ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

# Profiling and Synthesis of Leaky Bucket Algorithm for Network Processor

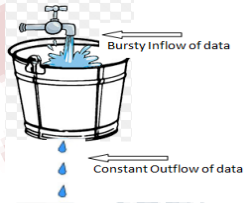
[1] Neha Jain, [2] Dr. M.K. Jain
[1][2] Department of Computer Science, Mohanlal Sukhadia University
Udaipur, Rajasthan 313001, India

***Abstract* - Leaky Bucket algorithm is used in packet switched networks for traffic shaping of data transmissions. In this paper we gather the profiling data of software implementation of the algorithm. Memory consumption of software algorithm is very high. Also the software implementation is not optimized. To implement the code we use hardware description language like VHDL. Hardware implementation of the algorithm is taking only 197520 kilobytes of memory. Only 2138 Slice registers are used out of 12490. Only 4129 Slice LUTs are used out of 12490. Only 25 bonded IOBs are used out of 172. Thus the devise utilization is 17%, 33%, 14% respectively.**

***Keywords* — Leaky Bucket, traffic shaping, network processor, profiling, Valgrind tool, Massif Visualizer.**

## 1. INTRODUCTION

Leaky bucket algorithm is used for traffic shaping in data transmissions. Traffic shaping means to regulate the data flow. It is a method of congestion control. Leaky Bucket algorithm is used to control the data rate in data transmission. Here bucket refers to buffer. If bucket or buffer overflows then the packets are discarded. Packets entered in the buffer in different – different rates. But output rate remains constant. By averaging the data rate this algorithm converts the bursty traffic into constant rate traffic.




In [1] authors studied Leaky Bucket algorithm with loss priorities. In [2] authors published that required bucket size increased linearly with the average number of bits generated during an on period, and increase logarithmically with the decrease in loss or mark probability. In [3] authors proposed Leaky Bucket algorithms. Author suggested these algorithms for sustainable-cell-rate usage parameter control in ATM networks. One is the fuzzy leaky bucket algorithm, and the other is the neural fuzzy leaky bucket algorithm. In [4] author proposed Buffered Leaky Bucket algorithm, for ATM networks. In [6] author proposed a Network processors (NPs) for active networks (AN). In [7] author studied the effect of concurrency in network processors on packet ordering In this paper to dynamically analyze (profiling) the algorithm we use Valgrind tool. Using this tool we can easily calculate space and time complexity of any algorithm. In Section 2 profile data is shown which is generated by using KCachegrind visualization tool. We also used Massif Visualizer to view the memory consumption. After generating profiling data we try to reduce space and time complexity of the program. For the purpose we switched to hardware platform. Section 3 shows the hardware implementation of the algorithm. We develop the program in Xilinx ISE using Vertex 5 board. Section 4 and section 5 shows the result and conclusion respectively.

## 2. PROFILING OF THE LEAKY BUCKET ALGORITHM

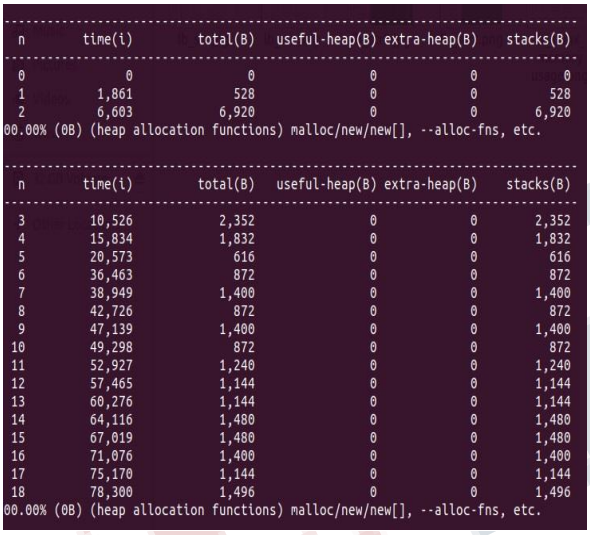
After compilation of the program we run Massif to collect the profiling information, and then we run ms_print command to present the results in a readable form.


***Fig. 1. Memory consumption occurred as the program executed***

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

In fig. 1 the graph shows how memory consumption occurred as the program executed. In the above graph each vertical bar represents a measurement of the memory usage at a certain point in time. Normal snapshots taken by Massif are represented in the graph by using ':' character bars. '@' character is used to represent detailed snapshots. In above graph total 75 snapshots have been taken. Out of which 7 snapshots [2(peak), 18, 26, 30, 38, 55, 65] are detailed snapshots. '#' character is used to represent the peak snapshot in the graph. In following graph 2nd snapshot is the peak snapshot. Peak snapshot shows that memory consumption was very high at this point.



