EMBEDDING DATAFLOW INFORMATION ON ARRAYS INTO SSA AND EXTENDED OPTIMIZATION SCHEMA FOR PARALLELIZATION

SATO, Hiroyuki
Information Technology Center, The University of Tokyo, Japan
Email: schuko@satolab.itc.u-tokyo.ac.jp

ABSTRACT
This paper presents an improvement of SSA (static single assignment) to embed dataflow information of arrays. Because dataflow representation in SSA is based on names, an extension of SSA to include array analysis needs essential improvement. We regard an array as a collection of individual scalars. By using symbolic analysis, we solve equations between indices, and extend SSA to include arrays. ϕ functions are extended to δ in order to represent dependence on previous iterations. Furthermore, γ is introduced to represent confluence of indexed variables. Optimizations using our extended SSA are also discussed. Scalar optimizations using SSA are naturally extended to arrays. In our model, δ corresponds to synchronization between iterations, and its elimination increases the opportunity of parallelization.

KEY WORDS
SSA, Parallelizing compiler, Compiler optimization, Dataflow analysis

1 Introduction

Much effort has been done on improving the performance of parallelizing compilers. Because most programs for high performance computing handle arrays in loops, loop-related optimizations on arrays are essential for performance improvement. Moreover, almost every loop-related optimization(= loop restructuring) is based on dataflow analysis of arrays, the precision of the dataflow analysis determines the performance. Therefore, there have been proposed a number of schemes for dataflow analysis[11, 13].

Dataflow is also essential for general compiler optimizations. A number of optimization techniques for scalars including dead code elimination, constant propagation, induction variable recognition, and register allocation, are based on specific dataflow equations. It is very natural that intermediate representations suitable for dataflow analysis have been intensively studied. Among those, SSA(static single assignment) form has been proposed in [2], and has been proved very powerful in dataflow analysis of scalars. SSA renames variables so that there is only one definition for each variable. By using this scheme, dataflow is naturally expressed as uses of only-once defined variables. In this sense, SSA has dataflow information in itself. Because of this advantage, we can naturally express def-use chains by using SSA. Furthermore, the pseudofunction ϕ representing control flow of branches and backedges is introduced. Thanks to ϕ, SSA obtains wider applicability to almost every scalar program. Today, it is one of standard intermediate representation forms.

The fact that SSA embeds dataflow information implies that SSA can be closely related with parallelism. Conditions that parallel execution is possible are expressed by using SSA. Moreover, there has been studied direct execution of SSA with connection to parallelism[4, 9].

However, when applying SSA to dataflow analysis of arrays, we have a flaw. The reason is essential: the dataflow representation in SSA is based on names. A naive application is based on "array as a monolithic object" scheme, which does not require the analysis of indices. This is out of question in parallelizing compilers.

In this paper, we propose an extension of SSA based on "array as a collection of indexed variables." This is a solution of the problem of integration of scalar analysis and array data dependence analysis. In our scheme, dataflow analysis of arrays is divided into loop-carried dependence and loop-independent dependence. Loop carried dependence, a major obstruct for parallelization, is naturally embedded by using our new pseudofunction δ. Symbolic analysis is applied to solve equations related with δ.

The rest of this paper is organized as follows: Section 2 defines our extension to SSA by using δ and γ. In Section 3, we show that optimizations using our SSA are closely related with synchronization optimizations and parallelism extraction. Section 4 discusses the relation of our SSA with previous work. Section 5 gives a summary of this paper.

2 Array SSA

In this paper, we measure the correctness of def-use chain by *covering*:

Definition 1 (Cover) First, we consider a translation of an original program to an SSA-ed one. A translated SSA-form *covers* the original program (fragment) if the def-use relation in the translated SSA-form is a superset of that in the original program.

The translation is required to cover the original program. The SSA translation for scalars covers the original program. We require the same criteria to be applied in the case of arrays.

