Open access • Journal Article • DOI:10.1145/1278349.1278353

A practical dynamic single assignment transformation — Source link

Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Francky Catthoor

Institutions: Katholieke Universiteit Leuven

Published on: 01 Sep 2007 - ACM Transactions on Design Automation of Electronic Systems (ACM)

Topics: Data-flow analysis, Data flow diagram and Pointer (computer programming)

A Practical Dynamic Single Assignment Transformation

PETER VANBROEKHOVEN, GERDA JANSSENS and MAURICE BRUYNOOGHE
Department of Computer Science, K.U.Leuven, Belgium
and
FRANCKY CATTHOOR
Interuniversity Micro-Electronics Center, Belgium

This paper presents a novel method to construct a dynamic single assignment (DSA) form of array intensive, pointer free C programs. A program in DSA form does not perform any destructive update of scalars and array elements, *i.e.*, each element is written at most once. As DSA makes the dependencies between variable references explicit, it facilitates complex analyses and optimizations of programs. Existing transformations into DSA perform a complex data flow analysis with exponential analysis time, and they work only for a limited class of input programs. Our method removes irregularities from the data flow by adding copy assignments to the program, so that it can use simple data flow analyses. The presented DSA transformation scales very well with growing program sizes and overcomes a number of important limitations of existing methods. We have implemented the method and it is being used in the context of memory optimization and verification of those optimizations. Experiments show that in practice, the method scales well indeed, and that added copy operations can be removed in case they are unwanted.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*optimization; compilers*

General Terms: Design, Performance

Additional Key Words and Phrases: Data flow analysis, single assignment, arrays, parallelization, reaching definitions

1. INTRODUCTION

In a program in dynamic single assignment (DSA) form, *during execution* there is only a *single* assignment to each *array element*. To the best of our knowledge, DSA was first used in the form of a system of recurrence equations, *e.g.,* [Karp et al. 1967]. In the form that we present here, it was first used in the context of parallelization [Feautrier 1988a]. Figure 2 shows the DSA form of the program in

Supported by a specialization grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)
A preliminary version of this paper appeared in Proceedings of The Third Asian Symposium on Programming Languages and Systems, 2005.
Author's address: P. Vanbroekhoven, Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Heverlee (Leuven), Belgium.

```
    for (j = 0; j < 10; j++)
      for (i = 0; i < 10; i++) {
S1:     output(a[j]);
S2:     a[j] = input();
      }
```

Fig. 1. Simple example program.

```
    for (j = 0; j < 10; j++)
      for (i = 0; i < 10; i++) {
S1:     output(i == 0 ?  a[j] :  a1[i - 1][j]);
S2:     a1[i][j] = input();
      }
```

Fig. 2. DSA form of Figure 1.

Figure 1. These figures show a few typical features of the DSA form. The first is the renaming of variables (typically by adding an index) to make sure different statements write to different variables. The second is the addition of extra array dimensions to variables to make each iteration of a statement write to a different array element. The third is the introduction of selection conditions at reads from variables to match the changes at the assignments.

The only limitation on reordering execution in DSA form is that an array element cannot be read before it is assigned. Any constraints resulting from multiple values being stored in array elements are eliminated. Besides the increased possibility for reordering execution, finding the constraints on reordering boils down to finding the *use-def* pairs: which iteration of which statement wrote (*def*) the value being read (*use*) by a given iteration of a statement. This is easily done by finding the assignments to the variable and matching the indexation; only one iteration of a single statement can match because of the DSA property. In Figure 2 we see that statement `S1` reads the initial value of `a` when `i` is 0, and reads the value written by `S2` in the previous iteration otherwise. Additionally, we can do transformations on Figure 2 that are not possible on Figure 1, *e.g.,* swap `S1` and `S2`.

DSA form plays an important role in the design of embedded systems, particularly for low power. It enables global loop transformations that reduce power consumption [Catthoor et al. 1998; Benini and Micheli 2000; Panda et al. 2001] as well as the verification of such transformations [Shashidhar et al. 2003; 2005]. Parallelization [Li 1992; Feautrier 1988a] can automatically spread the CPU load over multiple slower, less power-hungry CPUs. The same idea is used in [Kienhuis et al. 2000] using software and hardware components. Some of these hardware components are automatically synthesized from systolic arrays [Quinton 1984].

All of the techniques mentioned have one thing in common: they exploit the possibility to reorder the execution of the program.

The possibility for reordering needs to be detectable from the program as easily and accurately as possible, since fewer constraints on reordering – called dependencies – lead to better results. Also important are transformations that reduce the number of reordering constraints. Transforming the program into DSA form increases the possibility for reordering and makes it easily detectable.

However, existing methods for transformation to DSA form are too slow and consume too much memory for our purposes. This is especially a problem because global optimizations inline functions to optimize over function boundaries. Indeed, aggressive optimizations usually require specializing each call-site to obtain the best results. Another problem is that they disallow conditions that depend on input data, which in practice turns out to be a severe restriction. These data dependent conditions are hard to analyze, but the analysis is easier on the DSA form, so it is important to be able to transform these programs to DSA as well. Applications like

```
for (i = 0; i < 10; i++)
  for (j = i; j < 21 - i; i++)
    if (j != i + 6)
      a[j] = f(i, j);
for (k = 0; k < 21; k++)
  output(a[k]);
```

Fig. 3. Another example.

```
for (i = 0; i < 10; i++)
  for (j = i; j < 21 - i; i++)
    if (j != i + 6)
      a1[i][j] = f(i, j);
for (k = 0; k < 21; k++)
  output(k <= 9 ?  a1[k][k] :
      (k <= 11 ?  a1[9][k] :
      (k >= 20 ?  a[k] :
      (k == 13 ?  a1[6][k] :
      a1[20 - k][k])))));
```

Fig. 4. DSA form of Figure 3.

```
for (i = 0; i < 10; i++)
  for (j = 0; j < 25; i++)
    if (j != i+6 && j >= i && j < 21-i)
      a[j] = f(i, j);
    else a[j] = a[j];
for (k = 0; k < 21; k++)
  output(a[k]);
```

Fig. 5. Figure 3 with an extra copy operation.

```
for (i = 0; i < 10; i++)
  for (j = 0; j < 25; i++)
    if (j != i+6 && j >= i && j < 21-i)
      a1[i][j] = f(i, j);
    else a1[i][j] = j==0 ?  a[j]
                         :  a1[i-1][j];
for (k = 0; k < 21; k++)
  output(a1[9][k]);
```

Fig. 6. DSA form of Figure 5.

functional verification of global transformations [Shashidhar et al. 2003; 2005] can handle certain classes of data dependent conditions, but up to now there has been no general way of transforming programs with data dependent conditions to DSA form. This is a requirement for functional verification. Finally, transformation to DSA form is tedious and error-prone when done manually, and hence automation is needed.

