

Advanced Signal Propagation

Peter Vanbroekhoven Gerda Janssens
Maurice Bruynooghe Henk Corporaal
Francky Catthoor

Report CW 353, January 2003



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Advanced Signal Propagation

Peter Vanbroekhoven *Gerda Janssens*
Maurice Bruynooghe *Henk Corporaal*
Francky Catthoor

Report CW 353, January 2003

Department of Computer Science, K.U.Leuven

Abstract

The focus of this paper is on a data flow-transformation called advanced signal propagation. After an array is assigned, we can, under certain conditions, replace a read from this array by the righthand side of the assignment. This way we skip the intermediate assignment so it possibly becomes dead code and we eliminate it. Where necessary we distinguish between the different elements of the array as well as the different runtime instances of statements, allowing us to do propagation over global loop and condition scopes. To this end we have formalized two basic operations: non-recursive and recursive propagation. These two operations have been implemented in a prototype tool. Preliminary experiments on a cavity detector show a decrease of about 8% in memory accesses and about 5% in memory size.

Advanced Signal Propagation

Peter Vanbroekhoven^{1*}, Gerda Janssens¹, Maurice Bruynooghe¹, Henk Corporaal², and Francky Catthoor²

¹ Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{`peterv,gerda,maurice`}@`cs.kuleuven.ac.be`

² Interuniversity MicroElectronics Center
Kapeldreef 75, B-3001 Leuven, Belgium
{`heco,catthoor`}@`imec.be`

Abstract. The focus of this paper is on a data-flow transformation called advanced signal propagation. After an array is assigned, we can, under certain conditions, replace a read from this array by the righthand side of the assignment. This way we skip the intermediate assignment so it possibly becomes dead code and we eliminate it. Where necessary we distinguish between the different elements of the array as well as the different runtime instances of statements, allowing us to do propagation over global loop and condition scopes. To this end we have formalized two basic operations: non-recursive and recursive propagation. These two operations have been implemented in a prototype tool. Preliminary experiments on a cavity detector show a decrease of about 8% in memory accesses and about 5% in memory size.

1 Introduction

Since the rise of the World Wide Web, the number of multimedia and network applications has been growing rapidly. These are applications that process large amounts of data. In e.g. a C-program this data will be stored in large arrays. Techniques exist to allocate arrays to registers[1], but lots of arrays will still be in memory. This is however a problem because of the exponentially growing gap between processor speed and main memory speed[2]. Another problem with large, high-speed memories is that they consume lots of power and hence have a large heat dissipation[3].

Since current hardware technology is unable to provide a solution to this problem, a software solution is necessary. Most classic compiler optimization techniques however fail on array variables since they concentrate on simple data types and records.³ A very important optimization is copy propagation[4, 1]. The idea there is that after an assignment $\mathbf{f} = \mathbf{g}$, if possible, \mathbf{g} is used instead of \mathbf{f} .

* Supported by a specialisation grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)

³ These are Pascal-like records or `structs` in C.

```

for (i = 0; i < 50; i++)
  a[i] = b[i] * b[i]; /* S1 */
for (i = 0; i < 100; i++) {
  if (i < 50)
    a[i + 50] = b[i]; /* S2 */
  c[i] = a[i];        /* S3 */
}
temp = a[50];        /* S4 */

for (i = 0; i < 50; i++)
  a[i] = b[i] * b[i]; /* S1 */
for (i = 0; i < 100; i++) {
  if (i == 0)
    a[i + 50] = b[i]; /* S2 */
  if (i < 50)
    c[i] = a[i];      /* S3' */
  else
    c[i] = b[i - 50]; /* S3'' */
}
temp = a[50];        /* S4 */

```

Fig. 1. *Left:* an example on which classic copy propagation fails. *Right:* Propagated S2 to S3 using advanced signal propagation

If then the assignment becomes dead code, it can be eliminated. In that case we save two memory accesses and a memory location.

Nonetheless, this technique fails in the example of Fig. 1. First note that this piece of code can be part of a larger program, the only thing of importance is that there are no further reads from array `a`, so S4 is the last read from `a`. We would now like to propagate copy operation S2 to S3. One bump in the road is that the copy propagation from [1] does not take subscripts into account. But even if it did, we cannot just propagate S2 to S3 since elements 0 through 49 read by S3 are not written by S2 but by S1 which has a different righthand side. Also if we did propagate, S2 would not become dead code yet because S4 still reads `a[50]`. Yet only `a[50]` should be written, i.e. S2 is dead code for all iterations except for `i` equal to 0.

We will show in this paper *how to overcome the limitations of classic copy propagation by distinguishing between the different runtime executions of a statement*, called instances. For e.g. S2 in Fig. 1 we distinguish between 50 instances, one for each value of iterator `i` smaller than 50. This allows us to propagate only part of the instances of a statement and then remove only the instances of the statement that have become dead code. Propagating S2 to S3 in the example above, and removing dead code, gives us the program in the right side of Fig. 1. As can be seen, S2 is only executed for `i = 0` and S3 is split in two statements S3' and S3'' that are conditionally executed to allow propagation to only S3'', as required. The net effect is that 49 instances of S2 are removed and thus there are 98 less memory accesses (49 for `a` and 49 for `b`). To do this we have to introduce extra conditions and there is some code duplication, but executing these is less expensive than memory accesses, especially in case of cache misses. Also note that elements 51 through 99 are no longer accessed, so the array can be shrunk which is even better for cache behaviour.

The method we consider in this paper is however not limited to propagating copy operations. Any assignment to an array (sometimes called signal, hence sig-

nal propagation⁴) can be propagated. This allows us to do for example constant propagation or expression propagation too.

To be able to easily distinguish between the runtime instances of statements, we apply our methods to programs in dynamic single assignment (DSA) form. This means that during execution of the program, only a single assignment happens to each array element or simple variable. This is a stronger form of single assignment than the well-known static single assignment[4, 1]. Since array elements are assigned a value only once, they become equivalent to the values they were assigned, allowing much easier transformations on the data flow since we need not concern ourselves with unnecessary details that result from data allocation and memory reuse.

