

Architecture of Network Devices

INSTRUCTOR: PROF. MASOUD SABAEI

AMIRKABIR UNIVERSITY OF TECHNOLOGY
(TEHRAN POLYTECHNIC)

APSARA Matching

Authors:

Reza Adinepour

adinepour@aut.ac.ir

Fall 2024

Contents

1	Abstract	2
2	Introduction	2
3	APSARA	2
4	Overview of Code Files and Functions	3
4.1	File: matching.h	3
4.2	File: matching.c	4
4.3	File: switch.h	4
4.4	File: switch.c	4
4.5	File: permutation.h	5
4.6	File: permutation.c	5
4.7	File: switch-apsara.c	5
4.8	File: main.c	5
5	How to Run the Project	6
6	Notes	7
7	Conclusion and Results	8
8	Key Insight	10

1 Abstract

The APSARA algorithm [2] employs the following two ideas:

1. Use of memory
2. Exploring neighbors in parallel. The neighbors are defined so that it is easy to compute them using hardware parallelism.

2 Introduction

The high demand for Internet bandwidth has led to increasingly higher speed links and caused an associated demand for routers with a high aggregate switching capacity. At the highest speeds, input-queued (IQ) switches have become the architecture of choice, mainly because the memory bandwidth of their packet buffers is very low compared to that of output-queued and shared-memory architectures.

3 APSARA

In APSARA, the “neighbors” of the current match are considered as candidates of the match in the next time slot. A match S' is defined as a neighbor of a match S if, and only if, there are two input-output pairs in S , say input i_1 to output j_1 and input i_2 to output j_2 , switching their connections so that in S' input i_1 connects to output j_2 and i_2 connects to output j_1 . All other input-output pairs are the same under S and S' . We denote the set of all the neighbors of a match S as $N(S)$. As shown in Figure 1 [1], the matching S for a 3×3 switch and its three neighbors S_1 , S_2 , and S_3 are given below:

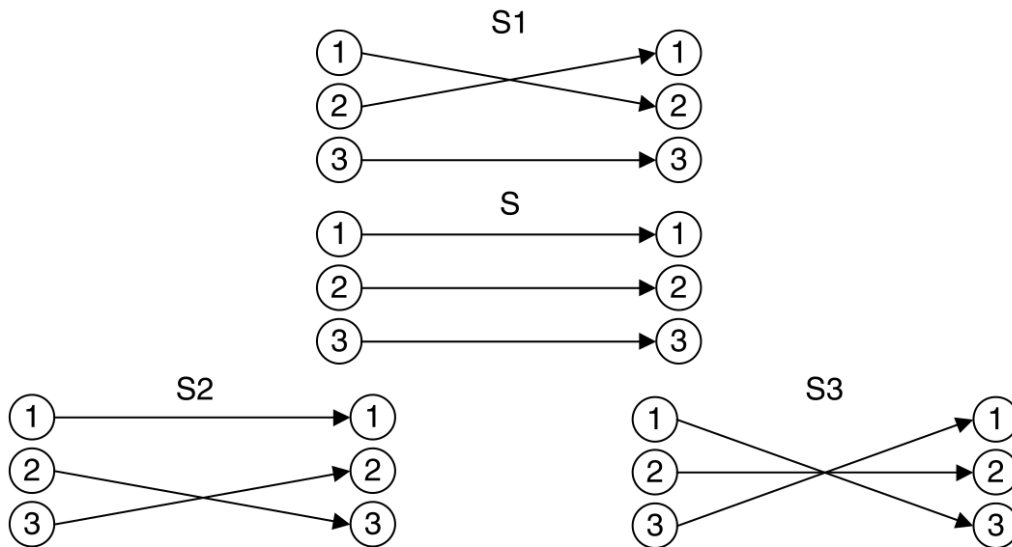


Figure 1: Example of neighbors in APSARA algorithms

$$S = (1, 2, 3), \quad S_1 = (2, 1, 3), \quad S_2 = (1, 3, 2), \quad \text{and} \quad S_3 = (3, 2, 1).$$

Let $S(t)$ be the matching determined by APSARA at time t . Let $H(t+1)$ be the match corresponding to the Hamiltonian walk at time $t+1$. At time $t+1$, APSARA does the following:

1. Determine $N(S(t))$ and $H(t)$.
2. Let $M(t+1) = N(S(t)) \cup H(t+1) \cup S(t)$. Compute the weight $\langle S', Q(t+1) \rangle$ for all $S' \in M(t+1)$.
3. Determine the match at $t+1$ by

$$S(t+1) = \arg \max_{S' \in M(t+1)} \langle S', Q(t+1) \rangle. \quad (7.8)$$

APSARA requires the computation of the weight of neighbors. Each such computation is easy to implement. However, computing the weights of all $\binom{N}{2}$ neighbors requires a lot of space in hardware for large values of N . To overcome this, two variations were considered in the work of Giaccone et al. [1] by reducing the number of neighbors considered in each time slot.

