Parallelization of while-loops in nested loop programs for real-time multiprocessor systems

Geuns, S.J.

Award date:
2010

Master's Thesis

Parallelization of While-Loops in Nested Loop Programs for Real-time Multiprocessor Systems

Stefan J. Geuns

August 2010

Eindhoven University of Technology
NXP Semiconductors

Supervisors:
Prof. dr. Henk Corporaal (Eindhoven University of Technology)
Prof. dr. ir. Marco Bekooij (NXP Semiconductors)
ir. Tjerk Bijlsma (NXP Semiconductors)

Abstract

Many applications with stream processing behavior contain one or more loops with an unknown number of iterations. These loops have to be parallelized in order to utilize the maximum capacity of an embedded multiprocessor platform and thus increase the total throughput. This thesis presents a method to automatically extract a parallel task graph based on function level parallelism from a sequential nested loop program with while-loops. The task graph does not rely on a fixed order of the execution of the tasks.

The notion of a single assignment section is introduced, which is used in the parallelization approach. A single assignment section is less restrictive when programming the sequential nested loop programs than when demanding single assignment code. Instead of the complete program, only in a single assignment section single assignment must hold. This makes the parallelization of all while-loops possible.

Communication between the extracted tasks is done via circular buffers. In a circular buffer reading and writing is only allowed inside a window. For each circular buffer a choice is made between a buffer with sliding or overlapping windows depending on the costs. In a circular buffer with sliding windows the read and write windows can not overlap while this is allowed in a circular buffer with overlapping windows. An analysis on the overhead of the circular buffer is performed and a comparison between circular buffers with sliding and overlapping windows is made. Synchronization for the used circular buffers is inserted into the task graph to ensure the same functional behavior as the sequential nested loop program.

Sufficient buffer capacities for a deadlock free execution of the parallel task graph are determined using data flow analysis. For each while-loop a cyclo-static data flow model is derived based on the models for circular buffers with sliding and overlapping windows. The model for overlapping windows is also presented in this thesis. The models for the while-loops are combined such that buffer capacities can be determined for the complete nested loop program. An extension to cyclo-static data flow models is proposed such that the combining of the while-loops can be modeled.

A DVB-T radio receiver where the user can switch channels after an undetermined amount of time illustrates the parallelization approach. By means of execution traces it is shown that a higher throughput can be achieved when the presented parallelization approach is used.

An extension to circular buffers with sliding or overlapping windows is presented

where windows can be disabled temporarily at run-time. This reduces the synchronization overhead when windows are not used for an extended period of time, which can be the case for while-loops.

Acknowledgments

I would like to thank my supervisors Marco Bekooij and Henk Corporaal for their valuable feedback I received during my masters project.
Marco was my supervisor at NXP Semiconductors. He helped me a lot by having regular discussions about many things related to my graduation subject. I would also like to thank him for the extended feedback I received on my paper and this thesis and for the arrangements he made for our trip to the MAPS workshop. Marco also created the hardware platform on which I could do all my experiments.
Henk was my supervisor from the Eindhoven University of Technology. Henk gave a lot of helpful insights during our meetings. He also provided lots of feedback on many versions of my presentation and paper and also of this thesis.

Furthermore, I would like to thank Rudolf Mak for being part of the assessment committee and for showing an interest in my graduation project.

I would also like to thank Tjerk Bijlsma and Joost Hausmans for their input and the many discussions we had. Tjerk was also part of the assessment committee, for which I would also like to thank him.

Finally I would like to thank my colleagues from the Distributed Systems Architectures group at NXP Research for the nice time I had during my project at NXP. Especially the coffee breaks and lunch were a welcome break during a day of hard work.

Stefan Geuns

Table of Contents

List of Figures

1 Introduction

Currently there is a trend in the embedded systems world to move to multiprocessor platforms [18]. Multiprocessor platforms can provide more processing power for less energy than single processor platforms are able to provide. However, programming these multiprocessor platforms proofs to be challenging to do by hand. In the embedded systems domain, many of the applications process streams of data, such as video systems or radio receivers. Often these stream processing applications are developed as sequential code after which a time consuming and error prone manual parallelization step is needed to convert the sequential application into a parallel task graph that can be executed on multiple processors.

A common problem in hand-written parallel code is deadlock. Deadlock means that all tasks are waiting on each other and therefore no progress can be made anymore. A common cause for deadlock is that variables are acquired in a different order by different processes, creating the risk for deadlock if only one process is allowed to acquire a variable [14].

Automatic parallelization tools have been developed to ease this manual process of converting a sequential program to a parallel task graph. However, to the best of our knowledge, none of these tools can handle while-loops where iterations can overlap in combination with function level parallelism and are therefore less suitable for the above mentioned stream processing applications. A higher utilization of the multiprocessor platforms can be achieved when loop iterations can overlap and thus tasks can be pipelined much more.

Radios process infinite input streams until the user indicates that a switch to a different channel is needed. After switching channels, reinitialization has to be performed in order to process the new input stream. See for example the DVB-T radio receiver as shown in Figure 1.1 which demonstrates this behavior. Parallelizing this application presents a number of difficulties, such as user dependent loop conditions, an endless loop and multiple writers to a single variable. Similar behavior is encountered in other applications which show streaming behavior, such as a video decoder or a telephone modem. The looping behavior corresponds to a while-loop with a termination condition that depends on, for example, the user input.

In this thesis we introduce a while-loop for use in a nested loop program (NLP) where the number of iterations of the loop and even the loop termination condition can be determined during the execution of the loop. The NLP is parallelized automatically by our tool that extracts function level parallelism. Buffers are created from

```
def channel_t channel;
def data_t x, y, z;
def int state;

loop {
  channel = selectChannel();
  reset(out state);
  loop{
    x = readInput(channel);
    switch(state){
      case 0:{
        acquisition(x, out state');
      }
      case 1:{
        y = fft(x);
        z = equalization(y);
        demap(z);
        verifySync(x, out state');
      }
    }
  } while(!channelChangeRequest());
} while(1);
```

Figure 1.1: NLP of a simplified DVB-T radio receiver.

shared variables and synchronization is inserted to ensure that data is written before it is read and remains available until it is no longer used. Inserting synchronization causes additional cycles to be used by the synchronization statements, thus reducing the performance of the parallelized application. Therefore, synchronization statements should be used as little as possible in order to reduce this overhead as much as possible.

Function level parallelism is extracted from the NLP by giving each function its own task. By extracting parallel tasks from sequential functions a pipeline is created, enabling a higher possible throughput on a multiprocessor platform for applications which execute the same functions often. From the extracted tasks a task graph is created where the shared variables form the edges in the graph and the tasks the nodes. An example task graph is shown in Figure 1.2. This task graph is extracted from the NLP of the DVB-T radio receiver from Figure 1.1.

From the extracted task graph code is generated which can be executed on a multiprocessor platform. Each node in the task graph becomes a process in the generated code. Each edge in the task graph, and therefore also each shared variable in the NLP, becomes a circular buffer (CB) where either sliding windows [3] or overlapping windows [4] are used. The choice depends on the costs in terms of buffer size and whether sliding windows causes deadlock or not. If deadlock is detected, overlapping windows are used for the buffers that cause deadlock. It is shown that if all buffers use overlapping windows, deadlock is not introduced by the presented parallelization approach. The NLP has to satisfy single assignment in all single assignment sections such that dependencies between functions can be determined. The task graph resulting from the parallelization process can execute using any schedule as permitted by the inserted synchronization and the schedule can even change at any moment during the execution of the task graph.

In addition to this thesis a paper was written about the parallelization approach presented in this thesis. This paper can be found in Appendix A.

There are a number of existing techniques which attempt to parallelize while loops. Rauchwerger [15] is able to parallelize while loops which contain linked lists. In contrast to our approach, data level parallelism is used to execute multiple loop itera-

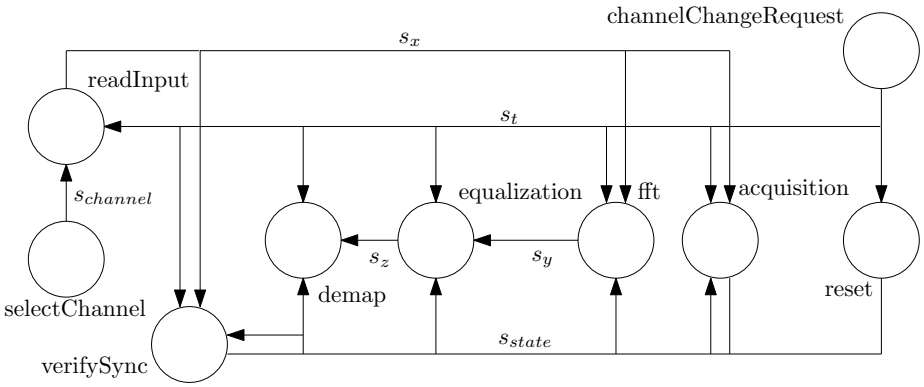


Figure 1.2: Task graph extracted from the NLP of the DVB-T radio receiver from Figure 1.1.

tions simultaneously. Our solution employs function parallelization which results in a pipelined execution of tasks. Collard [7] also uses data level parallelism and speculatively executes the next iteration independent of the loop condition. These parallelization methods are orthogonal to our method and can be combined to create a hybrid approach that exploits data as well as function level parallelism.

The approach presented by Turjan [17] transforms a NLP into a Kahn process network (KPN). The disadvantage of that approach is that it can only handle first-in first-out (FIFO) buffers. The consequence is that skipping locations, reading locations multiple times or reading locations in a different order than they are written requires a reordering buffer and a potentially complex controller or lookup table. Our buffers do not need such a controller task or lookup table.

Traditionally compilers apply software pipelining to schedule instructions in such a way that multiple iterations of loops can overlap [12, 16]. The compiler which performs instruction level software pipelining knows the execution times and finish times of all instructions and can therefore create a schedule that satisfies all dependency constraints. The difference with our method is that we do not assume a global clock nor do we create a fixed execution order and therefore we need to insert synchronization in order to satisfy all dependency constraints.

Douillet [9] performs software pipelining for multicore architectures. However, multiple producers where it is not known at compile time which producer writes the actual value are not supported by their synchronization statements.

1.1 Problem Statement

Existing approaches based on function level parallelism used for the automatic parallelization of sequential programs with while-loops can not handle the overlapping of loop iterations during execution. These while-loops can have arbitrary loop conditions, which include conditions that can only be determined during the execution of the loop.

Synchronization statements must be inserted in the generated code to guarantee that data is always written before it is read. Synchronization overhead must be kept as low as possible because it reduces the throughput of the parallel program. Deadlock may not be introduced by the inserted synchronization statements because this would stop

the execution of the complete program.

Real-time constraints are often present in applications that have streaming behavior. Therefore, throughput guarantees must be given in order to satisfy these constraints. When the throughput constraints are not stringent, buffer capacities can often be reduced in order to reduce costs.

1.2 Contribution

The main contribution of this thesis is an approach for the automatic parallelization of while-loops with an unknown number of iterations. This is an extension to work done by Bijlsma [3, 4]. From the statements of such while-loops, function level parallelism is extracted into a task graph.

Communication is done via CBs with either sliding windows or overlapping windows. For each CB a choice is made between these two types. Synchronization is inserted into the task graph in order to preserve the same functional behavior as the sequential nested loop program containing the while-loop. It is shown that the parallelization approach does not introduce deadlock if overlapping windows are used for all buffers.

The task graph is transformed into a data flow model on which sufficient buffer capacities are determined for known throughput constraints. An extension to cyclo-static data flow (CSDF) models is proposed to determine these buffer capacities.

An extension to the two types of CBs is proposed where windows can be temporarily disabled in order to reduce the synchronization overhead if a window is not used for a longer period of time. This extension can be used in the parallelization of the while-loop where initialization statements show this kind of behavior.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 discusses the target system in Section 2.1, the restrictions needed on the sequential input language in Section 2.2, the input language itself in Section 2.3 and task graphs in general in Section 2.4.

Chapter 3 discusses two different types of CBs, sliding in Section 3.1 and overlapping windows in Section 3.2. A CSDF model for CBs with overlapping windows is presented in Section 3.2.4. Section 3.3 makes a performance comparison between CBs with sliding and overlapping windows and Section 3.4 presents an algorithm to choose between the two types of buffers.

These CBs are used for the parallelization of while-loops, which is discussed in Chapter 4. Section 4.1 shows how synchronization statements can be inserted. Section 4.2 gives the essence of a proof for the deadlock freedom of the inserted synchronization. Section 4.3 shows that existing data flow models are not sufficiently expressive and Section 4.4 presents a modular cyclo-static data flow (MCSDF) model which is used to determine sufficient buffer capacities for a certain throughput.

Chapter 5 present an extension to CBs where synchronization for while-loops can be done more efficiently. A case-study containing a radio application with a while-loop is discussed in Chapter 6. Chapter 7 presents the conclusions.

Appendix A contains the paper written by me. This paper addresses the parallelization approach and focuses on the synchronization behavior of the parallel task graph.

2 Environment

This chapter discusses the environment in which the parallelization of sequential nested loop programs (NLPs) with while-loops is performed. Section 2.1 presents an overview of the platform on which the parallelized task graph will run. Section 2.3 explains the sequential language from which the task graph is extracted.

2.1 Target Platform

The target platform is an embedded many core multiprocessor system with shared memory. To the platform peripherals can be connected, such as a touch-screen display. An example platform is given in Figure 2.1. Each of the cores runs a micro kernel which can switch tasks and perform basic memory management. Task switching is required if there are more tasks than there are cores. To guarantee real-time behavior all schedulers must be able to give guarantees about throughput and latency. A time-division multiple access (TDMA) scheduler can give these guarantees and is therefore used in the task scheduler. Also the memory controller uses TDMA to schedule request coming from the different cores.

The multiprocessor compiler can generate code for a number of platforms. Already existing was an output to a two core ARM platform and a simulation environment based on SystemC [11]. Support is added for a platform using Microblaze [24] processors

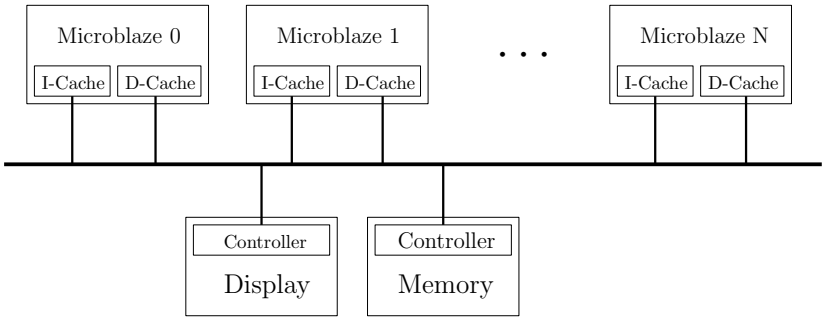


Figure 2.1: Shared memory multiprocessor platform using Microblaze cores connected by a bus.

```
def int x[10];
def int i;

while(1){
  for(i:0:9){
    x[i] = f();
  }
}
```

(a) Program that satisfies SSA, but not DSA.

```
def int y[10];
def int i;

for(i:0:9){
  switch(f()){
    case 0:{
      y[i] = g();
    }
    case 1:{
      y[i] = h();
    }
  }
}
```

(b) Program that satisfies DSA, but not SSA.

Figure 2.2: Example programs that either satisfy SSA or DSA, but not both.

and support for desktop computers based on pthreads [6] is also added.

The Microblaze based platform is used for the experiments from Section 3.3. The platform is implemented on a field-programmable gate array (FPGA) from Xilinx [23]. The FPGA contains 32 Microblaze cores, connected to a central DDR3 memory. Also a display is connected such that applications can show graphical output to the user.

2.2 Single Assignment

A program that is parallelized must be in single assignment form. Single assignment means that a scalar or a element in an array is written only once. If this would not be the case, a task that reads from a circular buffer (CB) does not know when the relevant writer has written its value. For example the variable $state$ from Figure 1.1 is written by three different tasks. Therefore, during execution it is not known which task writes a value to $state$ in which iteration of the surrounding while-loop and when the value is no longer used and can be destroyed.

Two different forms of single assignment exists, static single assignment (SSA) and dynamic single assignment (DSA). SSA [1] means that there is only one assignment statement in the code that writes to any scalar or array. DSA [19] requires that a scalar or a element in an array is written only once during the execution of the program. A program can be in SSA form, while it is not in DSA form and vice versa. Figure 2.2 illustrates the difference between the two forms of single assignment. Figure 2.2a shows a program which is in SSA form because there is only one statement writing to x. It is however not in DSA form as all elements of the array x are written infinitely often due to the endless loop. Figure 2.2b shows a program that is in DSA form because every element of the array y is written only once during the execution. However, this example is not in SSA form because there are two statements ($f()$ and $g()$) that potentially write to an element in y.