However, the use of DSA comes at a price. One thing is that currently only a subset of imperative programs like C-programs can be automatically converted to DSA form[5, 6]. Fortunately many multimedia applications belong to that subset. Hence DSA is already widely used in the context of parallelization [7] and design of systolic arrays [8]. A second problem is that array sizes can grow considerably. Advanced methods exist for compacting arrays[9, 10], and these methods should be used in combination with loop transformations[11].

In Sec. 2 we explain how we geometrically model the subset of programs we can handle. Then in Sec. 3, we explain our advanced signal propagation. We first introduce the basic idea in Sec. 3.1. Next we use this basic idea in Sec. 3.2 and 3.3 to explain the two basic operations, being respectively non-recursive and recursive signal propagation. We use a running example in our explanation and this example is wrapped up in Sec. 3.4. Finally we present some preliminary results in Sec. 4 and a comparison of our method to related work in Sec. 5.

2 Preliminaries

The data-flow transformations presented in this paper can handle programs that satisfy the following requirements:

- The program consists of a nest of `for`-loops and `if`-statements. Any possible nesting is allowed. The step of the `for`-loops should be a constant and is not restricted to 1.
- Anywhere in the nesting, assignment statements can be present. The lefthand side of an assignment is to an element of an array.⁵ The righthand side of the assignment can be any expression containing array references with no restrictions on indexing.
- All expressions used as bounds for `for`-loops or as tests for `if`-statements are affine combinations of the surrounding loop iterators, i.e. a linear combination of them plus a constant.

⁴ We could have called our method advanced copy propagation as well, but historically it has been called advanced signal propagation and we decided to go with that name.

⁵ Note that scalar variables like integers could be considered as arrays of length 1.