4 Overview of Code Files and Functions

This project consists of multiple files that implement the APSARA algorithm for managing input-queued switches. Below is a detailed description of each file and the functions within them.

4.1 File: **matching.h**

This header file defines the structure and prototypes for operations on matchings:

- **struct matching**: Represents a matching configuration with:
 - **n**: The number of ports.
 - **match**: An array representing the connections between input and output ports.
- **const struct matching *matching_new(int n, const int match[])**: Allocates and initializes a new matching.
- **void matching_delete(const struct matching *m)**: Frees memory allocated for a matching.
- **void matching_print(const struct matching *m, FILE *fp)**: Prints the matching details to the given file pointer.

4.2 File: **matching.c**

This file implements the functions declared in **matching.h**:

- **matching_new**: Creates and returns a new matching structure.
- **matching_delete**: Frees the memory of a matching and its internal components.
- **matching_print**: Prints the input-output pairs of the matching in a formatted table.

4.3 File: **switch.h**

This header file defines the switch structure and its operations:

- **struct sw**: Represents a switch with:
 - **m**: The current matching configuration.
 - **ports**: The number of ports.
 - **queue**: A 2D array representing the input-output queues.
 - **t**: The current time step.
 - **throughput**: Tracks the total packets successfully processed.
- **struct sw *switch_new(int ports)**: Creates a new switch with the given number of ports.
- **void switch_set_current_matching(struct sw *s, const int match[])**: Updates the current matching of the switch.
- **void switch_put_in_queue(struct sw *s, int in_port, int out_port, int number)**: Adds packets to the queue for a specific input-output pair.
- **void switch_process(struct sw *s)**: Processes packets based on the current matching.
- **void switch_next_matching(struct sw *s)**: Determines the next optimal matching using the APSARA algorithm.
- **void switch_print(struct sw *s, FILE *fp)**: Prints the switch state to the given file pointer.

4.4 File: **switch.c**

This file implements the functions declared in **switch.h**:

- **switch_new**: Initializes a new switch with default matching and empty queues.
- **switch_set_current_matching**: Updates the current matching configuration.
- **switch_put_in_queue**: Adds packets to the appropriate queue.

- **switch_process**: Processes packets based on the current matching, updating throughput and queue states.
- **switch_next_matching**: Computes the next optimal matching by considering neighbors and Hamiltonian matching.
- **switch_print**: Outputs the state of the queues and matching to a log file.

4.5 File: **permutation.h**

Defines the permutation generation function:

- **void permutation(int k, int *v, int size)**: Generates the **k**-th permutation of an array **v** of size **size**.

4.6 File: **permutation.c**

Implements permutation-related utilities:

- **static void swap(int *v, int i, int j)**: Swaps two elements in an array.
- **static void next(int *v, int size)**: Computes the next permutation of the array.
- **permutation**: Generates the **k**-th permutation by iteratively computing the next permutation.

4.7 File: **switch-apsara.c**

Implements APSARA-specific functionalities:

- **static const struct matching **neighbors_matching(const struct matching *m)**: Generates all neighbors of a given matching.
- **static int calculate_cost(int **queue, const struct matching *m)**: Computes the cost (number of packets served) for a matching.
- **static const struct matching *hamiltonian_matching(int t, int n)**: Generates a Hamiltonian matching based on the time step **t**.
- **switch_next_matching**: Implements the core APSARA algorithm for selecting the next matching based on neighbors and Hamiltonian matching.

4.8 File: **main.c**

The main entry point for the simulation:

- **void generate_data(int test_no, int ports[], int num_ports, int load, FILE *log_file)**: Simulates the switch operation under various conditions and logs results.

- `void generate_gnuplot_script()`: Creates a GNUplot script to visualize throughput vs. number of ports.
- `int main()`: The main function, responsible for running the simulation, generating data, and visualizing results.

