A Review on FPGA Scheduling in Apache Hadoop 3.1.0

Yudong Tao, Tianyi Wang

21st Jun, 2018

1 Introduction

After Apache Hadoop 3.1.0 is released on 12th Sept, 2017, a wide range of resource types are supported by the master branch of YARN[1] (an important component of Hadoop) as the task YARN-3926[2] is resolved. Based on it, YARN-5983[3] plays an important role in the realization of FPGA support. The guideline to configure FPGA on YARN can be found at https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/UsingFPGA.html. For now, YARN only support FPGA shipped with ”IntelFpgaOpenclPlugin”. FPGA is regarded as countable resources depending on sub-task 12 in YARN-3926, so one device cannot be used for multiple programs.

2 FPGA Schedule Workflow

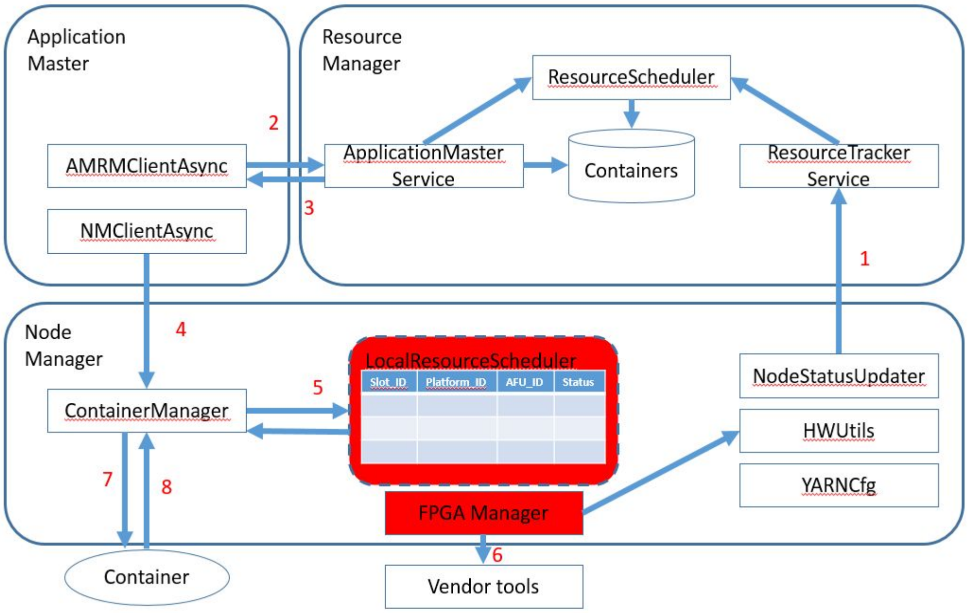

Figure 1: FPGA resource management workflow in Hadoop

As shown in the Fig. 1, the workflow of FPGA Hadoop resource management is as follows [1]:

1. Configure FPGA devices in YARN configration (resource-type.xml and node-resources.xml). And NodeManager (NM) initialize local FPGA resource scheduler with allowed FPGA devices and report the profile to ResourceManager (RM);
2. Application master requests container with a resource profile containing certain numbers of FPGA devices (may have different type). Currently the container just consists numeric FPGA resource but no detailed device information;

[1] Description can be found at http://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/YARN.html

[2] Information can be found at https://issues.apache.org/jira/browse/YARN-3926

[3] Information can be found at https://issues.apache.org/jira/browse/YARN-5983

3. RM allocate the containers;
4. ApplicationManager (AM) sets the IP UUID/name in container environment and sends requests to NM to launch the allocated containers. Only support one IP for all devices now;
5. Prior to each container launch, NM will ask FPGA local scheduler for which FPGA devices can be allocated and use vendor specific plugin to configure the device for container;
6. If a FPGA slots needs re-configuration, NM will use vendor plugin to reconfigure or isolate the FPGA slots if needed;
7. Launch the container;
8. After the container complete, NM will inform the local FPGA resource scheduler to clean up the FPGA resources;

