LF NETWORKING

LFN Developer & Testing Forum

LF NETWORKING

LFN Developer & Testing Forum

Developing and testing an out-of-tree plugin for VPP

Mauro Sardara
SW Engineer @ Cisco Systems

VPP

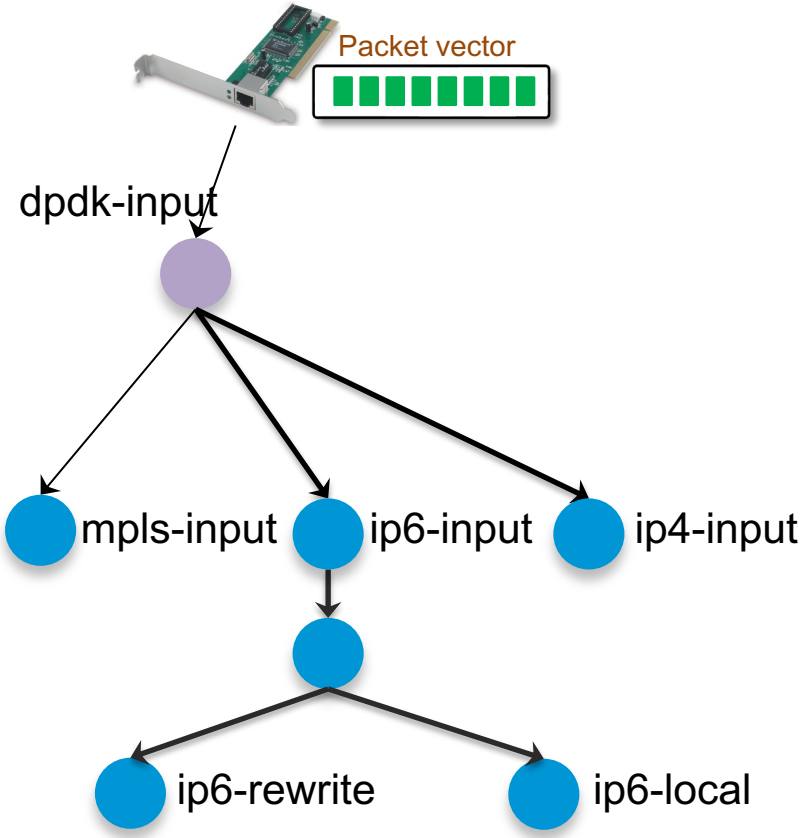
- Vector Packet Processing
  - High performance stack
  - Runs on commodity HW
- Part of Cisco products since 2006
- Open sourced in 2016 under FD.io
- Moved to LFN in 2018

- L3 – 19+ Mpps / Core
  - 2.1 GHz Ice Lake D
  - Small packets (64B): **15 Gbps**
  - Ethernet MTU (1500B): **246 Gbps**
  - Scalable FIB
    - Millions of entries
    - Fast convergence

- Optimizations
  - DPDK Plugin
  - ISA
    - SSE / AVX / AVX2 / AVX512 / NEON…
  - IPC
    - Batching
    - No Mode Switching
    - No Context Switching
    - No Blocking
  - Multi Core
    - Cache / Memory efficiency

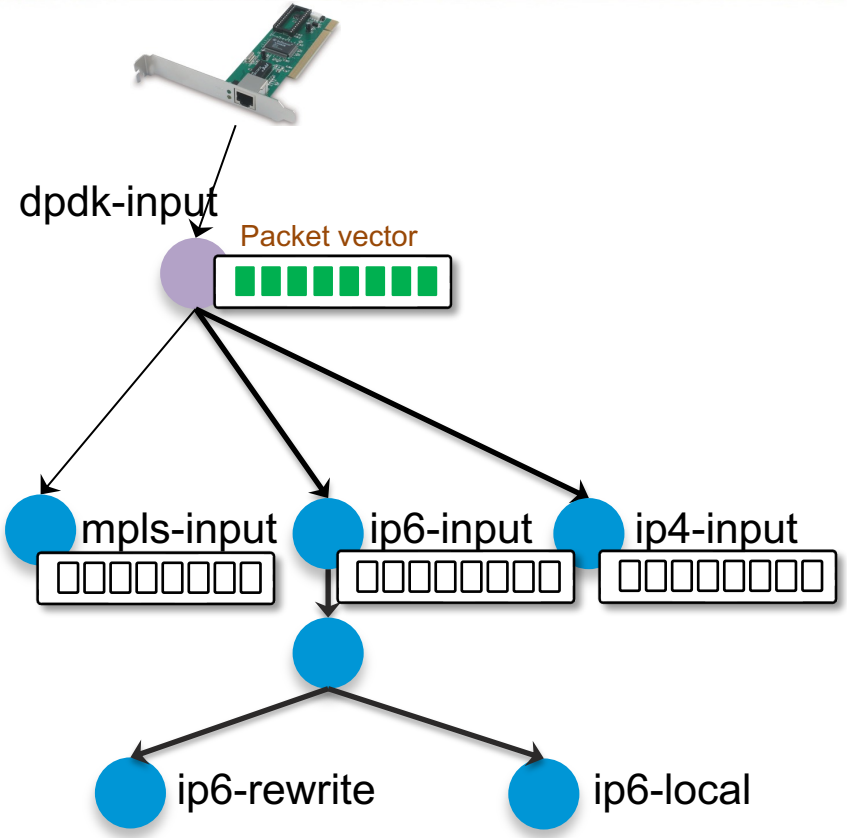
How does VPP work?

- VPP is a ‘packet processing graph’
  - Direct graph of forwarding nodes
- Nodes are
  - Small
  - Loosely coupled
- VPP processes vectors of packets
  - Passed from node to node
  - Multiple packets per invocation



How does VPP work?

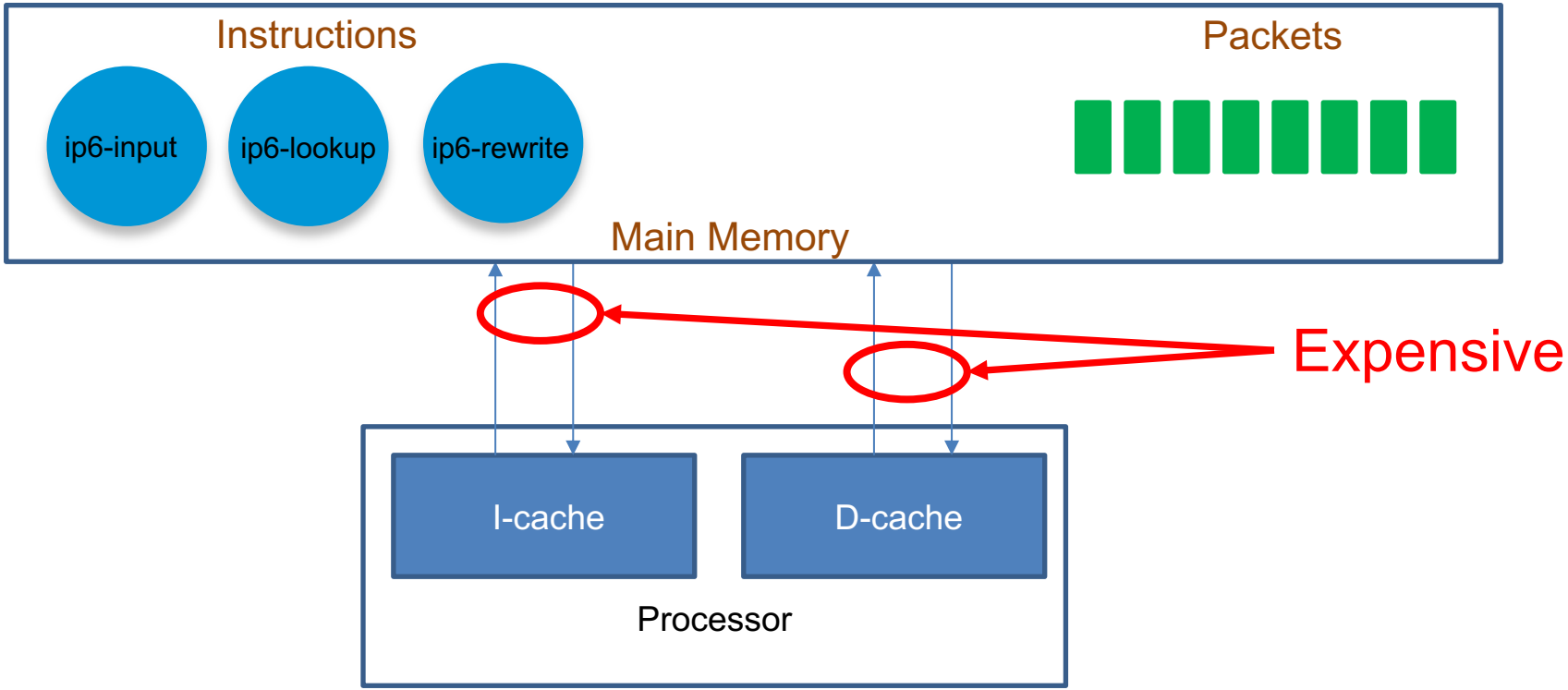
- Each node has its vector(s)
- Packets are “passed” from vector to vector