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i < 105)
      a[j + i][j] = f(in[j + i]);          /* S1 */
    else
      a[j + i][j] = a[j + i - 5][j - 3]; /* S2 */
for (int i = 0; i < 100; i++)
  out[i] = a[i + 99][i];                  /* S3 */

```

Fig. 2. Our running example

- The program is in DSA form. We demand that the indexation in the lefthand side of an assignment surrounded by n `for`-loops is of the following form:

$$A \cdot \mathbf{i} + \mathbf{c}. \quad (1)$$

Here A is a non-singular $n \times n$ matrix, \mathbf{i} is a vector containing the iterators of the surrounding loops and \mathbf{c} is any column vector of length n . Note that this is often the case in DSA programs, e.g. the ones that are produced by the DSA conversion in [5].

This is a well-known set of programs, both in the area of parallelization as well as the area of hardware synthesis [12], since these kinds of programs can be modeled and operated on using well-established mathematical methods. The example from Fig. 2 satisfies all of the conditions above.

In this paper the programs are represented by a (notationally) simplified version of the geometrical modeling of [3]. We will introduce our notation using the program in Fig. 2. In that program, there are two loop nests. The first one consists of two loops with iterators i and j . The body of this loop nest is executed for different combinations of integral values of i and j . We can represent each of these combinations as a point in a two-dimensional *iteration space*. The set of those points forms the *iteration domain* for the body of the loop and can be written as

$$\{(i, j) \mid 0 \leq i < 100 \wedge 0 \leq j < 100 \wedge (i, j) \in \mathbb{Z}^2\}. \quad (2)$$

However statement **S1** is not executed for all points in this iteration domain, but only for those points (i, j) for which $j + i < 105$ is true. So the iteration domain for that statement is

$$I_1 = \{(i, j) \mid 0 \leq i < 100 \wedge 0 \leq j < 100 \wedge j + i < 105 \wedge (i, j) \in \mathbb{Z}^2\} \quad (3)$$

in which the subscript refers to the number of the statement. For statement **S2** which is only executed if $j + i < 105$ is not true, the iteration domain is

$$I_2 = \{(i, j) \mid 0 \leq i < 100 \wedge 0 \leq j < 100 \wedge j + i \geq 105 \wedge (i, j) \in \mathbb{Z}^2\}. \quad (4)$$

The iteration domain for **S3** is one-dimensional because it has only one surrounding `for`-loop.

$$I_3 = \{(i) \mid 0 \leq i < 100 \wedge i \in \mathbb{Z}\}. \quad (5)$$

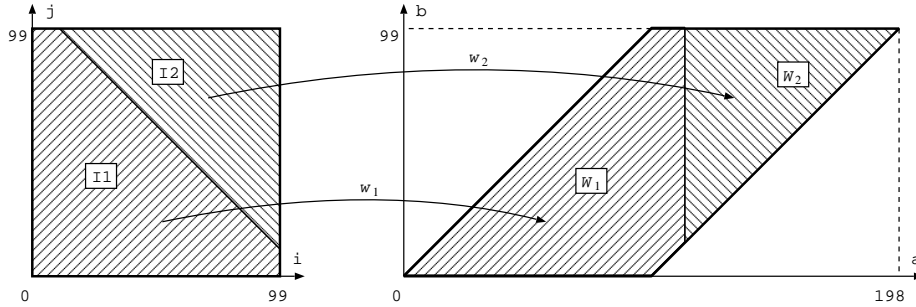


Fig. 3. Schematical representation of the iteration domains (*left*) and definition domains (*right*) for **S1** and **S2**

Now it is possible to refer to each instance of e.g. statement **S1** as $\mathbf{S1}(i_1)$ with $i_1 \in I_1$, or as $\mathbf{S1}(i, j)$ with $(i, j) \in I_1$. $\mathbf{S1}(i, j)$ with $(i, j) \in I_1$ writes to an element of array **a** indicated by $w_1(i, j)$. In our example this is

$$w_1 : \mathbb{Z}^2 \rightarrow \mathbb{Z}^2 : (i, j) \mapsto (j + i, j). \quad (6)$$

This *definition mapping* is from a two-dimensional iteration domain to a two-dimensional *variable domain*. Also $\mathbf{S1}(i, j)$ reads from an element of array **in** indicated by $r_1(i, j)$. In our case r_1 is specified by

$$r_1 : \mathbb{Z}^2 \rightarrow \mathbb{Z} : (i, j) \mapsto (j + i) \quad (7)$$

This is an *operand mapping* from a two-dimensional iteration domain to a one-dimensional variable domain. If there are multiple reads in a single statement, we can distinguish between the multiple operand mappings by adding an extra index. We will not need this facility in the remainder of the paper.

An important note is that all w_s , with s the number of a statement, are invertible. Since the program is in DSA form, there is a one-to-one mapping between the points in the iteration domain of a statement and the elements written by that statement. Such a one-to-one mapping is always invertible.

The set of elements of an array that are read or written are respectively called *operand* and *definition domain*. For **S1** the definition domain is given by:

$$W_1 = w_1(I_1) = \{(a, b) \mid \exists (i, j) \in I_1 : (a, b) = w_1(i, j)\}. \quad (8)$$

Filling in the specifics of statement **S1** gives

$$W_1 = \{(a, b) \mid \exists (i, j) \in \mathbb{Z}^2 : 0 \leq i < 100 \wedge 0 \leq j < 100 \wedge j + i < 105 \wedge a = j + i \wedge b = j\}. \quad (9)$$

The iteration and definition domains for statements **S1** and **S2** are schematically represented in Fig. 3. Similar specifications can be given for operand domains R_i of statements S_i .

Since we limited the expressions for the bounds of the loops to affine expressions of surrounding iterators, the iteration domains for each statements can be described as the integer points in an n -dimensional polyhedron, which is a part of n -dimensional space bounded by linear inequalities. Here n is the number of loops around the statement in question. In case the step of the `for`-loops is not 1, an extension of this, called \mathbb{Z} -polyhedra, is used. Finding definition and operand domains is then the image of a \mathbb{Z} -polyhedron by an invertible, affine mapping, which in turn is a \mathbb{Z} -polyhedron itself. Defining \mathbb{Z} -polyhedra and doing operations on them like image through an affine mapping or set operations like intersection can be handled by a polyhedral library as in [13]. An important feature of this library is that a conjunction of affine conditions on iterators can be simplified by discarding all conditions entailed by the bounds on the iterators.

In the remainder of the paper we will use the representation from this section together with the equivalent C-program. The data-flow transformations that we will introduce do not invalidate the memory reuse and alike, so we need not clutter the example with unnecessary details.

3 Advanced signal propagation method

In this section we will describe automatable operations necessary to do signal propagation over global loop and condition scopes. An important class of arrays (or signals) we want to handle are those that are copies of other arrays. The idea is to replace reads from such copy arrays by reads from the copied arrays. The effect of this is that (part of) the copy is no longer read and thus can be removed, along with the corresponding copy operations.

We will first explain the basic idea behind advanced signal propagation in Sec. 3.1. Then in the following sections we will apply this basic idea to elaborate two basic operations that are necessary to do signal propagation: propagate a statement that copies array elements written by another statement in Sec. 3.2 and propagate a statement that copies array elements written by itself in Sec. 3.3.

3.1 Basic idea

In the example of Fig. 2, statements **S1** and **S2** write to array **a**, and statements **S2** and **S3** read from it. In Fig. 4 the definition and operand domains related to **a** are represented. This figure is the same as the right side of Fig. 3, but with the operand domains added as the two shaded areas. We see that **R2** overlaps with **W1** and **W2**, which means that statement **S2** reads array elements written by both **S1** and itself. We also see that **R3** overlaps with **W1** and **W2**, meaning that statement **S3** reads array elements written by **S1** and **S2**.

Since **S2** reads array elements written by **S1**, we will try to *replace the read from array **a** in statement **S2** by the righthand side of **S1***, with the correct values for **i** and **j** filled in. To do this, we need to find out which **S1**(i_w, j_w) wrote the array element read by **S2**(i_r, j_r). If **S2**(i_r, j_r) is to read the value written by

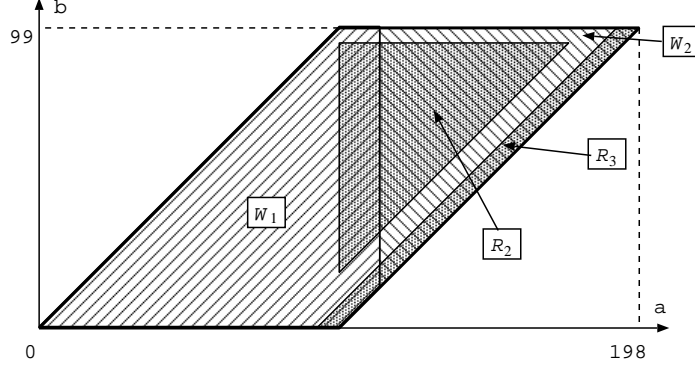


Fig. 4. Definition and operand domains for statements related to **a**

$S1(i_w, j_w)$, then they should refer to the same array element, i.e. the following condition should be true:

$$r_2(i_r, j_r) = w_1(i_w, j_w). \quad (10)$$

Recall that w_1 should be invertible. So we can then find i_w and j_w as follows:

$$(i_w, j_w) = (w_1^{-1})(r_2(i_r, j_r)). \quad (11)$$

We took the liberty of applying w_1^{-1} , a function in two arguments, to a single couple meaning that each of the elements of the couple are passed as arguments to w_1^{-1} in the same order as they appear in the couple. Then in statement $S2(i_r, j_r)$ we can replace the read from the array element indicated by $r_2(i_r, j_r)$ by the righthand side of $S1((w_1^{-1})(r_2(i_r, j_r)))$. This however is only true if $S1((w_1^{-1})(r_2(i_r, j_r)))$ is actually executed and this is when

$$(w_1^{-1})(r_2(i_r, j_r)) \in I_1. \quad (12)$$

Thus we can only propagate the array written by $S1$ to $S2(i_r, j_r)$ when (12) is true, otherwise we need to continue using the original array reference. Once we have propagated $S1$ to $S2$, some instances of $S1$ may have become dead code and hence we can remove them. Let $S = \{S_{i_1}, \dots, S_{i_s}\}$ be the set of other statements reading values written by $S1$. With R_{i_j} the operand domain of S_{i_j} , the set of array elements read by S is given by $R = R_{i_1} \cup \dots \cup R_{i_s}$. Hence $S1$ still has to be executed for the iterations

$$w_1^{-1}(R) \cap I_1. \quad (13)$$

The intersection with I_1 is necessary as it is possible that statements different from $S1$ write some of the elements in R .

3.2 Non-recursive signal propagation

In this section we show how to apply the basic idea of Sec. 3.1 to a statement that does not read any array elements that it wrote itself. The case where a statement does read values it wrote itself is handled in Sec. 3.3. The case we consider in this section is either when a statement reads from different arrays than it writes to, or when it reads from another part of the same array than where it writes to. Formally, the latter means that for statement S_x , $W_x \cap R_x = \emptyset$.

In the program from Fig. 2, statement **S1** falls into this category since it reads from array **in** and writes to array **a**, so it cannot possibly read any of the values it wrote. We will propagate the array elements written by **S1** to **S2**, i.e. we will replace the read from array **a** in **S2** by the righthand side of **S1**. We first need to calculate (11). We know r_2 , so we only need to calculate w_1^{-1} :

$$w_1^{-1} : \mathbb{Z}^2 \rightarrow \mathbb{Z}^2 : (x, y) \mapsto (x - y, y). \quad (14)$$

Equation (11) then becomes

$$(i_w, j_w) = (w_1^{-1})(j_r + i_r - 5, j_r - 3) = (i_r - 2, j_r - 3). \quad (15)$$

Substituting (15) in the righthand side of **S1**(i_w, j_w) then gives:

$$\mathbf{f}(\mathbf{in}[i_r + j_r - 5]) \quad (16)$$

This is the expression we want to substitute into **S2**(i_r, j_r). The condition under which this is allowed is given by (12):

$$(i_r - 2, j_r - 3) \in I_1. \quad (17)$$

Filling in (3) for I_1 , we get

$$0 \leq i_r - 2 < 100 \wedge 0 \leq j_r - 3 < 100 \wedge j_r - 3 + i_r - 2 < 105. \quad (18)$$

Note that we left out the condition that $(i_r - 2, j_r - 3)$ should be in \mathbb{Z}^2 because this is always the case since i_r and j_r are integer iterators. However in this case we can simplify this condition even more. Any element of array **a** is either written by **S1** or **S2**, as can easily be derived from Fig. 2. The condition under which **S1**(i_w, j_w) wrote the value read by **S2**(i_r, j_r) is given in (17). In the same way the condition under which **S2**(i_w, j_w) wrote the value read by **S2**(i_r, j_r) is the following:

$$(i_r - 2, j_r - 3) \in I_2. \quad (19)$$

Note that in both cases we find $(i_r - 2, j_r - 3)$ for (11). This is not surprising since both definition mappings, i.e. w_1 and w_2 , are the same. Filling in (4) for I_2 , we get

$$0 \leq i_r - 2 < 100 \wedge 0 \leq j_r - 3 < 100 \wedge j_r - 3 + i_r - 2 \geq 105. \quad (20)$$

We see that (18) and (20) have a part in common, in particular the part that refers to the loop bounds. This is not surprising either since the two statements

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i < 105)
      a[j + i][j] = f(in[j + i]);          /* S1 */
    else
      if(j + i < 110)
        a[j + i][j] = f(in[j + i - 5]);    /* S4 */
      else
        a[j + i][j] = a[j + i - 5][j - 3]; /* S5 */
for (int i = 0; i < 100; i++)
  out[i] = a[i + 99][i];                   /* S3 */