3 Implementation Details

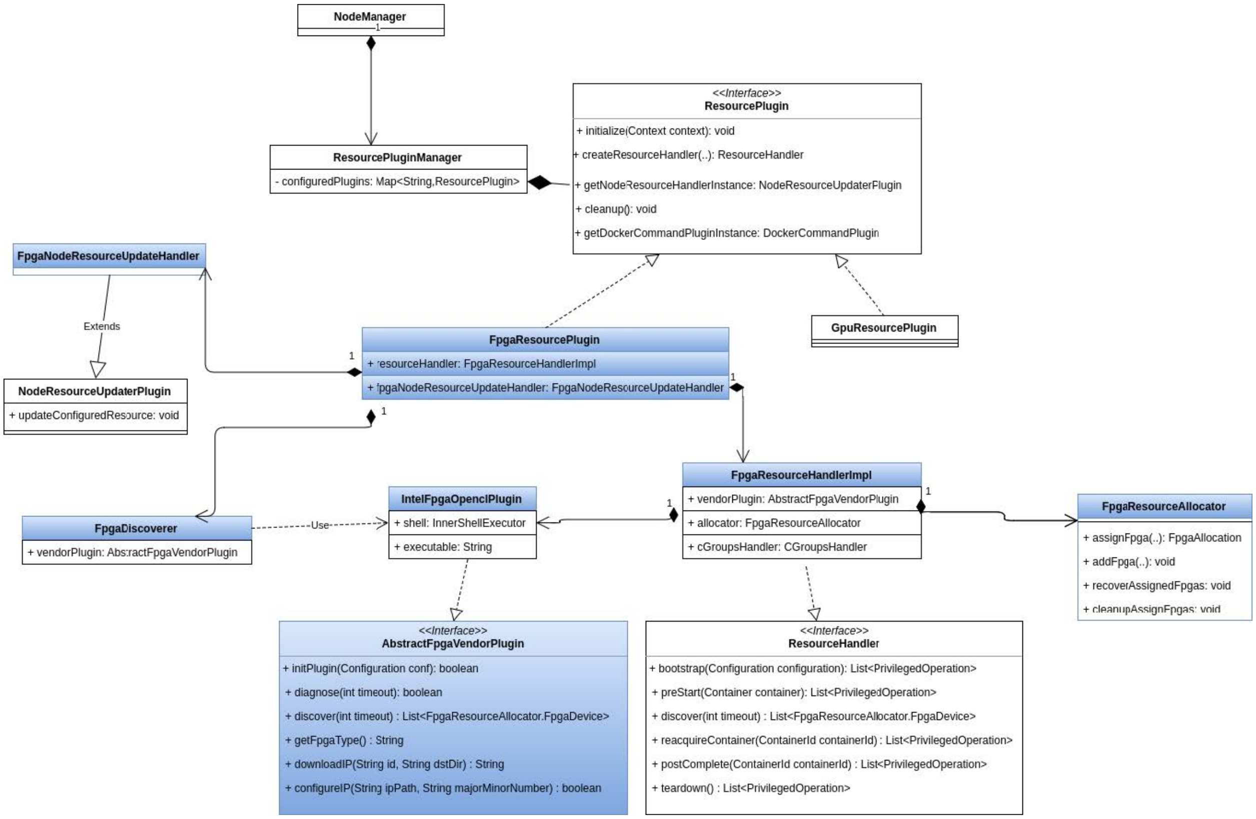
NodeManager
1
ResourcePluginManager
- configuredPlugins: Map<String,ResourcePlugin>
<<Interface>>
ResourcePlugin
+ initialize(Context context): void
+ createResourceHandler(..): ResourceHandler
+ getNodeResourceHandlerInstance: NodeResourceUpdaterFlugin
+ cleanup(): void
+ getDockerCommandPluginInstance: DockerCommandPlugin
FpgaNodeResourceUpdateHandler
Extends
NodeResourceUpdaterPlugin
+ updateConfiguredResource: void
FpgaResourcePlugin
+ resourceHandler: FpgaResourceHandlerImpl
+ fpgaNodeResourceUpdateHandler: FpgaNodeResourceUpdateHandler
GpuResourcePlugin
1
1
1
FpgaDiscoverer
+ vendorPlugin: AbstractFpgaVendorPlugin
Use
IntelFpgaOpenclPlugin
+ shell: InnerShellExecutor
+ executable: String
FpgaResourceHandlerImpl
+ vendorPlugin: AbstractFpgaVendorPlugin
+ allocator: FpgaResourceAllocator
+ cGroupsHandler: CGroupsHandler
1
1
FpgaResourceAllocator
+ assignFpga(..): FpgaAllocation
+ addFpga(..): void
+ recoverAssignedFpgas: void
+ cleanupAssignFpgas: void
<<Interface>>
AbstractFpgaVendorPlugin
+ initPlugin(Configuration conf): boolean
+ diagnose(int timeout): boolean
+ discover(int timeout) : List<FpgaResourceAllocator.FpgaDevice>
+ getFpgaType() : String
+ downloadIP(String id, String dstDir) : String
+ configureIP(String ipPath, String majorMinorNumber) : boolean
<<Interface>>
ResourceHandler
+ bootstrap(Configuration configuration): List<PrivilegedOperation>
+ preStart(Container container): List<PrivilegedOperation>
+ discover(int timeout) : List<FpgaResourceAllocator.FpgaDevice>
+ reacquireContainer(ContainerId containerId) : List<PrivilegedOperation>
+ postComplete(ContainerId containerId) : List<PrivilegedOperation>
+ teardown() : List<PrivilegedOperation>

