TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

OPTIMIZING OVERHEAD IN COMPILATION FOR SCAD ARCHITECTURES

Master Thesis

von

Max Stein

September 29, 2021

Technische Universität Kaiserslautern,
Department of Computer Science,
67653 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
Dr. Anoop Bhagyanath

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "OPTIMIZING OVERHEAD IN COMPILATION FOR
SCAD ARCHITECTURES" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 29.9.2021

_____
Max Stein

Abstract

Exposed datapath processors are capable of providing high performance since they reduce the register file pressure by allowing the compiler to move intermediate program values directly from the producer processing units to the consumer processing units. Synchronous Control Asynchronous Dataflow (SCAD) is an exposed datapath processor architecture that uses first-in-first-out (FIFO) buffers to store the input and output values of each processing unit. The FIFO buffers can hold more values in the processor chip compared to register file. SCAD executes a sequence of move instructions that transports values from the head of the output buffers to the tail of the input buffers via an interconnection network. The processing units fire when enough values are available in its input buffers and empty space is available in its output buffers to store the computed result.

The SCAD compiler strives to generate move instructions to execute the source program by always directly communicating intermediate program values between the processing units, thereby eliminating the use of any central register file to improve the processor performance. However, this is not always possible with a limited number of processing units (or buffers). With a limited number of buffers, the compiler cannot always guarantee that the relevant values are found at the head of the output FIFO buffers for direct communication. In this case, some values are stored in the main memory and loaded when necessary. However, the slow main memory access adversely affects the processor performance. This thesis presents a less expensive alternative technique by rotating the values currently at the output buffer heads through the input buffers to access the relevant values for direct communication. To this end, the source program is transformed by adding necessary copy assignments that results in the rotation of copied values during the program execution. The values to be copied (or rotated) are wisely chosen from a buffer interference graph with the objective of minimizing the overhead in terms of number of copy operations (or rotations). The copy assignments are repeatedly inserted until all the variables in the resulting program can be successfully mapped to the available processing units in the SCAD machine.

Contents

1 Introduction

1.1 Motivation

Conventional processors use registers to store intermediate program results, where instructions read their operands from registers and write their results to registers. In order to scale up the utilization of instruction-level parallelism (ILP) offered by applications, it is necessary to scale up the number of processing units (PU) as well as the number of registers and read/write ports in the register file. However, it is not easy to increase the number of registers or ports in a register file due to its poor scalability concerning the area, power consumption and access time [Rix+00; ZK98]. This leads to a limited use of ILP contained in application programs by the conventional processors. Exposed datapath architectures (XPDA) avoids register-file bottleneck by allowing the compiler to bypass register usage. To this end, these architectures reveal their internal datapath to the compiler so that the compiler can transport the intermediate program values directly from the producer PUs to the consumer PUs. This is termed as direct instruction communication or direct data routing. The direct communication of values between PUs reduces the demand for registers and bandwidth in a register file, and allows increased use of ILP.

Synchronous Control Asynchronous Dataflow (SCAD) [BJS15; BJS16; BS16; BS17a; BS17b; Bha20] is an exposed datapath architecture that employs first-in-first-out (FIFO) buffers at the inputs and outputs of PUs. The FIFO buffers can hold more values in the processor chip compared to register file. The output buffers are connected to the input buffers using any general (parallel) interconnection network. SCAD is programmed by a sequence of move instructions that transport values from the head of output buffers to the tail of input buffers. Computation is performed by the PUs as soon as enough data is available in its input buffers and empty space is available in its output buffer to hold the computed result. The input values are consumed from the input buffers and the computed result is produced in the output buffer.

Unlike other XPDA architectures, the SCAD architecture does not use any central register file for program execution. Instead, the move code generated by the SCAD compiler always transports the intermediate results of a program directly between the PUs in the SCAD machine. To this end, the compiler has to determine an appropriate PU assignment (which value is produced by which PU) and a partial variable order (order of values produced by each PU), so that relevant values are always found at the head of output buffers of producer PUs for direct transportation to the consumer PUs. This is, however, a challenging task due to FIFO buffer access restrictions especially if not enough

PUs are available in the SCAD machine. Clearly, the access restrictions lessen with an increasing number of PUs (consequently more numbers of output and input buffers) in a SCAD machine. Optimal compilation based on satisfiability (SAT) solvers [BS16; BS17a; BS17b], and heuristics [Bha20], are developed to compile source programs to move code that works with a minimal number of PUs.

When enough PUs are not available in a SCAD machine for execution by direct instruction communication, the compiler resorts to storing some intermediate program value(s) in the main memory (RAM) and loading them when necessary. This allows the compiler to access the remaining relevant values from the head of output buffers for direct transportation between PUs. However, the slower main memory access degrades processor performance. An alternative is to rotate values through the FIFO buffers in SCAD until relevant values can be accessed from the head of output buffers. Rotating a value involves first transporting the value from the output buffer head to an input buffer tail. When this value eventually seeps into the input buffer head, the PU simply copies the value to the tail of its output buffer. Clearly, the rotation of values is still overhead in terms of additional load on the interconnection network and the PUs. Nevertheless, it is less expensive than accessing the main memory.

1.2 Contribution

This thesis presents a code generation technique for SCAD that systematically rotates values through the buffers, avoiding main memory accesses in SCAD machines with insufficient PUs for execution by direct instruction communication. The code generator is based on the heuristic presented in [Bha20] that considers the variable definitions and uses in the given program order for overhead-free (i.e., without any value rotations) move code generation. The heuristic determines a PU assignment by coloring a novel buffer interference graph of variables. An edge between two variables in the buffer interference graph indicates that these variables must be produced by different PUs for successful move code generation. In this work, we consider the so-called critical cliques of the buffer interference graph with regard to a given SCAD machine. A clique is considered critical if more PUs than available in the SCAD machine are required to produce the variables that form the clique's nodes. This thesis focuses on resolving the critical cliques by splitting appropriate variables (or nodes). To split a variable, we add a copy assignment (assignment of the variable to itself) at a suitable place in the source program, which manifests as a rotation of variable value during the program execution. The variables to split are wisely chosen to minimize the resulting overhead (i.e., number of copy assignments or rotations). The copy assignments are repeatedly added until the remaining program can be scheduled on the available PUs in the given SCAD machine.

1.3 Outline

The rest of the report is organized as follows: Chapter 2 explains the organization and the working of SCAD architecture. We also introduce the basics of code generation for SCAD, explaining the need for main memory accesses or overhead in terms of rotation of values. The new code generator is described in Chapter 3. The algorithm for the transformation of the source program by adding copy assignments is explained using an example. We continue with the same example demonstrating buffer size aware PU assignment followed by the generation of SCAD move code. Finally, Chapter 4 summarizes the thesis work and outlines the future work.

2 Preliminaries

2.1 SCAD Architecture

The SCAD architecture shown in figure 2.1 works as follows. There are several universal processing units which perform any kind of logic or arithmetic operation and which have two input and one output buffer. The input and output buffers have an additional address buffer and any PU has further one buffer for the opcode and one for the number of copies that should be produced. The CU fetches move instructions of the form (src, tgt) with output buffer src and input buffer tgt and puts if place available into the address buffer of src the value tgt and into the address buffer of tgt the value src. If one has in the input buffer tgt the value $(\text{src}, \perp)$ as first value of the form $(\text{address}, \perp)$ and in the head of the output buffer src the value (tgt, val) where val is a computed value then the value val is sent from src to tgt. If a PU has its input operands and space is available in the output buffer then the PU can fire. Hence, the CU fills the address buffers ahead and the data is sent later on when the data is available meaning that the dataflow is asynchronous.

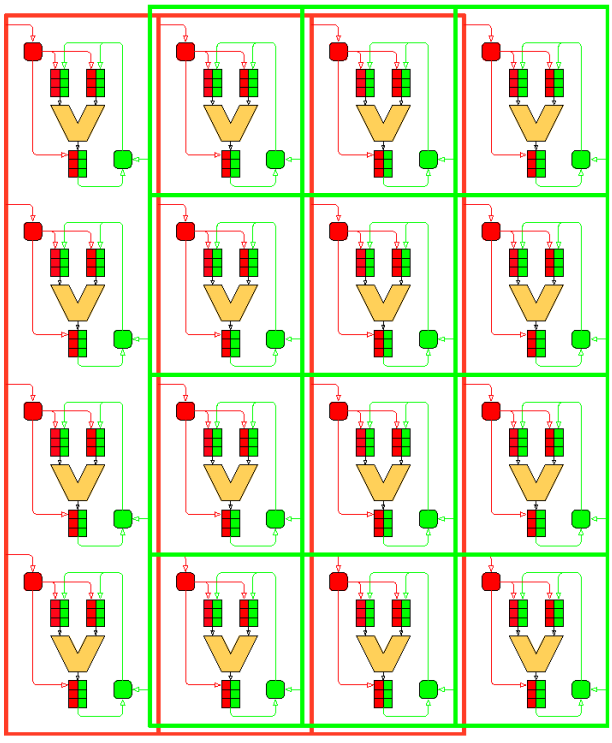


Figure 2.1: SCAD Architecture

2.2 Compilation Basics for SCAD

Considering the following code

$x_1 \leftarrow$ load mem_1
$x_2 \leftarrow$ load mem_2
$x_3 \leftarrow x_2 \times x_2$
$x_4 \leftarrow x_1 \times x_3$
store mem_4 x_4

it is not possible to schedule it on a single PU since there is an interference between the variables x_1 and x_2 where x_1 is produced before x_2 but x_2 is used before x_1. Thus there are two PUs needed assigning x_1 and x_2 to different PUs.

2.3 Introducing Overhead in SCAD Compilation

In the preceding example the interference between x_1 and x_2 can be resolved by introducing a copy operation for the variable x_1 as follows

$x_1 \leftarrow$ load mem_1
$x_2 \leftarrow$ load mem_2
$x_1 \leftarrow x_1$
$x_3 \leftarrow x_2 \times x_2$
$x_4 \leftarrow x_1 \times x_3$
store mem_4 x_4

Now with the help of this overhead operation the example can be scheduled on a single PU.

3 Overhead Optimized Compilation for SCAD

The main goal is to compute a buffer mapping that assigns to any variable a processing unit (PU) at which the variable value will be computed and produced in its output buffer. If a variable cannot be assigned to an available PU, it is mapped to a memory address that holds its value. In the latter case, the variable value is computed by a reserve PU (PU 0) which handles only sequential computations where the computed result is always stored to the main memory. The Section 3.1 presents an algorithm to compute the buffer mapping, explaining the application of this algorithm in detail on a bubble sorting example.