```

Fig. 5. S1 has been propagated to S2

have two loops in common. Either (18) or (20) has to be true—an array element, if read, should be written by some statement—and this means that the common part is always true. This supposes of course that the program shown is a complete program and no other assignments to \mathbf{a} are present. Hence we can leave this part of the condition out. Equation (18) can thus be simplified to:

$$j_r - 3 + i_r - 2 < 105 \Leftrightarrow j_r + i_r < 110. \quad (21)$$

Now we have all information to actually do the propagation on the example. The result is the program in Fig. 5. Notice that statement S2 has been split up in two statements, S4 and S5, and that an if-statement has been added so that S4(i_r, j_r) is only executed if (21) is true, while S5(i_r, j_r) is only executed if it is not true. Hence we are allowed to replace the righthand side of S4 by (16).

Now some instances of S1 have become dead code. The instances that are still alive are given by (13) where $S = \{S_3\}$, i.e.

$$\begin{aligned} w_1^{-1}(R_3) \cap I_1 &= w_1^{-1}(\{(a, b) \mid a = b + 99 \wedge 0 \leq b \leq 99\}) \cap I_1 \\ &= \{(i, j) \mid i = 99 \wedge 0 \leq j \leq 99\}. \end{aligned} \quad (22)$$

Note that we left S5 out from S since instances of S5 do not read elements written by S1 (due to the propagation). Inserting the conditions not implied by the iterator bounds yields the program of Fig. 6.

In general we can summarize the operation to do signal propagation from a statement S_x that writes to an array, to statement S_y that reads from the same array, as follows:

Algorithm 1. Non-recursive signal propagation

Goal: Propagate an assignment S_x , with definition mapping $w_x(\mathbf{i}_x) = A \cdot \mathbf{i}_x + \mathbf{c}$, to a statement S_y , with operand mapping r_y .