The *contribution* of this paper is a new, automated method for transformation to DSA that offers two advantages over existing work:

(1) It overcomes a number of limitations on input programs of existing methods. The sole restriction is that constant bounds must be derivable for the loop iterators.

(2) It is quadratic in the program size and polynomial in the depth of the loop nests, while existing methods are exponential.

Moreover our implementation confirms the scalability of the method and has been successfully used in the context of memory optimizations [Catthoor et al. 1998] and verification of those transformations [Shashidhar et al. 2003; 2005].

The complexity of the transformation to DSA form is caused by complex access patterns to the arrays. It is obvious that a program that writes an array row by row is easier to analyze than a program that writes array elements at random. It is however possible to change the access pattern without changing the result of the program, namely by adding no-op copy operations that copy a variable to itself. Figures 3-6 show how addition of copy operations can indeed simplify the DSA form noticeably.

Section 2 lists related work while Section 3 introduces our approach. Section 4 explains the preparatory steps and Section 5 describes how to perform the DSA transformation while Section 6 discusses some additional steps. Section 7 gives the complexity analysis and Section 8 presents results of applying our implementation to a number of multimedia kernels. Section 9 analyzes the overhead caused by our DSA transformation and how this overhead can be removed. Section 10 concludes.

2. RELATED WORK

Languages that impose DSA form, such as Silage [Genin and Hilfinger 1989] and Single Assignment C [Scholz 1994], are not widely used because DSA form can be awkward to write. Most applications are written in a multiple assignment[1] form.

There exist many other kinds of single assignment forms. Optimizing compilers use Static Single Assignment (SSA) form [Cytron et al. 1991] to encode data flow information. A program in SSA form has only a single assignment to a given variable *in the program text*. This means that SSA form does not differentiate between the different instances of a statement, unlike DSA form. Because of this, SSA form is less suited for array-intensive programs –although [Cytron et al. 1991] does propose a way to handle arrays. A better way to handle arrays is presented in [Knobe and Sarkar 1998], but it still concerns a kind of SSA form: Array SSA or ASSA form. Arrays are accurately handled by a full array data flow analysis and the corresponding DSA form of [Feautrier 1988a].

The method of [Feautrier 1988a] allows only affine loop bounds, conditions and indexation; [Kienhuis 2000] relaxes this by allowing piecewise affine expressions too.

Further extensions to the data flow analysis are provided by [Collard et al. 1997]. However, they lack the exactness of [Feautrier 1988a], and hence are unsuitable for building the DSA form. An alternative to DSA form is a System of Recurrence Equations (SRE), which basically is DSA form without a specific execution order. Methods to translate to SREs, *e.g.,* [Bu and Deprettere 1988], lack the generality of [Feautrier 1988a] and [Kienhuis 2000]. However, the approach of [Feautrier 1988a] can be easily adapted to generate SREs, as done by [Alias 2003].

Finally, we note that besides the simple categorization of SSA - ASSA - DSA, there are other variations in between such as [Ballance et al. 1990] and [Offner and Knobe 2003]. For more details on the difference between these single assignment forms, we refer to [Vanbroekhoven et al. 2005].

3. OUR APPROACH

The most significant difficulty in obtaining a scalable and generally applicable DSA transformation is that it requires an *exact* data flow analysis. Such an analysis is presented in [Feautrier 1988a], but it is not generally applicable and scales badly – both in execution time and memory use. Data flow analyses as presented in [Aho et al. 1986] give approximate answers when control flow paths join, or when an array is accessed; this allows them to be fast and generally applicable. The observation underlying our approach is that we can do away with approximations without sacrificing speed when *all variables are scalars, there are no* `if`*-statements, and every assignment is executed at least once in every iteration of every surrounding loop.* In case the program does not have these properties, we apply simple transformations, basically adding copy statements, until it does.

[1]As opposed to (dynamic) single assignment.

```
   for (t = 0; t < 1009; t++) {
S1:  c[t] = 0;
     for (i = 0; i <= t; i++)
       if (i > t - 1000 && i < 10)
S2:      c[t] = c[t]+a[i]*b[t-i];
   }
   for (t = 0; t < 1009; t++)
S3:  output_c(c[t]);
```

Fig. 7. Our running example.

```
transformToDSA(program)
    1. make arrays scalar (Section 4.1)
        for each right hand side array reference $a$ :
            replace $a$ with an Ac operation
        for each assignment $a$ :
            replace the assignment by an Up operation
    2. pad with copy operations (Section 4.2)
        find constant loop bounds & add no-op copy operations to the program
    3. do exact scalar data flow analysis (Section 5.1)
        for each variable reference:
          determine what assignment instance wrote the value read, and under what condition
    4. transformation to DSA (Section 5.2)
        transform to DSA using the result of the data flow analysis
    5. expand Up and Ac (Section 6)
        replace each Up and Ac by its definition
```

Fig. 8. High-level script of DSA transformation.

The programs that our method can handle, are as follows:

—Arbitrarily nested `if`-statements and `for`-loops with arbitrary upper and lower bounds, provided that a constant lower and upper bound can be found for each loop. Note that the conditions of `if`-statements can be taken into account for finding these bounds.

—Within those loops, assignments can be present.

—Each expression in the left or right hand side of an assignment, in the array indexation, or in conditions, can contain array references, basic operators like `+` and `*`, the ternary `?:` operator from C, and calls to unknown, purely functional, scalar functions, *i.e.,* functions with one or more scalar arguments and one scalar return value which depends solely on the arguments.

The programs resulting from our DSA transformation have the same properties; in addition they exhibit the DSA property. In our specific case, this means that each assignment is to a different array, and this array is indexed using the iterators of the surrounding loops.

Our running example is discrete convolution (Figure 7). The code assumes two signals are given. The first signal consists of 10 samples that are stored in array `a`. The second signal consists of `1000` samples that are stored in array `b`. The resulting 1009 samples are stored in array `c`.