3.1 Transform Source Program

Algorithm 1 Computing the buffer mapping

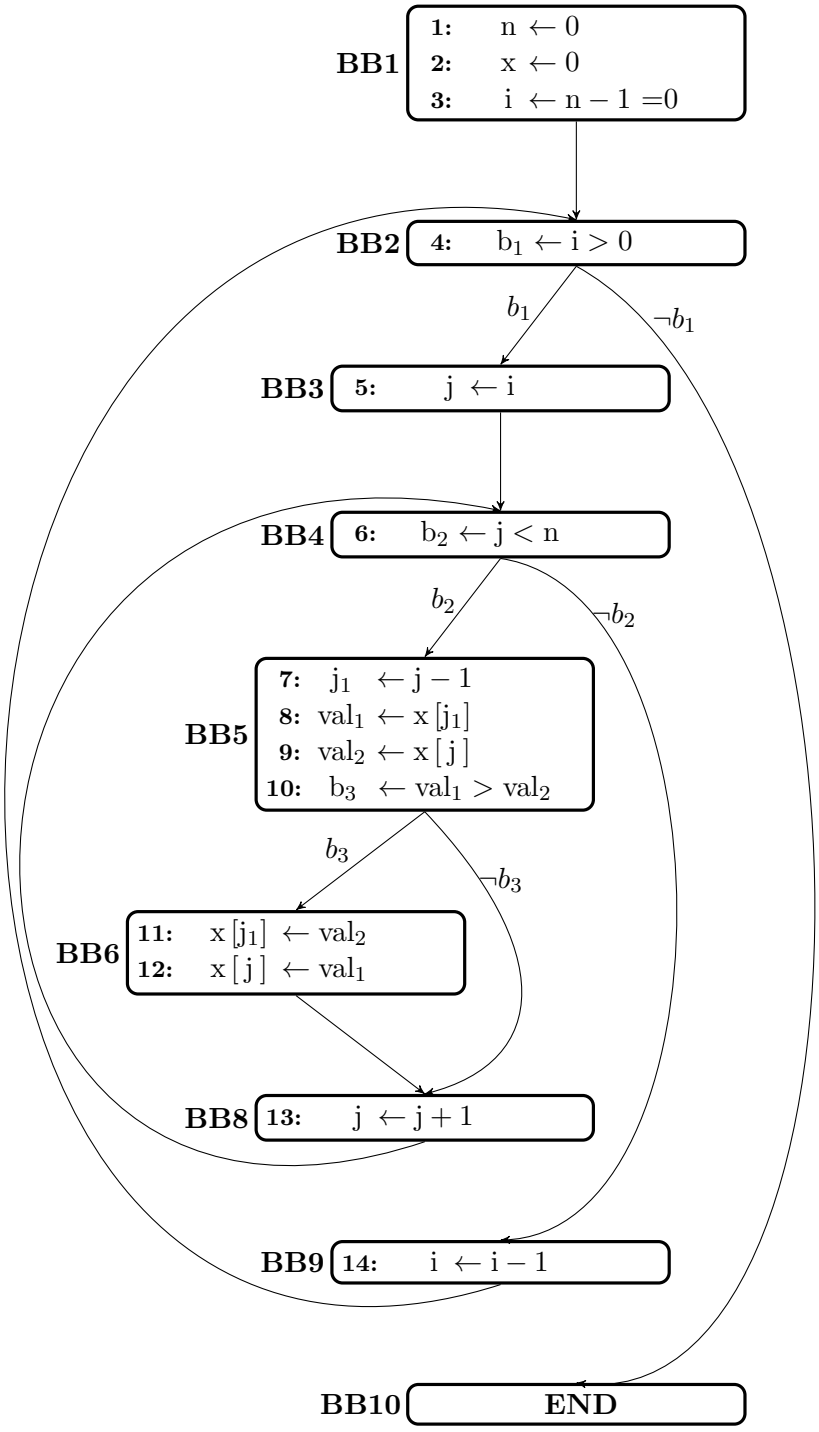
```
1: procedure EliminateCriticalCliques
2:     repeat
3:         Subdivide unmappable variable v into mappable sub-variables
4:     until all variables mappable
5:     repeat
6:         Choose variable v mostly involved in critical cliques
7:         Subdivide v into sub-variables
8:     until no critical clique exists
9: procedure Bufmap
10:    EliminateCriticalEdges
11:    EliminateUnboundedLoopVariables
12:    repeat
13:        EliminateCriticalCliques
14:        Map unmapped variable v to a PU
15:    until all variables mapped
```

The bubble sort procedure in 2 will be used for the illustration of the buffer mapping computation procedure shown in 1. The bubble sort procedure in provided in the form of pseudo code in 2. The compiler intermediate representation in three-address-code format is shown in and respectively in Basic-Block-Form is given by the following algorithms. It works by inserting the next value (value number j) at the correct position in the already sorted subset (values from number j+1 up to n). The insertion which is done in the

inner while loop of the following algorithm works by comparing neighboring values and swapping if the value at the lower index is higher. Inside the Basic-Block-Form of the algorithm the outer while loop is represented by the Basic Blocks BB2, BB3, BB4, BB5, BB6, BB8 and BB9. The inner while loop is represented by the Basic Blocks BB4, BB5, BB6 and BB8. The if-branch where values are swapped is represented by the Basic Blocks BB5 and BB6. There are branch conditions which are given in the following pseudo code form by the statement $i > 0$ for the outer while loop, the statement $j < n$ for the inner while loop and the statement $val_1 > val_2$ for the if-branch. These branch conditions are computed and stored into the variables b_1, b_2 and respectively b_3 and the control data flow lets the program move depending on them into different directions.

Algorithm 2 Bubble Sort

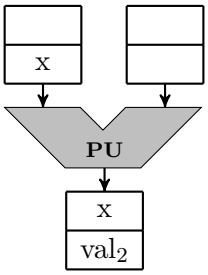
```
Require: n, x
  i ← n − 1
  while i > 0 do
      j   ← i
    while j < n do
        j1  ← j − 1
      val1 ← x[j1]
      val2 ← x[ j ]
      if val1 > val2 then
          x[j1] ← val2
          x[ j ] ← val1
        j   ← j + 1
      i   ← i − 1
```



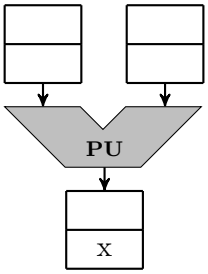
Algorithm 1: Bubble Sort

Step 10:
Inside the Bufmap procedure there are first the two preparation methods EliminateCriticalEdges and EliminateUnboundedLoopVariables needed. Concerning the first method an edge (s, e) between the Basic-Blocks s and e is called critical if there is more than one edge going out from s and more than one edge going into e. The critical edge is eliminated by simply inserting between s and e one further Basic Block. Inside the above Bubble Sort algorithm there is only one critical edge between BB5 and BB8 which is resolved in the following algorithm.

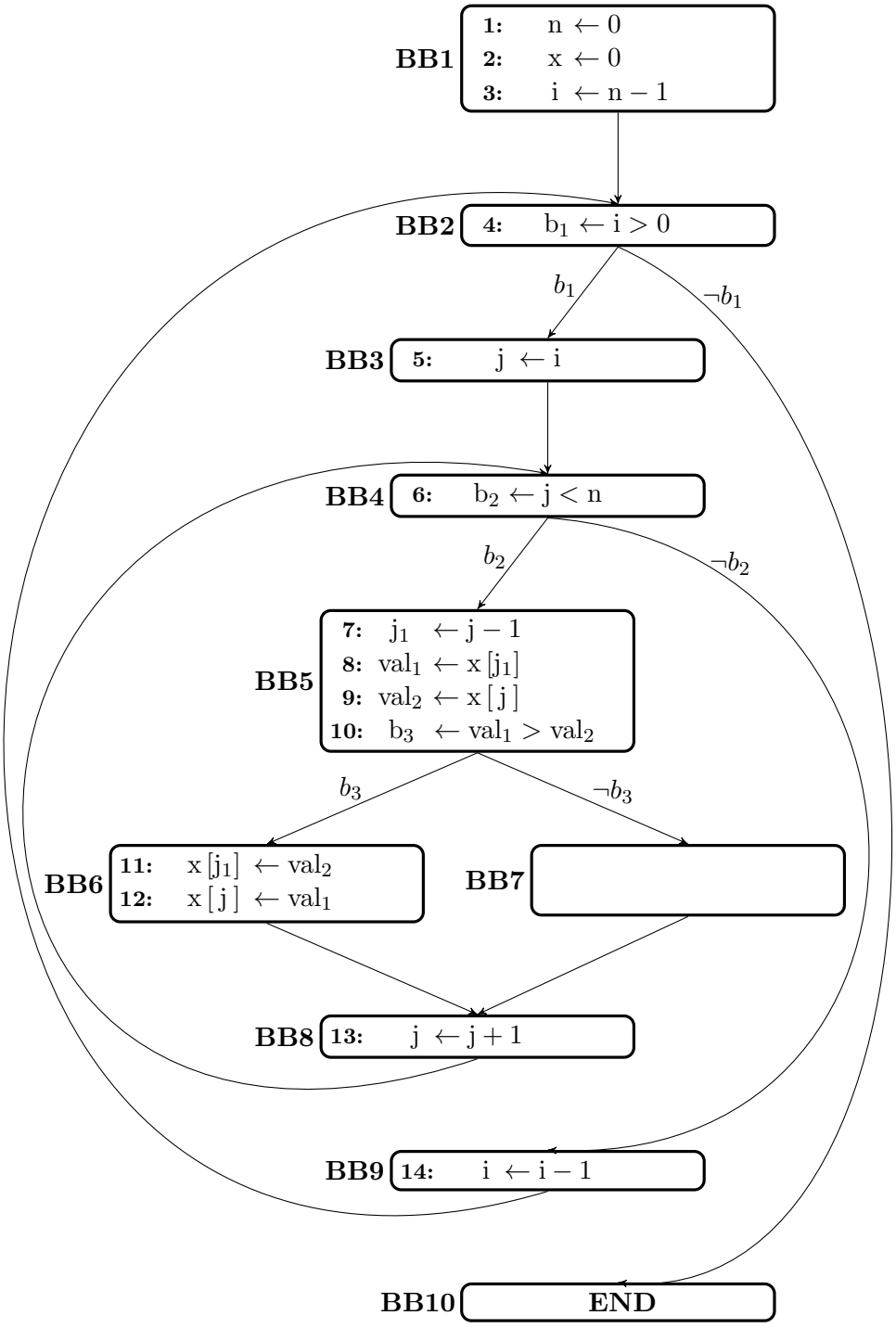
The reason for eliminating critical edges is to ensure a unique buffer state at the beginning of Basic Blocks. If e.g. x and val_2 would both be mapped to the same PU then in the above algorithm coming from BB5 into BB8 would cause the following buffer state at the beginning of BB8



and coming from BB6 into BB8 yields (since the values x and val_2 have been used)

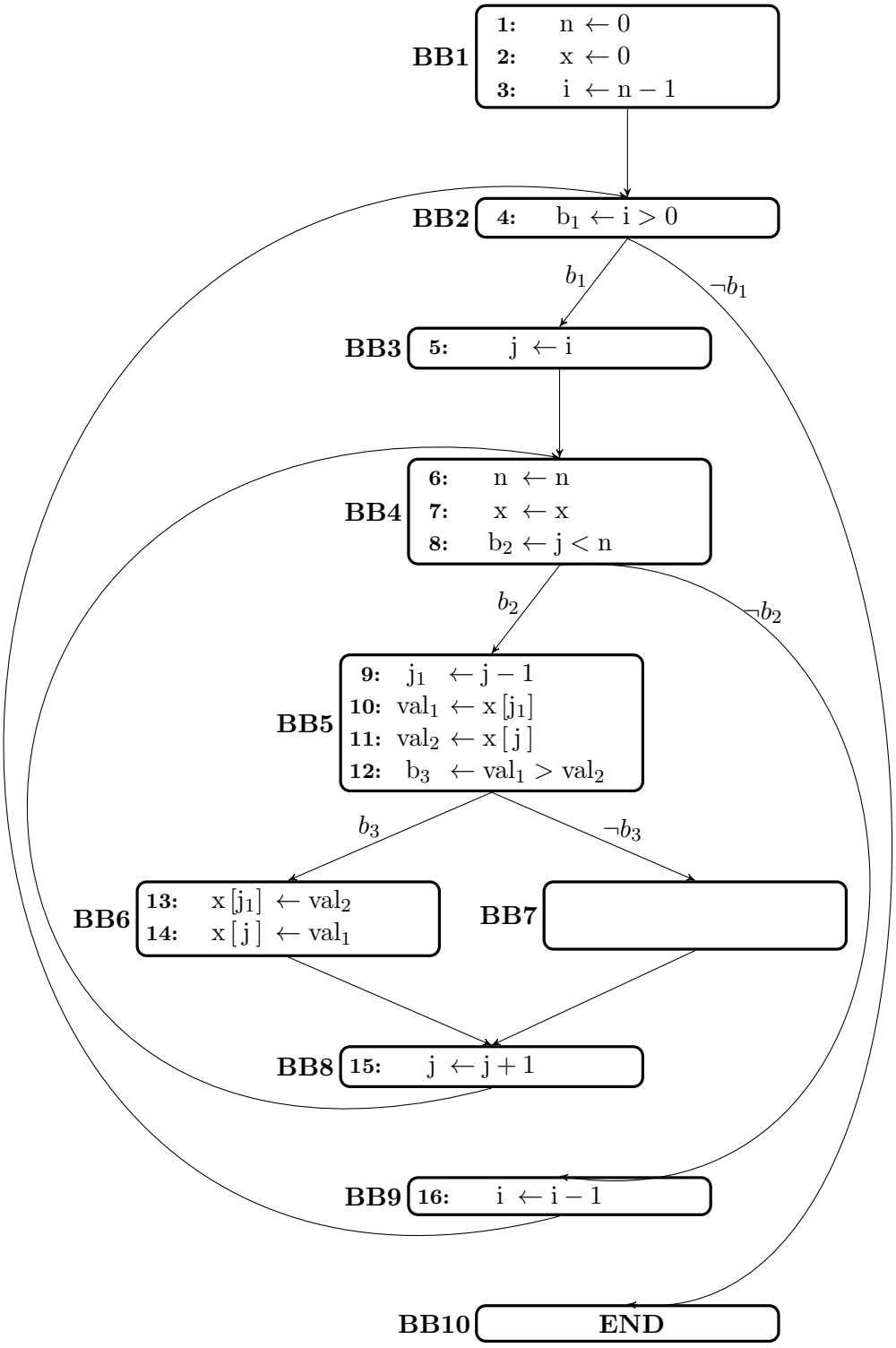


which is a different buffer state.



