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

Stefan J. Geuns*, Marco J.G. Bekooij†, Tjerk Bijlsma‡, Henk Corporaal*
*Eindhoven University of Technology, †NXP Semiconductors, ‡Twente University
stefan.geuns@gmail.com

***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 exploit single assignment to overlap iterations of the while loop during the execution 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 that 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 often 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 the execution of 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 an 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 approach does not introduce deadlock.

978-3-9810801-7-9/DATE11/©2011 EDAA

```
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.

Function level parallelism is extracted from an NLP by creating a task from each function. This creates a pipeline that enables 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) in which overlapping windows [1] are used, such that pipelining can be exploited. For shared-memory platforms, these CBs can be implemented efficiently.

A CB with overlapping windows can contain multiple read and write windows, which can overlap each other. 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, see Section IV. 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 [2]. 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 acquired first. After a location is acquired,

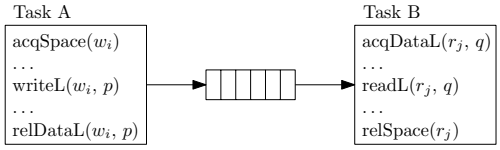


Fig. 2: Operations on a buffer. Task A is a task that writes to location p in a buffer using write window w_i. Task B is a task that reads from location q in the same buffer, using read window r_j.

```
loop{
 x[F(i)] = ...
 x[G(i)] = ...
 ... = x[H(i)];
} while(1);
```

Fig. 3: NLP where DSA is required for a pipelined parallel execution.

```
loop{
 x=A();
 loop{
  x=B();
 } while(...);
 C(x);
} while(1);
```

Fig. 4: NLP with statements before, in and after the loop.

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 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 statements 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 [1]. 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 gives the basic idea of the presented approach. Section IV presents the notion of a single assignment section. This single assignment section is used to parallelize while loops, as discussed in Section V. Section VI shows that the code from the parallelization process is deadlock free. Section VII discusses a case-study where a radio application with while loops is parallelized. Section VIII 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 [4] 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 level parallelization which results in a pipelined execution of tasks.

Collard [5] also extracts 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 [6] transforms an 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.

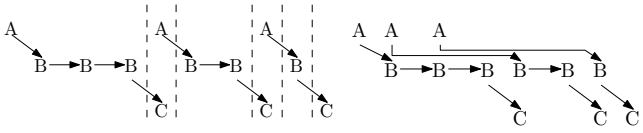
Traditionally compilers apply software pipelining to schedule instructions in such a way that multiple iterations of loops can overlap [7], [8]. The compiler which performs instruction level software pipelining knows the execution times and finish times of all instructions and can therefore create a 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.

III. Basic Idea

The parallelization process starts with extracting tasks from the sequential NLP. A program that is parallelized must be in single assignment form. Consider for example the NLP from Figure 3 in which two producers write to the same array x, both using an unknown data dependent index expression. If dynamic single assignment (DSA) [10] was not guaranteed, the given sequential order must be preserved because the second assignment can possibly overwrite the result of the first assignment. Therefore, no parallelism can be extracted from this NLP without DSA. However, DSA can not be required on the complete program as this is impossible for infinite sized loops. Therefore, it is required that DSA holds for each loop body and initialization statements of that loop. Because DSA can not be verified automatically, NLPs are assumed to be in DSA form.

Figure 4 shows an NLP with a statement using the same variable before, in and after a loop. The optimal schedule occurs when there is pipelining between the statements outside and inside the loop, see Figure 5b. This in contrast to when a barrier is used, see Figure 5a. To achieve the optimal schedule, the tasks extracted from the functions A and C must know if there are enough empty or full buffer locations after the loop has ended. Therefore, these tasks must synchronize at the same rate as task B and therefore an extra loop is added to the corresponding tasks.



(a) Schedule with barriers, indicated by the dashed lines.

(b) Schedule without barrier.

Fig. 5: Possible schedules with and without barriers.

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

(a) Legal C program

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

(b) Illegal NLP

Fig. 6: Sequential program as C code on the left and as NLP on the right.

IV. Single Assignment Section

A program that is parallelized must be in single assignment form. Single assignment means that a scalar or an 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 multiple times by three different tasks. Two different forms of single assignment exists, static single assignment (SSA) and DSA. SSA [11] means that there is only a single assignment statement in the code that writes to a scalar or array. DSA requires that a scalar or an element in an array is written only once during the execution of a 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 while loops with an unknown iteration upperbound. SSA can not handle multiple assignment statements 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 that are not yet written. For an infinite loop, this would result in 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.

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 an 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 the first loop where it is written in or until the end of the NLP, whichever comes first.

Consider for example the program in Figure 6 which shows a program with a loop. The program from Figure 6a is valid according to the semantics of C. However, it is illegal as an NLP containing SASs, see Figure 6b. 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.

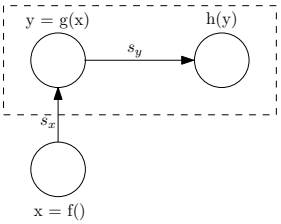
Because the value of all variables is temporally scoped, the values are destroyed at the end of a SAS. Because a SAS ends at the end of a loop iteration there is no way to pass values to the next loop iteration. To solve this we re-use the stream concept also found in Silage [12]. In Silage a variable is essentially a stream which can be shifted forward in time using the "@" operator. Also in synchronous guarded actions as proposed by Baudisch [13] 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.

V. 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 [1], [3]. 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.

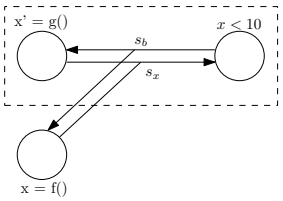
From the input NLP a task graph is extracted. A task graph is a directed graph $H = (T, S, A, \sigma, \theta)$. The set of vertices is T, where each $t_i \in T$ represents a task. The set of hyperedges is S, where each $(U, V) \in S$, with $U, V \subseteq T$ represents a buffer. Each buffer s_i, with $s_i \in S$, 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 7b a hyperedge is present for the buffers s_x and s_b. In




```
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. 7: 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.

the context of buffers, a hyperedge $s_i = (U, V)$, with $U, V \subseteq T$, means that every task in U potentially writes to s_i and every task in V potentially reads from s_i. 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. 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 V-A, V-B and V-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 7a 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 8.

