional shift operation. Inlining the definitions of these functions results in a nesting of WITH-loop-defined let expressions of the form:

```
...let
     R = let
           C = with( 0*shape( A) <= iv < shape( A)) :
                  sel( iv, A) + sel( iv, B)
               genarray( shape( A), 0)
         in let
              D = with( cv <= iv < shape( C)) :
                     sel( iv-cv, C)
                 genarray( shape( C), 0)
            in with( 0*shape( A) <= iv < shape( A)) :
                  sel( iv, A) + sel( iv, D)
               genarray( shape( A), 0)
   in ...
```

Optimizations such as WITH-LOOP-FOLDING[3] transform this expression into an expression that contains a single WITH-loop:

```
...let
     R = with( cv <= iv < shape( A)) :
            sel( iv, A) + sel( iv-cv, A) + sel( iv-cv, B)
         genarray( shape( A), 0)
   in ...
```

A translation of such an expression into C-code leads to a loop nesting where the innermost loop contains the computation of `sel( iv, A) + sel( iv-cv, A) + sel( iv-cv, B)` as well as an assignment of the resulting value into the array `R` at

the index position `iv`. As we can see from the semantics definition in Section 2, these operations require the indices `iv` and `iv - cv` to be translated into suitable offsets. Using `vect2offset(` *iv*`,` *shp*`)` as a short-cut notation for this conversion of indices into offsets, we obtain code within the innermost loop that is similar to:

```
A_off0 = vect2offset( iv, shape(A));
for( k = 0; k < shape(iv)[0]; k++) {
  jv[k] = iv[k] - cv[k];
}
A_off1 = vect2offset( jv, shape(A));
B_off0 = vect2offset( jv, shape(B));
R[R_off0] = A[A_off0] + A[A_off1] + B[B_off0];
```

where the write-back offset `R_off0` is defined by the surrounding loop constructs. The n-element vector `jv` serves as a compiler-introduced variable that holds the result of the element-wise vector-subtraction `iv-cv` computed by the `for`-loop in lines 2-4. A closer look at the example reveals that all arrays involved have the same shape: we demanded `A` and `B` to have the same shape, which guarantees the element-wise addition to be well-defined. Similarly, the result needs to be of the same shape as well, since the shift operation's result matches the shape of its array argument.

With this knowledge of matching shapes, we can deduce that `vect2offset( jv, shape(A))` and `vect2offset( jv, shape(B))` in fact compute the same offset allowing us to reuse `A_off1` within the selection into `B`. Following the same line of reasoning for `R` and `A`, we can reuse `R_off0` for `A_off0`. These modifications lead to an improved loop body of the form:

```
for( k = 0; k < shape(iv)[0]; k++) {
  jv[k] = iv[k] - cv[k];
}
A_off1 = vect2offset( jv, shape(A));
R[R_off0] = A[R_off0] + A[A_off1] + B[A_off1];
```

In order to improve this code further, we need to exploit the relation between `iv` and `jv` and the consequent relation between `R_off0` and `A_off1`. This, in turn, requires us to have a closer look at the definition of `vect2offset`. From the semantics definition in Section 2 we obtain that an index vector $[i_1, ..., i_n]$ into an array of shape $[s_1, ..., s_n]$ corresponds to to the offset $\sum_{j=1}^{n}(i_j * \prod_{k=j+1}^{n} s_k)$. From linear algebra, we know that

Lemma 1. *For all vectors* $iv, cv, shp \in \mathbb{Z}^n$ *we have* `vect2offset(` $iv + cv$`,` *shp*`) = vect2offset(` *iv*`,` *shp*`) + vect2offset(` *cv*`,` *shp*`)`.

Proof. By definition of `vect2offset` *we have:*

$$
\begin{aligned}
&\texttt{vect2offset(iv + cv, shp)}\\
&= \sum_{i=1}^{n}((\texttt{iv}_i\texttt{+cv}_i)\texttt{*} \prod_{j=i+1}^{n} \texttt{shp}_j)\\
&= \sum_{i=1}^{n}(\texttt{iv}_i\texttt{*} \prod_{j=i+1}^{n} \texttt{shp}_j)\texttt{+}\sum_{i=1}^{n}(\texttt{cv}_i\texttt{*} \prod_{j=i+1}^{n} \texttt{shp}_j)\\
&= \texttt{vect2offset(iv, shp) + vect2offset(cv, shp)}
\end{aligned}
$$