| n | time(i) | total(B) | useful-heap(B) | extra-heap(B) | stacks(B) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1,861 | 528 | 0 | 0 | 528 |
| 2 | 6,603 | 6,920 | 0 | 0 | 6,920 |

00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

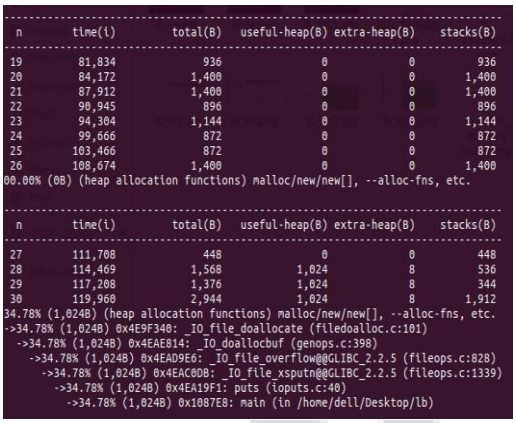
| n | time(i) | total(B) | useful-heap(B) | extra-heap(B) | stacks(B) |
|---|---|---|---|---|---|
| 3 | 10,526 | 2,352 | 0 | 0 | 2,352 |
| 4 | 15,834 | 1,832 | 0 | 0 | 1,832 |
| 5 | 20,573 | 616 | 0 | 0 | 616 |
| 6 | 36,463 | 872 | 0 | 0 | 872 |
| 7 | 38,949 | 1,400 | 0 | 0 | 1,400 |
| 8 | 42,726 | 872 | 0 | 0 | 872 |
| 9 | 47,139 | 1,400 | 0 | 0 | 1,400 |
| 10 | 49,298 | 872 | 0 | 0 | 872 |
| 11 | 52,927 | 1,240 | 0 | 0 | 1,240 |
| 12 | 57,465 | 1,144 | 0 | 0 | 1,144 |
| 13 | 60,276 | 1,144 | 0 | 0 | 1,144 |
| 14 | 64,116 | 1,480 | 0 | 0 | 1,480 |
| 15 | 67,019 | 1,480 | 0 | 0 | 1,480 |
| 16 | 71,076 | 1,400 | 0 | 0 | 1,400 |
| 17 | 75,170 | 1,144 | 0 | 0 | 1,144 |
| 18 | 78,300 | 1,496 | 0 | 0 | 1,496 |

00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

***Fig. 2. Snapshot with Normal Information Recorded***

Fig. 2 shows the information taken on various snapshot. These snapshots are normal, so only a small amount of information is recorded for them.

In fig. 3 we can also see an allocation tree which indicates exactly which pieces of code is responsible for allocating heap memory. We read allocation tree from top to down. This allocation tree shows that 1,024B of useful heap memory has been allocated and arrows show that this is from different code location. The '->' indicates that _IO_file_doallocate called malloc function. _IO_doallocbuf called _IO_file_doallocate. _IO_doallocbuf is responsible for 1024B. Thus we can see at this point every allocation so far is due to main. In this allocation tree we can also see the time taken in every point on which snapshot is taken. This time is measured in millisecond.



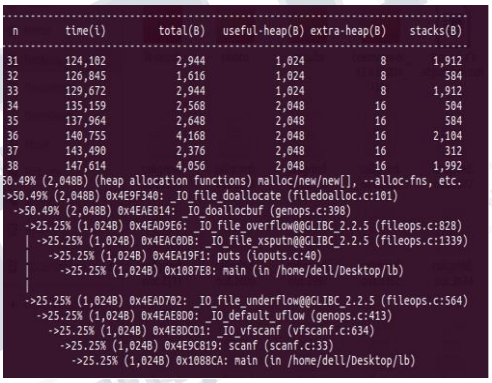
| n | time(i) | total(B) | useful-heap(B) | extra-heap(B) | stacks(B) |
|---|---|---|---|---|---|
| 19 | 81,834 | 936 | 0 | 0 | 936 |
| 20 | 84,172 | 1,400 | 0 | 0 | 1,400 |
| 21 | 87,912 | 1,400 | 0 | 0 | 1,400 |
| 22 | 90,945 | 896 | 0 | 0 | 896 |
| 23 | 94,304 | 1,144 | 0 | 0 | 1,144 |
| 24 | 99,666 | 872 | 0 | 0 | 872 |
| 25 | 103,466 | 872 | 0 | 0 | 872 |
| 26 | 108,674 | 1,400 | 0 | 0 | 1,400 |

00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

| n | time(i) | total(B) | useful-heap(B) | extra-heap(B) | stacks(B) |
|---|---|---|---|---|---|
| 27 | 111,708 | 448 | 0 | 0 | 448 |
| 28 | 114,469 | 1,568 | 1,024 | 8 | 536 |
| 29 | 117,208 | 1,376 | 1,024 | 8 | 344 |
| 30 | 119,960 | 2,944 | 1,024 | 8 | 1,912 |

