# Towards a High-Speed Programmable Hardware Architecture for the Network Transport Layer

Kimiya Mohammadtaheri, Nachiket Kapre, Mina Tahmasbi Arashloo

University of Waterloo

**Abstract**

As network speeds keep increasing, there is a growing interest in accelerating network functions using programmable Network Interface Cards (NICs). Transport protocols play a critical role in managing traffic flow and resource allocation, yet their high-speed implementation on NICs remains a challenge due to the complexity of stateful operations and the need for low-level hardware expertise. This project introduces a high-level programming platform called MTP that allows users to specify transport protocols without concerning themselves with hardware-specific details. We will then introduce a compiler that can translate this code to a low level hardware design. We will then explain the general structure of such hardware design and the challenges of mapping the components of the high-level platform to hardware and leveraging the capabilities of hardware.

## 1 Introduction

With network speeds increasing to 100s of Gbps and beyond, there is a growing interest in accelerating network functionality using programmable Network Interface Cards (NICs). Accelerating transport protocols is of particular interest as they are essential in determining how traffic flows from different applications share network resources and significantly impact application performance. There is no generally accepted solution for network transport and several network operators, e.g., hyper-scalers, develop their own transport protocols with customized congestion control algorithms and/or periodically modify them to maximize network performance and utilization. However, developing a high-speed implementation of the stateful and often-complex transport protocols on NICs is challenging – it requires working with low-level programming interfaces and acquiring deep knowledge of the NIC's specific hardware architecture. This project aims to solve this problem by developing a high-level programming platform for transport protocols. Users would specify protocols in a high-level language, without having to consider the NIC's hardware architecture and performance-related optimizations. A compiler will then create an optimized implementation for the specific hardware target intended to run that protocol.

In this report, we will first introduce the components of our high-level model in section 2, then explain the general components and ideas used in implementing the hardware in section3. Then we will focus on the implementation of the scheduler component, and explain the ideas and challenges in implementing different parts in section 4. In the end we will discuss the implementation details of the scheduler in section 5

## 2 MTP Language

MTP is a language designed to define transport layer functionality through high-level "transport programs." These programs process events and flow state to produce updated states and target-independent instructions for core transport tasks like data reassembly, packet handling, scheduling, and timer control. The language was developed by analyzing shared functionalities across transport protocols, studying their implementation on existing systems, and creating a unified, high-level way to represent them. Consequently, many elements of MTP programs align with protocol-specific implementations already present in current systems, making integration feasible.

The transport layer is the moderator between the application and the network layer. Broadly speaking, it receives data transfer requests from the application layer, and generates the appropriate data segments for the network layer to transfer to the other endpoint of the communication. It uses feedback from the other endpoint and the network, in form of packets, as well as timers, to decide how fast to generate segments and whether it needs to retransmit potentially lost data. As a receiver, it reassembles data segments received from the network if necessary and transfers them to the application.

Given the event-driven nature of transport protocols, MTP 's model significantly revolves around events and event processing. The various components within the transport layer interact by passing around events, and four "interface" modules take care of creating and consuming events at the boundary of the transport layer and the application and network layers.

### 2.1 Events and event processing

The transport layer receives events from multiple event streams for each flow, including (1) *network events* stream, i.e., the sequence of control and data packets received from the network layer for that flow, (2) *application events* stream, i.e., the sequence of data transfer requests coming from the application layer, and (3) *timer events* stream, which represents events that trigger after a configured amount of time, such as retransmission timers.

**Event schedulers** Since a flow receives events from concurrent event streams, it can have more than one outstanding event at a time waiting to be processed. Moreover, the transport layer has to handle multiple concurrent flows at the same time. These can be points of contention as event processing requires access to memory and computational resources. As such, MTP has a dedicated moduel for event scheduling. The extent to which an MTP program can influence event scheduling is dependent on the target. However, no matter what the internal logic of the scheduler is, its interface with the rest of the transport layer is to receive multiple streams of events from multiple flows as input, and output the next event that should be processed.

**Flow context** Transport protocols are stateful – they maintain some state for each flow that persists across events and throughout the lifetime of that flow. Examples include the first unacknowledged sequence number or the congestion window size for TCP, or next expected packet sequence number (PSN) in RoCEv2. The per-flow state is called *flow context* in MTP . A (logical) context table maintains the set of flow context for all active flows. Once the scheduler decides the next event, the context of its corresponding flow will be retrieved from the context table for processing that event.

**Event processors and dispatcher** Event processors are the building blocks that, together, specify the core logic of the protocol. Each event processor takes an event, the corresponding flow context, and some metadata as input, and updates the flow context and metadata. It may also generate some instructions that would guide the target to generate packets, update the data memory, notify the application or start a timer. The dispatcher sits between the scheduler and event processors, deciding how to *chain event processors* to process an event. Once the event reaches the end of the chain, the context table is updated with the modified flow context.

Having a modular event processing logic was a deliberate design decision. Given the complexity of transport protocols and the diversity of events they need to handle, we found it essential to break down event processing logic into smaller modules, i.e., event processors, and have a dedicated module, i.e., the dispatcher, to explicitly call out the list of event processors that are involved in processing each type of event.

**One-event-at-a-time semantics** Concurrent consistent access to shared state is a challenge in stateful network functionality, and the transport layer is not an exception. Handling concurrent state access correctly and efficiently depends on the target and is orthogonal to the protocol's logic. As such, MTP 's logical model assumes that the transport layer processes one event at a time, i.e., no other event will start processing until the current event reaches the end of its event processing chain. The compiler will take care of batching and/or parallelizing event processing to optimize performance while ensuring consistent state access.

