TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Instruction Scheduling for Exposed Datapath Architectures

BachelorThesis

by

Anika Debora Lütke-Bordewick

August 19, 2022

Technische Universität Kaiserslautern,
Department of Computer Science,
67663 Kaiserslautern,
Germany

Examiner: Prof. Dr. Schneider
M.Sc. Julius Roob

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Instruction Scheduling for Exposed Datapath Architectures" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 19.8.2022

Anika Debora Lütke-Bordewick

Abstract

Exposed datapath architectures are a different approach to commercially available processor architectures. They do not work with registers for instructions but with the underlying data dependencies which can be represented by datapath process networks. Scheduling datapath process networks is something done in high level synthesis and is therefore not a new problem. However, exposed datapath architectures work differently than micro-architectures. They do not use registers to store intermediate results which proves to be a challenge when scheduling the datapath process network. In this thesis three commonly known scheduling algorithms are applied to an assorted set of datapath process networks and empirically compared.

Zusammenfassung

Exposed Datapath Architectures (Architekturen mit offengelegtem Datenpfad) bieten einen anderen Ansatz als käuflich erwerbbare Prozessorarchitekturen. Sie arbeiten nicht mit Registern für Instruktionen sondern mit den unterliegenden Datenabhängigkeiten. Diese Datenabhängigkeiten können durch Datapath Process Networks (Datenpfad Prozess Netzwerke, DPN) abgebildet werden. DPNs werden regelmäßig in der high-level Synthese zeitlich eingeplant, daher stellt dies kein neues Problem dar. Exposed Datapath Architectures funktionieren jedoch anders als Mikroarchitekturen. Sie benutzen keine Register um Zwischenergebnisse zu speichern. Dies stellt als eine Herausforderung bei der Planung dar. In dieser Arbeit werden drei allgemein bekannte Planungsalgorithmen auf eine Reihe von DPNs angewendet und anschließend empirisch verglichen.

Contents

List of Figures

1 Introduction

Most modern processor architectures use multiple cores to increase their performance. However, utilising multiple cores often requires an active effort by the programmer for optimal results. This approach is not always feasible. Therefore, efforts have been made to enable processors and compilers to automatically exploit instruction level parallelism(ILP); executing instructions of a sequential program in parallel. This approach has limits. These processor architectures need registers for each instruction. Increasing the amount of registers adds more complexity, i.e. circuits get more complex and need more power. This modern processor architecture does not scale well. A different approach to processor architecture might alleviate some of those problems by abandoning registers.

Exposed datapath architectures expose their processing units (PUs) and the paths between them. This results in them working differently than common processor architectures. A different perspective on programs leads to the realisation that programs are on a very basic level, data dependencies with some control flow. By leveraging explicit knowledge and control over these dependencies, fetching and execution of instruction happen in parallel. In fact, computation is simply the result of moving data along the paths. Data dependencies can be modelled as datapath networks, opening up various new approaches for optimisation. While in high level synthesis scheduling datapath networks is a known and understood problem that is not the case for instruction scheduling for exposed datapath architectures. Instruction scheduling itself is not a focus of research although it can have a crucial impact on the length of the resulting schedules and the required amount of PUs to execute a specific program. Applying traditional scheduling algorithms may result in interesting insights on improving the execution. Either by identifying a traditional scheduling algorithm well suited for this problem or by providing a starting point for developing a new algorithm specific to exposed datapath architectures. In this thesis, three well known scheduling algorithms are adapted to exposed datapath architectures. The algorithms are applied to an assorted set of programs for exposed datapath architectures. The resulting benchmark data is presented and empirically compared.

2 State of the Art

Exposed datapath architectures are a different approach to the commonly used processors. Commercially available processors generally have many cores to further increase their performance. This comes with the downsides that these architectures require more refined programming skills to take full advantage of their abilities. This gets more challenging the more cores a processor has. The problems are the synchronisation and weak memory models. Furthermore, commercially available processors rely on registers which limits IPL. Globally visible registers do not scale well since every single instruction must read and write to the registers. The processor reads the instruction's operands and after execution it writes the result back into a register. Simply increasing the amount of registers is rarely a viable solution, not only because of the registers but also because access times, circuit size and power consumption of register files all rise super-linearly with the number of registers. Of course there are ways to execute single threads more efficiently. These refined techniques, exploiting instruction-level parallelism, are already implemented in current processors and compilers [SBR22b].

This is where exposed datapath architectures come in. They abandon the concept of registers as storage for intermediate results. Instead they communicate the intermediate results directly between their processing units. Exposed datapath architectures simply put are PUs which are connected by fist-in-first-out (FIFO) buffers, i.e. queues. They are also sometime referred to as transport-triggered architectures because the main focus of these architectures is moving data along paths which trigger the computation as a side-effect. In exposed datapath architectures instructions are fetched and executed in parallel simply by considering the respective data dependencies. The allocation of processing units (PUs) in regards to the data dependencies is done by the compiler. Implementing this is hardware would result in complex circuits and those rarely scale well. The compiler needs all the information about the internal datapaths and PUs from the processors to allow the allocation of PUs.

As mentioned previously, regular processors have synchronisation issues. Exposed datapath architectures can use FIFO buffers as communication paths between the PUs. This avoids direct synchronisation between the PUs. These specific architectures are called *buffered* exposed datapath architectures. Novel processor architectures, i.e. SCAD, are hybrid dataflow architectures. This means, the execution of a sequential program mainly uses dataflow graphs and a separate control-flow, for example a program counter and branch instructions. There are, albeit fewer, architectures that do not need a program

counter.

As exposed datapath architectures are highly parallel, a logical approach to increase their performance is exploiting instruction-level parallelism. To achieve that, sequential programs need to be translated into datapath process networks (DPNs) which directly show data dependencies. That in turn enables parallel execution of sequential programs. The translation makes use of expression trees of sequential programs. They describe the dataflow in one such program and when evaluating these expression trees level-by-level the use of IPL is increased. However, the expression trees are not easily evaluated level-by-level in an exposed datapath architecture because the rules of the FIFO buffers can not be ignored. Hence, the expression trees get translated into DPNs [SBR22a].