34.78% (1,024B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->34.78% (1,024B) 0x4E9F340: _IO_file_doallocate (filedoalloc.c:101)
->34.78% (1,024B) 0x4EAE814: _IO_doallocbuf (genops.c:398)
->34.78% (1,024B) 0x4EAD9E6: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:828)
->34.78% (1,024B) 0x4EAC0DB: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1339)
->34.78% (1,024B) 0x4EA19F1: puts (ioputs.c:40)
->34.78% (1,024B) 0x1087E8: main (in /home/dell/Desktop/lb)

***Fig. 3 Snapshot with Detail Information Recorded***



| n | time(i) | total(B) | useful-heap(B) | extra-heap(B) | stacks(B) |
|---|---|---|---|---|---|
| 31 | 124,102 | 2,944 | 1,024 | 8 | 1,912 |
| 32 | 126,845 | 1,616 | 1,024 | 8 | 584 |
| 33 | 129,672 | 2,944 | 1,024 | 8 | 1,912 |
| 34 | 135,159 | 2,568 | 2,048 | 16 | 504 |
| 35 | 137,964 | 2,648 | 2,048 | 16 | 584 |
| 36 | 140,755 | 4,168 | 2,048 | 16 | 2,104 |
| 37 | 143,490 | 2,376 | 2,048 | 16 | 312 |
| 38 | 147,614 | 4,056 | 2,048 | 16 | 1,992 |

50.49% (2,048B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->50.49% (2,048B) 0x4E9F340: _IO_file_doallocate (filedoalloc.c:101)
->50.49% (2,048B) 0x4EAE814: _IO_doallocbuf (genops.c:398)
->25.25% (1,024B) 0x4EAD9E6: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:828)
| ->25.25% (1,024B) 0x4EAC0DB: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1339)
| ->25.25% (1,024B) 0x4EA19F1: puts (ioputs.c:40)
| ->25.25% (1,024B) 0x1087E8: main (in /home/dell/Desktop/lb)
|
->25.25% (1,024B) 0x4EAD702: _IO_file_underflow@@GLIBC_2.2.5 (fileops.c:564)
->25.25% (1,024B) 0x4EAE8D0: _IO_default_uflow (genops.c:413)
->25.25% (1,024B) 0x4E8DCD1: _IO_vfscanf (vfscanf.c:634)
->25.25% (1,024B) 0x4E9C819: scanf (scanf.c:33)
->25.25% (1,024B) 0x1088CA: main (in /home/dell/Desktop/lb)

***Fig. 4 Snapshot with Detail Information Recorded***

Tree allocation of fig. 4 shows that 2,048B has been allocated. _IO_file_overflowGLIBC_2.2.5 is responsible for 2,048 B. The lines and arrows indicate that this is also from two different code locations line no 1339 in file fileops.c is responsible for 1,024B and line no 564 in fileops.c is responsible for other 1,024B.

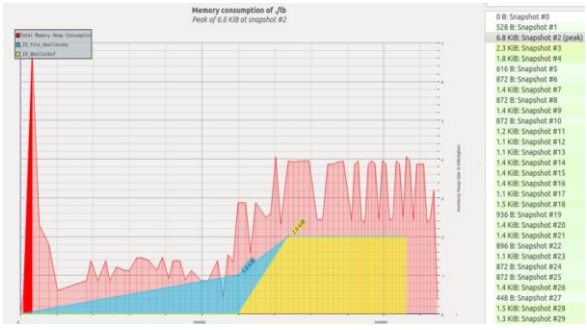



***Fig. 5 Memory Consumption Graph***

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

Graph showed in fig. 5 is generated by using massif Visualizer. It shows the total memory consumption in each snapshot during the execution of the program. In fig. 6 (A) we can see that Inclusive cost for main () is 62.61 and self cost is .45. Where the inclusive cost is high and self cost is low then we have to optimize that function. Similarly in fig. 6 (B) inclusive cost is 30.08 and self cost is 2.30. We have to also optimize this function.

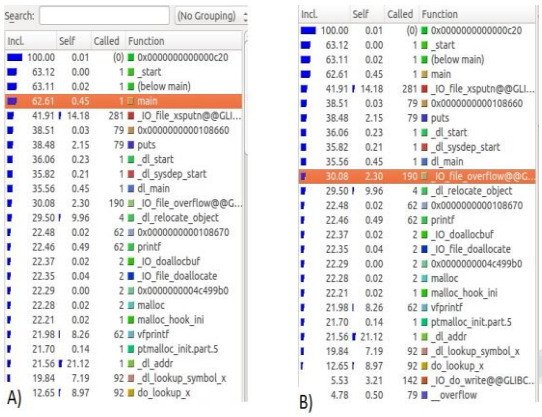



***Fig. 6 Inclusive and self cost of functions***

By using above profiling data we conclude that when we execute the algorithm in C environment time and space complexity is very high. To efficiently use this algorithm in network processor we implement this algorithm in hardware platform using Xilinx ISE Vertex 5 board.