The conventional and naivest covering translation is to regard an array as a monolithic object. In this translation, an index part is ignored. An update of a part of an array is considered to be an update to the whole array. We call this translation *Naive*. By this definition, *Naive* is a covering, because every write occurence of an array is linked in the order they appear. The approach of [5] is also a covering, because in every write, there is inserted a ϕ-pseudofunction, and every write occurrence is linked in the order they appear.

In our "array as a collection of individual variables" scheme, we will show another covering translation under the restriction that all indices of arrays are affine expressions of loop indices. Note that this restriction is reasonalbe in the field of high performance computing. Moreover, we assume that loop indices are not allowed to be written as in Fortran.

Let two occurences $a[g(i)]$ and $a[f(i')]$ for iterations i, i' be given. If the access $a[g(i)]$ uses $a[f(i')]$ of iteration i' such that $i - i' > \vec{0}$ ($i > i'$ for short), the equation is written as $f(i') = g(i)$. The solution of i' is $f^{-1}(g(i))$ if $f^{-1}(g(i)) < i$. If f and g are given as matrices, this can also be given as an affine expression of loop indices. We denote such f's by *access function* of the occurence $a[f(i)]$. Furthermore, the dimensions of f and g are the nest levels of the loop. We call this solution the *solution vector*. A subspace of solution vectors is given as a function of i, f and g.

Note that given an index vector i, and a solution vector $s(i)$, the distance vector is calculated as $(s - I)(i)$. In this sense, a solution vector is a generalization of a distance vector.

Formally, we first extend the definition of variables to define our array SSA.

Definition 2 (Indexed Variable) Given an array name a and an expression e, $a[e]$ is a variable. We denote it by an *indexed variable*. The identity between indexed variables is given in the way such that $a[e]$ and $a'[e']$ are identical only if $a \equiv a'$ and $e \equiv e'$.

For the indexed variable $a[e]$, a is denoted by its *name part*, and e is denoted by its *access part*.

Definition 3 Let an occurrence in a program be given. Associated with the occurrence, there is defined a loop nest with a vector of loop index $(i_0, i_1, \cdots, i_n)$. We denote $(i_0, i_1, \cdots, i_n, 1)$ by *loop nest vector*.

Note that with every occurrence of variable is associated a loop nest vector. We call it *loop nest vector of the occurrence*. Loop nest vectors are pre-ordered by the relation that for each element, i is greater than i' iff $i - i' > \vec{0}$. As for loop nest vectors, v is greater than v' if the occurrence of v appears lexically before that of v'. We confusingly use $>$ for this order.

Definition 4 A δ-pseudofunction is composed of two lists of a pair of indexed variable occurrences and its associated access function. Specifically, δ-function is of the form: $\delta((v_0, acc_{v_0}), \cdots, (v_n, acc_{v_n}); (w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m}); \ lnv)$, where $v_i (i = 0, \cdots, n)$ and $w_j (j = 0, \cdots, m)$ are occurrences of a variable, acc_i is its associated access function, and lnv is a vector representing an access function of the indexed variable at the occurrence.

We denote v_i's by its initialization part, and w_j's by its backedge part.

Definition 5 Consider a δ-pseudofunction $\delta((v_0, acc_{v_0}), \cdots, (v_n, acc_{v_n}); (w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m}); \ lnv)$. Its interpretation in a location whose loop nest vector is $(i_0, i_1, \cdots, i_\ell, 1)$ is given as: if it is not the first visit to this δ, δ indicates w_i at the iteration $s \in acc_{w_i}^{-1}(lnv)$ where s is the maximum among solutions in w's, or the backedge part such that $s < (i_0, \cdots, i_\ell)$. Otherwise, this means that it is the first visit to the δ or there is no solution in the above equations. Therefore, δ indicates v_j where $s \in acc_{v_j}^{-1}(lnv)$ is the maximum among the solutions of the initialization part.

From those discussions, we can conclude that δ represents the loop-carried data dependence and synchronization between iterations.

Definition 6 We use $\perp$ as an access function meaning 'don't care.'