The DSA transformation consists of several steps that are described in subsequent sections. The general overview of the DSA transformation is shown in Figure 8,

```
   for (t = 0; t < 1009; t++) {
S1:  c = Up(c, t, 0);
     for (i = 0; i <= t; i++)
       if (i > t-1000 && i < 10)
S2:      c = Up(c,t,Ac(c,t)+a[i]*b[t-i]);
   }
   for (t = 0; t < 1009; t++)
S3:  output_c(Ac(c, t));
```

Fig. 9. Figure 7 with Up and Ac.

```
for (it = 0; it < 1009; it++)
  if (it == i)
    v1[it] = e;
  else
    v1[it] = v2[it];
```

Fig. 10. Meaning v_1 = Up(v_2,i,e).

with references to the corresponding sections.

4. PREPARATORY STEPS

4.1 From array variables to scalars

Array indexation complicates the DSA transformation; therefore, we start by hiding it. The main difference between scalars and arrays is that scalars are assigned as a whole while arrays are typically assigned element per element. One way to handle this, as suggested in [Cytron et al. 1991], is to introduce an Up operation for each assignment to an array element that builds a whole new array with all elements equal to the corresponding elements of the old array except for one element that gets a new value. The result for our running example is shown in Figure 9. The Ac functions are technically not necessary as they can be trivially replaced by direct array references, but they do help encapsulating the use of arrays. The meaning of an Up function is shown in Figure 10. If the loops in the Up operation are unrolled, we end up with the original assignment together with a large number of no-op copy assignments. Hence, the introduction of the Up functions does not change the functionality of the program.

4.2 Execute every assignment in every iteration

In the actual DSA transformation, for each read, we need to find the last statement instance that wrote to the array element being read. This search is complex, and to simplify it we make sure every assignment is executed in each iteration of each loop.

The two causes preventing the execution of an assignment are if-statements and loops that have zero iterations (possibly depending on the values of surrounding iterators). The simplest way to overcome these problems is to transform away if statements and replace all loop bounds by constants. If these constant upper and lower bounds on iterator values indicate that the body of the loop is never executed, we remove it as the loop was useless to begin with.

For Figure 9 the maximum value for the i iterator is 9. We can use this as upper bound on the i loop provided we move the condition that i should be smaller than t to the if-statement. The first step in removing the condition is to add an else-branch and a no-op assignment for each variable assigned in the other branch. For our running example, this results in Figure 11. Note that c is always assigned a value now, regardless of the value of the condition of the if-statement. This can be made explicit in the C language as shown in Figure 12. Now, it is trivial to see that when arriving at S3, the last assignment to c is done by S2 for t equal to 1008

```
  for (t = 0; t < 1009; t++) {
S1: c = Up(c, t, 0);
    for (i = 0; i < 10; i++)
      if (i > t-1000 && i <= t)
S2:     c = Up(c,t,Ac(c,t)+a[i]*b[t-i]);
      else c = c;
  }
  for (t = 0; t < 1009; t++)
S3: output_c(Ac(c, t));
```

Fig. 11. Figure 9 padded with copy operations.

```
  for (t = 0; t < 1009; t++) {
S1: c = Up(c, t, 0);
    for (i = 0; i < 10; i++)
S2:   c = i > t-1000 && i <= t ?
        Up(c,t,Ac(c, t)+a[i]*b[t-i]):
      c;
  }
  for (t = 0; t < 1009; t++)
S3: output_c(Ac(c, t));
```

Fig. 12. `S2` assigns `c` in each iteration.

and `i` equal to 9. As we have only added more no-op copy assignments, we have changed the access pattern of the program while leaving the functionality of the program unchanged.

5. TRANSFORMATION TO DSA FORM

5.1 Reaching definitions analysis

In the following two steps, all array references will be changed to attain DSA form. First the left hand sides are transformed, then the right hand sides are changed accordingly using information about which definitions can reach each use. This information is obtained by running a reaching definitions analysis [Aho et al. 1986]. Because of our preparatory transformations, we can perform a fast and exact analysis. Remember that we transformed all arrays to the equivalent of scalars, and that we made sure every assignment is executed at least once in each iteration of each surrounding loop.

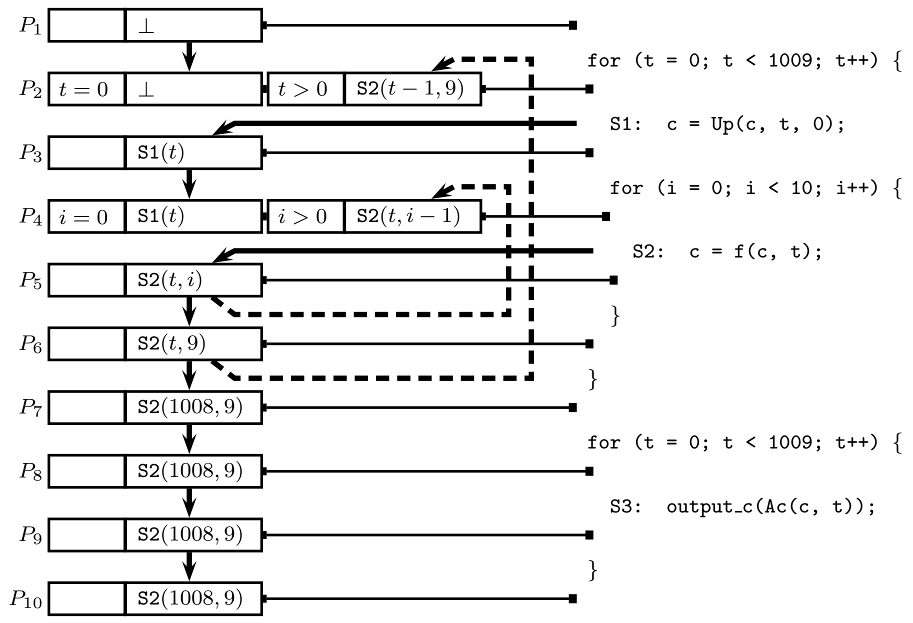
The analysis can be performed for each variable separately because variables cannot interfere with the data flow of other variables. For our running example, the analysis for `c` is depicted graphically in Figure 13.

We define program points P_1 through P_{10}, one at the start of the program, one after each assignment, one after the head of each loop, and one after each loop. This means that each statement or loop has a program point preceding it, namely the one after the preceding statement or loop. At each program point, we keep the reaching definition instances and the conditions under which they reach. We denote the instances of a statement by specifying the statement and the value of the iterators at the moment when that instance is executed. For example, the instance of `S2` in Figure 11 when `t`$= x$ and `i`$= y$ is denoted `S2`(x, y). Here x and y do not need to be constants. For example, the assignment instance preceding `S2`(x, y) is `S2`$(x, y-1)$, provided that $y-1$ is within `i`'s range.