This linearity in the first argument of `vect2offset` lets us lift the loop-invariant part of the index computation from the loop body by pre-computing an offset:

```
coffset = vect2offset( cv, shape(A))
```

and by defining `A_off1` as:

```
vect2offset( jv, shape(A)) + coffset
```

Subsequently, we can reuse the offset `R_off0` within the computation of `A_off1`, which yields a loop body of the form:

```
A_off1 = R_off0 + coffset;
R[R_off0] = A[R_off0] + A[A_off1] + B[A_off1];
```

Assuming `A` to be an n-dimensional array, our optimizations have reduced the number of arithmetic operations within the loop body from $7 * n - 4$ to 3, *i.e.*, we eliminate 70% of the arithmetic operations when `A` is a rank-2 matrix, and 84% when `A` is of rank 3. Furthermore, since all `vect2offset` operations have been eliminated, neither `iv` nor `jv` need to be allocated or freed within the loop body anymore.

4 Index Vector Elimination

From our example, we can see that the intended optimizations cannot be done on the level of SaC$_\lambda$ itself. Instead, we need to apply them to a level that is closer to the generated C code. The way we achieve this is to make the transformation of index vectors into offsets explicit and to separate it from the selection into the linearized array representation.

4.1 Splitting the Selection Operation

The basic idea is to introduce two new primitive operations: `vect2offset` and `idxsel`, which represent the offset computation and the selection within the linearized representation, respectively. A formal definition of their semantics is given in Figure 3. The VECT2OFFSET rule is almost identical to the SEL rule. The only difference is that instead of returning an element from the array, only the scalar offsel l is returned. This allows the IDXSEL rule to simply expect a scalar as index argument for a selection in the linearized representation of the array. Together, these two operations can be used to replace applications of the operation `sel`. The code transformation to that effect is shown in Figure 4. It shows the essential rule of a transformation scheme SPLIT which recursively traverses SaC$_\lambda$ programs and replaces every occurrence of an application `sel(` *iv*`,` *A*`)` by an expression of the form `idxsel( vect2offset(` *iv*`,` *A*`),` *A*`)`. However, the nesting restrictions on SaC$_\lambda$ require a slightly more complex pattern to look for and a nesting of let expressions as replacement. Note here, that all inserted

$$\textsc{vect2offset}: \frac{\begin{array}{c} iv \rightarrow < [\,n], [\,i_1, \ldots, i_n] > \\ e \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \end{array}}{\begin{array}{c} \texttt{vect2offset(}\ iv\texttt{,}\ e\texttt{)} \rightarrow < [], [\,l] > \\ \text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) \end{array}}$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{idxsel}: \frac{\begin{array}{c} idx \rightarrow < [], \ l > \\ e \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \end{array}}{\texttt{idxsel(}\ idx\texttt{,}\ e\texttt{)} \rightarrow < [], [\,d_{l+1}] >}$$
$$\iff 0 \leq l < m$$

Fig. 3. Operational semantics for **vect2offset** and **idxsel**

$$\text{SPLIT}\left[\!\!\left[\begin{array}{l}\texttt{let} \\ \quad Id = Expr \\ \texttt{in}\ Expr_b\end{array}\right]\!\!\right] \rightsquigarrow \begin{cases} \texttt{let} & \text{if } Expr \equiv \texttt{sel(}\ iv, A) \\ \quad idx = \texttt{vect2offset(}\ iv, A) & \\ \texttt{in let} & \\ \qquad Id = \texttt{idxsel(}\ idx, A) & \\ \quad \texttt{in}\ \text{SPLIT}[\![Expr_b]\!] & \\ \texttt{let} & \\ \quad Id = \text{SPLIT}[\![Expr]\!] & \text{otherwise.} \\ \texttt{in}\ \text{SPLIT}[\![Expr_b]\!] & \end{cases}$$

Fig. 4. Inserting explicit index computations

identifiers *idx* need to be unique, *i.e.*, they must not be used anywhere else in the given program. Those rules of the SPLIT scheme that match the remaining constructs of SaC$_\lambda$ are not shown as they only propagate the scheme into all existing sub-expressions.