Example 1 Consider the program below:

```
(1) for i = 1, 100
      for j = 1, 100
        a(i, j) = a(i, j − 1) + 1
      endfor
(2)   for k = 1, 100
        a(i, k) = a(i, k + 1) + 2
      endfor
    endfor
```

As for indexed variables, we define variables associated with occurrences $a(i, j)$, $a(i, j - 1)$, $a(i, k)$, and $a(i, k + 1)$ respectively. Moreover, the loop nest associated with each variable is $(i, j, 1)$, $(i, j, 1)$, $(i, k, 1)$, and $(i, k, 1)$ respectively. The access function associated with each variable is given as

$$A_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, A_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix},$$
$$A_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, A_4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix},$$

respectively.

At (1), the loop nest vector is given as $(i, 1)$, while at (2), the vector is given as $(i, k, 1)$. At (1), the expression $\delta((a_{init}, \perp); (a[i, j]_0, A_1), (a[i, k]_0, A_3); (i, k + 1, 1))$ is a valid δ function. Because the loop nest vector of occurrences $a[i, j]$ and $a[i, k]$ are given as (i, j) and (i, k) respectively, the semantics of the δ function is

given by an occurrence of $a[i,j]$ or $a[i,k]$ whose loop nest (i',k') satisfies the constraints: $(i',k') = (i,k+1)$ and $(i',k') < (i)$. In this case, there is no solution, and a_{init} is taken[1] At the point (2), the expression $\delta((a[i,k+1]_0,\perp);(a[i,k]_0,A_3),(a[i,k+1]_1,A_4);(i,k,1))$ is also valid, and $a[i,k+1]_0$ is taken as the solution.

We need one more pseudofunction γ which represents loop-independent dependence.

Definition 7 γ-pseudofunction is composed of a condition, and the nodes representing confluence of the control flow. Specifically, γ-function is of the form:

$$\gamma(P; v_0, v_1),$$

where P is a formula expressing conditions , and $v_i (i = 0, 1)$ is a variable. The interpretation of γ is given as:

if (P) then (v_0 s.t. (P)) else (v_1 s.t.$\neg$(P)).

In the rest of this paper, we use the form of $acc_v = acc_w$ as the condition of γ pseudofunction. We describe the interpretation of $\gamma(acc_v = acc_w; v, w)$ as: let $\Gamma \equiv \gamma(acc_{v_0} = acc_{v_1}; v_0, v_1)$ be given at a location ℓ whose loop nest vector is $(\ell_0, \cdots, \ell_n, 1)$. Let $I_{v_i} = (i_0, \cdots, i_{m_{v_i}}, 1)$ be the loop nest vector associated with $v_i (i = 0, 1)$. We assume $n \leq m_{v_i}$. Then, Γ's interpretation is given as:

if $(\exists j_{n+1} \cdots j_{m_{v_0}}, j'_{n+1} \cdots j'_{m_{v_1}}$.
$acc_{v_0}[\ell_0/i_0, \cdots, \ell_n/i_n, j_{n+1}/i_{n+1}, \cdots, j_{m_{v_0}}] =$
$acc_{v_1}[\ell_0/i_0, \cdots, \ell_n/i_n, j'_{n+1}/i_{n+1}, \cdots, j'_{m_{v_1}}]$
and the locations which have I_{v_i}s as loop nest vectors are lexicographically before the location ℓ.)
then v_0 such that its occurrence is the maximum of those satisfying the above condition
else v_1

For this fixed interpretation, we omit its condition-part if it is clear from context. Moreover, we write $\gamma(v_0, v_1, \cdots v_n)$ for nested γ-functions $\gamma(v_0, \gamma(v_1, \cdots \gamma(v_{n-1}, v_n) \cdots))$.