## 2.2 Interfacing with the network layer

**Net-to-transport** This module receives a packet from the network layer and generates one or more events that will enter the transport layer through the event scheduler. MTP programs can specify how packets should be parsed to extract the information needed for creating appropriate transport-layer events, using a syntax similar to existing packet processing languages like P4 [1].

Most of this information comes from transport-layer headers. However, for data packets, this module needs to handle the payload as well. Specifically, it stores the payload in a temporary memory, the *holding area*, and returns the address. The address can be included in the event, so that it can be further used as input for data reassembly instructions

The MTP programming model does not require specifying implementation details about the holding area and buffer memory spaces, such as how memory is allocated to data segments, or how the payload is extracted from the packet and moved between different modules, as they are target-dependent and orthogonal to the protocol logic.

## 2.3 MTP syntax

An MTP program is written against the logical model of the transport layer. It describes the events a protocol expects and how they should be created from incoming packets or application requests. It also describes the protocol's flow context and event processing logic. In this section, we use TCP as an example to introduce MTP 's language, particularly its built-in functions and special language constructs as well as its restrictions.

Note that MTP programs specify the contents of the model's "programmable" blocks, not the flow of events between them. That is, an MTP program is a collection of code blocks that can be plugged into the abstract model; the abstract model governs how these blocks connect and interact with each other.

**Events** MTP has several built-in event types from which protocol-specific events can be derived. Specifically, the type event_t is the base event type for all events, incoming is for events entering the transport layer from the network or application, and timer_event is for timer-related events. Moreover, every event in MTP is tagged with a flow ID that can be set using set_fid in interface modules and retrieved using get_fid in other parts of the program. Protocol-specific events are defined with the keyword event, specifying one of the above "major" event types as its parent, and listing the event fields in a syntax similar to C++ structs, for example, the code snippet below describes an incoming event from the application layer for data send requests:

```
event tcp_send : incoming::app_event {
   addr_t addr; int32 data_size;}
```

**Deciding the next event** the programmability of the event scheduler depends on the target. MTP provides language constructs for the target to expose the available scheduling blocks to the programmer to instantiate and use . Independent of the extent of its programmability, the event scheduler interface is to accept events as input and output the next event for processing.

**Defining the flow context** Once the next event is decided, the transport layer looks up the context for its corresponding flow from the context table. MTP programs can use the keyword context and struct-like syntax to define the state that should be maintained for each flow in its context.

```
context tcp_ctx {
  // sender-side
  int32 send_una, send_nxt, cwnd, last_rwnd;
  timer_t rto_timer; int8 duplicate_acks = 0;
  ... // receiver-side
  int32 recv_next = 0;
  sliding_wnd meta_rwnd;
  buffer_id_t bid; ...}
```

**Dispatching events** MTP programs can use the dispatch table to specify the sequence of event processors for an event:

```
dispatch tcp_dispatch {
  tcp_send     -> {record_data, gen_seg};
  tcp_ack      -> {rto, cong_ctrl,
                   fast_retransmit, gen_seg};
  tcp_data_pkt -> {proc_recv, send_ack};
  tcp_timeout  -> {proc_timeout};
  tcp_read     -> {flush_data};}
```

The dispatch table is a mapping between an event type and a sequence of event processors, resulting in a simple sequential control flow logic between event processors. In our running example, the event is a tcp_event and will be processed by the record_data event processor first and then gen_seg.

We considered making the dispatcher more "flexible", from a control flow perspective, by having it be a function that takes an event as input and uses simple imperative language constructs such as assignments, conditionals, and function calls to call the event processors needed to process the input event. However, our observation from developing transport protocols in MTP is that event processing can be naturally divided into a set of rather disjoint tasks. For instance, in our TCP example, on receipt of an ack, we would want to re-calibrate the RTO based on RTT, adjust the congestion window, decide if we need to retransmit a lost segment and if we can transmit new segments. Most of the control-flow complexity is typically within a task not across tasks – we just need to make sure the tasks are called in the right order (e.g., adjusting the window before trying to transmit new segments).

MTP 's dispatch table does just that. It also provides a precise yet high-level view of the protocol's response to various events. This can help in customizing and maintaining a protocol as it evolves – it is easy to track which event processors are involved in processing which events as well as overlaps between processing chains, identify what event processors need to get updated for a new feature, or plug in a different event processor for the same task (e.g., a different congestion control algorithm). Moreover, imposing the domain-specific restriction of sequential control flow at the language level can potentially simplify automated analysis and help compilers generate more efficient implementations.

## 3 MTP Hardware Backend Overview

The focus of this project is to design a setup that would serve as a base for the translation of MTP programs. There have been multiple attempts to offload different network functionalities to hardware FPGA. Tonic [2] presents a programmable hardware architecture for transport layer algorithms. It does this by providing a programming interface consisting of common transport layer operations. ClickNP[5] and AccelNET[4] are attempts to offload different layers of the network to FPGA hardware.