## 3. IMPLEMENTATION OF CODE IN XILINX ISE

The following code shows that how data is generated on different-different data rates.

3.1 Clk_divider.vhd

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
-- ENTITY DESCRIPTION
entity CLK_DIVIDER is
Port (
CLK      : in STD_LOGIC;
RST      : in STD_LOGIC;
EN       : in STD_LOGIC;
INDataRate: in STD_LOGIC_VECTOR (2 downto 0);
OUTDataRate: in STD_LOGIC_VECTOR (2 downto 0);
WriteEn : out STD_LOGIC;
ReadEn : out STD_LOGIC ); end CLK_DIVIDER;
 -- ARCHITECTURE DESCRIPTION
architecture Behavioral of CLK_DIVIDER is
signal WriteEnable : STD_LOGIC;
signal ReadEnable : STD_LOGIC;
begin
clk_divider_process : process (CLK)
variable MaxCountValue_WR : natural range 0 to 1000000 -1;
variable FreeRunningCount_WR : natural range 0 to 1000000 - 1;
variable MaxCountValue_RD : natural range 0 to 1000000 - 1;
variable FreeRunningCount_RD : natural range 0 to 1000000 – 1;
begin
if rising_edge(CLK) then
if RST = '1' then
   FreeRunningCount_WR := 0;
   FreeRunningCount_RD := 0;
   MaxCountValue_WR := 500;
   MaxCountValue_RD := 500;
   WriteEnable <= '0';
   ReadEnable <= '0';
else
if EN = '1' then
case (INDataRate) is
    when "000" => MaxCountValue_WR := 500;
    when "001" => MaxCountValue_WR := 250;
    when "010" => MaxCountValue_WR := 167
    when "011" => MaxCountValue_WR := 125;
    when "100" => MaxCountValue_WR := 100;
    when "101" => MaxCountValue_WR := 84;
    when "110" => MaxCountValue_WR := 72
    when "111" => MaxCountValue_WR := 63;
    when others => MaxCountValue_WR := 500;
end case;
case (OUTDataRate) is
    when "000" => MaxCountValue_RD := 500;
    when "001" => MaxCountValue_RD := 250;
    when "010" => MaxCountValue_RD := 167;
    when "011" => MaxCountValue_RD := 125;
    when "100" => MaxCountValue_RD := 100;
    when "101" => MaxCountValue_RD := 84;
    when "110" => MaxCountValue_RD := 72;
    when "111" => MaxCountValue_RD := 63;
    when others => MaxCountValue_RD := 500;
end case;
     FreeRunningCount_WR := MaxCountValue_WR;
 end if;
if FreeRunningCount_WR = 0 then
    FreeRunningCount_WR := MaxCountValue_WR;
    WriteEnable <= '1';
```

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

```
else
    FreeRunningCount_WR := FreeRunningCount_WR - 1;
    WriteEnable <= '0';
end if
if FreeRunningCount_RD = 0 then
     FreeRunningCount_RD := MaxCountValue_RD;
     ReadEnable <= '1';
else
      FreeRunningCount_RD := FreeRunningCount_RD - 1;
      ReadEnable <= '0';
end if;
end if;
end process;
    WriteEn <= WriteEnable;
     ReadEn <= ReadEnable;
end Behavioral;
```

3.2 design.vhd

This is the main code. It calls Clk_divider.vhd and fifo.v in turn.

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
-- ENTITY DESCRIPTION
entity LEAKY_BUCKET is
Generic (
constant DATA_WIDTH  : positive := 8;
constant FIFO_DEPTH   : positive := 256);
 Port (
CLK      : in  STD_LOGIC;
RST      : in  STD_LOGIC;
EN       : in  STD_LOGIC;
INDataRate : in  STD_LOGIC_VECTOR (2 downto 0);
OUTDataRate : in  STD_LOGIC_VECTOR (2 downto 0);
DataIn: in  STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
DataOut: out STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0)););
end LEAKY_BUCKET;
-- ARCHITECTURE DESCRIPTION
architecture Behavioral of LEAKY_BUCKET is
signal WriteEn    : STD_LOGIC;
signal ReadEn     : STD_LOGIC;
-- COMPONENT DECLARATION OF STANDARD FIFO
 component STD_FIFO
 Generic (
constant DATA_WIDTH  : positive := 8;
constant FIFO_DEPTH  : positive := 256);
Port (
CLK  : in  STD_LOGIC;
RST   : in  STD_LOGIC;
WriteEn : in  STD_LOGIC;
DataIn: in  STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
ReadEn  : in  STD_LOGIC;
DataOut: out STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0)
);  end component;
 -- COMPONENT DECLARATION OF CLOCK DIVIDER
 component CLK_DIVIDER
Port (
CLK : in  STD_LOGIC;
RST   : in  STD_LOGIC;
EN    : in  STD_LOGIC;
INDataRate : in  STD_LOGIC_VECTOR (2 downto 0);
OUTDataRate  :  in   STD_LOGIC_VECTOR (2 downto 0);
WriteEn : out STD_LOGIC;
ReadEn  : out STD_LOGIC);
end component;
begin
-- INSTANCE OF STANDARD FIFO
FIFO: STD_FIFO
GENERIC MAP (
FIFO_DEPTH => FIFO_DEPTH,
DATA_WIDTH => DATA_WIDTH )
PORT MAP (
CLK      => CLK,
RST      => RST,
WriteEn => WriteEn,
DataIn   => DataIn,
ReadEn  => ReadEn,
DataOut => DataOut
);  -- INSTANCE OF CLOCK DIVIDER
 CLKDIV: CLK_DIVIDER
PORT MAP (
 CLK      => CLK,
 RST      => RST,
 EN       => EN,
 INDataRate=> INDataRate,
 OUTDataRate=> OUTDataRate,
WriteEn => WriteEn,
ReadEn   => ReadEn);
end Behavioral;
```