1. Calculate w_x^{-1} . Since w_x is of the form of (1), its inverse is simply

$$w_x^{-1} : \mathbf{a} \mapsto A^{-1} \cdot (\mathbf{a} - \mathbf{c}). \quad (23)$$

2. Calculate $\mathbf{i}_x = (w_x^{-1})(r_y(\mathbf{i}_y))$.

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i < 105) {
      if (i == 99)
        a[j + i][j] = f(in[j + i]);          /* S1 */
      } else
        if(j + i < 110)
          a[j + i][j] = f(in[j + i - 5]);    /* S4 */
        else
          a[j + i][j] = a[j + i - 5][j - 3]; /* S5 */
for (int i = 0; i < 100; i++)
  out[i] = a[i + 99][i];                    /* S3 */

```

Fig. 6. Dead instances of S1 removed

3. Calculate condition $i_x \in I_x$. If all statements in the program that write to the array in question, have the same definition mapping and if they share some loops, then remove from the condition the parts that are related to the bounds of these loops.
4. If the condition always evaluates to true (e.g. if there is just one statement that writes to the array in question), then replace the array reference in $S_y(i_x)$ by the righthand side of $S_x((w_x^{-1})(r_y(i_y)))$.
5. If the condition does not always evaluate to true, split statement S_y in two statements S_{y_1} and S_{y_2} and add an if-statement so that S_{y_1} is only executed when the condition is true and S_{y_2} is executed when it is not true. Then replace the array reference in $S_{y_1}(i_x)$ by the righthand side of $S_x((w_x^{-1})(r_y(i_y)))$.
6. Let $S = \{S_{i_1}, \dots, S_{i_s}\}$ such that each S_{i_j} reads from the array that S_x writes to and such that $i_j \neq y$. Then calculate $I' = w_x^{-1}(R_{i_1} \cup \dots \cup R_{i_s}) \cap I_x$. If I' is empty, remove S_x from the program. Otherwise restrict the iteration space of S_x to I' by adding conditions.

Using the above algorithm we also propagate S1 to S3, resulting in the program in Fig. 7. Since S1 was now completely dead code, it has been removed.

3.3 Recursive signal propagation

In this section we show how to apply the basic idea of Sec. 3.1 to a statement that reads array elements that it wrote itself. An example of this in Fig. 7 is statement S5. The behaviour of this statement is schematically represented in Fig. 8. An arrow from one array element to another array element means that the first element is copied by S5 from the other array element. Only arrows are drawn for two chains of copies. From Fig. 8 we can conclude that if we want to remove the copy operation from S5, we would need to apply Algorithm 1 as many times as the longest copy chain, each time removing one copy operation

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i >= 105)
      if(j + i < 110)
        a[j + i][j] = f(in[j + i - 5]);    /* S4 */
      else
        a[j + i][j] = a[j + i - 5][j - 3]; /* S5 */
for (int i = 0; i < 100; i++)
  if (i < 6)
    out[i] = f(in[i + 99]);                /* S6 */
  else
    out[i] = a[i + 99][i];                 /* S7 */

```

Fig. 7. S1 propagated to S3 and then removed

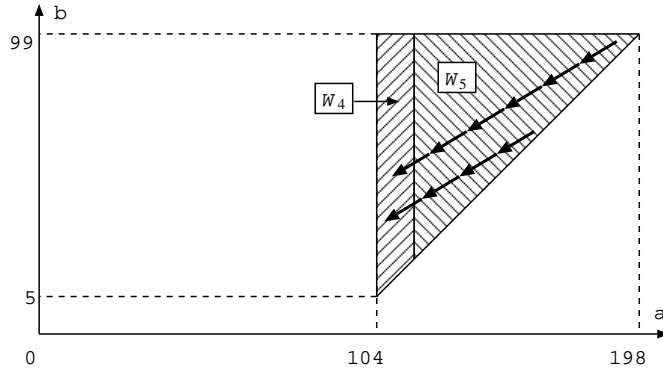


Fig. 8. S5 reads array elements that it wrote itself

from the chain. Doing this is not a good idea since the length of copy chains can be very large and each propagation possibly introduces an `if`-statement, so the program size may blow up.

Equation (15) also applies to statement S5, so $S5(i_r, j_r)$ reads the value written by $S5(i_r - 2, j_r - 3)$. Now consider a chain beginning from $S5(i_r, j_r)$ that contains n recursive copy operations, i.e. n arrows within W_5 followed by one arrow from W_5 to W_4 . The intention is to bypass the n recursive copy operations. To achieve this, the righthand side of $S5(i_r, j_r)$ needs to be replaced by the righthand side of $S5(i_r - 2 \cdot n, j_r - 3 \cdot n)$. To perform such a transformation, we need to find the value of n (which depends on the value of the iterators i_r and j_r). The iteration space of S5 is

$$I_5 = \{(i, j) \mid 0 \leq i < 100 \wedge 0 \leq j < 100 \wedge j + i \geq 110 \wedge (i, j) \in \mathbb{Z}^2\}. \quad (24)$$

