

A Step towards a Scalable Dynamic Single Assignment Conversion

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

Report CW 360, April 2003



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

A Step towards a Scalable Dynamic Single Assignment Conversion

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

Report CW 360, April 2003

Department of Computer Science, K.U.Leuven

Abstract

In this report we present two new ways to transform a program to dynamic single assignment form or DSA form. The importance of DSA form is that all restrictions on reordering statements due to memory reuse have been eliminated, thus allowing advanced memory optimizations required for multimedia applications to do a better job without a need to revert to more complex analyses. Though methods exist to do the transformation to DSA, they are both limited in applicability and scalability. To overcome these problems we developed two methods to do this conversion and for both the essential difference with existing methods is that we add copy operations to simplify the data flow, thus simplifying the DSA conversion considerably. However we cannot just add copy operations because that defeats the purpose of optimizing the memory use of a program. To solve this problem we present an advanced copy propagation method that removes copy operations such that we can obtain the DSA form of the program that we aim for. An added advantage of this method is that we can remove copy operations selectively, thus leaving some in in favor of lower complexity of the resulting program.

Contents

1	Introduction	1
2	Motivation	2
2.1	Context : why is there a problem?	2
2.2	A solution: Data Transfer and Storage Exploration	3
2.3	Dynamic single assignment	8
2.4	The why of pruning	11
2.4.1	Increase the explicit freedom for optimizations	11
2.4.2	Speed up DTSE	16
2.5	Why automate pruning?	16
3	Preliminaries	18
4	Dynamic single assignment	21
4.1	Reprise definition	21
4.2	Static single assignment	24
4.2.1	Principles	24
4.2.2	Extension for arrays	25
4.3	Feautrier’s method	26
4.3.1	The method itself	26
4.3.2	Improvements to the method	29
4.3.3	Discussion	29
4.4	Extension static single assignment to dynamic single assignment	30
4.4.1	Scalars	31
4.4.2	Arrays	33
4.5	A second method for dynamic single assignment conversion	34
4.5.1	Array static single assignment	35
4.5.2	Adding dimensions	41
4.5.3	Discussion	49
5	To DSA or not to DSA?	50
5.1	Inplace mapping	51
5.2	Data flow transformations	52
5.3	Discussion	55
6	Conclusion	56
A	Overview of unhandled constructs	57

1 Introduction

In this report we present our basic methods that allow us to develop an automated conversion of a program into a dynamic single assignment (DSA) form. Conversion to single assignment is a well-studied topic in the context of compiler technology. The reason we need DSA is that it is able to deal in a precise way with programs that manipulate array elements. We need to deal with this kind of programs because conversion to DSA is actually one of the enabling preprocessing steps of DTSE. DTSE is a methodology for optimizing data-intensive programs for power efficiency. Multimedia applications are a typical example where DTSE has proven its benefits.

This report has several purposes. First we provide some background in DTSE and its application domain, we explain the role of DSA conversion in this context, and we motivate why the automation of DSA conversion is desired. Then we discuss existing basic forms of single assignment. An important aspect is their treatment of arrays. We also discuss the method of Feautrier for DSA conversion and its shortcomings when being used in the context of DTSE. We sketch two ways for feasible DSA conversion. Our first proposal is an extension of the static single assignment form. The second proposal is based on the observation that there are two kinds of overwriting of memory elements and removes them in two subsequent steps.

Finally we present our ideas for some form of copy propagation in the context of arrays. This method is of interest here because it is able to remove redundant copies of array elements. If we know that they can be removed (in an automatic way), we can allow them to be introduced during DSA conversion and that is exactly the case in our proposals.

The main contribution of this report are the proposals for feasible DSA conversion and the advanced copy propagation for arrays. These approaches are explained in terms of examples. The detailed elaborations are not the subject of this report.

The report is structured as follows. Section 2 provides some background on DTSE, the role of DSA within DTSE, and the motivation for an automatic conversion. We discuss some preliminary material in Sec. 3. Single assignment is discussed in Sec. 4, together with our two proposals for automating DSA conversion. The advanced copy propagation for arrays is the subject of Sec. 5.

2 Motivation

In this section we argue that power consumption and heat dissipation of memories is an important problem in the context of data-intensive applications. The Data Transfer and Storage Exploration (DTSE) methodology aims at optimizing the power consumption by transforming the program into one with a better memory use. One of the enabling steps for DTSE is the conversion to dynamic single assignment (DSA). Converting a program to DSA makes the data flow of the program explicit; every memory element is written (produced) only once during the execution of the program and statements that read (consume) the values of those memory elements can be linked with the single producer of those values. We also elaborate on reasons why DSA conversion and pruning are effective for DTSE and why automation is desired.

This section is structured as follows; Sec. 2.1 explains why power consumption of memories is becoming a more and more important issue. The memory use of programs can be optimized for lower power consumption by the Data Transfer and Storage Exploration (DTSE) methodology, which is briefly sketched in Sec. 2.2. Section 2.3 defines dynamic single assignment (DSA) and explains that the role of DSA in the context of DTSE is to make the data flow in the program explicit. Some of the reasons to have the pruning step are explained in Sec. 2.4 and the motivation for automating it is given in Sec. 2.5.

2.1 Context : why is there a problem?

We live in a world nowadays where the Internet starts to play a central role. The Internet is ever-expanding and the amount of data that travels the earth becomes huge. All this data will be used and processed in due time. This has a number of consequences.

Let us start at the server machine that gets a request for certain data from another machine. As servers become more intelligent, their job is not only to send the data. Sometimes processing is needed. A simple example of this is a web server that keeps the pages it offers in a compressed file that needs to be decompressed before sending. Web servers also compress web pages before sending to reduce bandwidth. Another example is a web server that keeps its data in a database and when a web page is requested it needs to generate it from the database. Even the use of secure, encrypted connections requires a lot of processing on the server side.

The next step is getting the data over the network to another machine. The data passes by a number of routers that know which way to send the data to get it to its destination. Those routers have to accept loads of data and send it further in the correct direction. To find the correct direction, routers keep big tables that are growing as the Internet is growing and they need to do lookups in those tables. Efficient access is important. Also routers become smarter too in the sense that they do not just send the data anymore but also interpret part of the data to work more efficiently, but this requires complexer calculations and a lot more data accesses.

Once the data has reached its destination, it has to be processed. A large part of the data that travels the Internet these days is multimedia. Video, sound, games and animated pictures, the possibilities are endless. But as technology evolves, algorithms that use these

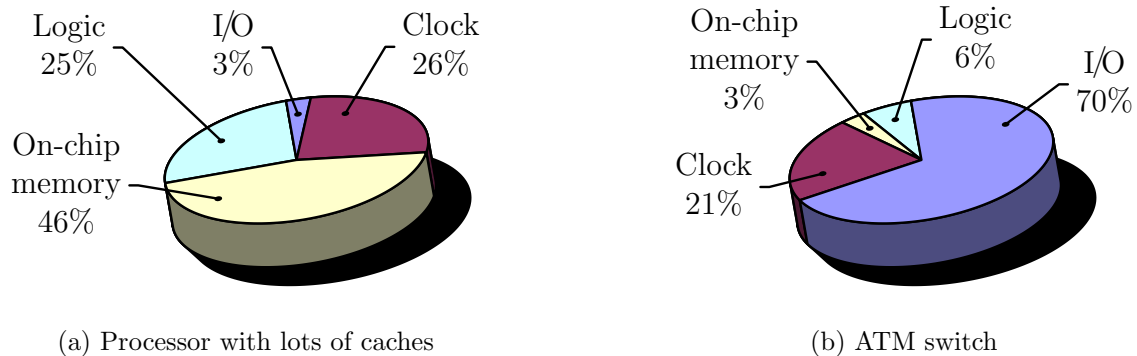


Figure 1: Power usage in a few example systems [CV00]

data become complexer and the amount of data processed is growing larger.

People do not only want to access the huge data pile, they also want to access it from anywhere on earth (and soon in space too?). That is why the hand-held devices become so very popular (it is even said that one day general-purpose computers will be replaced by those small dedicated devices). These hand-held devices like (WAP-)GSMs, portable mp3-players, portable cd-players and so on are almost without exception embedded systems. Those hand-held devices also typically work on batteries, and this is where the problem comes into the picture.

When we look at the systems used to process these loads of data, we see that a big part of the power consumption is due to memory (or at least in the digital part of the system). In Fig. 1, we see the power consumption of a few systems divided over the different parts of the system. As we can see in Fig. 1(a), in a processor with lots of caches almost half of the power consumption is due to the memory. In Fig. 1(b) it is even worse: almost three quarters of the power consumption lies with memories, where the power consumption in external memories is reported as I/O. This is a problem because the power consumption of memories increases with the frequency of the memory accesses and since we want to process more and more data at a higher speed, we soon use too much power. This is a problem for portable devices that work on batteries but also for other systems where for example heat dissipation is an issue.

Since it is not possible to increase the power efficiency of memories as much as we want or need, and since the problem of power usage and heat dissipation seems only to increase, we need a solution to decrease the power usage of the memories in another way than optimizing the hardware only.

2.2 A solution: Data Transfer and Storage Exploration

A possible methodology to overcome the problem of high power consumption, is Data Transfer and Storage Exploration or DTSE [CWD⁺98]. DTSE has as input a C program and as output a transformed C program and a memory hierarchy to go with it. The aim is to reduce the power consumption of the memories.

DTSE consists of several orthogonal steps. This means that each step can in principle

be done independently of the following ones. As such it is possible to focus on that one single step without having to worry about the subsequent steps. The major advantage of this is that it allows to work on one single step and spend all of one's effort on elaborating or automating it without the complexity of managing the whole methodology.

The DTSE methodology is explained in detail in [CWD⁺98]. A short summary is given here. DTSE has 3 main phases: (1) the pruning phase preprocesses the program to enable the actual optimization phase, (2) the DTSE methodology that optimizes the preprocessed program for power consumption and derives an accompanying memory hierarchy and layout, (3) de pruning and address optimization to remove the overhead introduced by DTSE. We give more detail about the pruning step, because the focus of this paper is dynamic single assignment which is part of that step.

1. Pruning

The goal of this step is to make DTSE possible on the program given as input. The general idea is to remove certain constructs (*e.g.*, pointers) and introduce extra properties (*e.g.*, dynamic single assignment). The different sub-steps are described in more detail.

(a) Division in layers

The original program is rewritten into 3 layers. The layers are as follows:

Layer 1 This layer contains the top-level process flow. It is irrelevant for DTSE because it just determines what is called in layer 2 and in which order. The division is such that no significant optimization is possible between the different calls to layer 2 so the calls can be handled separately.

Layer 2 This layer primarily contains loops and indexed arrays. This is the part that DTSE tries to optimize. We try to keep layer 2 free of all constructs that are irrelevant to DTSE so that DTSE need not bother with it. Since the use of loops and arrays leads to the largest memory use, DTSE focuses on exactly that.

Layer 3 This layer contains arithmetic operations, logic and data-dependent operations. DTSE does not need to know what the operations are that are involved, it only needs to know what data is used (with as exception data flow transformations). Hence we can hide these operations in a separate layer.

So everything that can be optimized by DTSE is put in layer 2 and DTSE only looks at this layer.

(b) Hide undesired constructs in layer 3

Some constructs like data-dependent `ifs` and `while`-loops are hidden in layer 3. If these constructs cannot be analyzed, DTSE cannot optimize them and there is no need for DTSE to do even the effort of just passing by it, so we hide it. First look at a simple example. Suppose we have a piece of code as in Fig. 2(a). The condition (`a[i] > b[i]`) is data-dependent and hard to analyze. Consequently, DTSE can only assume that the value of either `a[i]` or `b[i]` is read, and that a value is surely written to `max`. We can approximate this by saying that both `a[i]` and `b[i]` are read. This is on the safe side for DTSE, but it limits what DTSE

```

if (a[i] > b[i])
    max = a[i];
else
    max = b[i];

```

(a) Condition is visible

```

max = maximum(a[i], b[i]);

```

(b) Condition is invisible

Figure 2: Hiding data-dependent conditions : maximum calculation

could do if it could analyze the condition. So we can transform the piece of code in Fig. 2(a) to the code in Fig. 2(b). Here the implementation of the function `maximum` is hidden in layer 3 and not visible to DTSE.

(c) **Selective function inlining**

The classic idea of function inlining is removing the overhead of a function call by replacing the call site with an instance of the body of the called function. This of course goes at the expense of an increase in code size and hence possibly a worse cache behavior that may destroy all benefits of the inlining. That is why compilers rather stick to selective inlining, often driven by user directives.

In the context of DTSE we want to minimize the power consumption of the memories used by our program. For that, DTSE needs to do transformations and function calls limit the freedom to do so (this freedom is called exploration freedom). To this end, we inline functions where it has the most effect on the freedom for transformations or where transformations can have the largest effect on the power consumption.

(d) **Pointer conversion and dynamic single assignment**

The first intention in this step is to make explicit all the freedom that is available for transforming the given program. One may also argue that it actually increases the freedom, but the aim is rather to relieve DTSE from searching for the implicit freedom in the program over and over again. The second reason to do this step is to make transformations easier to perform. Pointer conversion does this by making the addressing of arrays explicit from the implicit addressing done by pointers. Dynamic single assignment does this by making the data flow explicit and by decreasing the number of dependencies that limit the transformations of the program. Basically dynamic single assignment means that once a value is written to a memory location, it is never overwritten so we can regard memory locations as values which is very convenient for transformations. Dynamic single assignment is the focus of this report and is discussed in detail later on.

(e) **Pruning data flow chains**

Sometimes programs have data flow chains where one link can be removed (or rather collapsed) without limiting exploration freedom for DTSE. A simple example of this is shown in Fig. 3(a). The data flow chain consists here of data flowing from array `a` to array `b` through function `f` and then from array `b` to array `c` through function `g`, and all this in a point-wise fashion. This means we can

<pre> for (i = 0; i < N; i++) b[i] = f(a[i]); for (j = 0; j < N; j++) c[i] = g(b[i]); </pre>	<pre> for (i = 0; i < N; i++) c[i] = g(f(a[i])); </pre>
(a) A data flow chain	(b) Collapsing the chain

Figure 3: Pruning data flow chains

simply merge the two loops as in Fig. 3(b) without losing any freedom. Here the temporary buffer `b` is no longer needed. The consequence is that our program becomes more concise so that DTSE has less work to do and operates faster without losing accuracy.

(f) **Weight based removal**

Since we are trying to optimize for power consumption, it has a small effect to consider small arrays or arrays that are accessed rarely because power consumption of memories increases with their size and the frequency of the accesses. If we hide these arrays from DTSE such that it does not have to take them into account, it saves us time.

(g) **Graph partitioning**

This is an instance of the good old divide&conquer technique. We cut our program in smaller pieces that can be handled much faster (and because a program can be represented by a data flow graph, this is called graph partitioning). Because a program typically takes input in the beginning and produces output at the end, cutting the program in pieces also cuts off the data flow between those pieces. This means that we need buffers in between in the form of arrays and good candidates for this are the arrays that occur in the program. Because larger arrays offer more opportunity for useful optimizations than smaller arrays, we only cut at the latter.

So the general idea of pruning is to make sure DTSE can work in optimal conditions, *i.e.*, it has to carry along as little unnecessary overhead and it gets all the freedom it needs. Note that steps 1a through 1e increase the exploration freedom for DTSE or at least do not change it, while step 1f and 1g actually decrease that freedom to have DTSE attain better run time. Because of this, these last two steps should be performed very carefully.

2. DTSE

After the original program has been properly prepared and transformed, DTSE can be set loose upon the transformed program. This is where the real optimizations and transformations begin. The goal of this step is to arrive at a transformed program, a memory hierarchy and a placement of the arrays in the transformed program in the new memory hierarchy. We briefly enumerate the different steps and we only go deeper into a few steps that are relevant for us.

<pre> for (i = 0; i < N; i++) { b[i][0] = 0; for (j = 0; j < M; j++) b[i][j+1] = b[i][j] + a[i][j]; } </pre>	<pre> for (i = 0; i < N; i++) { for (j = 0; j < M; j++) b[i][j+1] = (j == 0 ? 0 : b[i][j]) + a[i][j]; } </pre>
(a) With explicit initialization	(b) With propagated initialization

Figure 4: Advanced copy propagation applied to an example

(a) **Data flow transformations**

Data flow transformations are – as the name says – transformations that change the data flow of a program. So what does it mean for the data flow to change? A simple way to formulate this is by saying that the algorithm itself changes and not just the implementation. One could state that the algorithm can be seen as the mathematical formulation of the calculation – independent of variable names – while all the details of the allocation of variables and the scheduling of all the operations (*i.e.*, the order in which they are executed) belong with the implementation.

A type of data flow transformation that is important in the light of this report is called advanced copy propagation. The idea here is to remove the use of an extra array by substituting the production of the array into the consumption. Consider the simple example in Fig. 4(a) where the sum of the rows of a matrix **a** is calculated. We have an initialization of part of **b** to 0. The values in **b** are used in the inner loop and so we substitute the initialization into it to get the program in Fig. 4(b). This way we succeed in avoiding the extra initialization and allocation of part of **b**.

(b) **Loop transformations**

Loop transformations try to change the definitions of the loops. The aim is to increase the regularity this way as well as the locality of reference such that the subsequent DTSE steps can take advantage of this in their optimizations. Examples are loop unrolling, loop merging and loop reversal.

(c) **Data reuse**

In a program values are produced just once. However values can be consumed multiple times or in other words data can be reused. So this step copies parts of arrays to smaller arrays that can be put in smaller memories that in turn require less power and are faster.

(d) **Cycle budget distribution**

In this step a Pareto curve is created that allows the designer to find the best memory allocation and assignment and the according power consumption given a certain memory cycle budget. It allows the designer to choose the best solution given constraints on *e.g.*, maximum power consumption.