Scalar vs Vector packet processing

- 19 Mpps on 2.1GHz CPU
  - 110 cycles/packet
- Need to reduce cache misses to the minimum
  - Cache hit:
    - ~2-30 cycles
  - Cache miss (main memory)
    - ~140 cycles

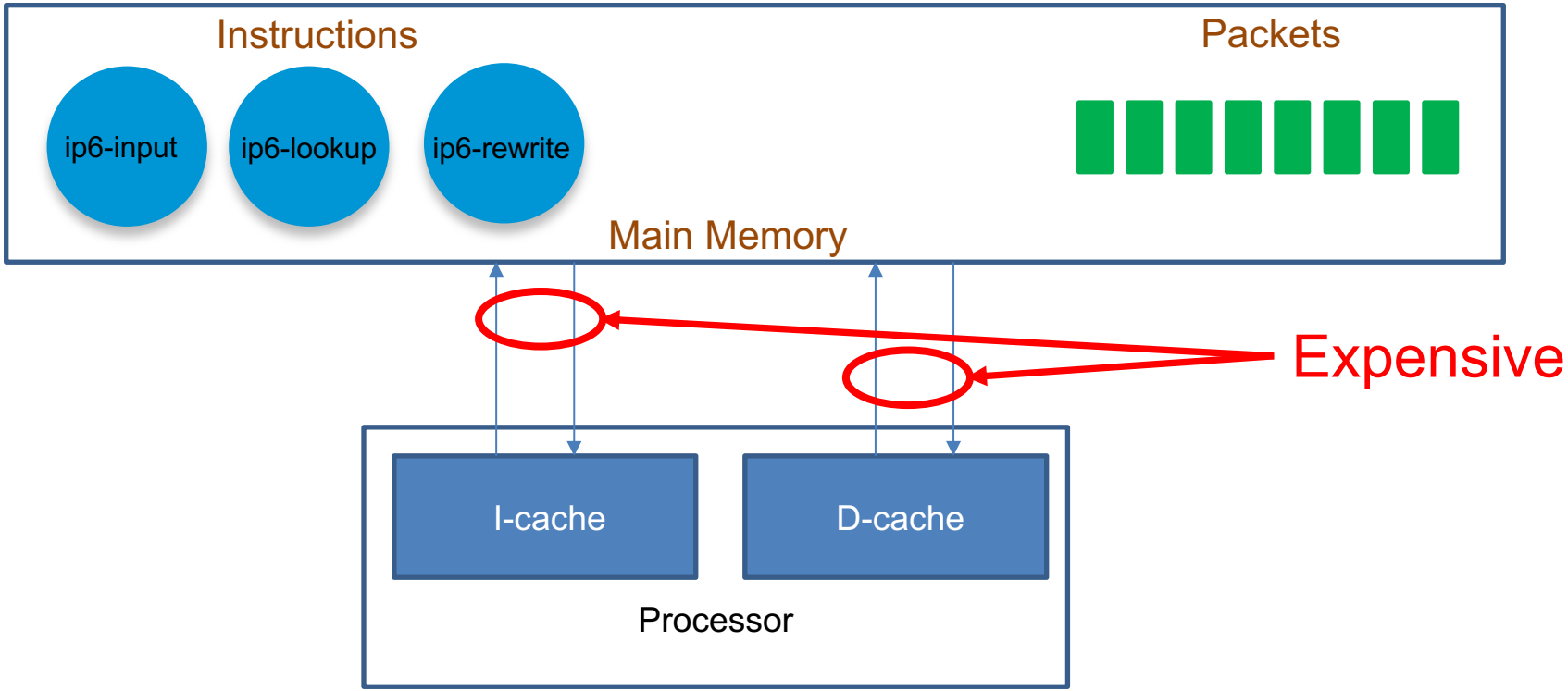
Scalar vs Vector packet processing



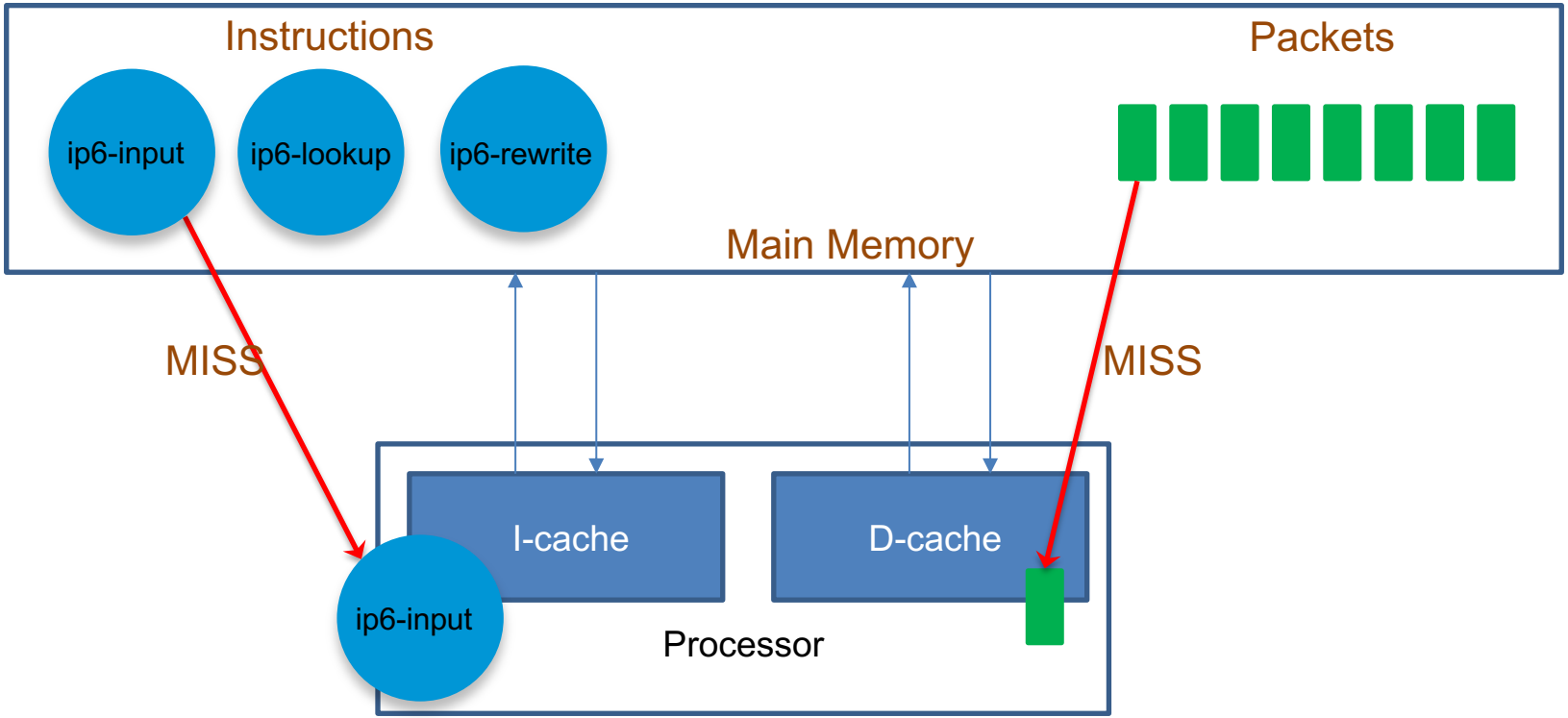
Scalar packet processing

- One packet at a time
  - ISR takes a packet off a rx ring
  - Processes by traversing a set of functions
    - function_a() -> function_b() -> function_c(), ...
    - return, return, return, return, ... return from interrupt
  - Outcomes
    - Drop
    - Punt
    - Rewrite & Fwd

