Network Traffic Monitoring – an architecture using associative processing.

Gerald Tripp

Technical Report: 7-99
Computing Laboratory, University of Kent

1st September 1999

Abstract

This paper investigates possible associative processing architectures for use in the implementation of a real time network traffic monitoring system. The proposed solution is a simple associative mono-processor based on a small number of electronic components including state-of-the-art ternary content addressable memory. This would enable a large number of finite state machines to be implemented which could be used to track the activity of multiple data channels at several protocol layers. The system would receive a stream of packets of data from a network and could be programmed to generate output event messages consisting of selectively captured network data or other information.

1. Introduction

This report looks at possible associative processing architectures for implementing a network traffic monitoring system. The aim here is to be able to collect useful traffic and other information from a computer network for further processing. To be able to perform this function, any processor needs to be able to follow operations on multiple traffic streams from the network – only passing on occasional packets and other data as required.

A previous report [1] looked at various methods for performing data reduction in network monitoring systems and made a proposal for a finite state machine (FSM) based system that used associative memory to provide the tables for generating next state and any FSM outputs. This report looks at how this can be extended to enable a system to be built that would enable a large number of finite state machines to be implemented which could be used to track the activity of multiple data channels at several protocol layers. The system would receive a stream of packets of data from a network and could be programmed to generate output event messages consisting of selectively captured network data or other information.

The remainder of this document is structured as follows. The next section gives a brief review of the proposal made in [1]. The third section looks at some of the problems with implementing multiple FSMs and introduces the concept of a hierarchical channel based system. Section four looks at the type of actions that we may perform including proposals for arithmetic operations and output event generation. The fifth section describes the various ways in which the network data may be accessed by the FSM. Section six looks at practical implementation issues and the final section looks at conclusions and ideas for further work.

2. Associative FSM Proposal

The system proposed in [1] simply uses associative memory to implement the next state and output functions for FSM implementation. A benefit of this method of implementation is that it allows us to use a relatively large word of data from the network as input to our FSMs. As an example, a 32 bit input word would be perfectly feasible using current associative memory components. The type of associative memory used is ternary content addressable memory, sometimes referred to as functional memory [2]. This is useful as it allows patterns to be stored in the memory with each bit having a value of 0, 1 or don't care. The input value we are searching for is presented as a 'key' to the content addressable memory (CAM) along with details of which bits of the memory word should be used for the search. The CAM compares the key with all (pattern) words stored in memory and gives a Match/Fail result. For a match, the CAM either gives an address of the word or gives direct access to that memory location. If there are multiple matches, then the CAM will typically select the one with the highest priority – such as the one with the lowest (or highest) memory address.

We then implement our FSM by using the CAM to store a table of code words such as below:

State	Data match pattern	Next state	Actions

The FSM operates by presenting the current state and network input data as a `key', to the CAM. These values are compared against the State and Data match patterns respectively of each code word in memory. The highest priority code word to give a match will be used to generate the next state and the FSM output (the action). Hence we are able to generate a series of these code words that specify the operation of our FSM. One or more code words will be used for each current state, to specify the next state and actions for a variety of different values of input data. For each state, we will need to ensure that all possible combinations of input data values have been covered – this may require a code word for some states that matches don't care for all bits in the network data. As the matches are prioritized, care will need to be taken that the code words are stored in memory in the correct order – e.g. the don't care case will obviously need to be the lowest priority.

It is assumed that there will be an initial state that a FSM enters at the start of a received packet. The FSM will then operate by using as input, words of data from that packet. This could be all words from the packet in sequence, or the FSM itself might be able to select the data required. Each time a code word is retrieved from the CAM, there may be an associated action that needs to take place. The actions could be used for modifying the value of local variables – such as counters – they could modify the initial state that the FSM enters at the start of a packet and they might also generate an output event such as capture of the current packet.

3. Implementing Multiple FSMs

Any monitoring system will typically need to be able to monitor traffic from a large number of streams. As an example, with ATM [3] we might have several AAL5 [4][5] frames that are being transmitted on different VCI/VPIs. We can build a simple FSM that can track the framing of an AAL5 frame. However if we try to do this for several streams at the same time, then combining the FSMs together will quickly give a machine with a very large number of states – it is far easier to have a separate FSM for each stream.

Assuming that this system is to be implemented with off the shelf components, we will be limited to the number of separate areas of CAM that we can use. If we assert that only one FSM can receive network data input events at any instance in time, then we can build multiple FSMs by having a single hardware implementation – but have numerous sets of state information for independent FSMs stored in RAM. For each FSM type, we will need a set of code words in CAM. Where there are several instances of the same FSM type, then these will often all be able to share the same set of code words – as the only state information is held in RAM.