In this case, the main goal of the design is to stick to the overall structure of MTP while leveraging potential hardware capabilities for optimization. The overall structure of the proposed module can be seen in figure 1. In this module, the network packets will arrive at the network parser, while application requests will arrive at the application parser. Both these parsers will generate events corresponding to their input. The timer module can also generate some events, which will be explained in later. These incoming events arrive at the scheduler. The scheduler will store these events and choose the next event from the stored events. It will chose one event of each type to be processed in the next stages. The MTP framework has a dispatcher at this stage, however, this module is not necessary here since the events can be processed in parallel, The information in the dispatcher will be used by the compiler to determine the structure of the event processors module. After leaving the scheduler, the flow id of these events will be sent to the access memory to fetch corresponding rows from the context memory. The event processor will process these events based on the program defined in the MTP file. After this process is over, the event processor will update the context memory. Additionally, it might generate some instructions based on the original program, and send them to the corresponding modules. The "Application Message Generator" is responsible for sending messages to the application, the "Data Memory Handler" would send messages and instructions to the external memory that is keeping the data, "Packet Generator" can generate and schedule packets, and the "Timer" module will start a timer that would generate an event upon expiration.

## 3.1 Network Parser

The role of the network parser is to receive the packets coming from network, parse them and generate new events according to the rules defined by the programmer. The format of the parser programs bears a close similarity to the match-action rules in the programmable switches. These parser programs contain rules that generate events based on the value of the certain header fields. Due to this similarity, we expect to be able to leverage existing hardware components that provide similar benefits to switches, such as RMT to provide this functionality for our module.

## 3.2 Event Processors

Event processors consist of the main logic of the protocol, which explains how all incoming events will be processed. The syntax used to describe the event processors is a C like language that provides a few extra structures and functions specific to MTP . While we can design a process to optimize and convert these codes to hardware, this attempt is not trivial and there are already a group of tools, called high-level synthesis tools (HLS) that are optimized to preform this task. High-level synthesis tools are designed to simplify the

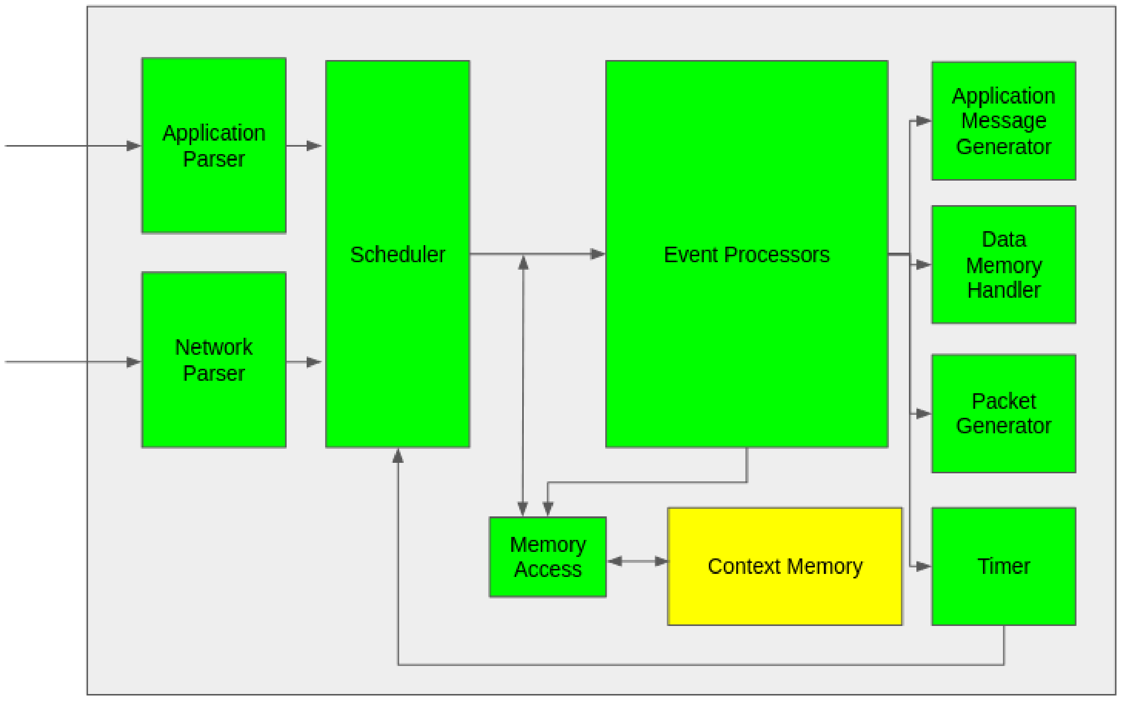


Figure 1: MTP hardware overview.

design process of hardware by allowing a higher level of abstraction than hardware design languages such as Verilog or VHDL.

For this project we use Vitis-HLS. This tool, developed by Xilinx, is a HLS tool specifically tailored for Xilinx FPGA platforms. It enables designers to accelerate their design process by converting high-level C/C++ code into RTL (Register Transfer Level) code, which can then be synthesized and implemented onto Xilinx FPGAs. Vitis-HLS provides various optimization techniques to improve performance, area utilization, and power consumption of the synthesized hardware design. These optimizations include loop unrolling, pipelining, memory partitioning, resource sharing, and more.

We are able to convert MTP codes to C codes with some minor adjustments, and then we can use HLS capabilities to convert these codes into hardware ones. The MTP framework allows the programmer to divide the processing logic of each event into multiple smaller parts. While this division improves the readability of the design, it is not necessarily the best way to design the hardware pipeline for these chains, therefore, in the compiler, we will compact each chain into one program, and generate a pipelined version of each one separately.

### 3.3 Timer

For designing the timer module, we use the model and observations made by the authors of the Tonic[2] paper. In Tonics "Periodic Updates" module, the module would iterate over the active flows every clock cycle, check their status, and preform certain tasks upon expiration. The authors observe that although this method might detect time-outs with some delay, since the clock cycle of the module is in the order of nanoseconds, this delay would ultimately be negligible. These observations would also apply to our case, therefore we use a similar model for our timer. In our case the timer module will store the events that it is supposed to generate in a small memory, and retrieve them when their corresponding timer expires.