Figure 2: FPGA resource management design UML

As shown in the Fig. 2, the design of FPGA is based on ResourcePlugin which regards FPGA as countable resources. The FPGA related codes can be found mainly in hadoop-yarn-server-nodemanager project[4]. Both FpgaResourceAllocator and FpgaResourceHandlerImpl can be found in "containermanager/linux/resources/fpga/" while FpgaNodeResourceUpdateHandler, FpgaResourcePlugin, and IntelFpgaOpenclPlugin

[4] Source codes are with in Hadoop 3.1.0, avaliable at http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-3.1.0/hadoop-3.1.0-src.tar.gz

can be found in “containermanager/resourceplugin/fpga/”. IntelFpgaOpenclPlugin is currently the only implementation of the abstract class AbstractFpgaVendorPlugin. The roles of each class are as follows:

- Each “FpgaResourcePlugin” is the interface of a FPGA device, which implements ResourcePlugin to redirect the resource discovery fairs to “FpgaDiscovery” and the allocation/isolation fairs to “FpgaResourceHandlerImpl”. All of these are using “AbstractFpgaVendorPlugin” as backend to accomplish the real task by calling corresponding commands from FPGA toolchain.

```
public class FpgaResourcePlugin implements ResourcePlugin {
  private static final Log LOG = LogFactory.getLog(FpgaResourcePlugin.class);

  private ResourceHandler fpgaResourceHandler = null;

  private AbstractFpgaVendorPlugin vendorPlugin = null;
  private FpgaNodeResourceUpdateHandler fpgaNodeResourceUpdateHandler = null;

  ...
}
```

- The “FpgaDiscovery” uses “discover” function in “VendorPlugin” to get a list of devices and match them with the configurations. A list of accessible devices matched with configuration are returned.

```
public class FpgaDiscoverer {
  private Configuration conf = null;

  private AbstractFpgaVendorPlugin plugin = null;

  // shell command timeout
  private static final int MAX_EXEC_TIMEOUT_MS = 10 * 1000;

  // get avialable devices minor numbers from toolchain or static configuration
  public synchronized List<FpgaResourceAllocator.FpgaDevice> discover() throws ResourceHandlerException {
    List<FpgaResourceAllocator.FpgaDevice> list;
    String allowed = this.conf.get(YarnConfiguration.NM_FPGA_ALLOWED_DEVICES);
    // whatever static or auto discover, we always needs the vendor plugin to discover. For instance, IntelFpgaOpenclPlugin need to setup a mapping of <major:minor> to <aliasDevName>
    list = this.plugin.discover(MAX_EXEC_TIMEOUT_MS);
    ...
    if (allowed.matches(”(\\d,)*\\d”)){
      String[] minors = allowed.split(”,”);
      // remove the non-configured minor numbers
      ...
      }
      // if the count of user configured is still larger than actual, continue with a warning
      if (list.size() != minors.length) {
        ...
      }
    }
    return list;
  }

  ...
}
```