Every instance of a FSM here is described as a **channel**, and each channel can have its own area of RAM in which its current state and other information may be stored. Each channel has a single FSM type, which is referred to as its **process**. Each process may have several channels. Any channel that needs to receive network input events will need to be selected as the current channel and its current state set from memory. When it has finished processing input events it will then be suspended awaiting its next invocation. The next state to operate in may need to be saved in its local memory.

3.1 Selecting the current channel

We need to specify next how a channel is selected. The simplest method is probably for the current channel to send an event to the channel that it would like to invoke. This could cause the new channel to become active and for the old channel to become inactive. The next state for the old channel could then be saved pending reception of another invocation event. Using this technique, we are able to have a number of processes, each dealing with different parts of a network packet. For example, separate processes could be provided to deal with each protocol layer.

To enable channels to send events to each other for various reasons, an event type can be included. This is specified in the send action and can become the current invocation event for the new channel. To enable the process of the new channel to test the value of the invocation event, it can be included as part of the search key for the CAM. Hence the new process could have a number of code words for its initial state, with different values for the invocation event. This can allow the process to take different actions in each case.

Following is an example of how we could select the required channel to handle a particular stream of data from the network. Some of the code words below expect the current invocation event to be *cell*, match various values of a network address field and have an action to select an associated channel. Other code words are used to catch various error conditions.

Key			Search Result	
State	Event	Data Pattern	Next state	Action
Lookup	Cell	1	Init	Send(Event <= NewPacket, Channel <= 101)
Lookup	Cell	2	Init	Send(Event <= NewPacket, Channel <= 102)
Lookup	Cell	3	Init	Send(Event <= NewPacket, Channel <= 103)
Lookup	Cell	X	LookupFail	
Lookup	X	X	Error	

(Note: X = Don't Care)

Here, the next state for the current channel is *Init*, unless the search for the network address fails, in which case we go to state *LookupFail*. If the current invocation event is not *cell* then we go to state *Error*. The next state for the new channel will be found in its local memory. It should be noted that in the three cases above, 'Send' would probably be invoking the same new process – although this does not need to be the case.

3.2 Hierarchical channels

With some simple additions to the system so far described, we can make this channel selection system hierarchical and thus apply this to channels at higher protocol layers – such as IP addresses and TCP port numbers. The only addition we require here is to enable the current channel to be used as part of the key for searching the content addressable memory; this typically won't be used for much of the time except when performing this hierarchical channel look up. Using this simple extension, it is still possible for multiple channels to share the same process – the only additional requirement being a table of address lookup code words for each channel it is managing.

As an example, we can rewrite the table of code words above, assuming that we may be entering this process on one of three different channels (1,2,3), each relating to a different low level network address.

Key				Search Result	
State	Event	Chan	Data	Next state	Action
Lookup	Cell	1	1	Init	Send(Event <= NewPacket, Channel <= 101)
Lookup	Cell	1	2	Init	Send(Event <= NewPacket, Channel <= 102)
Lookup	Cell	1	3	Init	Send(Event <= NewPacket, Channel <= 103)
Lookup	Cell	2	10	Init	Send(Event <= NewPacket, Channel <= 105)
Lookup	Cell	2	15	Init	Send(Event <= NewPacket, Channel <= 104)
Lookup	Cell	2	16	Init	Send(Event <= NewPacket, Channel <= 108)
Lookup	Cell	3	1	Init	Send(Event <= NewPacket, Channel <= 106)
Lookup	Cell	3	5	Init	Send(Event <= NewPacket, Channel <= 109)
Lookup	Cell	3	3	Init	Send(Event <= NewPacket, Channel <= 107)
Lookup	Cell	X	X	LookupFail	
Lookup	X	X	X	Error	

Each of the channels selected above relates to an address pair. This scheme may be extended to handle addresses for multiple protocol layers, the only disadvantage being the potential size of the lookup tables that we may have to use.

In some cases we have network addresses that are larger than the word size used by our system, or we may have multiple address fields – such as source and destination address – within the same protocol layer. In this case we may need to use this hierarchical address look up within a single process. To allow this, we could introduce a new variable to modify and use as part of the search key. However, for the sake of efficiency, this can be done by changing the value of the current channel number within the process as we move through stages of hierarchical address lookup. This will need to be done vary carefully as we are also changing the current context – it appears appropriate to refer to this as a *Goto*.

3.3 Procedure calls

In many cases, after a process has finished reading network input data we may wish to return to the channel from which we were invoked. This may be particularly the case with network protocols, where after handling one protocol layer we wish to return to processing the layer below. Although we may know the process we wish to return to, we may not know which channel was involved – we may of course be running either process for any number of different channels.