Besides transport-triggered architectures, there are also other exposed datapath architecture that operate by a different principle. For example Coarse-grained reconfigurable architectures (CGRA) exist. It shares the complexity shift from the architecture to the programmer/compiler. CGRA describes architectures that have spacial granularity at the PU level and temporal configuration granularity at higher level[Vad+19]. However, this is simply another exposed datapath architecture and is just mentioned as that. Unlike the buffered exposed datapath architectures which are the focus of this work.

3 Dataflow Process Networks

3.1 Introduction to Dataflow Process Networks

A DPN is a model of computation. Models of computation generally describe how the output of a mathematical function is computed for a given input and in which way the single atomic actions in the computations are executed. A DPN can be thought of as a directed graph where the nodes each represent a process and the edges FIFO (first-in-first-out) buffers, i.e. queues. The processes are generally atomic actions like an addition. Nodes have a set of predecessors. If node a has an outgoing edge to node b then node a is a predecessor of node b. Nodes also have a set of successors, in this example node b is a successor of node a. The sets describing those successors or predecessors can of course be empty. A node without predecessors is referred to as a *start node*. A node without successors is referred to as either a *sink*, if there is only one node without successors, or as an *end node*/output. Looking at 3.1 x, 3 and 2 are start nodes and z is a sink or an end node [LP95].

These nodes are static for a given DPN. As DPNs are turing complete, they can be used to describe structured sequential programs [Sch21]. Nodes fire as soon as all input data is available thus each node in isolation behaves deterministically. DPNs however may not be deterministic. This is one of the main disadvantages of DPNs. The possible non-determinism of DPNs is outside the scope of this work because the translation process ensures deterministic behaviour of the DPNs by following rules laid out in a seminal paper by Gilles Kahn [KM76][Gil74][Sch21]. This is explained further in section 3.2. The possible non-determinism is still noted for completeness' sake.

Because of the underlying Kahn process network DPNs don't require synchronisation between processes. The Kahn process network is for example responsible for the firing structure of DPNs. Writes to the FIFOs are non-blocking and therefore succeed immediately. Reads however are blocking and the process waits and doesn't execute unless it has all the necessary inputs [LP95] resulting in a distributed and highly parallel model of computation.

3.2 DPNs and Buffered Exposed Datapath Architecture

As previously mentioned, a program running on a buffered exposed datapath architecture can be modelled as a DPN. DPNs directly show the data dependencies which are essential for an exposed datapath architecture to work. The

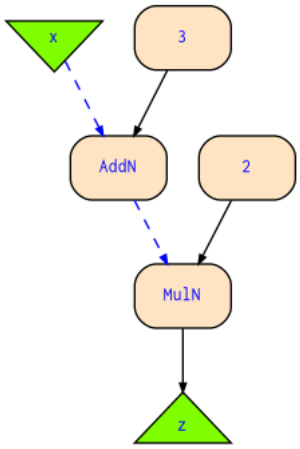

Figure 3.1: *Simple Dataflow Process Network (DPN)*

nodes represent processes that need to be executed by a processing unit (PU).

The modelled DPN needs to behave like a homogeneous synchronous actor, meaning they consume one value of each input buffer and produce one value for each output buffer. This is achieved by making sure there is not more than one value to be stored in each of the FIFO buffers. The DPN needs to consume one value of each input and produce one value on each output port for each statement of the sequential program. All nodes, except for load/store nodes, behave like the nodes in a Kahn process network this means writing to a buffer is non-blocking and succeeds immediately but reading from a buffer is blocking and the process waits until all necessary input values are available. This automatically makes the nodes behave deterministically because inputs are always mapped to the same outputs without concern for the execution schedule. Furthermore, ensuring the nodes also do not have a state and implement a sequential function without any side-effects makes the whole DPN deterministic [Sch21].

DPNs are considered with a suitable set of nodes, see Table 3.2, that is sufficient for Turing completeness. This set ensures the DPNs are powerful enough so they can be translated from sequential programs to this kind of DPN. This set explains the purpose of different nodes on such an exposed datapath architecture. DPNs used in buffered exposed datapath architectures should be levelled. In levelled DPNs a copy node is added on each edge that would otherwise pass through a level. For example, a node is scheduled on level 3 and its successor is scheduled on level 5, in a levelled DPN there must be a copy node on level 4 as intermediate [SBR22a]. In case of this thesis, it is very convenient for creating the benchmark tables 5.1, 5.2 and 5.3. Since each copy node uses one PU it makes it easier to count the required amount of PUs for any DPN.

syntax	semantics
token control	
(y) := C(x)	t = get(x) push(t, y)
(y_0, y_1):= S(x, x)	t=get(x) push(t, y_0) push(t, y_1)
(y) := J(x_0, x_1)	t_0=get(x_0) t_1=get(x_1) push(t_0, y)
():= K(x)	t=get(x)
control flow	
(y) = SEL(c, x_1, x_0)	t_c=get(c) td=if(t_c)then get(x_1) else get(x_0) push(td, y)
(y_1, y_0)=SWT(c,x)	t_c=get(c) t_x=get(x) if(t_c) then push(t_x, y_1) else push(t_x, y_0)
memory access	
(tk_{out}, y)=LD$_a(adr, tk_{in})$	ta=get(adr) t_m=get(tk_{in}) push(mem[a,t_a],y) push(t_m, tk_{out})
(tk_{out})=ST$_a(adr, tk_{in}, x)$	t_a=get(adr) t_m=get(tk_{in}) t_x=get(x) mem[a,t_a]:=t_x push(t_m, tk_{out})
data operations	
(y)=Const(c)	push(c, y)
(y)=MonOp(f, x)	t_x=get(x) push$(f(t_x), y)$
(y)=BinOp$(\odot, x_0, x_1)$	t_0=get(x_0) t_1=get(x_1) push$(t_0 \odot t_1, y)$
(y)= ITE(c, x_1, x_0)	t_c=get(c) t_1=get(x_1) t_0=get(x_0) if(t_c) then push(t_1, y) else push(t_0, y)

Table 3.1: *Syntax and Semantics of used DPN Nodes [Sch21]*

4 Scheduling

4.1 Introduction

Any given DPN can be scheduled in many ways. Scheduling is not a novel problem. In high-level synthesis scheduling, it is important to construct faster and cheaper micro architectures. The question arises whether the traditional scheduling algorithms used in high-level synthesis perform as well and are as useful when applied to exposed datapath architecture scheduling.
To try and answer this question, we examine three well known scheduling algorithms in this work. They will be applied to an assortment of DPNs without any constraints regarding available PUs.