The soundness of this transformation follows directly from the semantics of `sel`, `vect2offset`, and `idxsel`:

Theorem 1. SPLIT *is sound wrt. the semantics of* SaC$_\lambda$

Proof. From the semantics definitions in Figure 2 and Figure 3 we can see that it suffices to show that

$$\frac{\begin{array}{c} iv \rightarrow < [\,n], [\,i_1, \ldots, i_n] > \\ e \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \end{array}}{\begin{array}{c} \texttt{idxsel(}\ \texttt{vect2offset(}\ iv\texttt{,}\ e\texttt{)}\texttt{,}\ e\texttt{)} \rightarrow < [], [\,d_{l+1}] > \\ \text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) \end{array}}$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

We have

$$\frac{\begin{array}{c} e \rightarrow < [\, s_1, \ldots, s_n], [\, d_1, \ldots, d_m] > \\ iv \rightarrow < [\, n], [\, i_1, \ldots, i_n] > \end{array}}{\texttt{vect2offset(}\ iv\texttt{,}\ e\texttt{)} \rightarrow < [], [\, l] >}\ [\textsc{vect2offset}]$$

$$\frac{e \rightarrow < [\, s_1, \ldots, s_n], [\, d_1, \ldots, d_m] >}{\texttt{idxsel(vect2offset(}\ iv\texttt{,}\ e\texttt{),}\ e\texttt{)} \rightarrow < [], [\, d_{l+1}] >}\ [\textsc{idxsel}]$$

$$\textit{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k)$$

$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k \quad \wedge \quad 0 \leq l < m$$

All that remains to show is that the condition $0 \leq l < m$ *is redundant.* $0 \leq l$ *follows directly from the definition of* l *and the requirement that* $0 \leq i_k < s_k$ *for all* k*. From the latter we can furthermore deduce that* $l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) \leq \sum_{j=1}^{n} ((s_j - 1) * \prod_{k=j+1}^{n} s_k) = \sum_{j=1}^{n} (\prod_{k=j}^{n} s_k - \prod_{k=j+1}^{n} s_k) = \prod_{k=1}^{n} s_k - \prod_{k=n+1}^{n} s_k < \prod_{k=1}^{n} s_k = m.$ *q.e.d.*

Once all offset computations are explicit, the three optimizations explained informally in the previous section can now be formalized.

4.2 Reusing Offset Computations

In order to reuse offset computations, we must identify expressions of the form `vect2offset(`iv`,` A`)` and `vect2offset(`iv`,` B`)` where the shapes of A and B are statically known to match. There are several ways to determine this equality. Once the shapes of A and B are statically known [9], equality can be statically decided. Even without the presence of static shape knowledge, shape equality often can be statically decided using inference techniques such as *Shape Clique Inference*, outlined in [10], or *Symbolic Array Attributes*, outlined in [11]. For our purposes here, we assume this information to be available.

Figure 5 shows the key rule of a transformation scheme REUSE that identifies such situations and replaces the second application of `vect2offset` by the offset computed from the first one. The REUSE scheme maps SAC_λ programs and an environment S of identifier triples to a potentially modified program. Triples (iv, A, idx) each represent an existing definition of an offset idx by an application of `vect2offset` to an index vector iv and an array A. The scheme starts out with an empty environment and traverses into all subexpressions. Whenever an application `vect2offset(`iv`,` A`)` is found, the environment is searched for an entry (iv, B, idx) with `shape(` B`)` = `shape(` A`)`. If found, the application of `vect2offset` is replaced by the variable idx, otherwise a new triple is appended to the end of S, denoted by $+\!\!+$ as symbol for concatenation. Note that our syntactic restrictions ensure that we always find identifiers in argument position.

$$\text{REUSE}\left[\!\!\left[\begin{array}{l}\texttt{let}\\ \quad Id = Expr,\ S\\ \texttt{in}\ Expr_b\end{array}\right]\!\!\right]$$

$$\rightsquigarrow \begin{cases} \texttt{let} & \text{if } Expr \equiv \texttt{vect2offset}(\ iv,\ A) \\ \quad Id = idx & \text{and } \exists < iv,\ B,\ idx > \in S \\ \texttt{in REUSE}[\![Expr_b,\ S]\!] & \text{with } shape(\ A) = shape(\ B) \\ \texttt{let} & \text{if } Expr \equiv \texttt{vect2offset}(\ iv,\ A) \\ \quad Id = Expr & \text{and } \nexists < iv,\ B,\ idx > \in S \\ \texttt{in REUSE}[\![Expr_b,\ S +\!\!+ < iv,\ A,\ Id >]\!] & \text{with } shape(\ A) = shape(\ B) \\ \texttt{let} & \\ \quad Id = \text{REUSE}[\![Expr,\ S]\!] & \text{otherwise.} \\ \texttt{in REUSE}[\![Expr_b,\ S]\!] & \end{cases}$$

Fig. 5. Reusing index computations

The soundness of this transformation follows almost directly from the shape equality predicate:

Theorem 2. REUSE *is sound wrt. the semantics of* SaC$_\lambda$.