## 4 Event Scheduling Hardware

Event scheduling is one of the main parts of the MTP design. The responsibility of the event scheduler is to receive events from different sources, which could be application, network or timer, and choose one event of each type to be processed by the event processors. The scheduler is not one of the programmable parts of the MTP, and the programmers have limited control over its structure. This will give us more freedom in designing the hardware module for the scheduler. This freedom can help us in designing the scheduler in a way that would improve hardware utilization in later sections.

All events access the shared context memory before getting processed by the event processors, and update the context memory after finishing the process in the event processor pipeline. This means that when an event enters the pipeline, all events of the same flow have to wait for that event to leave the pipeline and update the context before entering the pipeline themselves, so that they can have access to the latest version of the shared memory. Data collision happens when an event wants to access the shared context memory while another event from the same flow is being processed in the event processor. The typical way to avoid this problem is to stall the pipeline until the offending event has left. However, if the scheduler can keep some information about the flow with an active event[1], it can schedule the events in a way that ensures such collision would not happen. Additionally, In the domain of networking, we are expecting the number of active flows to be much greater than the number of event processor chains , meaning that there would always be at least one flow with no active events that the scheduler can safely choose to send to the event processors.

In the hardware, The event scheduler is expected to receive one event of each type (i.e. network, application and timer) per clock cycle. The scheduler will then store the event in an event queue corresponding to its flow. The main responsibility of the scheduler is choosing one event of each type to send to event processors while ensuring fairness between flows and avoiding data collisions. The hardware structure of the schedule can be seen in Figure2. Events of different type will arrive at the queue cache (1). At the same time, the

[1]active event is and event that is being processed in the event processor pipeline

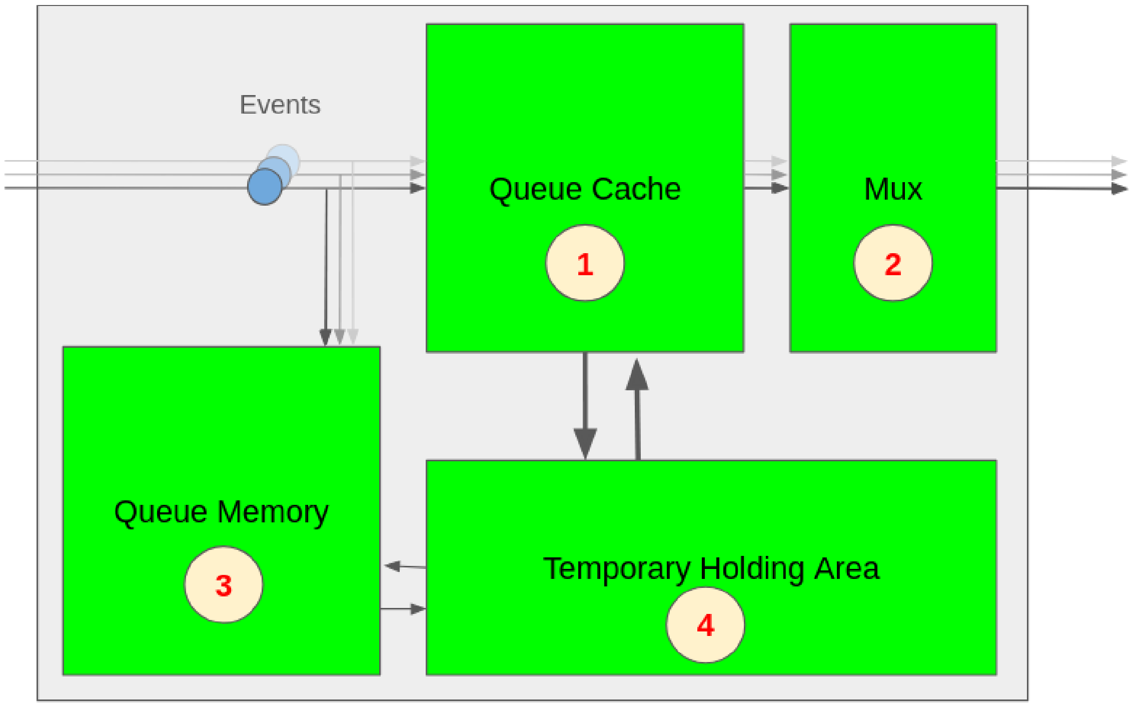


Figure 2: MTP hardware scheduler overview.

queues corresponding to the flow id of the incoming events will be read from the queue memory (3) and stored in the temporary holding area (4). by the next clock cycle, the new event will be inserted in the queue and the queue cache will decide to either replace its current row with the one from memory, or to keep the row. Meanwhile, the multiplexer (2) will choose the next event to be sent out for each event type, and sends a dequeue command to the corresponding queues. After receiving the dequeue command, the queues will remove the first element in the list and start their timer. the queue is considered invalid by the multiplexer as long as the timer is active.

