Abstract—Some of the newer processor architectures are no longer based on registers in order to increase their potential of instruction-level parallelism. Instead, they expose their data paths to the compiler so that the program is able to directly move data values between function units using suitable instructions. Some of these architectures require a synchronous transfer of data values while others use asynchronous transfers by buffering values. In this paper, we discuss the out-of-order execution of function units of exposed data path architectures with asynchronous data transfers. The execution of these function units may locally deviate from the program order which is in analogy to dynamic scheduling used by processors with out-of-order execution. Since our out-of-order execution has only effects inside the function units, it requires no modifications of the compiler or instruction set. We have implemented different variants on FPGAs, and evaluated these for a set of application scenarios showing that the out-of-order extension can considerably increase the performance of these architectures.

I. INTRODUCTION

It is well-known that the performance of processors for sequential programs can be increased by executing instructions of the sequential program in parallel, thus utilizing the instruction-level parallelism (ILP) [1], [2] of sequential programs. Pipelining, dynamic [3] and static [4] scheduling have been widely used to significantly increase the performance of state-of-the-art processors. Powerful compiler techniques like trace scheduling [5] and software pipelining (also called modulo scheduling) [6], [7] can even exploit ILP that is hardly found by programmers.

However, processors and compilers face some limits for exploiting ILP [1]: One bottleneck is the limited number of registers that sooner or later force the compiler to temporarily spill out values to the main memory, even though the processor would have space for these values in local memories. Increasing the number of registers is however difficult due to the complex wiring of the register files which already forced designers to implement clustered architectures [8].

To overcome these limits, a new class of processor architectures called exposed data path architectures (XDPAs) (see Section II-A) has been introduced whose instruction sets are no longer based on registers. Instead, their function units (FUs) are exposed to the programmer/compiler, and values are sent between them as determined by the program. In general, XDPAs have a large number of function units and delegate instruction scheduling as well as data transports to the compiler. This way, XDPAs circumvent limits on ILP due to the use of registers and increase the ILP by utilizing rather a more dataflow driven execution.

However, since the function units can be viewed as specialized processor cores that execute a sequential instruction stream, typical optimizations known from sequential processor architectures can be also applied to them. In particular, dynamic scheduling of the operations inside the function units may improve the overall performance, while neither the compiler nor the existing code need any modifications.

In this paper, we consider the out-of-order execution of buffered function units as used in our SCAD architecture (see Section II-B) which is an XDPA with asynchronous data transfers among its FUs. This optimization may reschedule the operations within FUs for execution to avoid idle times of the FUs. We show that restoring the program order before transferring the results to other FUs is only necessary for data values having the same target addresses. We also discuss different implementations on FPGAs to optimize either circuit size or runtime, and to evaluate parameters like buffer sizes for concrete applications.

The paper is organized as follows: Section II discusses exposed data path architectures in general, and the SCAD architecture in particular. Section III describes the concept of our out-of-order execution. Implementation details are given in Section IV considering also partial out-of-order execution, and experimental results are shown in Section V.

II. RELATED WORK

A. Exposed Data Path Architectures (XDPAs)

The overall idea of exposed data path architectures (XDPAs) is a further evolutionary step forward from VLIW architectures as explained in [9], [10]. In addition to instruction scheduling, the compiler is also responsible for the allocation of function units (FUs) and the data transports between them. Thus, even more work has to be done by the compiler, keeping the processor architecture as simple as possible.

For example, the Raw machine [11] is a spacial architecture whose computing elements are arranged in a two-dimensional array where the compiler has to determine all communications between the elements still using registers. WaveScalar [12] and AMIDAR [13] are implementations of classic dataflow architectures where programs are dataflow graphs whose nodes can fire as soon as operands are available. TRIPS [14], [15] is a processor architecture that is based on Explicit Data Graph Execution (EDGE): The architecture offers an array of arithmetic-logic units, and instructions contain information about mapping operations to these processing elements. Once issued an instruction to the function units, the execution is...
done asynchronously following the data path configuration contained in the instruction.

Transport-triggered architectures (TTAs) [9], [10], [16] consist of a set of function units whose input and output ports are registers. Programs consist of move instructions that move data values from output registers to input registers of function units. As soon as new operands arrive, the function unit computes new outputs as a side effect of the data transfer.