The number of reaching definitions to distinguish, and the conditions for each, are determined unambiguously by the rules below.

—At the start of the program (P_1), we distinguish one case and denote the reaching instance with $\perp$. This can be interpreted as either an uninitialized variable, or the initial value of the variable wherever it may come from.

—Upon entering a loop that contains an assignment to the variable we are analyzing (for example P_2), we split cases. Either we are in the first iteration of the loop (`t` $= 0$), in which case we distinguish the same reaching definitions as just before



```
for (t = 0; t < 1009; t++) {
  S1:  c = Up(c, t, 0);
  for (i = 0; i < 10; i++) {
    S2:  c = f(c, t);
  }
}
for (t = 0; t < 1009; t++) {
  S3:  output_c(Ac(c, t));
}
```

Fig. 13. Exact reaching definitions analysis on Figure 12.

the loop, or we are in a later iteration ($\texttt{t} > 0$), in which case the assignment in the loop has been executed at least once and has killed all reaching definitions from before the loop. In the latter case, the reaching definition comes from the last program point in the loop. In case of P_2, the end of the loop is P_6, and for $\texttt{t} > 0$, the reaching definition there is found to be S2$(t, 9)$. Taking into account that we are in the next iteration of the t loop, the definition reaching P_2 becomes S2$(t-1, 9)$.

—Upon entering a loop that contains no assignment to the variable being analyzed (*e.g.,* P_8), the reaching definitions are obviously not affected and hence, they are just copied from before the loop.

—Just after a loop (*e.g.,* P_6), the reaching definitions are those that reach the end of the last iteration of that loop. These are the definitions that reach the start of the loop in case there is no assignment in the loop, or that reach the last program point inside a previous iteration of the loop in case there is not. The only definition reaching P_6 is that at P_5 for $\texttt{i} = 9$; i.e., S2$(t, 9)$.

—Immediately after an assignment (*e.g.,* P_5) all other reaching definitions are killed. Thus the assignment is the only reaching definition, and the reaching instance is obviously the one from the current iteration of the loop. For P_5 this is S2(t, i).

—Immediately after any other statement (including assignment to other variables than the one er are analyzing,) the reaching definitions are the same as before the statement.

Each case in a data flow problem as in Figure 13 can be filled out in two passes; the back references are filled out in the second pass.

The rules above clearly deal with all the types of program points we have intro-

```
  for (t = 0; t < 1009; t++) {
S1: if (t == 0) c1[t] = Up(c, t, 0);
      else c1[t] = Up(c2[t-1][9], t, 0);
    for (i = 0; i < 10; i++)
S2:   if (i == 0) c2[t][i] = i > t-1000 && i <= t ?
                           Up(c1[t],t,Ac(c1[t],t)+a[i]*b[t-i]) :  c1[t];
      else c2[t][i] = i > t-1000 && i <= t ?
                Up(c2[t][i-1],t,Ac(c2[t][i-1],t)+a[i]*b[t-i]) :  c2[t][i-1];
  }
  for (t = 0; t < 1009; t++)
S3: output_c(Ac(c2[1008][9], t));
```

Fig. 14. Converting Figure 13 to full DSA form.

duced. As with any data flow analysis, the information at each program point is calculated from the program points that precede it in the execution. If there are several predecing program points, the information needs to be merged. A distinct difference with standard data flow analyses is that, because of our preparatory transformations, the merge is exact; we do not lose information because we know exactly under what condition the information of each preceding program point applies.

5.2 Transformation to DSA form

In multiple assignment code involving only scalars, there are two causes of multiple assignments to a variable; either there are two assignments to the same variable, or an assignment is in a loop. The former cause is removed by renaming the variables in the left hand side of each assignment such that they each write to a different variable. Typically this is done by adding a sequence number at the end of the variable. The latter cause is removed by changing the variable assigned to an array with one dimension for each surrounding loop, and we index that dimension with the iterator of the corresponding loop. For our running example this is shown in Figure 14.

After adjusting the left hand sides, we still need to adjust the variable accesses in the right hand sides. It is possible that a number of assignments (definitions) reach a variable access; which one needs to be read from depends on the values of the iterators. This information was derived by applying the reaching definitions analysis previously described. To achieve a correct transformation, it is necessary and sufficient that the reaching definition instances stay the same. Thus, for each reaching definition instance and accompanying condition, we need to read the new array element assigned by that reaching definition, but only under the given condition. This ensures that the reaching definition instances still write the same array element. Because of the DSA property just introduced, there is no other definition instance that writes to the same array element, and it is guaranteed that the reaching definition instance remains reaching.

For our example, S1(t) is reached by $\perp$ when $t = 0$. In other words when S1 is executed for t equal to 0, we should read from the variable written by the implicit assignment at the start of the program, which is just c. If $t > 0$, S1(t) is reached by S2$(t-1, 9)$. So we should read from the variable written by S2$(\mathtt{t} - 1, 9)$, which is c2[t-1][9]. Because of the special form of the left hand sides, the correspondence

```
   for (t = 0; t < 1009; t++) {
     for (a = 0; a < 1009; a++) {
S1:  if (a == t)
       c1[t][a] = 0;
     else
       c1[t][a] = (t == 0 ?  c :  c2[t-1][9][a]);
     }
     for (i = 0; i < 10; i++)
       for (a = 0; a < 1009; a++) {
S2:      if (i > t-1000 && i <= t && a == t)
           c2[t][i][a] = (i == 0 ?  c1[t][t] :  c2[t][i-1][t])+a[i]*b[t-i];
         else
           c2[t][i][a] = (i == 0 ?  c1[t][t] :  c2[t][i-1][t]);
       }
   }
   for (t = 0; t < 1009; t++)
S3:  output_c(c2[1008][9][t]);
```

Fig. 15. Full expansion Figure 14.

between S2$(t-1,9)$ and `c2[t-1][9]` is direct. Putting it all together results in Figure 14.

6. ADDITIONAL STEPS

At the end, we need to *replace the* `Up` *and* `Ac` *functions* by their respective implementations. This is a straightforward replacement of these two functions by their definition. In our transformation, we can create redundant conditions, or conditions that cannot be satisfied. This happens often enough to justify the effort of detecting these cases. To this end, we use the Omega library [Pugh 1991]. For our running example, the expansion results in Figure 15. In this code, the copy operations were regrouped in a single else branch. In fact, the separation of the copy operations in the else branch of statement S2 in Figure 12 and those in the Up operation in the same statement are an artifact of our explicit introduction of the Up operation.