4.2 As-Soon-As-Possible (ASAP)

The As-Soon-As-Possible (ASAP) scheduling is a relatively simple scheduling algorithm without any regard for resource usage. It first schedules all the start node(s). After that it schedules all nodes which predecessors have already been scheduled [CM10]. The algorithm can be seen in the pseudo code of ASAP in 1. Line 2 prepares the nodes that are ready to be scheduled. This means all nodes that do not have any predecessors, all being start nodes, are added to *nodesReady* and simultaneously are removed from *nodes*. Then the algorithm runs for as long as there are unscheduled nodes. In line 4 all nodes which predecessors have already been scheduled are added to *nodesReady*. Then all nodes in *nodesReady* get scheduled. The $schedule(level, node)$ function saves which nodes are scheduled on which level. With that information it is possible to build a scheduled DPN. In the last line all nodes in *nodesReady* are removed because they have all been scheduled in the for-loop.
Since the size of *nodesReady* is not restricted in any way, the amount of PUs used by this algorithm is also not restricted and can get very high.

Algorithm 1 ASAP scheduling

```
1: nodes ← allNodes
2: nodesReady ← nodes.Where(predecessors = empty)
3: while nodes not Empty do
4:     nodesReady ← nodes.Where(predecessors = scheduled)
5:     level = level +1
6:     for node in nodesReady do
7:         schedule(level, node)
8:     nodesReady.removeAll()
```

4.3 As-Late-As-Possible (ALAP)

The As-Late-As-Possible (ALAP) scheduling is the inverse of ASAP. Starting from the end node(s), it schedules all nodes whose successors have been scheduled. ALAP also isn't resource restricted nor does it try to optimize latency or resource usage [CM10].
The algorithm can also be seen as pseudo code in 2. First, all end nodes, nodes without successors, are added to *nodesReady* while being removed from the set of all nodes. Then the schedule actually starts. As long as there are unscheduled nodes the schedule will run. In line 4 all nodes which successors have been scheduled are added to *nodesReady*. Then all nodes in *nodesReady* get scheduled. With this information again, it is possible to build the scheduled DPN. Then all the nodes in *nodesReady* get removed because they have been scheduled.
When actually building the scheduled DPN it is essential to know if ASAP or ALAP was used as scheduler because in this implementation the levels are reversed. A low level in ASAP means, the node is close to the start nodes but a low level in ALAP means the node is close to an output.

Algorithm 2 ALAP scheduling

```
1: nodes ← allNodes
2: nodesReady ← nodes.Where(successors = empty)
3: while nodes not Empty do
4:     nodesReady ← nodes.Where(successors=scheduled)
5:     the levels are reversed, low level means later scheduled
6:     level = level +1
7:     for node in nodesReady do
8:         schedule(level, node)
9:     nodesReady.removeAll()
```

4.4 Hu's Algorithm

Hu's algorithm works differently than ALAP or ASAP. It's a multi-step process. Each node gets an index α which describes the length of the longest path from an output to the node. For that purpose a virtual sink needs to be added in the implementation if there is more than one output. The virtual sink is the starting point for the breadth-first search used for indexing. Usually in a breadth-first search a node that has already been visited is ignored. In this case this could result in wrong indices. Therefore an already visited node's index i is compared to what the current path's index i_{new} of that node would be. If $i < i_{new}$ the node's index is updated to $i = i_{new}$. The node's predecessors are updated by continuing the breadth-first search from that node.

The algorithm can be seen as pseudo code in 3. The indexing is required for this schedule to work. Hu's algorithm requires a set amount of PUs

as it is a resource aware scheduling algorithm. The first lines of the algorithm are the same as for ASAP. All nodes without predecessors are added to *nodesReady* while being removed from *nodes*. As long as there are unscheduled nodes, either in *nodes* or *nodesReady* the scheduler will continue to run. First all nodes which predecessors have been scheduled are added to *nodesReady*. Then, in line 7 the difference to ASAP can be seen. Instead of scheduling all nodes that are ready to be scheduled, there will be maximum $\#PUs$, the amount of available PUs, nodes be scheduled. It is possible less than $\#PUs$ nodes get scheduled. In that case all nodes in *nodesReady* get scheduled. The nodes in *nodesReady* with the maximum α will get scheduled. The $schedule(level, node)$ function does the same as in the ASAP and ALAP algorithms. It saves the level to the corresponding node. The level in Hu is the same as ASAP in that it describes the level as distance to the start nodes of the schedule. After the node is scheduled, it is removed from *nodesReady*. This is repeated until all nodes are scheduled [Hu61].

Hu's algorithm is resource aware. In contrast to ASAP it needs a set number of available PUs to work. Since there are no constraints on the available PUs given, an appropriate amount of processing units needs to be determined. This determination only works for the constraints given by the exposed datapath architecture. An initial number of PUs ($\#PUs = 1$) was set. This is the difference between using Hu's algorithm in high level synthesis and exposed datapath architecture. In high level synthesis a DPN can be scheduled with one PU since the intermediate result is not communicated directly between the PUs. When scheduling for an exposed datapath architecture the schedule can get stuck. In exposed datapath architectures a PU is unavailable for as long as the executed node has unscheduled successors. It can therefore happen all PUs are busy waiting for their successors to be scheduled and the scheduler is either stuck or running in an endless loop. To alleviate that problem a new PU is dynamically added during the schedule. The scheduler then continues to run with one PU more. When the schedule is done, the scheduler is run a second time with the final amount of PUs.