Unfortunately, both notions of single assignment are not sufficiently expressive for the while loop with an unknown iteration upperbound. SSA can not handle multiple assignments to a single variable, therefore if-statements with multiple branches writing to the same variable are problematic. DSA can not guarantee a finite array size because each loop iteration requires array elements which are not yet written. Because the loop can potentially execute an infinite amount of times, this would require an infinite

```
int x;
x = 3;
do{
  int x, y;
  x = 5;
  y = 2;
  print(y);
} while(...);
print(x);
```

(a) Legal C program

```
def int x;
x = 3;
loop{
  def int x, y;
  x = 5;
  y = 2;
  print(y);
} while(...);
print(x);
```

(b) Illegal NLP

Figure 2.3: Sequential program as C code on the left and as NLP on the right.

number of array elements and therefore also an infinite array size. For example in Figure 1.1 the variable $state$ is written by three different statements, therefore the code does not satisfy SSA. The variable x is written an infinite number of times and therefore this example is also not DSA. This code can also not be rewritten such that DSA holds because an array can not have an infinite number of elements.

We therefore introduce the notion of a *single assignment section (SAS)*. During the execution of a SAS each scalar and array element can be written only once. A SAS belongs to a scalar or a complete array. Therefore, at each point in the code multiple SASs may exist and a scalar or array can have multiple SASs. At the end of a SAS the value of the variable from the SAS is lost, giving a variable a *temporal scope*. Temporally scoped means that the value of a variable is not only bounded by the location in the program, but also by time.

The length of each SAS is defined at compile time by the semantics of the sequential input program. These input programs are explained in the next section. The first SAS of all variables always starts at the beginning of a NLP. For each scalar or array written before the end of the while loop, its corresponding SAS ends at the end of the loop. This SAS is executed as often as the number of iterations of the loop. A new SAS is started by the statements after the while loop, if any. From the second iteration onwards the SAS has a more limited scope from the start of the loop until the end of the loop as this part of the code contains the repeating statements. Note that the variables in the loop condition are part of the loop and therefore belong to the same SAS as if the variables were written inside the loop. If a scalar or array is not written before the end of the while loop, its SAS does not end at the loop but continues until the end of first loop where it is written in or until the end of the NLP, whichever comes first.

Consider for example the program in Figure 2.3 which shows a program with a loop. The program from Figure 2.3a is valid according to the semantics of C. However, it is illegal as an NLP containing SASs, see Figure 2.3b. When the program is executed as C the print of the variable x would give 3 because the scope of the x in the loop ends at the end of the loop. When the program is used as an NLP, two problems arise. The first problem is that the variable x is written two times in the same SAS, first with the value 3 and then with the value 5. The second problem is that the printing of x uses a value of x that is not yet written as the last write action to x was in another SAS.

A SAS is valid if its corresponding variable is in DSA form within the SAS and all locations are written before they are read. The complete program is valid if all SASs of all variables are valid.

Because the value of all variables is temporally scoped, the values are destroyed at

```
def private int i;
def int x[10];

for(i:0:9){
  if(b()){
    switch(f()){
      case 0:{
        x[i] = g();
      }
      case 1:{
        x[i] = h();
      }
    }
  }
  else{
    d(out x[0], out x[1], i);
  }
}

do{
  print(x[0]);
} while(a());
```

Figure 2.4: Example NLP showing the existing language constructs.

the end of a SAS. Because a SAS ends at the end of a loop iteration there is no way to pass values to the next loop iteration. To solve this problem we re-use the stream concept also found in Silage [20]. In Silage a variable is essentially a stream which can be shifted forward in time using the "@" operator. Also in synchronous guarded actions as proposed by Baudisch [2] this concept can be found in the form of the $next$ operator. This $next$ operator delays the write of a value by one macro step. In our case the macro step is the execution of a SAS. This concept of writing to the next execution of a SAS is annotated with a ' in our NLPs. A direct consequence is that this scalar or array element can not be written anymore when executing the next SAS as this would violate the DSA requirement.

2.3 Input Programs

As an input language NLPs are used. These NLPs encode sequential programs. In contrast to C, in NLPs it is always possible to determine if there is a dependency between two scalars or arrays because there is no aliasing. For example pointers are not supported in NLPs because they can point to any memory location without showing in the source code which variable allocated this space. The consequence is that by looking at the variable name it can be determined if two variables point to the same memory range. This property is required for the verification of single assignment.

The compiler for real-time multiprocessor systems already supports for-loops, if-statements, switch-statements and do-while-loops. A sample use-case of these control statements is shown in Figure 2.4. The corresponding task graph is given in Figure 2.5. The for-loops are loops where an iterator enumerates all values. There must always be an iteration upper bound known at compile time and during the execution of all iterations of the loop, DSA must be satisfied by all assignments.

The do-while-loop is a loop that is more flexible than the for-loops as there does not need to be an iterator. The loop condition can be any condition, but all iterations of the loop together still have to satisfy DSA. The consequence is that the number of

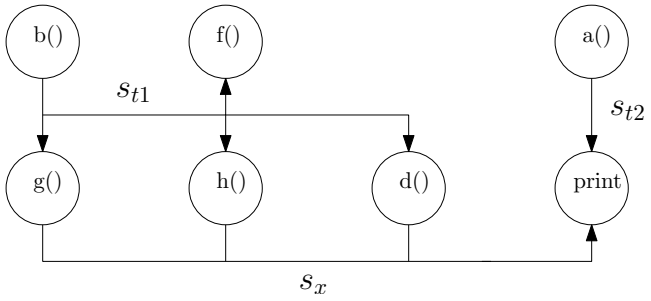


Figure 2.5: Task graph of the NLP from Figure 2.4.

iterations is bounded by the size of the arrays written to in the loop.

The loops introduced in this thesis are less restrictive than the above mentioned loops because DSA does not have to hold during the execution of the complete loop, but only in a SAS.

Already supported were private variables. These are variables that are not converted to CBs but are duplicated in each task. A typical usecase for these type of variables are loop iterators, such as used for example in the for-loop.

Also additional changes to the accepted input programs have been made. Previously, only functions were supported where there was at most a single output, but many input parameters. This restriction is removed and multiple outputs are now also possible. A function parameter has to be marked with "out" in order for it to be an output. For an example of how to use these output parameters, see the function d in Figure 2.4 which assigns a value to two elements in the array x.

2.4 Task Graph

From the sequential NLP a task graph is extracted. This task graph forms the base for the generated code and for the throughput analysis and buffer capacity calculations. A task graph describes which statements from the NLP exchange data with each other at some point in time. However, it does not contain any information when data is exchanged between tasks. This information is only present in the sequential NLP.

A task graph is a directed graph $H = (T, S, A, \sigma, \theta)$. The set of vertices is T, where each $t_i \in T$ represents a task. The set of hyperedges is S, where each $(V, W) \in S$, with $V, W \subseteq T$, represents a buffer. Each buffer s_i, with $s_i \in S$, corresponds with an array a_i, with $a_i \in A$. This array corresponds to a scalar or array declared in the NLP. The capacity of each buffer s_i is given by $\theta(s_i)$, with $\theta : S \rightarrow \mathbb{N}$. The size of each array a_i is given by $\sigma(a_i)$, with $\sigma : A \rightarrow \mathbb{N}$.

A hyperedge is an edge which can have multiple source and/or destination nodes. For example in the task graph from Figure 2.5 a hyperedge is present for the buffers s_{t1} and s_x. In the context of CBs a hyperedge means that at some point in time all source tasks t_i, with $t_i \in V$, write a location in the CB and at some point in time all destination tasks t_j, with $t_j \in W$ read a location from the CB. The consequence for buffers with overlapping windows is that each source and destination task have a window in the buffer and therefore need to move this window correctly. During the execution of a SAS there can be only one producer which writes a value to a location (DSA), even though there can be more producers acquiring a location. If a hyperedge

has multiple consumers, all of these tasks can read any location, as long as it is written first.

3 Circular Buffers

Applications with streaming behavior running on multiprocessor systems, such as a radio receiver, need buffers between tasks to arrange communication. These buffers need to be sufficiently large in order to avoid deadlock. A specific type of buffers is used where space is reused circularly, thus only a finite amount of space is needed. These circular buffers (CBs) also allow for pipelining and random access in the buffer. In these buffers read and write windows are added such that non-affine index expressions are supported without the need for complex control flow. There are two different types of windows, sliding windows and overlapping windows. Each of these types has advantages and disadvantages, therefore for each buffer the best type has to be selected.

3.1 Sliding Windows

CBs with sliding windows are introduced by Bijlsma [3] in order to support the unknown exact data dependencies often found in applications. These CBs contain windows for each reading and writing task. A task can have both a read and write window. In a read window locations can only be read, not written and vice versa for a write window. The tasks that read from a buffer are called consumers while the tasks that write to a buffer are called producers. Producer and consumers have random access to all locations inside their corresponding window. The read windows in the buffers can not overlap with the write windows. Since a value must be written before it can be read, the write windows must always be in front of the read windows in the buffer.

When writing to the CB it is not checked if a value was already written by another producer. It is assumed that the user of the CB does this correctly. In the parallelization approach this is the case as the statements for accessing the CBs are automatically generated.

The task graph that will be used in following example can be found in Figure 3.1. There is a CB s_x with two producers and two consumers. Each of these producers and consumers requires a window in the CB. An example of these windows in the CB when sliding windows are used can be found in Figure 3.2. All windows have a head and tail pointer. The head pointer is annotated with a cap above the pointer. The two write windows have an overlap of two locations. Read window r_0 spans between locations 6 and 7, without location 7. Read window r_1 is located between the locations 3 and 5. Write window w_0 spans between locations 11 and 14 and write window w_1 between locations 9 and 13, so these write windows overlap each other.

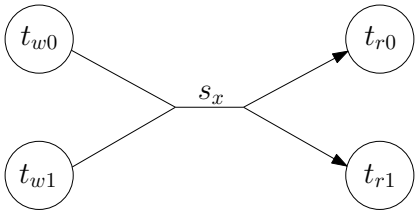


Figure 3.1: Task graph with two producers and two consumers

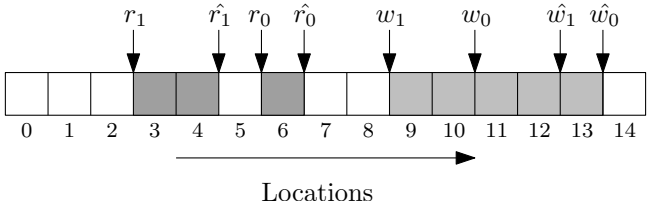


Figure 3.2: Circular buffer with sliding windows.

These CBs with sliding windows use similar synchronization operations as the release consistency memory model [13]. In this model a task has to request access to a location in the buffer by means of an acquire statement. After the task is done with a location, is must be released before other tasks can use it. This is illustrated in Figure 3.3. A producer has to acquire empty locations ($acquireSpace$) before data can be written to them. After the locations are released ($releaseData$) the consumer can acquire them ($acquireData$), read them and release them as space again ($releaseSpace$). All functions have as parameters the buffer and window on which they operate. The windows are represented by a integer identifier which is unique for every window per buffer.

All operations on the CB do not need support for atomic operations because there is only a single writer to all variables. Therefore, no race conditions can occur.

Locations are acquired and released from the buffer consecutively. Therefore a window is always a consecutive number of locations in the buffer and no gaps can exist in a window. Because only consecutive locations can be added to a window, acquiring and releasing locations must potentially be done earlier or later respectively. However if there are cyclic dependencies in the dependency graph this is not always possible and deadlock can occur. This is shown in [4] and the example is repeated in Figure 3.4 for completeness.

The task graph is shown in Figure 3.4a and consists of only two tasks communicating via two buffers s_x and s_y. The implementation of these tasks is shown in Figure 3.4b. The helper function F returns a non-affine pattern, which is used to de-

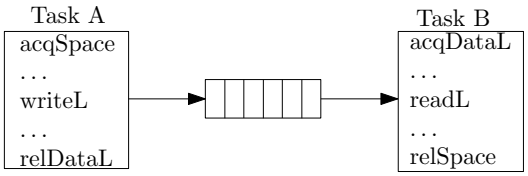
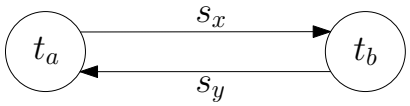


Figure 3.3: Operations that are available for a buffer.



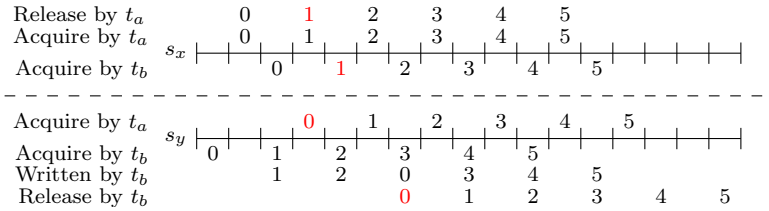
(a) Task graph.

```
int i;

sx[0] = 1;
sx[1] = 2;
for(i=2;i<=5;i++){
  sx[i] = sy[i-1];
}
```

```
int i;

for(i=0;i<=5;i++){
  sy[F(i)] = sx[i];
}
```

```
int F(int i){
  if (i < 2){
    return i+1;
  } else if ( i == 2){
    return 0;
  } else {
    return i;
  }
}
```

(b) Implementation in C of the tasks t_a and t_b and the helper function F.



(c) Operations as scheduled on the target platform. The offending operations are in red.

Figure 3.4: Task graph that deadlocks when sliding windows are used.

termine the written locations in s_y by task t_b. When these tasks are scheduled the data dependencies force the schedule as shown in Figure 3.4c. However, as shown in red, location 0 in s_y is acquired before the data is released by t_b. This is however not allowed by the acquire statement and it will block. But before location 0 is released, location 1 in s_x must acquired by t_b, which is not possible because this location is not released yet by t_a. Both tasks wait on each other now and the task graph deadlocks.

3.2 Overlapping Windows

A generalization of CBs with sliding windows are CBs with overlapping windows [4]. In this type of buffers read and write windows can overlap each other. Locations that are removed from the write window and locations that are added to the read window do not have to be consecutive, therefore preventing deadlock. For the proof of the deadlock freedom of overlapping windows, see Section 4.2.

Since locations not at the end of a write window can be released immediately, the tail pointer can no longer indicate which locations are in the write window and which locations are not. Therefore, full-bits are used to indicate which location is already written. Each write window has its own set of full-bits such that no atomic operations are required.

Also for CBs with overlapping windows it is not checked if single assignment holds, thus if another producer has already written a value to a location.

An example of a CB with overlapping windows can be found in Figure 3.5. This example is extracted from the task graph in Figure 3.1, which was also used for the

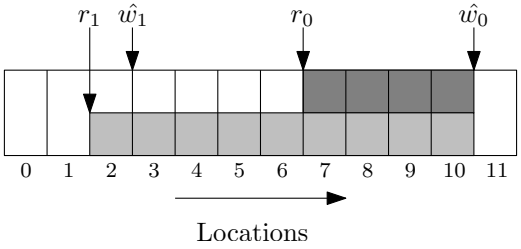


Figure 3.5: Circular buffer with multiple overlapping windows.

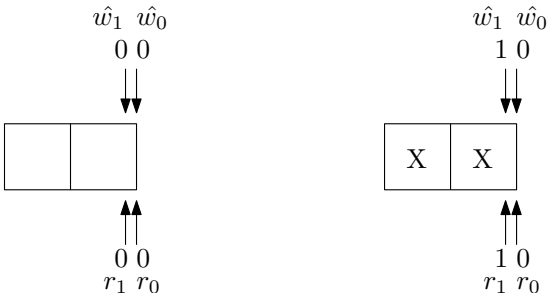
sliding windows example. The CB contains two read windows and two write windows. As is shown, no head of the read window and no tail of the write windows are present. These are no longer needed due to the full-bits. All read windows span from the tail of the read window to the head of the write window that is the furthest in the buffer. This is illustrated by the windows shown in gray in the figure. The read window spans from the head of the producer advanced the furthest in the buffer to the tail of the read window. The write window spans from its head pointer to the tail pointer of the read window furthest back in the CB. The write window is not a window with consecutive locations as it can contain gaps if locations in the middle of the window are released. For read window r_0 this means it spans from locations 7 to 11, without location 11 itself. Read window r_1 spans between locations 2 and 11. Write window w_0 spans the same locations as r_1, from 2 to 11. Write window w_1 spans between locations 2 and 3.

Also CBs with overlapping windows employ release consistency as explained for CBs with sliding windows. Again no support for atomic operations is required for the same reasons as before. The operations for releasing data ($releaseDataL$) and acquiring data ($acquireDataL$) take as an additional parameter the location in the CB which they acquire or release.

3.2.1 Wrapping

Wrapping in a CB is the moment that the head or tail pointer of a window in the CB wraps from the last index to the first. This means that the equations guarding the acquire and release statements must be reversed to reflect this situation. The difficulty is that not all windows wrap around at the same moment so each window must have some way of showing how many wrappings there have been. Because the CBs reflect applications that can run forever, a simple counter is not sufficient because it can overflow.