(e) **Memory allocation and assignment**

In this step the actual layout of the memories is determined, such as the number of memories, the bit width, the size and the number of read and write ports.

(f) **Data layout optimization**

At this point in the DTSE script, we still have a program in dynamic single assignment form. However mostly only a fraction of the values in an array is live so we only need enough room to store that fraction. We call this step in-place mapping and it is more or less the inverse of conversion to dynamic single assignment.

(g) **Data layout for caching**

This step tries to decrease the number of cache misses. Since previous steps already result in a good locality of reference, the focus is on decreasing capacity and conflict misses.

The thread through all these steps is optimizing the program for power consumption and after all previous steps, this minimum should have been reached. However by focusing on power consumption only, we introduced quite a lot of overhead in terms of the execution of the program and this overhead is removed by the rest of the DTSE script.

3. Depruning and address optimization

The final step in DTSE eliminates the major overhead introduced by DTSE itself. One example of this is to undo function inlining to decrease the code size again which is part of the depruning. Another example is address optimization or ADOPT that removes the addressing overhead because for one thing DTSE introduces lots of modulo expressions that are expensive to calculate. After this step so much of the overhead disappears that as an extra to optimization for power, the program also runs faster than before.

To summarize the DTSE script consists of three parts. The first part is called pruning and it enables the actual DTSE. The second part is the actual DTSE that optimizes the given program for power consumption and derives an accompanying memory hierarchy and layout. The third part removes all the overhead introduced by DTSE so that the run time of the program improves greatly. Transformation to dynamic single assignment belongs in the pruning step of DTSE.

2.3 Dynamic single assignment

In this section we zoom in on dynamic single assignment and explain what it means for a program to be in dynamic single assignment form. We also explain the effect of dynamic single assignment conversion on a program, *i.e.*, how conversion to dynamic single assignment removes restrictions on changing the order of execution in a program. We start with the definition of dynamic single assignment form.

Definition A program is in dynamic single assignment form when during the execution of the program only a single assignment happens to each memory element.

<pre>for (i = 0; i < 10; i++) a = i * (i + 1) / 2;</pre>	<pre>for (i = 0; i < 10; i++) a[i] = i * (i + 1) / 2;</pre>
(a) Multiple assignment code	(b) Dynamic single assignment code

Figure 5: Example of transformation to dynamic single assignment

There are three elements in this definition. The *first* is that we have only *a single assignment*. This means that no other assignment overwrites a value written by that single assignment and so we can identify the producer of a value very easily. The *second* element is that we want this to be true *during execution*. Suppose we have a single assignment to a variable in the program text, but this assignment is placed within a loop that is executed say 10 times. In that case we have 10 instances of the assignment instead of a single one and the code is not single assignment. Finally the *third* element of the definition is that we look at *each memory element* separately. It is obvious that we can look at *e.g.*, an integer as being one atomic memory element that is always read and written as a whole. Arrays however consist of multiple elements and these can in one operation be written separately without touching any of the other elements. Thus we regard those array elements as separate memory elements.

What makes this definition interesting is the observation that an assignment to an array uses an indexation function to select an array element. Hence for every iteration over a statement in a loop, the element we assign a value to can vary depending on the indexation function.

A look at an example will make this a bit clearer. The program in Fig. 5(a) contains only one assignment. This program however does not satisfy our definition of dynamic single assignment. Due to the loop, the assignment to scalar variable¹ `a` is executed 10 times so we have more than one run time assignment to `a`. This does not mean that an assignment appearing in a loop is never in dynamic single assignment form. Look at the program in Fig. 5(b) where `a` has magically been transformed into an array. We see that the assignment to array `a` is executed 10 times too, but this time we do not write the same memory element each time. Instead we write to 10 different elements of `a` and so according to the definition above, we have dynamic single assignment!

This settles the question of what dynamic single assignment means at the code level. In the remainder of this section we describe what the effect is of dynamic single assignment. While storing multiple values in a variable is convenient for programmers – why else call them variables? – it is not always ideal for optimizations. The reason for this is that multiple assignment code obfuscates the data flow. The data flow is concerned with values, but programs use variables that can contain multiple values. Thus program optimizations (like the ones done by DTSE) need to unravel memory locations into the multiple values they contain. This way, they can ensure that the flow of values (or data flow) remains the same, even though the placement of these values in memory changes. This disambiguation of values can be done by transforming the program to dynamic single assignment which allows

¹As opposed to array variable

<code>t = f1(x);</code>	<code>t = f1(x);</code>	<code>t1 = f1(x);</code>	<code>t1 = f1(x);</code>	<code>t1 = f1(x);</code>
<code>a = f2(t);</code>	<code>t = g1(y);</code>	<code>a = f2(t1);</code>	<code>t2 = g1(y);</code>	<code>t2 = g1(y);</code>
<code>t = g1(y);</code>	<code>a = f2(t);</code>	<code>t2 = g1(y);</code>	<code>a = f2(t2);</code>	<code>a = f2(t1);</code>
<code>b = g2(t);</code>	<code>b = g2(t);</code>	<code>b = g2(t2);</code>	<code>b = g2(t2);</code>	<code>b = g2(t2);</code>

(a) Code 1 (b) Code 2 (c) DSA code 1 (d) DSA code 2 (e) DSA code 3

Figure 6: Two similar examples

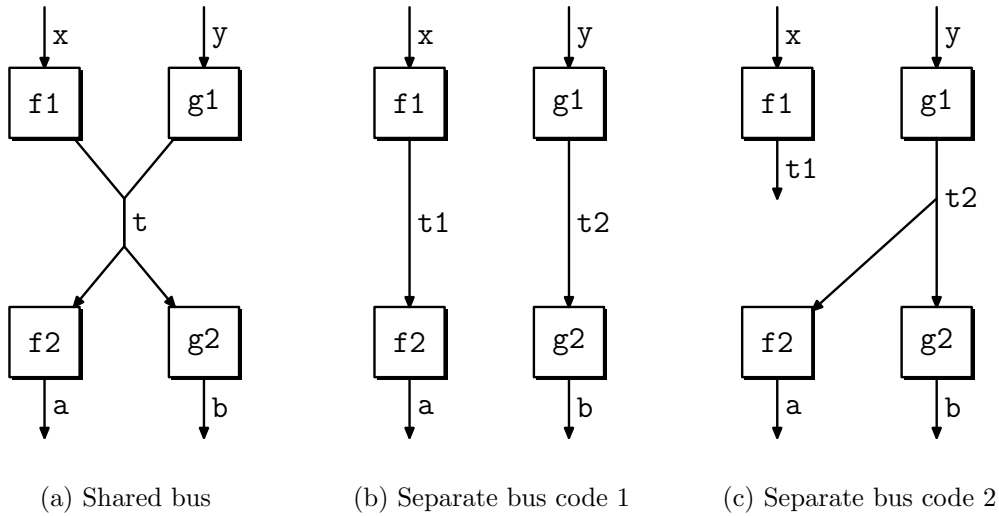


Figure 7: The advantage of dynamic single assignment

only a single value to be contained in a memory element, thus eliminating any ambiguity and making the data flow explicit: there is one assignment instance that produces a value so there is a unique assignment instance that writes the same array element as the statement instance that reads that array element.

We illustrate this with a simple example. The example in Fig. 6(a) is clearly not in dynamic single assignment form since there are two assignments to `t`. Here `f1`, `f2`, `g1` and `g2` represent any calculation with no side effects. We can thus represent them as black boxes with one input and one output. *E.g.*, `f1` is a black box with as input the value of `x` and which outputs the result to `t`. We can represent both `x` and `t` as data buses, and calculations can get their values from those buses and put their results on other buses. For Fig. 6(a), this representation as black boxes and data lines is illustrated in Fig. 7(a).

At the end of this program `a` has the value `f2(f1(x))` and `b` has the value `g2(g1(y))`. If we now try to reorder the computations, we might get something like Fig. 6(b). This is a valid C program, but it no longer computes the same values. Taking a closer look, we see that `a` ends up with `f2(g1(y))` in it and `b` with `g2(g1(y))`. So what went wrong with the transformation? Well, in this case have two distinct calculations. They are however coupled

because we share the memory location of \mathbf{t} in the two calculations. We can look at $\mathbf{f1}$, $\mathbf{f2}$, $\mathbf{g1}$ and $\mathbf{g2}$ as computation nodes and at the variables \mathbf{x} , \mathbf{y} , \mathbf{a} , \mathbf{b} and \mathbf{t} as data buses where \mathbf{t} is a shared bus. So the first program above is represented as in Fig. 7(a). Here we see that the output of $\mathbf{f1}$ and $\mathbf{g1}$ is put on the shared bus \mathbf{t} . Since we share the bus, we have to take into account a certain bus protocol to have everything go according to plan. In this case this means that if we put a value on the bus, we need to take it off before we put another value on it because otherwise the first value will be lost. Conclusion is that bus sharing, or – in the non-abstract case – memory location sharing, constrains the order in which we can do our computations because of this bus protocol.

In the example above we broke the protocol by putting $\mathbf{g1}(\mathbf{y})$ on the bus before $\mathbf{f1}(\mathbf{x})$ was taken off the bus. The solution to this is very simple: do not share the bus! This is depicted in Fig. 7(b) where bus \mathbf{t} is split into two separate buses $\mathbf{t1}$ and $\mathbf{t2}$. The program that goes with this situation is shown in Fig. 6(c). This program is now in dynamic single assignment form. For the version in Fig. 6(b) the bus structure is as in Fig. 7(c), with the code shown in Fig. 6(d). The bus structure makes it very clear that the reordering of statements in Fig. 6(b) changes the values \mathbf{a} and \mathbf{b} end up with.

If we now do the same reordering as before for the dynamic single assignment version in Fig. 6(c), we get the program in Fig. 6(e). After execution of this code fragment, \mathbf{a} contains $\mathbf{f2}(\mathbf{f1}(\mathbf{x}))$ and \mathbf{b} contains $\mathbf{g2}(\mathbf{g1}(\mathbf{y}))$, just as it should be. We have split the shared bus in two non-shared buses that can work in parallel. Since this way no bus protocol needs to be taken in account, we get a lot more freedom. And in case of full dynamic single assignment we have no bus protocol at all so we have all the freedom we can get.

Although this example was about scalar variables, it applies to arrays as well. We can look at arrays as an array of buses where each bus in the array represents a single element of the array. Sharing of this array of buses happens for each bus in the array separately. Although we can look at bus splitting at the level of the whole array, it is much more accurate to do this at the level of the separate buses in the array. Or in other words we have to look at the array at the element level to do the maximum value disambiguation possible and to thus remove all limitations on statement reordering due to memory reuse.

2.4 The why of pruning

In Sec. 2.2 we stated that DTSE needs pruning because it actually ‘enables DTSE’. Here we elaborate on this statement some more with focus on dynamic single assignment. In general the aim of pruning is twofold: increasing the explicit freedom for optimizations – discussed in Sec. 2.4.1 – and speeding up transformations – discussed in Sec. 2.4.2.

2.4.1 Increase the explicit freedom for optimizations

In this section we argue that there is an interaction between the way values are allocated to memory and the execution order of the different instances of statements. On the one hand, if we want to optimize the memory allocation, it is possibly necessary to change the execution order. But on the other hand, if we want to change the execution order, it may be necessary to change the data allocation as well. As it turns out, any valid change in execution order in a program – even if it requires a change in data allocation – is valid for the dynamic single

assignment version of the program without any change in data allocation. This allows to concentrate on changing the execution order without having to worry about retaining the data flow of the program. This is perceived as explicit freedom for reordering optimizations that typically do not change the data allocation to make the reordering possible.

In the remainder of the section we first explain what the data flow of a program is, and how the addition of a specification of the execution order and the data allocation makes for a complete executable program. Then we show that dynamic single assignment is sometimes needed to change the data allocation sufficiently for optimization to be able to attain an optimal solution.

We can consider an instance of a statement as an operation that has a number of values as arguments and produces new values calculated from its arguments. The arguments of such a statement instance are either produced by other statement instances or given as input to the program. The latter are thus not produced by any statement instance within the program. Likewise the values produced by statement instances are either passed on as arguments to other statement instances or belong to the output of the program. The description of all statement instances together with the information about how the values are passed on between statement instances, is called the data flow of a program. Although it is a complete description of the way the results are computed, it abstracts two aspects of a program that are necessary to be able to execute the calculation on (most) processors. Firstly the order of execution of the different statement instances is important since most processors require that you specify a sequential order. Secondly there is the allocation of the different values to memory since most processors have no notion of a value, but they do have a notion of memory locations.

Allocating values to memories boils down to having statement instances store values in certain memory locations and having other statement instances retrieve the value from those memory locations. It is not necessary that each value is allocated to different memory locations because a value can be overwritten when it is no longer needed, *i.e.*, when all statement instances that use the value as an argument have retrieved the value from the memory location. We say that a value is live between the time it is produced by a statement instance and the time it is consumed by all statement instances that have the value as argument. So we cannot allocate two values to the same memory locations if they are live simultaneously. Because of this and the fact that the order of execution clearly determines when a value is live, there is an interaction between the order of execution and the data allocation.

Let us look at the piece of code in Fig. 8(a). **S1** produces a value that is immediately consumed by **S2** in the same iteration of the loop. Afterwards the value is no longer used and hence the value is live only for a very short time. The next value produced by **S1** is produced in the following iteration and is only live in that iteration. We can say that the value is private to the iteration in which it is produced; it is not shared between iterations. Hence the liveness of all these values do not overlap and we can use a single, simple variable to store the values.

Suppose that we need to split the loop to be able to do some optimization. There are two statements in this loop, and so we want to place each in its own loop so that all instances of **S1** are executed before all instances of **S2**. If we would just do this – disregarding the

```

for (i = 0; i < N; i++) {
  b = f(a[i]); // S1
  c[i] = g(b); // S2
}

```

(a) An example

```

for (i = 0; i < N; i++)
  b = f(a[i]); // S1
for (i = 0; i < N; i++)
  c[i] = g(b); // S2

```

(b) Invalid loop split

```

for (i = 0; i < N; i++) {
  b[i] = f(a[i]); // S1
  c[i] = g(b[i]); // S2
}

```

(c) Changed allocation

```

for (i = 0; i < N; i++)
  b[i] = f(a[i]); // S1
for (i = 0; i < N; i++)
  c[i] = g(b[i]); // S2

```

(d) Valid loop split

Figure 8: An example to illustrate the interaction between execution order and data allocation

question whether this changes the values that end up in array `c` or not – we get the program in Fig. 8(b). It is easy to see that this program is no longer equivalent to the original program in the sense that the values stored in array `c` after executing the code fragments are not the same for both fragments. The reason for this is that we extended the liveness of all values and had those livenesses overlap, but still we used a single variable for storing those values. At the beginning of the second loop, all values produced by `S1` are live since none have been consumed yet by `S2`. Hence we need to allocate enough storage for all these values, which results in the code in Fig. 8(c). Now we can split the loop, as in Fig. 8(d). We can conclude that we cannot just split the loops unless we change the data allocation as well.

However the converse can also be true. In the code fragment in Fig. 8(d), we cannot just compact array `b` to a single variable, because then we would get the illegal, intermediate version of the code again. If we want to do the compaction, we need to merge the loops again. Hence we can conclude that it is also not always possible to change the data allocation without changing the execution order. This is exactly the reason why DTSE needs to do loop transformations to be able to reduce memory size.

Note that if we look at the original version of the code above, it looks more efficient than the transformed code: there is only a single loop, so we do not have the overhead of the extra loop and we do not need to allocate a complete array but instead we can use a single register, and depending on what expressions `f` and `g` represent some other optimizations may become possible. However this is a local view on the problem, and if the code fragment is part of larger code, the picture may very well change. It is possible that to compact an array, we need to do a loop transformation, and to be able to do that loop transformation, we need to change the data allocation for another array. This means that to decrease the size of one array, we possibly need to increase the size of another array, which seems counter-intuitive. Instead of changing the data allocation every time a loop transformation requires so, we start by converting the program to dynamic single assignment form because in that form

```

for (x = 0; x < N * M; x++) // loop nest 1
  for (y = 0; y < N * M; y++)
    im1[x][y] = in();
for (x = 0; x < N; x++) // loop nest 2
  for (y = 0; y < N; y++) {
    sum = 0;
    for (i = 0; i < M; i++)
      for (j = 0; j < M; j++)
        sum = sum + im1[x * M + i][y * M + j];
    im2[x][y] = sum / (M * M);
  }
for (x = 0; x < N; x++) // loop nest 3
  for (y = 0; y < N; y++)
    for (i = 0; i < M; i++)
      for (j = 0; j < M; j++)
        im3[x * N + i][y * N + j] = im2[x][y];
for (x = N * M - 1; x >= 0; x--) // loop nest 4
  for (y = N * M - 1; y >= 0; y--)
    out(im3[x][y]);

```

Figure 9: An example of image processing

there are no loop transformations impediments arising from memory reuse because there is no such thing as memory reuse in a dynamic single assignment program.

We illustrate this using the code fragment in Fig. 9. First a two-dimensional image is read, then it is scaled down by a factor M in both dimensions, next it is scaled up again by the same factor creating a blocking effect² and finally the image is output while mirroring it around its center point (so the upper-left point and the lower-right point are swapped and the upper-right and lower-left point too).

Chances are that the above code would originally be separated in three functions: one to read the input, one to do the blocking effect and one to do the output. In that case, it is possible to optimize the blocking code by merging the two loops and replacing the temporary array `im1` by a single scalar. The code would then look as in Fig. 10. Let us assume that this is the code presented to DTSE. Without changing data allocation, DTSE basically has two interesting options, namely merge the middle loop nest with either the first or the third loop nest. Note that merging with the first loop nest requires that the second loop nest be left as is, while merging with the third loop nest requires that the second loop nest be reversed. Because of this, we cannot merge all three loop nests. But for either option, we save an $(N * M) \times (N * M)$ image, leaving a single $(N * M) \times (N * M)$ image, so we saved 50% in memory usage compared to Fig. 10.