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

3.3 fifo.vhd

In following code we have maintained first in first out buffer. This buffer is used to store the incoming and outgoing packets.

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
 entity STD_FIFO is
Generic (constant DATA_WIDTH  : positive := 8;
constant FIFO_DEPTH : positive := 256);
Port (
CLK : in  STD_LOGIC;
RST : in  STD_LOGIC;
WriteEn : in  STD_LOGIC;
DataIn : in  STD_LOGIC_VECTOR (DATA_WIDTH - 1
downto 0);
ReadEn  : in  STD_LOGIC;
DataOut: out STD_LOGIC_VECTOR(DATA_WIDTH -
1 downto 0)
); end STD_FIFO;
architecture Behavioral of STD_FIFO is
signal Full : STD_LOGIC;
signal Empty : STD_LOGIC;
begin  -- Memory Pointer Process
fifo_proc : process (CLK)
type FIFO_Memory is array (0 to FIFO_DEPTH - 1) of
STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
variable Memory : FIFO_Memory;
variable Head : natural range 0 to FIFO_DEPTH -1;
variable Tail : natural range 0 to FIFO_DEPTH - 1;
variable Looped : boolean;
begin
if rising_edge(CLK) then
if RST = '1' then
  Head := 0;
  Tail := 0;
  Looped := false;
   Full  <= '0';
   Empty <= '1';
else
if (ReadEn = '1') then
if ((Looped = true) or (Head /= Tail)) then  --

Update data output
   DataOut <= Memory(Tail);   -- Update Tail pointer as
needed
if (Tail = FIFO_DEPTH - 1) then
   Tail := 0;
   Looped := false;
else
    Tail := Tail + 1;
end if; end if; end if;
if (WriteEn = '1') then
if ((Looped = false) or (Head /= Tail)) then
   Memory(Head) := DataIn; -- Write Data to Memory
    -- Increment Head pointer as needed
if (Head = FIFO_DEPTH - 1) then
     Head := 0;
     Looped := true;
else
     Head := Head + 1;
end if; end if; end if; -- Update Empty and Full flags
if (Head = Tail) then
if Looped then

    Full <= '1';
else
    Empty <= '1';
end if;
else
    Empty<= '0';
     Full   <= '0';
end if; end if; end if;
end process;
end Behavioral;
```

3.4 testbench.vhd

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.MATH_REAL.ALL;
-- ENTITY DESCRIPTION
entity TESTBENCH is
end TESTBENCH;
-- ARCHITECTURE DESCRIPTION
architecture Behavioral of TESTBENCH is
constant DATA_WIDTH : positive := 8;
constant FIFO_DEPTH : positive := 1024;
signal CLK                  : STD_LOGIC;
signal RST                  : STD_LOGIC := '0';
signal EN                   : STD_LOGIC := '0';
signal INDataRate: STD_LOGIC_VECTO(2 downto 0)
:= "000";
signal OUTDataRate: STD_LOGIC_VECTOR (2 downto
0) := "000";
signal DataIn      :              STD_LOGIC_VECTOR
(DATA_WIDTH - 1 downto 0) := "00000000";
signal      DataOut      :      STD_LOGIC_VECTOR
(DATA_WIDTH - 1 downto 0) := "00000000";
```