For sliding windows a single bit is enough to indicate in which round the pointers are because read windows can not overlap with write windows and all write windows are always in front of all read windows. The wrapping bit is incremented by one modulo two after each wrapping event. However for overlapping windows this is no longer sufficient due to the overlapping of the windows. See for example Figure 3.6 for an example where it is shown that a single bit is no longer sufficient. On the left side in the figure is the initial situation where all windows have a size of 0 locations. Producer 0, of which the head is indicated by w_0, executes two acquire, write and release calls. This means w_0 is again at the same position as where it started, but with the wrapping bit set to 1. This is indicated with the number above the arrow. Consumer 0, of which the tail is indicated by r_0, also does two acquire and release calls. This gives



(a) All head and tail pointers at the same position.

(b) Read and write window number 1 moved one round through the buffer.

Figure 3.6: Circular buffer with wrapping bits indicated for each window.

the situation on the right in the figure. In this situation it can no longer be determined which producer is in front. The same holds for the consumers. The consequence is that all windows wait for another window, thus creating deadlock.

Because a single wrapping bit is not sufficiently powerful to detect which producer or consumer is in which round, more values have to be used. We show here that three values are sufficient. However, four values are more efficient as the modulo operations used can than be replaced by bitwise operations. A producer can never acquire a location past the tail of all consumers which are a round behind and a consumer can never acquire a location past the head of the producer which is the furthest in the buffer. This means that the producers and consumers can never be more than one iteration apart from each other so only two wrap values are in use. The largest difference occurs when a producer is before all consumers and another producer is behind all consumers. Due to the wrapping, these tasks can all have different wrapping values. A larger difference can not occur as blocking would have occurred.

Because the wrapping counter indicates the iteration in which the pointer it belongs to is in, it should be updated atomically with the head or tail pointer of the corresponding window. However, it is assumed that the target system does not have atomic operations. To solve this problem the wrapping counter is stored in the same 32 bit word as the pointer. A small disadvantage is that the CB can never have more than 2^{30} locations as two bits are now reserved for the wrapping counter.

3.2.2 Synchronization Statements

In order to perform the synchronization, two synchronization function calls are required for both the consumers and producers, see Figure 3.3. The acquire statements are explained below. Both the statements for the releasing of space and data are non-blocking. The statement for releasing space advances the tail pointer of the corresponding read window by one. The statement for releasing data sets the full-bit of the corresponding write window. These release statements are not discussed any further. Also rules for writing a value to the buffer have to be defined, these are also explained below.

The assumption made for the sections below is that an acquire statement always precedes a release statement. Because these statements are automatically inserted, this should be correct by construction.

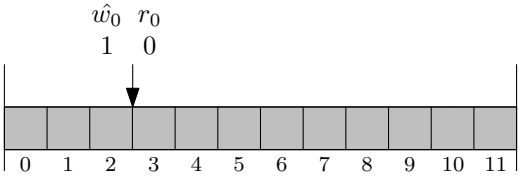


Figure 3.7: The write window must wait for the read window to move.

Acquire Statement for the Producer

A producer must acquire empty locations in the buffer in order to add these locations to its window. This acquire statement must satisfy the constraints given in Equations 3.1 and 3.2. The first constraint specifies that a producer can not overtake a consumer which is an iteration in the buffer behind itself. This ensures that a producer does not overwrite a value that the consumer has not read yet. In the equation b_p is the wrapping bit of producer p and b_c is the wrapping bit of consumer c. This situation is illustrated by Figure 3.7 where the producer must wait for the consumer to move. The wrapping counters are indicated above the arrows. Because the producer has a wrapping counter of one more than the consumer (respectively 1 and 0) and both pointers point to location 3, the read window spans the complete CB.

$$\exists_{c \in Consumers}(b_p = (b_c + 1)\%3 \wedge \hat{w}_p = r_c) \tag{3.1}$$

A producer p also has to wait for another producer p' if the other producer is a round behind p. If not, p' can overwrite locations written by p because dynamic single assignment (DSA) is only guaranteed for one round in the CB.

$$\exists_{q \in Producers}(b_q = (b_p + 1)\%3 \wedge \hat{w}_p = \hat{w}_q) \tag{3.2}$$

Acquire Statement for the Consumer

A consumer must also acquire locations in a buffer before they can be added to the consumer's read window. Also this acquire statement has to satisfy constraints when acquiring a location. A consumer can never acquire a position for reading before a producer has written a value to that position. An example can be found in Figure 3.8 where the full-bits are indicated inside the buffer. The locations 4 and 7 in front of the tail r_0 can not be acquired as they are not yet written. The other locations between r_0 and $\hat{w_0}$ can be acquired.

Let c be a consumer and l the location that will be acquired. Then we define z as given in Equation 3.3.

$$z = \begin{cases} b_c & \text{if } l > r_c \\ (b_c + 1)\%3 & \text{if } l \leq r_c \end{cases} \tag{3.3}$$

This checks if the location is in front of the tail of the read window or behind it.

The condition to verify if a producer has written a value can now be defined as is done in Equation 3.4.

$$\exists_{p \in Producers}((l \leq \hat{w} \wedge b_p = z \wedge f_p(l)) \vee (l > \hat{w}_p \wedge b_p = (z+1)\%3 \wedge f_p(l))) \tag{3.4}$$

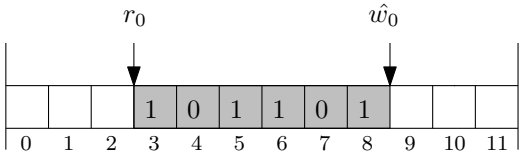


Figure 3.8: The consumer can not acquire all locations.

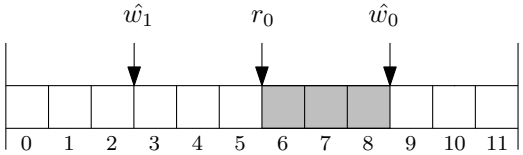


Figure 3.9: A producer is behind a consumer.

The function $f_p(l)$ returns the full-bit at location l for producer p. If there is no producer p which satisfies the constraint, the consumer c must block until there is such a p and the constraint is satisfied.

Writing Values

Writing values is of course only allowed after the written location is acquired first. A second requirement is that a value may only be written if there is a consumer that can potentially read it. Thus there should be a tail pointer of a read window behind the head of the producer's write window. The condition that reflects this is given in Equation 3.5. An example situation is given in Figure 3.9. In the example are two producers and one consumer. Producer 1 is behind the consumer and can therefore not write any values to the CB.

$$\exists_{c \in Consumers}((b_c = b_p \wedge r_c < l) \vee ((b_c + 1)\%3 = b_p \wedge l \leq r_c)) \tag{3.5}$$

3.2.3 Conditional Statements

Overlapping windows, as explained by Bijlsma, has a problem with conditional statements, such as if-statements and while-statements. If a conditional branch is never executed due to the condition being always false, no blocking occurs because no position is ever acquired. This problem is explained in the following sections and also a solution is proposed to solve the problem.

If-statement

It is not always the case that an acquire for the consumer precedes a release statement. For example if the read action, and therefore also the acquire, are done conditionally. This problem is illustrated by the code fragment from Figure 3.10. The release of space must always be done unconditionally because otherwise there could be a trace where there is no progress for this consumer as the tail pointer would not moved.

However, if the read statement is never executed because the condition preceding it is always false, the consumer will never wait for the producer and can update its

```
if(f()){
  acquireDataL(b,0,x);
  g(b);
}
releaseSpace(b,0);
```

Figure 3.10: NLP with an if-statement and synchronization inserted.

```
if(...){
  if(...){
    acquireDataL(b,0,x);
  }
  else{
    checkL(b,0,x);
  }
}
else{
  checkL(b,0,x);
}
```

Figure 3.11: Nested if-statements.

tail pointer unconditionally more than once and thus going past all producers. If this happens twice and the wrapping counter of the consumer is set to the one less than the wrapping counter of the producer, the producer will wait for the consumer while it is actually far behind the consumer. This can lead to deadlock if there is another buffer which is written by the same thread as the runaway consumer and if there is a thread which reads from the other buffer and also serves the waiting producer.

A possible solution is to let the release of the consumer block if the release is unsafe, so if the tail pointer would go past the head pointer of all producers. The disadvantage is that the current buffer capacity analysis of nested loop programs (NLPs) is invalidated by this change [22].

Another possibility is to always acquire a position. This can lead to deadlock if a location is never written, but it must be acquired for reading. Releasing a location from the write window must be done conditionally because else the full-bit will be set and the consumer does not know which producer actually wrote a value.

The last possibility is to add a new acquire statement for a consumer. This acquire statement only checks the pointers of the window but not the full-bits, solving the problem of never written and read locations explained above. It does make sure that blocking occurs in case the producer is slower because this acquire happens unconditionally. An optimization would be to do this new acquire statement only if the if condition is false because otherwise a real acquire would be executed, doing the same checks.

Note that this new acquire statement must be executed for all buffers which are conditionally read inside the if-statement and released outside of the if-statement. So also for reads in nested if-statements which are released outside of the outer if-statement. See for an example Figure 3.11 where a location in buffer s_b is acquired conditionally in a nested if-statement.

While-statement

The same problem can occur for a while-statement as all locations are released after the while-loop [4]. The releases of the consumer always need to occur after the while-loop as it is, or can be in general, unknown which locations are read in advance due to indices being dependent on input data. Because the while-statement only exists in the form of a do... while-statement, always at least one execution of the loop is performed. Each loop iteration a location is acquired, either using the normal acquire or the checking function introduced for the if-statement. This means blocking will occur for the next execution of the while-loop because the consumer is in front of the producer.

The only potential issue is that the consumer releases that many locations that it wraps to a place behind the producer and thus creating the scenario sketched for the if-statement where the producer waits for a consumer which is actually in front of the producer itself. However, this situation can not occur in the generated code from the multiprocessor compiler. The amount of positions that can be released is limited by the size of the array, which is declared in the NLP.

It can be the case that the used CB is smaller than the array declared in the NLP if, for example, the array was declared with a larger size than it is accessed. However, for a while-loop with an unknown access pattern this can never be the case because it is unknown in advance which locations are read or written. The consequence is that the capacity of the used CB for a while-loop must always be at least the size of the array from the NLP. Therefore the number of positions released after the while-loop is equal to the size of the array. This in turn means that a consumer which is ahead, can never have a wrap counter which is one less than the producer and it can therefore never block the producer. During the next execution of the while-loop the first executed acquire statement for the consumer will see that no producer is ahead of this consumer and it will block until a producer catches up again.

The equation for the acquire of the consumer as given in Equation 3.4 checks if there is a valid producer. This implicitly says that, if the consumer is in front of all producers, it must wait until a producer has written the requested value in the iteration of the consumer.

3.2.4 CSDF Model

This section explains how a cyclo-static data flow (CSDF) model of a CB with overlapping windows can be derived from a parallel task graph. With a CSDF model buffer capacities can be calculated for a given throughput, for example using the approach of Wiggers [22].

A CSDF model is defined as a graph $G = (V, E, \delta, \pi, \gamma, \phi)$, with E the set of directed edges and V the set of actors. An edge $e_{ij} = (v_i, v_j)$, with $e_{ij} \in E$, is directed from actor v_i to actor v_j with $v_i, v_j \in V$. Actors communicate tokens over edges. Each edge e_{ij} has δ_{ij} initial tokens, with $\delta : E \rightarrow \mathbb{N}$. Each actor v_i contains $\phi(v_i)$ phases, with $\phi : A \rightarrow \mathbb{N}$. An actor v_j is enabled for firing number c, with $0 \leq c$, when each of its input edges e_{ij} contains $\gamma(e_{ij}, ((c-1) \bmod \phi(v_j)) + 1)$ tokens and $\gamma : E \times \mathbb{N} \rightarrow \mathbb{N}$. These tokens are consumed when firing. After actor v_j finishes firing number c, $\pi(e_{ji}, ((c-1) \bmod \phi(j)) + 1)$ tokens, with $\pi : E \times \mathbb{N} \rightarrow \mathbb{N}$, are produced on output edge e_{ji}.

The phases for the edge between a producer and a consumer can be calculated by using the formulas presented below. The formulas are only valid for task graphs with a known access pattern. If the access pattern is unknown, the worst case must be assumed

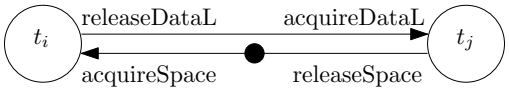


Figure 3.12: CSDF Model of an edge in a CB with overlapping windows.

where locations are always acquired before the first possible use and released after the last possible use.

The CSDF model for CBs with overlapping windows is based on the model for CBs with sliding windows [3]. In this approach, a lead-in is determined which specifies how many locations have to be acquired before the processing phase is entered. The lead-out defines how long a task must wait before it can start releasing locations. The processing phase of the execution contains the function from which the task was extracted. The processing phase consists of synchronization sections. A synchronization section starts at a group of acquire statements in a task and ends at the next group of acquire statements.

The lead-out for a producer in a CB with overlapping windows is always 0 as the producer can release its locations immediately. The lead-in for the consumer is also always 0 as the consumer only has to acquire the locations it needs.

A CB is modeled as two edges connected to the same two tasks, see Figure 3.12 for an example CSDF model. The bottom edge is called the back edge and contains the same number of tokens as the buffer capacity of the CB. For the back edge, the same formulas hold as given by Bijlsma [3]. This is because the statements for the back edge do not contain position information and are acquired and released consecutively, analogues to CBs with sliding windows.

For the forward edge there are two options to model the phases in a CSDF model. This is because a consumer does not need to acquire all or even any location from a particular producer. In a CSDF model all produced tokens must be consumed such that tokens do not heap up over time on an edge. Because the produced tokens that are never consumed do stay on the edge forever, they must either be consumed anyway or should never be produced. If tokens are always produced, then the consumer must acquire them in his lead-out. This can potentially lead to more phases than really needed. The second option is not to produce tokens that are never consumed. Using this scheme the consumer does not have to acquire tokens it is not going to use. A consequence is that if a consumer acquires no tokens from a producer, the phases on the edge contains all zeros. The consequence is that there is no constraint between this producer and consumer and the edge in the model can be removed. This also potentially reduces the number of cycles in the graph, potentially allowing for more pipelining and shrinking the buffer size. The last option is chosen because of the mentioned advantages.

Phases for the Releases of the Producer

Consider two tasks t_i and t_j connected by two oppositely directed edges, see Figure 3.12. The phases for task t_i on the edge e_{ij} are specified by the list of phases produced by $[0 \leq p < \phi(t_i) : \pi(e_{ij}, p)]$, where $\pi(e_{ij}, p)$ is defined as in Equation 3.6.

$$\pi(e_{ij}, p) = \begin{cases} 0 & \text{if } 0 < p \leq \hat{\rho}(t_i) \\ \#\Upsilon(t_i, t_j, p - \hat{\rho}(t_i)) & \text{if } \hat{\rho}(t_i) < p \leq \phi(t_i) - \check{\rho}(t_i) \\ 0 & \text{if } \phi(t_i) - \check{\rho}(t_i) < p \leq \phi(t_i) \end{cases} \tag{3.6}$$

The value given by $\hat{\rho}(t_i)$ represents the number of phases used by other edges connected to t_i, to release data to other CBs. The function $\phi(t_i)$ returns the number of phases of task t_i. The function $\check{\rho}(t_i)$ returns the number of phases used by other edges, which are connected to t_i, to acquire locations in their corresponding CBs. A more detailed and formal definition of $\hat{\rho}(t_i)$, $\check{\rho}(t_i)$ and $\phi(t_i)$ can be found in [3]. The function $\Upsilon(t_i, t_j, l)$ is defined as the set of locations released by the producer t_i in synchronization section number l and that are consumed by the consumer t_j at some point in time.

Phases for the Acquire of the Consumer

The phases for the consumer t_j on the edge e_{ij} are given by the sequence $[0 \leq p < \phi(t_j) : \gamma(e_{ij}, p)]$. The formula $\gamma(e_{ij}, p)$ is defined in Equation 3.7.

$$\gamma(e_{ij}, p) = \begin{cases} 0 & \text{if } 0 \leq p < \hat{\rho}(t_j) \\ \sum_{l \in \Upsilon(t_j, t_i, p - \hat{\rho}(t_j))} \eta(e_{ij}, l) & \text{if } \hat{\rho}(t_j) \leq p < \phi(t_j) - \check{\rho}(t_j) \\ 0 & \text{if } \phi(t_j) - \check{\rho}(t_j) \leq p < \phi(t_j) \end{cases} \tag{3.7}$$

The function $\eta(e_{ij}, l)$ returns the number of tokens that have to be acquired before the token representing the read location is released. It is defined in Equation 3.8.

$$\eta(e_{ij}, l) = \begin{cases} ((l - f(e_{ij}, l-1)) + (g(e_{ij}, l) - l + 1)) & \text{if } l < \sigma(a_h) \\ 0 & \text{if } l \geq \sigma(a_h) \end{cases} \tag{3.8}$$

Where the function $f(e_{ij}, l)$ returns the number of tokens that already have been acquired on edge e_{ij} for phase l.

$$f(e_{ij}, l) = \sum_{q=0}^{l} \eta(e_{ij}, q)$$

The function $g(e_{ij}, l)$ returns the index in the producer t_i its list of phases for the lth array access done by the consumer t_j.

$$g(e_{ij}, l) = \{q : 0 \leq q < \phi(t_i) - \hat{\rho}(t_i) - \check{\rho}(t_i) \wedge \alpha(t_j, a_h, l) \in \Upsilon(t_i, t_j, q)\}$$

The function $\alpha(t_i, a_h, l)$ returns the array index accessed in the lth access to a_h by the task t_i. Also for this function a more detailed explanation and definition can be found in [3].