B. The SCAD Architecture

A drawback of statically scheduled architectures with synchronous data transfers like TTAs is that the compiler has to know the precise latencies of the function units to transfer data values at the right point of time. This is relaxed by architectures using asynchronous transfers which buffer values until these are processed. For example, the Synchronous Control/Asynchronous Dataflow (SCAD)\(^1\) [17]–[19] architecture replaces the registers at the input and output ports of TTAs by input and output buffers so that data transfers can be initiated by the sending function units without synchronizing with the receiving ones (space has already been allocated in the target buffer due to issuing the move instruction synchronously before). Hence, SCAD follows more the idea of dataflow process networks, but programs still consist of move instructions as in case of TTA. As TTA, SCAD allows any kind of application-specific function units even with arbitrary numbers of inputs and outputs, and is therefore an interesting application-specific processor architecture.

A machine based on the SCAD paradigm is therefore built of a set of arbitrary function units (FUs), each having input and output buffers for operands and results, respectively, and an operation unit for computing an arbitrary function (see Figure 1). A restriction that applies to the computed function is that it has to consume and to produce a statically known number of values like nodes in synchronous dataflow graphs (SDF) [20].

Input and output buffers store pairs of addresses and values. In the input buffers, the addresses describe the origin of the corresponding value (src); in the output buffers, they describe the target (tgt) buffer where the corresponding value shall be sent to. In the following, addresses and values which are not yet available, are denoted by \(\perp\). Buffers behave like FIFO buffers for the addresses, which are written synchronously by a control unit. It broadcasts a move instruction \((src, tgt)\) via the move instruction bus (MIB) and both addressed buffers synchronously enqueue the corresponding address at that point of time. If one of the buffers is full, it broadcasts a signal back to the bus, so that both writes are aborted and the instruction is issued again in the next cycle. Synchronously registering the instructions ensures that each data transfer will be possible later on since the corresponding space has already been allocated in the target buffer.

A move instruction can specify a number of copies to be generated from a result value. Copy counts define how often a result will be written to an output buffer by the operation unit. Result values are sent asynchronously by the output buffers to the specified input buffers through the data transport network (DTN). The DTN can be an arbitrary network connecting all FUs. Input buffers insert an arriving value at the entry containing its source address and value \(\perp\) closest to its head. FUs start execution as soon as all input buffers contain complete values at their heads. The results are inserted in the entry closest to the head of the output buffers which contains no valid value. If there is no space, execution stalls until space becomes available due to data transfers.

The SCAD architecture allows a lot of flexibility for application-specific processors. As mentioned above, the DTN can be chosen freely and different function units with arbitrary functionality can be combined to optimize the performance for specific applications.

III. Out-Of-Order Execution of FUs

A. Motivation of Out-Of-Order Execution

SCAD is a statically scheduled processor architecture since the compiler takes care of producing a good schedule for a given machine [17]–[19] in the sense of utilizing as many as possible function units at a time. Nevertheless, we consider in this paper a dynamic out-of-order execution within single FUs of a SCAD machine. This out-of-order execution is not observable outside the FUs, and can therefore be safely added without affecting the semantics of the programs. Using the out-of-order execution, we can utilize FUs for computing result values of operations that already have available operands, even though others whose results have to be sent out first are not yet ready for execution. It is to be noted that the output buffers work as FIFO buffers per target address, i.e., we only have to keep the order of sending out values for those values that are sent to the same target address. Values to different target addresses, however, can be sent out in any order.

For example, consider the grey arrow in Figure 1: Here, all values which are required to execute the third entry in the

\(^{1}\)An instruction set simulator for universal function units is available at http://es.cs.uni-kl.de/tools/teaching/ScadSim.html.
input buffers are available, so they can be processed by the execution unit. Results are then inserted at the corresponding position in the output buffer. Note that those results can now be directly sent through the DTN, since the control flow defines no values to be sent to address $y$ prior to these results. Hence, the execution should trigger as soon as values are available in the input buffers, this way reducing the idle time of FUs, and results should be transferred as soon as they are first in the buffer for their target address.

B. Organisation of Out-Of-Order Execution

Data values sent from different output buffers to the same input buffer are reordered according to already registered move instructions of that input buffer as given by the SCAD program. Therefore, for sending values from an output buffer, it is sufficient to maintain the order of those values having the same target address.