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

```
constant clock_period : time := 1 ns;
-- COMPONENT DECLARATION OF DESIGN
component LEAKY_BUCKET
Generic (
constant DATA_WIDTH  : positive := 8;
constant FIFO_DEPTH     : positive := 256);
Port (
CLK      : in  STD_LOGIC;
RST      : in  STD_LOGIC;
EN       : in  STD_LOGIC;
INDataRate       : in  STD_LOGIC_VECTOR(2 downto 0);
OUTDataRate     : in  STD_LOGIC_VECTOR(2 downto 0);
DataIn    : in  STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0);
DataOut : out STD_LOGIC_VECTOR (DATA_WIDTH - 1 downto 0));
end component;
-- Variable for Input Data Rate --
-- 0: 2Mbps | MaxCountValue = 500 for InputClockFre = 1GHz --
-- 1: 4Mbps | MaxCountValue = 250 for InputClockFre = 1GHz --
-- 2: 6Mbps | MaxCountValue = 167 for InputClockFre = 1GHz --
-- 3: 8Mbps | MaxCountValue = 125 for InputClockFre = 1GHz --
-- 4: 10Mbps | MaxCountValue = 100 for InputClockFre = 1GHz --
-- 5: 12Mbps | MaxCountValue = 84 for InputClockFre = 1GHz --
-- 6: 14Mbps | MaxCountValue = 72 for InputClockFre = 1GHz --
-- 7: 16Mbps | MaxCountValue = 63 for InputClockFre = 1GHz --
----------------------------------
begin
-- INSTANCE OF DESIGN
UUT: LEAKY_BUCKET
GENERIC MAP (
FIFO_DEPTH => FIFO_DEPTH,
DATA_WIDTH => DATA_WIDTH )
PORT MAP (
CLK => CLK,
RST      => RST,
EN       => EN,
INDataRate       => INDataRate,
OUTDataRate     => OUTDataRate,
DataIn   => DataIn,
DataOut => DataOut       );
clock_process: process begin
CLK <= '0';
wait for (clock_period/2);
CLK <= '1';
wait for (clock_period/2);
end process;
stimulus_process: process
variable s1: positive := 200;
-- Some seed value
variable s2: positive := 500;
-- Some seed value
variable RandomNum: real; -- variable to store the random number between 0.0 to 1.0
variable Count : natural range 0 to 100000 -1;
variable walking_pattern : natural range 0 to 1000000 - 1 := 0;
variable InputDataRate, OutputDataRate: STD_LOGIC_VECTOR (2 downto 0) := "000";
begin
RST <= '1';
wait for (5*clock_period);
RST <= '0';
wait until CLK'event and CLK='1';
EN <= '1';
-- Input Data Rate : 2MBps, Output Data Rate : 2MBps
InputDataRate := "000";
INDataRate       <= InputDataRate;
OutputDataRate := "000";
OUTDataRate     <= OutputDataRate;
wait until CLK'event and CLK='1';
EN <= '0';
repeat_my_dear_design_for_input_data_rate_2MBps: for i in 0 to 10 loop
case (InputDataRate) is
when "000" => Count := 500;
when "001" => Count := 250;
when "010" => Count := 167;
when "011" => Count := 125;
when "100" => Count := 100;
when "101" => Count := 84;
when "110" => Count := 72;
when "111" => Count := 63;
when others => Count := 500;
end case;
DataIn <= std_logic_vector(to_unsigned(walking_pattern,DATA_WIDTH));
walking_pattern := walking_pattern + 1;
wait_my_dear_design_for_input_data_rate_2MBps: for j in 0 to Count-1 loop
wait until CLK'event and CLK='1';
```