The example NLP from Figure 3.13 shows an example where non-affine index expressions are used to read from and write data to a CB. Location 4 is never written nor read and location 5 is written but not read by this consumer. From the NLP two tasks are extracted, one producer t_f and one consumer t_g. Both tasks access the CB s_x. The functions $\hat{\rho}$ and $\check{\rho}$ return all zero for both tasks because there are no other tasks and buffers.

In each iteration of the for loop for task t_f only one location is written, therefore $\Upsilon(t_f, t_g)$ returns 1 for the locations 0, 1, 3, 2 and 6. The concatenated phases are therefore $< 1, 1, 1, 1, 1 >$ for producer t_f.

```
def int x[7];
for(i:0:5){
  x[F(i)] = f();
}
for(i:0:4){
  g(x[G(i)]);
}
```

```
int F(int l){
  if(l < 2) return l;
  else if(l == 3) return 2;
  else if(l == 2) return 3;
  else return l+1;
}
```

```
int G(int l){
  switch(l){
  case 0: return 0;
  case 1: return 3;
  case 2: return 1;
  case 3: return 2;
  case 4: return 6;
  }
}
```

Figure 3.13: Example NLP that uses a non-affine index expression for reading and writing data.

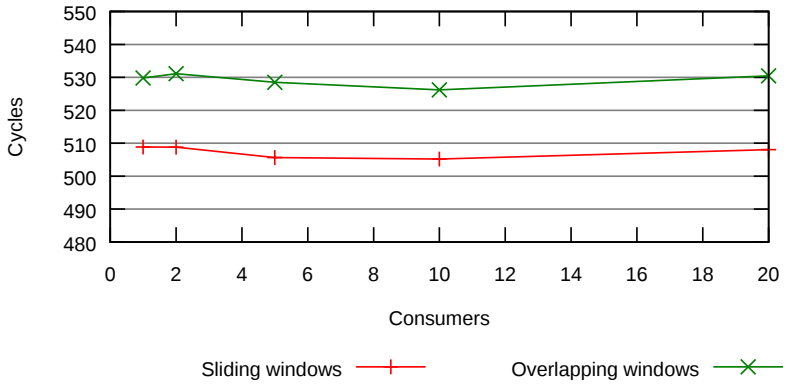


Figure 3.14: Execution times of the acquire statement of the consumer when a single producer and a variable number of consumers are present.

The consumer wants to read location 0, which is the first location released. The function $\eta(e_{fg}, 0)$ therefore returns 1. The next read location is 3, but this location is released 2 phases later. Therefore $\eta(e_{fg}, 1)$ returns 2. Location 1 is released before location 3 and is therefore already acquired so $\eta(e_{fg}, 2)$ returns 0. The last two phases both return 1 because location 5 is not produced.

3.3 Performance Comparison

In this section the performance of CBs with sliding and overlapping windows is compared for each acquire and release function. All tests are performed at the 32-core Microblaze system, of which only one core is used in the tests. The gcc compiler is used and compiler optimizations are set to level three and instruction and data caches enabled, except for the buffers itself which are not cached.

Figure 3.14 shows the execution time of acquiring data under worst case conditions without blocking. In the figure is a single producer versus a variable number of consumers. As can be seen the acquiring of data is more expensive for CBs with overlapping windows than for CBs with sliding windows. The extra time required for overlapping windows is caused by the checking of the full-bit. The acquiring of data is not dependent on the number of consumers because the read windows can overlap each other and consumers can not produce any data. This can also be seen in Equation 3.4.

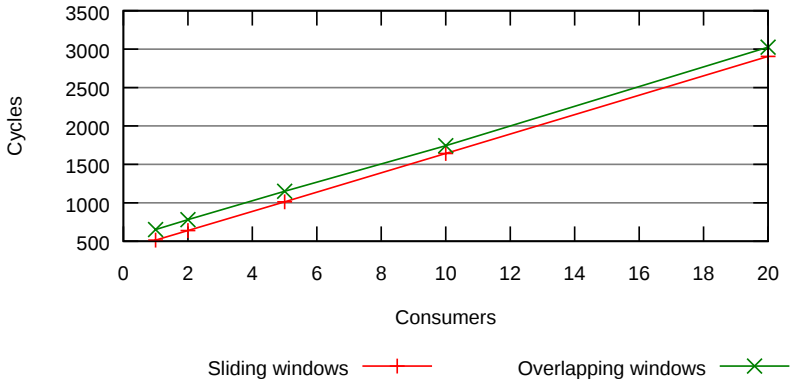


Figure 3.15: Execution times of the acquire statement of the producer when a single producer and a variable number of consumers are present.

Figure 3.15 shows the execution time of the acquire statement for empty locations. Again a single producer is considered versus a variable number of consumers. The window of the producer is moved first, data is written to the buffer and the locations are released. Afterwards all read windows are moved one by one. As can be seen in the figure, overlapping windows are again more expensive than sliding windows in terms of number of cycles. This is because for overlapping windows both the read windows and write windows have to be checked. Also the full-bit has to be set to zero for overlapping windows. The linear increase in execution time is caused by the extra consumers. Each check on a read window consists of checking the window's pointers, which is constant for all read windows.

Figures 3.16 and 3.17 show a execution time comparison of the two buffer types when a single consumer is used and a variable number of producers. As before, all producers move their write windows first and only then the consumers move their read windows. The last producer writes the value to the buffer and all other producers only acquire and release all locations.

The comparison shown in Figure 3.16 for acquiring empty locations again shows that overlapping windows are more expensive. As can be seen, the acquire statement has a constant number of cycles for sliding windows, but not for overlapping windows. The reason for this behavior can be found in Equation 3.2. This equation shows that the producer has to wait for all producers in the case of overlapping windows. This is not required for sliding windows as read windows can not overlap write windows and therefore write windows can never take over another write windows as there is always a read window in between.

Figure 3.17 shows the same trend as can be found in all comparisons. Overlapping windows are always more expensive than sliding windows. However, in this comparison, the execution time of overlapping windows is increasing much more rapidly than the execution time of sliding windows. The checking of the write window's pointers is present in both types of CBs, but for overlapping windows also the full-bits have to be checked. Note that these measurements are performed under worst-case conditions where a value is always produced by the last producer. Therefore, the consumer has to check all write windows. Under average-case circumstances this will not be the case in many programs and the acquire statement will cost less cycles as the acquire can abort

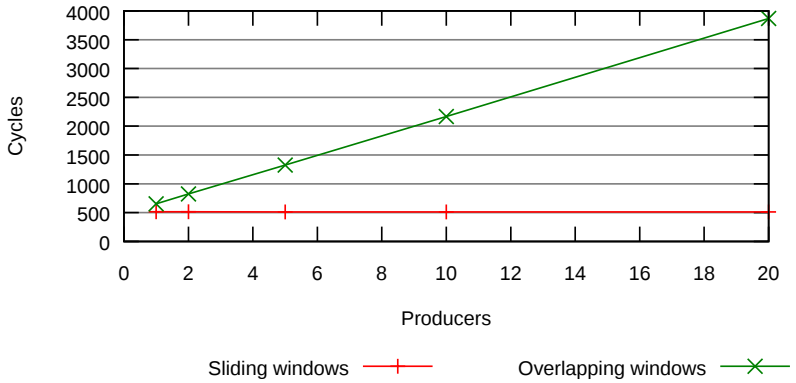


Figure 3.16: Execution times of the acquire statement of the producer when a single consumer and a variable number of producers are present.

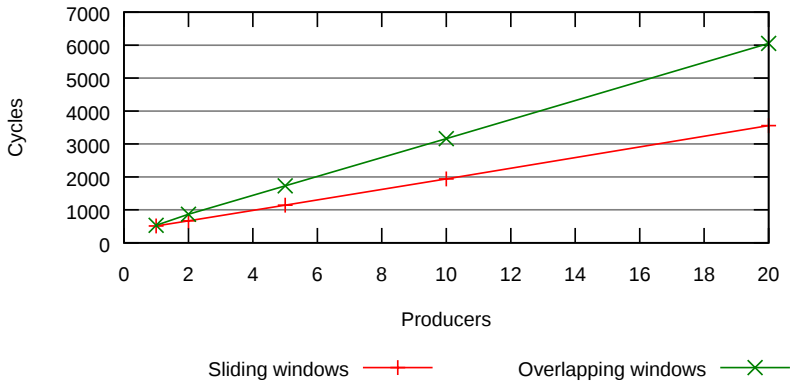


Figure 3.17: Execution times of the acquire statement of the consumer when a single consumer and a variable number of producers are present.

	Sliding windows	Overlapping windows
Cycles	235.0	234.84

Table 3.1: Release statement producer.

	Sliding windows	Overlapping windows
Cycles	221.31	196.51

Table 3.2: Release statement consumer.

its execution of a value is found at the requested location.

Table 3.1 shows a comparison of the release statement for both sliding and overlapping windows. The execution time of these statements is constant in the number of windows as they do not need to check the location of other windows. Also the difference between sliding and overlapping windows is very small. The implementation however is different as the release statement for sliding windows updates the tail pointer of the write window and the release statement for overlapping windows sets the full-bit.

Table 3.2 shows a comparison of the release statement for the consumer. The execution time for overlapping windows is lower, however the implementation for both statements is equal. Therefore, the compiler must have compiled both release statements differently. The task scheduler can not have an effect because no other tasks were executed during the measurements.

For all statements, except the release for the consumer, sliding windows are less expensive in terms of execution time than overlapping windows and are therefore preferred if the task graph allows so.

3.4 Buffer Type Selection

Since buffers with sliding windows are more efficient than buffers with overlapping windows, sliding windows should be used as much as possible. However, CBs with overlapping windows never introduce deadlock while sliding windows can. Also, CBs using overlapping windows can give smaller buffer capacities due to the earlier release of data to read windows. Because of the difference in buffer capacities resulting from the two buffer types, the choice of buffer type must be done simultaneously with the calculation of the buffer capacities.

Wiggers presents an efficient method to calculate sufficient buffer capacities for a single buffer type [22]. Using this method, linear bounds for array accesses are determined and from these bounds a linear program (LP) problem is defined, see Algorithm 1 from Wiggers. After solving this LP, buffer capacities can be calculated from the starting times of the tasks.

Since the objective is to determine the best buffer type simultaneously with the buffer capacities, the LP formula is extended to allow for the choice between two buffer types. Algorithm 1 shows the resulting formula for determining both the buffer type and capacities. The variable c_b represents the buffer type and $s(v, i)$ again represents the starting time of task v. The variable c_b is a binary choice between the two buffer types for buffer b. Because of this binary decision, the problem can no longer be given as an LP, but has to be defined as an integer linear program (ILP) problem.

As can be seen in Algorithm 1, two distinct β variables are present. The variable β^S represents the offset of the linear bounds for sliding windows, while β^O represents the offset for overlapping windows. The β^O for overlapping windows is determined using the model defined by the formulas given above in Section 3.2.4. The β^S for sliding windows is taken from the model as given in [3], where it is named β.

Algorithm 1 Buffer type selection using an ILP.

$\min \sum_{v_i \in V} s(v_i, 1)$
with
$s(v_i, 1) - s(v_j, 1) + c_b \cdot \beta^S_{(v_i,v_j)} + \neg c_b \cdot \beta^O_{(v_i,v_j)} \leq 0 \qquad \forall (v_i, v_j) \in E_0$
$s(v_0, 1) = 0$

To reduce the number of variables and constraints in the ILP, the first condition is rewritten to the condition shown in Equation 3.9. Note that all β parameters are determined before the ILP is solved.

$$s(v_i, 1) - s(v_j, 1) + c_b \cdot (\beta^O_{(v_i,v_j)} - \beta^S_{(v_i,v_j)}) \leq \beta^S_{(v_i,v_j)} \qquad \forall (v_i, v_j) \in E_0 \quad (3.9)$$

If the variable c_b contains a 0, the buffer must use sliding windows, otherwise the buffer must use overlapping windows.

4 Parallelization of While-Loops

This chapter presents our parallelization approach where nested loop programs (NLPs) containing while-loops with unknown upper bounds can be parallelized. After parallelization the statements in the while-loops can be pipelined and also iterations of the loop can overlap if the dependencies allow for this.

First synchronization statements are inserted into the parallel task graph. This is explained in Section 4.1. Based on the generated code a corresponding data flow model is created. Existing models are not sufficiently expressive, see Section 4.3. Therefore an extension to an existing model is proposed such that the while-loops can be modeled. This extension is discussed in Section 4.4.

The paper from Appendix A discusses only the code generation and not the data flow models.

4.1 Code Generation

The first step after extracting the task graph from the sequential NLP is to generate code for execution on the target platform. For the generated code to produce correct results, synchronization statements have to be inserted for all shared variables. There are three cases each in which synchronization statements are added at different locations in the program. In the sections 4.1.1, 4.1.2 and 4.1.3 these cases will be discussed.

4.1.1 Read-Only Variables

The simplest case to insert synchronization occurs when a variable is only written in the statements before a loop. Consider the example NLP in Figure 4.1 where the value of x is determined before the loop construct. After parallelization two buffers, s_x and s_y, are created and since there are three functions in the example, also three tasks are created. These tasks are shown in Figure 4.2.

In the example, the only writer of the variable x is the function f. This means that the single assignment section (SAS) for x starts at the beginning of the program from Figure 4.1 and ends at the end of the program as there are no other while loops. Inside the loop the variable y is written by the function g. Therefore a SAS for y is created from the beginning of the program to the end of the while loop. From the second iteration of the loop and onwards the SAS for y is defined from the start of the

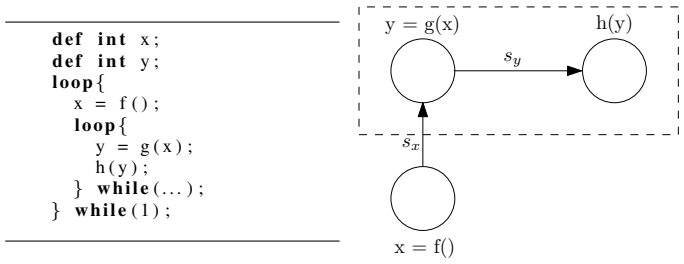


Figure 4.1: NLP with a variable that is only written in the statements before the while loop. The dots in the loop condition indicate that any condition is allowed. The task graph is shown next to the NLP. The contents of the dashed box are the tasks in the inner loop.

Task 1

```
do{
  acqSpace(x);
  x[0] = f();
  relDataL(x, 0);
} while(1);
```

Task 2

```
do{
  acqDataL(x, 0);
  do{
    acqSpace(y);
    y[0] = g(x[0]);
    relDataL(y, 0);
  } while(...);
  relSpace(x);
} while(1);
```

Task 3

```
do{
  do{
    acqDataL(y,0);
    h(y[0]);
    relSpace(y);
  } while(...);
} while(1);
```

Figure 4.2: Parallelized code for the NLP in Figure 4.1.

loop until the end of the loop. This program is valid as all SASs are in dynamic single assignment (DSA) form.

The synchronization statements for the buffers which are only written by the statements before the loop are inserted before and after any while loop in which they are read. In the example the synchronization statements for buffer s_x are inserted before and after the while loop. Synchronization statements for buffers written in the statements in the loop are inserted in the loop. In the example this is done for the buffer s_y.

It would also be possible to insert the synchronization for read-only variables in the loop. However, due to the temporal scope of the variables all values would be lost after the first iteration of the loop. A method to prevent this is to copy the old value to the next SAS execution using the assignment statement "x'$= x$;". This statement must be inserted in the loop. The disadvantages are that a new task is created for this statement and the synchronization overhead is significantly increased as every loop iteration synchronization must be performed for the copy action.