However we can do a better job by undoing the merge of the middle loops. Then the code presented to DTSE is the one in Fig. 9. Now DTSE has an extra interesting option to

²This effect is sometimes used on television to block out the face of people who want to remain anonymous.

```

for (x = 0; x < N * M; x++) // loop nest 1
  for (y = 0; y < N * M; y++)
    im1[x][y] = in();
for (x = 0; x < N; x++) // loop nest 2+3
  for (y = 0; y < N; y++) {
    sum = 0;
    for (i = 0; i < M; i++)
      for (j = 0; j < M; j++)
        sum = sum + im1[x * M + i][y * M + j];
    im2 = sum / (M * M);
    for (i = 0; i < M; i++)
      for (j = 0; j < M; j++)
        im3[x * N + i][y * N + j] = im2;
  }
for (x = N * M - 1; x >= 0; x--) // loop nest 4
  for (y = N * M - 1; y >= 0; y--)
    out(im3[x][y]);

```

Figure 10: Code from Fig. 9 with the middle loop nests merged

```

for (x = 0; x < N; x++) // loop nest 1+2
  for (y = 0; y < N; y++) {
    sum = 0;
    for (i = 0; i < M; i++)
      for (j = 0; j < M; j++)
        sum = sum + in();
    im2[x][y] = sum / (M * M);
  }
for (x = N * M - 1; x >= 0; x--) // loop nest 3+4
  for (y = N * M - 1; y >= 0; y--)
    out(im2[x / M][y / M]);

```

Figure 11: Code from Fig. 9 with two sets of loop nests merged

merge the first and second loop nest as well as the third and fourth. The resulting code is shown in Fig. 11. Note that to merge the third and fourth loop nest, we implicitly adjusted the loop structure of the third loop nest to match that of the fourth loop nest (we could also have done vice versa). This includes the reversal of the third loop nest, but this was allowed now because of the different data allocation for `im2`.

Now we need an image with a size of only $N \times N$. Depending on the value of M , this can be quite a bit smaller than the $(N * M) \times (N * M)$ we need when we did not transform `im2` into an array such that `im2` is dynamic single assignment. The moral of the story is that to be able to attain the best data allocation, we possibly need to destroy other optimizations relating

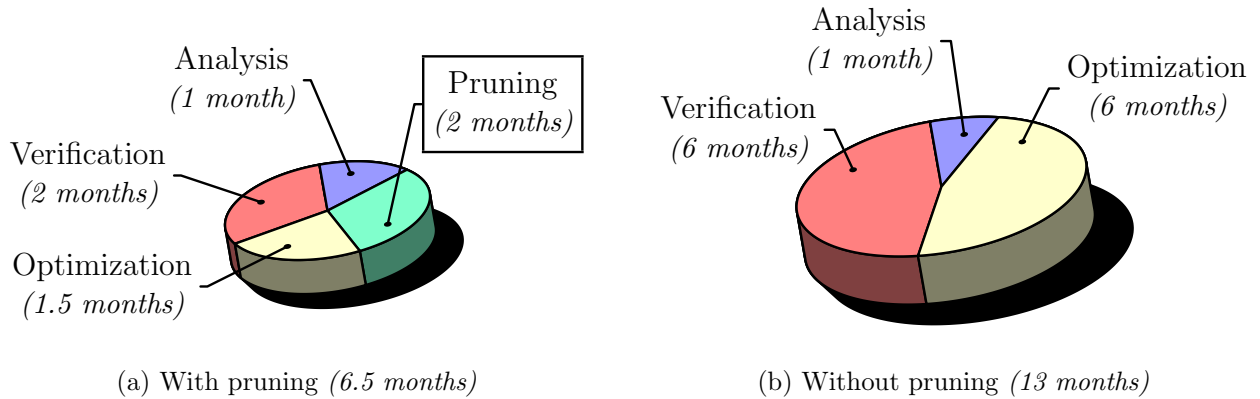


Figure 12: Time in man months for DTSE for MPEG-4 optimization [CV00]

to data allocation by making the program dynamic single assignment because memory reuse in the data allocation can hinder some essential loop transformations.

2.4.2 Speed up DTSE

Another goal of pruning is speeding up DTSE. This goal is achieved in several ways. The first and most obvious way is by hiding constructs from DTSE that are of no interest to DTSE. This can either be the abstraction of the actual operations so that only what is read and written remains, or the hiding of constructs that DTSE cannot analyze and needs to assume the worst case for anyway. An example of the latter is `while` loops. These things are done mostly in steps 1a and 1b of the DTSE script.

If it is put this way, it sounds logical that it leads to a speedup. In practice however this leads to a significant speedup. In Fig. 12 we see the time (in man years) needed to manually complete the entire DTSE methodology for an MPEG4 encoder (200.000 lines of code) in two cases. On the left the division over time is depicted for DTSE with pruning and on the right for DTSE without pruning. On the left we can see that we spend a substantial amount of the total time on pruning – almost a third. However if we compare with the pie chart on the right we can see that we also gain a lot; we need only half the time we would need if we did no pruning at all. This shows that pruning decreases the time necessary to complete the full optimization quite drastically. So the extra effort put into pruning is definitely worth it; pruning gives DTSE the ability to get to a better solution with less effort.

2.5 Why automate pruning?

Section 2.4 explained why pruning is essential for the quality and the efficiency of the transformations in DTSE. This however does not yet explain why we want to automate the pruning step of DTSE. It is evident that it is interesting to automate at least part of DTSE because it is a long process to do by hand (*e.g.*, 6.5 months on an MPEG4-encoder [CV00]) and it is used among others on the very competitive market of consumer electronics where

a short time-to-market is an important issue. However this does not necessarily mean that the pruning step needs to be automated too.

There are however several good reasons to automate the pruning step:

- Pruning is a *long and tedious* job. For one thing it is the only step that sees the complete, uncut source with all the dirty details that it has to remove for the rest of DTSE. It is easy to miss something when it is not represented explicitly in the code or when it is surrounded by lots of code that is irrelevant for what you try to find out. Also if the original program needs to be changed because of speed demands (*i.e.*, in iterative design) it needs to be done multiple times which makes it even less interesting to do by hand.
- The step is also error prone and the slightest error in this step means that all of DTSE has possibly to be redone if it is discovered too late since DTSE is a global optimization. A problem is also that transformations on a program in dynamic single assignment form cannot always easily be verified because conversion to dynamic single assignment results in a blow-up of memory use and so the often used verify-by-running may be impossible. The same is true for the dynamic single assignment conversion itself because running the resulting dynamic single assignment program is likely to be infeasible. Verification of transformations without executing the program can assist the designer in these conversions but it will probably never be able to verify the transformations of the pruning step completely but only check certain parts, especially since the pruning step sees the uncut code that is hard to work with.
- Parts of DTSE will be automated, so if we automate the rest and do not automate pruning, its share in the time needed to do the complete DTSE will become larger and so it will become the limiting factor as far as speed is concerned.

Basically, pruning is not exactly interesting to do by hand, so we need to automate it.

```

for (int i = 0; i < 100; i++)
  for (int j = 0; j < 100; j++)
    if (j + i < 105)
      a[j + i][j] = 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] = f(a[i + 99][i]);            // S3

```

Figure 13: Our running example

3 Preliminaries

The transformations presented in this report 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 left hand side of an assignment is to an element of an array.³ The right hand 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.
- The program is in DSA form. We demand that the indexation in the left hand side of an assignment surrounded by n `for`-loops is of the following form:

$$A \cdot \vec{i} + \vec{c}. \tag{1}$$

Here A is a non-singular $n \times n$ matrix, \vec{i} is a vector containing the iterators of the surrounding loops and \vec{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 [Fea91].

This is a well-known set of programs, both in the area of parallelization as well as the area of hardware synthesis [TA93], since these kinds of programs can be modeled and operated on using well-established mathematical methods. The example from Fig. 13 satisfies all of the conditions above.

In this report the programs are represented by a (notationally) simplified version of the geometrical modeling of [CWD⁺98]. We will introduce our notation using the program in Fig. 13. 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

³Note that scalar variables like integers could be considered as arrays of length 1.

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:

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

Now it is possible to refer to each instance of *e.g.*, statement **S1** as $\mathbf{S1}(\vec{i}_1)$ with $\vec{i}_1 \in I_1$, 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 operand mappings by adding an extra index. We will not need this facility in this report.

An important point is that all definition mappings w_s 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

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

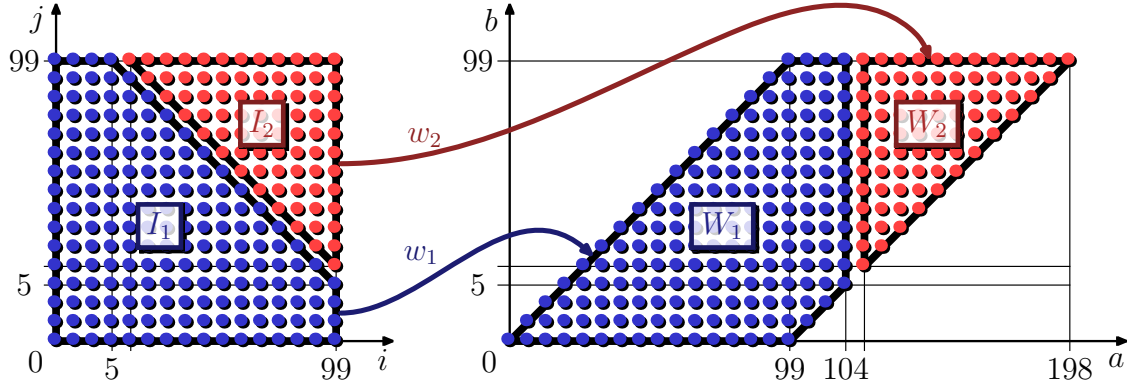


Figure 14: Schematic representation of the iteration domains (*left*) and definition domains (*right*) for S1 and S2

The iteration and definition domains for S1 and S2 are schematically represented in Fig. 14. 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 statement 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 [QRR96]. 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 this report we often leave out the conditions that variables like i and j should be integer, so it is implicitly assumed that they are.

<pre> if (a < b) max = a; else max = b; </pre> <p>(a) DSA despite multiple textual assignments</p>	<pre> for (i = 0; i < 100; i++) { if (i == 50) halfway = ...; ... } </pre> <p>(b) DSA despite the loop</p>
---	---

Figure 15: Dynamic single assignment in case of scalars

4 Dynamic single assignment

4.1 Reprise definition

In this section we will repeat the definition of dynamic single assignment and we will go into more detail about what this means. The definition given in Sec. 2.3 was the following:

Definition A program is in dynamic single assignment form when during the execution of the program only a single assignment happens to each memory element.

The two important elements in this definition are the fact that we look at programs during execution and that we look at each memory element separately. The implications of both will become clearer in the following discussion of different language elements. First we will discuss scalar variables and arrays as the most important data types. Next we will discuss `for`-loops, indexation of arrays and `if`-statement.

Scalars Scalar variables (or simply scalars) are simple data types like integers and characters. Any assignment to a scalar variable will overwrite the complete value stored in that scalar variable and hence if there is more than one write operation to the variable, we no longer have dynamic single assignment. For straight-line code without loops, a sufficient condition for dynamic single assignment is that textually only one assignment is present to each variable. One can see that this is not a necessary condition by looking at the piece of dynamic single assignment code in Fig. 15(a). Since only one of the assignments happens, this code is in dynamic single assignment form, but still textually there are two assignments.

When an assignment to a scalar is in a loop, it is seldom dynamic single assignment since a loop is typically executed more than once and then so is the assignment to the scalar, but this can change because of conditions like in the example in Fig. 15(b). This illustrates the dynamic aspect in our definition of dynamic single assignment.

Arrays Array variables (or simply arrays) are basically collections of elements of the same type that can be indexed using one or more expressions that evaluate to an integer. An array is different from a scalar in the sense that a write operation to an array does not write the whole array but only one element. This means that we can look at an array as a collection of separate memory elements so that a program can still be in dynamic single assignment form even when during execution of the program, there are multiple

<pre> a[0] = 0; for (i = 1; i < N; i++) a[i] = f(a[i - 1]); </pre>	<pre> for (i = 0; i < 10; i++) for (j = 0; j < 10; j++) a[i + j] = f(i, j); </pre>
(a) An example that is DSA	(b) An example that is not DSA

Figure 16: Example using arrays that are DSA or not

```

DO I=1,10,2
C   a loop with iterator I going from 1 to 10 with stride 2
   anything but an assignment to I
END DO

```

Figure 17: Skeleton of a Fortran DO-loop

writes to the same array, as long as they write to another element of that array. An example that illustrates this is the one in Fig. 16(a). Of course it is also possible to have programs that are not in dynamic single assignment form as in Fig. 16(b). This program is not in dynamic single assignment form because there are multiple values for the iterators i and j within their respective bounds so that $a[i + j]$ denotes the same array element. For example when i has as value 0 and j has as value 3, then the same element is written as when i has as value 1 and j has as value 2, namely element $a[3]$. Actually it can be verified that element $a[3]$ is written to 4 times during execution of the program fragment above.

for-loops We first need to define what we mean by **for**-loops since this differs from language to language. Ironically, the kind of **for**-loop we consider is like the DO-loop in Fortran, which is of the form of Fig. 17. In Fortran, the requirement that no assignment happens to the iterator within the loop itself is imposed by the compiler. If we then also suppose that the stride of the loop is constant, the lower and upper limit for the iterator is either constant or a function only of iterators of surrounding loops only. This ensures that the control flow of the program is known at compile time and does not depend on the input for the program. So the ‘behavior’ of the iterators (*i.e.*, which values they subsequently take) is represented explicitly in the program and this is very interesting for analysis and transformations. Now if we look at a C program, what we consider as a **for**-loop is shown in Fig. 18. i is here the iterator, i_{surr} is the set of the iterators of surrounding **for**-loops and s is a constant stride. All other loops in C using the **for**-construct are considered as **while**-loops (and can be written like that) that are typically not analyzable at compile time.

An important remark in the context of **for**-loops is that we do not look at the iterators of **for**-loops as scalar variables that we will include in our transformation to dynamic single assignment form. The reason is that we want to look at iterators as being part of the control flow which can be analyzed at compile time, but we do not try to optimize the control flow but rather change it to optimize the data flow. It is pointless anyway to convert an iterator to dynamic single assignment form since the only viable way

```

for (i = l(isurr); i < u(isurr); i += s) {
    anything but an assignment to i
}

```

Figure 18: Skeleton of what we consider a `for`-loop in C

<pre> for (i = 0; i < 10; i++) { t = f(in[i]); t = g(t,t+1); out[i] = t*(t+1)/2; } </pre>	<pre> for (i = 0; i < 10; i++) { t[i] = f(in[i]); t[i+10] = g(t[i],t[i]+1); out[i] = t[i+10]*(t[i+10]+1)/2; } </pre>
(a) Multiple assignment	(b) Single assignment

Figure 19: Indexation is an important issue in DSA conversion

to do this is expand it to an array that needs to be indexed using another iterator, and this way we never get rid of all iterators. Another way to get rid of iterators is to introduce recursion, but in the context of transformations on C-programs this is not exactly a smart thing to do since recursion is much harder to analyze in general than the simple `for`-loops we have been talking about. Conclusion is that iterators definitely are no scalar variables.

Indexation of arrays Arrays consists of different array elements and code manipulating these arrays need to select different elements of these arrays. For code to be dynamic single assignment it is necessary that at different points in the execution of the program, a different array element is selected to write values to. Thus it is pretty logical that array indexation is an important aspect of the dynamic single assignment concept. Actually the indexation makes the difference between dynamic single assignment and multiple assignment. Figure 19 shows two versions of the same program. Just by adding indexation to variable `t`, and thus making it an array, we transformed the program to dynamic single assignment form.

To be able to analyze the existing indexation, and to find out how to change the indexation, it needs to be in a certain form since we cannot analyze just any indexation expression. We need to know exactly at compile time what the access pattern of the program is. Also we want to use efficient mathematical techniques to analyze the indexation. A class of indexation that suits our requirements is affine indexation, *i.e.*, the indexation needs to be a linear function of the iterators of surrounding `for`-loops plus a constant. Things that are evil are data-dependent expressions or quadratic expressions for which analysis may even be undecidable.

if-statements `if`-statements restrict the values of the iterators for which the statements guarded by the `if`-statement are executed. Thus in a certain sense, these `if`-statements fulfill the same roles as loop bounds. More specifically these conditions, given that they are also affine, are easily incorporated in the mathematical methods used for analyzing

our multimedia applications.

From this discussion it is clear that the programs we consider have array references and `for`-loops as most important constructs. Since all other constructs can be mostly handled before transforming the program to dynamic single assignment form, we will assume in this report that they are not present. A small overview of how this can be done for some constructs is given in Appendix A.

4.2 Static single assignment

Static single assignment is a form of single assignment that is considered as an interesting form of a program to do analyses and transformations on [ASU86, CFR⁺91, Muc97, App98]. For classic compiler optimizations like constant propagation, dead code elimination, . . . this is indeed true if we restrict the transformations to simple data elements and not to arrays which are our main interest. It is however a good starting point to discover the principles of dynamic single assignment and the need for dynamic single assignment. To do this, we will first discuss the idea behind static single assignment. Next we will show an extension of static single assignment towards programs with arrays. Finally we will discuss the shortcomings of static single assignment form and why dynamic single assignment is needed.

4.2.1 Principles

The general idea of static single assignment is to make sure that textually only one assignment to each variable is present in the program. This means that after this transformation it is very easy to determine for a read operation what write operation wrote the value that is being read, namely by simply looking at the name of the variable whose value is read.