We have described our DSA transformation for the case where all loop bounds, indexation and conditions are affine functions of surrounding iterators. This fact is only used in Section 4 to find a constant upper and lower bound for the iterators; the other steps do not analyze the indexation and conditions.

In general, conditions can be data dependent or non-affine. We can still use linear programming to find bounds if we drop the data dependent conditions. This can create domains without bounds on some iterators, but this appears to be rare in practice. In our experiments, this has never occurred. If it happens though, we can use the data type of the iterators to find reasonable bounds, or the model could be extended to handle infinite domains. The latter is not a problem as DSA form is meant to be an intermediate representation and not necessarily executable code.

7. COMPLEXITY ANALYSIS

The complexity analysis developed in this section is illustrated on array `c`. Features of programs used in the complexity analysis, along with the symbols used to denote

n	maximum loop nest depth	2
d	maximum dimension of an array	1
a	number of assignments	2
l	number of loops	3
r	number of variable references	4
s	number of statements	7
v	number of variables	1
c	maximum number of geometrical constraints on a statement	6

Fig. 16. Program characteristics with the values for array c.

them, are shown in Figure 16. The column on the right contains the corresponding values for array `c` of our example program.

The most time-consuming steps in the transformation script (Figure 8) are the padding with copy operations (step 2), the data flow analysis (step 3) and the expansion of `Up` and `Ac` (step 5). Step 2 solves a linear programming problem involving $O(c)$ conditions for each of the l loops, and adds a copy operation to each of the a assignments, giving a complexity of $O(p(c)\cdot l+a)$ with $p(c)$ a polynomial in c. Step 3 requires going over each of the s statements in the program twice (the second time to propagate information back over the loops). The information we propagate consists at maximum of $n+1$ different cases for each of the v variables, and each case has an array reference of size d. Thus, step 3 requires a time that is $O(s\cdot n\cdot d\cdot v)$. Step 5 creates $n+d$ dimensional array references for each of $a+r$ references in the program, giving a complexity of $O((n+d)\cdot(a+r))$. In total the complexity is the sum of these three, resulting in $O(p(c)\cdot l+a+s\cdot n\cdot d\cdot v+(n+d)\cdot(a+r))$. Assuming that loop depth and array dimension are bounded by a constant, this simplifies to $O(r+a+s\cdot v+p(c)\cdot l)$. Because every measure in this formula is worst case proportional to the size of the program, our method is polynomial in the size of the program. c is typically proportional to the depth of nesting, hence we can reasonably assume that $p(c)$ is bounded by a constant as well, and we obtain $O(a+s\cdot v+r+l)$. The presence of $s\cdot v$ is due to the reaching definitions analysis, and this is the only term that could have a tendency to grow large because as the number of statements s increases, the number of variables v is likely to increase as well. This makes our method quadratic. However, our experiments suggest that our method tends towards linearity. The difference between our experiments and our analysis is probably caused by the fact that the big-O notation discards the constant coefficients. This seems to indicate that the coefficients of the linear parts of the complexity weigh heaviest – most notably $p(c)$ can be quite a large constant.

8. EXPERIMENTAL RESULTS

The DSA transformation has been implemented and our tool uses the Omega library [Pugh 1991] for modeling and simplifying the iteration domains, and for finding the extremal values of the iterators. The experiments were run on a Pentium 4 2.4GHz with 768 MB RAM. We have run experiments with MatParser [Kienhuis 2000] as well. MatParser is a mature, heavily optimized tool that uses Feautrier's DSA transformation and is believed ready for commercial use. This makes it a good candidate for comparison. The experiments with MatParser were done on a Pentium 4 2.8GHz with 768 MB RAM. The reason is that MatParser is not freely

bench.	LOC	n	time	Mat.
*gauss*1	16	2	0.008s	0.174s
*gauss*2	17	2	0.009s	0.153s
durbin	63	2	0.058s	0.549s
schdc	101	2	0.075s	1.654s
tomcatv	111	3	0.175s	1.469s
usvd	131	3	0.084s	0.731s
swim	191	3	0.258s	1.949s
voc	541	3	0.441s	2.108s
qsdpcm	495	12	1.173s	×
*gauss_dd*1	65	3	0.047s	N/A
*gauss_dd*2	64	3	0.043s	N/A
*mp*3_1	75	4	0.080s	N/A
*mp*3_2	66	4	0.053s	N/A
*cavdet*1	70	4	0.048s	N/A
*cavdet*2	96	4	0.048s	N/A
*cavdet*3	93	4	0.046s	N/A
*cavdet*4	53	4	0.038s	N/A
*cavdet*5	53	4	0.037s	N/A
*cavdet*6	54	4	0.036s	N/A
*cavdet*7	54	4	0.036s	N/A
*cavdet*8	53	4	0.035s	N/A
*cavdet*9	61	4	0.043s	N/A

Table I. Benchmarks with transformation times (X: memory problems, N/A: not applicable because of data dependent behavior).

available and thus had to be run on the computer systems of LIACS.

While our DSA tool is written in C++, MatParser is written in Java. However MatParser uses the Omega library and an implementation of Feautrier's PIP algorithm [Feautrier 1988b], written in C++ and C respectively. These two libraries are the main consumers of processor time and memory, and Java serves mainly as glue code. Hence, we consider the comparison fair. When speaking of the program running out of memory, this means a slightly different thing for our tool and MatParser. When our tool runs out of memory, this means that the Linux kernel kills the process for using too much memory. In the case of MatParser this means that the machine starts thrashing and we reboot it after 15 minutes of unresponsiveness. This difference is caused by the configuration of the machines and not by the tools.

8.1 Comparison with MatParser

We have applied the transformation to a number of multimedia and numerical kernels. These benchmarks are shown in Table I. For some benchmarks that we use in verifications, we have transformed versions of the original program. They are distinguished by a sequence number at the end of the name of the benchmark. The LOC column lists the number of lines of code, and the n column lists the maximum loop nest depth. The time columns lists the time required by our tool to do the DSA transformation. The last column lists –where applicable– the corresponding time for MatParser.

Of these benchmarks, the first 9 do not contain data dependent indexation or conditions and can be handled by MatParser. A notable exception is the *qsdpcm* benchmark where MatParser runs out of memory after a few minutes of processing. The remaining benchmarks are data dependent and thus cannot be handled by MatParser. For the first 8 benchmarks, we see that we outperform MatParser by almost an order of magnitude. If the *qsdpcm* is slimmed down to about one fourth of the code, MatParser can barely handle it with 768MB memory, while our tool handles the whole benchmark in about 450MB. This indicates that our tool is much more memory efficient.