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

```
end loop;
end loop;
-- Input Data Rate : 4MBps, Output Data Rate : 2MBps
EN <= '1';
InputDataRate := "001";
INDataRate        <= InputDataRate;
OutputDataRate := "000";
OUTDataRate     <= OutputDataRate;
wait until CLK'event and CLK='1';
EN <= '0';
repeat_my_dear_design_for_input_data_rate_4MBps:  for
i in 0 to 10 loop
case (InputDataRate) is
when "000" => Count := 500;
when "001" => Count := 250;
when "010" => Count := 167;
when "011" => Count := 125;
when "100" => Count := 100;
when "101" => Count := 84;
when "110" => Count := 72;
when "111" => Count := 63;
when others => Count := 500;
end case;
DataIn                                          <=
std_logic_vector(to_unsigned(walking_pattern,DATA_W
IDTH));
walking_pattern := walking_pattern + 1;
wait_my_dear_design_for_input_data_rate_4MBps:  for  j
in 0 to Count-1 loop
wait until CLK'event and CLK='1';
end loop;
end loop;
-- Input Data Rate : 6MBps, Output Data Rate : 2MBps
EN <= '1';
InputDataRate := "010";
INDataRate        <= InputDataRate;
OutputDataRate := "000";
OUTDataRate     <= OutputDataRate;
wait until CLK'event and CLK='1';
EN <= '0';
repeat_my_dear_design_for_input_data_rate_6MBps:  for
i in 0 to 10 loop
case (InputDataRate) is
when "000" => Count := 500;
when "001" => Count := 250;
when "010" => Count := 167;
when "011" => Count := 125;
when "100" => Count := 100;
when "101" => Count := 84;
when "110" => Count := 72;
when "111" => Count := 63;
when others => Count := 500;
end case;
DataIn                                          <=
std_logic_vector(to_unsigned(walking_pattern,DATA_W
IDTH));
walking_pattern := walking_pattern + 1;
wait_my_dear_design_for_input_data_rate_6MBps:  for  j
in 0 to Count-1 loop
wait until CLK'event and CLK='1';
end loop;
end loop;
-- Input Data Rate : 8MBps, Output Data Rate : 2MBps
EN <= '1';
InputDataRate := "011";
INDataRate        <= InputDataRate;
OutputDataRate := "000";
OUTDataRate     <= OutputDataRate;
wait until CLK'event and CLK='1';
EN <= '0';
repeat_my_dear_design_for_input_data_rate_8MBps:  for
i in 0 to 10 loop
case (InputDataRate) is
when "000" => Count := 500;
when "001" => Count := 250;
when "010" => Count := 167;
when "011" => Count := 125;
when "100" => Count := 100;
when "101" => Count := 84;
when "110" => Count := 72;
when "111" => Count := 63;
when others => Count := 500;
end case;
DataIn                                          <=
std_logic_vector(to_unsigned(walking_pattern,DATA_W
IDTH));
walking_pattern := walking_pattern + 1;
wait_my_dear_design_for_input_data_rate_8MBps:  for  j
in 0 to Count-1 loop
wait until CLK'event and CLK='1';
end loop;
end loop;
kill_time_my_dear_design: for i in 0 to 1000000 loop
wait until CLK'event and CLK='1';
 end loop;
-- Hard Stop
assert false
report "Simulation Completed"
severity failure;
end process;
end Behavioral;
```

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 5, Issue 1, January 2018

The register level schematic view of the implementation is shown in fig. 7. This register level is generated by using vertex 5 board in Xilinx.

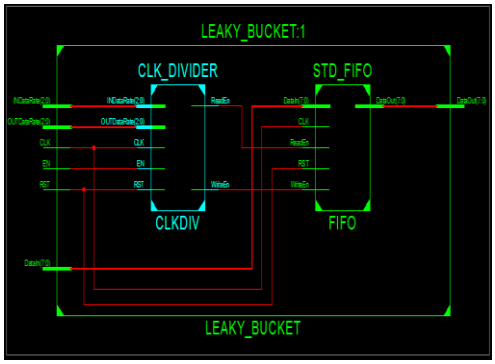



*Fig. 7 Schematic view of implementation*

## 4. RESULTS

Fig. 8 and fig. 9 shows the wave form generated from hardware implementation. In both figures we can see that if we change the input value from 000 to 101 the output rate remains constant.

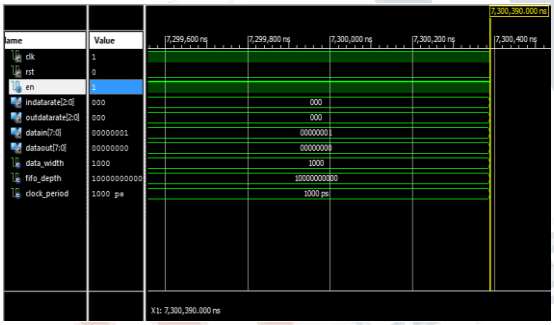


*Fig. 8 Wave form*

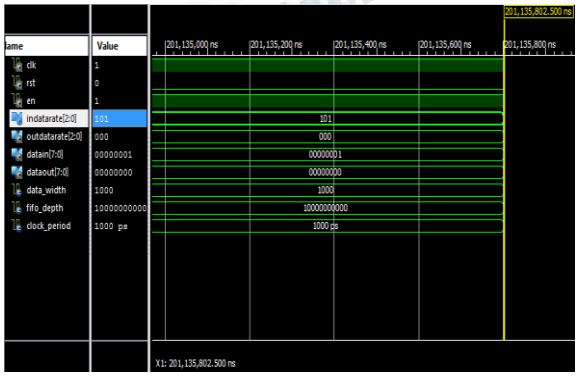


*Fig. 9 Wave form*

The total memory consumption is only 197520 kilobytes. It is very much less then the software implementation. The timing details of the implementation are as follows:
Minimum period: 5.088 (Maximum Frequency: 196.524 MHz.

Minimum Input arrival time before clock: 3.612ns.
Maximum output required time after clock: 2.826 ns. Total Real time to Xst completion is 37:00 second and Total CPU time to Xst completion is 36.51 second. Table 1 shows the device utilization of the implementation. 2138 Slice Registers are used out of 12480. It means only 17 % of available registers are used. Similarly 4129 Slice LUTs are used out of 12480. It means only 33% of Slice LUTs are used.