Proof. Given a subexpression `vect2offset`*(iv, A). From the definitions in Figure 5 we know that* $< iv, B, idx > \in S$*, iff we already encountered a subexpression of the form idx* `= vect2offset(` *iv, B) while traversing the program. Given that* `vect2offset`*(iv, A)* $\rightarrow < [], [\ l] >$ *and the shape equivalence of A and B, we can conclude that* `vect2offset`*(iv, B)* $\rightarrow < [], [\ l] >$*. As all identifiers and idx in particular are unique, i.e. there is only a single assignment, it follows that* $idx \rightarrow < [], [\ l] >$*.* *q.e.d.*

4.3 Reusing WITH-loop Offsets

The REUSE scheme introduced in the previous subsection only detects previous applications of `vect2offset` as potential reuse candidates. From our example in Section 3, we have seen that we often can reuse the offset for storing individual WITH-loop-computed elements into the overall WITH-loop-result. Due to the data-parallel nature of WITH-loops, this "assignment" and thus the required offset is not explicit in SaC$_\lambda$. We formalize this optimization on a higher level than the generated C-code by taking a similar approach as with the initial splitting of the `sel` operation. We transform our WITH-loops into a slightly lower-level variant `idxwith` that makes the "write-back-offset" explicit. Its syntax differs from that of standard WITH-loops only by an additional identifier within the generator. This second generator variable introduces a name for the write-back-offset which in the body of the WITH-loop can be referred to. The formal semantics of `idxwith` are given in Figure 6. As we can see from the semantics it is almost identical to that of the standard WITH-loop. In fact, the only difference is that

$$\text{IDXWITH}: \frac{\begin{array}{c} e_l \rightarrow < [\,n], [\,l_1, \ldots, l_n] > \\ e_u \rightarrow < [\,n], [\,u_1, \ldots, u_n] > \\ e_{shp} \rightarrow < [\,n], [\,shp_1, \ldots, shp_n] > \\ e_{def} \rightarrow < [], [\,d] > \\ \forall i_1 \in \{l_1, ..., u_1 - 1\} \ldots \forall i_n \in \{l_n, ..., u_n - 1\} : \\ (\lambda\, Id\,.\,(\lambda\, Idx\,.\,e_b\; p)\;\texttt{[}\; i_1\texttt{,}\; \ldots\texttt{,}\; i_n\texttt{])} \rightarrow < \texttt{[]}\texttt{,}\; d^{[i_1,\ldots,i_n]} > \end{array}}{\begin{array}{c} \texttt{idxwith(}\; e_l\; \texttt{<=}\; Id\texttt{,}Idx\; \texttt{<}\; e_u\texttt{)}\; \texttt{:}\; e_b\; \texttt{genarray(}\; e_{shp}\texttt{,}\; e_{def}\texttt{)} \\ \rightarrow < [\,shp_1, \ldots, shp_n], [\; d^{[0,\ldots,0]}, \ldots, d^{[shp_1-1,\ldots,shp_n-1]}] > \\ \text{where } d^{[x_1,\ldots,x_n]} = d \\ \text{iff } \exists j \in \{1, ..., n\} : x_j \in \{0, ..., l_j - 1\} \cup \{u_j, ..., shp_j - 1\} \\ \text{and } p = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} shp_k) \end{array}}$$

Fig. 6. Further extended operational semantics for SaC$_\lambda$

$$\text{REUSE}\left[\!\!\left[\begin{array}{l} \texttt{let} \\ \quad Id = \texttt{with(}\; lb\; \texttt{<=}\; iv\; \texttt{<}\; ub\texttt{)}\; \texttt{:}\; Expr_{body}\texttt{,}\; S \\ \qquad\quad \texttt{genarray(}\; shp\texttt{,}\; def\texttt{)} \\ \texttt{in}\; Expr \end{array}\right]\!\!\right]$$

$$\rightsquigarrow \begin{array}{l} \texttt{let} \\ \quad Id = \texttt{idxwith(}\; lb\; \texttt{<=}\; iv\texttt{,}\; wlidx\; \texttt{<}\; ub\texttt{)}\; \texttt{:} \\ \qquad\quad \text{REUSE}[\![Expr_{body}\texttt{,}\; S\; \texttt{++} <\; iv\texttt{,}\; Id\texttt{,}\; wlidx\; >]\!] \\ \qquad \texttt{genarray(}\; shp\texttt{,}\; def\texttt{)} \\ \texttt{in}\; \text{REUSE}[\![Expr\texttt{,}\; S]\!] \end{array}$$

Fig. 7. Reusing WITH-loop offsets

the body expression e_b now is extended by two variable definitions rather than one: Id for the actual index vector and Idx for the corresponding write-back-offset.