Another issue is that of fragmentation. Other than at the lowest level, we may only have addressing information in the first fragment that forms a packet at a higher level. As an example using ATM, in the 2nd cell of an IP frame we have no indication of the IP addresses to which the cell belongs. As there is no multiplexing of AAL5 frames on a single VCI/VPI, this cell must be the 2nd cell for the IP frame that began in the previous cell on this VPI/VCI. We need to have a general method such that one channel may re-invoke the channel that was previously handling data for it at the next protocol layer above – both for the current packet and for subsequent packets if there is a longer duration relationship. Unfortunately, this may not be the channel that we actually sent an event to last time, as the process itself may have changed channels as part of an address lookup.

The proposal here is to introduce a form of procedure call. The procedure call takes as parameters, the number of the channel being called and an event type. For the sake of clarity, we define here the calling channel to be A and the called channel B.

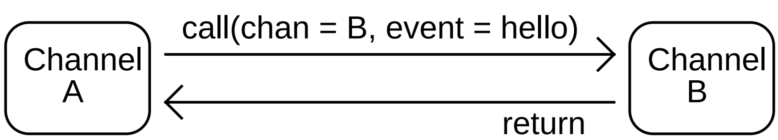

Figure 1 – Procedure call

During the procedure call, channel A's: current channel, current event and the next state are pushed onto a procedure call stack. The next state referred to is the next state for channel A – which is the state it will return to after the procedure call. The procedure call will take us to channel B and the initial state will be loaded from memory in channel B. After channel B has finished it issues a return action. This causes the next state, channel and event for channel A to be restored from the stack. Channel B's next state is saved in memory at channel B as a new initial state. Channel B's channel number is also returned to channel A as an 'uplink' and can be saved in memory. This last action gives us a link forward up to the last channel we have returned from.

The final operation introduced here is that of an up-call. This is similar to the standard call action, but allows the value of channel to be provided dynamically from the calling channel's local memory. This will typically use the value returned by the last procedure called from this channel – although a channel could hold several 'uplinks' if required.

3.3.1 Example of the use of channels and procedure calls

I look here at a simple program that counts the number of cells transferred between a group of three IP addresses – all carried over ATM on a given VPI/VCI. Here, the table entries for testing different addresses are all shown as part of the specification – in practice, we would probably generate these automatically. For the sake of simplicity, the code is shown as having random access into the contents of the cell. In a real implementation, the access to the cell contents might be implemented as sequential access as will be described later in Section 5.

The software operates on three protocol layers: ATM layer; AAL5 layer and IP layer. The first two layers each have a single channel that retains their current state. The IP layer has fourteen channels that relate to the various combinations of IP addresses and other channels that are not actually used to retain any significant state information but only as interim channels used as part of address decoding. Whilst performing this address decoding, the software uses a *goto* instruction to change the current channel number. This is to avoid the overhead of using the call instruction, which would require us to build up intermediate stages of return information on the stack.

The function of the software for each layer is briefly as follows:

- At the ATM layer, we call the AAL5 layer for cells that have the correct value of VPI and VCI. The invocation event is *streamf* for cells with the end of frame marker set and *stream* for the others.

- At the AAL5 layer, we call into the preamble stages of the IP layer for the first cell in the AAL5 frame and then make an up-call for the subsequent cells in the frame.

- In the IP layer, for the first cell of a packet, the preamble stages perform the selection of the appropriate channel for the given IP address pair and then generate an output event consisting of the current cell and the old value of *cellcount* for this IP address pair. Each channel maintains a counter of the number of cells that have been transferred between the given IP address pairs. Finally a call back channel is returned for the AAL5 layer to call for subsequent cells in this frame.