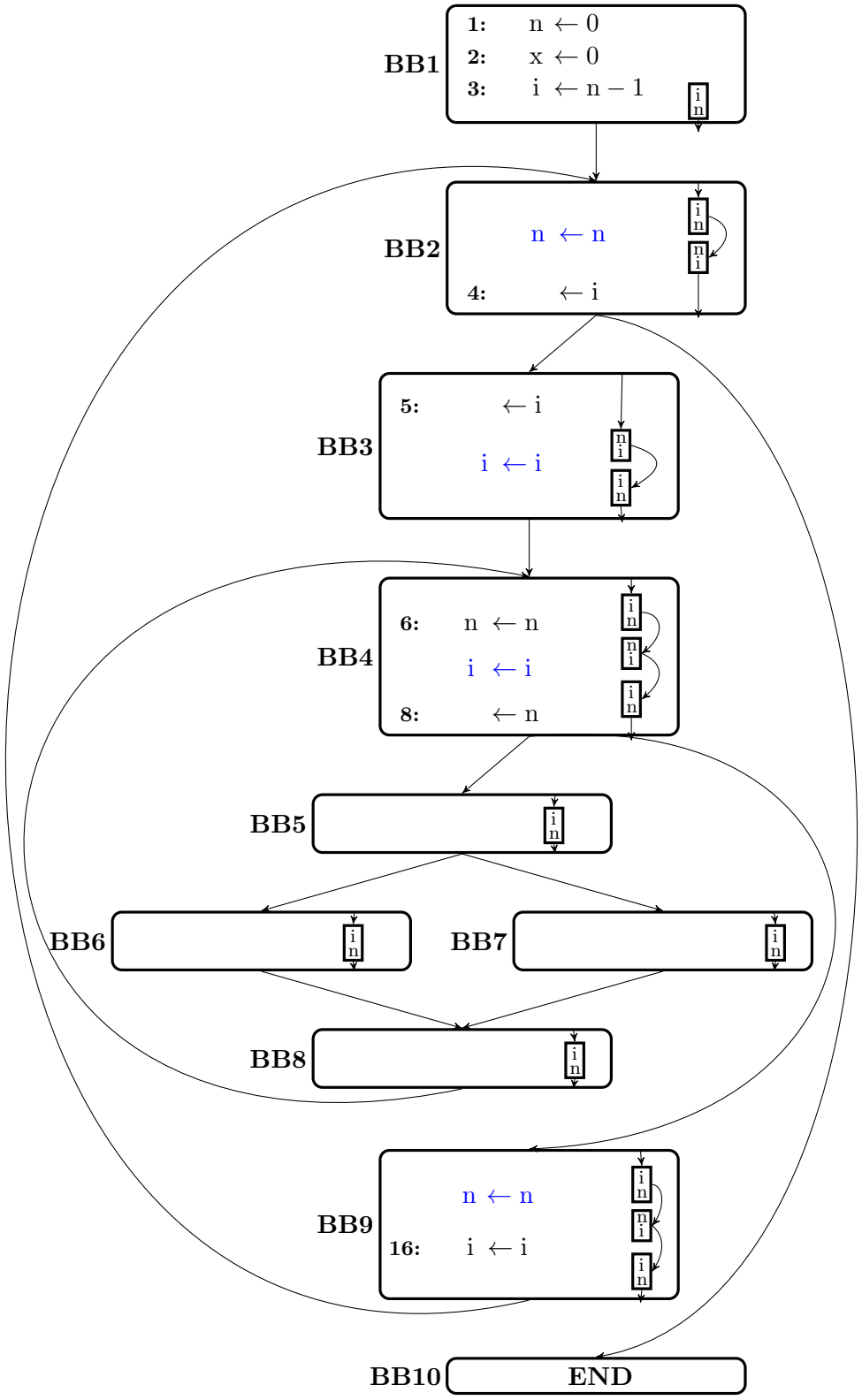
Algorithm 2: Bubble Sort

Step 11:
Concerning the method EliminateUnboundedLoopVariables there will for any loop in which a variable is used but not defined a copy operation of this variable be inserted inside the loop. This is necessary since else it would even not be possible to count the number of uses of such variables since it would be infinite. There are only the variables n and x used but not defined in the two while loops which can be resolved by placing a copy operation for both variables inside BB4 yielding the following algorithm.



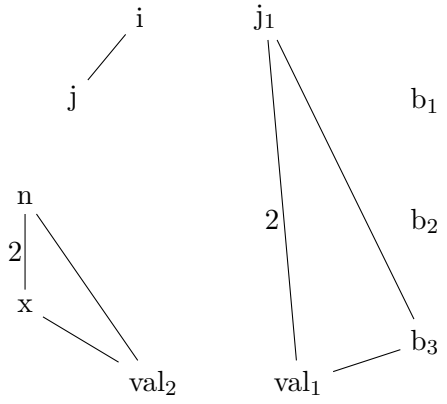
Algorithm 3: Bubble Sort

The notion of interference between two variables stands for the number of copy operations that are needed when the two variables are mapped to the same PU. E.g. the interference between the variables n and i is equal to 4 since as illustrated in the following algorithm there are 4 copy operations (drawn in blue) needed. There is at any command line the state which is the order between n and i in the FIFO-buffer given. Inside the command program there are only uses and definitions of n and i shown. At any point where the order doesn't fit meaning a variable is used which is behind the other in the FIFO-buffer then a swap (which is equal to a copy operation) is performed. Furthermore there is the notion of clique denoting a set of variables where between any two of them there is an interference. A clique will be called critical if the number of assignable PUs (which is the union of all the assignable PUs of all the variables inside the clique) is smaller than the number of variables inside the clique. Thus, a critical clique means that the variables inside can not all be assigned to PUs since the PUs need to be different due to the pair-wise interference of the variables and there are not enough assignable PUs. Thus if a critical clique exists then there are copy operations necessary or variables need to be mapped to the main memory.



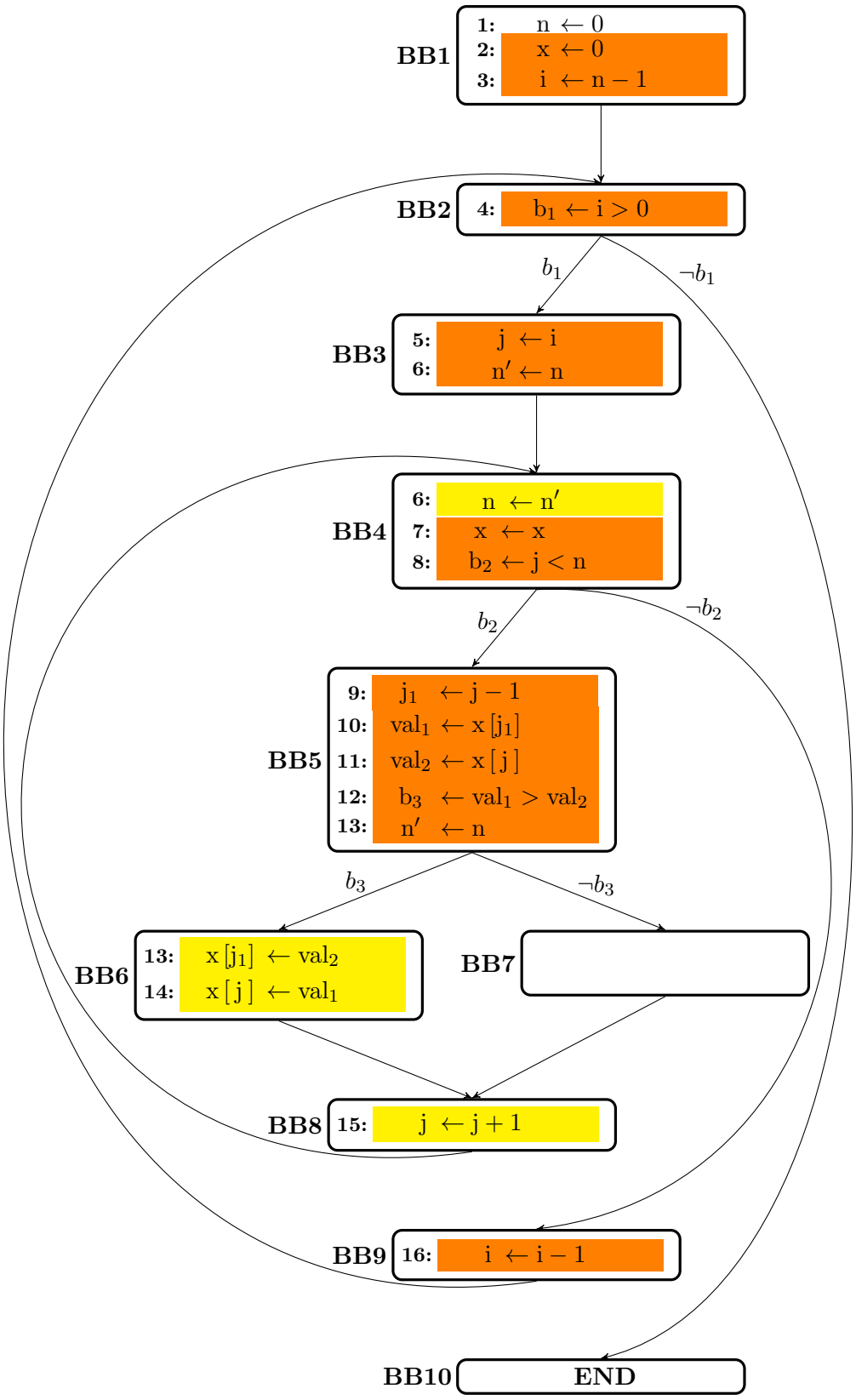
Algorithm 4: Bubble Sort

Step 6:
There will first all critical cliques be computed. Then a minimal set of pairs of variables resolving these critical cliques, that is any critical clique contains one of these pairs, is computed which is given by the following interference graph wherein any edge stands for a pair of variables. In other words omitting the interferences for any of the variable pairs would result in an interference graph without any critical cliques. The variables with maximal sum of interference values of all pairs they are contained in can be chosen inside the Bufmap algorithm to be subdivided. This are the variables n, x, j_1 and val_1 which all have sum of interference values equal to 3.