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i >= 105)
      if (j + i < 110)
        a[j + i][j] = f(in[j + i - 5]);           /* S4 */
      else
        a[j + i][j] = a[j + i - 5 * ((-105 + i + j) / 5)]
                      [j - 3 * ((-105 + i + j) / 5)]; /* S5 */
for (int i = 0; i < 100; i++)
  if (i < 6)
    out[i] = f(in[i + 99]);                       /* S6 */
  else
    out[i] = a[i + 99][i];                       /* S7 */

```

Fig. 9. Solved the recursive copy propagation for S5

We need the maximal k such that $S5(i_r - 2 \cdot k, j_r - 3 \cdot k)$ is in the iteration space I_5 .⁶ This means that

$$0 \leq i_r - 2 \cdot k \leq 99 \wedge 0 \leq j_r - 3 \cdot k \leq 99 \wedge j_r - 3 \cdot k + i_r - 2 \cdot k \geq 110. \quad (25)$$

Note that we removed the strict inequalities using the fact that all variables are integer. Solving for k gives

$$\max\left(\frac{i_r - 99}{2}, \frac{j_r - 99}{3}\right) \leq k \leq \min\left(\frac{i_r}{2}, \frac{j_r}{3}, \frac{j_r + i_r - 110}{5}\right). \quad (26)$$

Note that there is always a non-negative solution as $k = 0$ expresses that $(i_r, j_r) \in I_5$. The value n of the longest recursive copy chain is given by the maximal integer value of k , hence

$$n = \min(i_r \div 2, j_r \div 3, (j_r + i_r - 110) \div 5) \quad (27)$$

For $(i_r, j_r) \in I_5$, the minimal value is always $(j_r + i_r - 110) \div 5$ so we obtain

$$n = (j_r + i_r - 110) \div 5 \quad (28)$$

Therefore the first statement in the copy chain that no longer reads array elements written by the statement itself is

$$S5(i_r - 2 \cdot ((j_r + i_r - 110) \div 5), j_r - 3 \cdot ((j_r + i_r - 110) \div 5)) \quad (29)$$

To skip the recursive copy chain we read the righthand side of this statement instance instead of the original righthand side. This results in the program of Fig. 9. Remember that division of integers is always integer division in C.

Let us now look at the general case. We have a statement $S_x(\mathbf{i}_x)$ that copies element $r_x(\mathbf{i}_x)$ to element $w_x(\mathbf{i}_x)$ of the same array. This statement also reads

⁶ Also $S5(i_r - 2 \cdot l, j_r - 3 \cdot l)$ should be in I_5 for $0 \leq l \leq k$. This holds as I_5 is convex.

elements it wrote itself, i.e. $W_x \cap R_x \neq \emptyset$. A *first assumption* is that the operation executed by S_x is a copy operation, in contrast to Sec. 3.2 where any righthand side was allowed in the assignment we propagated. A *second assumption* is that $w_x^{-1}(r_x(\mathbf{i}_x))$ is of the following form:

$$w_x^{-1}(r_x(\mathbf{i}_x)) = \mathbf{i}_x + \mathbf{\Delta}. \quad (30)$$

This means that the iteration that wrote the value read by iteration \mathbf{i}_x is displaced by a constant vector $\mathbf{\Delta}$. Because of the affine indexing, this also means that the arrows in Fig. 8 all have the same length and all point in the same direction. A *third assumption* is that the iteration domain I_x is convex, i.e. the condition on the iterators is a conjunction (i.e. an “and”) of affine inequalities. If the iteration space happens to contain a disjunction (i.e. an “or”), it can be split up into a union of convex iteration domains and each should be treated separately. If an iteration domain is convex, the inequalities governing the iterators can be concisely written as follows:

$$A \cdot \mathbf{i}_x + \mathbf{b} \geq 0. \quad (31)$$

Here A has m rows, so there are m inequalities. Strict inequalities (e.g. $a < b$) are converted to non-strict inequalities (e.g. $a + 1 \leq b$) where necessary. We again leave out the condition that the iterators should be integer because we work with integer variables only.

We can then, as in the example, find an appropriate value for n such that statement S_x for the iteration indicated by

$$\mathbf{i}_x + n \cdot \mathbf{\Delta} \quad (32)$$

no longer reads values written by S_x itself. We do this by filling (32) in for \mathbf{i}_x in (31), giving the following:

$$A \cdot (\mathbf{i}_x + n \cdot \mathbf{\Delta}) + \mathbf{b} \geq 0 \Leftrightarrow n \cdot A \cdot \mathbf{\Delta} + A \cdot \mathbf{i}_x + \mathbf{b} \geq 0 \quad (33)$$

This is a set of m inequalities. Referring to the i th element of a vector by a subscript i , these inequalities are:

$$n \cdot (A \cdot \mathbf{\Delta})_i + (A \cdot \mathbf{i}_x)_i + \mathbf{b}_i \geq 0 \text{ with } 1 \leq i \leq m \quad (34)$$

We are only interested in the inequalities that provide an upper bound on n . These are the ones for which $(A \cdot \mathbf{\Delta})_i$ is negative. We can group the index of these inequalities in the set

$$J_- = \{j \mid 1 \leq j \leq m \wedge (A \cdot \mathbf{\Delta})_j < 0\} \quad (35)$$

The upper bound on n then becomes:

$$n \leq \min_{j \in J_-} (((A \cdot \mathbf{j}_x)_j + \mathbf{b}_j) \div -(A \cdot \mathbf{\Delta})_j) \quad (36)$$