Another observation from Table I is that the DSA transformation time usually shows a slight improvement as more transformations are performed. This is because

$\boldsymbol{k}$	2	4	6	8	10	12	14	16	18	20	100
$\boldsymbol{t(s)}$	0.094	0.196	0.301	0.390	0.502	0.595	0.693	0.789	0.889	0.995	5.454
$\boldsymbol{t/k}$	0.047	0.049	0.050	0.049	0.050	0.050	0.050	0.049	0.049	0.050	0.055

Table II. Scalability experiment.

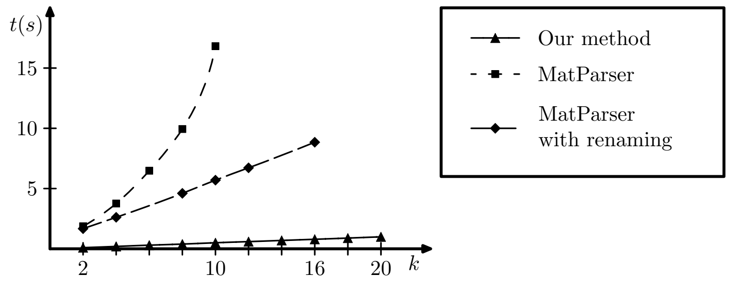


Fig. 17. Comparison with MatParser of the transformation time as function of the number of instances of the cavity detector.

these transformations try to line up assignments to and reads from array elements, making the data flow simpler and easier to determine, as shown by our timings. An exception to this rule is *cavdet*9 where transformation time goes up. The transformation applied between *cavdet*8 and *cavdet*9 is the introduction of extra arrays to contain copies of heavily used data. This creates extra variables and extra assignments, and causes an increase of the transformation time.

8.2 Scalability experiment

To ascertain that our method scales well with growing program size, we have run our DSA transformation on a program of growing size. Different loop structures in different benchmarks cause noise on such measurements. To reduce this noise we have opted to chain the cavity detector kernel (a data dependence free adaptation of *cavdet*1) together a number of times (denoted k) with the output array of one instance of the cavity detector the same as the input array of the next instance. This way we can build reasonable programs of a size that is a multiple of the original cavity detector, but the extra code that is added each time has a comparable loop structure. The measurements are shown in Table II. The transformation time is shown as t, and the transformation time per instance of the cavity detector is shown as the ratio t/k. This ratio lies around 50 ms, although it rises when k reaches 100. This indicates that the linear component of the complexity is most prominent, except for large programs where the quadratic component dominates.

A comparison with MatParser is plotted in Figure 17. Timings for MatParser cannot be given for $k > 10$ because it runs out of memory for larger k. While the behavior is non-linear, there are not enough data points to judge whether it is exponential. As it turns out, the problem with MatParser (in this case, but also in general) is its poor handling of multiple assignments to the same variable. This is a problem specifically highlighted by our scalability experiment as each assignment to a variable is repeated k times.

If the same experiment is repeated with each instance of the cavity detector given

```
for (y=0; y<=8; y++)
   ⋮
  for (n=0; n<=1; n++) {
    p2 = ···;
    ···p2···;
  }
```

Fig. 18. A tough loop.

```
for (y=0; y<=8; y++)
   ⋮
  for (n=0; n<=1; n++) {
    p2[y]···[n] = ···;
    ···p2[y]···[n]···;
  }
```

Fig. 19. DSA of Figure 18.

```
for (y=0; y<=8; y++)
   ⋮
  for (n=0; n<=1; n++) {
    p2[y] = ...;
    ...p2[y-1]...;
  }
```

Fig. 20. Variation on Figure 18.

var.	y	x	vy	vx	g	h	i	j	k	l	m	n
ours	0.060	0.060	0.060	0.060	0.060	0.060	0.060	0.060	0.060	0.060	0.060	0.060
Mat.	0.422	0.423	0.497	0.605	0.742	1.043	1.576	2.711	4.894	9.527	19.05	41.39

Table III. DSA transformation time comparison for instances of Figure 20.

its own private set of variables (by renaming them), an improvement in MatParser's performance should be seen. This is confirmed by the results shown in Figure 17. A first observation is that it is now possible to get up to $k = 16$ before running out of memory. A second observation is that the run time is now linear in k. Note that the variable renaming has no impact on the run time of our tool. This follows from its remarkable capability to discern different uses of the same variable.

8.3 A tough loop

Another experiment with the *qsdpcm* benchmark shows that MatParser has problems with deeply nested loops for the same reason. MatParser, like our method, searches for the last preceding assignment in a previous iteration of the inner loop, or, if there is no such iteration, in the previous iteration of the surrounding loop, and so on. This creates a separate case for each loop, each of which needs to be investigated separately. In the case of *qsdpcm*, the most deeply nested loop gives rise to 12 cases, causing MatParser to go out of memory. The statements that cause MatParser to fail are shown in Figure 18. Each of the 12 loops are perfectly nested with constant bounds on the iterators. The DSA form of this program is simple and shown in Figure 19. As mentioned above, the split-up in 12 cases –one for each loop– causes an exponential effect. However the number of dimensions causes an exponential effect as well. To single out the former effect, we need a constant loop dimension and vary the number of cases in which MatParser needs to split up. This can easily be achieved by adapting the loop nest as shown in Figure 20. The new indexation makes sure that we need to look at a previous iteration of the `y`-loop or the –in this case absent– outer loops. So in the case of Figure 20, only a single case is found feasible, and the combinatorial effect is avoided. Substituting `y` by another iterator allows more cases, and increases the combinatorial effect.

The results of applying both MatParser and our system to each of the 12 instances of Figure 20, are shown in Table III. The **var.** row shows the variable that is used instead of `y` in the indexation of the statement in Figure 20. The numbers in the table confirm that MatParser behaves exponentially. The run times of our tool remain the same since the indexation is never manipulated and hence cannot influence the timings.

benchmark	*gauss1*	*gauss_dd1*	*qsdpcm_dd*
number of executed assignment instances (min/max)	5347/5347	10101/2115351	5019006/5692206
original array size	201	10007	67741
array size for Feautrier	10198	N/A	N/A
array size for our DSA	101089	30301080201	1407717534

Table IV. Measurements of the overhead of DSA.