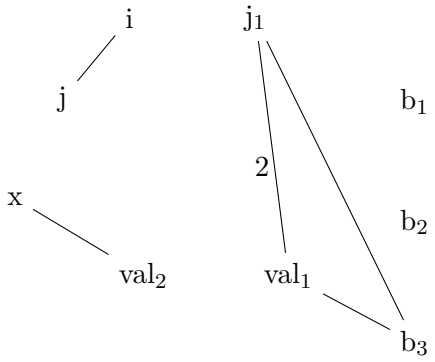
Interference Graph 1: Bubble Sort

Step 7:
Assume that in the previous step the variable n has been chosen to be subdivided. Then it will be ranged over all combinations of Basic Blocks to insert copy operations for subdividing the variable n where from the subdivision two sub-variables result and the set of Basic Blocks is minimal. The letter minimality means that any subset of the set of Basic Blocks results not also already in two sub-variables but only in a single sub-variable. Let e.g. the Basic Blocks BB3 and BB5 be taken, then the following algorithm shows the two resulting sub-variables n and n' where the liveness domains are drawn in orange and respectively yellow. Taking only BB3 or BB5 would both result in a single sub-variable, thus taking the Basic Blocks BB3 and BB5 satisfies the minimality criteria.



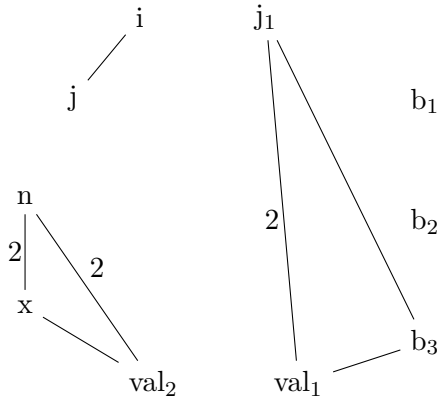
Algorithm 5: Bubble Sort

The Basic-Blocks BB3 and BB5 yielding the sub-variables n and n' are assumed to be taken in the following. Then firstly removing variable n yields the following critical cliques resolving interference graph (containing the variable pairs necessary to eliminate all critical cliques of the remaining variables without variable n).



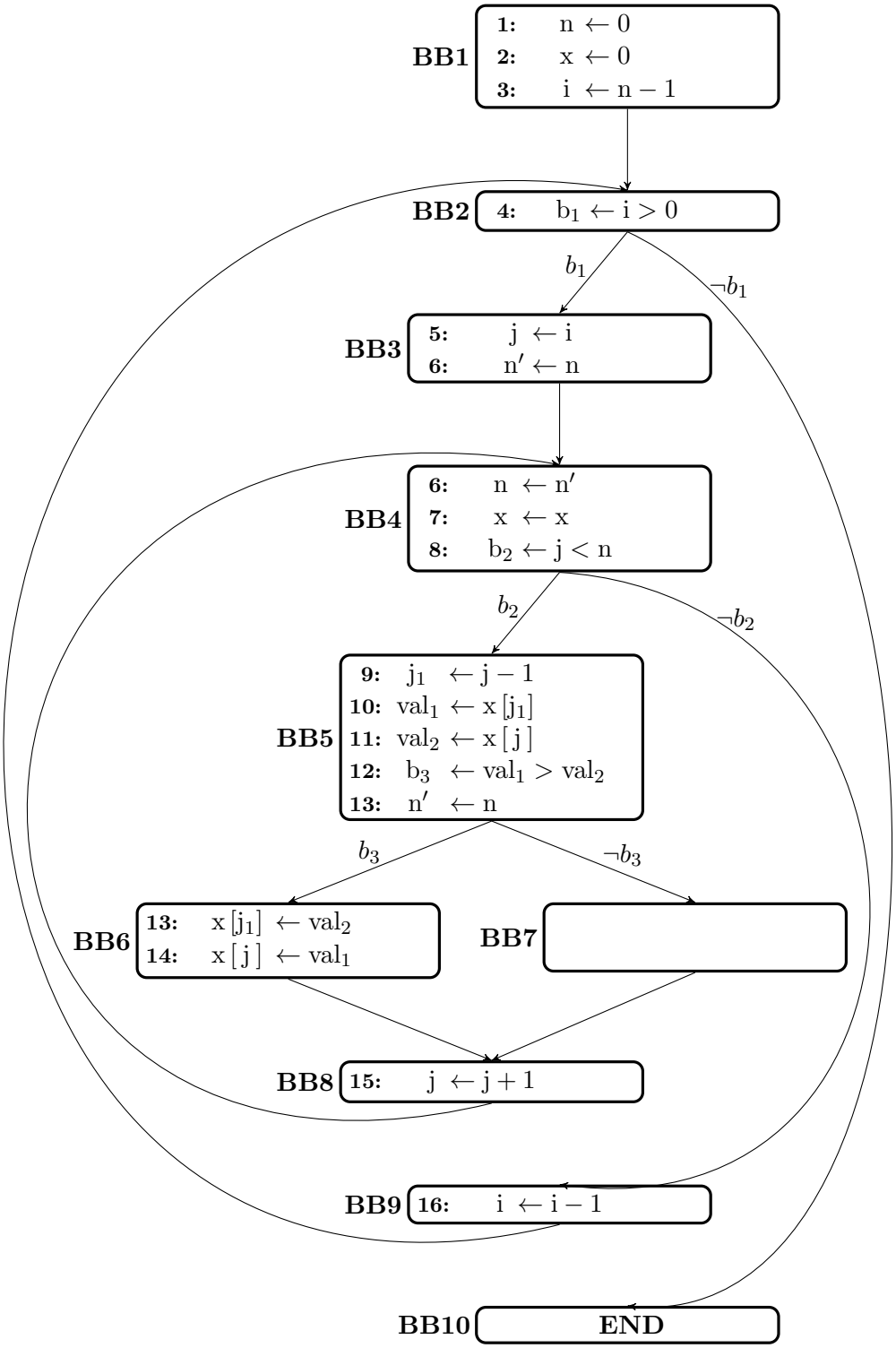
Interference Graph 2: Bubble Sort

For adjusting the positions of the copy operations there will firstly only the sub-variable n be considered. The copy operations will firstly be placed at the end of the Basic-Blocks which is represented by the following algorithm. Keeping the above critical cliques resolving variable pairs there are new unresolved critical cliques containing the sub-variable n arising as e.g. the critical clique (n, x, j_1, i). These critical cliques will be resolved by a minimal set of variable pairs containing the new sub-variable n. This is concretely done by adding the variable pairs (n, x) and (n, val_2) as shown in the following interference graph.



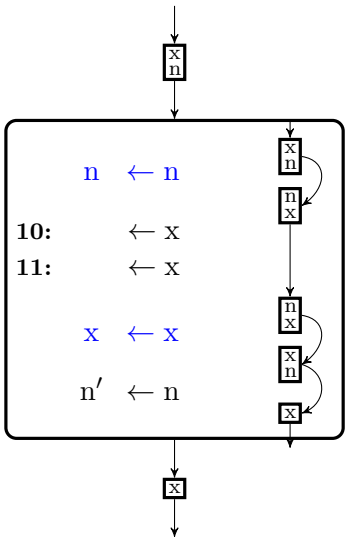
Interference Graph 3: Bubble Sort

Hence, the positions of the copy operations inside the Basic-Blocks BB3 and BB5 should be adjusted to minimize the interference to the variables x and val_2. Placing the copy operations secondly at the start of the Basic-Blocks will not yield any further unresolved critical clique, thus there will also no further variables be added to the set of variables to which the interference should be minimized.

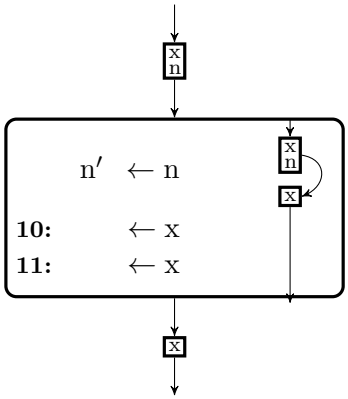


Algorithm 5: Bubble Sort

In order to minimize the interference between the variables n and x the copy operation inside BB5 should be placed before line 10 since otherwise there will two copy operations (drawn in blue in the following BB5 where the order of the variables n and x in the FIFO-buffer is shown) be necessary.

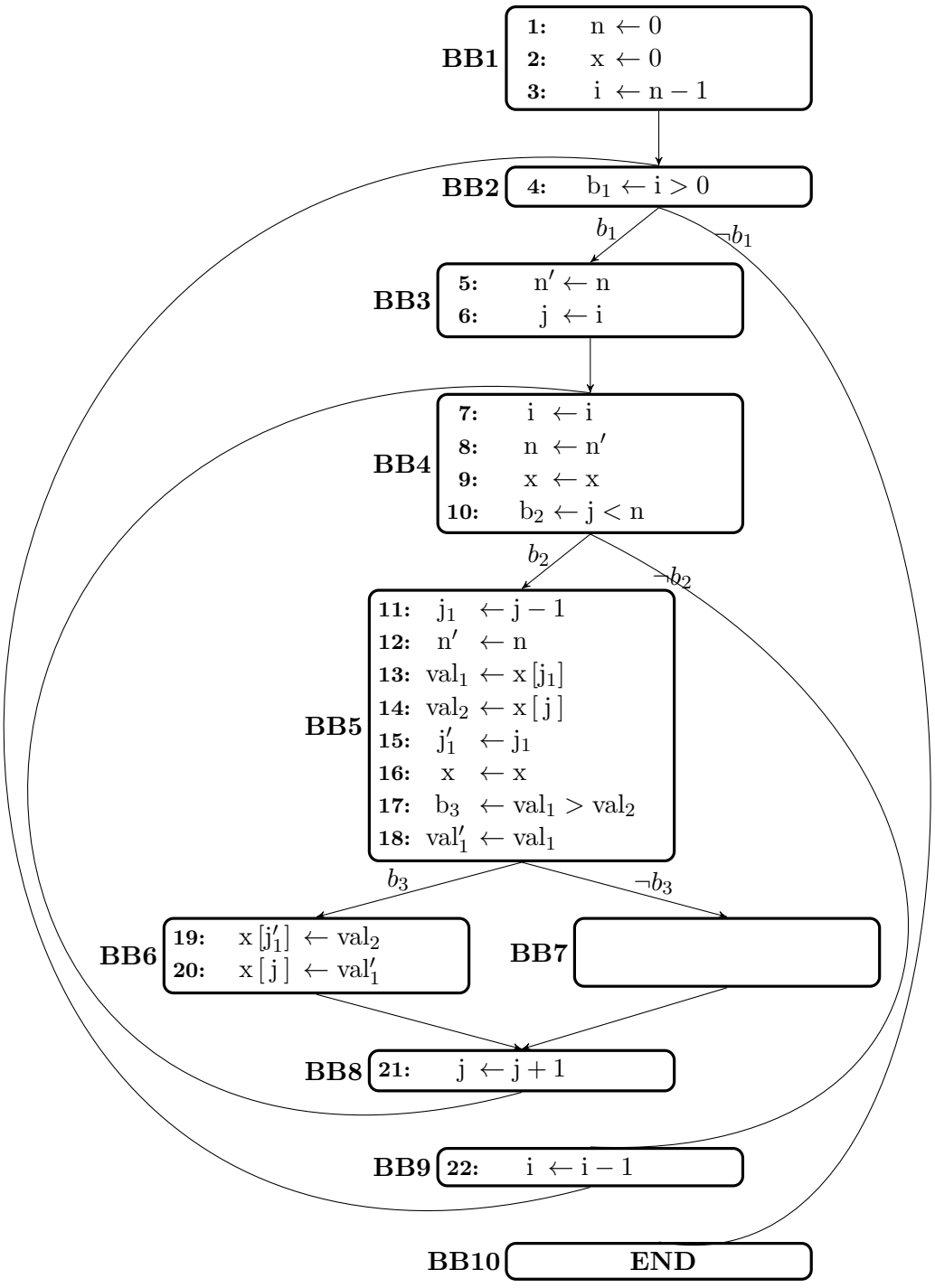


Placing the copy operation correctly before line 10 yields the following BB5 without any copy operation inside.