Remember however from reaching definitions analysis that multiple definitions can reach a certain statement in a program. This means that when at a certain point we need to use the value of variable, it is not always possible to point out a single write statement and hence a single variable in static single assignment form. The reason is that at some points in a program control flows can join and consequently different definitions of a single variable might join. To capture this, static single assignment introduces ϕ -functions that select one of its arguments depending on where the control flow comes from. A function like this is not effective in the sense that it cannot be implemented as such, but it does not keep us from doing analyses and transformations and it is possible to transform them away afterwards.

An example will make this definition a bit clearer. Let us look at the function in Fig. 20(a). Transforming this function to static single assignment form by adding subscripts to the variable names and adding ϕ -functions where necessary, produces the program in Fig. 20(b). The ϕ -function is introduced because the point just after the `if`-statement is reached by two definitions for `a`. The two definitions of `a` are merged by the ϕ function that selects the value of `a0` if the `if`-statement is not executed and the value of `a1` if it is. In general ϕ -functions are used at each point where control flow joins and where multiple definitions of the same variable reach. This includes loops since the beginning of a loop is reached when entering a loop or when going from a previous iteration to the next one.

<pre>int abs(int a) { if (a < 0) a = -a; return a; }</pre> <p>(a) Maximum calculation</p>	<pre>int abs(int a₀) { int a₁, a₂; if (a₀ < 0) a₁ = -a₀; a₂ = ϕ(a₀, a₁); return a₂; }</pre> <p>(b) SSA version</p>
--	---

Figure 20: An example of scalar static single assignment form

<pre>for (i = 0; i < 10; i++) a[i] = f(i);</pre> <p>(a) An example with an array</p>	<pre>for(i = 0; i < 10; i++) { a₁ = ϕ(a₀, a₃); a₂[i] = f(i); a₃ = ϕ(a₁, a₂); } a₄ = ϕ(a₀, a₃);</pre> <p>(b) The SSA version of the same program</p>
---	--

Figure 21: Static single assignment form for programs with arrays

So the general idea is to rename all definitions by adding a subscript to the name of the defined variable and by inserting ϕ -functions where necessary.

4.2.2 Extension for arrays

Since DTSE targets data-intensive applications, arrays are most important to us. However the simple static single assignment scheme introduced in the previous section does not work for arrays. This is because an array is seldom written in one operation. Rather each element is written separately, so the role of a ϕ -function in case of arrays is no longer to choose between its inputs depending on where the control flow comes from, but to combine multiple arrays into one so that only the most recently assigned values are kept.

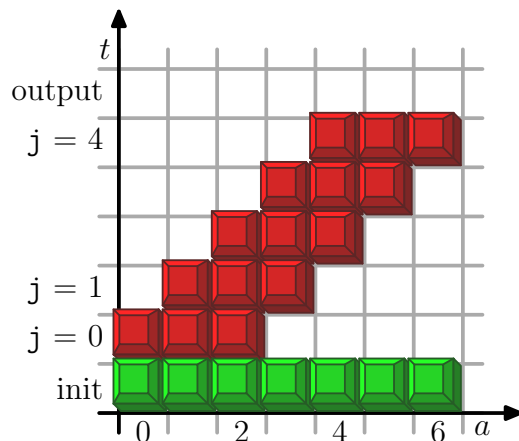
An extension to static single assignment that does this, is explained in [KS98]. There is no need to go into full detail here, we will just demonstrate this using the example in Fig. 21(a). Transforming to static single assignment form gives us the version in 21(b). Here \mathbf{a}_0 refers to array \mathbf{a} as it was before the piece of code we considered here, be it either uninitialized values or previously assigned values; writing \mathbf{a}_0 allows us to make abstraction of that fact. Note that now we have to introduce ϕ -functions after each assignment to an array too and not only when multiple control flow paths join. The reason for this is that we only write a single element of the array and hence all the other elements of the array should still be retained. This is exactly what the ϕ -function does for us: it copies the respective elements from the arrays it has as parameters to the resulting array so that this resulting array contains the last value written to those respective elements. If interpreted this way, a

```

for (i = 0; i < 7; i++)
  c[i] = 0; // S1
for (j = 0; j < 5; j++)
  for (k = 0; k < 3; k++)
    c[j+k] = c[j+k] + a[j] * b[k]; // S2
for (l = 0; l < 7; l++)
  output(c[l]); // S3

```

(a) The C code



(b) Access pattern schematic

Figure 22: Polynomial multiplication example

ϕ -function writes the whole array even if a single element is written in the original program.

These ϕ -functions are not effective either in the sense that a ϕ -function does not know in which array the last element for a certain index is located just by having those arrays as parameter. In [KS98] a way of transforming those ϕ -functions is given, but since it employs run time techniques to do so, we cannot use it because we want to do further compile-time analysis.

4.3 Feautrier’s method

In this section we describe Feautrier’s method of dynamic single assignment conversion as proposed in [Fea91, Fea88a]. This method heavily relies on an exact data flow analysis described in [Fea88b]. Because of this, the method can handle only a limited set of programs. Programs should consist only of nested `for`-loops with bounds that are affine expressions of the surrounding iterators. `if`-statements can be present, but the conditions need to be affine expressions in the surrounding iterators. Further only assignments to scalars and arrays can be present, and the indexation needs to be an affine expression of the surrounding iterators. The right hand side of these assignments can be anything. These are exactly the conditions a program has to satisfy such that it can be modeled in the polyhedral model as presented in Sec. 3. In Sec. 4.3.1 we sketch the basic method and in Sec. 4.3.2 we describe some extensions.

4.3.1 The method itself

We will discuss Feautrier’s method using the simple polynomial multiplication algorithm depicted in Fig. 22(a). This program multiplies two polynomials $a[x]$ and $b[x]$ with $c[x]$ as result. The coefficient of x^k in $a[x]$ is stored in $a[k]$ ⁴, and likewise for the other polynomials.

⁴ $a[x]$ is a polynomial in x , $a[k]$ is the k th element of array a .

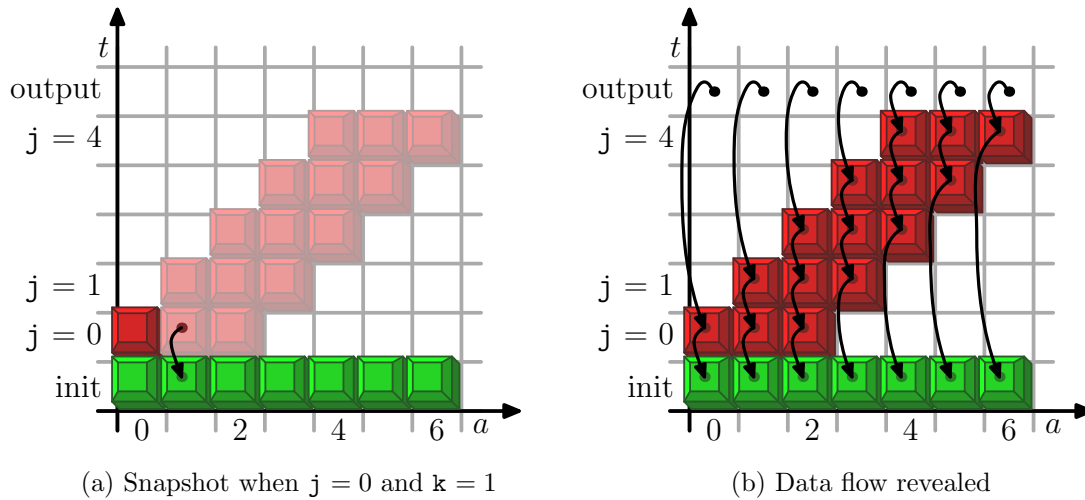


Figure 23: Polynomial multiplication example continued

To ease our understanding of the data flow, the access pattern for array c is schematically presented in Fig. 22(b). There are two statements writing to array c . The blocks at the bottom of the picture belong to the initialization statement, *i.e.*, the first statement in the program. The group of blocks at the top belong to the second statement in the program. The horizontal axis represents the array elements. The vertical axis represents some notion of time. Since the initialization statement accesses different array elements in each iteration, we represent them as if they happen at the same time. This statement also assigns each element of c , hence we have shown a complete row of blocks at the bottom of the picture. The second statement is enclosed within two loops, but for a fixed value of the outer iterator, the inner loop writes to different elements of the array. More precisely the k in $c[j + k]$ makes that the inner loop writes subsequent values, while the j in the indexation makes that the row of elements written in the inner loop is shifted by 1 in each iteration of the outer loop. Again, all iterations of the inner loop can be represented as if happening at the same time. This is not true for the outer loop as there is some overlap between the elements written so each iteration of the outer loop is executed in subsequent time slots, as shown in Fig. 22(b). Note also that the red statement in each iteration reads the same element of c as it writes. Note that every block in Fig. 22(b) actually represents a value, and the column in which it resides indicates the location of the value in the array, while its row position gives some indication as to the (relative) time at which it was written to that array element.

The first step is to determine for each statement that reads from c and for each combination of values for the surrounding `for`-loops which statement wrote the value of the element of c that is read and for which values of the iterators surrounding that statement. Let us first look at statement $S2$ that reads from $c[j + k]$. More specifically we look at the moment when $j = 0$ and $k = 1$. Thus each iteration of statement $S1$ has been executed, as well as one iteration of statement $S2$, namely for $j = 0$ and $k = 0$, which wrote element 0 of c . This situation is shown in Fig. 23(a). The arrow goes from the value that is written by the statement to the value that is read by it. Statement $S2$ reads the same element of array

c as it writes, so it can only read values that are on a line below the value it produces, and not on the same line (or above the line for that matter). Hence the arrow points straight down, and to the block it encounters first because that is the last value written to the array element. So for each value in Fig. 22(b), we can do the same thing to find out which value needs to be read for the calculation of that value. This is shown in Fig. 23(b). The top row represents the last statement that outputs the elements of array c , and since it does not produce values there are no blocks shown. Again each arrow starts at a block and goes down up to the first block it hits. So Fig. 23(b) represents the complete data flow as related to c .

The first step is to capture this data flow in the form of source functions. Let us first analyze statement $S2$. Two cases can be distinguished in Fig. 23(b). Either it is the first time $S2$ writes to a certain element of c , and then we grab back to the value written by statement $S1$. This is the case for the bottom line of the group of values for statement $S1$, *i.e.*, when $j = 0$, and for the rightmost, slanted line of values, *i.e.*, when $k = 2$. Remember that the three values at each row are written, from left to right, for k going from 0 to 2. $S2$ reads element $j + k$, which is written by statement $S1$ for $i = j + k$. We can capture this information in a source function

$$S2(j, k) \rightarrow \text{if } j = 0 \vee k = 2 \text{ then } S1(j + k)$$

For all other iterations of $S2$, the value read is written by $S2$ itself in a previous iteration of the j -loop. More precisely $S2$ for a certain j and k reads the element written by $S2$ for $j' = j - 1$ and $k' = k + 1$. This can be observed in Fig. 23(b) where the arrows all point to the previous row, and *e.g.*, the second element in a row points to the third element in the row below. We capture this extra information by completing the source function above, *i.e.*,

$$S2(j, k) \rightarrow \text{if } j = 0 \vee k = 2 \text{ then } S1(j + k) \text{ else } S2(j - 1, k + 1)$$

Next we analyze $S3$. Looking at Fig. 23(b) again, we can again discriminate two cases. Either we are reading from the top line of values written by $S2$, or the leftmost slanted line of values. The latter case is for when we read one of the first 5 elements of c , *i.e.*, for $1 \leq 5$. We always read the leftmost value of the row the value is on, which is written when $k = 0$ and hence $j = 1$. The former case is when we read one of the last 3 elements of c , *i.e.*, for $1 \geq 5$. Then the value is always written when $j = 4$. A little calculation shows that for a certain $1 \geq 4$ we read the value written by $S2$ for $j = 4$ and $k = 1 - 4$. Note that for $1 = 4$, both cases agree. The source function for $S3$ thus becomes

$$S3(l) \rightarrow \text{if } l \leq 4 \text{ then } S2(l, 0) \text{ else } S2(4, l - 4)$$

Both source functions capture all we need to know about the data flow. After determining the source functions, we are ready to do the actual conversion to dynamic single assignment. First we rename the left hand side of each statement such that each statement writes to a different array. Also we replace the indexation by a canonical indexation, namely each array has a number of dimensions equal to the number of loops surrounding the sole assignment to that array and each dimension is indexed with a different one of the surrounding iterators. This is shown in Fig. 24(a). Note that the reads have not been filled in yet since they remain

<pre> for (i = 0; i < 7; i++) c1[i] = 0; // S1 for (j = 0; j < 5; j++) for (k = 0; k < 3; k++) c2[j][k] = ? +a[j]*b[k]; // S2 for (l = 0; l < 7; l++) output(?); // S3 </pre> <p style="text-align: center;">(a) Changing writes</p>	<pre> for (i = 0; i < 7; i++) c1[i] = 0; // S1 for (j = 0; j < 5; j++) for (k = 0; k < 3; k++) if (j == 0 k == 2) c2[j][k]=c1[j+k]+a[j]*b[k]; else c2[j][k]=c2[j-1][k+1]+a[j]*b[k]; for (l = 0; l < 7; l++) if (l <= 4) output(c2[l,0]); else output(c2[4,l-4]); </pre> <p style="text-align: center;">(b) Adjusting reads</p>
--	--

Figure 24: Converting to dynamic single assignment

to be determined. However this is easily done by just looking at the source functions we have previously determined. If $j = 0$ or $k = 2$ then S2 reads the array element written by S1 for $i = j + k$, *i.e.*, it reads $c1[j + k]$. This is easily found by substituting $j + k$ for i in the left hand side of S1. Likewise when neither $j = 0$ nor $k = 2$, we read the array element written by S2($j - 1, k + 1$), which we find by substituting $j - 1$ and $k + 1$ for j and k in the left hand side of S2, resulting in $c2[j - 1][k + 1]$. We can do the same for statement S3 and the result is shown in Fig. 24(b).

So provided the source functions can be constructed, this method easily does the transformation to dynamic single assignment.

4.3.2 Improvements to the method

In the example in Sec. 4.3.1, there were no parameters present. It is however easy to imagine that the degrees of the polynomials were provided as parameters N and M . Feautrier’s method can still handle a parametrized program like that and then constructs the source functions as functions of those parameters. A parametrized version of Fig. 22(a) is shown in Fig. 25(a) and its dynamic single assignment version in Fig. 25(b).

An extension to Feautrier’s method is described in [Kie00]. While Feautrier’s method is limited to affine expressions, Kienhuis’s method also allows modulo and integer division to appear in the indexation and the loop bounds.

4.3.3 Discussion

Feautrier’s method for dynamic single assignment conversion can perfectly convert programs with only `for`-loops and `if`-statements that can be nested in arbitrary ways. However the bounds and conditions need to be affine expressions of iterators only. An extension is possible

```

for (i = 0; i < N + M + 1; i++)
  c[i] = 0;
for (j = 0; j < N + 1; j++)
  for (k = 0; k < M + 1; k++)
    c[j+k] = c[j+k] + a[j] * b[k];
for (l = 0; l < N + M + 1; l++)
  output(c[l]);

```

(a) A parametrized version of Fig. 22(a)

```

for (i = 0; i < N + M + 1; i++)
  c1[i] = 0; // S1
for (j = 0; j < N + 1; j++)
  for (k = 0; k < M + 1; k++)
    if (j == 0 || k == M)
      c2[j][k]=c1[j+k]+a[j]*b[k];
    else
      c2[j][k]=c2[j-1][k+1]+a[j]*b[k];
for (l = 0; l < N + M + 1; l++)
  if (l <= N)
    output(c2[l,0]);
  else
    output(c2[N,l-N]);

```

(b) Adjusting reads

Figure 25: Parametrized programs

for expressions containing also integer division and modulo, and for expressions containing parameters, but this is too restrictive for practical applications. Indexation can contain references to data, conditions can be data-dependent, `while`-loops can occur, etc. Although extensions exist for analyzing programs containing these constructs, like [CBF97, Mas94], the result of the analysis is no longer exact. Instead of a source function that identifies the producer of a value uniquely, a set of possible producers is returned. However Feautrier’s method requires the producer to be uniquely identified, and there is no obvious way to overcome that problem. Thus we need a different method that does allow extensions to most importantly data dependent indexation and conditions.

A second problem with Feautrier’s method is that it is too slow for real-size applications. Doing an exact array data flow analysis has exponential complexity [Pug91], and although this is a worst-case complexity Feautrier’s method in practice does not do much better than that. So we need a more general and faster method to do dynamic single assignment conversion.

4.4 Extension static single assignment to dynamic single assignment

In this section we present our first alternative method for dynamic single assignment conversion. It does the conversion starting from a program in static single assignment form. Given that the loop bounds are analyzable, this is a simple process. However the dynamic single assignment form produced is not very precise in case of arrays because it approximates the arrays as scalars and hence does not reveal the data flow at the element level. It does however encode information about the loop structure, *e.g.*, by making a difference between the first iteration of a loop where there is no previous iteration and the remaining iterations. Despite

<pre> a = 5; for (i = 0; i < 100; i++) if (i < 50) a = a * 2; else a = a * 3; out(a); </pre>	<pre> a₁ = 5; for (i = 0; i < 100; i++) { a₂ = $\phi(a_1, a_5)$; if (i < 50) a₃ = a₂ * 2; else a₄ = a₂ * 3; a₅ = $\phi(a_3, a_4)$; } a₆ = $\phi(a_1, a_5)$; out(a₆); </pre>
(a) A scalar example...	(b) ... and its SSA form

<pre> a₁ = 5; for (i = 0; i < 100; i++) { a₂[i] = $\phi(a_1, a_5[i - 1])$; if (i < 50) a₃[i] = a₂[i] * 2; else a₄[i] = a₂[i] * 3; a₅[i] = $\phi(a_3[i], a_4[i])$; } a₆ = $\phi(a_1, a_5[99])$; out(a₆); </pre>	<pre> a1 = 5; for (i = 0; i < 100; i++) { a2[i] = (i == 0 ? a1 : a5[i - 1]); if (i < 50) a3[i] = a2[i] * 2; else a4[i] = a2[i] * 3; a5[i] = (i < 50 ? a3[i] : a4[i]); } a6 = a5[99]; out(a6); </pre>
(c) Adding dimensions	(d) Making the ϕ -functions effective

Figure 26: Converting an SSA form to a DSA form

the inaccuracy, this method may prove to be important since advanced copy propagation as described in Sec. 5.2 can extract the data flow from a program in DSA form. This is part of future work and will not be described in this report. We describe the method for the case of scalar variables in Sec. 4.4.1, and for the case of array variables in Sec. 4.4.2.