The program below consists of code fragments written in a pseudo code, based partly on VHDL syntax. For each process, cs represents the current state and ns represents the next state. Only one channel is active at a time and the process for that channel will determine how it changes state.

```
-- process for cell level, channel 1
cell: process( cs, data )
begin
    case cs is
        ATMCELL =>
            if (data.vpi = 1) and (data.vci = 15) and (data.pti /= Tail) then
                -- Cell
                ns <= ATMEND;
                call(chan <= 2, event <= stream);
            elsif (data.vpi = 1) and (data.vci = 15) then
                -- Last cell in an AAL5 frame
                ns <= ATMEND;
                call(chan <= 2, event <= streamf);
            else
                -- cell not on correct VPI/VCI
                ns <= ATMEND;
            end if;

        -- cell processing complete
        ATMEND =>
            Action(Get next cell);
            ns <= ATMCELL;

    end case;
end process;


-- process to decode AAL5 framing, channel 2
-- all cells on appropriate VPI/VCI
AAL: process( cs, data )
begin
    case cs is
        AAL5IDLE =>
            if (event = stream) then
                -- first cell of AAL5 frame
                ns <= AAL5IPEND;
                uplink := call(chan <= 3, event <= first);
            else
                -- first and last cell of AAL5 frame
                ns <= AAL5END
                uplink := call(chan <= 3, event <= first);
            end if;

        AAL5IPEND =>
            ns <= AAL5BODY;
            return;

        AAL5BODY =>
            if event = stream then
                ns <= AAL5IPEND;
                upcall(chan <= uplink, event <= body);
            else
                ns <= AAL5END;
                upcall(chan <= uplink, event <= body);
            end if;

        AAL5END =>
            ns <= AAL5IDLE;
            return;

    end case;
end process;
```

```
-- process to look up IP addresses and count cells
IP: process( cs, data )
begin
    case cs is
        -- first cell of an AAL5 frame, channel 3 (check IP source)
        IPSRC =>
            if event = first then
                if data.ipsrc = IP1 then
                    ns <= IPDST;  goto(chan <= 10);
                elsif data.ipsrc = IP2 then
                    ns <= IPDST;  goto(chan <= 20);
                elsif data.ipsrc = IP3 then
                    ns <= IPDST;  goto(chan <= 30);
                else
                    ns <= IPNONE; goto(chan <= 4);
                end if;
            else
                ns <= IPSRC;
                return;
        end if;

        -- not one of the IP address pairs we were looking for, channel 4
        IPNONE =>
           ns <= IPNONE;
           return;

        -- first cell of an AAL5 frame, channels 10, 20, 30 (Check IP destination)
        IPDST =>
            if (chan = 10) and (data.ipdst = IP1) then
                ns <= IPCELL; goto(chan <= 11);
            elsif (chan = 10) and (data.ipdst = IP2) then
                ns <= IPCELL; goto(chan <= 12);
            elsif (chan = 10) and (data.ipdst = IP3) then
                ns <= IPCELL; goto(chan <= 13);
            elsif (chan = 20) and (data.ipdst = IP1) then
                ns <= IPCELL; goto(chan <= 21);
            elsif (chan = 20) and (data.ipdst = IP2) then
                ns <= IPCELL; goto(chan <= 22);
            elsif (chan = 20) and (data.ipdst = IP3) then
                ns <= IPCELL; goto(chan <= 23);
            elsif (chan = 30) and (data.ipdst = IP1) then
                ns <= IPCELL; goto(chan <= 31);
            elsif (chan = 30) and (data.ipdst = IP2) then
                ns <= IPCELL; goto(chan <= 32);
            elsif (chan = 30) and (data.ipdst = IP3) then
                ns <= IPCELL; goto(chan <= 33);
            else
                ns <= IPNONE; goto(chan <= 4);
            end if;

        -- cell on a known IP address pair, chans 11, 12, 13,  21, 22, 23, 31, 32, 33
        IPCELL =>
            -- First cell, generate an output event with a copy of the cell and the old
            -- value of cellcount for this channel.
            OutputEvent(Currentcell, cellcount);
            ns <= IPCELL2;

        IPCELL2 =>
            cellcount := cellcount + 1;
            ns <= IPCELL2;
            return;

    end case;
end process;
```

3.4 Dynamic Channels.

A difficulty with the system proposed so far is that it requires all of the addresses relating to channels to be known in advance at specification time. This may be unrealistic, as we may not know what data is travelling on a network until we start monitoring it. The addresses may also be rather transitory. Web access for example is likely to produce multiple TCP connections to changing destinations. Because of this, the monitoring system itself needs to be more dynamic and change if required with network conditions.

At a low level, we may wish to just give a series of masks and collect all data that matches the masks. For example we could capture all IP frames to/from a given IP address. If we wish to try to investigate the operations of protocols such as TCP, we need to ensure that we have properly de-multiplexed the various streams – each of which may need to retain its own state information. To do this we need to be able to identify individual streams as they are created. We could do this by reporting new streams back to a control processor and allowing it to generate a new program. This however is likely to be slow and may risk losing the very data that we are trying to capture. An alternative method is to allow the program to create new channels dynamically, along with new CAM entries to invoke them. Such a system that allows a program to modify itself will obviously need to be well managed.

3.4.1 Creating a new channel

To create a new channel dynamically, we need a certain amount of information about the channel type required. Primarily, this is the process that the channel executes and also how and when it should be invoked. The latter details are the concern of the calling process and this will need to provide some form of template for the code word to perform the invocation. The code word template will need to be updated with information concerning the state, channel, event and data to use for the search fields and also any details such as the new channel number for the code word result. This new code word will need to be written into the content addressable memory in a position that gives it the correct priority with respect to other code words for that state. In terms of instantiating a new channel, we will need to select an unused channel [number] and update this to point to the required process. The new channel can then be invoked and required to initialize itself.