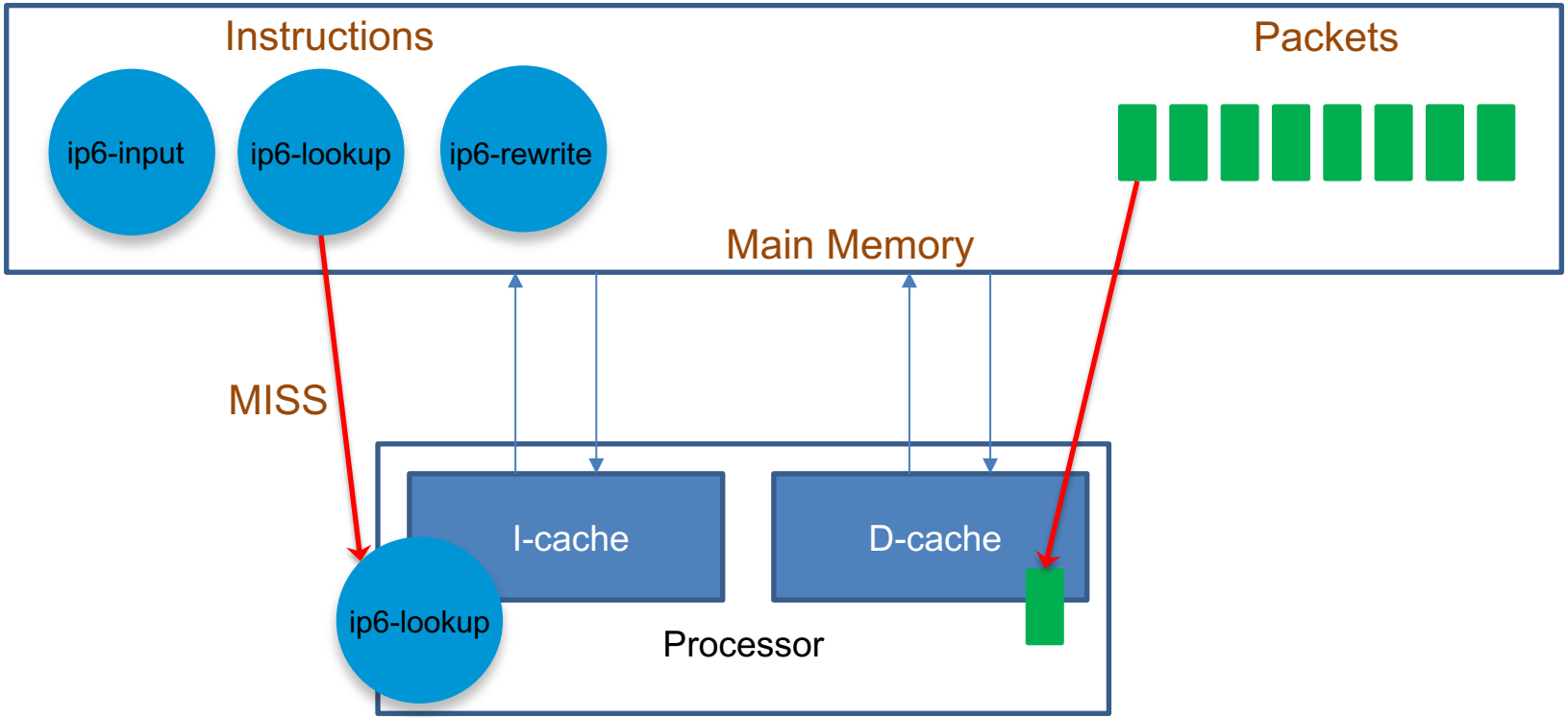
Scalar packet processing



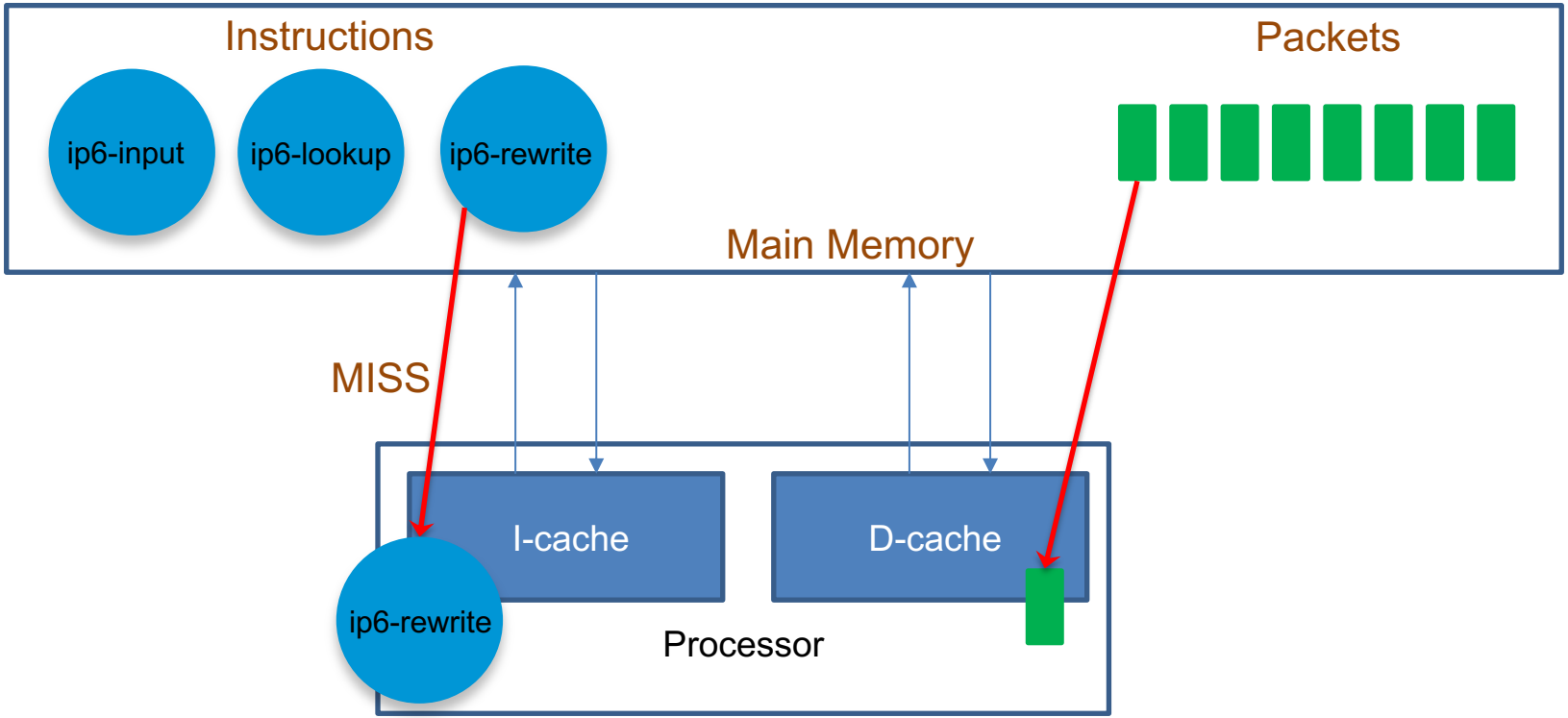
Scalar packet processing



Scalar packet processing



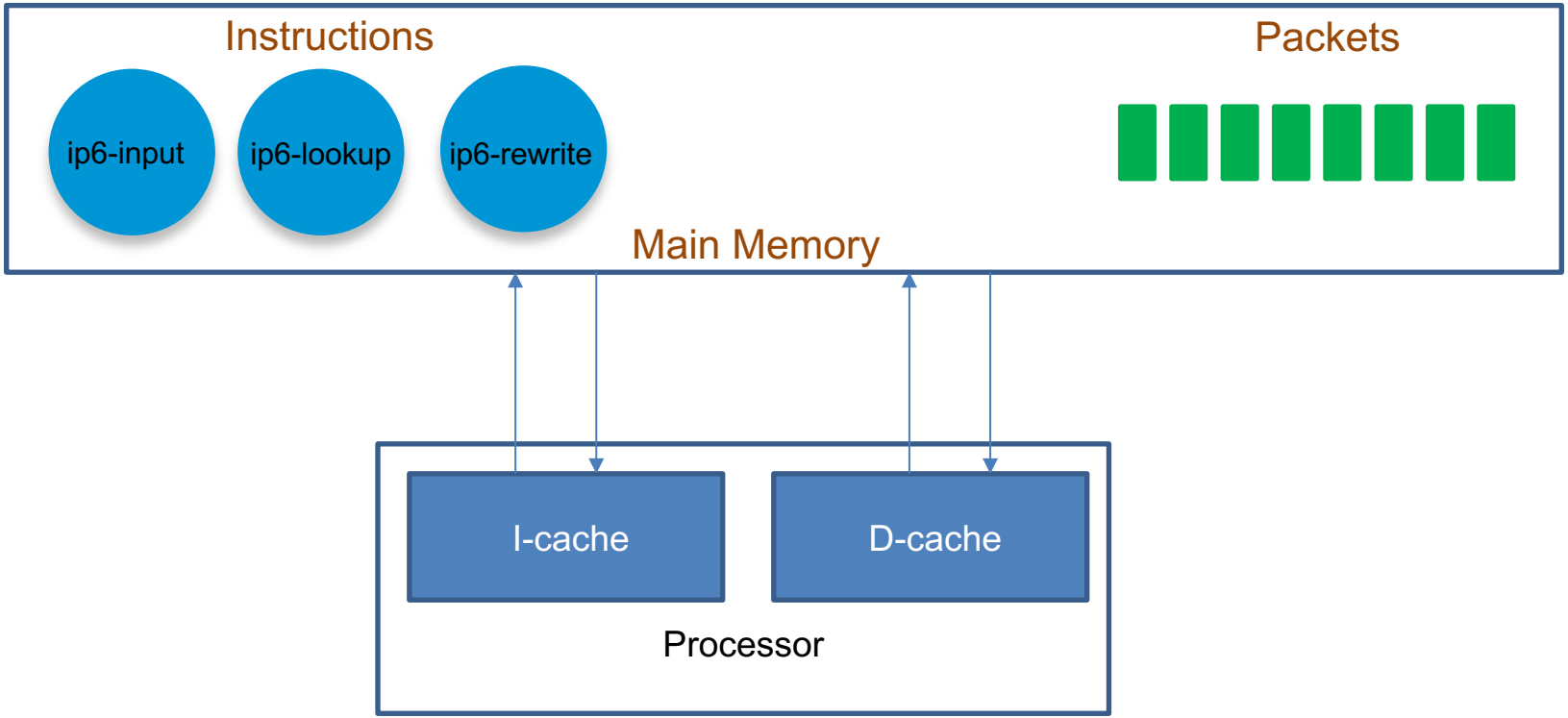
Scalar packet processing



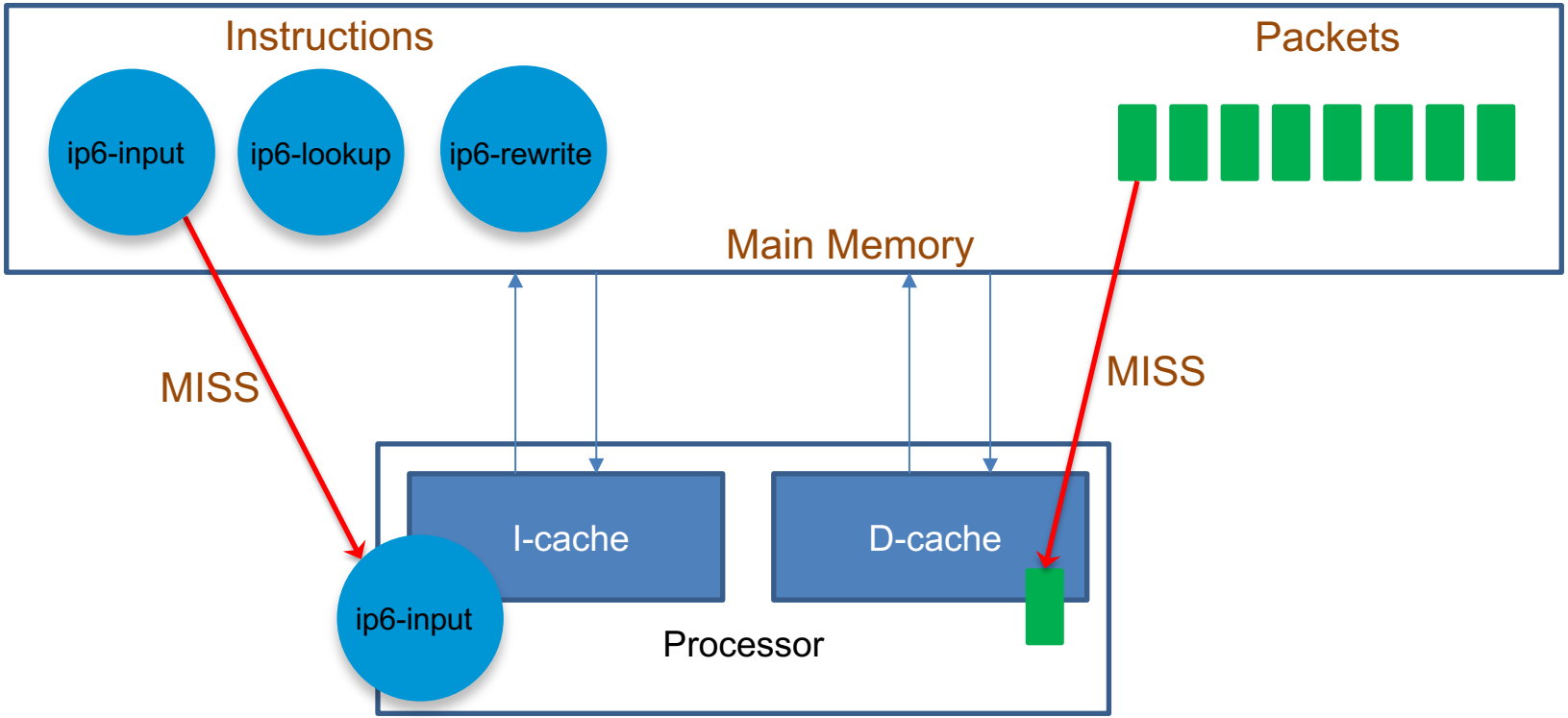
Scalar packet processing

- Issues
  - When the call path length exceeds the size of I-cache, thrashing occurs
    - Each packet incurs an identical set of I-cache misses
    - Only solution: bigger caches
  - Deep call stack adds load-store-unit pressure since stack-locals fall out of the L1 D-cache