4.4.1 Scalars

We will illustrate the method on the example of Fig. 26(a). Its SSA form is shown in 26(b). Note that the assignment to a_2 and a_6 both have the same right hand side. This is no coincidence since we duplicated the assignment for clarity. Technically, SSA form would place the assignment to a_2 just before the test $i < 100$, and this is possible in C, but it obfuscates the code a bit.

The assignments within the loop still happen more than once. We can alleviate this by adding dimensions to the a variables as in Fig. 26(c). Adjusting the left hand sides of

assignments is easy: we just add as many extra dimensions as there are surrounding `for`-loops and index them using the iterators of those loops. That way, if one or more of the iterators change value, we will index a different array element. In the example there is just one surrounding loop and thus we add one dimension indexed with `i`. The only tricky part is if the iterators take negative values in which case we can add a constant such that we are sure that the indexation will be non-negative, but this does not occur in our example. If the loop bounds are linear, then we can use integer programming techniques to find the minimum values of the iterators and use those values to determine the required constants.

Adjusting the right hand sides in general is much trickier, but because the variables are scalars and the program is in SSA form, we are in luck. We can discern three different cases. The first case is as variable `a5` in the assignment to `a2`. `a5` is assigned in the previous iteration of the `i`-loop, and thus we add `i - 1` as indexation. The next case is that of `a3` in the assignment to `a5`. The assignment to `a3` happens in the same iteration of the `i`-loop, so we just index with `i`. The third case is that of `a5` in the assignment to `a6`. The assignment happened in the last iteration of the `i`-loop, which is for `i` equal to 99, and so we index with 99. There are no other cases for one loop. If there are multiple loops, that are possibly nested and thus also multiple dimensions, we can consider each dimension and the corresponding indexation separately and cut it down to these three simple cases every time. The only difficulty is in general to find the values of the iterators at the last iteration of a loop, but this can be done using integer programming as before.

The final step is to make the ϕ -functions effective. The ϕ -functions are not effective as they are because all they say is that they select one of its arguments and return it, but they do not say which one of its arguments they return. However, under certain conditions, we can find conditions under which either of the arguments are returned. We will illustrate this in our example above. The ϕ -function in the assignment to `a2` is easiest because it selects between the value of a variable coming from before the loop and the value of a variable coming from the previous iteration of the loop. It should select the variable from before the loop in the first iteration and the other variable otherwise, so the required condition just checks whether the iterator is equal to the lower bound on the iterator. This is shown in 26(d). In the same figure, the ϕ -function in the assignment to `a5` is made effective as well. This ϕ -function selects between values coming from both branches of the `if`-statement, and the condition determining which value to select is of course the same as the one in the `if`-statement. This is shown in 26(d) where the condition `i < 50` is simply duplicated. This can be done with any condition depending only on the iterators, because they do not change value. In case of a condition depending on other variables, those variables may change value such that evaluating the same expression might give a different outcome. Or at least if we did not have a program in DSA form, because if we did, the value of a variable is never overwritten but instead written to a new variable, *i.e.*, we can just copy the condition at all times. The final case remaining is that of the ϕ -function in the assignment to `a6`. This one selects between the value of the variable before the loop (in case it never gets executed because the upper bound is smaller than the lower bound) and between the value coming from the loop. Since the body of the loop is always executed at least once, the ϕ -function always needs to choose `a5` [99]. In general we can put in the condition that the upper bound of the loop should be larger than the lower bound of the loop, *i.e.*, `99 > 0`, which in this case

<pre> for(i = 0; i < 10; i++) { a₁[i] = φ(a₀, a₃[i - 1]); a₂[i][i] = f(i); a₃[i] = φ(a₁[i], a₂[i]); } a₄ = φ(a₀, a₃[9]); </pre> <p style="text-align: center;">(a) SSA form for arrays</p>	<pre> for(i = 0; i < 10; i++) { for (j = 0; j < 10; j++) a₁[i][j] = (i==0 ? a₀[j] : a₃[i-1][j]); a₂[i][i] = f(i); for (j = 0; j < 10; j++) a₃[i][j] = (j!=i ? a₁[i][j] : a₂[i][j]); } for (j = 0; j < 10; j++) a₄[j] = a₃[9][j]; </pre> <p style="text-align: center;">(b) Made the φ-functions effective</p>
---	--

Figure 27: Going from SSA to DSA for arrays

clearly evaluates to true. Note also that in Fig. 26(d) we crystallized the subscripts into the variable names making them effectively different – remember that the indices in the static single assignment form are only annotations.

As a closing remark we add that the resulting program encodes the data flow really well. When looking at it, it is really clear where the values come from, thanks to the selection conditions that are made explicit. Also the occurrence of $i - 1$ shows explicitly that a value is produced in the previous iteration. Thus we are content.

4.4.2 Arrays

The extension towards arrays comes naturally from the extension of static single assignment to arrays. In that case we need to do more work in making the ϕ -functions effective because they look at arrays as scalars by always copying the full array and we need to make this copying explicit. Let us go back to the example from Fig. 21(b) to illustrate this. Applying the technique of adding dimensions from Sec. 4.4.1, we get the program in Fig. 27(a). This step is analogous to the one in Sec. 4.4.1 exactly because we regard the arrays as scalars. The next step is to make the abstraction of the ϕ -function explicit. This consists of two parts: introducing the conditions for selecting the correct arrays, and adding extra loops that copy the complete arrays (without the extra dimensions of course). The result is shown in Fig. 27(b). Only the second ϕ -function needs some explanation. The `for`-loop that replaces it copies all elements from \mathbf{a}_1 except the i th element which is taken from \mathbf{a}_2 instead. Note that when $i = j$, $\mathbf{a}_2[i][j]$ and $\mathbf{a}_2[i][i]$ refer to the same array element, as it should be.

When looking at the program in Fig. 27(b), we find that we do get some information because of the selection conditions that are made explicit, but we get no information about the data flow at the element level. This is a consequence of the fact that we considered the arrays as scalars thus avoiding a possibly complex analysis of the indexation, and so it is obvious that we get no specific information about the element indexed by that indexation. This is surely not the DSA form that we are aiming for. However the methods from Sec. 5.2 allow to remove many of the copy operations we added and filter out the information we want. Our future work will further explore this approach.

```

for (i = 0; i < 10; i++)
  a[i] = f(i);
for (i = 0; i < 10; i++)
  a[i] = g(i);

```

(a) Interstatement: between two statements

```

for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    a[i + j] = f(i, j);

```

(b) Intra-statement: within a single statement

Figure 28: Types of overwriting in a program

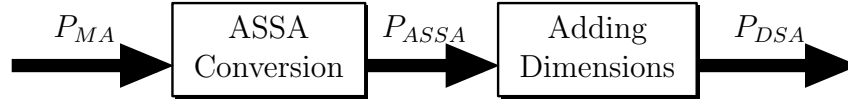


Figure 29: Division in steps of the conversion to dynamic single assignment

4.5 A second method for dynamic single assignment conversion

Recall that the idea of conversion to dynamic single assignment is that once a value is written, it can never be overwritten afterwards. For one thing this means that we need to allocate enough memory to contain all the values produced by the program, or in other words we need to expand the arrays in the program. Another thing is that some values will have to be mapped to another memory location so we need to adjust the indexation of the arrays as well according to this new mapping. These are basically the two problems we need to solve.

If we look at a program in multiple assignment form, we can distinguish two kinds of overwriting: between two statements and within a single statement. We respectively call these interstatement and intra-statement overwriting. Examples of both are given in Fig. 28. In Fig. 28(a) there are two statements that write to the same array \mathbf{a} , so they can possibly overwrite each other. A closer look tells us that they both write to elements 0 through 9 of \mathbf{a} and so the second statement overwrites values written by the first statement and hence we have multiple assignment. If we look at the two statements separately, no overwriting happens. This is called interstatement overwriting.

In Fig. 28(b) however, one can see that there is only a single statement that writes to \mathbf{a} . Since this statement is in a loop, it is executed multiple times for different values of i and j . It is easy to see that this statement overwrites values it has written itself, *e.g.*, $\mathbf{a}[1]$ is written when $i=0$ and $j=1$ as well as when $i=1$ and $j=0$. Note that not all elements of \mathbf{a} are necessarily written more than once, or in general written the same number of times as the other elements, *e.g.*, $\mathbf{a}[0]$ is only written once while $\mathbf{a}[4]$ is written 5 times. This type of overwriting is called intra-statement overwriting. Of course we can also have a combination of interstatement and intra-statement overwriting.

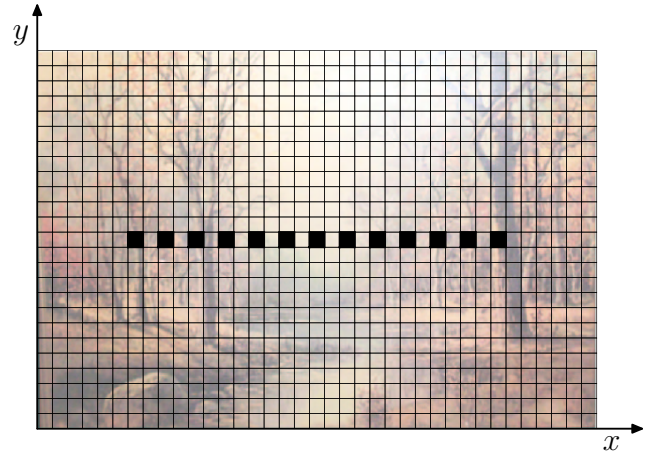
The idea of the proposed conversion to dynamic single assignment form is to solve the two kinds of overwriting in two separate substeps. This is depicted in Fig. 29. The input to the conversion is a program in multiple assignment form: P_{MA} . The first substep of the conversion is called the array static single assignment conversion or ASSA conversion. This conversion is discussed in Sec. 4.5.1. After this conversion there is textually only a single assignment to each array in the program and the resulting program is logically said to be

```

for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    im[y][x] = in(x, y); // S1
for (i = 10; i < 310; i++)
  im[240][2 * i] = 0; // S2
for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    out(x, y, im[y][x]); // S3

```

(a) The image processing C code



(b) Schematic effect of the C code

Figure 30: Drawing a dotted line on an image

in “array static single assignment” form: P_{ASSA} . After this conversion the second substep takes place to get to the full dynamic single assignment form: P_{DSA} . This substep is called adding dimensions and the reason for this, among others, is explained in Sec. 4.5.2.

The resulting program from this conversion will not be the same as the program resulting from Feautrier’s conversion as presented in Sec. 4.3. In Sec. 5 it is shown that this is no problem in the context of DTSE. So the idea is to split up the conversion in two substeps and these substeps is elaborated in the following sections.

4.5.1 Array static single assignment

In this section we describe a way to transform a program to array static single assignment, but without doing a full data flow analysis. A program is in array static single assignment form when there is only one textual assignment to each array in the program, analogously to static single assignment form. Adjusting all write statements to comply is easy, we only need to rename the array in each left hand side to a unique name. The problem is that the read statements need to be adjusted as well, and this can be a complex matter since a read statement could read array elements written by multiple statements. However when there is a single write statement whose values reach the read statement – *i.e.*, it overwrites all values written by other write statements – then the read statement is easily adjusted by just changing the name of the array. This observation is the basis of the method proposed in this section.

We will illustrate the idea with the simple example of Fig. 30(a). This piece of code gets an image from input, draws a dotted line on it, and outputs the result. The effect is shown schematically in Fig. 30(b). Now suppose we would try to determine the source function for S3, as presented in Sec. 4.3.1. This would look as follows

$$\begin{aligned}
\text{S3}(x, y) \rightarrow & \text{if } y = 240 \wedge x \geq 20 \wedge x \leq 618 \wedge x \bmod 2 = 0 \text{ then } \text{S2}(x \div 2) \\
& \text{else } \text{S1}(x, y)
\end{aligned}$$

As one can see, this source function encodes how the resulting image is composed of the original image and the dotted line added to it. In general if more operations happen on the image, like drawing rectangles on it or some more lines, the source function will still describe how it all composes into one image, but obviously gets more and more complex. Unless there happens to be an operation that overwrites the complete picture such that composition is based on the data produced by that operation and anything following it, but not what was overwritten. So what we are going to do is make sure that every statement overwrites what the statement before that wrote.

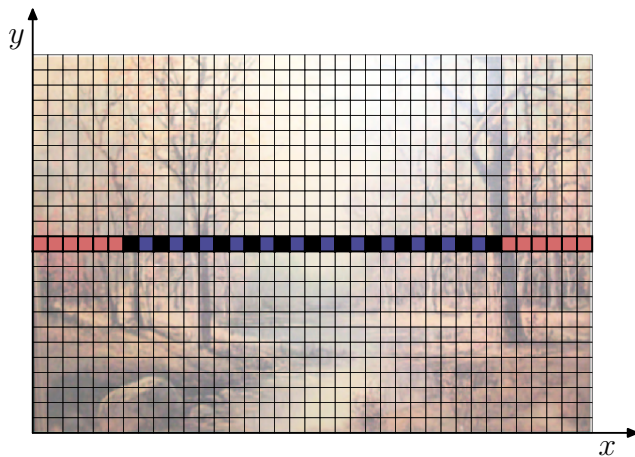
In case of our example we want to expand the statement **S2** that draws the line with extra assignments such that that statement produces the complete image, thus shadowing all of statement **S1**. Since we do not want to change the image that we produce, we let the assignments just assign what was already there, *i.e.*, we just add copy operations. These copy operations do not do anything, but as we will see it will simplify things when they are there. **S2** only writes a dotted line. To make it write the whole image, we need to add three kinds of copy operations, as shown in Fig. 31(a).

- We need to fill the spaces between the dots. So instead of only writing every two elements, we write the dots interspersed with copies. These are the blue pixels in Fig. 31(a).
- We need to add copy operations such that the line extends over the complete width of the image. So we are padding the line on the sides with copies. These are the red pixels in Fig. 31(a).
- We need to add copy operations such that not just the one horizontal line is written, but each horizontal line in the image such that we end up writing the whole image.

The changes to the program required to do these steps subsequently are shown respectively in figures 31(b), 31(c) and 31(d). In Fig. 31(d), the **S2-S4** combination overwrites all values written by **S1**, thus **S3** can only read values produced by the **S2-S4** combination. Note also that by construction, **S2** and **S4** do not overwrite each others values. Also both their left hand sides are equal so we can regard the condition as a selection between the different right hand sides and regard the **S2-S4** combination as one write statement. Once we have the code in Fig. 31(d) it is a trivial step towards array static single assignment form. We rename the left hand sides of **S1** and the **S2-S4** combination such that they are different, and since **S4** only copies elements written by **S1**, and **S3** only reads elements written by **S2-S4**, adjusting the right hand sides is a piece of cake. The result is shown in figure 32. Thus by adding copy operations we are able to substantially simplify the transformation to array static single assignment form.

We can distinguish two kinds of transformations for adding copy operations:

- **changing the indexation/loop structure.** We need to change the indexation when we add copies to fill the gaps between the dots. To do so, we need to make the statement write each element and not just every two elements. To do this, we change the indexation such that all pixels are visited, and use a condition $i \% 2 == 0$ to select between writing the dots and copying the old pixel values. This is shown in Fig. 31(b).



(a) The 3 types of copies to add

```

for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    im[y][x] = in(x, y); // S1
for (i = 20; i < 619; i++)
  if (i % 2 = 0)
    im[240][i] = 0; // S2
  else
    im[240][i] = im[240][i]; // S4
for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    out(x, y, im[y][x]); // S3

```

(b) Adding the gaps between dots

```

for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    im[y][x] = in(x, y); // S1
for (i = 0; i < 640; i++)
  if (i % 2 = 0 && i >= 20
      && i < 619)
    im[240][i] = 0; // S2
  else
    im[240][i] = im[240][i]; // S4
for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    out(x, y, im[y][x]); // S3

```

(c) Extending the line to the picture width

```

for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    im[y][x] = in(x, y); // S1
for (i = 0; i < 640; i++)
  for (j = 0; j < 480; j++)
    if (i % 2 = 0 && i >= 20
        && i < 619 && j == 240)
      im[j][i] = 0; // S2
    else
      im[j][i] = im[j][i]; // S4
for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    out(x, y, im[y][x]); // S3

```

(d) Extending to whole picture

Figure 31: Adding copy operations to make each composition step explicit

When we add copy operations to get up to writing the whole image, we need to change both the loop structure and the indexation. Since we want to write the complete, two-dimensional image instead of just the one-dimensional line, we need two loops instead of just one. Thus we add the j -loop as in Fig. 31(d). Also we need to change the indexation such that we can write each j th line instead of just the 240th.