Algorithm 3 Hu's Algorithm

Require: node.α the length of longest path from the sink to the node

```
 1: procedure HU(#PUs)
 2:     nodes ← allNodes
 3:     nodesReady ← nodes.Where(predecessors=empty)
 4:     while nodes not Empty or nodesReady not Empty do
 5:         nodesReady ← nodes.Where(predecessors=scheduled)
 6:         level = level +1
 7:         for i=0; i<#PUs do
 8:             node ← nodesReady.maximum(α)
 9:             schedule(level, node)
10:             nodesReady = nodesReady \ node
```

5 Evaluation

5.1 Observations

In the Embedded Systems group at the University of Kaiserslautern, there is an assortment of DPNs used for testing and benchmarking exposed datapath architectures. In 5.1, 5.2 and5.3 the results of scheduling the DPNs with either of the three introduced scheduling algorithms can be seen. Looking at 5.1, the numbers in the charts were determined as follows:

- **Nodes**: The total number of nodes are all nodes excluding input and output nodes, as seen in the green triangles.
- The number of PUs, **#PUs**, is determined by looking at the widest part of the scheduled DPN, in this case at the third last level are 4 parallel nodes which each need a PU, excluding input and output nodes and constants since neither actually need a PU. Input and output nodes and constants are data dependencies but are not executed. Hence, the total number of PUs needed to execute this specific schedule is 4.
- **Length** describes the amount of levels from top to bottom, again excluding the input nodes and constants. In this example the length is 6.
- **#Copy** describes the number of copy nodes, here we have 6.
- **%Copy** describes the percentage of the copy nodes in relation to the total amount of nodes without the input and output nodes. This is an interesting metric because it shows how efficiently the PUs are being utilised. Since constants are not excluded the percentage is not exact but since all schedules are evaluated the same way they are still comparable.

5.1.1 General observations

Looking at the benchmarks, it stands out that schedules with ASAP and ALAP *always* have the same length. This is easily explained by one shared property of both algorithms. Both resulting schedules are as long as the critical path of the DPN. The critical path of a DPN is the longest path between the first and the last node. ASAP and ALAP both immediately schedule all nodes that are ready. This results in the shortest schedule possible. If there was a shorter schedule it would mean during the scheduling a node that was ready was not scheduled. That is impossible by design.
One such critical path can be seen in 5.1, for example. The critical path here

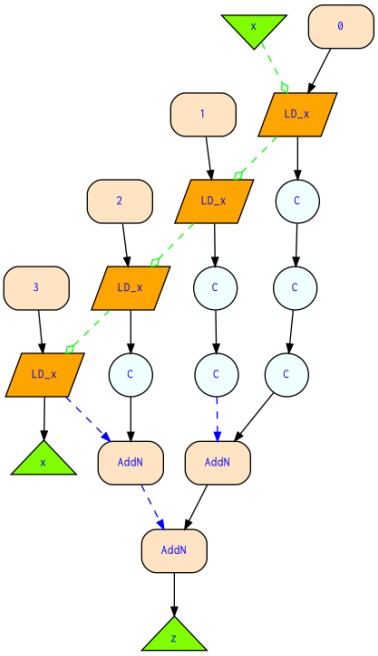

Figure 5.1: *Binary Tree Arr 4 scheduled with ALAP*

is $0 \rightarrow LD\_x \rightarrow LD\_x \rightarrow LD\_x \rightarrow LD\_x \rightarrow AddN \rightarrow AddN \rightarrow z$. This is the only path without any copy nodes.

5.1.2 Binary Tree

In the example BinaryTree_Arr_n there are n values stored in an array. These values are subsequentially loaded from the array and then added in a tree structure. In figure 5.1, it can be seen what an array with 4 entries scheduled with ALAP looks like. From this example one can already see how ALAP is the worst choice for this kind of DPN. Because it schedules the "AddN" nodes at the last possible level, it inevitably leads to more and longer copy node chains for larger DPNs of this structure. This is also reflected by the percentage of copy nodes for the array with $n = 64$ where copy nodes make up over 90% of all nodes in total. ASAP and Hu's algorithm perform exactly equally for this example. They each need one new node for each time the length of the array doubles. The length of the schedules grows slightly over linearly with the length of the array.

When the values are not stored in an array but given directly the resulting schedules look differently. Looking at BinaryTree_Scl_n, ASAP and ALAP result in identical schedules. They always need half the inputs' amount of PUs and take one cycle longer for each doubling in inputs. There are no copy nodes so the PU utilisation is optimal. Here we can also see that either path from a start node to the sink is critical. If that were not the case either ALAP or

ASAP would have copy nodes. Hu's algorithm does need one additional cycle but with more than $n = 2$ inputs it needs 1 PU less. This of course introduces copy nodes to the schedule but they do not make up a significant portion of the nodes. The percentage of copy nodes even decreases to less than 7% for $n = 64$ inputs.
If the goal is to minimise on PUs instead of cycles Hu's algorithm outperforms ASAP and ALAP.

5.1.3 Matrix Multiplication

Both matrix multiplications multiply two $n \times n$ matrices and store the result in a new matrix. The MatrixSimple_ArrGbl_n directly multiplies the values loaded from the matrices where MatrixMultCannon_ArrLoc_n loads the values one by one and needs to duplicate them at least once to compute the matrix multiplication.

In the simple multiplication benchmark, ASAP and Hu's algorithm perform very similarly. They both need one additional PU for each doubling of dimensions. The length of the schedule roughly doubles with each additional dimension. Hu's algorithm always takes one extra cycle compared to ASAP. ALAP however uses substantially more PUs. It uses 200 where ASAP uses 6 for the same amount of cycles. It always takes more than n^3 PUs. This is again underlined by the over 94% of copy nodes.

Looking at the common matrix multiplication, ALAP for the first time outperforms Hu and ASAP. ALAP takes less than half the amount of PUs for matrices with $n = 5$. It also has a remarkably low percentage of copy nodes that slowly rises. Compared to the increasing percentage of copy nodes of all other schedules this increase is minimal. ASAP and Hu's algorithm produce identical results for the statistic. Both their PU requirements grow more than linearly with respect to their dimensions.
The length of the schedule can be described by $10 + \sum_{x=3}^{x} 2x$ where $x \geq 3$ describes the dimension of the matrix.

5.1.4 Parallel Prefix Tree

In the parallel prefix tree benchmark, the parallel prefix sums are computed in a tree structure.

In ParallelPrefixTree_SclLoc_n it can be seen once again that ASAP and Hu's scheduling algorithm perform identically with respect to the measured results. They each use $n - 1$ PUs and take $1 + (log_2(n) - 1) * 4$ cycles where $n \geq 4$. ALAP again uses more PUs but does not need many more copy nodes, i.e. for $n = 64$ ASAP uses 63 PUs and 74.60% copy nodes, ALAP uses 66 PUs and 75.54% copy nodes.