***Table – 1***

| Device Utilization | | | |
|---|---|---|---|
| Logic Utilization | Used | Available | Utilization |
| Number of Slice Registers | 2138 | 12480 | 17% |
| Number of Slice LUTs | 4129 | 12480 | 33% |
| Number of Fully used LUT-FF pairs | 1739 | 4528 | 38% |
| Number of bonded IOBs | 25 | 172 | 14% |
| Number of BUFG/ BUFGCTRLs | 1 | 32 | 3% |

The total power consumption is 0.338 W. The power consumption in dynamic stage is 0.017 W and the power consumption in quiescent stage is 0.321 W.

## 5. CONCLUSION

Thus In this paper we showed the results generated by Valgrind and Massif Visualizer profiling tools. Profiling data shows that software implementation is taking more memory and also it is not optimized. Leaky Bucket algorithm is used to control the flow of data on communication channel. If we used software implementation then the processing overhead is very high. In this paper we also show the results of hardware implementation. The hardware implementation of Leaky Bucket algorithm consumes only 197520 kilobytes of memory, it also takes less time in execution. We see that it takes only 2138 out of 12480 of slice registers. Thus device utilization is also excellent. The total power supply is 0.338 w out of which .017w is used dynamic stage and .321w is used in quiescent stage. We conclude that the

execution of this implementation is fast and it consumes less power. Thus this implementation is efficient for a network processor.

## REFERENCES

[1] J. Zhigang, L. Lemin, "Analysis of the leaky bucket algorithm for priority services," Journal of Electronics China, vol. 13, pp. 333-338, October 1996.

[2] N.Yin, M. Hluchyj, "Analysis of the Leaky Bucket Algorithm for ON-OFF Data Sources," Journal of High Speed Networks, vol. 2, pp. 81-98, January 1931.

[3] C. Chang, Z. Eul, L. Lin "Intelligent leaky bucket algorithms for sustainable-cell-rate usage parameter control in ATM networks," Information Networking, 2001. Proceedings. 15th International Conference on, August 2002.

[4] P. Indumathi, S. Shanmugavel, HC. Mahesh, "Buffered Leaky Bucket Algorithm for Congestion Control in ATM Networks," IETE Journal of Research, vol. 48, pp. 59-67, March 2015.

[5] M. Ahmadi, S. Wong," Network Processors: Challenges and Trends," In Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc, pp. 265-269, 2006

[6] A. Kind, R. Pletka, M. Waldvogel "The Role of Network Processors in Active Networks, IFIP International Working Conference on Active Networks, pp. 20-31, 2003.

[7] S.Govind, R.Govindarajan, J. Kuri,"Packet Reordering in Network Processors," Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, June 2007.

[8] S. Ata., M. Murata, H. Miyahara, "Analysis of network traffic and its application to design of high-speed routers", IEICE Transactions on Information and systems, pp. 988-995, 2000.

[9] M. Abdelall, A. F. Shalash, H. M. Hassan, M. Hassan, O.A. Nasr, "Design and implementation of application-specific instruction-set processor design for high-throughput multi-standard wireless orthogonal frequency division multiplexing baseband processor", IET Circuits, Devices & Systems, pp.191-203, 2015.

[10] M. Ahmadi, S. Wong, "Network Processors: Challenges and Trends", In Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc, pp. 265-269, 2006.

[11] S. Bhagwani, "Comparative Study of Various Network Processors Processing Elements Topologies", Int. Journal of Engineering Science and Innovative Technology (IJESIT), pp. 157-16, 2013.

[12] P. Cascón, J. Ortega, Y. Luo, E. Murray, A. Díaz, I. Rojas," Improving IPS by network processors", The Journal of Supercomputing, pp.99-108, 2011.

[13] D. Chaurasiya, P. Singh, A. Joshi, S. K. Pandey, "Analysis of Network Processor Processing Elements Topologies", Int. Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE), pp.66-70, 2012.

[14] J. Fu, O. Hagsand,"Designing and Evaluating Network Processor Applications", In Proc. of 2005 IEEE Workshop on High Performance Switching and Routing (HPSR) Hong Kong, pp. 142-146, 2005.

[15] J. Fu, O. Hagsand O, G. Karlsson, "Queuing Behavior and Packet Delays in Network Processor Systems", In Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on, pp.217-224, 2007.

[16] J. Guo, J. Yao, L. Bhuyan, "An Efficient Packet Scheduling Algorithm in Network Processors", IEEE Infocom, pp.807-818, 2005.

[17] Y. Kanada," High-Level Portable Programming Language for Optimized Memory Use of Network Processors", Communications and Network, pp.55-69, 2015.

[18] A. Kind, R. Pletka, W. Marcel," The Role of Network Processors in Active Networks", International Federation for Information Processing, pp. 20–31, 2004.