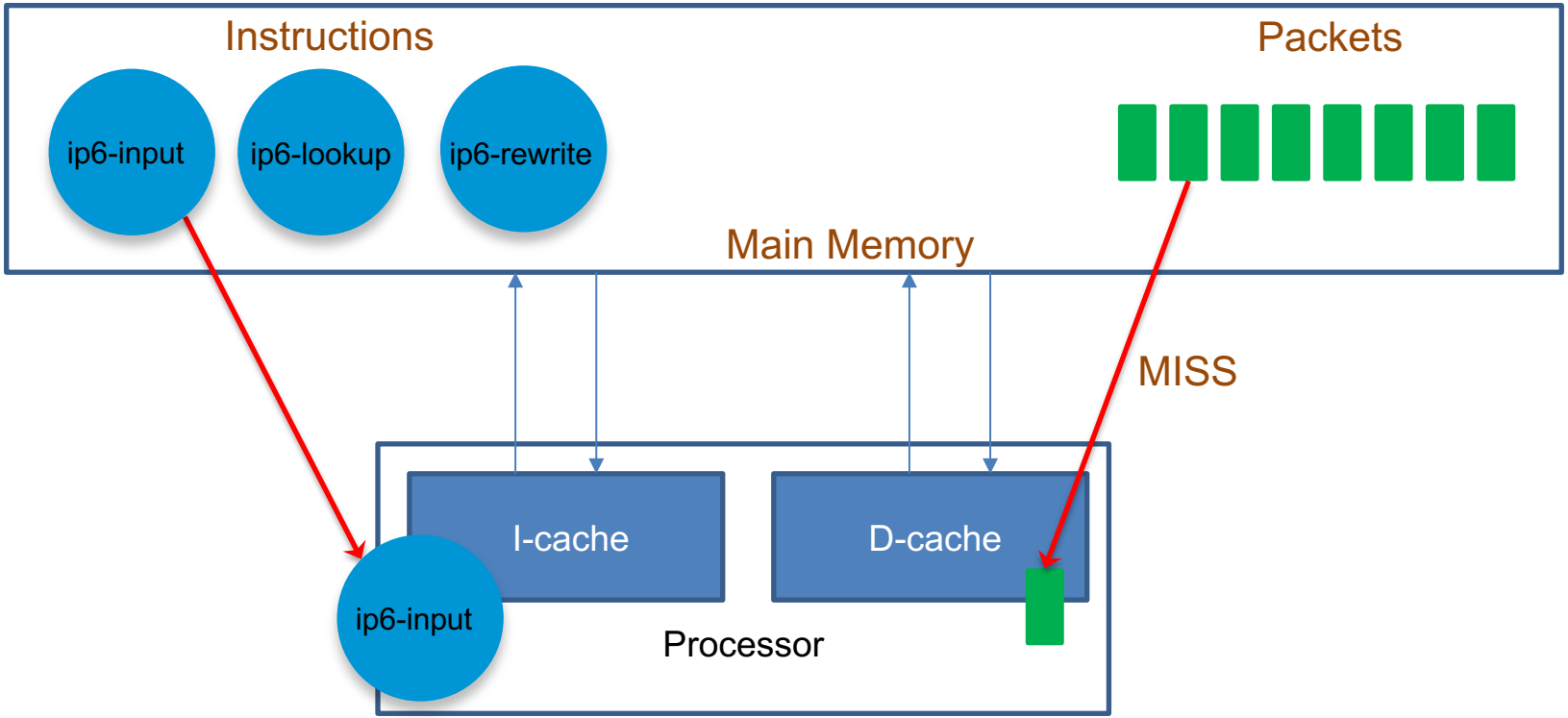
Vector packet processing



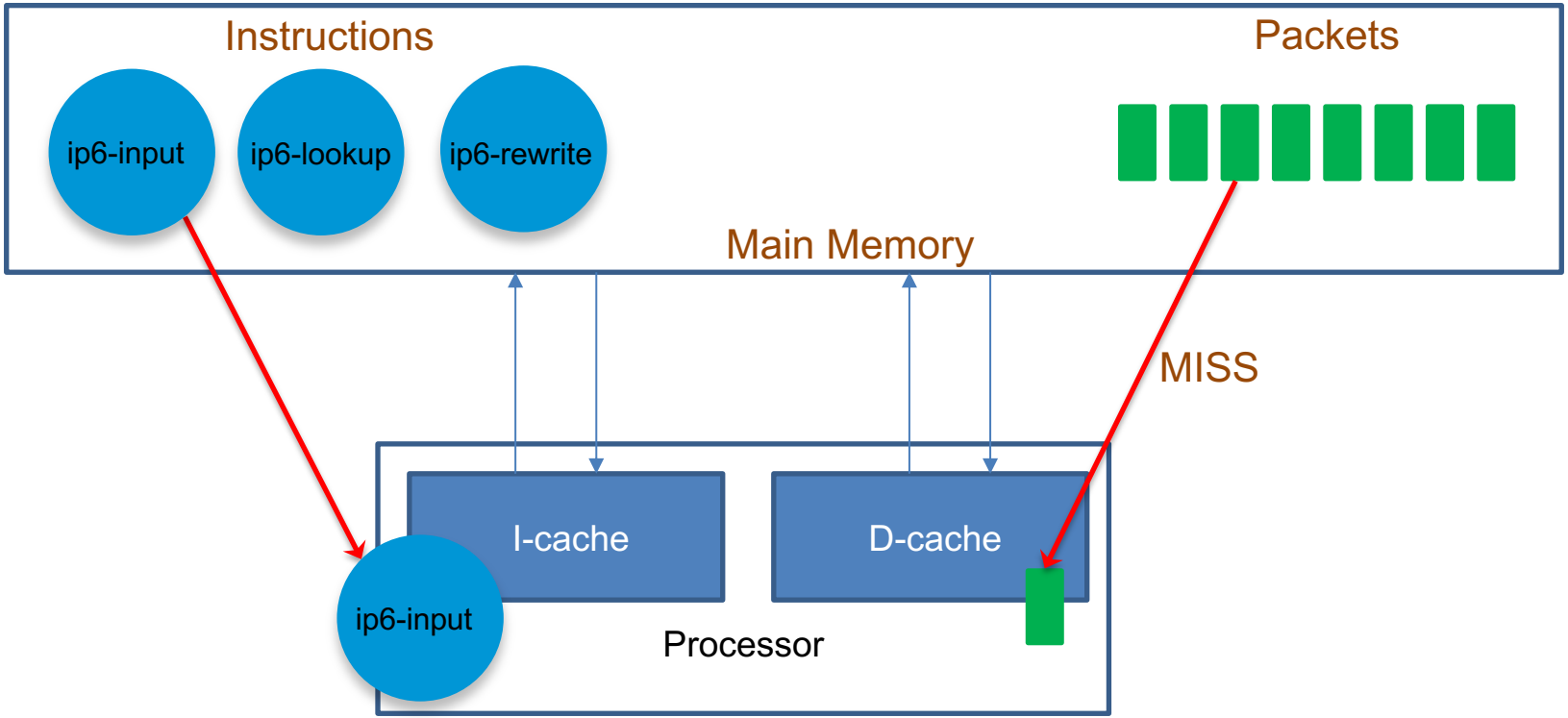
Vector packet processing



Vector packet processing



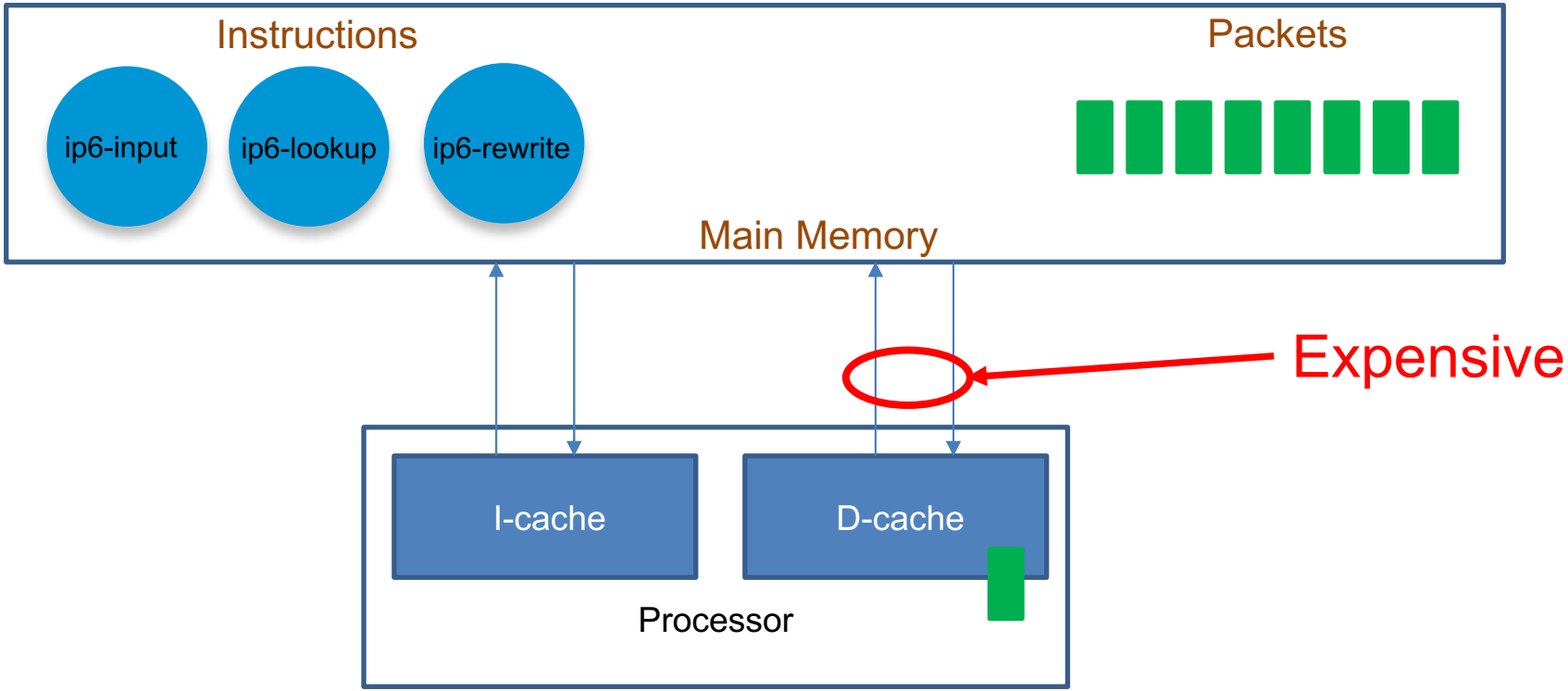
Vector packet processing



Vector packet processing

- Consume up to 256 packets at a time from device rx ring
- Invoke one node function at a time against this ‘vector’
- Fixes I-cache thrashing problem
  - Graph node dispatch functions iterate over up to 256 elements
  - Processing first packet ‘warms up’ the I-cache
  - Remaining packets all hit the I-cache
    - Reduces I-cache miss stalls by up to 2 orders of magnitude
- Downside: increase of D-Cache misses.. but
  - Given a vector of packets
    - Pipeline and prefetch to cover dependent read latency