## 4.1 Next Event Selection

### 4.1.1 Queue Cache

The queue cache is a module that keeps the events of a subset of flows, and chooses the next event from them. each row of this cache is represented by a queue box, where each flow would be mapped to one of these queue boxes. The queue boxes are supposed to keep track of the information that would allow them to avoid data collisions. The queue box contains one queue per event type, for example, it can have one queue for network, one for application and one for timer events. We will refer to these queues as "mini-queues". When choosing the next event, Each queue box is supposed to report the validity of its

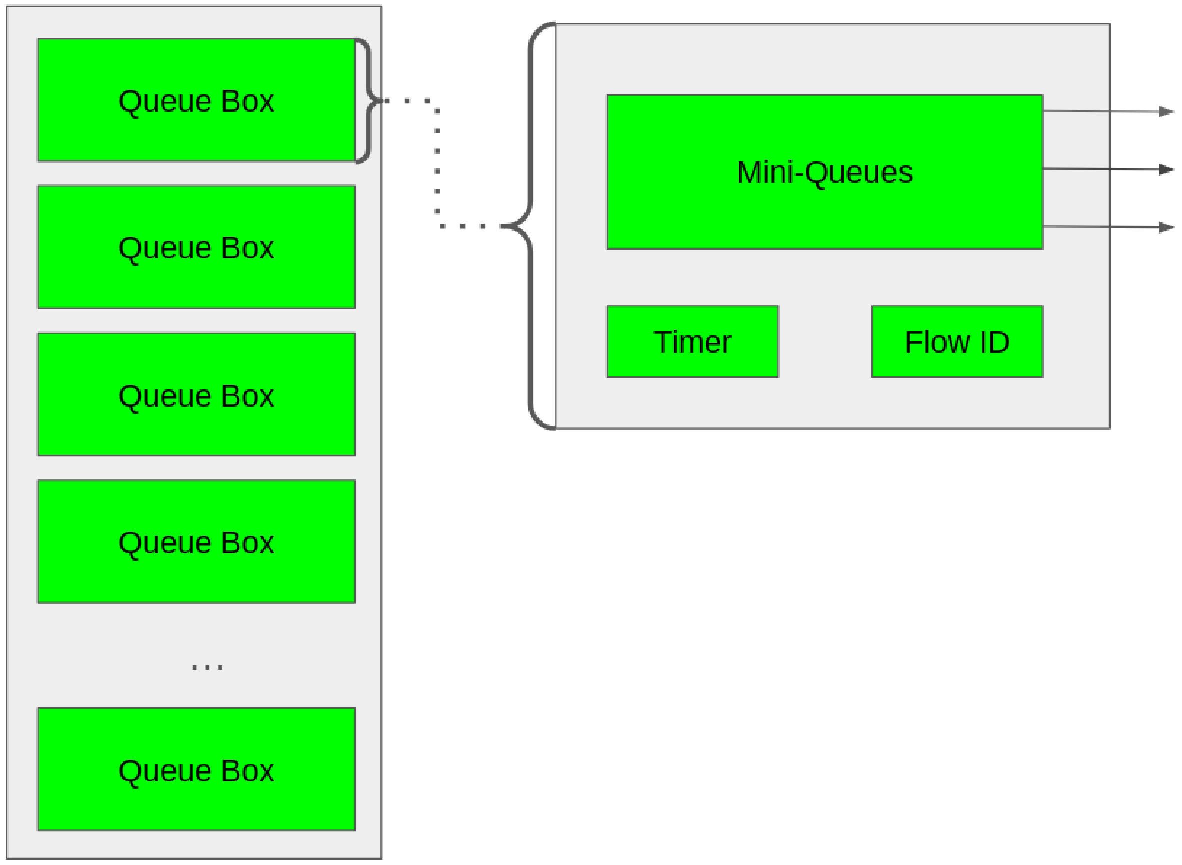


Figure 3: Queue cache and Queue box.

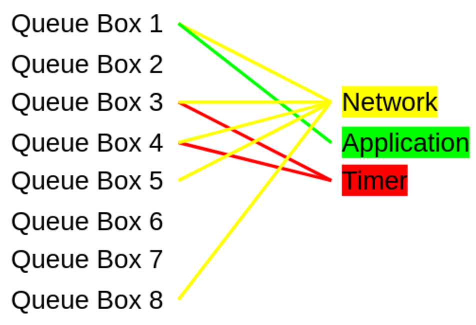
mini-queues to the multiplexer, so that the multiplexer can choose the next event of each type accordingly. The validity bit is decided by two factors: 1.Whether the queue actually contains any data and 2.Whether the flow has any active events in event processors or not. If there is no data in the queue, or if the flow has an active event, the event queue would be considered invalid.

In order to track the active event, each flow will start a count-down after dequeuing an event from any of its mini-queues. The starting point of the counter is the number of cycles it would take for an event to leave the event processor pipeline, and this number can be known beforehand. if the counter has not reached zero, it would mean that there is still an event from this flow id inside the event processors.

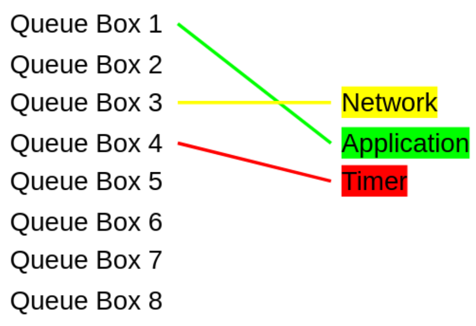
The overall structure of the queue cache and queue boxes can be found in Figure 3 . In summary, each queue box has the following parts:

- **Mini-queues:** These are the FIFO structures inside the queue box, each mini-queue is responsible for keeping the events of one event type.

- **Timer:** There is a counter module in the queue box that we will refer to as ”timer”, whenever the timer becomes active, it starts counting down from the ”memory update



(a) The valid queues for each queue box, represented by the lines connecting the queue box to the event types



(b) Selecting one valid event for each type

Figure 4

cycle", which is a pre calculated value that shows the number of cycles needed to process events. When the counter reaches zero, the timer will become inactive.