4.1.2 Shared Variable Before the Loop

It can also be the case that an array is written in the statements before the loop as well as in the statements inside the loop. The generated synchronization code will be inside the loop as the SAS is defined from before the loop until the end of the loop. For all tasks extracted from functions in the statements before the while loop,

```
def int x;
loop{
  x = f();
  loop{
    x' = g();
  } while(x < 10);
} while(1);
```

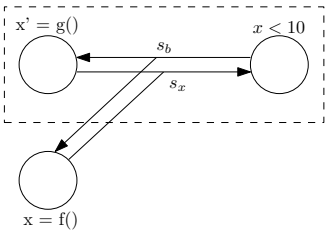


Figure 4.3: NLP and task graph with a variable that is written in the statements before the while loop and also in the while loop. The dashed box indicates the contents of the inner loop.

extra synchronization statements have to be inserted, if buffers are used which are also written in the statements in the while loop. This extra synchronization is required by the used circular buffers (CBs). The CBs require that all windows are no more than one iteration apart because a write window can never overtake a read window and a read window can never overtake the write window that is the furthest in its execution.

An example NLP which writes to a variable both in the statements before and in the loop is given in Figure 4.3. The variable x is written before the loop and also inside the loop, this forces the synchronization to be inside the loop. The SAS for x is defined from the start of the program up to and including the loop condition. The generated code for this example can be found in Figure 4.4. Task 1 from Figure 4.4 shows that the space acquired before the loop is conditionally acquired for the next execution of the same SAS, depending on if the same SAS is actually executed again. When this task is created using this scheme and executed, the synchronization is done equally often as the synchronization of all other tasks.

Figure 4.4 also shows that a new task is created when a variable or function is used in the loop condition. In the figure a new buffer b is created to store the condition value.

Note that sometimes renaming the variables written to before the loop can decrease the number of tasks in which an extra loop must be added. For example if there are many producers before the loop for a buffer x this would result in many tasks which would all need the extra loop. However, if these tasks write to a new variable y which is not written in the loop, no extra loops must be added for y. Only a copy statement must be added to copy y to x to make the values available for the statements in the loop. This copy task will be the only task with the extra loop as this is the only task which writes to a shared variable.

4.1.3 Shared Variable After the Loop

The NLP from Figure 4.5 gives an example of a variable being used in the statements before, in and after the first while loop. Since the example has statements using x after the first while loop, a second SAS is created for x next to the SAS ending at the first while loop. This second SAS starts after the first while loop and ends at the end of the second while loop.

As before, all tasks have to synchronize an equal number of times during the execution of the program. Therefore also for the tasks that use a value after the loop, a loop containing synchronization statements must be added. The consequence for the example is that all tasks need both while loops.

It might be more intuitive to have the last SAS span beyond the while loop, thus

Task 1

```
int t, i;
do{
  acqSpace(x);
  x[0] = f();
  relDataL(x,0);
  do{
    acqDataL(b,0);
    t = b[0];
    relSpace(b);
    if(t){
      acqSpace(x);
      relDataL(x,0);
    }
  } while(t);
  acqSpace(x);
  relDataL(x,0);
} while(1);
```

Task 2

```
int t, i;
do{
  acqSpace(x);
  relDataL(x,0);
  do{
    acqSpace(x);
    x[0] = g();
    relDataL(x,0);
    acqDataL(b,0);
    t = b[0];
    relSpace(b);
  } while(t);
} while(1);
```

Task 3

```
int t, i;
do{
  do{
    acqDataL(x,0);
    t = (x[0] < 10);
    relSpace(x);
    acqSpace(b);
    b[0] = t;
    relDataL(b,0);
  } while(t);
  acqDataL(x,0);
  relSpace(x);
} while(1);
```

Figure 4.4: Parallelized code for the NLP in Figure 4.3.

```
def int x;
loop{
  x = f();
  print(x);
} while(...);
loop{
  x = g();
  print(x);
} while(...);
```

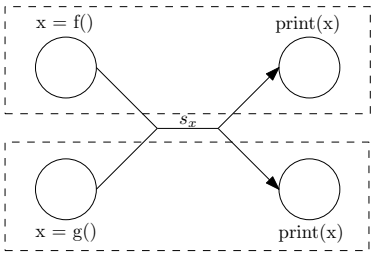


Figure 4.5: NLP and corresponding task graph with two successive while loops.

letting variable values also be available after the first loop. For instance by extending the temporal scope of a variable in the last iteration of the loop to include statements after the loop. However, this will cause problems as SASs can now overlap. Consider for example the NLP from Figure 4.5 where there are two while loops after each other. If the values from a SAS would be valid until after the loop, x will be written twice, once in the last iteration of the first loop and once in the first iteration of the second loop. The consequence is that the code is no longer DSA. Therefore a SAS runs up to the end of a loop.

4.2 Deadlock Freedom

In this section we give the essence of a prove that using CBs with overlapping windows never introduces deadlock if all buffer capacities are at least the array sizes and the SASs in the sequential NLP are all valid.

The parallel task graph can only deadlock due to a cyclic dependency caused by the insertion of acquire statements because these are the only blocking operations, see also Figure 3.3. By construction the parallelized task graph with inserted synchronization statements can always execute deadlock free using the sequential order defined in the NLP as a schedule, assuming the sequential NLP was valid. It is possible that when the

application executes, it deviates from this schedule. However, since the sequential application is deadlock free with the extra synchronization statements inserted using the same method as for the parallel program it must also be deadlock free when executed in parallel because the sequential order constraints are removed. Removing constraints can never introduce deadlock as no cycles can be formed in a graph by removing edges.

To show that the parallel task graph with synchronization statements inserted does not introduce deadlock, we have to consider the insertion of acquire statements in more detail. In CBs with overlapping windows there are two blocking acquire statements, one statement for data and one statement for space in the buffer.

First we consider the simple case where space is acquired in order to write data to the CB. Initially, no location is acquired in the buffer. By construction, each task can acquire a location in the buffer maximally once during the execution of a SAS. Therefore, at most the array size of locations in the buffer can be acquired by a task during a single execution of a SAS. Because we assumed the buffer capacity is at least the array size, the consequence is that during the first execution of any SAS all acquire calls for space can succeed. By construction all locations in the buffer are released before the end of the execution of a buffer's SAS, therefore after the execution of all first SASs all windows are again in their initial position and the process can start from the start again. Because all locations are released eventually during the execution of a SAS, an acquire for space from the second execution of an SAS will always eventually succeed.

Consider now the second case where data is acquired. Suppose the acquire and release statements are inserted in the sequential code using the following method. If an element in an array is written, the release of data for the corresponding location in the buffer is inserted in the NLP immediately after the element is written. The acquire statement for data is inserted in the NLP just before the location is read. In the NLP all acquire statements use an unique window. Because we assumed that in the sequential NLP every element is written before it is read, a release of data for location i is always performed before the acquire of space for location i. Note that read windows can overlap each other so a location can be acquired by multiple read windows during a single iteration. The parallelization process inserts these synchronization statements for data using the same method, so essentially a task is formed from every function and the corresponding inserted synchronization statements. Since in the parallelized task graph the only change with respect to the sequential NLP is the removal of the sequential order constraints, the acquire and release of data does not cause deadlock.

For a parallelized while loop the acquire statements for variables which are not written in the loop are moved to immediately before the loop compared to immediately before the statement. As all SASs were assumed to be valid in the sequential NLP, all written locations in the buffer are still released by a producer before they are acquired by a consumer. Therefore, the parallelization of the while loop does not introduce deadlock.

Since neither the acquire for space nor the acquire for data introduces deadlock, CBs with overlapping windows are deadlock free provided that the sequential NLP is valid.

4.3 Existing Models

This section shows that existing data flow models are not sufficiently expressive to model the while-loop with an unknown iteration upper bound.

```
def int x[2], i;
loop{
  for(i:0:1){
    x[i] = f();
  }
} while(g());
```

Figure 4.6: Example where VPDF is not sufficiently expressive.

4.3.1 VPDF Model

Variable-rate phased data flow (VPDF) [21] supports repetition of individual phases, where it is not known in advance how often a phase is repeated. This fits the definition of the while-loop and would eliminate the need for a new model.

However, the repeating factor in the while-loop is not a single phase but a sequence of phases. Consider for example Figure 4.6 statement $x[i] = f()$. This statement will give the phases $< 1, 1 >$ for acquiring space for x inside the while-loop and execution times e_i for $0 \leq i < 2$. The while-loop dictates that this sequence of phases is repeated n times, where n is determined by the unknown function $g()$. The phases for the complete task would be $< n\cdot < 1, 1 >>$. This can not be directly modeled in VPDF as VPDF only allows a single number to be repeated n times, so for example $< n \cdot 1 >$. A conservative solution would be to model the phases as an aggregate phase so the result for the example would be $< n \cdot 2 >$ and the execution time $\sum_{i=0}^{1} e_i$.

However, when while-loops are nested, this aggregation can no longer be applied for VPDF. For example when two while-loops are nested, a pattern in the phases like the following can arise: $< n\cdot < 0, m\cdot < 0, 1 >>>$. This can no longer be modeled conservatively in VPDF if n and m are not bounded.

4.3.2 CSDF Model

cyclo-static data flow (CSDF) [5] is a specialization of VPDF where repetition of individual phases is not allowed. Only the complete set of phases can be repeated. Therefore, CSDF is also not sufficiently expressive.

For the CBs with sliding windows a CSDF model exists that models the acquire and release statements. This paper also introduced a CSDF model for CBs with overlapping windows. Also there already exists a fast analysis that determines buffer capacities for these CBs [22]. Therefore, an extension to CSDF is introduced to allow while-loops to be expressed.

4.4 Modular CSDF Model

Since the while-loop models a potentially infinite loop, throughput can not be analyzed outside of the loop as nothing is known about the number of times the loop is executed. Therefore throughput is analyzed only for the statements inside the while-loop. For the statements outside of the while-loop, minimum buffer capacities for a deadlock free execution are determined.

In order to express the throughput constraints of while-loops, a modular cyclo-static data flow (MCSDF) model is introduced. A MCSDF model is an extension to a CSDF model where a single actor can consist of a complete CSDF model. A MCSDF model

is defined is a graph $G = (W, E, \delta, \pi, \gamma, \phi)$, with E the set of directed edges and $W = V \cup H$ the set of actors where V is the set of atomic actors and H the set of composite actors. Composite actors are MCSDF graphs themselves and can therefore be analyzed as such, thus H is the set of graphs G. Atomic actors are indivisible actors. An edge $e_{ij} = (w_i, w_j)$, with $e_{ij} \in E$, is directed from actor w_i to actor w_j with $w_i, w_j \in W$. Actors communicate tokens over edges. Each edge e_{ij} has $\delta(e_{ij})$ initial tokens, with $\delta : E \rightarrow \mathbb{N}$. Each actor w_i contains $\phi(w_i)$ phases, with $\phi : W \rightarrow \mathbb{N}$. An actor w_j is enabled for firing number c, with $0 \leq c$, when each of its input edges e_{ij} contains $\gamma(e_{ij}, ((c-1) \bmod \phi(v_j))+1)$ tokens and $\gamma : E \times \mathbb{N} \rightarrow \mathbb{N}$. These tokens are consumed when firing. After actor w_j finishes firing number c, $\pi(e_{ji}, ((c-1) \bmod \phi(j)) + 1)$ tokens, with $\pi : E \times \mathbb{N} \rightarrow \mathbb{N}$, are produced on output edge e_{ji}.

The input NLP is modeled as a MCSDF model where a while-loop is replaced by a composite actor. If there is another while-loop in a while-loop, this process is repeated recursively and each while-loop is therefore modeled as a separate MCSDF model. This is discussed in Section 4.4.1. The modular step is discussed in Section 4.4.2.

A while-loop in the MCSDF can be seen as a function where the function parameters are all read-only scalars and array elements. The output of this function are all variables that are written with the ' annotation, see Section 3 from Appendix A. Because the statements in the while-loop are modeled separately from the statements outside of the loop, the MCSDF model creates a barrier in the analysis. During the actual execution this barrier is not present, therefore the model is conservative with respect to the execution.

After an execution of a SAS, all values available in this SAS are lost. The consequence is that all producers and consumers in the CB corresponding to this SAS have their windows in the same position as before the execution of the SAS. Therefore, only one iteration of the while-loop has to be considered for the MCSDF model because creating a MCSDF model from two loop iterations results in the same MCSDF model.

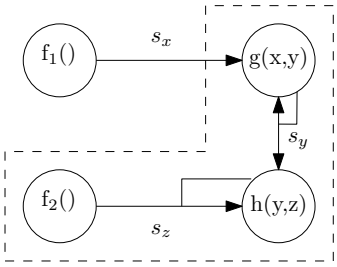
4.4.1 Model of the While-Loop

Because we want to derive a MCSDF model from a subset of tasks, also the task graph should consist of only this subset such that the translation from task graph to data flow model is preserved. We use a hierarchical task graph [10], which is defined as $H = (U, S, A, \alpha, \sigma, \theta)$. The set of tasks U in H contains all tasks t_i that are inside the while-loop for which this task graph is specified and not inside another while-loop, so $U = \{t_i | t_i \in T \wedge (\forall_{U' \in W} : U \neq U' : t_i \notin U')\} \cup U''$ where U'' is the set of all task graphs formed from loops inside H, T the set of all atomic tasks and W the set of all task graphs, or composite tasks, extracted from the NLP. The set of arrays A consists of all arrays which are written by the tasks from the set $U \backslash U''$. The array element for access number l from task t_i on array a_j is given by $\alpha(t_i, a_j, l)$, with $\alpha : U \times A \times \mathbb{N} \rightarrow \mathbb{N}$. The set of edges is S, with S containing all hyperedges (V, W), with $V \subseteq U$ and $W \subseteq U$. All edges from t_i, with $t_i \in T \wedge t_i \notin U$, to t_j, with $t_j \in U$ and all reversed edges are moved from t_j to the task formed from H. The size of the array a_k is given by $\sigma(a_k)$, with $\sigma : A \rightarrow \mathbb{N}$. The capacity of a buffer s_j is given by $\theta(s_j)$, with $\theta : S \rightarrow \mathbb{N}$. The capacity function θ only gives the capacity of the buffers as required by tasks in H, thus requirements from tasks outside of H are not considered.

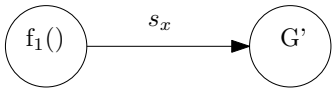
The tasks extracted from the statements that are inside the most deeply nested while-loop are considered for the first MCSDF model. Also the tasks that need extra synchronization loops due to shared variables with the while-loop must be modeled in this MCSDF model because they also need execution time to move the windows.

```
def int x[10], y[7], z;
for(i:2:7){
  x[i] = f1();
}
z = f2();
loop{
  y[0] = 0;
  for(i:1:6){
    y[i] = g(x[i+1], y[i-1]);
  }
  z' = h(y[6], z);
} while(...);
```

Figure 4.7: Example NLP.



(a) Flat task graph of the complete NLP. The hierarchy is indicated by a dashed box.



(b) Task graph of the highest level.

Figure 4.8: Task graphs extracted from the NLP in Figure 4.7.

If a variable is read-only in the while-loop the synchronization behavior specifies that the variable is acquired before the loop starts and released after the loop ends. The consequence is that, once the loop starts, this variable is always available when read and therefore this variable can never block any task that reads that variable in the loop once the loop is entered. Therefore all tasks outside the loop that use this variable are not required to be in U, assuming there is no other relevant dependency. Also all edges between these tasks and any task in the while-loop do not need to be considered for this model, but are considered in a higher level in the hierarchy.

The MCSDF model at the lowest level in the hierarchy is created the same as a CSDF model as explained in Section 3.2.4 for CBs with overlapping windows and by Bijlsma [3] for CBs with sliding windows. Also the buffer type selection approach as presented in Section 3.4 can be used.

The example NLP in Figure 4.7 and the corresponding hierarchical task graphs in Figure 4.8 illustrate the creation of a MCSDF model. Figure 4.8a shows the task graph of the complete NLP with the hierarchy indicated in the dashed box. In the NLP, the array x is not written in the while-loop and is therefore omitted from the task graph of the while-loop, together with the task formed from function $f_1()$, which writes to x. See the tasks in the dashed box from Figure 4.8a. The variable z is written both inside of the while-loop as well as outside the loop and is therefore included in the task graph, together with function $f_2()$, which writes to z.

4.4.2 Modular Step

To ensure deadlock free execution, buffer capacities have to be calculated for the complete program and not only the while-loop. Because the execution time of the while-loop is unknown, no throughput guarantees can be given for the complete model. Therefore, only sufficient buffer capacities are determined for deadlock free execution.

All tasks from the task graph are transformed into a MCSDF model using the same method as explained before for the while-loop. The edges between tasks outside of the while-loop and inside of the loop (variables which are read-only in the loop) are also transformed using the same method. From the MCSDF model of the loop a single composite actor is created, similarly to a task in the task graph. Since all buffers locations are acquired before the loop starts and released after the loop ends, this must be reflected it the model. A phase is added on each edge with as a token count the sum of the locations accessed. For sliding windows this must be a consecutive list of locations, so the sum also includes gaps of unused locations. The unused locations at the beginning of the CB can therefore be released immediately before the composite actor fires. Figure 4.9 shows an example to illustrate this. The buffer in the example is larger than the array size and not all array elements are read in the loop. The two array elements on the left are acquired and released before the loop starts. This is done in phase 0 of the composite actor formed from the loop. In phase 1 the locations required for the loop are acquired, this is illustrated with the gray box in the buffer. The last three locations are acquired and released after the loop is finished, in phase 2.