Before we can try to use this new offset, we need to transform all WITH-loops accordingly. This is achieved by slightly extending the REUSE scheme from the previous section. The additional rule presented in Figure 7 replaces all standard WITH-loops by the new `idxwith`-version. Again, all introduced identifiers *wlidx* need to be unique.

Whenever a WITH-loop is transformed, a new triplet is added to the environment containing the name of the index variable *iv*, the name of the array to be computed Id, and the write-back-offset *wlidx* introduced by the `idxwith`-version of the WITH-loop. This information can then be used for the substitution of redundant `vect2offset` computations as described in the previous subsection.

Theorem 3. *The extended* REUSE *is sound wrt. the semantics of* SaC$_\lambda$.

Proof. Analog to Theorem 2. *q.e.d.*

4.4 Splitting Offset Computations

In our motivating example, we have seen that splitting an offset computation whose index vector stems from a sum/difference of vectors into a sum/difference of offset computations often triggers further reuse or other optimizations such as LOOP-INVARIANT-REMOVAL [12]. With the offset computations being made explicit by the SPLIT scheme, we can now define another scheme SOC which detects such situations and transforms the offset computation accordingly. Similar to the REUSE scheme, the SOC scheme takes an additional parameter which carries quadruples consisting of 3 identifiers and one arithmetic operation; it collects these quadruples (Id, $LinOp$, jv, kv), which represent an application of either `+` or `-` (denoted by $LinOp$) to two arrays jv and kv whose result is kept in a variable Id, while traversing through the program. Whenever the traversal finds an application of `vect2offset` to an identifier that is the first component of any of the quadruples seen so far, the code transformation is triggered. Figure 8 shows the main rule of the SOC scheme. Due to our restricted syntax both situations of interest are captured in the context of a let expression. If an addition or a subtraction is encountered, a new quadruple is inserted into the environment. Applications of `vect2offset` are only transformed if the index argument is known to be a sum/difference of vectors, *i.e.*, if a quadruple with the index variable as first component is contained in the environment. Note here, that the scheme is applied recursively to the result of the transformation. This ensures that arbitrary nestings of index operations will be properly split. In all other cases, the transformation is applied to the subexpressions only.

$$
\mathrm{SOC}\left[\!\left[\begin{array}{l}\texttt{let}\\ \quad Id = Expr,\ E\\ \texttt{in}\ Expr_b\end{array}\right]\!\right]
$$

$$
\rightsquigarrow \begin{cases}
\begin{array}{l}\texttt{let}\\ \quad Id = Expr\\ \texttt{in}\ \mathrm{SOC}[\![Expr_b,\ E \cup < Id,\ LinOp,\ jv,\ kv >]\!]\end{array} & \text{if } Expr \equiv\ LinOp(\ jv,\ kv), \\
\mathrm{SOC}\left[\!\left[\begin{array}{l}\texttt{let}\\ \quad jv_{off} = \texttt{vect2offset}(\ jv,\ A)\\ \texttt{in let}\\ \quad\ kv_{off} = \texttt{vect2offset}(\ kv,\ A),\ E\\ \quad \texttt{in let}\\ \quad\quad Id = LinOp(\ jv_{off},\ kv_{off})\\ \quad\quad \texttt{in}\ Expr_b\end{array}\right]\!\right] & \begin{array}{l}\text{if } Expr \equiv\\ \quad \texttt{vect2offset}(\ iv,\ A)\\ \text{and}\\ \quad < iv,\ LinOp,\ jv,\ kv> \in E\end{array}\ . \\
\begin{array}{l}\texttt{let}\\ \quad Id = Expr\\ \texttt{in}\ \mathrm{SOC}[\![Expr_b,\ E]\!]\end{array} & \text{otherwise.}
\end{cases}
$$

Fig. 8. Splitting of `vect2offset` operations on linear combinations

Theorem 4. SOC *is sound wrt. the semantics of* SaC$_\lambda$.