- **Flow id:** This register shows the flow id of the events that are currently being kept in the queue box.

In the hardware, in order to choose the next event, each queue box will output N bits, with N being the number of different event types, that represent the validity of each mini-queue. each validity bit is determined by an "and" between two factors: 1. the validity of the element in the front, which can be determined by the "valid" bit of the element and 2. the state of the flow timer. If the timer is active, all mini-queues would be considered invalid. After these calculations, all validity bits and the front element of each queue will be passed to the next module, the multiplexer.

### 4.1.2 Multiplexer

After determining the validity bits in the queue cache, the multiplexer will receive these bits from the queue cache and must choose one valid event from each type to send out. The multiplexer cannot choose two events from the same flow, as this would lead to data collision. Therefore we need to consider the following conditions when designing the multiplexer:

1. It has to be fair, meaning that all queue boxed must have a fair chance of getting chosen.
2. It must try to maximize utilization, meaning that if possible, it should try to output one event for each event type if at least on of the queues has a valid event of that type.
3. It must run at line rate.

If we look at the problem from a different angle, in this problem, each queue box is sending a request to some event types, asking to be able to send events to its corresponding event processor pipeline. This request can be shown as a bipartite graph in figure 4a. In this figure, five queue boxes have valid network events, two have valid timer events and one has valid application event. One way to select the next event is to choose the application event from box 1, network event from box 3 and timer event from box 4, as shown in figure 4b. We can see that the problem resembles the maximum bipartite matching problems and we can consult the existing hardware solutions for this problem to help us in finding the best algorithm.

One such solution is the iSLIP algorithm [6] algorithm, which is a well-known algorithm for scheduling switch crossbars. Additionally, there are existing works that use and improve upon this algorithm for the purpose of network switches[3]. These solutions are particularly interesting to us because they attempt to solve a similar problem at line rate. Specifically, in switch cross bar scheduling, each input port can have packets destined to one or more output ports. An ideal algorithm will try to match as many input ports to outputs ports without any conflict, where a conflict happens when two input ports try to send to the same output port simultaneously, or an input port is chosen to send to more than one output port simultaneously. In our event scheduling problem, flows correspond to input ports, and processing chains for specific event types correspond to output ports. So, we can use a similar approach to find conflict free maximal matchings between flows and processing chains.

#### 4.1.3 An Example

To better illustrate this section, we propose the following scenario:

## 4.2 Queue Swapping

As mentioned in section 4.1, Each queue box would need its dedicated timer and registers, as well as allowing us to enque/dequeue and access the front element of all queues at any given moment. All these requirements would make the memory more costly, and prevents us from using the pre-existing and optimized memory modules such as BRAM. All these restrictions prevents us from using queue boxes to store all flows separately. Therefore, we have use the queue cache to only keep a subset of the flows, and keep the rest in the queue memory. We expect the queue memory to be wide enough to store all mini-queues of a queue box in a single row, so that we can access the memory in fewer clock cycles.

By adding this separation, we must devise a policy to swap the queues to and from memory. We want to ensure that all flows will get their events processed at reasonable times. Additionally we must have a way to enqueue the incoming events belonging to flows outside of the queue boxes. To achieve the second objective, every time a new event arrives, we find its flow and read the queues of that flow from memory. This will give us an advantage when deciding whether to swap the contents of the queue cache or not. When deciding on the swap policy we note that we are expecting a subset of the queue boxes to have active timers at any given time. We can use this property to our advantage and swap out the queues that are currently waiting. In order to have an effective replacement policy, we need to make sure that (1) none of the flows starve, (2) we maximize the utilization and (3) each flow would get a fair share of the cache.

### 4.2.1 Replacement Policies

One natural way to do the swapping is to replace the flow in the queue cache whenever it is inactive (i.e. its timer is active). Whenever a flow is inactive, it is using resources but cannot do anything, therefore it would be beneficial to swap out such flow with a new flow that can send new events out. We call such scenario the ”New Arrival” scenario. In this case, Whenever a new event arrives, the queue box may choose to replace it's current flow id with the flow id of the incoming event, and replace the mini-queues accordingly. The queue would choose to make the switch if its timer is active, since the active timer means that the flow id have recently sent an event to the event processors and will be invalid for some time, therefore it would be beneficial to switch the event out.

After preforming this switch, we need to ensure that the newly swapped flow will have the correct timer state, since it is possible for the flow to still be inactive. to illustrate this, we consider the following example. In a module with event processing pipeline of length 10, the network queue of queue box 1 receives events from flows 1,6,6 and 1 across four cycles and holds events from flow 1 at the start. It gets chosen to send flow 1 in the first cycle, and starts its timer. after receiving an event from flow 6, it will replace its current queues with those of queue 6 and deactivates the timer. in the next cycle, it gets chosen to send out one event from flow 6, and activates the timer again. Afterwards, it will receive an event from flow 1 and has to bring it back. It can be seen that the timer of the flow 1 should still be active, therefore flow 1 is still inactive, however we would not be able to track this if we don't track this flow's timer after swapping it out. In order to track the timers of the swapped out flows, we can dedicate a small set of counters that keep counting down after the event gets swapped.