9. HANDLING THE OVERHEAD

If regarded as an executable program, the DSA form seems to incur a large overhead. On the one hand, memory use can become huge as each assignment instance needs its own array element to assign. On the other hand, our transformation technique can add large numbers of copy assignments, which both increases the memory overhead as each copy assignment needs its own memory element too, and increases the computational overhead. We stress here again that DSA form is intended as an intermediate representation rather than an executable program. The intended use case of DSA form is thus to first transform a program to DSA, then to transform or optimize it taking advantage of the DSA property, and then remove the overhead again. In this section, we evaluate this overhead and show how it can be removed again.

The overhead introduced by our DSA transformation is no problem for applications like functional verification [Shashidhar et al. 2003; 2005] as it can handle the copy assignments and it has no problems with the increased memory requirements as it does not actually execute the programs passed to it. In applications like parallelization, the addition of copy assignments does not decrease the possibility for reordering. Instead of reading from an array element directly, sometimes a copy is read from. This can be another copy, leading to copy chains. The read and write can still be reordered as before, and the copy chain simply needs to be executed in between. However, the extra copy assignments and increased array size do incur a performance penalty.

9.1 Measuring the overhead

To get an idea of the amount of overhead, we list some representative measurements in Table IV. Note that the *qsdpcm_dd* benchmark used here is the original one with data dependent behavior, contrary to the *qsdpcm* version used in our previous experiments. The measurements for *gauss2* and *gauss_dd2* are the same as for their respective original version, so we do not list them. The second and third row of the table list the number of executed assignment instances and the total array size of the original program respectively. The third and fourth row list the total array size if the program were transformed to DSA using Feautrier's method and our method respectively. Although in theory, only as many array elements are needed as there are assignment instances, in practice we waste space because arrays need to be rectangular. In our method, loop bounds are turned into constants, and hence, we write a rectangular array section and we do not waste space. In other words, the fourth row of Table IV can be interpreted as the new number of assignment instances too. Finally, we mention that for the number of executed

$\boldsymbol{k}$	2	4	6	8	10	12	14	16	18	20	26
$\boldsymbol{t(s)}$	1.435	3.020	4.763	6.498	8.412	10.44	12.48	14.77	17.26	19.74	25.82
$\boldsymbol{t/k}$	0.718	0.755	0.794	0.812	0.841	0.871	0.892	0.924	0.957	0.987	0.993

Table V. Scalability experiment with the tool chain.

statement instances, we specify two bounds. This is because the actual number varies depending on the value of the data dependent conditions that govern the execution of some statements.

The increase in array size and statement instances after our DSA transformation is worst in the *gauss_dd1* and *gauss_dd2* benchmarks. This is caused mainly by huge array sizes, as the `Up` operation writes the whole array, which is more expensive with larger arrays. The other factor is how close the loop bounds are to constants. In the Gaussian elimination benchmarks, a triangular loop nest is used, and introducing constant bounds about doubles the number of statement instances.

9.2 Removing the overhead

The problem of increased array size is not specific to our method. Array compaction has been studied in the context of memory optimization, *e.g.,* [De Greef et al. 1996] or [Tronçon et al. 2002]. If a value in an array element is no longer needed, it can be overwritten with another value. This is done by changing the indexation of the assignments such that multiple assignment instances can store values in the same array element, taking care that no values are overwritten that are still needed. The two approaches from [De Greef et al. 1996] and [Tronçon et al. 2002] differ in the changes to the indexation that they consider. In-place mapping intentionally destroys the DSA property, and thus it should be applied to the code after the optimizations that require DSA form.

A problem specific to our approach is the large amount of extra copy assignments. Advanced copy propagation followed by dead code elimination [Vanbroekhoven et al. 2003] can remove the copy operations again. The idea is to read a copied array element instead of the copy of it, following chains of copy operations if necessary. Chaining our DSA transformation and advanced copy propagation tools can then be used as a drop-in replacement for DSA transformation based on Feautrier's method, *e.g.,* MatParser. This works because advanced copy propagation does not destroy the DSA property.

First, we apply the tool chain to the benchmarks without data dependent behavior. We repeat the scalability experiment, resulting in Table V. In this case only up to 26 repetitions of the cavity detector can be handled before running out of memory. The resulting times are not linear as t/k rises, but t/k only rises slowly so this is not a problem. All of the copy assignments could be and have been removed in each instance.

Next, we repeat the experiment with the single 12-dimensional loop. The results are shown in Table VI. The times are generally larger than the corresponding times for MatParser for the outer iterators, but for the inner iterators we gain by an order of magnitude even though the times rise slightly for the inner iterators. The times however do not rise monotonically, but this is probably due to the Omega library's unpredictability performance wise. Again, all copy assignments have been removed.

var.	y	x	vy	vx	g	h	i	j	k	l	m	n
Mat.	0.422	0.423	0.497	0.605	0.742	1.043	1.576	2.711	4.894	9.527	19.05	41.39
ours	1.814	1.975	2.114	2.141	2.516	2.574	2.384	2.607	2.766	2.569	3.409	4.966

Table VI. Tool chain time for instances of Figure 20.

If data dependent behavior is present, not all copy assignments can be removed. Applying the tool chain to the toughest benchmark in the set, namely *qsdpcm_dd*, requires 136.8 seconds. The remaining number of assignment instances is 5692206, the maximum number for the original program (see Table IV). This is a good result as on average, we have about 7% extra executed statement instances –assuming that statements guarded by a data dependent condition execute about half of the time– and we have a maximum of 14% of extra statement instances.

The *gauss_dd* benchmarks have both data dependent indexation due to the row swap because of the pivoting, and data dependent conditions that remove useless row operations and row scaling. Applying the tool chain results in a range of 515101 through 2115351 assignment instances. This means that we have, on average, an extra of 24% of statement instances, and a maximum extra of 410%. This is worse than it seems as this maximum is reached for the zero matrix. For random matrices, the number of executed statement instances is closer to 2115351, and the overhead is negligible.

Finally, we mention that it may be impossible to remove all of the copy assignments due to data dependent behavior, or it may be prohibitively expensive to do so. The bulk of the copy assignments can usually be removed with little effort, and the remaining few copy assignments tend to be tough. To illustrate this, the *cavdet1* benchmark has four disequalities inside conditionals, and these are padded by adding a copy in the else branch. Usually, disequalities are false only rarely, so the overhead is negligible. Consider the case of *cavdet1* after removal of all copies except those because of disequalities. The cost of removing these copies too is 34% extra execution time, giving only a decrease between 3.5% and 5.5% in executed statement instances. These tough copy operations can be left in, and it is still possible to remove them during the in-place mapping phase. The idea is to map the source and target of a copy operation onto the same array element, making the copy assignment a no-op operation that can simply be left out. This is part of future work.