Input buffers with out-of-order execution do not differ much from the operation of normal input buffers: Registering addresses of move instructions and storing arriving values of other FUs is done without changes. The selection of operands for execution, however, is done differently: Instead of reading the head of the buffer, the next value to be read is the one closest to the head which contains a valid entry in all ports of the FU. To determine if there is enough space in all output buffers, a reservation attempt is sent to the output buffer before firing the FU. It contains the current offset from the head of the input buffer, based on the sum of all copy counts of entries closer to the head including the copy count of the entry itself.

On a successful reservation, the output buffer responds with a tag. This tag is passed to the execution unit in addition to values and copy counts and is later used to write the results at the reserved positions. If the reservation is not successful, searching for a set of valid values is stopped until a new value arrives at the input buffer.

In the output buffers, registering target addresses of move instructions works also the same way as in the unmodified buffers. However, the correct order of the generated results of the FU has to be restored, and it has to be decided which values of the output buffer should be sent out next.

Restoring the order of the generated result values works hand in hand with the reservation process: On a reservation request from the input buffer, the output buffer adds the requested offset to an offset between in- and output tracked over time. It is essential to take this additional offset into account, since entries can move independently in buffers on read operations. If the resulting position still lies within the buffer, it is checked if there is enough space for the requested number of copies above this position. The reservation success is then signaled to the input buffer together with a reservation tag. Such a tag is also stored in the output buffer along with a reference to the insertion position. Since this position can change during execution, the reference has to be kept up-to-date all the time. As a side note, the reserved space does not need to be explicitly marked as occupied, offset tracking ensures that later reservations will always occupy unused positions. When a result arrives at the output buffer, the tag given along with the result by the execution unit is looked up in the list of pending reservations and the value and all following copies are written to the buffer starting at the position referenced by the reservation tag.

A second independent process checks for each output buffer whether a completed entry is available for sending out. To this end, it is checked if there is any other entry with the same target address closer to the head than the considered entry. If this is not the case, the value is sent through the DTN and offsets are updated. Otherwise, the next entry with valid address and value entries is checked.

IV. IMPLEMENTATION

To evaluate the algorithm, we present two different FPGA implementations in VHDL optimizing either circuit size or runtime. This section only gives a rough overview of the different buffer implementations, a more detailed description can be found in [21].

A. Buffers Without Out-of-Order Execution

The in-order buffers consist of several modules to make the design as flexible as possible. In particular, input and output buffers consist of a container unit handling communication and providing control logic. Those units further contain a number of port units equal to the number of buffers provided by the FU. They manage the physical memory which is encapsulated in an additional memory unit for each port. Input buffers have additional port units containing the copy counts.

In-order buffers manage all addresses, copy counts and values in the output buffers like a first-in-first-out (FIFO) buffer: Writes to the inputs however need random access and are handled differently: A linear search, starting at the head of the buffer, searches an entry without valid data and an address matching the origin of the value to insert it into that entry. Read operations on the inputs are done simultaneously on all ports and remove all addresses, values, and copy counts from the heads of all buffers. For the output buffers, reads only affect a single port, also removing an address and a value. Further, read operations are spread evenly across all output ports containing valid values at their heads.

B. Buffers With Out-of-Order Execution

In the following, the in-order buffers of the previous section are extended to support out-of-order execution. The focus for this implementation is to keep the circuit size close to that of the in-order buffers. Since out-of-order execution only affects the data flow, everything related to address management is kept unmodified.

In the input buffer port units, entries are not read from the head, but from a position given by the container unit. This position is the same for all ports. Further, entries are not directly removed on a read, but are only marked as read. Entries are then removed from the buffer when they reach the head. Additionally, copy count units signal the number of values leaving the buffer to the output container to update the
Cpy output buffers in the input and the last three entries from the output are already read, resulting in an offset of 2. The insert position is the sum of the offset and the absolute position in the input.

In the output buffers, all tasks concerning out-of-order execution are handled within the port units, since no synchronization between the single ports is needed. Each port unit keeps an offset to the input buffer. This offset describes for how many results, which have not yet been read, the entries that have already been removed from the head of the input buffer have generated in the corresponding output ports. The offset together with the position given during reservation defines the final insertion position (cf. example in Figure 2). Every read from the inputs increases the offsets by the corresponding copy counts, and removing entries from an output decreases the attached offset by one.