While the new arrival swap may be beneficial to the overall utilization, it is not enough to ensure fairness and avoid starvation, since this way, the only flows that have a chance to joint the queue cache are the ones that receive new events, meaning that if a flow does not receive new events, or receive its events when the flow in the queue box is active, it would never return to the queue cache. To address this issue, we need to also preform the replacement independent of the input. One option would be to add new ports to the memory dedicated to preform these independent swaps, however, this would imply adding new ports to the memory and this would increase our design complexity. In this case, there is another way to access memory for these independent swaps without needing to add new ports to the memory. When looking at the expected patterns in network behavior, We observe that even though the schedule is designed to handle one event of each type per clock cycle, in reality, network is the only source that is likely to produce one event per clock cycle, and other sources such as application or timer will be less frequent, and this would lead their corresponding links to be idle. Therefore, whenever these links are idle

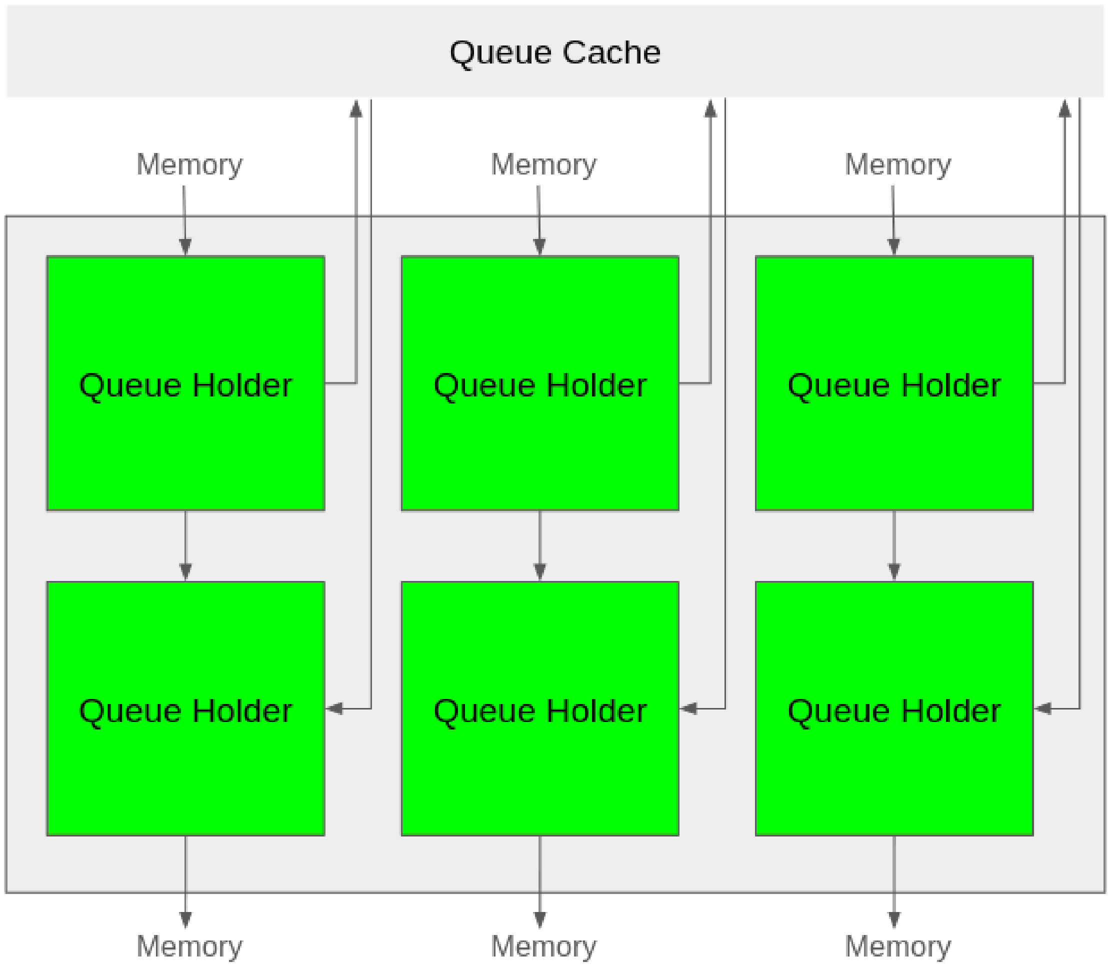
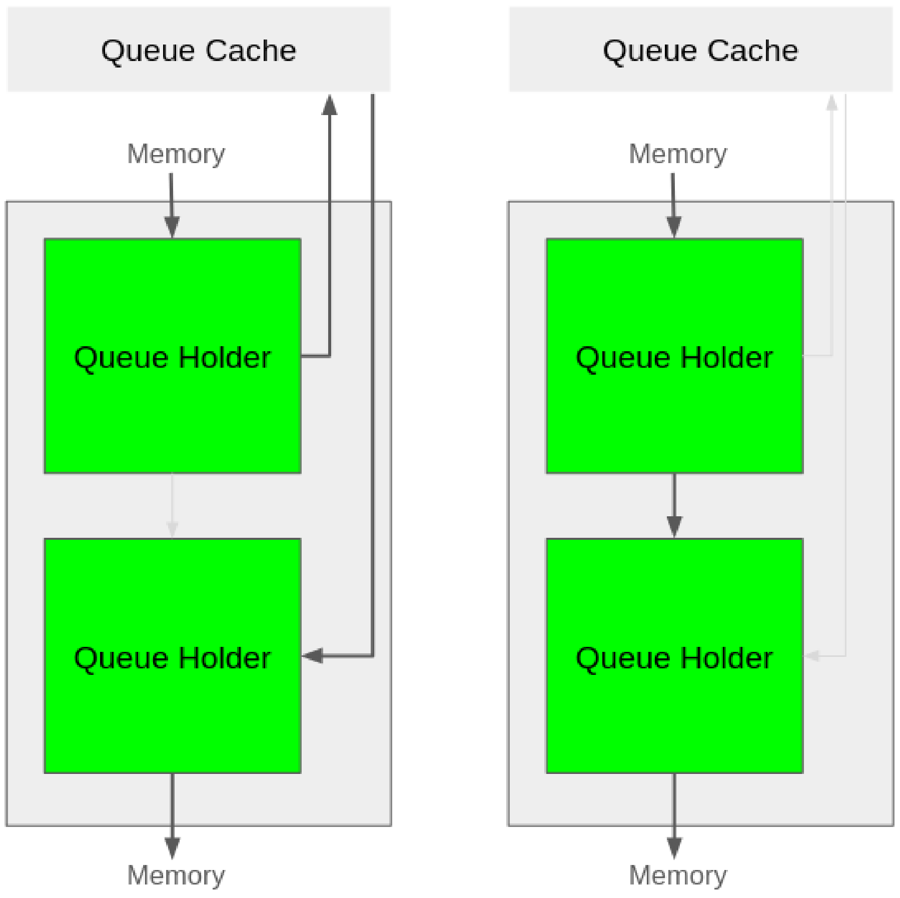