Concerning the interference between n and val_2 the copy operation of BB5 should with analogous argumentation to above be placed before line 12, thus it is sufficient to place the copy operation before line 10. Analogously to the above with sub-variable n one gets for sub-variable n′ that the interferences to the variables j and i should be minimized which is done by placing the copy operation of BB3 before line 5. Hence, taking both together yields that the copy operations need to pe placed before line 10 in BB5 and before line 5 in BB3 which is the final restriction on the position of the copy operations, thus the copy operations can now be placed. Repeating the above procedure until

all critical cliques are eliminated yields the following algorithm.

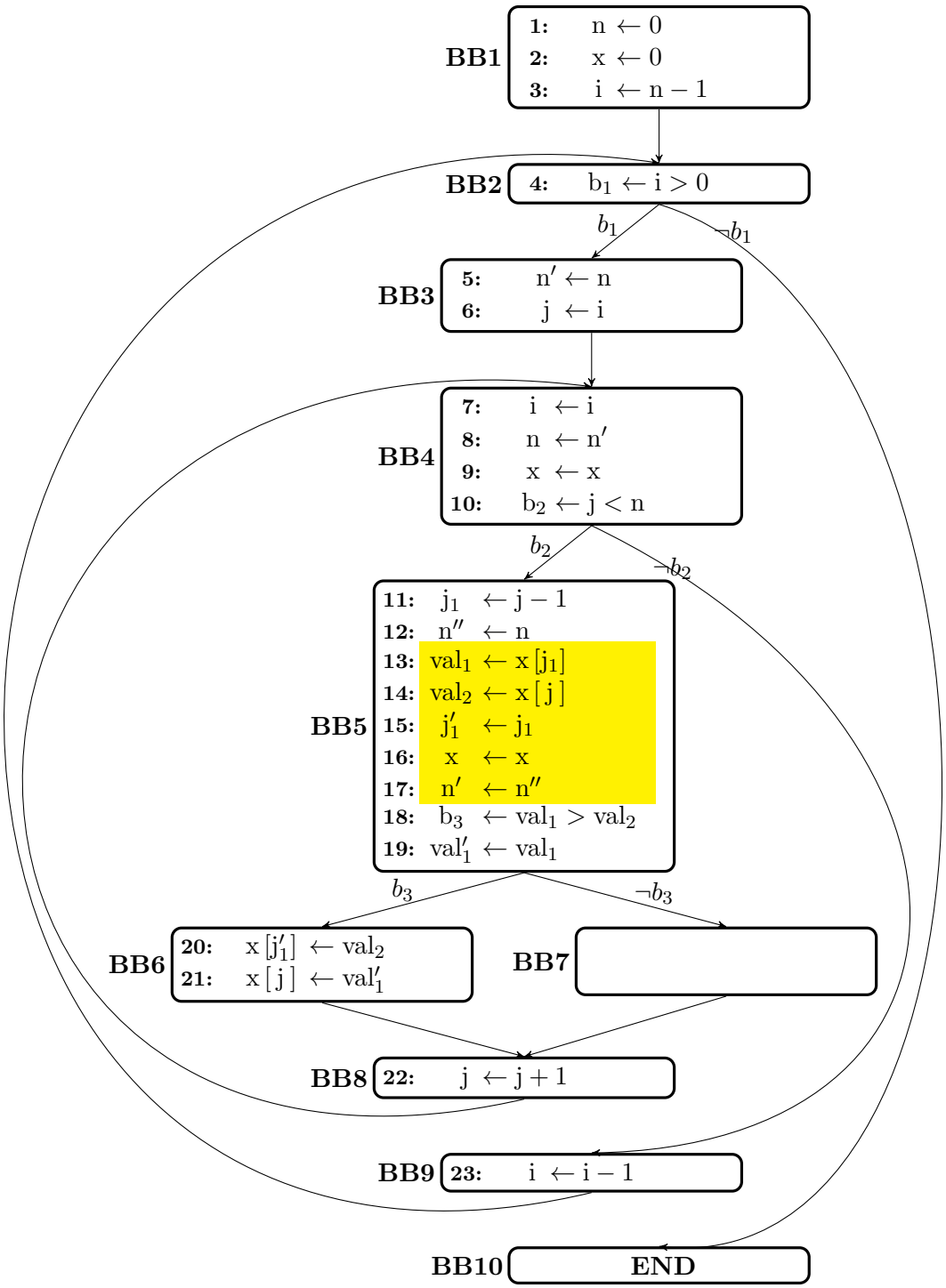


Algorithm 6: Bubble Sort

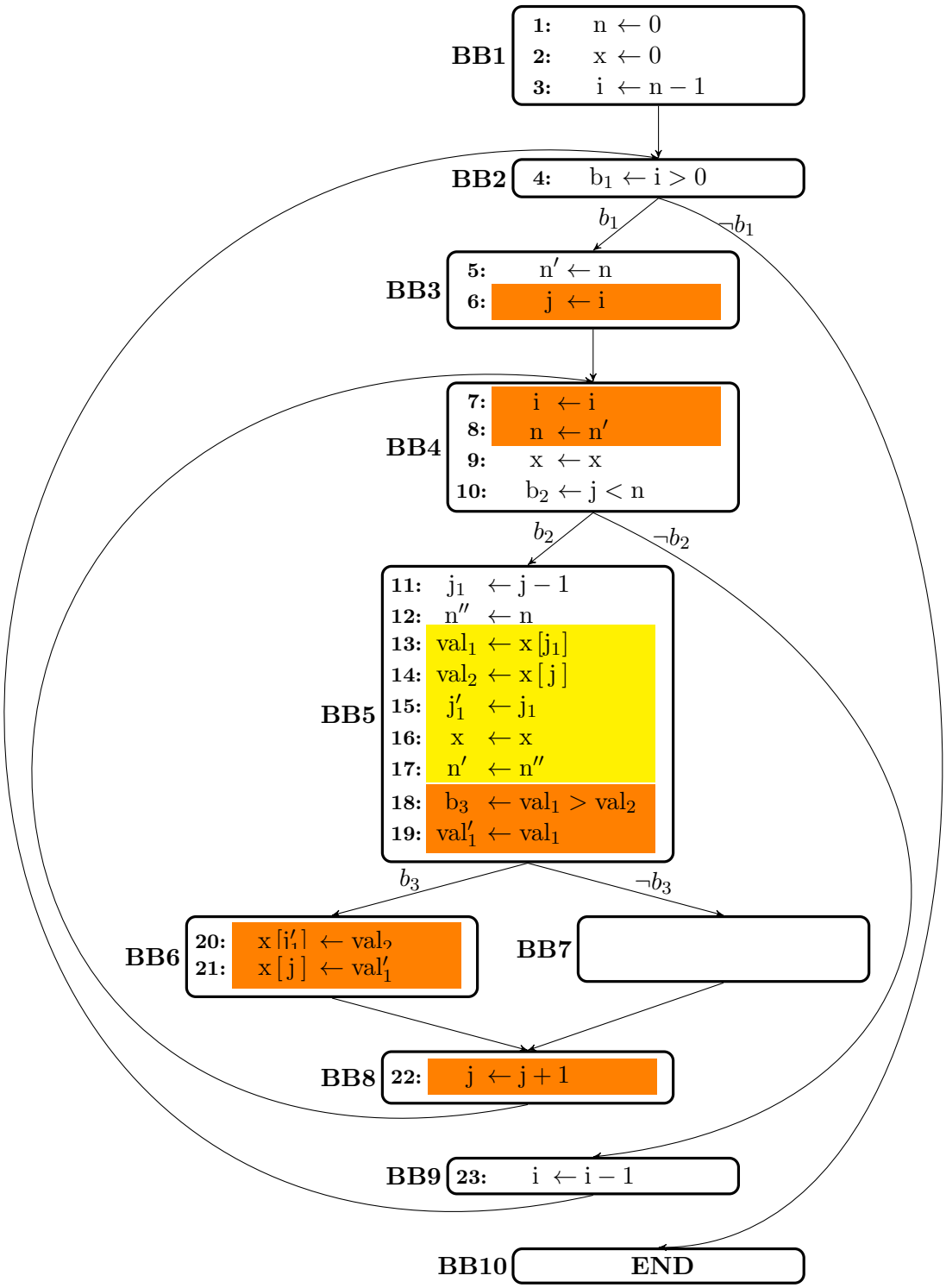
Step 3:
After elimination of all critical cliques and the following mapping of variables

PU0	PU1	PU2
n	j_1	i
val_2	val_1	j
	val'_1	
	b_3	

one gets for the first time an unmappable variable n′. The variable n′ is unmappable since mapping to PU0 is not possible due to the interference to val_1, mapping to PU1 is not possible due to the interference to val_2 and mapping to PU2 is not possible due to a buffer size problem at line 12 if input buffer and output buffer have a size of 2. Hence step 3 inside the Bufmap procedure will be applied which subdivides the variable into mappable sub-variables. The first sub-variable is defined at line 12 and is chosen to be mappable to PU1, thus its liveness domain (drawn in yellow in the following algorithm) is from line 13 to line 17. The liveness domain is always extended as far as possible as long as no interferences or buffer size problems arise. The first sub-variable can't be further extended since it would get an interference to the variable val_1. The first sub-variable could also have been chosen to be mappable to PU0 with same liveness domain but not to PU2 since the buffer would get overfilled directly at definition at line 12. Then the second sub-variable is added which starts at line 17 and which is chosen to be mappable to PU2. The liveness domain of this sub-variable (drawn in orange) can be extended to cover the whole remaining liveness domain of the initial variable n′. Hence the variable n′ is completely subdivided into mappable sub-variables.