We translate a loop with access to an array to an SSA form with δ-nodes and γ-nodes. These principles apply to multiple definitions and uses. Let us consider a program:

loop i

$a[f(i)] \longleftarrow \cdots$
$\cdots$
$\longleftarrow a[g(i)]$
$\cdots$
$a[h(i)] \longleftarrow \cdots$
$\cdots$
$\longleftarrow a[k(i)]$
endloop

[1] a_{init} is also written as $a_\perp$ in Figures.

The loop can be translated to the form below:

loop i
$a[f(i)]_0 \longleftarrow$
$\delta((a[f(i)]_{init},\perp);(a[f(i)]_1,f),(a[h(i)]_1,h);(f(i),1))$
$a[g(i)]_0 \longleftarrow$
$\delta((a[g(i)]_{init},\perp);(a[f(i)]_1,f),(a[h(i)]_1,h);(g(i),1))$
$a[h(i)]_0 \longleftarrow$
$\delta((a[h(i)]_{init},\perp);(a[f(i)]_1,f),(a[h(i)]_1,h);(h(i),1))$
$a[k(i)]_0 \longleftarrow$
$\delta((a[k(i)]_{init},\perp);(a[f(i)]_1,f),(a[h(i)]_1,h);(k(i),1))$
$\cdots$
$a[f(i)]_1 \longleftarrow \cdots$
$\cdots$
$a[g(i)]_1 \longleftarrow \gamma(f(i)=g(i); a[f(i)]_0, a[g(i)]_0)$
$\longleftarrow a[g(i)]_1$
$a[h(i)]_1 \longleftarrow \cdots$
$a[k(i)]_1 \longleftarrow$
$\gamma(h(i)=k(i); a[h(i)]_1, \gamma(f(i)=k(i); a[f(i)]_1, a[k(i)]_0))$
$\longleftarrow a[k(i)]_1$
endloop

Let us closely analyze δ and γ in this example. δ functions can be considered to represent the loop carried dependence. If the loop i is executed in parallel, there must be synchronization codes between loops at the beginning of the loop. Four δ nodes in the example represent this kind of synchronization, because two definitions ($a[f(i)]_1$ and $a[h(i)]_1$) of previous iterations are catched there. On the other hand, the γ function can be considered to represent the loop-independent dependence.

Furthermore, by using γ, we can represent dependence over loops. From the view of definitions, the loops can be summarized as a collection of definitions. Let us consider a simple example:

loop i
loop j
$a[f(i,j)] = \cdots$
$\cdots$
endloop
$\cdots = a[g(i)]$
endloop

The effect of **loop** j is a collection of definitions of the form $a[f(i,j)]$. Therefore, when referring to $a[g(i)]$, its whole effect must be considered. This can be reflected as inserting the code just after the loop:

$$a[g(i)] = \gamma(f(i,j) = g(i); a[f(i,j)], a[g(i)]),$$

which means by definition: if $\exists j.f(i,j) = g(i)$, then $a[f(i,j)]$ such that $f(i,j) = g(i)$, else $a[g(i)]$.

We list the result of the translation.

loop i
$a[g(i)]_0 = \delta((a[g(i)]_{init}, \bot); (a[f(i,j)]_0, f); (g(i), 1))$
loop j
$a[f(i,j)]_0 = \delta((a[g(i)]_0, g); (a[f(i,j)]_1, f); (f(i,j), 1))$
$a[f(i,j)]_1 = \cdots$
endloop
$a[g(i)]_1 = \gamma(f(i,j) = g(i); a[f(i,j)]_1, a[g(i)]_0)$
$\cdots = a[g(i)]_1$
endloop

Let us review how problems of considering "array as a collection of individual variables" are solved:

1. An indexed variable is introduced to represent the "collection of variables." It is given a symbolic manipulation with δ and γ.
2. the loop carried dependence and other dependence are represented as a certain kind of equations in δ and γ respectively. The property that a location where an equation must be solved is also specified gives a major difference from conventional data dependence analysis.

3 Optimizations of δ

Because δ is closely related to synchronization, its optimization can be considered as a synchronization optimization. Optimizations of δ include conventional SSA-based scalar optimizations. A major difference to optimizations for scalar SSA is that the def-use chain is further optimized by using index calculation.