Figure 5: Temporary Holding Area.

and no new event arrives, we can use these idle links to preform the independent swap.

The independent swap also needs a policy to choose the flow that will replace the current inactive flows, as well as the inactive flow to replace. the policy for chosing which queue box to replace can be round robin between inactive flows, with new additions joining the end of the list, this scenario would be fair since there is no particular difference between different queue boxes.

For choosing an event to swap-in, one Idea is to swap in the event that has been swapped out the earliest. In this case, we would need to keep a small queue of the flow ids that leave the queue box, and dequeue the front every time we preform an independent switch. In the cache, each flow will be mapped to a specific queue box, therefore each queue box will be able to keep track of the flows that leave it.



(a) Replacing the queues in the queue box

(b) No replacing happens

Figure 6

### 4.2.2 Temporary Holding Area

The process of swapping needs multiple cycles to be done, since we would need to first read from memory, then store the current queue contents in the memory before replacing them with new ones. It is also possible to not replace the flow in the cache with the one we read from memory, in this case, we would need to be able to store the new event that has arrived for this flow, and store the queues in the memory. To achieve these two goals, we implement a new module called the ”temporary holding area”

The structure of the temporary holding area can be found in Figure5. It consists of two rows, which correspond to the number of cycles we need to preform the switching action, and shape the two stages of the pipeline in this module. The first row will hold the queues that we read from memory, it also has the ability to insert the new event to it’s corresponding mini-queue, it will either send its queue to be replaced with the current content in the queue box, or it passes the updated queues to the next row to be rewritten in the memory. The second row will keep the data that is supposed to be stored in the memory. This data will either come from the previous row or the queue cache, depending on wether we choose to replace the cache or not. Figure 6 illustrates different scenarios that can happen with temporary holding area. in 6a, The queues in the first row will be sent to the cache, and the old contents of the cache will be stored in the second row, ready be written in the memory. In 6b, The swap does not happen, so after the queue gets updated in the first row, it moves to the second row to be written in the memory.

## 5 Implementation

In order to test and implement our proposed design, we implemented the proposed structure of the scheduler in a hardware-style C++ code, taking into account the cycles and wire connections between each module. The overall loop is as follows:

```
void runCycle(array<Event*,MAJOR_TYPE_COUNT> incoming_events){
        MapperOut newMOut;
        QueueCacheOut newQCOut;
        QueueMemoryOut newQMOut;
        TempHoldingAreaOut newTHAOut;
        MultiplexerSetOut newMuxOut;

        array<int,MAJOR_TYPE_COUNT> readAddr;
        for(int i = 0 ; i < MAJOR_TYPE_COUNT ; i++){
            readAddr[i] = incoming_events[i]->flow_id;
        }

        newMOut = mapper.runCycle(incoming_events);
        newQCOut = queueCache.runCycle(mOut.incoming_events,mOut.
           enqueue,muxOut.dequeue,thaOut.cacheInput);
        newMuxOut = muxSet.runCycle(newQCOut.validity_matrix,newQCOut.
           events);
        newQMOut = queueMemory.runCycle(readAddr,thaOut.writeAddr,
           thaOut.writeSignal,thaOut.memOutput);
        newTHAOut = tempHoldingArea.runCycle(qmOut.output,qcOut.isSwap
           ,qcOut.SwappedOut,mOut.incoming_events);

        storeOutput(newMuxOut.event_out, newMuxOut.isValid);

        mOut = newMOut;
        qcOut = newQCOut;
        qmOut = newQMOut;
        thaOut = newTHAOut;
        muxOut = newMuxOut;
    }
```

In these implementation, we added each module as described in section 4. Each module has a runCycle function that simulates the behavior of that module in one cycle. The input of these modules are the outputs of other modules from the previous cycle, this way we simulate the pipeline behavior of the module.

## 6 Conclusion

This project introduces a high-level programming platform called MTP that allows users to specify transport protocols without concerning themselves with hardware-specific details. A compiler can translate this code to a low level hardware design. We explained the general structure of such hardware design and the challenges of mapping the components of the high-level platform to hardware and leveraging the capabilities of hardware.

## References

[1] P4 Open Source Programming Language. https://p4.org/. Accessed: September 2024.

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, Santa Clara, CA, February 2020. USENIX Association.

[3] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcpim: near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 53–65, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[5] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[6] N. McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, 1999.