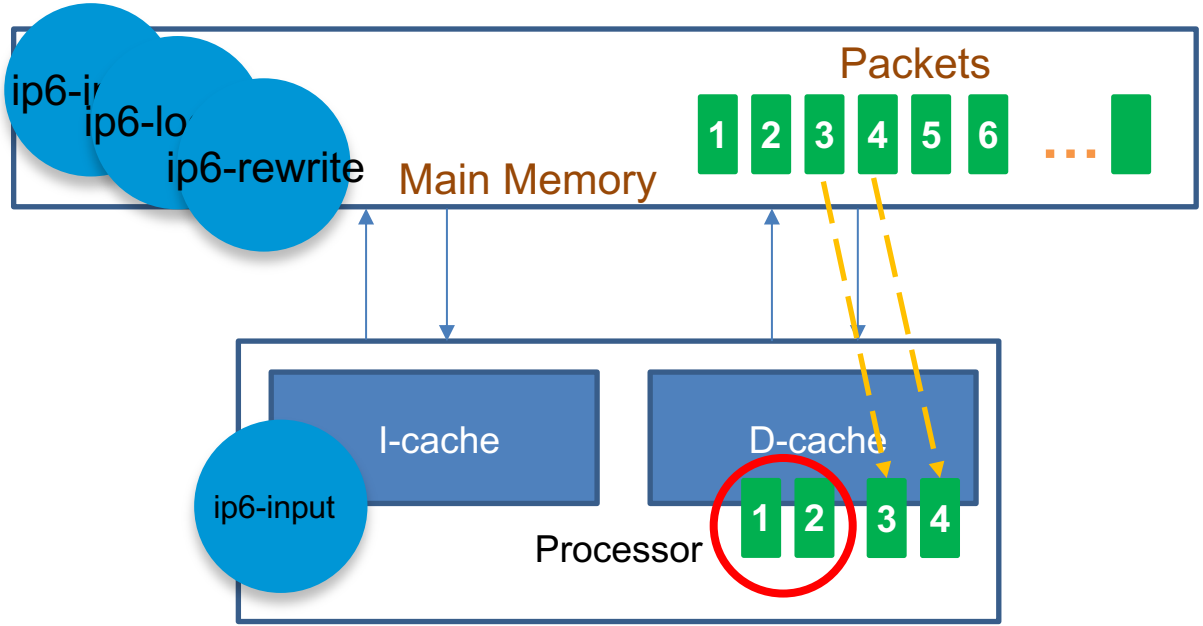
Vector packet processing – D-Cache



Vector packet processing – D-Cache

Example: Processing packet 1 & 2

Might have a cache miss for packet 1 & 2



VPP node pseudocode

while packets in vector

while 4 or more packets

PREFETCH #3 and #4

PROCESS #1 and #2

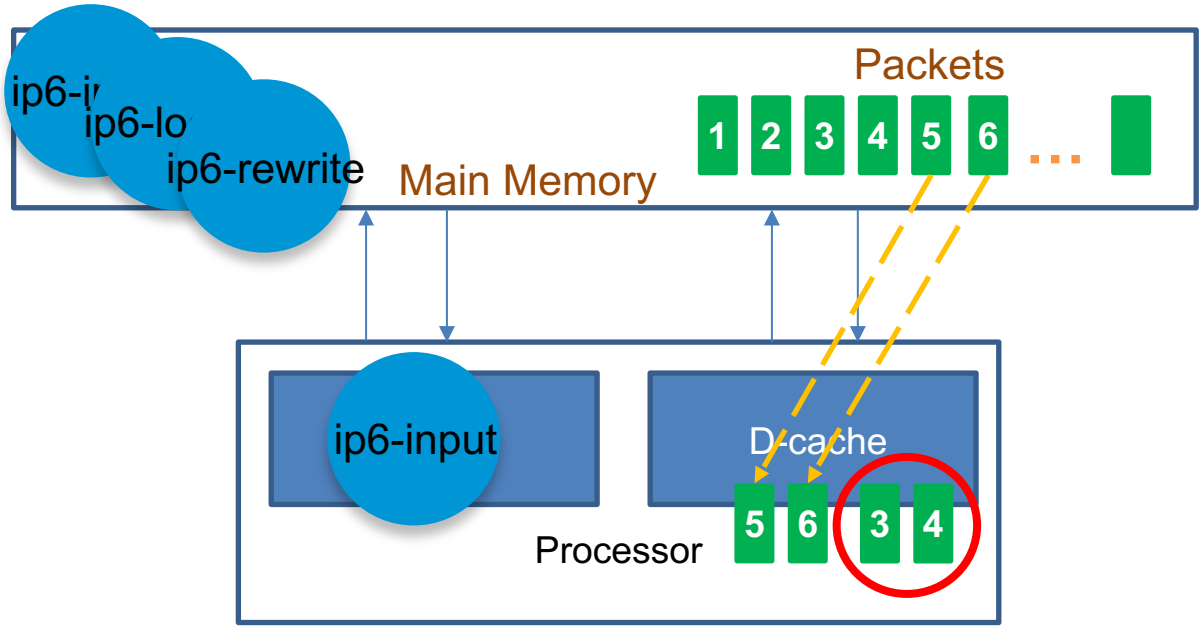
while any packets

<as above but single packet>

Vector packet processing – D-Cache

Example: Processing packet 3 & 4

The cost of the first D-cache miss is amortized by the subsequent D-cache hits.



VPP node pseudocode

while packets in vector

while 4 or more packets

PREFETCH #5 and #6

PROCESS #3 and #4

while any packets

<as above but single packet>

Extend VPP with plugins

- Plugins are first class citizens
- Plugins can:
  - Add nodes
  - Add API
  - Re-arrange the graph
- Most VPP features are implemented as in-tree plugins
  - And then loaded at run-time via `dlopen()`

Develop out-of-tree plugin

- Examples are base don the hicn plugin
- Code
  - https://github.com/FDio/hicn/tree/master/hicn-plugin
- Docs
  - https://hicn.readthedocs.io/en/v22.02-rc0/vpp-plugin.html

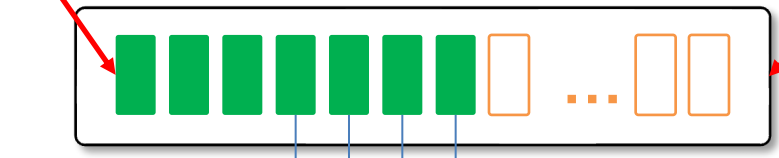
Develop out-of-tree plugin

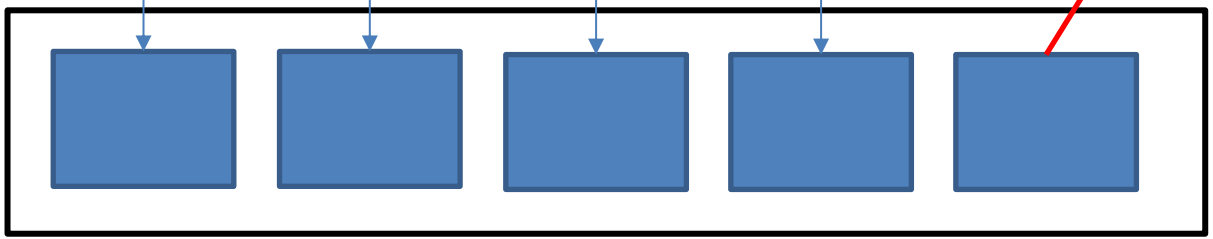
- VPP Structures
- Design your nodes
- Insert your nodes in the vlib graph
- Compile and install your plugin

VPP Base Structures

The vector of packets is called FRAME

Each element is called VECTOR





A vector is an index to a vlib_buffer_t

```
struct vlib_buffer_t
{
    ...
    u8 data[0];
    ...
}
```

Pointer to packet data.