3.4.2 Simple channel allocation system

As an initial study, I cover here a simple dynamic channel system that has a fixed number of channels allocated for each process and a set of pre-allocated static entries created to invoke these from the calling process.

The method proposed here is that processes may have a number of *place holder* entries for code words that are already associated with pre-allocated channels and contain appropriate actions – but are flagged in such a way that the CAM will ignore these during a standard search. If the search for a data value (or channel) fails, then the final entry for the given state can be an *allocate* instruction. This can search for one of the appropriate spare entries with the same value for state and update this with the values used for the current search. The entry can then be flagged as active. Any bits originally specified as don't care can be left unchanged – thus allowing the template to determine which fields will be used in a search. The action associated with the new entry can then be executed as if it had been found during the original search – this is likely to be a *Call* or *Goto* instruction.

To enable this system to operate for a period of time, we will need to free channels that are no longer required and de-allocate their invocation code word. A special return instruction could be implemented that performs a return from the current channel and also de-allocates the code word that took us into that channel. In practice, the major problem may be how to know *when* channels should be de-allocated. The whole topic of garbage collection needs further work and I plan to cover this in a later paper.

3.5 Summary

As can be seen, the introduction of channels and procedure calls has made quite a departure from a standard FSM and moved more towards a programming style of implementation. It is for this reason that I use the term **process** to refer to the FSM.

The system proposed has a number of special actions that control the scheduling of processes. These may need to be altered and extended as a result of experience. Those proposed so far are:

- Send(event, channel) Send event to channel and reschedule
- Call(event, channel) As send, but wait for return and reschedule calling process
- Return Return to calling process
- Upcall(event, *channel) As call, but use run-time value for channel number from memory
- Goto(channel) Change channel within a process. (Next state refers to **new** channel.)
- Allocate Create a new code word and execute it
- Dreturn As return, but de-allocate the code word that invokes the process

As well as the above, there will also need to be other actions that can be performed to enable the implementation of arithmetic operations and generation of output events. This is covered in the next section.

4. Actions

The previous section looked at examples of actions concerned with process scheduling, however, a number of different types of action could be associated with each code word such as given below.

- Scheduling
- Data flow
- Arithmetic operations
- Output event generation

The actions controlling data flow are required to determine which data items from the packet are used for searches. Some systems might not allow any control over this at all, whereas others may enable specific data items to be used. This topic is covered in more detail in the section 5.

4.1 Arithmetic Operations

It is likely that processes may wish to perform simple arithmetic operations, such as addition and subtraction – if only to enable counters to be implemented. In the last section, it was proposed that each channel should have an area of local memory in which to store information specific to that channel, such as current state and other variables. In addition to this, it may also be useful to have some global memory that is accessible by all channels. It is proposed here that a simple three-address architecture processor instruction could be used as an action and that this should have access to both the global memory and also the memory local to the current channel. A three-address instruction is proposed to allow most simple operations to be performed in a single action. This should allow as a minimum: add, subtract and move operations. Some provision should also be provided for immediate values to be used. To ensure that it is possible to fit such an instruction into the CAM result field, it may need to share a single immediate field with other actions and it may need to keep the memory address fields used relatively small.

In addition to simple arithmetic operations, there are likely to be functions that are specific to the network or communication protocol – such as calculating a CRC. It is envisaged that special hardware would be provided to assist with the performance of these functions, probably operating in parallel with other processor activity.

4.2 Output event generation

In his dissertation, Richards [6] uses event messages for the input and output of his FSMs. These event messages contain a length, type, timestamp, sequence number and an optional data field. At the lowest level he uses these to carry packets of data from the network to an initial FSM. Output from a FSM is also an event message and these can be inputs to other FSMs. The type field specifies the event being generated and also identifies how the data field should be interpreted.

For the inter-FSM communication within the processor described here, events are also used for scheduling – although in this case all the event consists of is an event type and a channel to which it is sent. Any of these FSMs could potentially wish to send event messages to other *external* FSMs – this might consist of multiple event messages from multiple channels for a single received network packet. These *external* event messages could potentially contain data from one or more packet, statistical information or just notification of an event.

In any hardware implementation, if we wish to operate in real time then we may need to restrict the complexity of the events we are able to pass around the system. The type of output events actually supported by a particular system is likely to depend on the amount of resources available and the likely performance overheads involved. There are a wide variety of possibilities and I identify a few example solutions below with their pros and cons.

1. Any number of processes can generate event(s) of any format for a single packet.

   Perhaps an ideal solution, however we need to consider the overhead in creating the event message and the data paths involved in getting the messages to an output port.

2. Any process can generate short fixed length event messages consisting of a few words of information, plus one process is able to generate a data event message that includes a sequence of words from the current packet.

   Possible compromise. The short event messages could be generated sequentially by any process that wishes to create them – if we have a resource conflict then it should be permissible to wait for a few clock cycles for the previous message to complete. The data event message could be created by separate hardware that operates autonomously.