In ParallelPrefixTree_SclGlb_n no two scheduling algorithms produce the same schedule. The benchmark for $n = 4$ is most curious. All three schedules use 3 PUs. ALAP uses the least amount of nodes (8), out of which only one is a copy node. ASAP needs 12 nodes, out of which 5 are copy nodes. Hu's schedule has 9 nodes in total and therefore has 2 copy nodes. It also takes one cycle longer. This is surprising as it is the only time there is such a stark difference in a small DPN. Especially because this trend does not continue for $n > 4$.
ASAP consistently uses $n - 1$ PUs and take $4 + (log_2(n) - 2) * 6$ cycles where $n \geq 4$. ALAP uses more PUs and uses more copy nodes than in the SclLoc but it is still less of a difference than in other benchmarks where ALAP does not outperform ASAP. Hu's schedule needs 2 to 3 PUs less than ASAP for $n > 8$ but does take multiple cycles more.

5.1.5 Fast Fourier Transform

In the Fast Fourier Transform benchmark, the DPN computes the Fast Fourier Transform for an array with the length n and stores the result in a different array.
The FastFourierTransform_ArrGbl_n directly uses the values loaded from the array where FastFourierTransform_ArrLoc_n loads the values one by one and cannot use them instantly.

For the FastFourierTransform_ArrGbl_n ASAP and Hu's schedule do not differ much. They require the same amount of PUs, which is roughly $2n$. They take the same amount of cycles which is approximately $4n$. The only difference is that Hu's schedule consistently needs less copy nodes. ALAP again produces interesting results in that it needs significantly more PUs but the schedule actually has fewer nodes. The required amount of PUs does not seem to follow an obvious pattern but it needs up to 32 more.

Looking at FastFourierTransform_ArrLoc_n, Hu's schedule and ASAP do not differ greatly from the FastFourierTransform_ArrGbl_n. They use more PUs, for $n = 32$ almost $3n \rightarrow 90$ for *ArrLoc* opposed to little over $2n \rightarrow 69$ for *ArrGbl* but take fewer cycles which is roughly $2n$. Hu's schedule and ASAP perform identically. However ALAP needs less PUs and far less nodes for $n = 32$. ALAP requires less than $2n$ PUs.

5.1.6 RadixB Multiplication

The RadixB multiplication multiplies two RadixB numbers which digits are stored in two arrays of the length n. The result is stored in a different array.
The RadixBMulDadda_ArrGbl_n directly uses the values loaded from the array where RadixBMulDadda_ArrLoc_n loads the values one by one and cannot use them instantly.

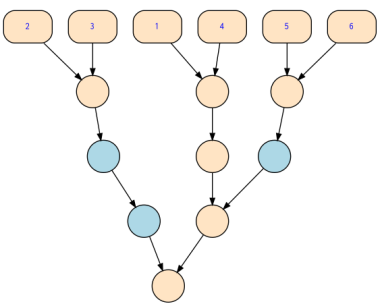

Figure 5.2: *Example Schedule with ASAP*

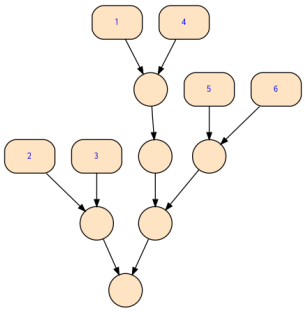

Figure 5.3: *Example Schedule with ALAP*

ASAP and Hu's resulting schedule of RadixBMulDadda_ArrGbl_n is identical. They take slightly more than linear ($> 11n$) cycles. The required amount of PUs rises slowly and stays constant for $5 \leq n \leq 7$. ALAP once again uses more PUs and more nodes with almost as many copy nodes as Hu and ASAP use in total for $n > 5$.

ASAP's and Hu's schedule of RadixBMulDadda_ArrLoc_n is again identical. It stands out that for $n = 6$ the schedule takes two cycles longer than for $n = 7$. Apart from that they require more than n^2 PUs and complete the computation in also more than n^2 cycles but less than the required amount of PUs. ALAP for every tested length n uses more PUs and almost double the amount of copy nodes. This can be seen most clearly for $n = 7$. ALAP's schedule uses 12 PUs more, where for smaller n it used 1 to 2 PUs more.

5.1.7 Comparison of ASAP, ALAP and Hu's algorithm

Generally speaking about the different algorithms on the basis of the benchmarks, we can see that each algorithm has its advantages. ASAP takes the least possible cycles but is not always optimal when it comes to the required amount of PUs.

ALAP works well for DPNs with a similar structure to what can be seen in figure 5.3. ASAP and Hu's algorithm respectively can not perform as well as visualised in 5.2. The problem is when the structure differs too much ALAP tends to require unfavourably large amounts of PUs. The worst case in the benchmarks for this behaviour is the BinaryTree_Arr_64 where it requires 64 PUs. ASAP and ALAP only need 7 PUs. This difference between the amount of used PUs explains why in ALAP the copy nodes make up more than 90% of the total amount of nodes.

Hu's algorithm performs generally well. If the number of used PUs is as high as the one used by ASAP its resulting schedule is (almost) identical. It may take a cycle longer (MatrixSimple_ArrGlb). It does have a few issues that

will be covered in the following section 5.2.

5.2 Difficulties with Hu's algorithm

Hu's algorithm was not developed for buffered exposed datapath architectures. It was designed for assembly line workers doing their tasks. After a task is done, the worker is free and the result of that work does not interfere with the future work of any other worker as long as the work is done. Under these circumstances Hu achieves optimum or near optimum results [Hu61]. For conventional architectures this is not a problem because they store the results in the registers. But this is not the case for buffered exposed datapath architectures. This introduces a new constraint in that copy nodes "block" a PU after the computation is done until the result is needed.

This poses a problem on top of finding a feasible resource constraint: the number of PUs. It is possible to start scheduling and as soon as there is no PU available at the beginning of a new level, a new PU is added. Once every node is executed, the DPN get scheduled again with the correct number of PUs from the start. This, sadly, does not guarantee a minimal number of PUs as changes at the beginning of the schedule can interfere with the scheduling of later levels. One such example can be seen in 5.5. While determining the amount of PUs, a third PU is added on the third level. The problem is that when the schedule is rerun with 3 given PUs, the schedule will actually need 4 because it schedules the "0" in the first level. Of course it is possible to simply not re-run the scheduler but that results in a less efficient schedule since it takes more cycles. The BinaryTree_Scl_8 for example without the rerun needs 4 PUs, the same as with the rerun, but takes 7 instead of 4 cycles.