In summary, there are methods to remove copy assignments and methods to reduce array sizes by reintroducing destructive assignments. In theory, they are able to remove all copy assignments and to undo the DSA transformation. This is not necessarily the case once the DSA code has been transformed. Indeed, it is then possible that some arrays cannot be compacted or that some copy assignments cannot be removed. If the resulting overhead is smaller than the gain that follows from transformations exploiting the DSA form, then we have a net gain; this is a design trade-off.

10. CONCLUSION

In this paper we have presented a new method that transforms programs to dynamic single assignment which is in practice linear in the size of the program (for a constant loop depth).

This is achieved by adding copy operations in such a way that we can use a fast, exact scalar reaching definitions analysis, whereas existing methods need an expensive, exact array data flow analysis.

Experiments with our tool implementing the DSA transformation confirms the scalability of our method. The tool is currently being used as an enabling step for functional verification [Shashidhar et al. 2003; 2005] and memory optimizations [Catthoor et al. 1998]. The extra copy operations can be removed by advanced copy propagation [Vanbroekhoven et al. 2003]. On the one hand, experiments show that for code that is not data dependent, all copy operations can be removed, and the resulting code is comparable to the code produced by Feautrier's method. On the other hand, for programs involving data dependent indexation or conditions, not all copy operations can be removed as that would imply that the program is fully analyzable, which is not so in general. However, there is the option to leave some of the copy operations. This would allow controlling the complexity of the transformation as well as the code complexity of the resulting program, as both are tightly linked. This complexity is often due to border conditions giving rise a small number of copy operation instances that are not worth removing given the cost of doing so. The investigation of the trade-off of remaining overhead versus transformation time and code complexity is left for future work.

In summary, the chain of DSA and advanced copy propagation is scalable and produces DSA code similar to that of MatParser for non data dependent programs. For data dependent programs it produces in most cases only a limited number of extra copies due to the data dependent parts of the code, and is the only available tool to do so.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Inc.

Alias, C. 2003. `f2sare`. http://www.prism.uvsq.fr/users/ca/progs/f2sare.tgz.

Ballance, R. A., Maccabe, A. B., and Ottenstein, K. J. 1990. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *Proceedings of PLDI'90.* 257–271.

Benini, L. and Micheli, G. D. April 2000. System-level power optimization techniques and tools. *ACM Trans. on Design Automation for Embedded Systems (TODAES) 5,* 2, 115–192.

Bu, J. and Deprettere, E. 1988. Converting sequential interative algorithms to recurrent equations for automatic design of systolic arrays. In *Proceedings of ICASSP '88.* Vol. vol IV. IEEE Press, 2025–2028.

Catthoor, F., Wuytack, S., De Greef, E., Balasa, F., Nachtergaele, L., and Vandecappelle, A. 1998. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic Publishers.

Collard, J.-F., Barthou, D., and Feautrier, P. 1997. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing 40,* 2, 210–226.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (October), 451–490.

De Greef, E., Catthoor, F., and De Man, H. 1996. Reducing storage size for static control programs mapped onto parallel architectures. In *Proceedings of Dagstuhl Seminar on Loop Parallelization*.

Feautrier, P. 1988a. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*. St. Malo, France, 429–441.

Feautrier, P. 1988b. Parametric integer programming. *Operationnelle/Operations Research 22,* 3, 243–268.

Genin, D. and Hilfinger, P. 1989. *Silage Reference Manual, Draft 1.0*. Silvar-Lisco, Leuven.

Karp, R., Miller, R., and Winograd, S. 1967. The organization of computations for uniform recurrence equations. *Journal of the ACM 14*, 563–590.

Kienhuis, B. 2000. Matparser: An array dataflow analysis compiler. Tech. rep., University of California, Berkeley. February.

Kienhuis, B., Rijpkema, E., and Deprettere, E. 2000. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign*. 13–17.

Knobe, K. and Sarkar, V. 1998. Array SSA form and its use in parallelization. In *Symposium on Principles of Programming Languages*. 107–120.

Li, Z. 1992. Array privatization for parallel execution of loops. In *ICS '92: Proceedings of the 6th International Conference on Supercomputing*. ACM Press, New York, NY, USA, 313–322.

Offner, C. and Knobe, K. 2003. Weak dynamic single assignment form. Tech. Rep. HPL-2003-169, HP Labs. Aug.

Panda, P., Catthoor, F., Dutt, N., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandecappelle, A., and Kjeldsberg, P. G. April 2001. Data and memory optimizations for embedded systems. *ACM Trans. on Design Automation for Embedded Systems (TODAES) 6,* 2, 142–206.

Pugh, W. 1991. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*. Albuquerque, NM.

Quinton, P. 1984. Automatic synthesis of systolic arrays from uniform recurrent equations. In *ISCA '84: Proceedings of the 11th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 208–214.

Scholz, S.-B. 1994. Single Assignment C - Functional Programming Using Imperative style. In *Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL'94)*. 21.1–21.13.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F., and Janssens, G. 2003. An automatic verification technique for loop and data reuse transformations based on geometric modeling of programs. *Journal of Universal Computer Science 9,* 3, 248–269.

Shashidhar, K. C., Bruynooghe, M., Catthoor, F., and Janssens, G. 2005. Verification of source code transformations by program equivalence checking. In *Compiler Construction, 14th International Conference, CC 2005, Proceedings*. LNCS, vol. 3443. Springer, 221–236.

Tronçon, R., Bruynooghe, M., Janssens, G., and Catthoor, F. 2002. Storage size reduction by in-place mapping of arrays. In *Verification, Model Checking and Abstract Interpretation, VMCAI 2002, Revised Papers*. LNCS, vol. 2294. 167–181.

Vanbroekhoven, P., Janssens, G., Bruynooghe, M., and Catthoor, F. 2005. Transformation to dynamic single assignment using a simple data flow analysis. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems, Tsukuba, Japan*.

Vanbroekhoven, P., Janssens, G., Bruynooghe, M., Corporaal, H., and Catthoor, F. 2003. Advanced copy propagation for arrays. In *Languages, Compilers, and Tools for Embedded Systems LCTES'03*. 24–33.