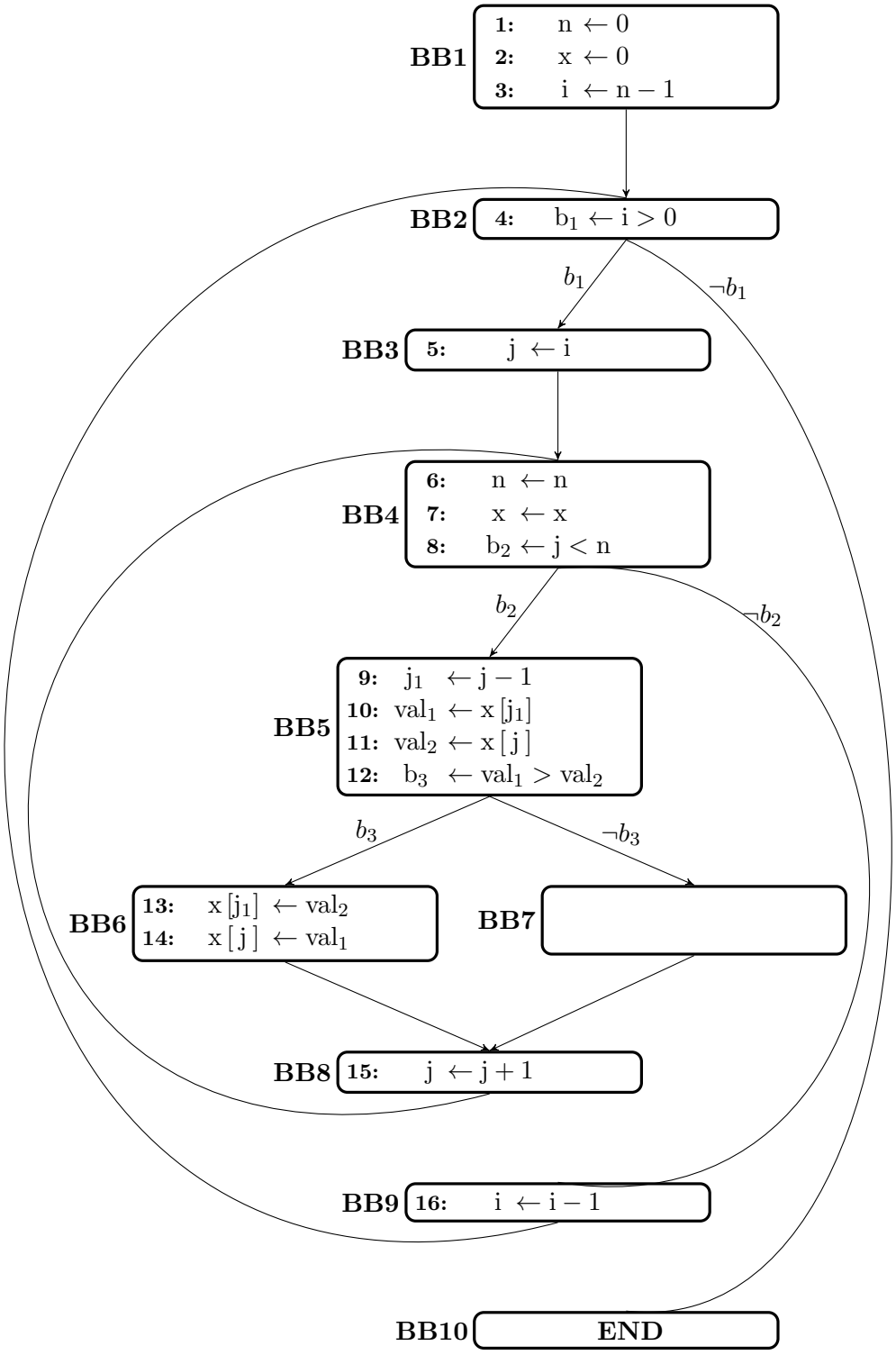
Algorithm 7: Bubble Sort



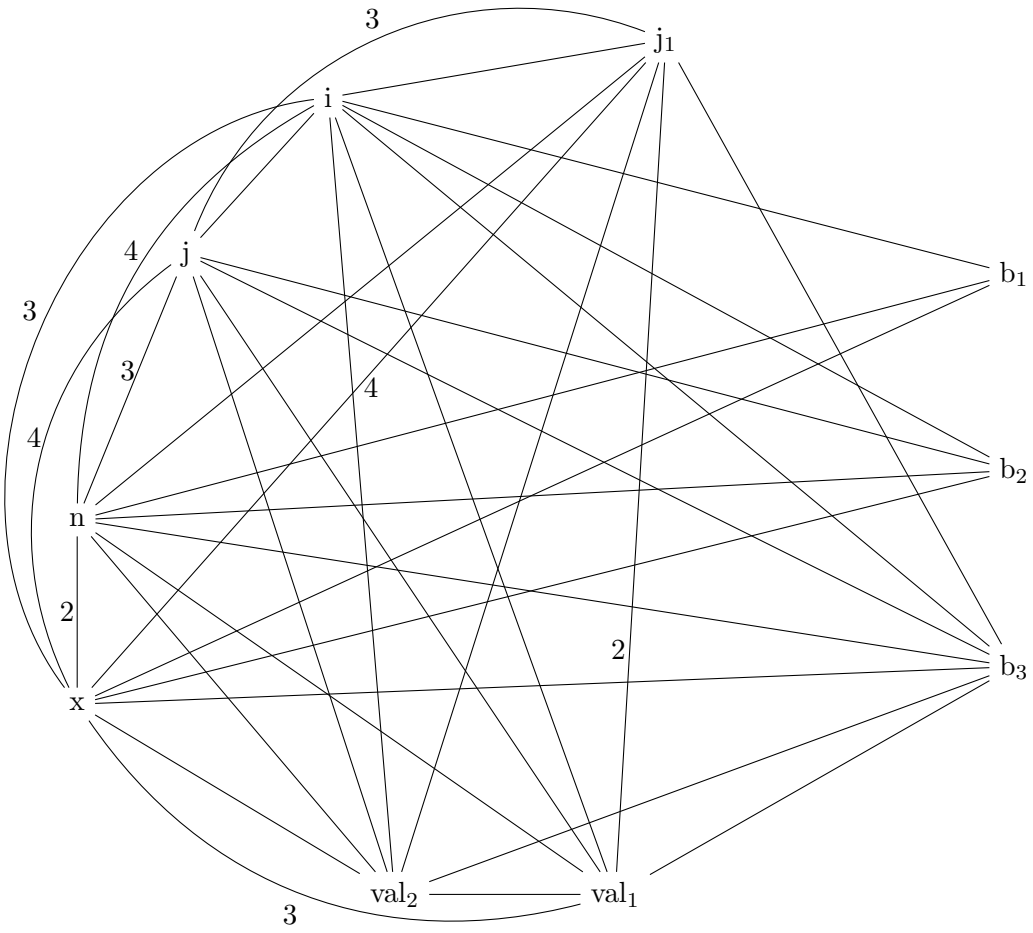
Algorithm 8: Bubble Sort

Step 14:
In this step an unmapped variable v with minimal number of assignable PUs is mapped to a PU p such that the number of critical cliques resulting from this mapping is minimal. The set of assignable PUs of the other unmapped variables needs to be updated by checking for any variable where the PU p is in the set of assignable PUs if it can still be mapped to p which is not the case if there is an interference to the variable v or some buffer size problem arises.

In the following example the SCAD move code for the Bubble Sort procedure is generated. Algorithm 1 shows the procedure in Basic Block Form and Interference Graph 1 shows the interferences between all pairs of variables.

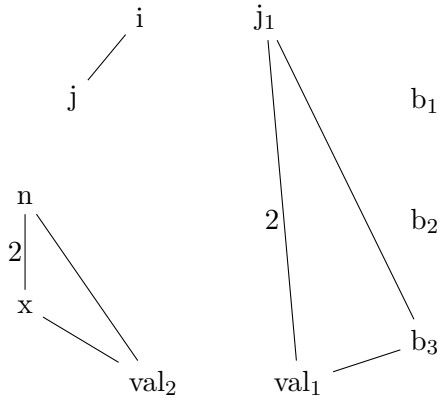


Algorithm 1: Bubble Sort



Interference Graph 1: Bubble Sort

The following Interference Graph 2 shows the variable pairs necessary to resolve all critical clicks.



Interference Graph 2: Bubble Sort

This are in total 9 interferences and copy operations should be inserted to eliminate the interferences to get rid of the critical cliques. Variables with highest total number of interferences are chosen to be subdivided which will be in the following the variable j_1 which has in total 3 interferences. The subdivision works by ranging over the different possibilities of splitting the variable into several sub-variables. Choosing BB5 as Basic Block splits the variable j_1 into the two sub-variables j_1 and j_1'. Finding the optimal line to insert the copy operation is done by firstly computing for any sub-variable the variables to which the interference should be minimized as described in section 3.1 step 7. Five copy operations are in the following necessary to get rid of the critical cliques.

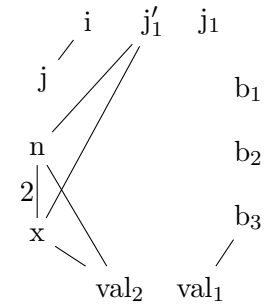
(1) For both the sub-variables j_1 and j_1' one gets the variables val_1 and b_3 where for j_1 one should put the copy operation before line 12 reducing the interference to val_1 and for j_1' one should put the copy operation behind line 10 reducing the interference to val_1 and behind line 12 reducing the interference to b_3. Line 12 is chosen to insert the copy operation yielding Interference Graph 3 and Algorithm 2.

(2) For the next copy operation the variable n is subdivided using the Basic Blocks BB3 and BB5 yielding the sub-variables n and n'. As set of variables to which the interferences should be minimized one gets for n the variables x, val_2 and for n' the variables i, j. Hence, the copy operation should be placed inside BB5 before line 10 due to the interference between n and x and inside BB3 before line 5 due to the interference between n' and j. This yields Interference Graph 4 and Algorithm 3.

(3) Next x is subdivided using the Basic Block BB5 yielding the sub-variable x. The interferences to the variables val_2 and j'_1 need to be minimized. Due to the interference to val_2 the copy operation should be placed before line 15 and due to the interference to j'_1 behind line 14. This yields Interference Graph 5 and Algorithm 4.

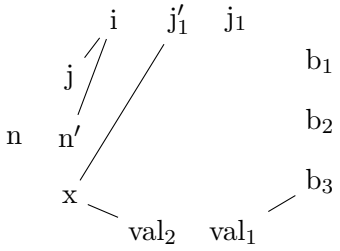
(4) Close to the preceding subdivision the variable i can be subdivided using the Basic Block BB4 yielding the sub-variable i with interferences to minimize to the variables j and n' which cause the copy operation to be put before line 5. This yields Interference Graph 6 and Algorithm 5.

(5) Finally variable val_1 is subdivided using the Basic Block BB4 yielding the sub-variables val_1 and val'_1 with interferences to minimize to the variable b_3 which cause the copy operation to be put at line 17. This yields an empty Interference Graph and Algorithm 6.



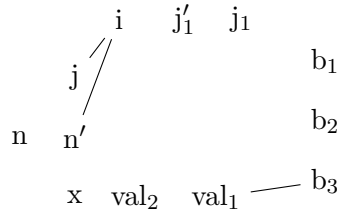
Interference Graph 3: Bubble Sort

Algorithm 2: Bubble Sort



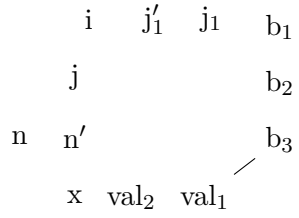
Interference Graph 4: Bubble Sort

Algorithm 3: Bubble Sort



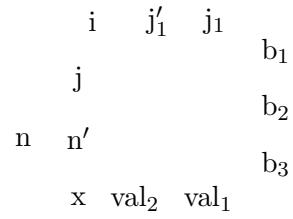
Interference Graph 5: Bubble Sort

Algorithm 4: Bubble Sort



Interference Graph 6: Bubble Sort

Algorithm 5: Bubble Sort



Interference Graph 7: Bubble Sort

Algorithm 6: Bubble Sort

3.2 Compute PU Assignments

In the following the variables are assigned to PUs where variables with lowest number of PUs to which they can be assigned are taken first and they are assigned to a PU such that the number of interferences to get rid of the critical cliques in the resulting interference graph is as small as possible. Using 3 PUs and an input buffer size of 2 and an output buffer size of 3 all the variables can be assigned without that any critical clique arises using the following PU assignment.

PU0	**PU1**	**PU2**
n	j_1	i
x	val_1	j
val_2	val_1'	n'
j_1'	b_3	

3.2.1 Consideration of Buffer Sizes

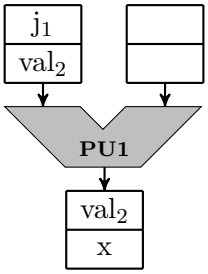
Changing the output buffer size from 3 to 2 the following variable assignment can be done without that any critical cliques arise.

PU0	PU1	PU2
n	j_1	i
val_2	val_1	
	val'_1	
	b_3	

The remaining variables x, j'_1, j and n' can now be assigned to the following PUs.

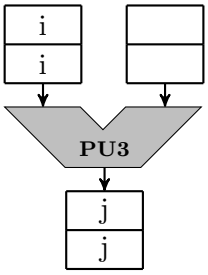
PU0	PU1	PU2
x		j
j'_1		n'

But assigning x and j'_1 to PU1 would cause the following buffer state after line 13



which yields a buffer size problem in line 14 since x can not be put on the already filled input buffer. Thus the PU assignment would cause a critical clique since the other variable would have no PU to which it can be assigned.