The visualisation of the consequences of scheduling nodes early also showcases another problem. Even if the minimum number of PUs is found scheduling with Hu's algorithm can not guarantee a schedule with that amount of PUs. This is by design not possible because Hu always tries to schedule as many processes as it has available PUs. And that will lead to schedules with similar problems as in the example 5.5. This is especially true when an algorithm like ALAP uses the smallest amount of PUs.

5.3 Alternative Implementations of Hu's algorithm

In the course of running the benchmarks a series of challenges concerning the implementation of Hu's algorithm were encountered. The first and main problem is determining the amount of PUs. The first idea relating to that was some sort of binary search with an upper bound. This requires some approach to compute a suitable upper bound for a given DPN. Deciding on the amount of PUs used by ASAP as an upper bound the next problem arose. Runtime for the scheduling process itself. First scheduling with ASAP then binary-search scheduling with Hu turns out to be very inefficient in regards to runtime.

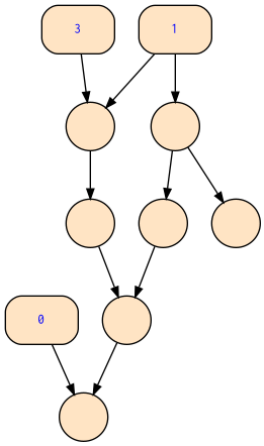

Figure 5.4: *Determining PUs*

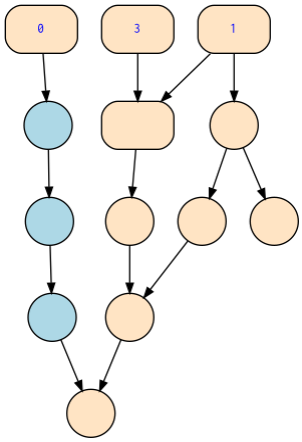

Figure 5.5: *Rerun scheduler*

Therefore, this idea was discarded before it was even properly implemented.

The next approach tried to make Hu's algorithm aware of the PUs it uses. So when running the algorithm it has initially 1 PU. The algorithm keeps track of which processes are still necessary as not all output-buffers were scheduled. Whenever the algorithm gets stuck because it does not have any free PUs a new PU is added. This was actually implemented and worked fine. The problem here however is that the schedules tend to get very long because at the beginning the schedule is very narrow due to the lack of available PUs.

This is where the idea of rerunning the scheduler stems from. Rerunning the scheduler may appear simple but it creates two problems. Firstly, new copy-node chains need additional PUs, as can be seen in the example 5.5. And secondly, a PU blocked by a copy node is not per-se unavailable for scheduling. It is actually available for scheduling for a binary operation like an *Add*. This can even free up one PU if both input nodes are blocking PUs, an example for that would be the most right *Add* in 5.1. It has two input nodes that are copy nodes. If the *Add* were to be scheduled one cycle earlier, the schedule would need one PU less in total. Of course the example was scheduled with ALAP so that is completely intentional.

This problem was solved by explicitly implementing that binary operations should be scheduled right at the beginning of scheduling a level if both inputs were scheduled regardless of the amount of available PUs. Doing that at the beginning of scheduling a level ensures that the input nodes have been executed. Also, the freed PU can be used for scheduling that same level. This helped with the length of scheduled.

This leaves one remaining issue. Either something in the determination process did not work out or in the schedule something was not working. Gen-

erally the schedules were long, quite a bit longer than their ASAP counterpart but they sometimes used less PUs. The problem however occurred every time when Hu's schedule needed **more** PUs than ASAP **and** needed more cycles.

This lead to the current implementation which is the best balance implemented. Currently, determining the required amount of PUs and the scheduling are separate. The number of PUs is determined by the already described way; the first scheduler is aware of the PUs blocked by copy nodes and adds a new PU when the schedule gets stuck. Before doing the second pass of the scheduler the number of PUs gets decreased by 2 simply because that yielded better results in the tests. The actual scheduling is done by applying the basic scheduling algorithm unaware of any PUs. This means, it is unclear how many PUs the schedule needs up until the schedule is done. However, the produced results are the most comparable to the other scheduling algorithms in number of PUs used as well as the length. The main advantage of scheduling this way is that if the amount of PUs is set too high the algorithm will perform the same as ASAP would which more often than not is better than ALAP.

Benchmark	#PUs	Length	Nodes	#Copy	%Copy
BinaryTree_Arr_4	3	6	15	4	26,67
BinaryTree_Arr_8	4	11	23	12	52,17
BinaryTree_Arr_16	5	20	47	31	65,96
BinaryTree_Arr_32	6	37	95	79	83,16
BinaryTree_Arr_64	7	70	382	191	50,00
BinaryTree_Scl_4	2	2	3	0	0,00
BinaryTree_Scl_8	4	3	7	0	0,00
BinaryTree_Scl_16	8	4	15	0	0,00
BinaryTree_Scl_32	16	5	31	0	0,00
BinaryTree_Scl_64	32	6	63	0	0,00
MatrixSimple_ArrGlb_2	5	10	52	6	11,54
MatrixSimple_ArrGlb_3	5	29	195	24	12,31
MatrixSimple_ArrGlb_4	6	68	508	108	21,26
MatrixSimple_ArrGlb_5	6	128	996	221	22,19
MatrixMultCannon_ArrLoc_2	8	10	56	8	14,29
MatrixMultCannon_ArrLoc_3	21	16	186	39	20,97
MatrixMultCannon_ArrLoc_4	43	24	492	164	33,33
MatrixMultCannon_ArrLoc_5	63	34	1095	480	43,84
ParallelPrefixTree_SclLoc_4	3	5	12	4	33,33
ParallelPrefixTree_SclLoc_8	7	9	43	21	48,84
ParallelPrefixTree_SclLoc_16	15	13	134	82	61,19
ParallelPrefixTree_SclLoc_32	31	17	367	253	68,94
ParallelPrefixTree_SclLoc_64	63	21	945	705	74,60
ParallelPrefixTree_SclGlb_4	3	4	12	5	41,67
ParallelPrefixTree_SclGlb_8	7	10	45	24	53,33
ParallelPrefixTree_SclGlb_16	15	16	159	108	67,92
ParallelPrefixTree_SclGlb_32	31	22	437	324	74,14
ParallelPrefixTree_SclGlb_64	63	28	1167	928	79,52
FastFourierTransform_ArrGlb_4	7	18	88	41	46,59
FastFourierTransform_ArrGlb_8	13	36	354	235	66,38
FastFourierTransform_ArrGlb_16	30	70	1426	1135	79,59
FastFourierTransform_ArrGlb_32	69	136	5702	5011	87,88
FastFourierTransform_ArrLoc_4	7	12	58	23	39,66
FastFourierTransform_ArrLoc_8	15	22	212	119	56,13
FastFourierTransform_ArrLoc_16	38	42	792	559	70,58
FastFourierTransform_ArrLoc_32	90	74	3092	2531	81,86
RadixBMulDadda_ArrGlb_4	23	46	699	273	39,06
RadixBMulDadda_ArrGlb_5	37	55	1077	409	37,98
RadixBMulDadda_ArrGlb_6	37	76	1744	778	44,61
RadixBMulDadda_ArrGlb_7	37	81	2210	920	41,63
RadixBMulDadda_ArrLoc_4	24	39	607	205	33,77
RadixBMulDadda_ArrLoc_5	35	43	916	288	31,44
RadixBMulDadda_ArrLoc_6	47	55	1406	500	35,56
RadixBMulDadda_ArrLoc_7	59	53	1746	540	30,93