3.1 Eliminating Backedges and Synchronization Optimizations

If we can prove that in an assignment of δ, the indices of *lhs* and backedges in δ do not overlap by using data dependence analysis, the backedge part can be eliminated from the δ. The complication is that the backedge does not disappear. Instead, a backedge must be hoisted to where the *lhs* in the assignment is used.

Example 2 Let us consider the matrix multiplication (A) and its translation to an arraySSA form (B) in Figure 1.

Consider the array variable $c(i,j)_{134589064}$ and the innermost loop. Because the innermost assinment of δ has no overlap with the backedge given as $c(i,j)_{134589064}$, it can be eliminated. Instead, the assignment $c(i,j)_{134588588} = \delta(c(i,j)_{134587296};;(i,j,1))$ is produced. Moreover, $c(i,j)_{134589064}$ is hoisted to the use of $c(i,j)_{134588588}$, and its use is replaced with $c(i,j)_{134589064}$. If the backedges disappear, there is no reason of using δ. Therefore, the wrapper is also removed. Note that for the outermost δ, we can apply the same optimization. The resulting program is (C). Furthermore, we can coalese simple assignments. In other words, we can replace the variable-to-variable assignment with the replacement of the use of *lhs* by *rhs*. Finally, we obtain the program (D).

Because backedges have the role of synchronization, hoisting of backedges means a kind of synchronization optimization. Given a δ-term $\delta((v_0, acc_{v_0}), \cdots, (v_n, acc_{v_n}); (w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m}); lnv)$ at a location ℓ, we first check whether $acc_{v_i}^{-1}(lnv)$ has a solution. This can be done with conventional data dependence analysis. In this sense, information on data dependence is embedded in δ-pseudofunctions. If there is no solution, the term (v_i, acc_{v_i}) can be eliminated, and there is no "backedge" or loop-carried dependence caused by v_i.

Let us assume that there is no loop-carried dependence. The δ-term can be optimized to a form of $\delta((w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m});;lnv)$. In this case, there is no need to have synchronization with previous iterations. Therefore, it deserves definition:

Definition 8 We denote a δ-pseudofunction of the form $\delta((w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m});;lnv)$ by *p-δ* form.

A p-δ form is not an obstruct for parallelization because there is no loop-carried dependence.

Theorem 1 If all the δ-pseudofunctions in a loop are in the p-δ form, the loop can be executed in parallel.

This optimization is applied to (D) resulting in (E) of Figure 1. **For** loops are transformed to **forall** loops.

Furthermore, if a solution among equations on w_j's is unique, then the δ-term itself can be substituted for the solution. Let us assume that the solution is (w_j, acc_{w_j}). Then, the p-δ form can be eliminated, and be converted to $a[acc_{w_j}^{-1}(lnv)]$, where a is the name part of w_j. If $m > 0$, it can be translated simply to $\gamma((w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m}))$. Thus inter-iteration synchronization can also be eliminated.

3.2 Loop Restructuring and δ

Loop restructuring[11] is a powerful optimization technique to maximize parallelism. It is natural that associated with optimizations of δ are loop restructuring optimizations because the restructuring affects the dependence.

Let us consider a δ-pseudofunction $\delta((v_0, acc_{v_0}), \cdots, (v_n, acc_{v_n}); (w_0, acc_{w_0}), \cdots, (w_m, acc_{w_m}); lnv)$, and let us assume that a loop restructuring $\vec{i} \mapsto \vec{i'}$ is given by a unimodular matrix P such that $P\vec{i'} = \vec{i}$, where $\vec{i}$ and $\vec{i'}$ are loop indices.

Because acc_{v_i}'s are given as functions of loop indices, we have $acc_{v_i} \mapsto acc_{v_i}P$. To summarize, we have $\delta((v_0, acc_{v_0}P), \cdots, (v_n, acc_{v_n}P); (w_0, acc_{w_0}P), \cdots, (w_m, acc_{w_m}P); lnv)$, as the loop restructuring transformation.