- **Changing/determining the loop bounds and the selection conditions.** When we extend the line to the whole width of the image, we only need to change the loop bounds, as shown in figure 31(c). Of course we also need to add extra selection conditions that just account for the old loop bounds.

```

for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    im1[y][x] = in(x, y); // S1
for (i = 0; i < 640; i++)
  for (j = 0; j < 480; j++)
    if (i % 2 = 0 && i >= 20
        && i < 619 && j == 240)
      im2[j][i] = 0; // S2
    else
      im2[j][i] = im1[j][i]; // S4
for (x = 0; x < 640; x++)
  for (y = 0; y < 480; y++)
    out(x, y, im2[y][x]); // S3

```

Figure 32: The result of converting to array static single assignment

Also when we add an extra loop, we need to determine the bounds for this new loop. These bounds depend in this case on the height of the image since statement S1 writes the whole height of the image.

The crucial observation is that we can first do all changes to the indexation/loop structure, and then determine the loop bounds and selection conditions by taking the old loop bounds and the part of the array that the previous statement writes into account. The indexation and loop structure determine what could be written, while the loop bounds determine what part of that is actually written and the selection conditions determine what is new and what is copied. In our case we change the indexation and the loop structure to allow writing each pixel of a two-dimensional image instead of just every two pixels on a single line. Then we determine the loop bounds such that only the image is written within its bounds, and determine the selection condition to filter out the dotted line.

In the remainder of this section we will go into more detail about how to do these two steps systematically. We do this for the example in Fig. 33(a) which will show us the different issues in performing this transformation step. This example is based on the example from Fig. 30(a). Figure 33(b) shows a schematic view of the array elements written in the program for $i = 4$. For other values of i , the dotted line would differ in its y coordinate. We model the indexation of both statements as follows:

$$\begin{bmatrix} j \\ k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot (j, k) + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = A_1 \cdot \vec{v}_1 + c_1$$

for $\vec{v}_1 \in \{(j, k) \mid 0 \leq j < 21 \wedge 0 \leq k < 15\}$

$$\begin{bmatrix} 2l + 1 \\ i + 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \cdot (l) + \begin{bmatrix} 1 \\ 3 + i \end{bmatrix} = A_2 \cdot \vec{v}_2 + c_2$$

for $\vec{v}_2 \in \{(l) \mid 1 \leq l < 9\}$

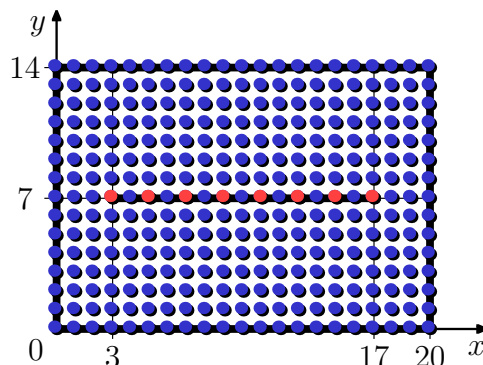
Note that we put the part of the indexation depending on i with the constant part of the indexation. We do this since we want to do the overwriting for each iteration of the i -loop,

```

for (i = 0; i < 9; i++) {
  for (j = 0; j < 21; j++)
    for (k = 0; k < 15; k++)
      a[j][k] = ...; // S1
  for (l = 1; l < 9; l++)
    a[2 * l + 1][i + 3] = ...; // S2
}

```

(a) C program



(b) Schematic view for $i = 4$

Figure 33: A modified version of Fig. 30(a) where the dotted line “moves”

and in each iteration of that loop i is a constant. Since the i -loop is the only loop the two statements have in common, this means that in a given iteration of that loop, S2 overwrites all the array elements that S1 wrote in the same iteration. For i equal to 4, the original situation is shown in Fig. 33(b). The blue dots are written by S1 and the red dots are written by S2. We want to adapt S2 such that the red dots completely overlap the blue ones.

The simplest way to add copy operations to the second statement such that all array elements written by the first statement for a certain value of i are copied to themselves, is as follows:

$$\begin{aligned}
 \begin{bmatrix} 2l + 1 + j' - x' \\ i + 3 + k' - 3x' + y' \end{bmatrix} &= \begin{bmatrix} 2 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -3 & -1 \end{bmatrix} \cdot (l, j', k', x', y') + \begin{bmatrix} 1 \\ 3 + i \end{bmatrix} = A'_2 \cdot \vec{v}'_2 + c'_2 \\
 \text{for } \vec{v}'_2 \in & \{(l, j', k', x', y') \mid 1 \leq l < 9 \wedge j' = k' = x' = y' = 0\} \cup \\
 & \{(l, j', k', x', y') \mid l = 0 \wedge 0 \leq j' < 21 \wedge 0 \leq k' < 15 \wedge x' = 1 \wedge y' = -i\}
 \end{aligned}$$

The iteration domain now consists of two parts. For the first part each of the added iterators is zero, and the old indexation remains, so we still write the new values (the dotted line) to the image. The second part has everything relevant to the old indexation set to zero, and appropriate values for the added iterators. We added j' and k' with the same bounds as the j and k loop of statement S1, and we added x' and y' to compensate for a difference in the constant term between the indexations (counting i again as a constant).

However this is overkill since we made a 5-dimensional loop nest out of a single loop. And we know from our experience with the code from Fig. 32 that a 2-dimensional loop nest ought to be sufficient. Let us look at the new indexation in a different way:

$$\begin{bmatrix} 2l + 1 + j' - x' \\ i + 3 + k' - 3x' + y' \end{bmatrix} = l \begin{bmatrix} 2 \\ 0 \end{bmatrix} + j' \begin{bmatrix} 1 \\ 0 \end{bmatrix} + k' \begin{bmatrix} 0 \\ 1 \end{bmatrix} + x' \begin{bmatrix} -1 \\ -3 \end{bmatrix} + y' \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 + i \end{bmatrix}$$

Incrementing l with 1 changes the indexation by $\begin{bmatrix} 2 \\ 0 \end{bmatrix}$, *i.e.*, we go 2 pixels two the right, and 0 pixels up or down. This is why we get a dotted line in the image. Incrementing j'

by 1 changes the indexation by $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, *i.e.*, we go 1 pixel two the right, and 0 pixels up or down. But instead of incrementing j' by one, we can achieve the same effect by incrementing l by $\frac{1}{2}$. However we want our iterators to be integers (floating point math is not necessarily accurate), so to achieve the possibility to increment by $\frac{1}{2}$, we substitute l by a new iterator $l' = 2 \cdot l$. Then we can get a change in indexation of $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ by incrementing l' by 1, so we no longer need j' to do this. We can still do the change of $\begin{bmatrix} 2 \\ 0 \end{bmatrix}$ required for the original indexation by incrementing l' by 2. The resulting indexation is this:

$$\begin{bmatrix} l' + 1 - x' \\ i + 3 + k' - 3x' + y' \end{bmatrix} = l' \begin{bmatrix} 1 \\ 0 \end{bmatrix} + k' \begin{bmatrix} 0 \\ 1 \end{bmatrix} + x' \begin{bmatrix} -1 \\ -3 \end{bmatrix} + y' \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 + i \end{bmatrix}$$

Of course we still need to determine the bounds for the l' -loop accordingly to take this change into account, but we do that at the end.

Incrementing k' by 1 results in a change in indexation of $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. This cannot be achieved by any change in l' for the simple reason that $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ is not a multiple of $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$. So we have no choice but to keep k' as an iterator.

Next an increase of 1 in x' results in a change of indexation of $\begin{bmatrix} -1 \\ -3 \end{bmatrix}$. This can be achieved by incrementing l' by -1 and incrementing k' by -3. We can find this by writing $\begin{bmatrix} -1 \\ -3 \end{bmatrix}$ as a linear combination of $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, *i.e.*, $\begin{bmatrix} -1 \\ -3 \end{bmatrix} = -1 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} - 3 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, which can be easily determined by a Gaussian elimination [Gar66]. Since these increments are integers, it is possible to do the change in indexation by “incrementing” both l' and k' so we do not need x' for this. Leaving it out gives this:

$$\begin{bmatrix} l' + 1 \\ i + 3 + k' + y' \end{bmatrix} = l' \begin{bmatrix} 1 \\ 0 \end{bmatrix} + k' \begin{bmatrix} 0 \\ 1 \end{bmatrix} + y' \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 + i \end{bmatrix}$$

Finally an increment of 1 in y' changes the indexation by $\begin{bmatrix} 0 \\ -1 \end{bmatrix}$, which is a linear combination of $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, *i.e.*, $\begin{bmatrix} 0 \\ -1 \end{bmatrix} = 0 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} - 1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Thus we do not need y' either to make such a change and leaving it out results in:

$$\begin{bmatrix} l' + 1 \\ i + 3 + k' \end{bmatrix} = l' \begin{bmatrix} 1 \\ 0 \end{bmatrix} + k' \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 + i \end{bmatrix}$$

As we see now, we get exactly two iterators, which agrees with Fig. 32.

The one thing left to do is determine the loop bounds, or equivalently the iteration domain. The first part of the iteration domain which does the assignments that S2 originally

did, is easiest. The original iterator of that loop is still present, be it that it is scaled by a factor 2. We compensate this by taking the old iteration domain $\{(l) \mid 1 \leq l < 9\}$, filling in $l = \frac{l'}{2}$, restricting l' to even values and adding the remaining iterators we found above and setting these to 0. The first part of the iteration domain then becomes:

$$I_2'' = \{(l', k') \mid 2 \leq l' < 18 \wedge l' \bmod 2 = 0 \wedge k' = 0\}$$

The second part of the iteration domain needs to copy all elements written by statement **S1**, for a given value of i . For such a value of i , **S1** is executed for $(j, k) \in \{(j, k) \mid 0 \leq j < 21 \wedge 0 \leq k < 15\}$. Given that the indexation of **S1** is $\begin{bmatrix} j \\ k \end{bmatrix}$, we find the elements written by **S1** as the image of a polyhedron $(j, k) \in \{(j, k) \mid 0 \leq j < 21 \wedge 0 \leq k < 15\}$ through an affine function $(i, j) \mapsto (i, j)$, which in this case is the identity function, so the array elements written by **S1** is

$$W_1 = \{(x, y) \mid 0 \leq x < 21 \wedge 0 \leq y < 15\}$$

However we only want to copy elements not written by **S2**, so we need to subtract

$$W_2 = \{(x, y) \mid 3 \leq x < 19 \wedge x \bmod 2 = 1 \wedge y = i + 3\}$$

giving

$$W_1 \setminus W_2 = \{(x, y) \mid 0 \leq x < 21 \wedge 0 \leq y < 15 \wedge \neg(3 \leq x < 19 \wedge x \bmod 2 = 1 \wedge y = i + 3)\}$$

Given the indexation of **S2'**, $\begin{bmatrix} l' + 1 \\ i + 3 + k' \end{bmatrix}$, we can determine the values for l' and k' – again given a certain value of i – for which elements in $W_1 \setminus W_2$ are written by applying the inverse indexation of **S2'** to $W_1 \setminus W_2$. This is done by filling in $l' + 1$ for x and $i + 3 + k'$ for y in the conditions on x and y in $W_1 \setminus W_2$, resulting in conditions in l' and k' :

$$I_2''' = \{(l', k') \mid 0 \leq l' + 1 < 21 \wedge 0 \leq i + 3 + k' < 15 \wedge \neg(2 \leq l' < 18 \wedge (l' - 1) \bmod 2 = 1 \wedge k' = 0)\}$$

The complete iteration domain of **S2'** is then

$$I_2' = I_2'' \cup I_2''' = \{(l', k') \mid 0 \leq l' + 1 < 21 \wedge 0 \leq i + 3 + k' < 15\}$$

Going back to a C program, gives the program in Fig. 34(a). After bumping the **l** and **k** loop such that their lower bounds are 0, and adding an index to **a**, we get the program in Fig. 34(b). In this version it is clear that we copy the whole image, except for the dotted line.

4.5.2 Adding dimensions

The second substep in the conversion of Fig. 30 is supposed to resolve the intra-statement overwriting. For reasons that will become apparent later on, this step is called adding dimensions. The intention of this step is to get to the full dynamic single assignment form. Since in the previous substep we already solved interstatement overwriting, we only need to

<pre> for (i = 0; i < 8; i++) { for (j = 0; j < 21; j++) for (k = 0; k < 15; k++) a[j][k] = ...; for (l = -1; l < 21-1; l++) for (k=-i-3; k<15-i-3; k++) if (2<=l && l<18 && (l-1)%2=1 && k=0) a[l+1][i+3+k] = ...; else a[l+1][i+3+k] = a[l+1][i+3+k]; } </pre>	<pre> for (i = 0; i < 8; i++) { for (j = 0; j < 21; j++) for (k = 0; k < 15; k++) a1[j][k] = ...; for (l = 0; l < 21; l++) for (k=0; k<15; k++) if (3<=l && l<19 && l%2=1 && k=i+3) a2[l][k] = ...; else a2[l][k] = a1[l][k]; } </pre>
(a) The ASSA version as we find it	(b) A cleaned up version

Figure 34: Completing conversion to array static single assignment

```

for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    a[i * 10 + j] = ...

```

Figure 35: An example that is DSA when loop bounds are taken into account and is not otherwise

solve intra-statement overwriting, *i.e.*, overwriting within the same statement. In this section we first derive a condition such that if that condition is met by the indexation of the left hand side of an assignment, the statement does not overwrite any of its values. Though this condition is not necessary, we still transform the statement to meet this condition because it is a simple condition to check for and to satisfy. After deriving the condition, we show how to make a statement meet that condition.

Initially, we do not take the bounds of the `for`-loops into account. This essentially means that the bounds are infinite so the iterators can take on any value. Or phrased in a more formal way, if the write statement W we consider is surrounded by n `for`-loops, the set of possible (n -dimensional) iteration vectors \vec{i} is \mathbb{Z}^n . Note that if we would take the bounds of the loops into account, that set would be a polytope. Also the condition for having no intra-statement overwriting in the case we consider here is stronger than in the case we do take the loop bounds into account. The example in Fig. 35 shows this. If we keep the loop bounds, it is clear that this piece of code does not do any overwriting: it writes elements 0 through 99 once. However if we throw away the loop bounds, *i.e.*, make them infinite, it is easy to see that *e.g.*, for $i = 0$ and $j = 10$ we write the same element as for $i = 1$ and $j = 0$. This means that by any criterion taking the loop bounds into account, a piece of code may not contain overwriting, while by any criterion that does not, the same piece of code may do overwriting. This is so in general since by making the loop bounds infinite, we still write the elements we wrote to in the original program, but also many more. So the set of writes

```

for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    for (k = 0; k < 10; k++)
      for (l = 0; l < 10; l++)
        a[i + k][i - j + 1][j + k - 2] = ...

```

Figure 36: An ASSA example which we will transform to DSA

in the original program is a subset of the set we consider by making the bounds infinite and hence we will always find at least as much overwriting in the subset we consider here as in the original program. However this is not a problem in our transformation, the only thing that can happen is that we still transform a program that is already in DSA form.

So we look at a single write statement surrounded by n `for`-loops, thus with iteration domain \mathbb{Z}^n , together with a definition mapping describing the indexation. This mapping is a function that maps a vector consisting of the values of the iterators of the surrounding `for`-loops onto a vector containing the values used for indexation the array involved. If the array involved is m -dimensional, this will be a mapping w from \mathbb{Z}^n to \mathbb{Z}^m since we need m indexation expressions. We only consider affine indexation, which means we can describe f as follows:

$$w : \mathbb{Z}^n \rightarrow \mathbb{Z}^m : \vec{i} \mapsto A \cdot \vec{i} + \vec{c}$$

Here \vec{i} is the iteration vector containing the variables representing the iterators, A is an $m \times n$ matrix and c is a vector of length m .

Let us clarify this definition with the example in Fig. 36. There are 4 `for`-loops around the write statement, so n is 4. There are three indexation expressions, so m is 3. The mapping w for the write statement then becomes:

$$f : \mathbb{Z}^4 \rightarrow \mathbb{Z}^3 : \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix} \mapsto A \cdot \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix} + \vec{c} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} i + k \\ i - j \\ j + k \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix}$$

The sufficient condition can be formulated as follows. Suppose statement W does write to the same element for two different iteration vectors \vec{i}_1 and \vec{i}_2 with $\vec{i}_1 \neq \vec{i}_2$. We write:

$$A \cdot \vec{i}_1 + \vec{c} = A \cdot \vec{i}_2 + \vec{c} \Leftrightarrow A \cdot (\vec{i}_1 - \vec{i}_2) = 0 \Leftrightarrow \text{rank}(A) < n$$

As $\vec{i}_1 - \vec{i}_2$ is not 0, $A \cdot (\vec{i}_1 - \vec{i}_2)$ is a linear combination of the n columns of A with non-zero coefficients and hence those n columns are linearly dependent and so the rank of A is smaller than n . So if there is intra-statement overwriting, matrix A used for indexation must have a lower rank than the number of surrounding `for`-loops. Rephrasing gives us the condition we are looking for:

Proposition (Sufficient condition for absence of intra-statement overwriting)

Given an assignment W with \mathbb{Z}^n as iteration domain and definition mapping $f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m : \vec{i} \mapsto A \cdot \vec{i} + c$.

There is no intra-statement overwriting if matrix A used for indexation has a rank equal to the number of surrounding `for`-loops.