```

for (i = 0; i < 100; i++)           for (i = 0; i < 100; i++)
  a[i] = a[min(100 - i, 200 - 2 * i)];  if (100 - i < 200 - 2 * i)
                                         a[i] = a[100 - i];
                                         else
                                         a[i] = a[200 - 2 * i];

```

Fig. 10. With (*left*) and without (*right*) minimum operation

Note that the transformed statement is not suited for further recursive signal propagation unless it can be decided which element is the minimum one (this was the case in our example) and the integer division can be simplified, *i.e.*, when $(A \cdot \Delta)_j = -1$ (this was not the case in our example). If all integer divisions can be simplified, we can still make the statement suited for further recursive signal propagation by eliminating the minimum function as illustrated in Fig. 10.⁷

The algorithm for solving the recursion then becomes:

Algorithm 2. Non-recursive signal propagation

Goal: Solve a recursive copy chain in \mathbf{S}_x with w_x and r_x as respective definition and operand mappings

1. Calculate $w_x^{-1}(r_x(\mathbf{i}_x))$ and find the vector Δ according to (30) ; exit if Δ is not constant.
2. Write the inequalities on the iterators as $A \cdot \mathbf{i}_x + \mathbf{b} \geq 0$. Let m be the number of rows of A .
3. Calculate n using (35) and (36).
4. Replace the righthand side of $\mathbf{S}_x(\mathbf{i}_x)$ by the righthand side of $\mathbf{S}_x(\mathbf{i}_x + n \cdot \Delta)$.
5. If enabling further recursive copy propagation, then remove the minimum operation as shown in Fig. 10.
6. Let $S = \{\mathbf{S}_{i_1}, \dots, \mathbf{S}_{i_s}\}$ such that \mathbf{S}_{i_j} reads from the array \mathbf{S}_x writes to and such that $i_j \neq x$. Then calculate $I' = w_x^{-1}(R_{i_1} \cup \dots \cup R_{i_s}) \cap I_x$. If I' is empty, remove \mathbf{S}_x from the program. Otherwise restrict the iteration space of \mathbf{S}_x to I' by adding conditions.

3.4 Wrapping up the example

Since the recursive copy chain has been removed in Fig. 9, we can now propagate statement S5 to S7 using Algorithm 1 and then remove S5 from the program since it is dead code. The result is shown in Fig. 11. In a final step, S4 is propagated to S8 and S9, again using Algorithm 1, and then removed. This is shown in Fig. 12. The final result is a fairly simple program compared to the intermediate results⁸. This example has however been engineered to keep the calculations rather simple, and this needs not be so in practice.

⁷ By the nature of the transformation, the transformed statement is not a candidate for recursive signal propagation, but after some non-recursive signal propagation steps, the righthand side can appear in a statement that is a candidate for recursive signal propagation.

⁸ The attentive reader has probably noticed that statement S8 and S9 can be melted together, resulting in an even simpler program. This is so because S9 is only executed

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i >= 105 && j + i < 110)
      a[j + i][j] = f(in[j + i - 5]);      /* S4 */
for (int i = 0; i < 100; i++)
  if (i < 6)
    out[i] = f(in[i + 99]);                /* S6 */
  else if (i >= 11)
    out[i] = a[i + 99 - 5 * ((i - 6) / 5)]
              [i - 3 * ((i - 6) / 5)];      /* S8 */
  else
    out[i] = a[i + 99][i];                 /* S9 */

```

Fig. 11. S5 propagated to S7 and then removed

```

for (int i = 0; i < 100; i++)
  if (i < 6)
    out[i] = f(in[i + 99]);                /* S6 */
  else if (i >= 11)
    out[i] = f(in[i + 94 - 5 * ((i - 6) / 5)]); /* S8 */
  else
    out[i] = f(in[i + 94]);                /* S9 */

```

Fig. 12. S4 propagated to S9 and then removed

4 Preliminary results

Applied on our running example, advanced signal propagation results in a substantial decrease in the number of memory accesses, namely from about 20000 to 200, i.e. a decrease of 99%. Furthermore the memory requirements decrease since the temporary array `a` is no longer used.

We have used a prototype implementation to apply advanced signal propagation on a medical imaging application, namely a cavity detector[14] where several filters are consecutively applied on an image. These filters calculate a new value for each pixel from the pixels in an area around the pixel. To account for border conditions, a black border is added to the image. This is schematically shown in Fig. 13. It relieves the filter code from taking border conditions into account. We applied the advanced signal substitution from this paper to replace the reads from the border pixels by the constant value of those pixels, and this on an image of 314 to 234 pixels and with a filter of 7×7 pixels, and hence with a border of 3 pixels wide. By doing this we could remove the border from each intermediate image and hence shrink them from 320×240 to 314×234 , i.e. by 5%. The number of array accesses decreased by about 8%.

for `i` between 6 and 10, and then $(i - 6) / 5$ simplifies to 0. The reader can check that if we first solved the recursion in S2 before doing any signal propagation, then we would get the version with S8 and S9 merged.

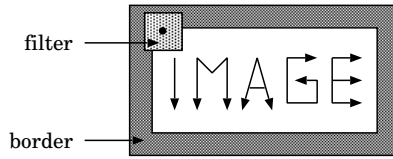


Fig. 13. Schematical view on the application of a filter to an image

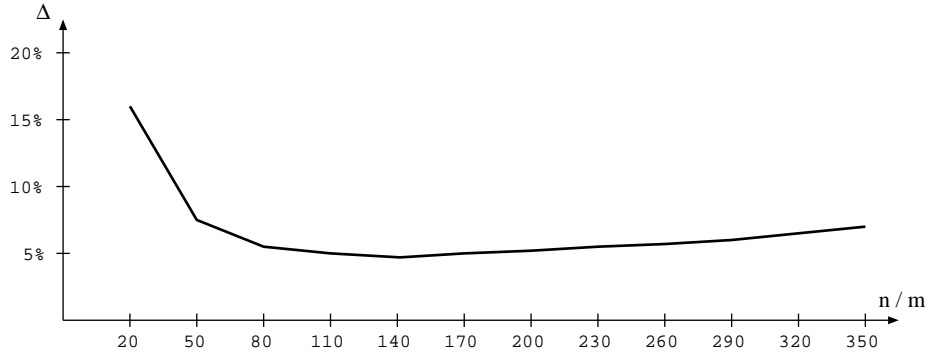


Fig. 14. The procentual decrease in memory accesses Δ_{acc} set out against the ratio of image size and image size n/m

This decrease depends on the size of the image relative to the size of the filter. Suppose the filter is square as in our example and is $n \times n$ in size, with n an odd value. The width of the border is then $(n-1)/2$. Next suppose the image is m wide. We suppose the aspect ratio is $3/4$, so the height is then $3/4 \times m$.⁹ The procentual decrease in memory accesses Δ depends only on the ratio of m and n and is set out against this ratio in Fig. 14. In our case above, we have a ratio of $314/7 \approx 45$ giving us a decrease of about 8%. Figure 14 shows that the larger the filter is compared to the image (i.e. go more to the left), the larger the gain since a larger border and corresponding memory accesses are removed. Larger filters like these are not uncommon in image processing. Figure 14 also shows that if the image size becomes about a factor 170 larger than the filter size, the decrease in accesses starts to grow again. This is mainly because then the code to initialize the image with the border gains relative importance in the number of accesses and hence removing it will give higher gains.