Table 5.1: *Benchmarks of ASAP scheduling*

Benchmark	#PUs	Length	Nodes	#Copy	%Copy
BinaryTree_Arr_4	4	6	17	6	35,29
BinaryTree_Arr_8	8	11	51	28	54,90
BinaryTree_Arr_16	16	20	166	119	71,69
BinaryTree_Arr_32	32	37	590	495	83,90
BinaryTree_Arr_64	64	70	2206	2015	91,34
BinaryTree_Scl_4	2	2	3	0	0,00
BinaryTree_Scl_8	4	3	7	0	0,00
BinaryTree_Scl_16	8	4	15	0	0,00
BinaryTree_Scl_32	16	5	31	0	0,00
BinaryTree_Scl_64	32	6	63	0	0,00
MatrixSimple_ArrGlb_2	10	10	83	31	37,35
MatrixSimple_ArrGlb_3	36	29	620	449	72,42
MatrixSimple_ArrGlb_4	98	68	3471	3071	88,48
MatrixSimple_ArrGlb_5	200	128	13125	12350	94,10
MatrixMultCannon_ArrLoc_2	8	10	57	9	15,79
MatrixMultCannon_ArrLoc_3	19	16	180	33	18,33
MatrixMultCannon_ArrLoc_4	24	24	404	76	18,81
MatrixMultCannon_ArrLoc_5	31	34	760	145	19,08
ParallelPrefixTree_SclLoc_4	4	5	10	2	20,00
ParallelPrefixTree_SclLoc_8	9	9	43	21	48,84
ParallelPrefixTree_SclLoc_16	17	13	136	84	61,76
ParallelPrefixTree_SclLoc_32	34	17	383	268	69,97
ParallelPrefixTree_SclLoc_64	66	21	981	741	75,54
ParallelPrefixTree_SclGlb_4	3	4	8	1	12,50
ParallelPrefixTree_SclGlb_8	8	10	46	25	54,35
ParallelPrefixTree_SclGlb_16	16	16	160	109	68,13
ParallelPrefixTree_SclGlb_32	34	22	463	350	75,59
ParallelPrefixTree_SclGlb_64	66	28	1208	969	80,22
FastFourierTransform_ArrGlb_4	8	18	104	57	54,81
FastFourierTransform_ArrGlb_8	19	36	362	243	67,13
FastFourierTransform_ArrGlb_16	42	70	1347	1056	78,40
FastFourierTransform_ArrGlb_32	101	136	5507	4816	87,45
FastFourierTransform_ArrLoc_4	6	12	51	16	31,37
FastFourierTransform_ArrLoc_8	12	22	172	79	45,93
FastFourierTransform_ArrLoc_16	28	42	601	368	61,23
FastFourierTransform_ArrLoc_32	62	74	2206	1645	74,57
RadixBMulDadda_ArrGlb_4	23	46	873	447	51,20
RadixBMulDadda_ArrGlb_5	37	55	1207	539	44,66
RadixBMulDadda_ArrGlb_6	57	76	2610	1644	62,99
RadixBMulDadda_ArrGlb_7	67	81	3399	2109	62,05
RadixBMulDadda_ArrLoc_4	26	39	726	324	44,63
RadixBMulDadda_ArrLoc_5	36	43	1070	442	41,31
RadixBMulDadda_ArrLoc_6	48	55	1830	924	50,49
RadixBMulDadda_ArrLoc_7	71	53	2133	927	43,46