Proof. The theorem follows immediately from Lemma 1. *q.e.d.*

Although the transformation is correct wrt. the semantics of SaC$_\lambda$ its application bears several problems.

Firstly, the semantics does not make any assumptions about the representation of the indices. However, Lemma 1 only holds for index vectors from $\mathbb{Z}^n$ not for finite subsets of $\mathbb{Z}^n$ such as n-element `integer`-vectors. Here, we may have to deal with overflow problems: while an expression `vect2offset( iv-cv, A)` may be within the limits of a given `integer` format, `vect2offset( iv, A)` or `vect2offset( cv, A)` may not. As a consequence, a transformed program may yield a runtime error although the untransformed one does not. The only way to avoid this problem is to restrict the transformation to those cases where we can statically prove that the offset computations are within the limits of the chosen index representation. It turns out that this is the frequent case as the indices are usually composed from a WITH-loop-generated index and a constant offset vector. For both of these, a static guarantee can be computed if the shape of the array to be selected from is statically known.

The second difficulty with this transformation stems from the fact that the transformation by itself leads to a code degradation if we are dealing with vectors of length ≥ 2: We replace a vector addition of an n-element vector (n operations) and an offset computation ($2*n-2$ operations) by one scalar operation and two offset computations ($4*n-4$ operations). Only the fact that this transformation often triggers other optimizations such as the REUSE scheme of the IVE or LOOP-INVARIANT-REMOVAL has a positive runtime effect. As a consequence, a conservative implementation needs to apply a reverse transformation if such sums of offset computations remain after an application of the aforementioned optimizations.

These considerations lead to the following order of transformations during INDEX-VECTOR-ELIMINATION: First, we apply the SPLIT-scheme in order to make the offsets explicit, followed by the SOC-scheme which may generate further offset computations. Then, we apply REUSE and LOOP-INVARIANT-REMOVAL in order to get rid of as many offset computations as possible. Finally, we revert those transformations of the SOC-scheme whose components have neither been eliminated nor have been moved by LOOP-INVARIANT-REMOVAL.

5 Performance

In our motivation example, we were able to save at least 70% of the arithmetic instructions within the inner loop by applying INDEX-VECTOR-ELIMINATION provided we were dealing with at least rank 2 arrays. Of course, this represents a best-case scenario, as we were able to remove all index computations. To get an idea of the impact of INDEX-VECTOR-ELIMINATION on real-world applications, we measured the performance gains archived by applying INDEX-VECTOR-ELIMINATION to two sets of benchmarks: The first set is taken from a SaC benchmark

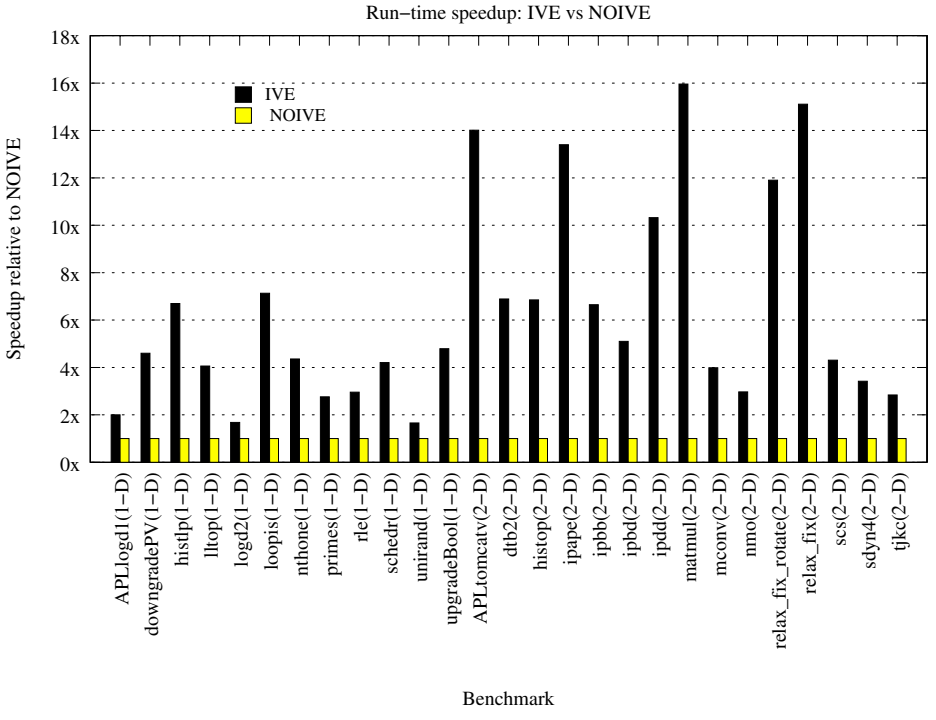

Fig. 9. Performance with and without IVE

suite that has previously been used for comparisons with FORTRAN, whereas the remainder are APL-derived, APEX-generated SAC programs. Our benchmark suite represents a mix of array ranks. However, all our benchmarks are either dominated by rank-1 or by rank-2 arrays.