5 How to Run the Project

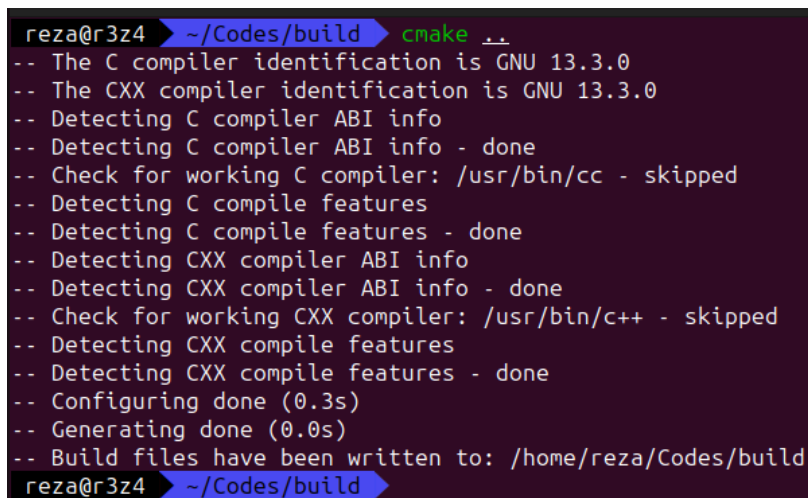
To build and execute the project, follow these steps:

1. Create and Navigate to the Build Directory:

```
$ mkdir build
$ cd build
```

2. Generate Build Files with `cmake`:

```
$ cmake ..
```



```
reza@r3z4 ~/Codes/build$ cmake ..
-- The C compiler identification is GNU 13.3.0
-- The CXX compiler identification is GNU 13.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /home/reza/Codes/build
reza@r3z4 ~/Codes/build$
```

Figure 2: Generate Build Files

- The `cmake ..` command runs CMake in the `build` directory, specifying the parent directory (`..`) as the location of the source code.
- CMake generates the necessary build system files (e.g., `Makefile`) required to compile the project.

3. Build the Project:

```
$ make
```

```

reza@r3z4 ~/Codes/build$ make
[ 16%] Building C object src/CMakeFiles/APSARA.out.dir/matching.c.o
[ 33%] Building C object src/CMakeFiles/APSARA.out.dir/switch.c.o
[ 50%] Building C object src/CMakeFiles/APSARA.out.dir/switch-apsara.c.o
[ 66%] Building C object src/CMakeFiles/APSARA.out.dir/permutation.c.o
[ 83%] Building C object src/CMakeFiles/APSARA.out.dir/main.c.o
[100%] Linking C executable /home/reza/Codes/bin/APSARA.out
[100%] Built target APSARA.out
reza@r3z4 ~/Codes/build$

```

Figure 3: Build the Project

- The `make` command uses the build files generated by CMake to compile the source code.
- This step produces the executable file for the project, which is typically placed in the `bin` directory.

4. Run the Program:

```
$ ../../bin/APSARA.out
```

- This command executes the compiled program `APSARA.out`.
- The `../../bin/APSARA.out` path assumes that the executable is located in the `bin` directory relative to the `build` directory.

```

reza@r3z4 ~/Codes/build$ ../../bin/APSARA.out
Simulation Times: 131055
Load (0 to 100): 80
Log file set to default: log.txt
Ports: 4, Throughput: 0.999976
Ports: 5, Throughput: 0.999954
Ports: 6, Throughput: 0.999948
Ports: 7, Throughput: 0.999916
Ports: 8, Throughput: 0.999881
Data saved to throughput_vs_ports.dat.
GNUplot script generated in throughput_plot.gp.
Plotting with GNUplot...
QSocketNotifier: Can only be used with threads started with QThread
reza@r3z4 ~/Codes/build$

```

Figure 4: Run Program

6 Notes

- Ensure that `CMake` and `Make` are installed on your system before running the above commands.

- The output and logs will be generated as specified in the `main.c` file. Review the `log.txt` file for detailed logs and `throughput_vs_ports.dat` for performance data.
- To visualize the results, the `generate_gnuplot_script` function creates a GNUplot script, which can be run using:

```
gnuplot -p throughput_plot.gp
```

- After running the project, the generated plots will illustrate the throughput vs. number of ports for the simulated switch.

7 Conclusion and Results

The simulation results, summarized in Table 1 and visualized in Figure 5, demonstrate that, with constant simulation cycles and load, the throughput decreases as the number of ports increases. This behavior can be explained by the following reasons:

Table 1: Simulation Results: Throughput vs. Number of Ports

Port	Throughput	Load	Simulation Cycles
4	0.999976	0.80	131055
5	0.999954	0.80	131055
6	0.999948	0.80	131055
7	0.999916	0.80	131055
8	0.999881	0.80	131055

1. Increased Contention for Output Ports:

- As the number of ports grows, more input ports attempt to connect to the available output ports, increasing the likelihood of multiple input ports competing for the same output port.
- When such contention occurs, only one connection can be matched at a time, leaving other packets unprocessed, as reflected in the decreasing throughput values in Table 1.