Table 5.2: *Benchmarks of ALAP scheduling*

Benchmark	#PUs	Length	Nodes	#Copy	%Copy
BinaryTree_Arr_4	3	6	15	4	26,67
BinaryTree_Arr_8	4	11	23	12	52,17
BinaryTree_Arr_16	5	20	47	31	65,96
BinaryTree_Arr_32	6	37	95	79	83,16
BinaryTree_Arr_64	7	70	191	382	200,00
BinaryTree_Scl_4	2	3	4	1	25,00
BinaryTree_Scl_8	3	4	9	2	22,22
BinaryTree_Scl_16	7	5	18	3	16,67
BinaryTree_Scl_32	15	6	35	4	11,43
BinaryTree_Scl_64	31	7	68	5	7,35
MatrixSimple_ArrGlb_2	5	11	59	7	11,86
MatrixSimple_ArrGlb_3	5	30	217	46	21,20
MatrixSimple_ArrGlb_4	6	69	509	109	21,41
MatrixSimple_ArrGlb_5	6	129	996	221	22,19
MatrixMultCannon_ArrLoc_2	8	10	56	8	14,29
MatrixMultCannon_ArrLoc_3	21	16	186	39	20,97
MatrixMultCannon_ArrLoc_4	43	24	492	164	33,33
MatrixMultCannon_ArrLoc_5	63	34	1095	480	43,84
ParallelPrefixTree_SclLoc_4	3	5	12	4	33,33
ParallelPrefixTree_SclLoc_8	7	9	44	22	50,00
ParallelPrefixTree_SclLoc_16	15	13	134	82	61,19
ParallelPrefixTree_SclLoc_32	31	17	391	277	70,84
ParallelPrefixTree_SclLoc_64	63	21	1066	826	77,49
ParallelPrefixTree_SclGlb_4	3	5	9	2	22,22
ParallelPrefixTree_SclGlb_8	7	11	47	26	55,32
ParallelPrefixTree_SclGlb_16	13	19	153	102	66,67
ParallelPrefixTree_SclGlb_32	29	28	449	336	74,83
ParallelPrefixTree_SclGlb_64	60	37	1195	956	80,00
FastFourierTransform_ArrGlb_4	7	18	87	40	45,98
FastFourierTransform_ArrGlb_8	13	36	349	230	65,90
FastFourierTransform_ArrGlb_16	30	70	1409	1118	79,35
FastFourierTransform_ArrGlb_32	69	136	5653	4962	87,78
FastFourierTransform_ArrLoc_4	7	12	58	23	39,66
FastFourierTransform_ArrLoc_8	15	22	212	119	56,13
FastFourierTransform_ArrLoc_16	38	42	792	559	70,58
FastFourierTransform_ArrLoc_32	90	74	3092	2531	81,86
RadixBMulDadda_ArrGlb_4	23	46	699	273	39,06
RadixBMulDadda_ArrGlb_5	37	55	1077	409	37,98
RadixBMulDadda_ArrGlb_6	37	76	1744	778	44,61
RadixBMulDadda_ArrGlb_7	37	81	2210	920	41,63
RadixBMulDadda_ArrLoc_4	24	39	607	205	33,77
RadixBMulDadda_ArrLoc_5	35	43	916	288	31,44
RadixBMulDadda_ArrLoc_6	47	55	1406	500	35,56
RadixBMulDadda_ArrLoc_7	59	53	1746	540	30,93

Table 5.3: *Benchmarks of Hu's scheduling algorithm*

6 Conclusion

In this thesis I took a closer look at instruction scheduling in exposed datapath architectures, more specifically in buffered exposed datapath architectures as they can be modelled by DPNs. This problem appears simple on the surface since it is well understood and generally considered solved for high-level synthesis. But there is a big difference between scheduling for conventional architectures and exposed datapath architectures, namely the copy nodes that need a PU where conventional architectures store intermediate results in a register instead of blocking a PU until its successors are scheduled.

Despite its challenges and sometimes less than optimal schedule lengths, Hu's performance is better over all or at least as good as the currently used ASAP scheduler. In the cases where ALAP uses the least amount of PUs, a different way of determining the required PUs needs to be developed as the presented way cannot work for this kind of schedule as discussed in section 5.2. The most obvious way would be to analyse the structure of the program beforehand and choose a scheduler on that result. I do not expect this to be feasible or easier than trying to find a better suited scheduling algorithm for exposed datapath architectures.

This thesis highlights one big flaw of traditional scheduling algorithms in regards to instruction scheduling for exposed datapath architectures. There is no penalty for scheduling nodes too early. This is completely fine for conventional architectures but in exposed datapath architecture scheduling too early results in copy node-chains.

7 Further Research

As stated in chapter 6 traditional scheduling algorithms are not ideal for exposed datapath architectures. There should definitely be more research done in regards to instruction scheduling. There are multiple approaches that could be taken. The first is trying to optimise the traditional scheduling algorithms to the new constraints and rules of exposed datapath architectures. The second approach is to view scheduling as a graph-theoretical problem. For instance if the scheduling algorithm were to try and minimise the level difference in nodes that are predecessors of one node would that have a measurable impact on the schedule (for a set amount of PUs)? Also, is a schedule for a set amount of PUs always better the less copy nodes it has?

This scheduling problem becomes even more complex when duplicator nodes are considered. Duplicator nodes can be moved upwards by doubling the predecessor node as successors of the duplicator node. There may be DPNs where using this property of duplicator nodes results in better schedules than scheduling duplicator nodes at the last possible level.
Another point to scheduling is the amount of PUs. Here I see two questions; first, how to reliably determine the minimal amount of PUs possible for any given algorithm. Second, are less PUs always better? Maybe there is something like an ideal balance between the length and the number of PUs in schedules.

There are a lot of unanswered questions and there will be more as we get more answers to the presented questions. The topic of instruction scheduling for exposed datapath architecture poses a complex problem which seems to get little to no attention in current research. This thesis shows there is a lot to discover on this topic.

Bibliography

[CM10] Philippe Coussy and Adam Morawiec. *High-level synthesis*. Vol. 1. Springer, 2010.

[Gil74] KAHN Gilles. "The semantics of a simple language for parallel programming". In: *Information processing* 74 (1974), pp. 471–475.

[Hu61] T. C. Hu. "Parallel Sequencing and Assembly Line Problems". en. In: *Operations Research* 9.6 (Dec. 1961), pp. 841–848. ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.9.6.841.

[KM76] Gilles Kahn and David MacQueen. "Coroutines and networks of parallel processes". In: (1976).

[LP95] Edward A Lee and Thomas M Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.

[SBR22a] Klaus Schneider, Anoop Bhagyanath, and Julius Roob. "Code Generation Criteria for Buffered Exposed Datapath Architectures from Dataflow Graphs". In: (2022).

[SBR22b] Klaus Schneider, Anoop Bhagyanath, and Julius Roob. "Virtual Buffers for Exposed Datapath Architectures". In: *MBMV 2022; 25th Workshop*. VDE. 2022, pp. 1–11.

[Sch21] Klaus Schneider. "Translating structured sequential programs to dataflow graphs". In: *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 66–77. ISBN: 978-1-4503-9127-6. DOI: 10.1145/3487212.3487343.

[Vad+19] Kanishkan Vadivel, Roel Jordans, Sander Stujik, Henk Corporaal, Pekka Jääskeläinen, and Heikki Kultala. "Towards efficient code generation for exposed datapath architectures". In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. 2019, pp. 86–89.