Optimising the resource utilisation in high-speed network intrusion detection systems.

Gerald Tripp

www.kent.ac.uk
Network intrusion detection

- Network intrusion detection systems are provided to detect the presence of various security attacks.
  - This could be a virus or an attack that takes advantage of some form of weakness in the system.

- Typically operates by searching for various patterns or strings within each network packet.

- Difficult for software to keep up with traffic rate for high speed networks.
  - Can build custom hardware for this within a Field Programmable Gate Array (FPGA)
  - Implement string matching using an 'automata' based design
FPGA based implementation

• One common method is to implement an automata as a series of comparator, flip flops and gates.
  • Good resource utilisation
  • But: need to rebuild the design if we change the search strings

• Can use a table based automata implementation.
  • Dynamically update-able at run time.
  • Use internal memory to avoid pipeline delays to external RAM
  • But: limited numbers of Block RAM primitives within FPGAs.
Using Logic ... 

- We can instead use logic cells (LUTs – Look Up Tables) as small blocks of memory.
  - But: they are rather small (16-bits each in Xilinx FPGAs)
  - However, there are plenty of them ...
  - The basic single LUT memory is also single port ...

- We can however use these as shift registers ...
  - SR16 primitive – implemented as a single LUT.
  - Use the shift data operation to load them with information
  - Use a selective shift out port to read out particular bits
Standard shift register: SR16

- The 'programmable length' facility enables Q to output the shift register bit selected by A
Generic Memory Block of size: \(2^N \times W\) bits

- Instantiate number of shift registers as required.
- Link shift registers together inside the memory block.
- Serial load data and enable, in and out
  - Daisy chain to link memory blocks together for loading.

![Diagram of memory block](image)

**UNIVERSITY OF KENT**
Basic string matching engine.

Generic design:

- A – bus width of compressed input
- S – number of bits in state variable
- N – number of different match strings
Basic Matching 'engine'.

- Use a compacted table for the automata based on “row displacement with state marking”
  - This is a traditional parser technique.
  - New variable P: address bus width into main automata table

- Use a similar technique for the compression system.
  - Variable CL: address bus width into main compression table

- Build state decoder as two stages:
  - First: compress current state into a value (width K bits) indicating one of the terminal states or that it's a non-terminal.
  - Secondly: decode this into a match vector.
Determining resource utilisation

- Now have a completely parametrised design ...
  - Build a rough (mathematical) model of resource utilisation for an 'matching engine' dependent on these parameters.

- For each valid set of parameters:
  - Process a set of Intrusion Detection rules to see how many 'engines' are needed
  - Determine the approximate per search byte resource utilisation.

- Pick the most likely candidates and plot a graph
  - Pick an optimal candidate and build an FPGA design for it ...
Search engine resource utilisation

Parameter key:
CL,A,S,P

\[ K = \left\lfloor \log_2 (N + 1) \right\rfloor \]
Results – Target device: Xilinx XC2VP7-7

Parameters: CL=7, A=6, S=7, P=7, K=4, N=15
  • Max. of 15 search strings / engine (average of about 10 to 11)

• Resources for each engine:
  • LUTs: 410 out of 9856 (4% of XC2VP7)
  • Can probably fit about 215 engines into a larger XC2VP100
    • i.e. Search for around 2200 strings in parallel

• Search rate: 1.2 Gbps
  • Independent of search strings or input data.

• Tested by simulation as a VHDL model …
Conclusions & Further work

• Flexible VHDL model for a string matching system
  • Using just FPGA LUTs.
  • Dynamically update-able at run time.

• Only byte at a time string matching so far
  • Can look at incorporating in existing work with multi-byte input matching systems and regular expression matching.

• This design uses just LUT primitives ...
  • Can look at how this might be used in conjunction with the larger BRAM primitives for more optimal implementations.