Equation 4.3 shows the number of tokens required for a composite actor G' for all phases on the edge $e_k = (w_j, G')$ between G' and an actor w_j connected to the composite actor. This equation holds for buffers with sliding windows. In the equations below, a buffer s_k is equal to (w_j, w_i). The array corresponding to the buffer s_k is a_k.

$$\chi((w_j, G')) = \max_{w_i \in U}(w_j, w_i) \in S' : (\max_{0 \leq q < \rho(w_i, a_k)} : \alpha(w_i, a_k, q)) \tag{4.1}$$

$$\xi((w_j, G')) = \min_{w_i \in U}(w_j, w_i) \in S' : (\min_{0 \leq q < \rho(w_i, w_k)} : \alpha(w_i, a_k, q)) \tag{4.2}$$

$$\gamma(e_k, p) = \begin{cases} \xi(e_k) & \text{if } p = 0 \\ \chi(e_k) - \xi(e_k) + 1 & \text{if } p = 1 \\ \sigma(a_k) - \chi(e_k) - 1 & \text{if } p = 2 \end{cases} \tag{4.3}$$

For overlapping windows $\gamma((w_j, G'), p)$ returns the number of distinct locations accessed.

The same formulas hold for the release of space by the composite actor. A single phase can be used to release all locations at once. Therefore, the number of tokens in this phase equals the number of locations acquired as data, thus $\pi(e_{ji}, p) = \gamma(e_{ij}, p)$.

Figure 4.8b shows the highest level task graph for the example NLP in Figure 4.7. The tasks from the while-loop are collapsed into the task G'. Since a task in G' reads from the buffer x, which is written outside of G', an edge which models x is added between $f_1()$ and G'. The corresponding MCSDF model for this task graph contains two actors, similar to the model shown in Figure 3.12. In the array x six locations are written. Therefore, $\gamma((f_1, G'), p)$ for $p = 0, 1, 2$ gives 2, $7 - 2 + 1 = 6$ and $10 - 7 - 1 = 2$ respectively. Therefore, also the release of space by the composite actor, defined by $\pi((G', f_1), p)$ gives the phases 2, 6 and 2 for $p = 0, 1, 2$.

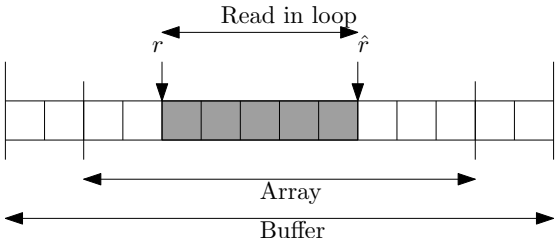


Figure 4.9: The window required by a loop does not have to be the complete array.

5 Circular Buffers with Dynamic Windows

As shown in Chapter 4 in the parallelization of nested loop programs (NLPs) with while-loops, extra while-loops need to be inserted into the tasks in order to move the windows in all buffers an equal amount of times. In order to overcome the need to insert these extra loops, an extension to circular buffers (CBs) with overlapping or sliding windows is proposed. Using this extension windows can be temporarily disabled such that they do not need to move forward until they are actively used again. Upon re-enabling the disabled windows are inserted again in the CB at the appropriate position.

5.1 Synchronization Statements

The synchronization statements from CBs with sliding or overlapping windows have to be modified for the buffers with dynamic windows. Each window w in the CB is extended with a status field. This status field indicates if w is enabled or disabled. When a window is disabled, it does not need to be considered when acquiring or releasing space or data. See the equations from Section 3.2.2 which show when the position of a window is checked.

Because the buffers are circular, buffer locations have to be reused and therefore re-enabling a window can not be done immediately. A consequence of using these circular buffer is that the correct wrap bit (for sliding windows) or wrap counter (for overlapping windows) has to be set. This bit or counter depends on the number of times the other windows wrap around, something which is not known in advance due to the unknown iteration count of the loop. Therefore, synchronization is inserted to the tasks that can decide when another window is allowed to be re-enabled. The same synchronization statements for acquiring and releasing space and data are used to decide when to enable or disable windows.

Also disabling a write window can not be done immediately as all read windows need to be able to check the full-bits of this write window to see if any values are written in the CB. A read window can be disabled at any time because it does not produce any values needed by other windows. The implementation for the enabling and disabling of windows is given in Figure 5.1 and 5.2 respectively.

Two additional CBs are created for each write window to synchronize on, while one additional CB is sufficient for each read window. All additional CBs need a buffer capacity of one location and can be of type sliding windows, which has the least amount

```
enableWindow(buffer b, window w){
  b.w.state = ENABLED;
  w.reset_full-bits();
  update pointers of w;
}
```

Figure 5.1: Activation pseudo-code implementation.

```
disableWindow(buffer b, window w){
  b.w.state = DISABLED
}
```

Figure 5.2: Deactivation pseudo-code implementation.

of overhead. When disabling a write window, $acquireData$ must first be called on the first extra buffer. When the acquire continues, the write window can be disabled and $releaseSpace$ can be called immediately after the window is disabled. The windows that are in the same buffer as the disabled window have to indicate that they no longer need the data produced by the disabled window. Therefore, the statements $acquireSpace$ and $releaseData$ are inserted after the windows are moved to the position where the disabled window is at the moment of its disabling.

Re-enabling is done using a similar method. The $acquireData$ statement is inserted at the point where the read or write window requests to become active again. After this statement continues, the window can be enabled again and $releaseSpace$ can be called. Again, the windows that use the same buffer as the disabled window must indicate, by means of a $acquireSpace$ and $releaseData$ call, that the disabled window can be re-enabled. These statements operate on the second buffer. The acquiring of space must be done before the buffer is used. The release of data must be done at the moment the window is allowed to be re-enabled.

Note that at all points in time, all buffers must have at least one producer and one consumer enabled. Also, these additional CBs do not need any memory space apart from the buffer administration as there is never a read or write done to these CBs.

5.2 Deadlock Freedom

The generated code for disabling a window does not introduce deadlock. This section gives an outline of the proof for this.

To show that deadlock is not introduced, it must be shown that all introduced acquire statements can always continue at some point in time. Because two acquire statements on two different buffers can not block each other, the two additional buffers can be considered separately. All tasks can initially acquire space for the first buffer as the capacity of the buffer is one and only one location is acquired. Since the generated code without these statements did not introduce deadlock, the acquired locations will also be released. Therefore, the task with the window that requests to be disabled can acquire this released location as data. The proof for the second buffer is analogues to the proof of the first buffer.

```
def int x;
loop{
  x = 0;
  loop{
    x' = x + 1;
  } while(h());
} while(1);
```

Figure 5.3: Example NLP where the number of iterations of the inner loop are counted.

Task 1

```
bool t;
int i;
do{
  acquireSpace(x,0);
  x = 0;
  releaseData(x,0);
  do{
    acquireData(c,0);
    t = c;
    releaseSpace(c,0);
    if(t){
      for(i=0;i<1;i++){
        acquireSpace(x,0);
        releaseData(x,0);
      }
    }
  } while(t);
  for(i=0;i<1;i++){
    acquireSpace(x,0);
    releaseData(x,0);
  }
} while(1);
```

Task 2

```
bool t;
do{
  acquireSpace(x,1);
  releaseData(x,1);
  do{
    acquireSpace(x,1);
    acquireData(x,1);
    x = x + 1;
    releaseData(x,1);
    releaseSpace(x,1);
    acquireData(c,1);
    t = c;
    releaseSpace(c,1);
  } while(t);
  acquireData(x,1);
  releaseSpace(x,1);
} while(1);
```

Task 3

```
bool t;
do{
  do{
    acquireSpace(c,2);
    t = h();
    c = t;
    releaseData(c,2);
  } while(t);
} while(1);
```

Figure 5.4: Parallelized code from Figure 5.3 using CBs with sliding windows.

5.3 Example Application

Figure 5.3 shows a small example NLP which counts the number of executions of the inner loop. Figure 5.4 shows the parallel task graph when using CBs with sliding windows. Note that if overlapping windows would have been used, the generated code would be almost identical for this example. Figure 5.5 shows the same example parallelized for use with dynamic sliding windows. The figure shows that the loop in Task 1 is replaced by the synchronization statements as discussed above. Because a write window is disabled, two CBs are added, s_1 and s_2. CB s_1 is used for disabling of write window and CB s_2 is used for the re-enabling of the read or write window.

Task 1

```
do{
  acquireSpace(x,0);
  x = 0;
  releaseData(x,0);
  acquireData(s1,0);
  disableWindow(x,0);
  releaseSpace(s1,0);
  acquireData(s2,0);
  enableWindow(x,0);
  releaseSpace(s2,0);
} while(1);
```

Task 2

```
bool t;
int init = 0;
do{
  acquireSpace(s2,1);
  acquireSpace(x,1);
  releaseData(x,1);
  do{
    acquireSpace(x,1);
    acquireData(x,2);
    x = x + 1;
    releaseData(x,1);
    releaseSpace(x,2);
    if(init < 1){
      acquireSpace(s1,1);
      releaseData(s1,1);
      init++;
    }
    acquireData(c,0);
    t = c;
    releaseSpace(c,0);
  } while(t);
  acquireData(x,2);
  releaseSpace(x,2);
  releaseData(s2,1);
} while(1);
```

Task 3

```
bool t;
do{
  do{
    acquireSpace(c,1);
    t = h();
    c = t;
    releaseData(c,1);
  } while(t);
} while(1);
```

Figure 5.5: Parallelized code from Figure 5.3 using the buffers with dynamic sliding windows.

6 Case-Study

This section illustrates our parallelization approach for while loops by means of a case-study. Figure 1.1 shows the structure of a, for explanatory reasons simplified, DVB-T receiver application which switches between two modes, detect and decode. After a potentially infinite amount of time, the user can switch between channels and the application will break the inner while loop and switch channels. This whole flow is repeated infinitely often.

This case-study does not use the improvements from Chapter 5 because they were not yet implemented due to time constraints.

The task graph for the program from Figure 1.1 can be found in Figure 1.2. As there are eight functions in the NLP, also eight tasks are created. Because the loop condition is a function which we want to execute only once in one iteration, the return value of the function must be distributed to all tasks in the loop. The buffer t is created for this purpose.

Because the $equalization$ and $demap$ functions are inside the switch statement, the tasks formed from these function also need read access to the variable $state$. An improved version of the parallelization process can remove this edge in the task graph as y and z already enforce this control flow. Note that the tasks that read from the CB $state$ can be pipelined, even though the loop condition is data dependent. Each of these tasks needs a different value of the variable $state$ if they are fully pipelined because they are in different iterations of the loop. Therefore, the CB $state$ needs a capacity of at least five locations.

The $state$ variable is a variable which is shared between the statements before the loop and in the loop. Therefore the task created from the assignment of 0 to $state$ must also contain the while loop. The synchronization statements are in the while loop for all tasks that use the $state$ variable. The $channel$ variable is a variable which is only written in the statements before the while loop. The synchronization statements for the $readInput$ task are therefore outside of the loop.

6.1 Experiments

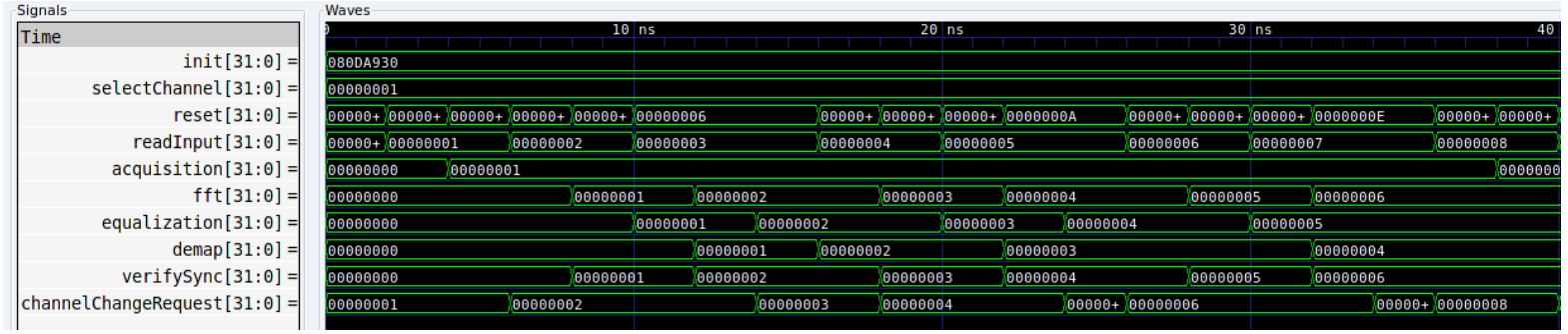
Figure 6.1 shows two traces of two executions of the parallelized DVB-T radio receiver from Figure 1.1. These traces are generated using the SystemC simulation output of the parallelization tool. The traces are visualized using GTKWave [8]. All functions in

all tasks have a constant execution time of 2 ns in this simulation.

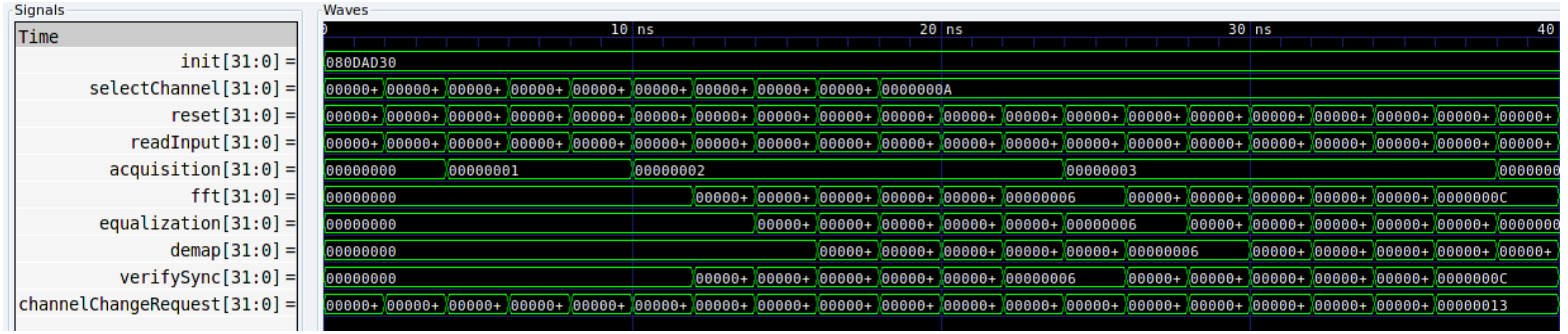
Figure 6.1a shows a trace where there is minimal overlapping of iterations. The buffer capacities are the same as the array sizes, which is the minimum required for this example. Note that this trace already performs better than when barriers where used at the end of each loop iteration because the $equalization$ and $demap$ tasks are not dependent on for example the variable x so the $readInput$ task can already execute the next iteration.

More pipelining can be achieved by increasing the buffer capacities. When the buffer capacities are larger more loop iterations can execute simultaneously as each loop iteration can use a part of the buffer. Figure 6.1b illustrates this with a trace where the buffer capacities are ten times as large as the array sizes. The figure shows that the tasks containing the functions fft, $equalization$ and $demap$ are executed fully pipelined. This pipeline is not interrupted by the loop being present and therefore loop iterations overlap. This can be seen in the figure in the $channelChangeRequest$ task which continues executing even though the pipeline of the above mentioned functions has not finished a single loop iteration. The only interruption in the pipeline is caused by the loss of synchronization in the radio.

Figure 6.1 also shows that throughput is increased when more pipelining can be enabled. When the synchronization is not lost in the radio stream, the $demap$ function outputs a sample every 2 ns, in contrast to the trace with less pipelining where a sample is outputted on average every 11 ns (not completely visible in the figure).



(a) Sample of a trace with minimal pipelining because minimal buffer capacities are used.



(b) Sample of a trace with maximal pipelining due to larger buffer capacities.

Figure 6.1: Parts of the execution traces of the DVB-T radio receiver from Figure 1.1.

7 Conclusions

This thesis presented an approach for the automatic parallelization of nested loop programs (NLPs) containing while-loops with unknown iteration upper bounds. Tasks are extracted from an NLP and circular buffers (CBs) are inserted to facilitate communication between the tasks. For each CB a choice is made between a CB with sliding windows and a CB with overlapping windows. Synchronization is added to the task graph to ensure schedule independent functional behavior that is equivalent to the sequentially executed NLP. We have also shown that deadlock is not introduced by the parallelization process if CBs with overlapping windows are used for all CBs. An extension to CBs with sliding or overlapping windows is proposed where windows can be disabled such that the synchronization overhead for while-loops is reduced. For all CBs in the task graph, sufficient buffer capacities are calculated for a known throughput using a modular cyclo-static data flow (MCSDF) model, which is introduced in this thesis. The generated task graph can be executed on an embedded multiprocessor platform.