Reducing the size of memory and the number of accesses to it, is a positive effect of advanced signal propagation. A possible negative effect is increased complexity of code (splitting of statements, extra conditions). Part of future work is to develop cost functions that take both effects into account and to determine the optimal amount of advanced signal propagation.

⁹ The aspect ratio of most monitors is $3/4$, so this value is not so unusual.

5 Discussion

In this paper we have introduced non-recursive and recursive signal propagation as basic operations facilitating advanced signal propagation over global loop and condition scopes of imperative programs in dynamic single assignment form. We have illustrated the method on a small example and have reported on some preliminary results obtained with a prototype implementation.

The goals of our methods are similar to those of copy propagation and constant propagation[4, 1] that have been used for a long time in compilers. They often limit themselves to simple data types. In [4] an extension to arrays is presented. A dependence analysis between array references is required but is left unspecified in [4]. In [1] Muchnick gives a number of dependence tests, ranging from Banerjee’s GCD test[15] to Pugh’s Omega test[16], which allow the data-flow analysis required for copy propagation to be made more accurate for array accesses. An extension to constant propagation that takes conditional branches into account is given by Wegman and Zadeck[17], but it is limited to branch conditions that evaluate to a constant true or false. To the best of our knowledge, our work is the first attempt to automate copy propagation and constant propagation for programs with arrays by treating each instance of each statement separately, and thus allowing array propagation on an element basis. Each of the previously existing methods for copy and constant propagation consider all instances of a statement as a whole and hence they do these transformations on an all-or-nothing basis. We can also propagate over conditional statements and global loops that often hinder classic copy and constant propagation.

The intention of doing classic copy propagation is to create dead code that can then be eliminated. Dead-code elimination is described in [4, 1]. Here too compilers often limit themselves to simple data types, while technically the same dependence tests can increase the precision in case of arrays. Partial dead-code elimination is presented in [18] and allows to remove code that is dead on only part of the program paths by moving it down the paths along which the code is not dead. However the code is textually moved without changing the branching structure of the program. Again to the best of our knowledge, we are the first to automate dead-code elimination on a statement instance basis, which allows us to handle our running example while none of the previously existing methods could do so.

To apply our signal propagation, programs have to be in dynamic single assignment form. This form is also widely used in the context of parallelization [7] and design of systolic arrays [8]. In [5], Feautrier presents an automated method to convert the programs we consider into dynamic single assignment form.

The need for advanced signal propagation as developed here has been recognized in the custom memory management framework of [3]. The aim is to optimize memory size and accesses to reduce power consumption (and improve execution speed) of embedded systems. Advanced signal propagation is currently applied by hand. Our work is a first step in the automation of such transformations. In the above framework, advanced signal propagation is to be followed

by other transformations. First, loop transformations [11], preferably applied to code in dynamic single assignment form, can improve locality of memory accesses¹⁰. Methods that introduce memory reuse (destructive assignment) [9, 19, 10] can reduce the sizes of the required memories.

The further development of our prototype, together with its integration in the other steps in the framework of [3] will allow us to do more realistic measurements on the effects of advanced signal propagation on large multimedia applications.

References

1. Muchnick, S.: *Advanced compiler design & implementation*. Morgan Kaufmann Publishers, San Francisco, CA (1997)
2. Hennessy, J., Patterson, D.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA (1996)
3. Catthoor, F., Wuytack, S., De Greef, E., Balasa, F., Nachtergaele, L., Vandecappelle, A.: *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers (1998)
4. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc. (1986)
5. Feautrier, P.: Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* **20** (1991) 23–53
6. Kienhuis, B.: *Matparser: An array dataflow analysis compiler*. Technical report, University of California, Berkeley (2000)
7. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.W., Bugnion, E., Lam, M.S.: Maximizing multiprocessor performance with the suif compiler. In: *IEEE Computer*. (1996)
8. Kung, H., Leiserson, C.: Systolic arrays for vlsi. In: *Sparse Matrix Proceedings, Philadelphia:SIAM* (1978) 245–282
9. De Greef, E., Catthoor, F., De Man, H.: Memory size reduction through storage order optimization for embedded parallel multimedia applications. In: *Proceedings of the Workshop on Parallel Processing and Multimedia of the International Parallel Processing Symposium*. (1997) 84–98
10. Tronçon, R., Bruynooghe, M., Janssens, G., Catthoor, F.: Storage size reduction by in-place mapping of arrays. In Cortesi, A., ed.: *Verification, Model Checking and Abstract Interpretation, Third Int. Workshop, VMCAI 2002, Revised Papers*. Volume 2294 of LNCS., Springer-Verlag (2002) 167–181
11. Danckaert, K.: *Loop transformations for data transfer and storage reduction on multiprocessor systems*. PhD thesis, K.U.Leuven (2001)
12. Thiele, L., Artz, U.: On the synthesis of massively parallel architectures. *International journal of high speed electronics and systems* **4** (1993) 99–131
13. Quinton, P., Rajopadhye, S., Risset, T.: *On manipulating z-polyhedra*. Technical report, Institut de Recherche en Informatique et Systemes Aleatoires (1996)
14. Catthoor, F., Vandecappelle, A.: *DTSE script illustrated on cavity detection demonstrator*. Technical report, IMEC (2000)

¹⁰ First applying our transformation will likely give a better result than applying loop transformations on the original code as there are already fewer memory accesses.

15. Banerjee, U.: Dependence testing in ordinary programs. Master's thesis, Dept. of Comp. Sci., Univ. of Illinois (1976)
16. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. In: Proceedings of Supercomputing '91, Albuquerque, NM (1991)
17. Wegman, M., Zadeck, K.: Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* **13** (1991) 181–210
18. Knoop, J., Ruthing, O., Steffen, B.: Partial dead code elimination. In: SIGPLAN Conference on Programming Language Design and Implementation. (1994) 147–158
19. Quilleré, F., Rajopadhye, S.: Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages* **22** (2000) 773–815