Develop out-of-tree plugin

- VPP Structures
- Design your nodes
- Insert your nodes in the vlib graph
- Compile and install your plugin

Design your nodes

- Follow VPP Style
  - Multi-loop, Branch prediction, Function flattening, Lock-free structures
- Implement processing function that
  - “Moves vectors” from your node’s frame to the next node’s frame
  - Processes packets as YOU want
- Add whatever else you need
  - Supporting Functions, macros, variables, etc.. (C code)

Register your node

- Each node must be registered to VPP through VLIB_REGISTER_NODE macro

```
/*
* Node registration for the data forwarder node
*/
VLIB_REGISTER_NODE(hicn_data_fwd_node) =
{
    .function = hicn_data_node_fn,
    .name = "hicn-fwd",
    .type = VLIB_NODE_TYPE_INTERNAL,
    .n_errors = ARRAY_LEN(hicn_data_fwd_error_strings),
    .error_strings = hicn_data_fwd_error_strings,
    .n_next_nodes = HICN_DATA_FWD_N_NEXT,
    /* edit / add dispositions here */
    .next_nodes = {
      [HICN_DATA_FWD_NEXT_V4_LOOKUP] = "ip4-lookup",
      [HICN_DATA_FWD_NEXT_ERROR_DROP] = "error-drop",
    },
};
```