On a reservation request, the buffer checks if the final insertion position is smaller than the buffer’s size. If so, there is enough space in the buffer to write the results and the reservation is acknowledged. The responded reservation tag is given by the memory address the result has to be written to. This is possible, because read and write operations only move pointers marking head and tail of the buffer, while the relation between entry position and memory address stays intact. Results arriving from the execution unit are simply written to the addresses defined by the reservation tag.

To find an entry ready to be sent through the DTN, the port unit iterates over all entries, starting at the head, until it finds a valid value which is not yet marked as read – the output unit tracks and removes read values just like the input unit. This entry is buffered and a new search is started at the head, this time comparing the target address to the currently buffered entry. If a matching address in a non-read entry is found, address iteration is canceled and the search for a valid value continues at the buffered entry. If the address search reaches the buffered entry without such an occurrence, it is the first one in the buffer for its target address and can be transferred already. When the container reads, a new search is started.

C. Buffers With Partial Out-of-Order Execution

With this implementation, we present a different buffer design with an emphasis on optimizing the runtime. Most operations are reduced to a constant number of clock cycles at the cost of a much larger circuit size. Since this is problematic for larger buffers, we consider partial buffers, as they are called from here on. The architecture is changed in a way that input and output containers now include a reorder unit which handles all out-of-order related processes for all ports at once and stores entries in parallel accessible registers.

The circuitry needed for searching the buffer entry having the lowest position index is based on a tree-based search algorithm with a low circuit area and depth [21]. While not optimal considering circuit size, this solution is selected for implementation, since it makes the intention clear and inherently solves the problem of checking if there even exists a valid entry.

The input reorder unit continuously determines the insertion position for DTN values using this circuit, reducing insertion time to one clock cycle. The same method is used to find the next entry to issue to the execution unit. Here, all entries containing valid values in all ports are possible candidates. Further, for each entry in the copy count ports, the sum of all counts beneath the entry is calculated by a combinatorial circuit. When a set of entries is ready to be executed, a reservation request is sent to the output buffer. Since the output of the search can change any time due to writes from the DTN or extension buffer, the found positions are buffered at the time of the reservation request. On a negative reply, the reservation process is restarted from the beginning, possibly using other entries which became available in the meantime. If the reply is positive, the reorder unit waits for a read from the execution unit. When a read happens, all entries in all register files above those entries are shifted down and the reservation process is started again. The input buffer extension port units work very similar to the simple port units, the main difference being that the source address is also available at the output. MIB messages are written to the reorder unit if it is not full and the extension buffer is empty, otherwise the write is applied to the extension buffer. Further, if the reorder unit is not full and the corresponding extension buffer contains an entry at its head, with or without a valid value, this entry is moved to the reorder unit.

In the output reorder unit, in contrast to the out-of-order buffers, a separate offset for each port and entry has to be stored. This is because entries are directly removed from the input buffer and therefore the remaining entries can move independently. The reorder unit concurrently searches entries

![Table 2](https://via.placeholder.com/150)

Fig. 2. Example for offset calculation and usage. Entries a and b in the input and the last three entries from the output are already read, resulting in an offset of 2. The insert position is the sum of the offset and the absolute position in the input.
ready to be sent through the DTN. In the first part of the circuit, an intermediate signal is generated, which indicates entries having the first occurrence of their target address relative to the head of the buffer of their port. Entries marked by the intermediate signal, which also have a valid value, are possible candidates for a DTN transfer.

Upon a reservation request, availability of space is checked by adding the offsets stored for the requested positions to the positions themselves and the requested result sizes for all ports. If none of the sums exceeds the buffer size, a reservation is possible. An arbitrary unused tag from a predefined pool of tags is selected and stored along with the insert position. Since offsets describe the state in the input buffer and are therefore not directly related to the entries in the output, all offsets above and including a requested position are shifted down by the copy count of the result, representing the removal of the entries in the input buffers. When results arrive at the reorder unit, the insertion position stored with the tag is used. When an entry is read, to represent changes in contrast to the input buffer, all offsets which would redirect an insertion to or above the read entry, i.e., \( position + offset \geq read \) position, are decreased. Further, all entries in the read port which are above the read entry are shifted down by one position and pending reservations are updated accordingly.