Let $lnv = F\vec{i}$ for some F. Then $lnv = FP\vec{i'}$. The original solution vector is given as $acc_{v_i}^{-1}(F\vec{i})$. On the other hand, when loop restructuring is applied, the solution vector is

$$P^{-1}acc_{v_i}^{-1}(lnv) = P^{-1}(acc_{v_i}^{-1}F)P\vec{i'},$$

which is conjugate to the original solution. The restriction that P must be unimodular in the case of loop restructuring guarantees the existence of P^{-1}.

To summarize, let us assume that P is given as a unimodular matrix for representing a loop restructuring optimization. Then

Theorem 2 A loop restructuring optimization gives change of δ-pseudofunctions to
$\delta((v_0, acc_{v_0}P), \cdots, (v_n, acc_{v_n}P);$
$(w_0, acc_{w_0}P), \cdots, (w_m, acc_{w_m}P));\quad lnv),$ with the condition

$$P^{-1}(acc_{v_i}^{-1}F)P\vec{i'} < \vec{i'}.$$

4 Related Work

SSA[2] has been proved suitable for analyzing scalar values. A number of optimizations are described by using SSA because of its efficiency in handling scalars. Its efficiency is proved both theoretically and practically[12, 7].

There are proposed several extensions of SSA to handle arrays. In [8], region arraySSA, a method to make accessed region of arrays as strict as possible is proposed. In [5], array SSA has been proposed. However, it is still in the "array as a monolithic object" scheme. Gated SSA(GSA)[10] adopts symbolic computation in improving the accuracy of analysis. LastWriteTree(LWT) is proposed in [3], and used in the optimization of high performance compilers[6]. Although LWT has much in common with SSA, it is not intended to be embedded in SSA. [1] is another approach that replaces arrays with scalars.

As previous works sharing the objective with ours, we can list array SSA[5], LWT[6] and region arraySSA[8]. We show the comparison of functions between our arraySSA and them in Table 1. Using our arraySSA, because arrays are handled as scalars (indexed variable), and dataflow on arrays is embedded in our arraySSA, both scalar optimizations (copy propagation etc.) and array optimizations (synchronization optimization, parallelism extraction etc.) can be naturally expressed in our scheme.

5 Conclusion

In this paper, we have proposed an extension of SSA with δ and γ pseudofunctions. This is a solution of the problem of integration of scalar analysis and array data dependence analysis

Moreover, we have studied optimizations by using our framework. Optimizations of δ's have been shown to be synchronization optimizations. Relations to loop restructuring optimizations have been also discussed. Finally, we have discussed the expressive power of parallelism extraction and scalar optimization extension, as a comparison with previous results, and have shown that our arraySSA is very powerful.

References

[1] Callahan, D., Carr, S., Kennedy, K.: "Improving Register Allocation for Subscripted Variables," *Proc. 1990 Programming Language Design and Implementations*, 1990, 53–65.

[2] Cytron, R., Ferrante, J., Sarkar, V.: "Compact Representation for Control Dependence," *Proc. 1990 Programming Languages Design and Implementation*, 1990, 337–351.

[3] Feautier, P.: "DataFlow Analysis of Array and Scalar References," *Int'l J. Parallel Programming*, 20(1), 1991, 23–54.

[4] Kim, C., Gaudiot, J., Proskurowski, W.: "Parallel Computing with the Sisal Applicative Language: Programmability and Performance Issues," *Software Practice & Experience*, 26(9), 1996, 1025–1051.

[5] Knobe, K., Sarkar, V.: "Array SSA form and its use in Parallelization," *Proc. 1998 Principles of Programming Languages*, 1998, 107–120,

[6] Mayden et.al.: "Array Data-Flow Analysis and Its Use in Array Privatization," *Proc. 1993 Principles of Programming Languages*, 1993, 2–15.