Similarly assigning j and n' to PU3 would cause the following buffer size problem at line 10 with buffer state



where n' can not be put on the PU. Thus the existence of a critical clique after the next assignment step is unavoidable and the variable j'_1 will be chosen to be assigned to PU0. This yields the critical clique containing only the variable x which has no PU to which it can be assigned.

For eliminating this critical clique a method will be used which subdivides the variable with no assignable PU again using copy operations into sub-variables mappable to PUs. This is done by first choosing a definition line of the variable and a PU as starting point for a sub-variable and then extending the liveness domain of this sub-variable as far as no interferences or buffer size problems arise. If the liveness domain of the sub-variable must end at command lines where the original variable is still live then the value will be transmitted to a next sub-variable using a copy operation.

Using this method the variable x is sub-divided into the variables x and x' as shown in Algorithm 7 where x can be assigned to PU0 and x' to PU1. This two variables can also directly be assigned to their PUs without that any critical clique arises which yields the new PU assignment table

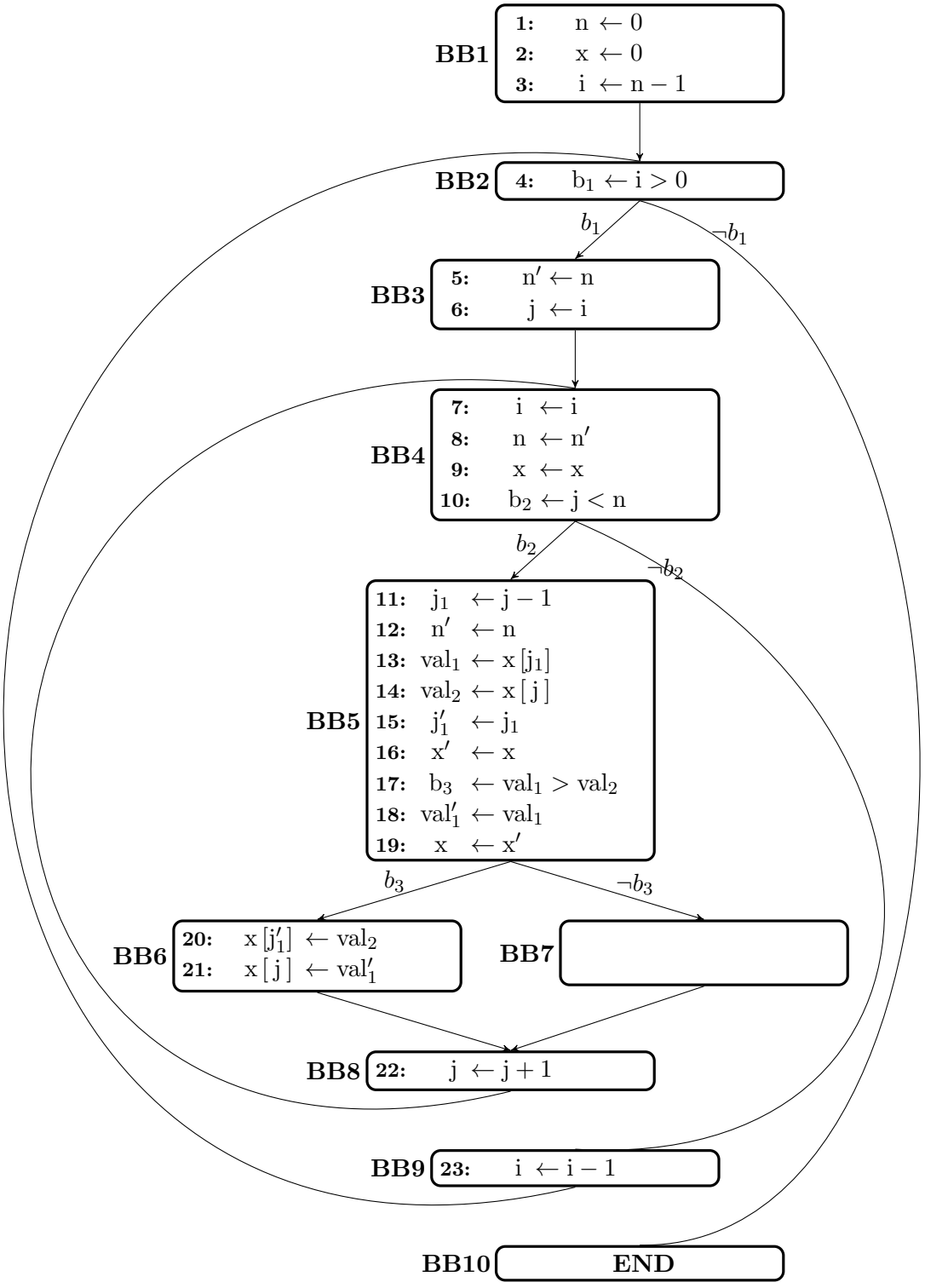
PU0	**PU1**	**PU2**
n	j_1	i
val_2	val_1	
j'_1	val'_1	
x	b_3	
	x'	

with remaining variables j and n' with assignable PUs

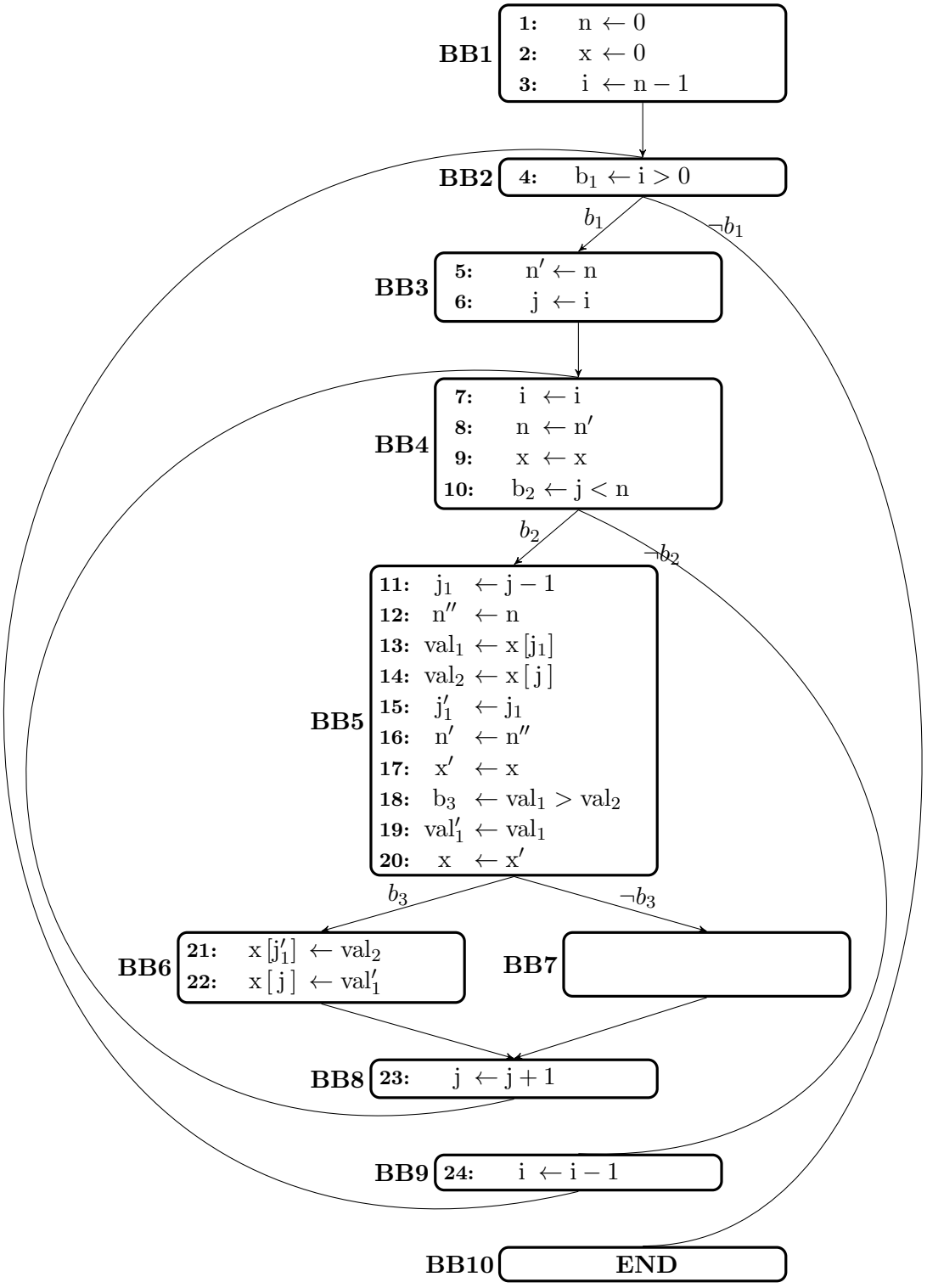
PU0	**PU1**	**PU2**
		j
		n'

Analogously the remaining buffer size problem is resolved assigning first j to PU2 and then splitting n' into n' and n'' as in the final Algorithm 8 which can then be assigned to PU2 and PU1 respectively which yields the final PU assignment table

PU0	**PU1**	**PU2**
n	j_1	i
val_2	val_1	j
j'_1	val'_1	n'
x	b_3	
	x'	
	n''	



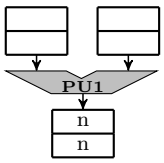
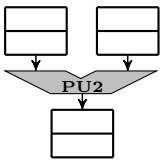
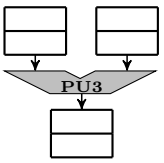
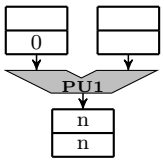
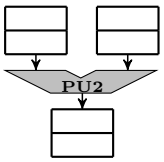
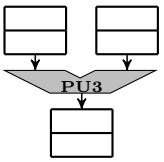
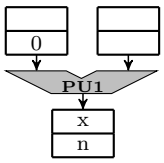
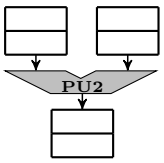
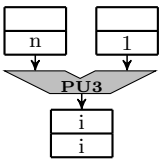
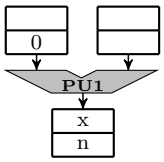
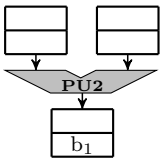
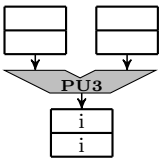
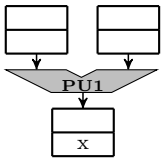
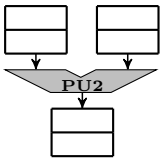
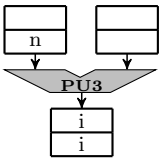
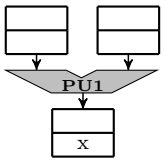
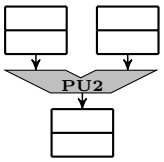
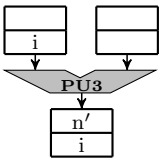
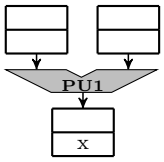
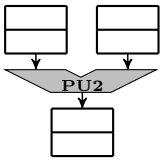
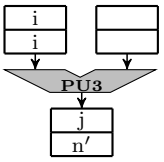
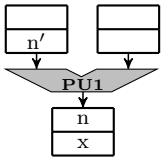
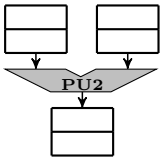
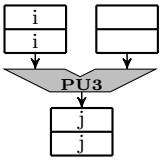
Algorithm 7: Bubble Sort