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 7a 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

Task 1

```
do{
 acqSpace(x);
 writeL(x,0,f());
 relDataL(x,0);
} while(1);
```

Task 2

```
do{
 acqDataL(x,0);
 do{
  acqSpace(y);
  writeL(y,0,
    g(readL(x,0)));
  relDataL(y,0);
 } while(...);
 relSpace(x);
} while(1);
```

Task 3

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

Fig. 8: Parallelized code for the NLP in Figure 7a.

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.

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 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 7b. 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 9. Task 1 from Figure 9 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 9 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

Task 1

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

Task 2

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

Task 3

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

Fig. 9: Parallelized code for the NLP in Figure 7b.

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 4 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 inner 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 inner while loop and ends at the end of the outer 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 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 from the same variable can now overlap each other. Consider the example NLP from Figure 4 again. The first SAS, containing the inner while loop, will overlap with the second SAS, containing C. If C would also write to x, this will violate DSA. Therefore a SAS runs up to the end of a loop.

VI. 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 times the number of simultaneous loop iterations used 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, 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 sequential NLP with synchronization statements inserted does not introduce deadlock, we have to consider the insertion of synchronization statements in more detail. From the sequential NLP an NLP with synchronization is created by inserting the acquire statements for a function immediately before the function and the release statements immediately after this function, analogously to the insertion into the parallel task graph. Because we assume that all SASs are valid, a written location is always released before it is acquired for reading. Empty locations can always be acquired as the buffer capacity is, by assumption, at least as large as the number of executed $acqSpace$ statements per write window. Read locations are always released by construction, therefore all following loop iterations can acquire them again.

For a while loop in the parallel task graph the acquire statements for variables which are not written in the loop, are moved to immediately before the loop. As all SASs were assumed to be valid in the sequential NLP, all written locations in the buffers are still released before being acquired for reading. Therefore, the parallelization of the while loop does not introduce deadlock.

Since the sequential NLP with synchronization statements inserted does not introduce deadlock and the parallelization process only splits this NLP, CBs with overlapping windows are deadlock free, provided that the sequential NLP is valid.

VII. 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 infinitely often.

The task graph for the NLP from Figure 1 can be found in Figure 10. 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 five 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 executed in different iterations

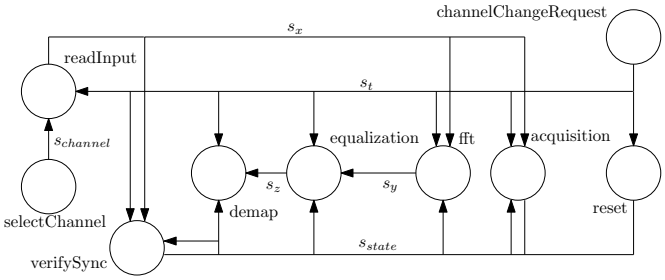


Fig. 10: Task graph of the inner while loop of the radio receiver from Figure 1. The buffer t is added to distribute the loop condition to all relevant tasks.

of the loop. Therefore, the CB $state$ needs a capacity of five locations. This pipelining behavior can be seen in the schedule in Figure 11, which is generated from the execution of the radio receiver. The five tasks are each in different loop iterations, indicated by the numbers in the schedule.

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 $reset$ statement 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.

VIII. 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 an 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.

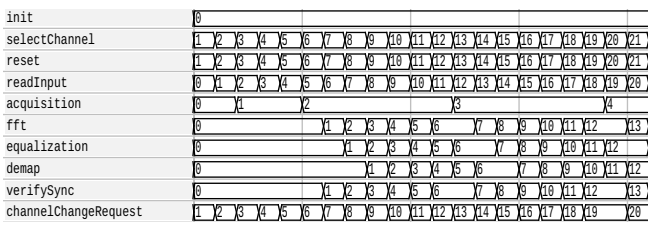


Fig. 11: Scheduling trace of the radio receiver from Figure 1.

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 times the number of used iterations. Using these buffers enables the extracted parallel program to be pipelined across loop iterations.

The presented approach is demonstrated on a DVB-T radio receiver that contains a while loop where the loop condition depends on user input. Despite the multiple producers writing to a single variable and multiple 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 9 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 similar requirements on while loops and can therefore benefit from the presented approach.

References

[1] T. Bijlsma, M. Bekooij, and G. Smit, "Circular buffers with multiple overlapping windows for cyclic task graphs," in press.

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

[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*. ACM, 2008, pp. 33–42.

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

[5] J. Collard, "Automatic parallelization of while-loops using speculative execution," *International Journal of Parallel Programming*, vol. 23, no. 2, pp. 191–219, 1995.

[6] 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*, pp. 62–76.

[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*. ACM, 1988, pp. 318–328.

[8] H. Rong, A. Douillet, R. Govindarajan, and G. 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*. Washington, DC, USA: IEEE Computer Society, 2004, p. 175.

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

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

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

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

[13] 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.