[7] O'Brien, K. et.al.: "XIL and YIL: The Intermediate Languages of TOBEY," *Proc. Workshop of Intermediate Representation'95*, 1995, 71–82.

[8] Rus, S., He, G., Alias, C., Rauchwerger, L.: "Region Array SSA," *Proc. Parallel Architectures and Compiler Techniques 2006*, 2006, 43–52.

[9] Skedzielewski, S.: SISAL, in Szymanski, B. (Ed.) *Parallel Functional Languages and Compilers*, 1991, 105–157.

[10] Tu, P., Padua, D.: "Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers," *Proc. 1995 Int'l Conf. Supercomputing*, 1995, 414–423.

[11] Wolf, M., Lam, M.: "A Loop Transformation Theory and Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Systems*, 2(4), 1991, 452–471.

[12] Wolfe, M.: "Beyond Induction Variables," *Proc. 1992 Programming Languages Design and Implementation*, 1992, 162–174.

[13] Wolfe, M.: *High Performance Compilers for Parallel Computing*, 1995.

(A)

for $j = 1, n$
  for $k = 1, n$
    for $i = 1, n$
      $c(i,j) = c(i,j) + a(i,k) * b(k,j)$
    endfor
  endfor
endfor

(B)

for $j = 1, n_\perp$
  $c(i,j)_{134586004} = \delta(c(i,j)_\perp; c(i,j)_{134587296}; (i,j,1))$
  for $k = 1, n_\perp$
    $c(i,j)_{134587296} = \delta(c(i,j)_{134586004}; c(i,j)_{134588588}; (i,j,1))$
    for $i = 1, n_\perp$
      $c(i,j)_{134588588} = \delta(c(i,j)_{134587296}; c(i,j)_{134589064}; (i,j,1))$
      $c(i,j)_{134589064} = c(i,j)_{134588588} + a(i,k)_\perp * b(k,j)_\perp$
    endfor
  endfor
endfor

(C)

for $j = 1, n_\perp$
  $c(i,j)_{134586004} = c(i,j)_\perp$
  for $k = 1, n_\perp$
    $c(i,j)_{134587296} = \delta(c(i,j)_{134586004}; c(i,j)_{134589064}; (i,j,1))$
    for $i = 1, n_\perp$
      $c(i,j)_{134588588} = c(i,j)_{134587296}$
      $c(i,j)_{134589064} = c(i,j)_{134588588} + a(i,k)_\perp * b(k,j)_\perp$
    endfor
  endfor
endfor

(D)

for $j = 1, n_\perp$
  for $k = 1, n_\perp$
    $c(i,j)_{134587296} = \delta(c(i,j)_\perp; c(i,j)_{134589064}; (i,j,1))$
    for $i = 1, n_\perp$
      $c(i,j)_{134589064} = c(i,j)_{134587296} + a(i,k)_\perp * b(k,j)_\perp$
    endfor
  endfor
endfor

(E)

forall $j = 1, n_\perp$
  for $k = 1, n_\perp$
    $c(i,j)_{134587296} = \delta(c(i,j)_\perp; c(i,j)_{134589064}; (i,j,1))$
    forall $i = 1, n_\perp$
      $c(i,j)_{134589064} = c(i,j)_{134587296} + a(i,k)_\perp * b(k,j)_\perp$
    endfor
  endfor
endfor

Figure 1. An example of δ-Optimization by using arraySSA

Table 1. Comparison of ArraySSA Solutions

	Naive	LWT[6]	ArraySSA[5]	Region arraySSA[8]	Our ArraySSA
Covering	*o*	*o*	*o*	*o*	*o*
Parallelism and related Opts.					
Parallelism Extraction		*o*	*o**	*o*	*o*
Loop Restructuring Repr'n					*o*
Synchroniz'n Code Opt.					*o*
Scalar Opts.					
Dead Code Elim'n, Const. Propagat'n etc.				*o*	*o*
Scalar Replacement				*o*	*o*

*) requires independent dependence analysis.