Extension buffers in the output container need special units to store incoming target addresses as well as completed entries. Therefore, a port extension unit is used which has two separate address memories. If the reorder unit signals a valid entry at its output and the corresponding part of the extension buffer is not full, this entry is moved from the reorder unit to the extension unit. DTN reads are distributed across the extension units like in the simple and partial buffers.

V. EVALUATION

Our evaluation is done local to the FUs so that it does neither depend on the compiler nor on a particular program. The goal of this evaluation is to explore chances and limits to increase the performance and to determine worthwhile application scenarios. Thus, the focus is put on the throughput of FUs, as well as its relation to the circuit size.

The evaluation of the algorithm is done in two steps: First, performance is measured using test benches simulating DTN, MIB and execution units which are parameterized by different buffer sizes, execution delays, and DTN speeds. The measured speedup is expressed as

\[
S_{type} = \frac{T_{simple}}{T_{type}},
\]

where time is measured in terms of DTN cycles. The type is either \( \text{simple} \) (Section IV-A), \( \text{ooo} \) (Section IV-B), or \( \text{p25/50/100} \) for partial buffers with different percentages of out-of-order area used (Section IV-C). Used addresses and value write delays describe different scenarios:

- \( \text{wc} \) is the worst-case for out-of-order execution: all values arrive in dataflow order.
- \( \text{bc} \) is the best-case: all values arrive in reverse dataflow order.

In a second step, circuit size is taken into account. To this end, buffers are synthesized using different buffer sizes. Circuit size is measured by the number of lookup tables (LUTs) of the FPGAs used for logic in combination with the number of registers needed, totaling to the slice count \( C \). Memory modules are not taken into account. For correlation with the performance measurements, the results are again converted to the factor

\[
I_{type} = \frac{C_{\text{simple}}}{C_{\text{type}}}
\]

describing the increase in circuit size relative to the simple buffers.

In \( \text{wc} \), the partial and out-of-order buffers, perform worse than the simple buffer because of the additional overhead. However, with increasing buffer size, the performance decrease becomes smaller. Partial buffers generally have less negative impact in the worst-case situation.

The graph in Figure 3 shows the average speedup in the best-case scenario for different buffer sizes. The size of the reorder area in the partial buffers has a huge impact on the overall speedup. This is due to the fact that buffers are filled in reverse and out-of-order execution cannot start until values are inserted into the reorder unit at the bottom of the buffer. While the partial buffers start at rather high speedup factors, they quickly saturate with increasing buffer size. Out-of-order buffers, on the other hand, further increase performance with growing buffer size. Looking at different DTN speeds, we found that partial buffers generally benefit from higher speeds, while for out-of-order buffers, the opposite is the case. Further tests showed that a delay induced by the execution unit, pipelined or not, has no impact if it is less than the DTN delay. Again, partial buffers benefit most from lower delays.

Looking at the circuit size in Figure 4 shows that the simple buffers have a small circuit growing linearly with increasing buffer size. Adding the out-of-order logic increases the growth factor, but size is still increasing linearly for the tested buffer sizes. Storing entries in registers heavily impacts the size of the partial buffers. All tested instances show exponential growth where the exponent increases with the size of the reorder area,
quickly generating circuit sizes magnitudes larger than the simple or ooo type buffers.

Figure 5 relates the average speedup of the interleaved stream tests to the size increase factor. Out-of-order buffers only yield improvements for sizes larger than 64 while buffers of size four do not yield any speedup in general. Partial buffers p25 with sizes 16 and 64 have a good speedup factor while still showing a comparably low increase of circuit size. At size 16, p50 and 100 both have the same good performance increase compared with the p25 buffer, while the p100 buffer has a notably larger size. The higher speedup from any of the larger buffers can unlikely be utilized, because of the high increase in circuit size.

VI. CONCLUSIONS

This paper considers an out-of-order execution for function units of exposed data path architectures with buffered function units, and discusses different implementations for the same. Experimental results show that the overall performance can be improved by out-of-order execution, and that it can be implemented with reasonable hardware overhead. However, it also becomes clear that none of the given implementations is optimal in every situation, so that a suitable parametrization and implementation has to be carefully chosen. The out-of-order buffers work best for processors with large buffers and slow data transfer speeds, while the partial buffers perform better in the opposite case.

REFERENCES