3. Single output event message consisting of the current data packet, plus a fixed number words of data.

   Allow any process to decide whether we wish to generate an output event. Simple to implement in hardware as this may be implemented as part of the flow of network data through the system. May be common to keep entire packets of data if we are filtering out packets to be processed in more detail elsewhere. However, the disadvantage is the requirement to output a whole packet, even if we only wish to send a short event message containing little or no data from the network.

Of the above, solution 2 looks like a good compromise as it gives autonomy to individual processes to be able to generate events. For a first implementation, solution 3 could be a possibility if resources were limited.

4.3 Summary

There are a number of actions which we may wish to specify as part of the result from the CAM search. These can consist of control over scheduling, network data flow, arithmetic operations and generation of output events. In any implementation there may be restrictions over the combinations of these that may be specified as a single action. There may be resource conflicts over use of memory for example, and because of this it might be appropriate to allow a single "instruction" that specifies one of a number of possible tasks including a scheduling or an arithmetic operation. Depending on the word size of the result from CAM, there may also need to be some compromise over the size and number of separate fields used to specify the Action. This may imply that fields may need to be shared between separate operations.

The CAM result word is likely to be structured in a similar way to a micro-instruction. In the first instance, micro-programming tools could be used for software development – although in the longer term, custom compilation or synthesis tools will be required.

5. Network Data

There are two basic methods that we can use to access the content of a packet to provide words of data to use in CAM search operations. We can provide random access into the body of the packet or we can provide sequential access. The choice of the access method may depend on the programming environment required and also on the resources that are available for the hardware design. The two methods of operation are outlined below.

5.1 Sequential Access

This is a common method of handling network data in hardware. The network data flows into the system via an input port and is only accessible by the processor in sequence. To avoid the processor needing to operate at exactly the same clock rate that the data arrives from the network, we may need to use some FIFO buffering to provide input queues. If the processor is fast compared to the network, then this means that the processor may examine each data item many times if required. We may also be able to choose which words from memory we wish to use – i.e. skipping those that we are not interested in. The action that is generally not available is that of rewinding the data stream and looking at a previous data item that has already passed.

One of the benefits of using sequential access is that we do not need to be able to store the whole of the packet that we are processing – storage can be a problem with some combinations of network and hardware implementation.

In terms of programming, sequential access can be difficult to use. We can move forward through the data using positive relative movements, although this requires us to know where we currently are. In general this probably requires us to always enter a process pointing at a particular word of data. As an alternative we could use an absolute movement – to go to a particular position within the packet.

5.2 Random access

In this case, the whole packet is stored in memory and may be accessed by the processor in any sequence. This can make programming a lot easier. However in some circumstances a process may not know where its data begins within the packet – e.g. if lower protocol layers have variable amounts of header information. In these cases it may be necessary to provide an index register that points at the start of data for the current process. The index register may need to be updated on a procedure call and the old value held on a stack awaiting a return.

The disadvantage of providing random access is that we need to have enough memory within or closely coupled to the processor to be able to hold any packet received. This may not be a problem with networks such as ATM, because of the small size of the cell. However with other networks it may be possible to receive packets with a variable length and with a large maximum packet size. Ideally, we would like to store the packet within the processor. However for some hardware implementations – such as with some FPGAs – the maximum packet size may be too high to allow this. Newer generations of FPGA are being produced with significantly larger amounts of memory, so this may be less of a problem now than it was in the past.

Another problem is that our memory needs to be multi-port. Access to the memory is required by the processor itself and also for network data input and output event generation. In practice a dual ported memory may be suitable by using one port for the processor and one to share between network data input and output event generation. The packet memory is likely to need to hold at least three packets – one for input of the next packet, one for the current packet being processed and one for the previous packet in case it is (still) required for output event generation. The memory can be managed easily by splitting this into pages, each of which can hold a maximum size packet.

5.3 Word size and alignment

To provide a high throughput, it is advantageous to use a relatively large word size for the network data – such as 32 bits wide – and also a high clock speed. This may run at a higher throughput than the data input from the network and hence require input FIFO buffering to provide rate adaptation and also to change the word size if this is required. Ideally, it would be useful to provide the input buffering within the processor itself to avoid external fast, wide data paths.

A problem with using a large data word for the network data is that depending on the format of the packet, words of data within the packet may not fall on the word boundaries introduced at the input FIFOs. In general, protocol designers try to avoid this problem as it can also generate problems for programmers writing protocol software. However to ensure we are not caught out by this problem, we may need to allow the data to be accessed on any byte boundary – this may require us to read two words of data and use a barrel shifter to extract the required word of data.