5.1 Experimental Framework

We used an AMD-based platform (Opteron 165 (1.8GHz)) equipped with 4GB of RAM, operating SuSE Linux 10.1 64-bit. For compiling the SAC source code we used the current version of the `sac2c` compiler (rev 15076) with the GNU `gcc` compiler version 4.1.0 as the back-end compiler. We enabled the default set of optimizations, which include standard optimizations, such as COMMON-SUB-EXPRESSION-ELIMINATION, LOOP-INVARIANT-REMOVAL and LOOP-UNROLLING, as well as SAC-specific optimizations like WITH-LOOP FOLDING, WITH-LOOP SCALARIZATION and WITH-LOOP FUSION (for details on the default optimizations of SAC see [3]). The resulting C code was compiled using the `-O3` option of `gcc`.

To enable measurement of the impact of INDEX-VECTOR-ELIMINATION on the run time of each benchmark, we created one executable with INDEX-VECTOR-ELIMINATION enabled and one with that optimization disabled. We measured execution time using **user time** from the Linux `/usr/bin/time` function.

5.2 Analysis

Our results are presented in Figure 9. For each benchmark there are two bars: a black one representing the runtime with IVE enabled and a light gray one denoting the runtime with IVE disabled. Since the IVE-enabled times were always faster, we use the non-IVE times as a reference time, displaying speedups against that time rather than absolute runtimes. Higher black bars indicate higher runtime performance.

We sorted the benchmarks according to their dominant array rank: rank-1 examples are on the left; rank-2 ones are on the right. We can see that the rank-1 examples gain by a factor of 2 to 4 times. Since we know that rank-1 arrays cannot profit from any reuse or splitting of offset computations, this effect can be attributed to the avoidance of dynamic allocation of 1-element vectors. The stark variation in the effect derives from the differences in memory access/computation ratios found in the various benchmarks.

For the rank-2 examples, the gains vary even more. Here, our reuse optimizations and the offset computation splittings contribute as well, producing speedups between about 3 and 16 times. The gain in speedup *vs.* the rank-1 examples thus varies between a factor of 1.5 and 4, showing that our theoretical example falls nicely into that range.

6 Related Work

INDEX-VECTOR-ELIMINATION addresses a rather specific setting: the translation of n-dimensional selections specified as shape vectors into scalar offsets into linearized representations of n-dimensional arrays. This setting prevails in array languages such as APL, NIAL, and J. However, most of these languages have traditionally been interpreted, rather than compiled, because it was thought, for many years, that language semantics precluded effective application of optimizations such as INDEX-VECTOR-ELIMINATION. Of the several APL compiler projects that have been conducted, including [13,14,15,16,17,18], most do not achieve a very high-level of optimization. The APEX [18] compiler is the only project we are aware of that aims at utmost run-time efficiency. That run-time efficiency was enabled, in some degree, by the advent of Static Single Assignment, and the SISAL project. The former was a key factor in improving data flow analysis; the latter pushed the state of the art with respect to vector-oriented optimizations. Both of these ultimately had an impact on run-time code efficiency.

Although the SISAL compiler achieved very good run-time performance [19], a counterpart to INDEX-VECTOR-ELIMINATION was not required, as SISAL represents n-dimensional arrays as nestings of vectors. However, that run-time representation is less favorable for higher-dimensional problems, as described in [18,20]. These observations led to a proposal for *true n-dimensional arrays* in SISAL [21]. Although several implementation issues and optimizations on these linearized representations are described in [20,22] none of them pertains to IVE.

The APEX project recently switched from generating SISAL code to generating SaC code, to avoid fundamental algebraic limitations of nested vectors as a method of representing arrays. and in order to be able to make use of optimizations such as INDEX-VECTOR-ELIMINATION. In fact, run-time deficiencies of APEX-generated SaC-code partially triggered this research.

One key element of INDEX-VECTOR-ELIMINATION is the existence and use of a shape predicate as explained in Section 4. It can be derived from such techniques as *Shape Clique Analysis* [10] or *Symbolic Array Attributes* [11].