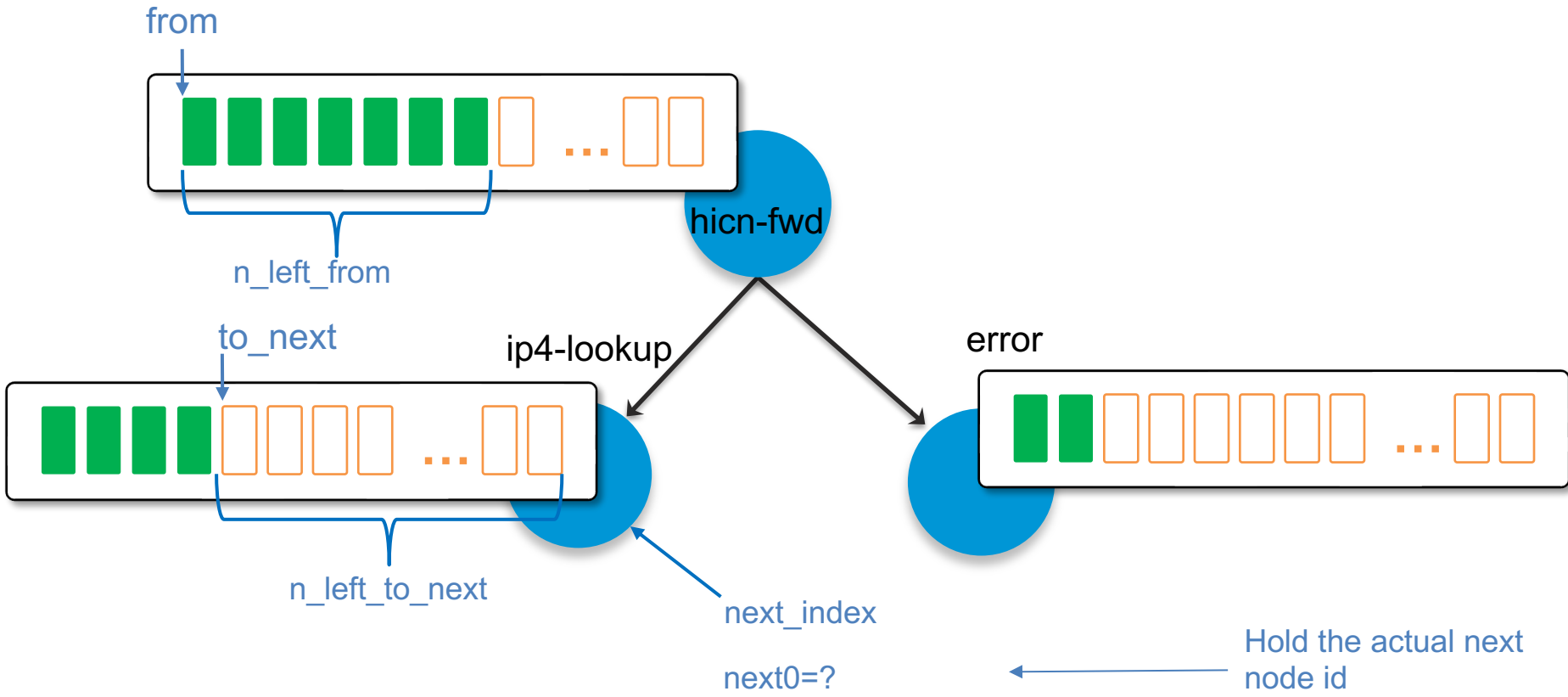
Node processing function

Name of the node

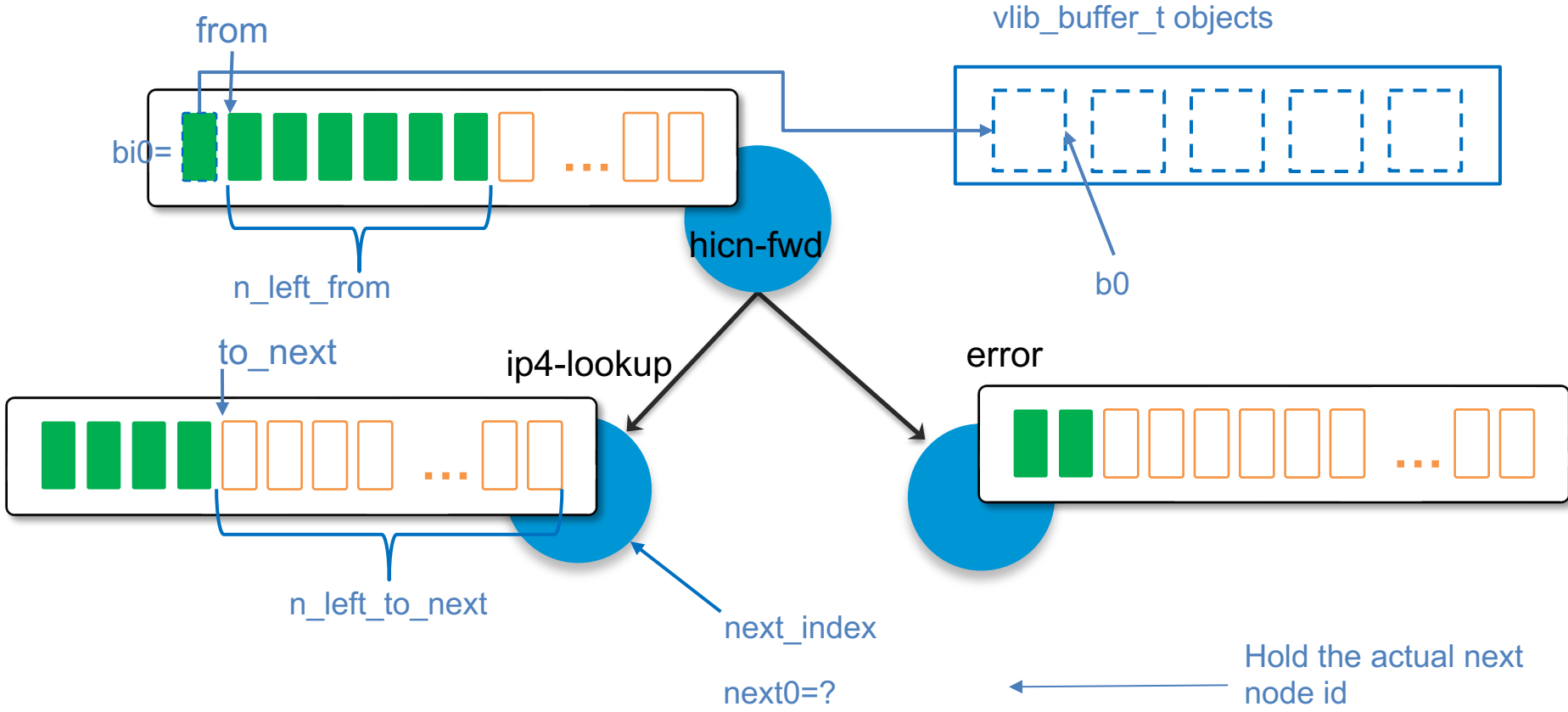
Type of node

Next nodes in the Vpp graph

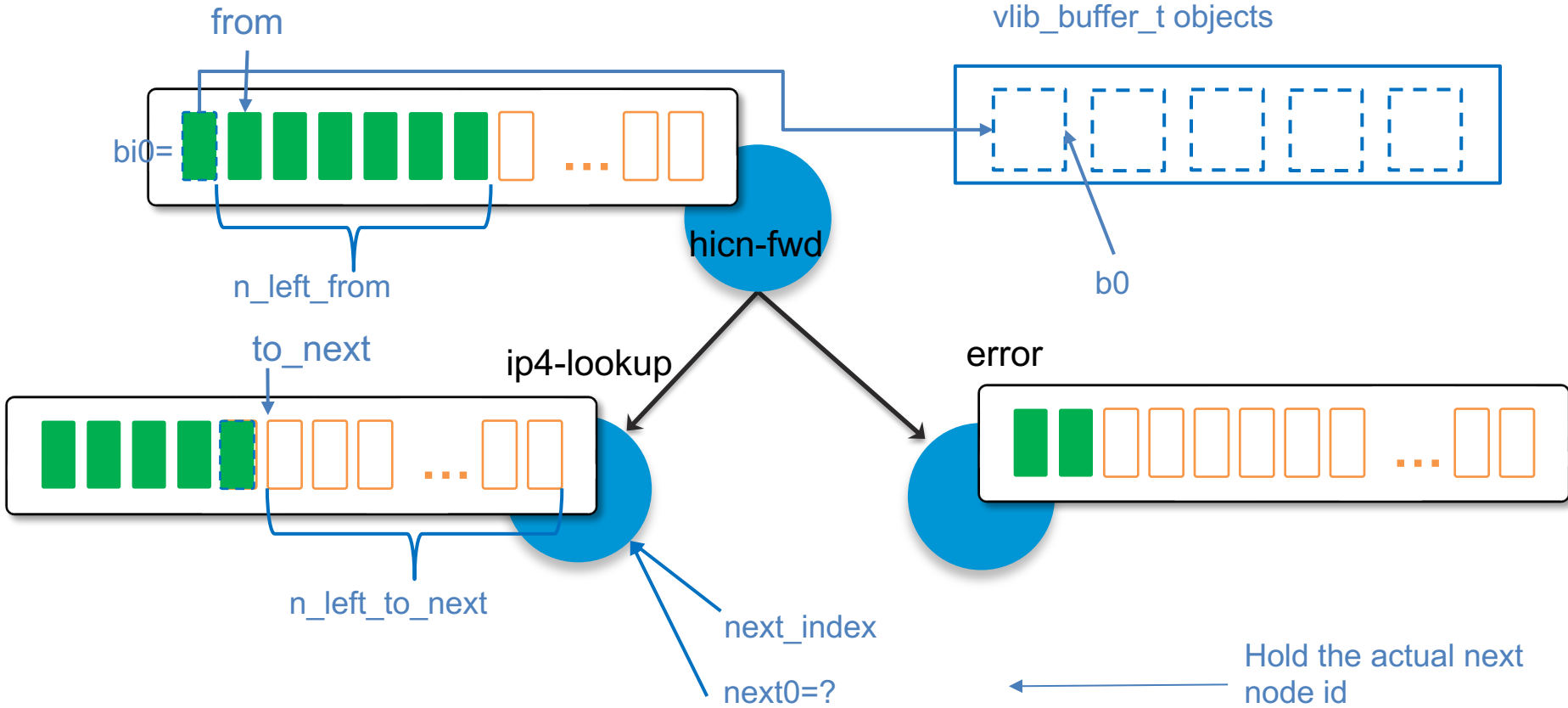
Process Packets



Process Packets



Process Packets



Register Plugin

- VLIB_PLUGIN_REGISTER() registers the plugin with the current VPP
- VLIB_INIT_FUNCTION() registers the function that will be called to initialize the plugin

```
VLIB_INIT_FUNCTION (hicn_init);

VLIB_PLUGIN_REGISTER () = { .description = "hICN forwarder" };
```

Develop out-of-tree plugin

- VPP Structures
- Design your nodes
- Insert your nodes in the vlib graph
- Compile and install your plugin

Insert your nodes in the vlib graph

- Direct all the packets from one interface
  - vnet_hw_interface_rx_redirect_to_node()
- Capture packets with a particular ethertype
  - ethernet_register_input_type()
- Packet for new protocol on top of IP
  - ip4_register_protocol()
- Packet sent to a specific UDP port
  - udp_register_dst_port()
- Direct all packets from one ip prefix
  - Create your own Data Path Object (i.e. result of a FIB lookup)

Compile and install your plugin

- VPP Structures
- Design your nodes
- Insert your nodes in the vlib graph
- Compile and install your plugin

Install Dependencies

- Install VPP in your system
  - https://s3-docs.fd.io/vpp/22.06/gettingstarted/installing
- Install a build system for your project
  - CMake is a good candidate for building C projects
    - You can use also other tools
  - Used also by VPP

CMake

- Set VPP architecture/dependent optimizations compilation flags

```
##############################################################
# Compiler Options
##############################################################
set(MARCH_COMPILER_OPTIONS -march=corei7 -mtune=corei7-avx)

set(COMPILER_OPTIONS
  ${DEFAULT_COMPILER_OPTIONS}
  ${MARCH_COMPILER_OPTIONS}
  PRIVATE "-Wno-address-of-packed-member"
)
```

CMake

- If compiling in debug mode, do not forget to define CLIB_DEBUG

```
if (${CMAKE_BUILD_TYPE} MATCHES "Debug")
  list(APPEND COMPILE_DEFINITIONS
    "-DCLIB_DEBUG"
  )
endif()
```

CMake

- Generate and install the API files
  - Use installed vppapigen and vapi_{c_,cpp}gen.py

```