Writing parallel programs by hand proves to be a very difficult task. Synchronization statements like lock and unlock must be placed manually by the programmer to guarantee mutual exclusive access to variables and when a mistake is made, this can result in deadlock of the complete program. To solve this problem automatic parallelization techniques are introduced which can automatically transform sequential programs to parallel programs. However, these techniques have their shortcomings, for example there is no support for while-loops in which no upper bound can be given at compile-time, or even during the execution of the loop. While-loops have a number of properties which makes the automatic parallelization difficult to perform. For example a while-loop can have an infinite number of iterations. Single assignment code can therefore no longer be created because arrays with an infinite number of elements will be the consequence.

User dependent loop conditions stop the current parallelization tools from overlapping loop iterations using a pipeline because the condition can not be known before a loop iteration starts. Often if-statements can be found in programs in which each branch writes a result to the same variable. These multiple writers must also not stop pipelining being applied.

This thesis presents a parallelization approach which overcomes these problems. A parallel task graph is extracted from a sequential NLP, which may contain while-loops. CBs are used for communication between tasks. The used CBs contain windows in

which a task has random access. By requiring each task to acquire a location before it can be used and releasing a location after it used, functional correctness is guaranteed by the presented approach. The statements that perform this acquiring and releasing of locations are automatically inserted by the parallelization approach. When overlapping windows are used for all CBs, it is proven that the parallelization technique does not introduce deadlock. The complete program is deadlock free if all single assignment sections (SASs) in the input NLP are valid. A SAS defines the part of a program in which values can be read from a variable and it defines when a variable can be reused. A SAS is valid if all scalars and array elements are written before they are read and if the SAS satisfies dynamic single assignment (DSA).

A performance comparison between CBs with sliding and overlapping windows is made. It shows that overlapping windows are more expensive than sliding windows in terms of the execution time needed to synchronize. However, sliding windows can cause deadlock while overlapping windows can not. An integer linear program (ILP) is presented which determines for each CB if sliding windows can be used or if overlapping windows have to be used.

For given throughput constraints, sufficient buffer capacities are determined for the buffers used by the parallelization of the NLPs with while-loops. A MCSDF model is presented, which is an extension to a cyclo-static data flow (CSDF) model where an actor can consist of another CSDF model. When a MCSDF model is used as an actor, it is called a composite actor, in contrast to atomic actors which are regular CSDF actors. For each while-loop a MCSDF model is created in which nested while-loops are embedded as a composite actors.

It is considered future work to implement the dynamic windows extension and to do perform experiments and measure how much difference the extension makes. It would also be good to know when the extension uses less cycles than when the window would remain enabled.

Another interesting point for future work is to experiment with the throughput constraint and research how much more throughput is achieved with respect to an approach employing barriers. The case-study already hints that for the DVB-T radio receiver the speedup is considerable.

The presented approach is demonstrated using a DVB-T radio receiver, however we expect many other applications with streaming behavior to have a similar kind of requirements on while-loops and can therefore benefit from the presented approach.

Bibliography

[1] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Proc. of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM, 1988.

[2] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. *Design, Automation and Test in Europe (DATE), Dresden, Germany*, 2010.

[3] T. Bijlsma, M. Bekooij, P. Jansen, and G. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *SCOPES '08: Proc. of the 11th international workshop on Software & compilers for embedded systems*, pages 33–42. ACM, 2008.

[4] T. Bijlsma, M. Bekooij, and G. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. Accepted for publication in Transactions on HiPEAC 2009.

[5] Bilsen, G. et al. Cyclo-static dataflow. In *IEEE Transactions on Signal Processing, Vol. 44 No. 2*, pages 397–408, February 1996.

[6] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[7] J.F. Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.

[8] GTKWave developers. GTKWave. `http://gtkwave.sourceforge.net`.

[9] A. Douillet and G.R. Gao. Software-pipelining on multi-core architectures. In *16th International Conference on Parallel Architecture and Compilation Techniques, 2007. PACT 2007*, pages 39–48, 2007.

[10] M. Girkar and C.D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[11] Open SystemC Initiative. SystemC. `http://www.systemc.org`.

[12] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proc. of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328. ACM, 1988.

[13] B. Li, P. van der Wolf, and K. Bertels. TTL inter-task communication implementation on a shared-memory multiprocessor platform.

[14] S. Oaks and H. Wong. *Java threads*. O'Reilly Media, Inc., 2004.

[15] L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. In *Proc. for the 9th International Parallel Processing Symposium*, pages 347–356. Citeseer, 1995.

[16] Hongbo Rong, Alban Douillet, R. Govindarajan, and Guang R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *CGO '04: Proc. of the international symposium on Code generation and optimization*, page 175, Washington, DC, USA, 2004. IEEE Computer Society.

[17] A. Turjan, B. Kienhuis, and E. Deprettere. An integer linear programming approach to classify the communication in process networks. *Software and Compilers for Embedded Systems*, pages 62–76.

[18] C.H. van Berkel. Multi-core for mobile phones. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.

[19] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. *Programming Languages and Systems*, pages 330–346.

[20] E. Wang. A compiler for Silage. *University of California, Berkeley*, 1994.

[21] Maarten H. Wiggers. *Aperiodic Multiprocessor Scheduling for Real-time Stream Processing Applications*. PhD Thesis University of Twente, June 2009.

[22] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC '07: Proc. of the 44th annual Design Automation Conference*, pages 658–663, New York, NY, USA, 2007. ACM.

[23] Xilinx. Virtex-6 FPGA ML605 Evaluation Kit. `http://www.xilinx.com/products/devkits/EK-V6-ML605-G.htm`.

[24] Xilinx. Microblaze processor reference guide. *reference manual*, 2006.

A

Paper

This appendix contains the paper written for my masters program. It presents an approach for parallelizing nested loop programs (NLPs) containing while-loops with an unknown number of iterations.

Parallelization of While Loops in Nested Loop Programs for Shared-Memory Multiprocessor Systems

Authors suppressed for blind review

***Abstract*—Many applications contain loops with an undetermined number of iterations. These loops have to be parallelized in order to increase the throughput when executed on an embedded multiprocessor platform. This paper presents a method to automatically extract a parallel task graph based on function level parallelism from a sequential nested loop program with while loops. In the parallelized task graph loop iterations can overlap during execution. We introduce the notion of a single assignment section such that we can use dynamic single assignment (DSA) to overlap iterations of the while loop during the exection of the parallel task graph. Synchronization is inserted in the parallelized task graph to ensure the same functional behavior as the sequential nested loop program. It is shown that the generated parallel task graph does not introduce deadlock. A DVB-T radio receiver where the user can switch channels after an undetermined amount of time illustrates the approach.**

I. Introduction

Radios process infinite input streams until the user indicates that a switch to a different channel is needed. After the switching reinitialization has to be performed in order to process the new input stream. See for example the radio receiver as shown in Figure 1 which demonstrates this behavior. Parallelizing this application presents a number of difficulties, such as multiple writers to a single variable, user dependent loop conditions and an endless loop. Similar behavior is encountered in other applications which show streaming behavior, such as a video decoder or a telephone modem. The looping behavior corresponds to a while loop with a termination condition that depends on, for example, the user input.

These stream processing applications are developed as sequential code after which a time consuming and error prone manual parallelization step is needed to convert the sequential application into a parallel task graph that can be executed on multiple processors.

Automatic parallelization tools have been developed to ease this manual process of converting a sequential program to a parallel task graph. However, to the best of our knowledge, none of these tools can handle while loops where iterations can overlap in combination with function level parallelism and are therefore less suitable for the above mentioned stream processing applications.

In this paper we introduce a while loop for use in a nested loop program (NLP) where the number of iterations and the loop termination condition can be determined during the execution of the loop. The NLP is parallelized automatically by our tool that extracts function level parallelism. Buffers are created from shared variables and synchronization is inserted to ensure that data is written before it is read and remains available until it is no longer used. It is shown that our parallelization does not introduce deadlock.

Function level parallelism is extracted from an NLP by creating a task from each function. This creates a pipeline,

```
def channel_t channel;
def data_t x, y, z;
def int state;

loop {
 channel = selectChannel();
 reset(out state);
 loop{
  x = readInput(channel);
  switch(state){
   case 0:{ acquisition(x, out state'); }
   case 1:{
    y = fft(x);
    z = equalization(y);
    demap(z);
    verifySync(x, out state');
   }
  }
 } while(!channelChangeRequest());
} while(1);
```

Fig. 1: NLP of a simplified DVB-T radio receiver.

enabling a higher possible throughput on a multiprocessor platform for applications which execute the same functions often. From the extracted tasks a task graph is created with the same functional behavior as the sequential NLP. In the task graph the shared variables from the sequential NLP form the edges in the graph and the tasks the nodes. Each shared variable is translated to a circular buffer (CB) where overlapping windows [4] are used. A CB with overlapping windows can contain multiple read and write windows, which can overlap eachother. In a window a reading task (consumer) or writing task (producer) can use any access pattern to read or write values, something which is not possible in a first-in first-out (FIFO) buffer. It is also possible to read locations multiple times and skip locations. In a read window a consumer can only read values and in a write window a producer can only write values. After a location is written it can not be written again (called single assignment) until all read windows no longer need the value at that location. Single assignment is required because the reader has no knowledge when which of the multiple producers actually writes to a location. Each producer has a set of full-bits where each full-bit corresponds with a location in the CB and indicates if the location contains data.

CBs with overlapping windows employ a release consistency memory model where both data and space have to be acquired and released in order to access shared data [8]. See Figure 2 for an overview of the operations that are possible on a buffer. Before a producer can write data, empty locations (space) have to be acquire first. After a loction is acquired, data can be written and the location has to be released. The consumer can then acquire the data, read it and release the locations as space again. Acquiring and releasing space can

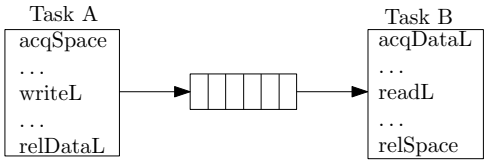


Fig. 2: Operations on a buffer. Task A is a task that writes to the buffer and task B is a task that reads from the buffer.

only be done on consecutive locations, for data this does not have to be the case. The acquire statements move the front of the window forward and can block, the release statement move the tail of the window forward and can not block. After a consumer acquired a location, it can be read multiple times. Using these buffers can result in more parallelism compared to using barriers at the end of each iteration because parts of iterations can now overlap. Bijlsma also introduced circular buffers with windows where the read and write windows are not allowed to overlap [3]. However it is shown that this buffer type can not guarantee a deadlock free execution for all input programs [4]. Therefore we use CBs with overlapping windows, which we show not to introduce deadlock. The NLP has to satisfy single assignment in all so called single assignment sections, which are introduced in this paper.

This paper is organized as follows. Section II presents related work. Section III presents the notion of a single assignment section. This single assignment section is used to parallelize while loops, as discussed in Section IV. Section V shows that the code from the parallelization process is deadlock free. Section VI discusses a case-study where a radio application with while loops is parallelized. Section VII presents the conclusions and gives some ideas for future work.

II. Related Work

There are a number of existing techniques which attempt to parallelize while loops. Rauchwerger [9] is able to parallelize while loops which contain linked lists. In contrast to our approach, data level parallelism is used to execute multiple loop iterations simultaneously. Our solution employs function parallelization which results in a pipelined execution of tasks. Collard [5] also uses data level parallelism and speculatively executes the next iteration independent of the loop condition. These parallelization methods are orthogonal to our method and can be combined to create a hybrid approach that exploits data as well as function level parallelism.

The approach presented by Turjan [11] transforms a NLP into a Kahn process network (KPN). The disadvantage of that approach is that it can only handle FIFO buffers. The consequence is that skipping locations, reading locations multiple times or reading locations in a different order than they are written requires a reordering buffer and a potentially complex controller or lookup table. Our buffers do not need such a controller task or lookup table.

Traditionally compilers apply software pipelining to schedule instructions in such a way that multiple iterations of loops can overlap [7], [10]. The compiler which performs instruction level software pipelining knows the execution times and finish times of all instructions and can therefore create a schedule that satisfies all dependency constraints. The difference with our method is that we do not assume a global clock nor do we create a fixed execution order and therefore we need to insert synchronization in order to satisfy all dependency constraints.

Douillet [6] performs software pipelining for multicore architectures. However, multiple producers where it is not known at compile time which producer writes the actual value are not supported by their synchronization statements.

III. Single Assignment Section

A program that is parallelized must be in single assignment form. Single assignment means that a scalar or a element in an array is written only once. If this would not be the case, a task that reads from a CB does not know when the relevant writer has written its value. For example the variable $state$ from Figure 1 is written by three different tasks. Two different forms of single assignment exists, static single assignment (SSA) and dynamic single assignment (DSA). SSA [1] means that there is only a single assignment statement in the code that writes to a scalar or array. DSA [12] requires that a scalar or a element in an array is written only once during the execution of the program. A program in SSA form is not necessarily in DSA form or vice versa.

However, both notions of single assignment are not sufficient for the while loop with an unknown iteration upperbound. SSA can not handle multiple assignments to a single variable, therefore if-statements with multiple branches writing to the same variable are problematic. DSA can not guarantee a finite array size because each loop iteration requires array elements which are not yet written. Because the loop can potentially execute an infinite amount of times, this would require an infinite number of array elements and therefore also an infinite array size. For example in Figure 1 the variable $state$ is written by three different statements, therefore the code does not satisfy SSA. The variable x is written an infinite number of times and therefore this example is also not DSA. This code can also not be rewritten such that DSA holds because an array can not have an infinite number of elements.

We therefore introduce the notion of a *single assignment section (SAS)*. During the execution of a SAS each scalar and array element can be written only once. A SAS belongs to a scalar or a complete array. Therefore, at each point in the code multiple SASs may exist and a scalar or array can have multiple SASs. At the end of a SAS the value of the variable from the SAS is lost, giving a variable a *temporal scope*. Temporally scoped means that the value of a variable is not only bounded by the location in the program, but also by time.

The length of each SAS is defined at compile time by the semantics of the sequential NLP. The first SAS of all variables always starts at the beginning of a NLP. For each scalar or array written before the end of the while loop, its corresponding SAS ends at the end of the loop. This SAS is executed as often as the number of iterations of the loop. A new SAS is started by the statements after the while loop, if any. From the second iteration onwards the SAS has a more limited scope from the start of the loop until the end of the

```
int x;
x = 3;
do{
 int x, y;
 x = 5;
 y = 2;
 print(y);
} while(...);
print(x);
```

(a) Legal C program

```
def int x;
x = 3;
loop{
 def int x, y;
 x = 5;
 y = 2;
 print(y);
} while(...);
print(x);
```

(b) Illegal NLP

Fig. 3: Sequential program as C code on the left and as NLP on the right.

loop as this part of the code contains the repeating statements. Note that the variables in the loop condition are part of the loop and therefore belong to the same SAS as if the variables were written inside the loop. If a scalar or array is not written before the end of the while loop, its SAS does not end at the loop but continues until the end of first loop where it is written in or until the end of the NLP, whichever comes first.

Consider for example the program in Figure 3 which shows a program with a loop. The program from Figure 3a is valid according to the semantics of C. However, it is illegal as an NLP containing SASs, see Figure 3b. When the program is executed as C the print of the variable x would give 3 because the scope of the x in the loop ends at the end of the loop. When the program is used as an NLP, two problems arrise. The first problem is that the variable x is written two times in the same SAS, first with the value 3 and then with the value 5. The second problem is that the printing of x uses a value of x that is not yet written as the last write action to x was in another SAS.

A SAS is valid if its corresponding variable is in DSA form within the SAS and all locations are written before they are read. The complete program is valid if all SASs of all variables are valid.

Because the value of all variables is temporally scoped, the values are destroyed at the end of a SAS. Because a SAS ends at the end of a loop iteration there is no way to pass values to the next loop iteration. To solve this we re-use the stream concept also found in Silage [13]. In Silage a variable is essentially a stream which can be shifted forward in time using the "@" operator. Also in synchronous guarded actions as proposed by Baudisch [2] this concept can be found in the form of the $next$ operator. This $next$ operator delays the write of a value by one marco step. In our case the marco step is the execution of a SAS. This concept of writing to the next execution of a SAS is annotated with a ' in our NLPs. A direct consequence is that this scalar or array element can not be written anymore when executing the next SAS as this would violate the DSA requirement.

IV. Parallelization of While Loops

In our approach function level parallelism is extracted from while loops. This means that each function in the NLP becomes a task in the task graph. Communication between the tasks is done via CBs. Circular buffers which support windows are used because they allow for flexibility when elements in arrays are produced out-of-order or when elements are not read [3], [4]. These CBs require that all of the windows in a buffer are moved an equal amount of times through the buffer during execution. If a location is released from the window the value at that location can no longer be acquired again by the same window.