It is interesting to note that the same result can be obtained by reasoning about the number of solutions to a system of equations. Consider a certain element of the array the write statement writes to. Let us indicate this element with vector x . We can find the values for the n iterators \vec{i} for which the write statement writes to this element by solving the following system of equations:

$$A \cdot \vec{i} + \vec{c} = \vec{x} \Leftrightarrow A \cdot \vec{i} = \vec{x} - \vec{c}$$

Remember from high-school that the number of solutions to this system can easily be determined as follows (A has dimensions $m \times n$ and $A \mid (\vec{x} - \vec{c})$ represents the augmented matrix):

- A has rank n
 - $A \mid (\vec{x} - \vec{c})$ has rank n : there is exactly 1 solution
 - $A \mid (\vec{x} - \vec{c})$ has rank $n + 1$: there is no solution
- A has rank $d < n$
 - $A \mid (\vec{x} - \vec{c})$ has rank d : there are infinitely many solutions
 - $A \mid (\vec{x} - \vec{c})$ has rank $d + 1$: there is no solution

We can conclude several things from this. First if the rank of $A \mid (\vec{x} - \vec{c})$ is larger than the rank of A (which means $\vec{x} - \vec{c}$ is outside the space spanned by the columns of A), then there is no solution. This means that the element indicated by $\vec{x} - \vec{c}$ is simply never written by the statement. Otherwise if the rank of $A \mid (\vec{x} - \vec{c})$ and A are the same, then there are solutions and hence the array element is written by the write statement. If matrix A has rank n , there is only one solution, so the array element is written only once and there is no intra-statement overwriting for that memory element. If however the rank of A is smaller than n , there are infinitely many solutions, so the array element is written infinitely many times and so for that array element we do have intra-statement overwriting.

Note that the case when there is no solution is not of interest to us since we can always find a $\vec{x} - \vec{c}$ such that the rank of $A \mid (\vec{x} - \vec{c})$ is the same as the rank of A (namely when $\vec{x} - \vec{c}$ is a linear combination of the columns of A). So if the rank of A is n , then either an element of the array is never written, or it is written just once, so in this case there is no intra-statement overwriting. If however the rank of A is smaller than n , there are array elements that are either never written or written an infinite number of times. Since we can always find an element that is actually written, that element must then be written an infinite number of times and hence there is intra-statement overwriting.

The condition we have just derived only holds in the case where all loop bounds are infinite. In case we do take the loop bounds into account, if the rank of A is the number of surrounding `for`-loops then there is no intra-statement overwriting, but when the rank is smaller it could be that there is no intra-statement overwriting. However, for our purposes we will try to meet this condition even if it is not necessary. An assignment like `a[i * 10 + j] = ...` in the example above looks a lot like a linearization of a 2-dimensional array and then we will want to make it 2-dimensional anyway, thus meeting the sufficient condition we derived above. A concrete reason for doing this is that the inverse mapping of $(i, j) \mapsto i \cdot 10 + j$ – taking into account the loop bounds of the example above – is $a \mapsto (\lfloor a/10 \rfloor, a \bmod 10)$ and it contains modulo and floor functions that are much harder to manipulate than simple linear mappings. The inverse mapping is needed in some analyses within DTSE and these are simplified if the mapping is easily invertible, *i.e.*, when matrix A is of rank n . If matrix A is also square, the inverse mapping is not only linear but also simply determined by A^{-1} which is simple to calculate. So if we can make A square and non-singular, this would be the ideal situation.

The next question to answer is how to make the program meet the sufficient condition. If we start with an assignment whose definition mapping has a matrix with rank smaller than n , we observe that if we add rows to matrix A , we can make it of rank n . Suppose A was of rank k , then it is enough to add the correct $n - k$ rows to get a matrix of rank n . The matrix will have $m + n - k$ rows (and still n columns) and since the matrix A represents the indexation the write statement uses, this means we now write to an array of dimension $m + n - k$ instead of dimension m . So this means we have added dimensions to the array, and hence the name of this substep. One thing to note is that each row we add to A has to be linearly independent of the initial rows of A and the other rows we add, adding linearly dependent rows is useless. A second thing to note is that there is an infinite number of sets of rows that we can add to A to make it rank n . Any set of rows that spans the complement of the space spanned by the rows of A (which is also of dimension $m + n - k$) will do, and there are infinitely many of these. Since it is not clear at this point whether one choice is better than another, we will concentrate on how to find solutions rather than which one to choose.

In order to find the rows we need to be added to A (and hence the expressions with which to index the dimensions we add to the array) we propose a method based on Gaussian elimination. One thing we can conclude from the elimination is the rank of A , but it will also allow us to find the solution we are looking for. We show this for an example and then generalize it. Suppose we have the piece of code in Fig. 36 with the matrix A used for indexation repeated here:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

We are now applying Gaussian elimination with pivoting to A . The indexation expressions are indicated with (a) through (c). The pivots and the corresponding iterator and expression indicator are encircled. The selection of the pivots is arbitrary here, but it will determine the solution found. Of course the pivot can't be 0. The pivot is first moved so that we get an upper triangular matrix in the end. Then the row is subtracted from the

rows below to make the entries below the pivot 0. This is the known algorithm for Gaussian elimination.

$$\begin{aligned}
 A \cdot I &= \begin{matrix} (a) \\ (b) \\ (c) \end{matrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix} \rightarrow \begin{matrix} (b) \\ (a) \\ (c) \end{matrix} \begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} j \\ i \\ k \\ l \end{bmatrix} \\
 &\downarrow \\
 &\begin{matrix} (b) \\ (a) \\ (c) \end{matrix} \begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} j \\ i \\ k \\ l \end{bmatrix} \rightarrow \begin{matrix} (b) \\ (c) \\ (a) \end{matrix} \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} j \\ k \\ i \\ l \end{bmatrix} \\
 &\downarrow \\
 &\begin{matrix} (b) \\ (c) \\ (a) \end{matrix} \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} j \\ k \\ i \\ l \end{bmatrix} = A' \cdot I'
 \end{aligned}$$

At this point in the algorithm, we notice that all that is left are rows consisting of only zeros and so we cannot pick another non-zero pivot and hence the algorithm stops. As there are 2 non-zero rows, we know the rank of matrix A is only 2. Since there are 4 surrounding for-loops, and hence A had 4 columns, the rank of A should be made 4 to meet the criterion derived earlier on in this section. We can do this by adding rows to A that are linearly independent of the existing rows in A . This is equivalent to saying that the rows we add should be linearly independent of the rows of A' that results after applying Gaussian elimination [Gar66], and this is easy to do since A' is in upper triangular form. All we need to do is add 2 rows with all zeros except for the 2x2 identity matrix in the rightmost columns of those rows:

$$A'' \cdot I' = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} j \\ k \\ i \\ l \end{bmatrix}$$

Suppose that we would always choose to add these simple rows. Now remember that the number of columns of A is n and the rank of A is k , so we need to add $n - k$ rows. After applying Gaussian elimination, A' looks like this:

$$A' = \begin{bmatrix} U_{k \times k} & B_{k \times (n-k)} \\ 0_{(m-k) \times k} & 0_{(m-k) \times (n-k)} \end{bmatrix}$$

Here $U_{k \times k}$ is an upper triangular matrix with k rows and columns. $B_{k \times (n-k)}$ is any matrix with k rows and $n - k$ columns. $0_{m \times n}$ refers to a matrix with m rows and n columns

```

for (i = 0; i < 10; i++)
  for (j = 0; j < 10; j++)
    for (k = 0; k < 10; k++)
      for (l = 0; k < 10; l++)
        a[i + k][i - j + 1][j + k - 2][i][l] = ...

```

Figure 37: Program of Fig. 36 after adding dimensions

containing only zeros. Note that we abuse the notation a bit by giving every matrix its dimensions as subscripts. Adding rows like we did above results in:

$$A' = \begin{bmatrix} U_{k \times k} & B_{k \times (n-k)} \\ 0_{(m-k) \times k} & 0_{(m-k) \times (n-k)} \\ 0_{(n-k) \times k} & I_{(n-k) \times (n-k)} \end{bmatrix}$$

This matrix has $n + m - k$ rows as said earlier on in this section. In our example the matrix now has $4 + 3 - 2 = 5$ rows. Again the rows we added to A' can also be added to A to make A of rank n . Keeping in mind that the Gaussian elimination swapped the columns (so we use the original I instead of the I' after the elimination), we get A''' to be the new indexation:

$$A''' \cdot I = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix}$$

Matrix A''' is of rank 4 too, so if we use this matrix for indexation, there is no intra-statement overwriting. The according program now looks like Fig. 37. As one can see the new index expressions are simply iterators. This is of course a consequence of our choice to add rows to A with all zeros except a single 1. There are lots of other choices however. For one thing instead of using $I_{(n-k) \times (n-k)}$ in constructing A'' , we could use any non-singular $(n-k) \times (n-k)$ matrix. Another thing is that we can add any linear combination of the original rows in A to the newly added rows (this is the inverse of Gaussian elimination actually, so applying the elimination again will subtract the linear combinations we added).

Besides that however there is another freedom we have in choosing the final indexation. Since the initial indexation possibly contains expressions that are linearly dependent on the remaining ones, we can actually leave those indexation expressions out as well as the corresponding dimension of the array. This is so because when we choose the value of the linearly independent indexation expressions, then the value of the linearly dependent ones are known and fixed and hence only a single element in the dimensions of the linearly dependent indexation expressions is used. However, suppose that there are k independent rows among the m rows of A , then we cannot just leave out any $m - k$ rows because we have to make sure the remaining ones are linearly independent. The way to ensure this is to leave the expressions out that the Gaussian elimination reduced to rows of only zero and then we know the remaining ones are linearly independent. Although not all choices are always valid, there are several choices, *e.g.*, in the example above any of three original index expressions

is acceptable to leave out. The thing to remember is that there is lots of freedom in choosing the expressions to add and maybe it is possible to use this to our advantage.

In our example above we notice that the first indexation expression $i + k$ is linearly dependent on the second and the third – respectively being $i - j + 1$ and $j + k - 2$ – and the constant 1. So the first expression can be written as a linear combination of the other two and 1, namely:

$$i + k = 1 \cdot (i - j + 1) + 1 \cdot (j + k - 2) + 1 \cdot 1$$

The constant 1 is included in the linear combination to catch a difference in constant term which does not make a difference in the rest of the story. Suppose we would choose the values of i , j , k and l to be so that $i - j$ equals a certain x and $j + k$ equals a certain y . There are several values of i , j , k and l that produce the same x and y , *e.g.*, increasing i and j by 1 and decreasing k by one will not change the values of x and y since the increases and decreases will cancel each other out. Also changing l has no effect. However since $i - j$ is linearly dependent of the other indexation expressions, it can be written in terms of those indexation expressions only, or – since x and y represent those indexation expressions – in terms of x and y only, namely:

$$i + k = x + y + 1$$

The point is that if we know the value of the expressions used for indexation of the second and third dimension, no matter what the values of i , j , k and l are, we also know the value of the expression used for indexation of the first dimension. So we could say that in the first dimension only one element is indexed since we need to change the indexation of the other dimensions to change the indexation of the first.

For our example this means we can leave out the first indexation expression giving a new indexation represented by A'''' :

$$A'''' \cdot I = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix}$$

So in general if we leave out the $m - k$ expressions that are converted to zero rows by the Gaussian elimination, then the result is a matrix A'''' with dimensions $((n + m - k) - (m - k)) \times n = n \times n$ which is square. Remember that this is ideal since now the inverse mapping of the indexation is simply given by the inverse of A'''' . The inverse mapping is important because it tells us, given a certain memory element, for which value of the iterators the element in question is written.

An issue that was completely left out up to now is what to do about the read operations. If we add and remove dimensions from arrays and change the write operations accordingly, the read operations should be adjusted accordingly too. For this we use data flow analysis, *e.g.*, Feautrier's [Fea91]. However now we have a simplified case because the program is in array static single assignment form such that the analysis only needs to consider one write statement. When that is the case, the largest portion of the work can be skipped because we do not have to combine the source functions for the different assignments [Kie02].

<pre> for (i = 0; i < 9; i++) if (cond(c[i])) a[c[i]] = i * i; </pre> <p>(a) Data-dependent code</p>	<pre> for (i = 0; i < 9; i++) for (j = 0; j < 9; j++) if (j == c[i] && cond(c[i])) a[j] = i * i; else a[j] = a[j]; </pre> <p>(b) Made manifest by adding copies</p>
---	---

Figure 38: Handling data-dependent conditions and indexation by adding copy operations

4.5.3 Discussion

We presented a two-step method to do dynamic single assignment conversion. The first step removes interstatement overwriting, *i.e.*, overwriting between two different statements. The resulting program is said to be in array static single assignment form. The second step transforms this program to full dynamic single assignment form by removing intra-statement overwriting. The novel part lies in the first step where we add copy operations to the program to simplify the data flow and thus to simplify the transformation. By adding copy operations to a statement to “overwrite” what a previous statement wrote, we eliminate the possibly complex interactions as far as data flow is concerned between the statements. So we can do a more efficient transformation at the cost of extra copy operations. This is not much of a problem because we can afterwards remove those copy operations by doing advanced copy propagation which is described in Sec. 5.2.

A second advantage of our approach of adding copy operations is that we can extend this technique to handling data-dependent conditions or indexation. This is illustrated in Fig. 38. The data-dependent condition as well as the data-dependent condition are both transformed in a condition that decides what to assign to `a[j]` which is indexed with an affine expression of iterators only. These can possibly be removed by an extension of the technique discussed in Sec. 5.2.

```

int c[19];
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        c[i + j] = c[i + j] + a[i] * b[j];

```

Figure 39: When transformed to DSA, C will be about 5 times as large

5 To DSA or not to DSA?

In the previous sections we have been discussing how the transformation to dynamic single assignment can be automated in the context of DTSE. Of course we do this because we hope that the transformation will have a positive effect on the result of DTSE. The first effect is already discussed in Sec. 2.4. Putting a program in dynamic single assignment form increases the explicit freedom for optimizations on the program and speeds up DTSE. These two parts are essential for DTSE to work.

Another effect however is that memory usage of a program in dynamic single assignment form is usually higher than the original program in multiple assignment form. This is because in general a variable or an element of an array contains more than one value during execution of the program, or in other words there is often more than one assignment to each array element and since we cannot allow this in dynamic single assignment form, we have to allocate an array element for every dynamically executed write statement. Let us look at the example in Fig. 39 to make this clear: This program represents the multiplication of two polynomials. The array c has 19 elements. However the assignment to c is executed 100 times, or in other words there are 100 dynamic instances of the assignment to c . Since each instance of the write statement should write to a different element, this means that c should contain at least 100 elements which it does not. The solution to this is to increase the size of the array. This is what is done in Sec. 4.5.2 by adding dimensions to the array. Note that in theory 100 elements should suffice in our example, in practice it will often be more, *e.g.*, because arrays need to be rectangular and the needed space is triangular, so we need to allocate a bounding box. For the example above the new memory usage is well within acceptable bounds, but for real applications, especially multimedia applications where huge amounts of data are processed using complex algorithms, this can become a problem because memory usage almost literally blows up. This would mean that although we can do many optimizations, the eventual optimized program would be impossible to execute because of memory limitations of computers. This is however not true since DTSE contains an optimization that takes care of this excessive memory usage. In a sense it is the inverse of the dynamic single assignment transformation. How this is done will be discussed in Sec. 5.1. For now it is enough to remember that a step is provided in DTSE to solve the problem of increased memory requirements due to dynamic single assignment. This step is necessary independently of the way we transform to dynamic single assignment because highly increased memory use is an inherent property of dynamic single assignment form.

The problem of high memory requirements can be much more than the theoretical minimum because the method we described in Sec. 4.5 adds copy operations to the program and since each copy operation is a combination of a read and write operation, the number of write operations will increase because of this. This means that we need to allocate even

more memory to accommodate this. However increased memory requirements are not the only problem caused by those extra copy operations. For one thing there are more memory accesses so both program execution and power consumption will increase. Also the extra copy operations may hamper the optimizations DTSE tries to do, unless we would adapt it to take the copy operations in account. For all these reasons the extra copy operations should be removed as much as possible.

A way to remove copy operations is to use data flow transformations. In essence, any transformation that removes the copy operations is a data flow transformation since the data flow is changed (for one thing the number of operations is changed, so it cannot be that all operations are still there and the same). We will discuss data flow transformations in Sec. 5.2. Note that data flow transformations are not limited to removing copy operations, as will be clear from the discussion in Sec. 5.2. Though this is present in the DTSE script for other reasons, it is essential if we use our new technique to do the dynamic single assignment conversion by adding copy operations. In a sense we use these data flow transformations to transform the dynamic single assignment form that is easier to construct to a dynamic single assignment form that is preferable.

5.1 Inplace mapping

The DSA form of a program accommodates a memory element for each value produced in the execution of that program. However not all of these values are live at the same time, and in principle we need to allocate at each time only the number of values that are live. Or, since values are stored in array elements, we only need to allocate as much memory as the number of array elements that are live simultaneously. This can be accomplished by storing values whose liveness does not overlap in the same memory location. So we want to find a function $f(v)$ that maps a value v to a memory location such that $f(v_1) \neq f(v_2)$ whenever the liveness of v_1 and v_2 overlaps.

Theoretically, the minimum overall storage size is achieved by allocating new values to memory elements containing dead values whenever possible. Memory size would then only be increased if we need to allocate another value and no memory element is free to store it in. However this can lead to a very complex function f because the liveness of the values can be quite complex. That is why we prefer a suboptimal solution, but one with a very simple function f . Often this function is constrained to only contain modulo functions, *e.g.*, $f(v_i) = i \bmod n$ as in [DGCDM96]. The number n then indicates the memory use. Note that we also assume we can order the values. In [DGCDM96] this is taken to be the order of the array the values are stored in for some linearization of the array.

We illustrate this on the simple example in Fig. 40(a). This program calculates the value of an element of array **a** using the two previous array elements. After that, one of the two previous array elements is no longer needed, namely **a**[**i**-2] and so its place can be taken by the new value, *i.e.*, **a**[**i**]. This can be achieved by applying a modulo function to the indexation of array **a** (so the indexation is taken as an ordering for the values), as in Fig. 40(b). The needed array size is then only 2, which is the minimum achievable. For further details we refer to [DGCDM96], or for a slightly different approach to [TBJC02].