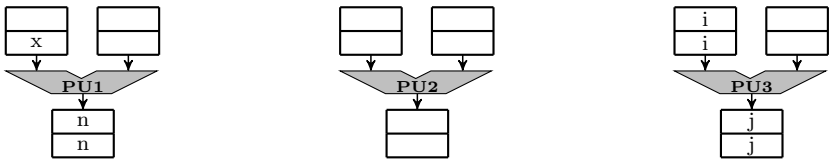
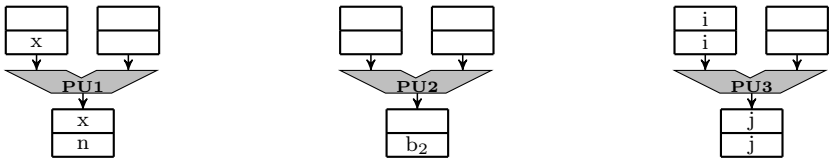
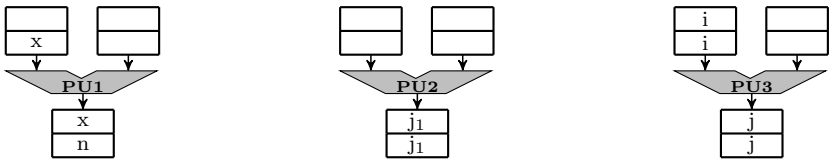
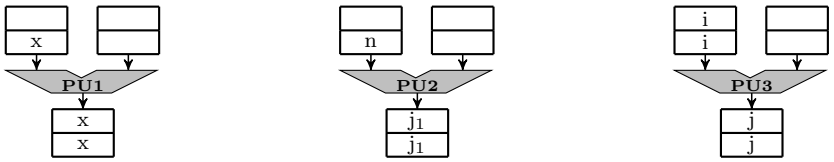
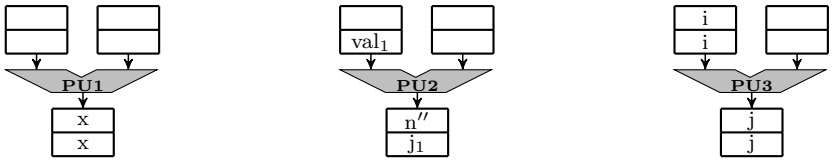
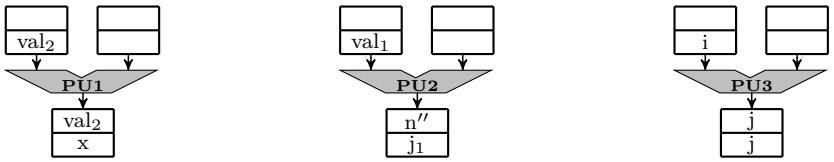
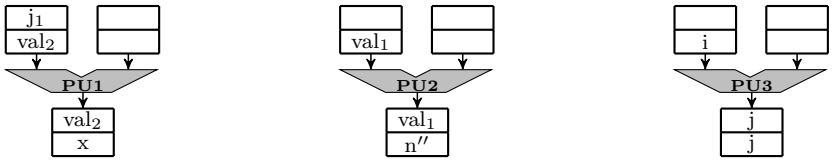
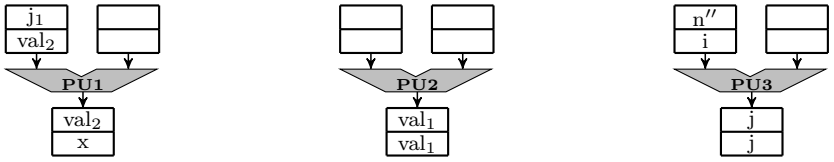


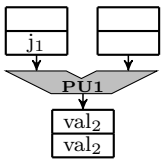
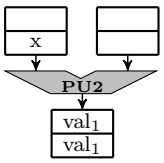
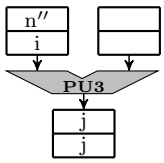
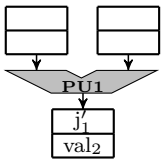
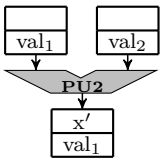
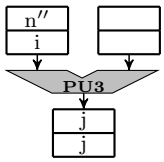
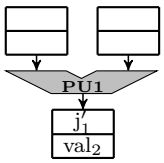
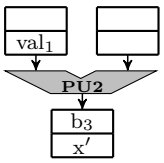
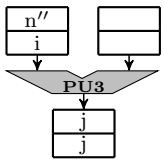
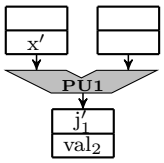
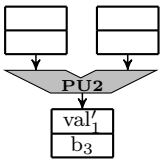
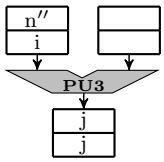
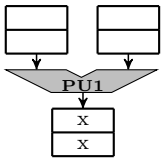
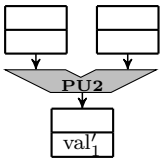
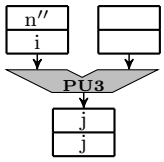
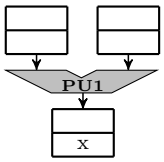
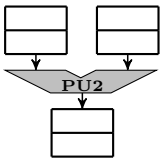
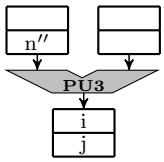
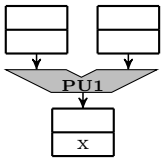
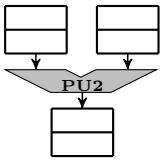
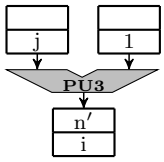
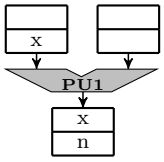
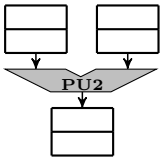
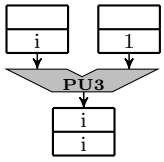
Algorithm 8: Bubble Sort

3.2.2 Buffer States during Execution

In the following the buffer states of the execution of the Bubble Sort example are given after any command line. There are only the data values shown while the opcode and number of copies are omitted.

line	buffer state		
1			
2			
3			
4			
5			
6			
7			
8			

line	buffer state
9	
10	
11	
12	
13	
14	
15	
16	

line	buffer state		
17			
18			
19			
20			
21			
22			
23			
24			

3.3 Move Code Generation

The move code is generated from the command program line by line using the bufmap. The data values are accessed in the order in which they occur inside the buffers of the PUs and unused data values in between are removed by discard operations. Assume in the following that the PUs are numbered by the number 0 for the CU, the number 1 for the LSU, the number 2 for the reserve PU and the numbers 3 to 5 for the three universal processing units.

1) Taking step 21, then inside PU3 there is the following buffer state

x	3
j_1'	1
val_2	1

Thus the values will be accessed in the order val_2, then j_1' and then x. The value val_2 is moved to LSU by MoveOutBf2InpBf$((3,0),(1,1))$. Then j_1' is moved to reserve PU by MoveOutBf2InpBf$((3,0),(2,1))$. Then x is moved to reserve PU by MoveOutBf2InpBf$((3,0),(2,0))$. Then for computing the store address on reserve PU the opcode is moved to reserve PU by MoveOpc2PU$((\text{AddN},1),2)$. Then the computed address is moved to LSU by MoveOutBf2InpBf$((2,0),(1,0))$. Then the value is finally stored by moving the store opcode to LSU which is done by MoveOpc2PU$((\text{false},0),1)$. Thus all together the following sequence of move instructions is generated for command line 21.

MoveOutBf2InpBf$((3,0),(1,1))$
MoveOutBf2InpBf$((3,0),(2,1))$
MoveOutBf2InpBf$((3,0),(2,0))$
MoveOpc2PU$((\text{AddN},1),2)$
MoveOutBf2InpBf$((2,0),(1,0))$
MoveOpc2PU$((\text{false},0),1)$

2) Taking step 24, then inside PU5 there is the following buffer state

i	1
j	4

Firstly the unused data values of j are discarded by 4 times MoveOutBf2InpBf$((5,0),(-1,-1))$. Then the value i is moved to PU5 by MoveOutBf2InpBf$((5,0),(5,0))$.

Then the constant 1 is moved to PU5 by MoveConst2InpBf$(1,(5,1))$. Then for computing 3 copies of $i+1$ on PU5 the opcode $(AddN,3)$ is moved to PU5 by MoveOpc2PU$((\text{AddN},3),5)$. Thus all together the following sequence of move instructions is generated for command line 24.

MoveOutBf2InpBf$((5,0),(-1,-1))$
MoveOutBf2InpBf$((5,0),(-1,-1))$
MoveOutBf2InpBf$((5,0),(-1,-1))$
MoveOutBf2InpBf$((5,0),(-1,-1))$
MoveOutBf2InpBf$((5,0),(5,0))$
MoveConst2InpBf$(1,(5,1))$
MoveOpc2PU$((\text{AddN},3),5)$

4 Conclusions

Exposed datapath processors are a scalable alternative to the conventional processors suffering from register file scalability bottleneck. These processors expose their internal datapath to the compiler. It is the responsibility of the compiler to move values directly between the PUs utilizing the exposed datapath. SCAD is a relatively new exposed datapath processor that employs FIFO buffers at the PU inputs and outputs. While a FIFO buffer can hold more values than a register file, its access restrictions make code generation for SCAD particularly challenging. The SCAD compiler cannot always ensure that the relevant values for direct transportation from the producer PUs to the consumer PUs are available at the heads of producer PU output buffers. In this case, the SCAD compiler stores some values in the main memory to directly communicate the remaining values between PUs in the SCAD machine. This thesis presented an alternative code generator that rotated values through the output and input buffers in SCAD so as to move the relevant values to output buffer heads for transportation. The values to rotate are carefully chosen to minimize the overhead on the interconnection network and the PUs. This way, expensive memory accesses are saved during program execution by better utilizing the exposed datapath.

4.1 Future Work

The new code generation technique is explained by its application to an example bubble sorting program. It is essential to carry out experiments using other benchmarks as well in order to quantify the performance gain from memory access savings. It is necessary to investigate the complexity of the optimal code generation problem where optimal refers to minimum overhead (rotation of values). The results from the presented compiler heuristic should be compared to the optimal result to judge its efficacy. From a hardware perspective, it is necessary to understand the differences in execution time and power consumption when data is rotated through the buffers in a SCAD processor and when data is stored in the main memory. This information can be used to improve the compilation techniques further.

Bibliography

[Bha20] A. Bhagyanath. “Code Generation for Synchronous Control Asynchronous Dataflow Architectures”. PhD. PhD thesis. Department of Computer Science, University of Kaiserslautern, Germany, Jan. 2020.

[BJS15] A. Bhagyanath, T. Jain, and K. Schneider. “Poster Abstract: A Time-Predictable Model of Computation”. In: *Real-Time Systems Symposium*. San Antonio, Texas, USA: IEEE Computer Society, 2015, p. 376.

[BJS16] A. Bhagyanath, T. Jain, and K. Schneider. “Towards Code Generation for the Synchronous Control Asynchronous Dataflow (SCAD) Architectures”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. Freiburg, Germany: University of Freiburg, 2016, pp. 77–88.

[BS16] A. Bhagyanath and K. Schneider. “Optimal Compilation for Exposed Datapath Architectures with Buffered Processing Units by SAT Solvers”. In: *Formal Methods and Models for Codesign*. Kanpur, India: IEEE Computer Society, 2016.

[BS17a] A. Bhagyanath and K. Schneider. “Exploring the Instruction-Level Parallelism Potential of Exposed Datapath Architectures with Buffered Processing Units”. In: *Application of Concurrency to System Design*. Ed. by A. Legay and K. Schneider. Zaragoza, Spain: IEEE Computer Society, 2017.

[BS17b] A. Bhagyanath and K. Schneider. “Exploring Different Execution Paradigms in Exposed Datapath Architectures with Buffered Processing Units”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Ed. by Y. Patt and S.K. Nandy. Samos, Greece: IEEE Computer Society, 2017, pp. 1–10.

[Rix+00] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. “Register organization for media processing”. In: *High-Performance Computer Architecture (HPCA)*. 2000, pp. 375–386.

[ZK98] V. Zyuban and P. Kogge. “The energy complexity of register files”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. New York, NY, USA: ACM, 1998, pp. 305–310.