From the input NLP a task graph is extracted. A task graph is a directed graph $H = (T, S, A, \sigma, \theta)$. The set of vertices is T, where each $t_i \in T$ represents a task. The set of hyperedges is S, where each $(V, W) \in S$, with $V, W \subseteq T$ represents a buffer. Each buffer s_i, with $s_i \in S$, corresponds with an array a_i, with $a_i \in A$. This array corresponds to a variable declared in the NLP. The capacity of each buffer s_i is given by $\theta(s_i)$, with $\theta : S \to \mathbb{N}$. The size of each array a_i is given by $\sigma(a_i)$, with $\sigma : A \to \mathbb{N}$.

A hyperedge is an edge which can have multiple source and/or destination nodes. For example in the task graph from Figure 4b a hyperedge is present for the buffers s_x and s_b. In the context of buffers a hyperedge $s_i = (\{t_i | t_i \in T\}, \{t_j | t_j \in T\})$ means that there is at some point in time a location written by all source tasks t_i and read by all destination tasks t_j. The consequence for buffers with overlapping windows is that each source and destination task have a window in the buffer and therefore need to move this window correctly. During the execution of a SAS there can be only one producer which writes a value to a location (DSA), eventhough there can be more producers. If a hyperedge has multiple consumers, all of these tasks can read any location, as long as it is written first.

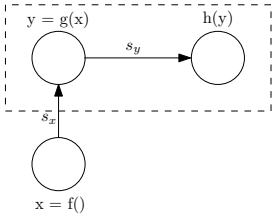
For the task graph to produce correct results, synchronization statements have to be inserted for all shared variables. There are three cases each in which synchronization statements are added at different locations in the program. In the sections IV-A, IV-B and IV-C these cases will be discussed.

A. Read-Only Variables

The simplest case to insert synchronization occurs when a variable is only written in the statements before a loop. Consider the example NLP in Figure 4a where the value of x is determined before the loop construct. After parallelization two buffers, s_x and s_y, are created and since there are three functions in the example, also three tasks are created. These tasks are shown in Figure 5.

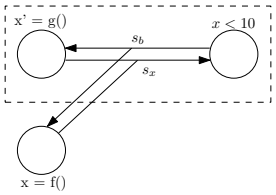
In the example, the only writer of the variable x is the function f. This means that the SAS for x starts at the beginning of the program from Figure 4a and ends at the end of the program as there are no other while loops. Inside the loop the variable y is written by the function g. Therefore a SAS for y is created from the beginning of the program to the end of the while loop. From the second iteration of the loop and onwards the SAS for y is defined from the start of the loop until the end of the loop. This program is valid as all SASs are in DSA form.

The synchronization statements for the buffers which are only written by the statements before the loop are inserted before and after any while loop in which they are read. In the example the synchronization statements for buffer s_x are inserted before and after the while loop. Synchronization




```
def int x;
def int y;
loop{
 x = f();
 loop{
  y = g(x);
  h(y);
 } while(...);
} while(1);
```

(a) NLP with a variable that is only written in the statements before the while loop. The dots in the loop condition indicate any condition is allowed.




```
def int x;
loop{
 x = f();
 loop{
  x' = g();
 } while(x < 10);
} while(1);
```

(b) NLP with a variable that is written in the statements before the while loop and also in the while loop.

Fig. 4: Two cases where different synchronization statements are added. The task graph is shown at the top of each NLP. The contents of the dashed box are the tasks in the inner loop.

Task 1

```
do{
 acqSpace(x);
 x[0] = f();
 relDataL(x, 0);
} while(1);
```

Task 2

```
do{
 acqDataL(x, 0);
 do{
  acqSpace(y);
  y[0] = g(x[0]);
  relDataL(y, 0);
 } while(...);
 relSpace(x);
} while(1);
```

Task 3

```
do{
 do{
  acqDataL(y,0);
  h(y[0]);
  relSpace(y);
 } while(...);
} while(1);
```

Fig. 5: Parallelized code for the NLP in Figure 4a.

statements for buffers written in the statements in the loop are inserted in the loop. In the example this is done for the buffer s_y.

It would also be possible to insert the synchronization for read-only variables in the loop. However, due to the temporal scope of the variables all values would be lost after the first iteration of the loop. A method to prevent this is to copy the old value to the next SAS execution using the assignment statement "$x'= x$;". This statement must be inserted in the loop. The disadvantages are that a new task is created for this statement and the synchronization overhead is significantly increased as every loop iteration synchronization must be performed for the copy action.

B. Shared Variable Before the Loop

It can also be the case that an array is written in the statements before the loop as well as in the statements inside the loop. The generated synchronization code will be inside the loop as the SAS is defined from before the loop until the end of the loop. For all tasks extracted from functions in the statements before the while loop, extra synchronization statements have to be inserted, if buffers are used which are also written in the statements in the while loop. This extra

Task 1

```
int t, i;
do{
 acqSpace(x);
 x[0] = f();
 relDataL(x,0);
 do{
  acqDataL(b,0);
  t = b[0];
  relSpace(b);
  if(t){
   acqSpace(x);
   relDataL(x,0);
  }
 } while(t);
 acqSpace(x);
 relDataL(x,0);
} while(1);
```

Task 2

```
int t, i;
do{
 acqSpace(x);
 relDataL(x,0);
 do{
  acqSpace(x);
  x[0] = g();
  relDataL(x,0);
  acqDataL(b,0);
  t = b[0];
  relSpace(b);
 } while(t);
} while(1);
```

Task 3

```
int t, i;
do{
 do{
  acqDataL(x,0);
  t = (x[0] < 10);
  relSpace(x);
  acqSpace(b);
  b[0] = t;
  relDataL(b,0);
 } while(t);
 acqDataL(x,0);
 relSpace(x);
} while(1);
```

Fig. 6: Parallelized code for the NLP in Figure 4b.

synchronization is required by the used CBs. The CBs require that all windows are no more than one iteration apart because a write window can never overtake a read window and a read window can never overtake the write window that is the furthest in its execution.

An example NLP which writes to a variable both in the statements before and in the loop is given in Figure 4b. The variable x is written before the loop and also inside the loop, this forces the synchronization to be inside the loop. The SAS for x is defined from the start of the program up to and including the loop condition. The generated code for this example can be found in Figure 6. Task 1 from Figure 6 shows that the space acquired before the loop is conditionally acquired for the next execution of the same SAS, depending on if the same SAS is actually executed again. When this task is created using this scheme and executed, the synchronization is done equally often as the synchronization of all other tasks.

Figure 6 also shows that a new task is created when a variable or function is used in the loop condition. In the figure a new buffer b is created to store the condition value.

Note that sometimes renaming the variables written to before the loop can decrease the number of tasks in which an extra loop must be added. For example if there are many producers before the loop for a buffer x this would result in many tasks which would all need the extra loop. However, if these tasks write to a new variable y which is not written in the loop, no extra loops must be added for y. Only a copy statement must be added to copy y to x to make the values available for the statements in the loop. This copy task will be the only task with the extra loop as this is the only task which writes to a shared variable.

C. Shared Variable After the Loop

The NLP from Figure 7 gives an example of a variable being used in the statements before, in and after the first while loop. Since the example has statements using x after the first while loop, a second SAS is created for x next to the SAS ending at the first while loop. This second SAS starts after the first while loop and ends at the end of the second while loop.

As before, all tasks have to synchronize an equal number of times during the execution of the program. Therefore also

```
def int x;
loop{
 x = f();
 print(x);
} while(...);
loop{
 x = g();
 print(x);
} while(...);
```

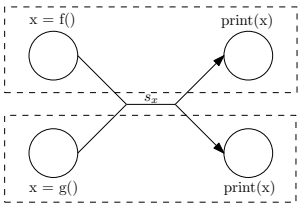


Fig. 7: NLP with two successive while loops.

for the tasks that use a value after the loop, a loop containing synchronization statements must be added. The consequence for the example is that all tasks need both while loops.

It might be more intuitive to have the last SAS span beyond the while loop, thus letting variable values also be available after the first loop. For instance by extending the temporal scope of a variable in the last iteration of the loop to include statements after the loop. However, this will cause problems as SASs can now overlap. Consider for example the NLP from Figure 7 where there are two while loops after each other. If the values from a SAS would be valid until after the loop, x will be written twice, once in the last iteration of the first loop and once in the first iteration of the second loop. The consequence is that the code is no longer DSA. Therefore a SAS runs up to the end of a loop.

V. Deadlock Freedom

In this section we give the essence of a prove that using CBs with overlapping windows never introduces deadlock if all buffer capacities are at least the array sizes and the SASs in the sequential NLP are all valid.

The parallel task graph can only deadlock due to a cyclic dependency caused by the insertion of acquire statements because these are the only blocking operations, see also Figure 2. By construction the parallelized task graph with inserted synchronization statements can always execute deadlock free using the sequential order defined in the NLP as a schedule, assuming the sequential NLP was valid. It is possible that when the application executes, it deviates from this schedule. However, since the sequential application is deadlock free with the extra synchronization statements inserted using the same method as for the parallel program it must also be deadlock free when executed in parallel because the sequential order constraints are removed. Removing constraints can never introduce deadlock as no cycles can be formed in a graph by removing edges.

To show that the parallel task graph with synchronization statements inserted does not introduce deadlock, we have to consider the insertion of acquire statements in more detail. In CBs with overlapping windows there are two blocking acquire statements, one statement for data and one statement for space in the buffer.

First we consider the simple case where space is acquired in order to write data to the CB. Initially, no location is acquired in the buffer. By construction, each task can acquire a location in the buffer maximally once during the execution of a SAS. Therefore, at most the array size of locations in the buffer can be acquired by a task during a single execution of a SAS. Because we assumed the buffer capacity is at least the array size, the consequence is that during the first execution of any SAS all acquire calls for space can succeed. By construction all locations in the buffer are released before the end of the execution of a buffer's SAS, therefore after the execution of all first SASs all windows are again in their initial position and the process can start from the start again. Because all locations are released eventually during the execution of a SAS, an acquire for space from the second execution of an SAS will always eventually succeed.

Consider now the second case where data is acquired. Suppose the acquire and release statements are inserted in the sequential code using the following method. If an element in an array is written, the release of data for the corresponding location in the buffer is inserted in the NLP immediately after the element is written. The acquire statement for data is inserted in the NLP just before the location is read. In the NLP all acquire statements use an unique window. Because we assumed that in the sequential NLP every element is written before it is read, a release of data for location i is always performed before the acquire of space for location i. Note that read windows can overlap each other so a location can be acquired by multiple read windows during a single iteration. The parallelization process inserts these synchronization statements for data using the same method, so essentially a task is formed from every function and the corresponding inserted synchronization statements. Since in the parallelized task graph the only change with respect to the sequential NLP is the removal of the sequential order constraints, the acquire and release of data does not cause deadlock.

For a parallelized while loop the acquire statements for variables which are not written in the loop are moved to immediately before the loop compared to immediately before the statement. As all SASs were assumed to be valid in the sequential NLP, all written locations in the buffer are still released by a producer before they are acquired by a consumer. Therefore, the parallelization of the while loop does not introduce deadlock.

Since neither the acquire for space nor the acquire for data introduces deadlock, CBs with overlapping windows are deadlock free provided that the sequential NLP is valid.

VI. Case-study

This section illustrates our parallelization approach for while loops by means of a case-study. Figure 1 shows the structure of a, for explanatory reasons simplified, DVB-T receiver application which switches between two modes, named 0 and 1 here. After a potentially infinite amount of time, the user can switch between channels and the application will break the inner while loop and switch channels. This whole flow is repeated infinitly often.

The task graph for the program from Figure 1 can be found in Figure 8. As there are eight functions in the NLP, also eight tasks are created. Because the loop condition is a function which we want to execute only once in one iteration, the return value of the function must be distributed to all tasks in the loop. The buffer t is created for this purpose.

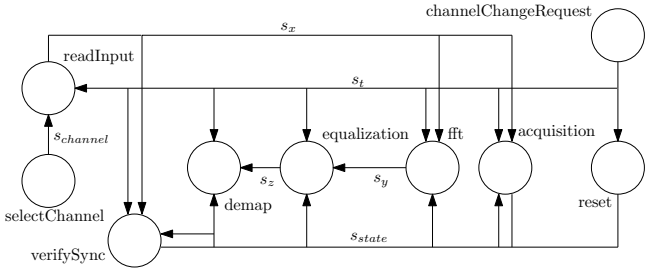


Fig. 8: Task graph of the inner while loop of the radio receiver from Figure 1. The buffer t is added to distribute the loop condition to all relevant tasks.

Because the $equalization$ and $demap$ functions are inside the switch statement, the tasks formed from these function also need read access to the variable $state$. An improved version of the parallelization process can remove this edge in the task graph as y and z already enfore this control flow. Note that the tasks that read from the CB $state$ can be pipelined, even though the loop condition is data dependent. Each of these tasks needs a different value of the variable $state$ if they are pipelined because they are in different iterations of the loop. Therefore, the CB $state$ needs a capacity of five.

The $state$ variable is a variable which is shared between the statements before the loop and in the loop. Therefore the task created from the assignment of 0 to $state$ must also contain the while loop. The synchronization statements are in the while loop for all tasks that use the $state$ variable. The $channel$ variable is a variable which is only written in the statements before the while loop. The synchronization statements for the $readInput$ task are therefore outside of the loop.

VII. Conclusions

We have introduced an approach for the automatic parallelization of nested loop programs (NLPs) that contain while loops with unknown iteration upper bounds. Tasks are extracted from the NLP and circular buffers (CBs) are inserted to facilitate communication between the tasks. Synchronization is added to ensure schedule independent functional behavior that is equivalent to the sequentially executed NLP. We have also shown that deadlock is not introduced by the parallelization process.

From each function in the NLP a task is formed. Dependencies between variables are analyzed and CBs are substituted for all shared variables. Synchronization statements to acquire and release data are inserted using three different schemes in such a way that synchronization can be performed for a while loop with an unknown iteration upperbound.

We have introduced the notion of a single assignment section (SAS), which defines the life-time of the variables from an NLP. The life-time of the variables is defined such that even in the case of an unbounded number of iterations, dynamic single assignment (DSA) can be defined for the NLP. The DSA property allows for a pipelined execution of the statements in the while loop.

The CBs use overlapping windows, for which we have shown that the execution is deadlock free if the buffer capacities are at least the array sizes. Using these buffers enables the extracted parallel program to pipeline across loop iterations.

The presented approach is demonstrated on a DVB-T radio that contains a while loop where the loop condition depends on user input. Despite the multiple producers writing to a single variable and consumers reading from a single variable, the while loop can be parallelized in such a way that iterations can overlap.

Potentially interesting future work is to search for an extension for circular buffers with overlapping windows where windows can be temporarily disabled. Instead of inserting the extra loops needed to move the windows an equal amount of times, the unused windows can be disabled such that it does not have to move in pace with the other windows, thus reducing synchronization overhead. The window can be reactivated if it is needed again. In this case the extra loop in Task 1 from Figure 6 would no longer be needed.

The presented approach is demonstrated for a single application, however we expect many other applications with streaming behavior to have a similar kind of requirements on while loops and can therefore benefit from the presented approach.

References

[1] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Proc. of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM, 1988.

[2] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. *Design, Automation and Test in Europe (DATE), Dresden, Germany*, 2010.

[3] T. Bijlsma, M. Bekooij, P. Jansen, and G. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *SCOPES '08: Proc. of the 11th international workshop on Software & compilers for embedded systems*, pages 33–42. ACM, 2008.

[4] T. Bijlsma, M. Bekooij, and G. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. Accepted for publication in Transactions on HiPEAC 2009.

[5] J.F. Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.

[6] A. Douillet and G.R. Gao. Software-pipelining on multi-core architectures. In *16th International Conference on Parallel Architecture and Compilation Techniques, 2007. PACT 2007*, pages 39–48, 2007.

[7] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proc. of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328. ACM, 1988.

[8] B. Li, P. van der Wolf, and K. Bertels. TTL inter-task communication implementation on a shared-memory multiprocessor platform.

[9] L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. In *Proc. for the 9th International Parallel Processing Symposium*, pages 347–356. Citeseer, 1995.

[10] Hongbo Rong, Alban Douillet, R. Govindarajan, and Guang R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *CGO '04: Proc. of the international symposium on Code generation and optimization*, page 175, Washington, DC, USA, 2004. IEEE Computer Society.

[11] A. Turjan, B. Kienhuis, and E. Deprettere. An integer linear programming approach to classify the communication in process networks. *Software and Compilers for Embedded Systems*, pages 62–76.

[12] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. *Programming Languages and Systems*, pages 330–346.

[13] E. Wang. A compiler for Silage. *University of California, Berkeley*, 1994.