5.4 Caching

To improve performance, it may be necessary to provide a caching system to enable the required word of network data to be accessed quickly. This is particularly the case with sequential access systems as there may be delays in moving forward along the stream of data. A simple method of caching that we can provide here is to cache the current item being accessed and a number of subsequent words in the data stream. As the program moves forward along the data stream, we can discard old items from the cache and fill with new items from the network. This should enable us to access a number of different words directly that we are likely to require next. If we assume that the packet usually keeps to the same word alignment, then we could cache only words at the current word alignment – although this means there will always be a cache miss if the program moves forward to a different word alignment.

6. Implementation Issues

From an engineering viewpoint, designs are only *really* interesting if you can build them ! I look here at some of the practical aspects of this type of processor architecture and argue that it should be possible to build a practical system such as discussed in this paper using components that are currently available or which should be available shortly.

6.1 CAMs, keys and actions

A significant change recently has been the availability of fast, moderately large, ternary CAMs. One such component is the NL85721 from Netlogic [7], which at the time of writing is available as samples. This CAM is 8K x 128 bits and has a 64-bit bus to carry the key or results of a search. The 128 bit word can be partitioned as required, however for best performance we can use this with a key and result field each <=64 bits – such that we can carry the data over the bus in a single clock cycle. This CAM will run at clock rates of up to 66 MHz, although due to pipelining it takes a few clock cycles between searching with the key and receiving the associated result.

Using a 64-bit key, we can have a 32-bit data key and still have 32 bits to divide between state, channel and event. If we use 12 bits for each of state and channel, then we can define some very large FSMs and also up to 4096 channels. 8K words relates to a lot of FSM, however the space is used up quite quickly if we wish to have a large number of channels – as we need at least one code word for each channel we wish to invoke.

With a 64 bit result, we can specify new values for state, channel and event. We should also be able to specify a number of different possible actions although possibly not all combinations together. A logical split may be to have an instruction that specifies a scheduling, arithmetic or output event operation and then to have separate control over data flow.

6.2 Conventional Memory

Conventional memory can be implemented using standard synchronous static memory components. These are readily available that operate at clock rates of 100 MHz or even higher. If it was acceptable to have a small amount of memory for each channel, then a basic system could be constructed that used a single 64K x 32 bit memory chip. This could provide – for example – 16 words of local memory for each of 4095 channels + 16 words of global memory.

6.3 Control and actions

Control of the CAM search and execution of action instructions can be implemented using a conventional micro-architecture, with micro-code control store, mapping table and sequencer etc. This could control the operation of most of the processor, except for any autonomous activities such as network data flow and any output event generation based on packet contents.

Arithmetic functions can be implemented using a conventional ALU and multiplexor structure with access to the external memory. Scheduling is closely tied in with key generation and this could be easily implemented as a series of key registers [state, channel, event, data], a local stack and access to external memory. Key registers could be updated as required from a number of possible sources including memory, stack or CAM result. For good performance, the data paths for scheduling should be separate from those used for arithmetic operations. This will not however be the case for access to memory unless this is kept in a separate memory space. The data key will be updated primarily from network data.

6.4 Network data and Output events

The implementation of the network data system will depend on whether this is via random or sequential access. A system using random access would probably be quite straightforward to design but would require implementation hardware that can provide large enough multi-port memory. A system that uses sequential access would require less resources but would ideally be enhanced with a caching mechanism which may not use many resources but will probably require quite a lot of logic to control the cache and transport the network data.

6.5 Implementation and resources

Apart from the external RAM and CAM, it should be possible to implement the remainder of a simple system within a single large FPGA. The choice of whether to use random or sequential access will depend on whether the FPGA technology used is able to provide large areas of multi-port memory. Alternatively, a sequential access system could be implemented albeit with a relatively small cache. The cache control logic should not consume much FPGA resources, although it could use a lot of 'design' time. The implementation of ALU and scheduling should not use too much resource despite the potentially wide bus widths used. The control system is generally light on resources, apart from the memory used for the micro-code control store – again this is resource hungry unless specifically supported by the FPGA. Implementation of output event generation will need to be combined with the network data system and should be straightforward.