2. Higher Matching Complexity:

- The APSARA algorithm evaluates neighbors to select the optimal matching at each step. As the number of ports increases, the number of possible neighbors grows exponentially, complicating the decision-making process.
- This complexity can result in suboptimal matching decisions within the constraints of the simulation cycles, contributing to the decline in throughput as shown in Figure 5.

3. Queue Saturation:

- With more ports, the system's queue capacity for each input-output pair may become limited relative to the number of packets being generated.
- This can result in packets waiting longer in queues, causing delays and reducing effective throughput, which is evident from the trend in both Table 1 and Figure 5.

4. Fixed Simulation Cycles:

- The simulation is conducted over a fixed number of cycles. As the number of ports increases, the matching process must handle more input-output combinations within the same cycles.
- This results in fewer overall packets being served, contributing to the observed trend in Figure 5.

5. Increased Load Distribution:

- For the same load percentage (e.g., 80%), the total number of packets increases with the number of ports. However, the system's ability to process these packets does not scale proportionally due to limitations in the matching algorithm and hardware resources.
- This imbalance leads to reduced throughput efficiency, as indicated in the throughput values in Table 1 and the declining curve in Figure 5.

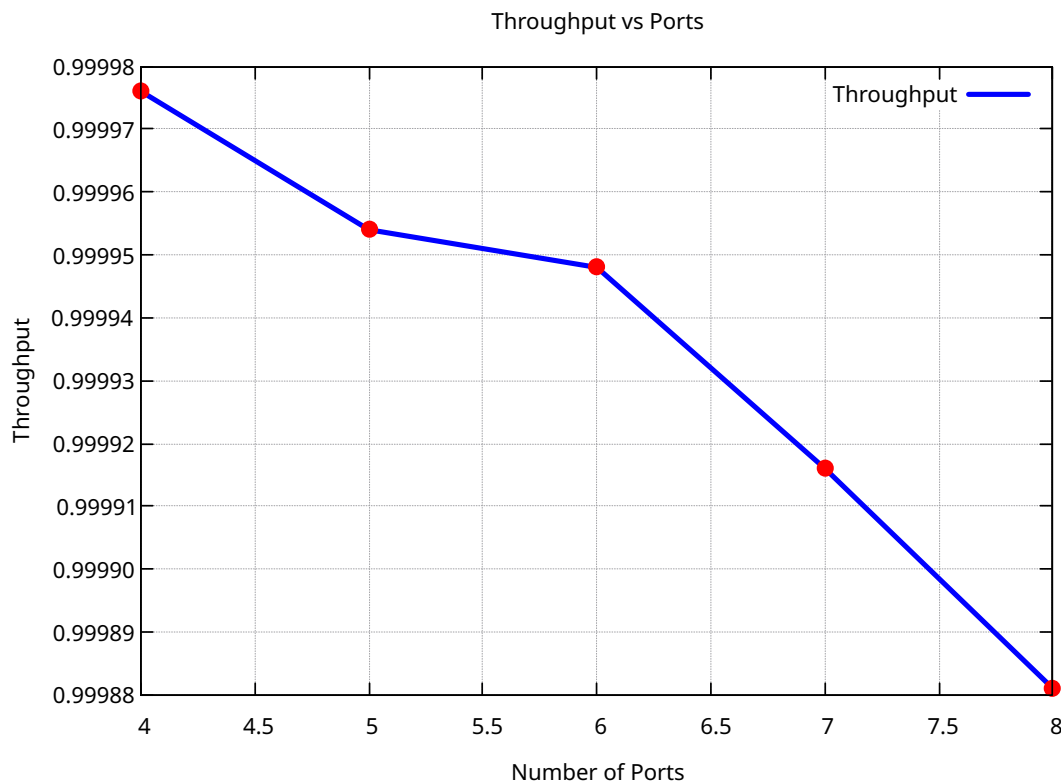


Figure 5: Throughput Graph

8 Key Insight

This behavior, illustrated in both Table 1 and Figure 5, highlights a fundamental trade-off in input-queued switch architectures: while adding more ports increases the switch's connectivity and capacity, it also introduces inefficiencies due to contention, complexity, and resource limitations. Advanced scheduling algorithms or additional resources, such as larger queues or faster matching processes, would be required to mitigate the decrease in throughput as the number of ports increases.

References

- [1] P. Giaccone, B. Prabhakar, and D. Shah. Towards simple, high-performance schedulers for high-aggregate bandwidth switches. In *Proceedings of IEEE INFOCOM'02*, volume 3, pages 1160–1169, 2002.
- [2] P. Giaccone, B. Prabhakar, and D. Shah. Randomized scheduling algorithms for high-aggregate bandwidth switches. *IEEE Journal on Selected Areas in Communications*, 21(4):546–559, May 2003.