##############################################################
# VPP API Generation
##############################################################
execute_process(
  COMMAND ${VPP_HOME}/bin/vppapigen ...
  COMMAND ${VPP_HOME}/bin/vppapigen JSON ...
)
execute_process(
  COMMAND ${VPP_HOME}/bin/vapi_c_gen.py ${PROJECT_BINARY_DIR}/vapi/hicn.api.json
  COMMAND ${VPP_HOME}/bin/vapi_cpp_gen.py ${PROJECT_BINARY_DIR}/vapi/hicn.api.json
)
install(FILES ${PROJECT_BINARY_DIR}/vapi/hicn.api.json
  DESTINATION ${CMAKE_INSTALL_DATAROOTDIR}/vpp/api/plugins
)
```

CMake

- Build plugin
- No need to link it against VPP libs
  - Symbols will be resolved at dlopen() time
- Install plugin
- In order for VPP to find it, you can:
  - Install it in the default plugin folder (/usr/lib/vpp_plugins)
  - Update VPP conf file (/etc/vpp/startup.conf) and add the installation folder of your plugin

```
add_library(hicn_plugin
  SHARED
  ${LIBHICN_FILES}
  ${HICN_PLUGIN_SOURCE_FILES}
  ${HICN_API_GENERATED_FILES}
  ${HICN_VAPI_GENERATED_FILES}
)

set_target_properties(hicn_plugin
  PROPERTIES
  LINKER_LANGUAGE C
  INSTALL_RPATH ${VPP_INSTALL_PLUGIN}
  PREFIX ""
)
```

Testing

- Writing unit tests for a VPP plugin may not be trivial
  - It works fine only if the tested class does not use VPP data structure
- VPP data structures are likely not initialized
  - The majority of VPP data structs exposed by VPP libs are initialized by the VPP main executable
  - You cannot just use them in your unit tests

Testing - Example

- We want to test a data structure that uses clib_bihash
  - https://github.com/FDio/hicn/blob/master/hicn-plugin/src/pcs.h
- clib_bihash is located in libvppinfra, but its memory is initialized by the vpp main executable
- If we try to use it out of VPP, the program segfaults

Testing

- To make unit testing work, we need to:
  - Initialize the memory used by the VPP data structure
  - Perform the rest of VPP initialization
- The test framework should do it as a first operation before running the test suites
  - https://github.com/FDio/hicn/blob/master/hicn-plugin/src/test/vpp.c
- Unit tests should be compiled with the same compile options and definitions of the plugins
  - To exploit architecture optimizations

Questions? ☺

LF NETWORKING

LFN Developer & Testing Forum