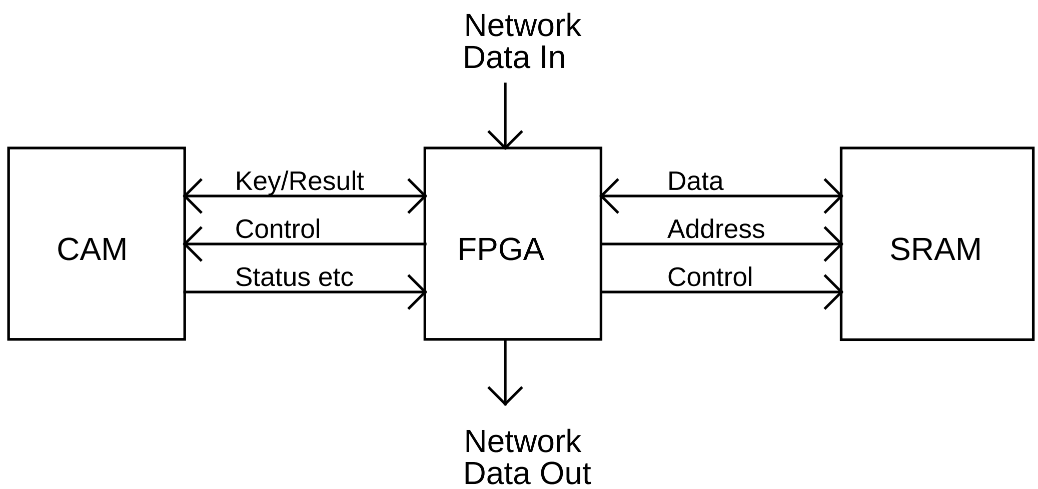

Figure 2 – Simple Associative Processor for Network Monitoring

In conclusion, it should be possible to build a simple system with three components: an FPGA, a ternary CAM and a static RAM. Compromises may need to be made depending on the type of FPGA used –

particularly concerning whether large areas of RAM can be synthesized. A simple design should be able to fit within one of the commonly used FPGAs so long as it can provide small areas of RAM for queues and stacks. If a state-of-the-art FPGA is used such as the Xilinx Virtex [8] series, then it should be possible to implement a more sophisticated system, using random access to network data and providing relatively general output event generation.

7. Conclusions

This paper looks at architectural proposals for a network traffic monitoring system that uses associative processing techniques. A method is presented that allows multiple FSMs to be implemented as processes that execute out of a single content addressable memory component. A process instance is referred to as a channel, and this retains its state in RAM between invocations. Only one channel can be active at any time, and channels can invoke each other using a variety of methods. This paper then looks at the types of actions that a FSM may perform and the various methods in which the FSM could access the network data. Finally, the practical issues of implementation are investigated.

A conclusion that is drawn from section six, is that it should be possible to build a practical system of the type discussed here using a small number of components. In terms of instruction execution time, this type of processor is slower than state-of-the-art von-neuman architectures. The associative processor however has an advantage when we wish to perform multiple comparisons – such as testing for network addresses or multiple possible values of packet formats – as we can perform these comparisons in parallel. Finally, the speeds of processors and conventional RAM have increased over time; it appears that this is currently happening with CAM components.

7.1 Further work

The next stage of this work is to build a model of a hardware implementation of such a processor and to simulate the design. This can then be tested with a variety of simulated input data and monitoring 'programs' to evaluate the performance. From this work, it should be possible to identify potential problems and any performance bottlenecks. It would be interesting to compare of the performance and complexity of the various algorithms discussed in this paper to determine those which are the most suitable. Finally here, there needs to be more work on management of dynamic channels and the associated problems of termination and garbage collection.

A second stage can be to synthesize logic for a simulated processor and hence evaluate the amount of logic required for the design and the possible performance in terms of overall clock rate. This will inevitably lead to modifications in the model to improve performance. This will also aid choice of appropriate algorithms as noted above. Finally here, prototype hardware could be constructed and tested with real network traffic.

For such a system to be useful in the field, there will need to be suitable tools provided to enable a user to write monitor 'programs' in a high level language and to compile or synthesize these into the tables of code words required by the processor.

References

[1] Gerald Tripp, Real Time Network Traffic Monitoring, Technical Report: 5-99, Computing Laboratory, University of Kent, May 1999.

[2] K.E.Grosspietsch, Associative Processors and Memories: A Survey. IEEE Micro Vol. 12, No. 3, June 1992, pp. 12-19.

[3] M. DePrycker, Asynchronous Transfer Mode: Solution for Broadband ISDN, 2nd Edition, Ellis Horwood, 1993.

[4] ITU-T Recommendation I.362, BISDN ATM Adaptation Layer (AAL) Functional Description, 1993.

[5] ITU-T Recommendation I.363, BISDN ATM Adaptation Layer (AAL) Specification, 1993

[6] S.G.Richards, The Use of State Machine Technology in Broadband Network Monitoring, Dissertation submitted for the degree of M.Sc. in Distributed Systems. University of Kent. 1996.

[7] IPCAM-2, NL85721, Ternary Content Addressable Memory (IPCAM™) 8K $\times$ 128 Advanced Information, Revision 2.1. Netlogic Microsystems.

[8] Xilinx Virtex™ 2.5V Field Programmable Gate Arrays. Advance Product Specification. February 16, 1999 (Version 1.3). Xilinx, Inc.