7 Conclusions

This paper presents INDEX-VECTOR-ELIMINATION, an optimization for avoiding run-time overhead arising from index vectors and their conversions into scalar offsets for linearized array representations. We describe three program traversals which, when orchestrated properly, for most examples, eliminate all index vectors within the innermost loop and reuse, to a large extent, offset computations. We formally describe the transformations, prove their soundness, and discuss their effectiveness in terms of arithmetic operations involved.

Although the run-time overhead may seem negligible, it turns out that nearly all array-dominated applications can benefit significantly from IVE. Our measurements for a set of benchmark kernels on a variety of array ranks show that speedups of 2 to 16 can be expected depending on the predominant array rank and the nature of the application.

Acknowledgments

We thank the anonymous referees for their valuable feedback and the European Union IST-FET research project Æther for funding this work. For more information on Æther, see `www.aether-ist.org`.

References

1. Iverson, K.E.: Notation as a tool of thought. Communications of the ACM vol. 23(8) (1979)
2. International Standards Organization: International Standard for Programming Language APL. ISO N8485 edn (1984)
3. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13(6), 1005–1059 (2003)
4. Hui, R.K., Iverson, K.E.: J Dictionary (1998)
5. Cann, D.: Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California (1989)
6. Cann, D., Evripidou, P.: Advanced Array Optimizations for High Performance Functional Languages. IEEE Transactions on Parallel and Distributed Systems 6(3), 229–239 (1995)

7. Grelck, C., Trojahner, K.: Implicit Memory Management for SaC. In: Grelck, C., Huch, F., Michaelson, G.J., Trinder, P. (eds.) IFL 2004. LNCS, vol. 3474, pp. 335–348. Springer, Heidelberg (2005)
8. Pierce, B.: Types and Programming Languages. MIT Press, Cambridge, ISBN 0-262-16209-1 (2002)
9. Grelck, C., Scholz, S.B., Shafarenko, A.: A Binding-Scope Analysis for Generic Programs on Arrays. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, Springer, Heidelberg (2006)
10. Bernecky, R.: Shape Cliques. In: Horváth, Z., Zsók, V., eds.: Proceedings of the 18th International Symposium on Implementation of Functional Languages (IFL'06), Eötvös Loránd University (2006)
11. Trojahner, K., Grelck, C., Scholz, S.B.: On Optimising Shape-Generic Array Language Programs using Symbolic Structural Information. In: Horváth, Z., Zsók, V. (eds.) Proceedings of the 18th International Symposium on Implementation of Functional Languages (IFL'06). Revised Selected Papers, LNCS, vol. 4449, Springer, Heidelberg (2006)
12. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22(2), 96–103 (1979)
13. Bernecky, R., Brenner, C., Jaffe, S.B., Moeckel, G.P.: ACORN: APL to C on real numbers. ACM SIGAPL Quote Quad 20(4), 40–49 (1990)
14. Weigang, J.: An Introduction to STSC's apl compiler. APL89 Conference Proceedings, ACM SIGAPL Quota Quad 15, 231–238 (1989)
15. Ching, W.M.: An APL/370 compiler and some performance comparisons with APL interpreter and FORTRAN. ACM SIGAPL Quote Quad 16(4), 143–147 (1986)
16. Wiedmann, C.: Field results with the APL compiler. ACM SIGAPL Quote Quad 16(4), 187–196 (1986)
17. Budd, T.A.: An APL compiler for the UNIX timesharing system. ACM SIGAPL Quote Quad 13(3) (1983)
18. Bernecky, R.: APEX: The APL parallel executor. Master's thesis, University of Toronto (1997)
19. Cann, D.: The Optimizing SISAL Compiler: Version 12.0. Lawrence Livermore National Laboratory, LLNL, Livermore California. Part of the SISAL distribution (1993)
20. Oldehoeft, R.: Implementing Arrays in SISAL 2.0. In: Proceedings of the Second SISAL Users' Conference. pp. 209–222 (1992)
21. Böhm, A., Cann, D., Oldehoeft, R., Feo, J.: SISAL Reference Manual Language Version 2.0. CS 91-118, Colorado State University, Fort Collins, Colorado (1991)
22. Fitzgerald, S., Oldehoeft, R.: Update-in-place Analysis for True Multidimensional Arrays. In: Böhm, A., Feo, J., eds.: High Performance Functional Computing. pp. 105–118 (1995)