<pre> a[0] = 0; a[1] = 1; for (i = 2; i < 100; i++) a[i] = a[i-1] + a[i-2]; </pre>	<pre> a[0] = 0; a[1] = 1; for (i = 2; i < 100; i++) a[i%2] = a[(i-1)%2] + a[(i-2)%2]; </pre>
(a) Before array compaction (DSA program)	(b) After array compaction (multiple assignment program)

Figure 40: Example program

5.2 Data flow transformations

The data flow transformations that we discuss here are the ones related to the removal of copy operations from a C program. There are two possible reasons why a copy operation is not necessary and can be removed. The first reason is that when we read the copied data, we could as well read the original data and save us the copies. The second reason is because the copied data is never read so we can save us those copies too. As we will see, we can sometimes solve both in one shot.

The general idea of removing copies is to read the original data rather than the copy. If we replace every read operation that reads the copy with a read operation that reads the original, then the copy is no longer needed and we can remove it. The fact that we remove the copy operations immediately takes care of the copies that were never used because all copy operations are removed in the end. However for this we need to be able to remove every read operation that reads from the copy, and this is not always possible.

In the remainder of this section we explain the method of advanced copy propagation which removes the unnecessary copy operations. We will do this by example. For a more detailed exposition of the matter, we refer to [VJB⁺02, VJB⁺03].

The example we look at is shown in Fig. 41(a). This program runs over the array swapping two subsequent elements, using a variable `tmp` to do the swap. The dynamic single assignment version of the program is shown in Fig. 41(b). We will work on that version because it simplifies the transformation a great deal, as we will see as we go along.

The program contains a number of copy operations. Let us pick the S1-S2 combination first. Both statements can be regarded as a single assignment to `tmp[i]` since the condition that selects between either statement really just selects what is assigned, not what is assigned to. Though it is perfectly valid to regard them as separate, it saves us some work if we do not because then we can propagate both in one breath, so let us do so. The only statement reading from array `tmp` is S4. Thus we can replace its reference `tmp[i]` by the right hand side of the assignments to `tmp[i]`, taking the condition of the `if`-statement into account. The result is shown in Fig. 42(a) where statement S4 was split up in S7 and S8 to account for the condition. In a sense, we can say that the condition moved from the write to to read. Now every reference to `tmp` has disappeared and thus it can be removed as it has just become dead code. This is shown in Fig. 42(b).

That was pretty easy. So let us pick another copy statement, say S3. S3 writes to `a1` which is read by S5 only. But this time S3 writes `a1[i]` and S5 reads `a1[k]`, so we need

<pre> for (i = 0; i < 99; i++) { tmp = a[i]; a[i] = a[i + 1]; a[i + 1] = tmp; } for (k = 0; k < 100; k++) output(a[k]); </pre> <p>(a) Example program</p>	<pre> for (i = 0; i < 99; i++) { if (i == 0) tmp[i] = a[i]; // S1 else tmp[i] = a2[i]; // S2 a1[i] = a[i + 1]; // S3 a2[i + 1] = tmp[i]; // S4 } for (k = 0; k < 100; k++) if (k < 99) output(a1[k]); // S5 else output(a2[k]); // S6 </pre> <p>(b) DSA version</p>
---	--

Figure 41: A program with copy operations

<pre> for (i = 0; i < 99; i++) { if (i == 0) tmp[i] = a[i]; // S1 else tmp[i] = a2[i]; // S2 a1[i] = a[i + 1]; // S3 if (i == 0) a2[i + 1] = a[i]; // S7 else a2[i + 1] = a2[i]; // S8 } for (k = 0; k < 100; k++) if (k < 99) output(a1[k]); // S5 else output(a2[k]); // S6 </pre> <p>(a) Propagated S1 and S2 to S4</p>	<pre> for (i = 0; i < 99; i++) { a1[i] = a[i + 1]; // S3 if (i == 0) a2[i + 1] = a[i]; // S7 else a2[i + 1] = a2[i]; // S8 } for (k = 0; k < 100; k++) if (k < 99) output(a1[k]); // S5 else output(a2[k]); // S6 </pre> <p>(b) Removed S1 and S2</p>
--	---

Figure 42: Propagation of S1 and S2

to match the indexation. Clearly, they refer to the same array element when $i = k$. So we substitute this for i in the right hand side of S3 to get $a[k + 1]$ which is what would be assigned to $a1[k]$ and thus we replace the reference to $a1[k]$ by $a[k + 1]$. This is shown in figure 43(a). As such all references to $a1$ have died and we can remove the useless assignment S3 to $a1$. The result of this is depicted in Fig. 43(b).

We are doing well. The next copy statement on the to-do list is the S7-S8 combination. These write to $a2$, which is referenced by both S8 and S6. Of these two, S8 looks a bit threatening since it both reads and writes to the same array, so we first propagate to S6. S6 reads $a2[k]$, while S8 writes $a2[i + 1]$, so we need to match up the indexation again. It is easy to check that $a2[k]$ and $a2[i + 1]$ reference the same array element when $i = k - 1$.

<pre> for (i = 0; i < 99; i++) { a1[i] = a[i + 1]; // S3 if (i == 0) a2[i + 1] = a[i]; // S7 else a2[i + 1] = a2[i]; // S8 } for (k = 0; k < 100; k++) if (k < 99) output(a[k + 1]); // S5 else output(a2[k]); // S6 </pre> <p>(a) Propagated S3 to S5</p>	<pre> for (i = 0; i < 99; i++) { if (i == 0) a2[i + 1] = a[i]; // S7 else a2[i + 1] = a2[i]; // S8 } for (k = 0; k < 100; k++) if (k < 99) output(a[k + 1]); // S5 else output(a2[k]); // S6 </pre> <p>(b) Removed S3</p>
---	---

Figure 43: Propagation of S3

<pre> for (i = 0; i < 99; i++) { if (i == 0) a2[i + 1] = a[i]; // S7 else a2[i + 1] = a2[i]; // S8 } for (k = 0; k < 100; k++) if (k < 99) output(a[k + 1]); // S5 else output(a2[k - 1]); // S6 </pre> <p>(a) Propagated S8 to S6</p>	<pre> for (i = 0; i < 99; i++) { if (i == 0) a2[i + 1] = a[i]; // S7 else a2[i + 1] = a2[i]; // S8 } for (k = 0; k < 100; k++) if (k < 99) output(a[k + 1]); // S5 else output(a2[k - 2]); // S6 </pre> <p>(b) Propagated S8 to S6 again</p>
---	---

Figure 44: And trouble strikes...

Note that S6 is only executed when k is 99⁵, and so we read a value written by S8 and not S7. Filling $i = k - 1$ in in the right hand side of S8 and substituting the result for $a2[k]$ in S6 give figure 44(a).

So what did we achieve by this? S6 still reads an element of $a2$, and this element is still written by S8. The difference is that S8 now no longer needs to write $a[99]$, so it is dead code for one iteration of the i -loop. One down, 98 to go. Just for fun, let us propagate S8 to S6 again and see what happens. Turns out what happens is that $a2[k - 1]$ is replaced with $a2[k - 2]$, as in figure 44(b). It is not difficult to see where this is going; the indexation will be decremented by 1 until it becomes 1 in which case it is S7 that writes $a2[1]$ such that we can replace the read from $a2$ with a read from $a[0]$. This is shown in Fig. 45(a). At this point all of statements S7 and S8 have become dead code and hence we surgically remove it resulting in Fig. 45(b).

⁵So we can replace k by 99, but that destroys the obvious correspondence between the left hand sides of S5 and S6

```

for (i = 0; i < 99; i++) {
  if (i == 98)
    a2[i + 1] = a[0]; // S7
  else a2[i + 1] = a2[i]; // S8
}
for (k = 0; k < 100; k++)
  if (k < 99)
    output(a[k + 1]); // S5
  else output(a[0]); // S6

```

(b) Removed S8

(a) Recursively propagated S8 to S5

Figure 45: And now we can wrap it up...

All that is left now are the non-copy statements. This program still outputs the same values, but saves many a memory access. Also the data flow has become clear. The final program shows that the effect of the program is to shift the array element down a notch and put the first element in the last place, and output the elements in this new order. This may not have been apparent from the original program.

More details about the general, automated method to do copy propagation can be found in [VJB⁺02, VJB⁺03].

5.3 Discussion

This section discussed two important issues concerning dynamic single assignment conversion. The first is intrinsic to dynamic single assignment, namely the fact that array sizes can grow considerably. The second is a consequence of our choice of conversion to dynamic single assignment by adding extra copy operations. We discussed that this is not a problem because the DTSE script contains two steps that tackle these issues. Firstly array compaction succeeds in compressing array to a smaller size than the original arrays even when the initial conversion to dynamic single assignment form creates very large arrays. Secondly advanced copy propagation removes extraneous copy operations, but can also remove the copy operations we added in our dynamic single assignment conversion, additionally allowing us to keep some copy operations in favor of program simplicity. Thus we can turn these apparent disadvantages in our advantage.

6 Conclusion

This report presents our work on conversion of a program to dynamic single assignment form. We argued that this conversion is an essential preparatory step for a methodology that globally optimizes memory use because it makes transformations both simpler and more powerful though remaining simple.

We described the existing work on dynamic single assignment conversion and how it is not satisfactory for our purposes since they are not generally applicable and not very scalable. The major contribution of our work is the development of a method that overcomes these two shortcomings. This is achieved by adding copy operations to the program which simplifies the data flow considerably if done well. This allows us to do a simpler dynamic single assignment conversion that has the potential of being faster, being more general and allowing a trade-off of complexity of transformation and complexity of the resulting program against the overhead of addition of some copy operations. We have described two such methods in this document, where the one starting from static single assignment is expected to be more general but it is more brute-force in its addition of copy operations. In our future work we will implement the brute-force approach and we will investigate how we can integrate the other method in it to prevent us from adding easily avoidable copy operations.

The methods in this report still focus on the type of programs that existing dynamic single assignment conversion methods can handle already. Future work will also investigate deeper how we can adapt our method to a more general type of programs.

```

struct vector10
    int length;
    int elements[10];
myVector;

myVector.length = 5;
for (i = 0; i < 5; i++)
    myVector.elements[i] =
0;

```

Figure 46: A `struct` representing a vector of length 10

A Overview of unhandled constructs

In Sec. 4 we discussed scalars, arrays, `for`-loops, affine indexation and affine condition `n` in the context of dynamic single assignment. Here we describe some other programming constructs and how they can be handled in a dynamic single assignment conversion.

structs Structured types in C exist of multiple variables of possible different types grouped together. Since in the absence of pointers, any program using `structs` can be rewritten to a program without `structs` by splitting the `struct` into its members and since mostly the variables in the `structs` are assigned separately, we consider each variable in the `struct` as a separate memory element. Of course if the `struct`'s members are composed data structures like arrays or `structs`, then we also have to recursively consider the different parts of those as separate memory elements. This consideration is what determines the piece of code in Fig. 46 to be in dynamic single assignment form: However since it is possible to rewrite the code without `structs`, and because this possibly even allows better optimization because the grouping of the data is no longer required, we can assume that the code we look at for conversion to dynamic single assignment form does not contain any `structs`.

unions `unions` are very much like `structs`, except that the members of a `union` share the same memory space. This means that we can no longer split the `union` up in different memory elements. However mostly we can replace the `union` with a `struct` so that the functionality of a program does not change although memory use will increase. Once we have only `structs`, we can split them up as before and then we can rewrite the program without `structs` or `unions`. This means that we can again assume that in the input program to our conversion to dynamic single assignment form, no `unions` are present.

There are however cases where we cannot simply replace a `union` with a `struct`, but these depend on how the compiler arranges the fields within memory and on platform-dependent representations like little-endian and big-endian byte ordering and hence we assume the programmer did not use this kind of code or at least rewrote it in another way.

<pre>int b[10]; for (i=0; i<10; i++) { int a = f(i); b[i] = g(a); }</pre>	<pre>int a[10], b[10]; for (i=0; i<10; i++) { a[i] = f(i); b[i] = g(a[i]); }</pre>	<pre>int a[10], b[10]; for (i=0; i<10; i++) { a[i] = f(i); for (i=0; i<10; i++) b[i] = g(a[i]); }</pre>
(a) With a loop-local variable	(b) Changing it to an array	(c) Now we can split the loop

Figure 47: Loop-local variables prevent loop splitting

while-loops `while` loops are very different from `for`-loops because they often do not have an (explicit) iterator, and the condition is often data-dependent such that the number of iterations is not known in advance. For the exact analyses DTSE wants to do, this is a full-size problem. However often it is possible to introduce an explicit iterator, and find some upper bound on the number of iterations. This upper bound usually depends on the structure of the data that is being processed, *e.g.*, the length of the array being processed. In any case we are often forced to approximate in some way when it comes to `while`-loops.

Other approaches towards `while`-loops add a dynamic mechanism to the program to account for the difference of the actual number of iterations of the `while`-loops, *e.g.*, see [LG95, GC95, RP95]. However these dynamic mechanism do not lend themselves to static analysis, and are hence not very useful in the context of DTSE. So handling `while`-loops remains a sizeable problem in the context of DTSE.

Local variables Variables can be local to a function or in general to any block of code. Every time the function or the block of code is executed, a new instance of the variable is made. Is the code in dynamic single assignment form when within such a function or block, only a single assignment happens to the local variable? The answer to this question is not as simple as yes or no. That is why we will rephrase the question: are such local variables wanted in the pruned program or not. The answer to this is that they are not. Take for example the code in Fig. 47(a). Although this form could be regarded as dynamic single assignment, it does not give us the freedom for transformations we might want. It is for example not possible to split the loop above in two loops each containing one statement from the previous program. Suppose however that we transform the variable `a` to a global variable and then expand `a` to an array so that we get a dynamic single assignment form. The result is shown in Fig. 47(b). Now we can split the loop as in Fig. 47(c). Note that we introduce an array this way that cannot be removed unless the loops are merged again. However if the splitting allows us to collapse other, larger arrays into a single scalar variable (*e.g.*, by the inverse of the program above) then this may be well worth the extra array.

So local variables should be transformed to global variables to give full transformation freedom. This is true for both local variables in blocks and in functions. However when

only freedom is desired for interprocedural code transformations, it is sufficient that all variables local to a block are at least moved to the scope of the function it is in.

Global variables Global variables, as discussed in the previous point, are exactly what we want, though sometimes the complexity of putting global variables in dynamic single assignment form can be tricky since method calls tend to be more chaotic than the body of those methods themselves. *E.g.*, recursive functions are much harder to analyze than *e.g.*, `for`-loops. If we only need dynamic single assignment within a function, it's sufficient to copy the state of the global variable to a local variable in the function, and back in then end, and make the local variable dynamic single assignment.

Library functions For library functions, the source code is often not available. However since we do transformations at the source level, this poses a problem. That is why often library functions are not considered, which means in our case they belong in layer 3. So it does not matter whether the library function internally is in dynamic single assignment form, but we do need to know what is read and what is written by the function as a whole, or at least a safe approximation of that. In the worst case we must assume everything might be written and read, or might be not.

Pointers Pointers are used to access memory locations in different ways than simply by the names of variables. This on one hand makes it a powerful programming construct but it is also difficult to analyze in an exact way since a pointer can point to anything. It is possible however through transformation to remove the largest part of the pointers in the programs we consider. As far as this report is concerned, we assume all pointers have been removed from the program our transformation to dynamic single assignment form gets as input.

References

- [App98] A. Appel. *Modern Compiler Implementation in C*. Cambridge, 1998.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., 1986.
- [CBF97] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–226, 1997.
- [CFR⁺91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [CV00] F. Catthoor and A. Vandecappelle. DTSE script illustrated on cavity detection demonstrator. Technical report, IMEC, August 2000.
- [CWD⁺98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.
- [DGCDM96] E. De Greef, F. Catthoor, and H. De Man. Reducing storage size for static control programs mapped onto parallel architectures. In *Proceedings of Dagstuhl Seminar on Loop Parallelization*, 1996.
- [Fea88a] P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, pages 429–441, St. Malo, France, 1988.
- [Fea88b] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.
- [Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
- [Gar66] B. S. Garbow. Integer-preserving gaussian elimination, Nov. 21, 1966.
- [GC95] Martin Griebel and Jean-Francois Collard. Generation of synchronous code for automatic parallelization of while loops. In *European Conference on Parallel Processing*, pages 315–326, 1995.
- [Kie00] B. Kienhuis. Matparser: An array dataflow analysis compiler. Technical report, University of California, Berkeley, February 2000.
- [Kie02] Bart Kienhuis. private communication, 2002.
- [KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Symposium on Principles of Programming Languages*, pages 107–120, 1998.

- [LG95] Lengauer and Griebel. On the parallelization of loop nests containing while loops. In *AIZU: Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*. IEEE Computer Society Press, 1995.
- [Mas94] V. Maslov. Lazy array data-flow dependence analysis. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 311–325, Portland, Oregon, 1994.
- [Muc97] S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.
- [Pug91] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, 1991.
- [QRR96] P. Quinton, S. Rajopadhye, and T. Risset. On manipulating \mathbb{Z} -polyhedra. Technical report, Institut de Recherche en Informatique et Systemes Aleatoires, 1996.
- [RP95] Lawrence Rauchwerger and David Padua. Parallelizing while loops for multi-processor systems. In *Proceedings for the 9th International Parallel Processing Symposium*, pages 347–356, April 1995.
- [TA93] L. Thiele and U. Artz. On the synthesis of massively parallel architectures. *International journal of high speed electronics and systems*, 4(2):99–131, 1993.
- [TBJC02] Remko Tronçon, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Storage size reduction by in-place mapping of arrays. In *Verification, Model Checking and Abstract Interpretation, VMCAI 2002, Revised Papers*, volume 2294 of *LNCS*, pages 167–181, 2002.
- [VJB⁺02] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catt-hoor. Advanced signal propagation. Technical Report CW 353, Department of Computer Science, K.U.Leuven, Leuven, Belgium, December 2002.
- [VJB⁺03] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catt-hoor. Advanced copy propagation for arrays. In *Languages, Compilers, and Tools for Embedded Systems LCTES'03*, pages 24–33, June 2003.