- The “FpgaResourceHandlerImpl” implements the functions in “ResourceHandler”, including “bootstrap”, “prestart”, “reacquireContainer”, and “postComplete” while “teardown” is not accomplished.

```java
public class FpgaResourceHandlerImpl implements ResourceHandler {
  private AbstractFpgaVendorPlugin vendorPlugin;
  private FpgaResourceAllocator allocator;
  private CGroupsHandler cGroupsHandler;

  public String getRequestedIPID(Container container) {
    String r= container.getLaunchContext().getEnvironment().
        get(REQUEST_FPGA_IP_ID_KEY);
    return r == null ? "" : r;
  }

  @Override
  public List<PrivilegedOperation> bootstrap(Configuration configuration) throws ResourceHandlerException {
    // The plugin should be initilized by FpgaDiscoverer already
    vendorPlugin.initPlugin(configuration);
    ...
    // Get avialable devices minor numbers from toolchain or static configuration
    List<FpgaResourceAllocator.FpgaDevice> fpgaDeviceList = FpgaDiscoverer.getInstance().discover();
    allocator.addFpga(vendorPlugin.getFpgaType(), fpgaDeviceList);
    this.cGroupsHandler.initializeCGroupController(CGroupsHandler.CGroupController.DEVICES);
    return null;
  }

  @Override
  public List<PrivilegedOperation> preStart(Container container) throws ResourceHandlerException {
    // 1. Get requested FPGA type and count, choose corresponding FPGA plugin(s)
    // 2. Use allocator.assignFpga(type, count) to get FPGAAllocation
    // 3. If required, download to ensure IP file exists and configure IP file for all devices
    List<PrivilegedOperation> ret = new ArrayList<>();
    String containerIdStr = container.getContainerId().toString();
    Resource requestedResource = container.getResource();
    ...
    // allocate even request 0 FPGA because we need to deny all device numbers for this container
    FpgaResourceAllocator.FpgaAllocation allocation = allocator.assignFpga(
        vendorPlugin.getFpgaType(), deviceCount,
        container, getRequestedIPID(container));
    LOG.info("FpgaAllocation:" + allocation);
    if (deviceCount > 0) {
      ipFilePath = vendorPlugin.downloadIP(getRequestedIPID(container), container.getWorkDir(),
          container.getResourceSet().getLocalizedResources());
      ...
    }

    //isolation operation
    ret.add(new PrivilegedOperation(
        PrivilegedOperation.OperationType.ADD_PID_TO_CGROUP,
        PrivilegedOperation.CGROUP_ARG_PREFIX
        + cGroupsHandler.getPathForCGroupTasks(
        CGroupsHandler.CGroupController.DEVICES, containerIdStr)));
    return ret;
  }

  @Override
  public List<PrivilegedOperation> reacquireContainer(ContainerId containerId)
    throws ResourceHandlerException {
    allocator.recoverAssignedFpgas(containerId);
    return null;
  }

  @Override
  public List<PrivilegedOperation> postComplete(ContainerId containerId) throws
    ResourceHandlerException {
    allocator.cleanupAssignFpgas(containerId.toString());
    cGroupsHandler.deleteCGroup(CGroupsHandler.CGroupController.DEVICES,
        containerId.toString());
    return null;
  }

  @Override
  public List<PrivilegedOperation> teardown() throws ResourceHandlerException {
    return null;
  }

  ...
}
```

- The “FpgaResourceAllocator” contains a embedded class “FpgaDevice”, which shows the metadata used for an FPGA device. The current support type is “IntelFpgaOpencl”.

```
public static class FpgaDevice implements Comparable<FpgaDevice>, Serializable {

  private static final long serialVersionUID = 1L;

  private String type;
  private Integer major;
  private Integer minor;
  // IP file identifier. matrix multiplication for instance
  private String IPID;
  // the device name under /dev
  private String devName;
  // the alias device name. Intel use acl number acl0 to acl31
  private String aliasDevName;
  // lspci output's bus number: 02:00.00 (bus:slot.func)
  private String busNum;
  private String temperature;
  private String cardPowerUsage;

  ...
}
```

- The “IntelFpgaOpenclPlugin” is an implementation of the abstract vendor, wrapping the commands of Intel OpenCL FPGA toolchain, aocl, in the class and provide interface to execute them via a shell.

If we want to add support for Xilinx FPGA in Hadoop, basically an implementation for the Xilinx FPGA toolchain is required. Moreover, the FpgaResourceHandlerImpl is currently designed for Intel toolchain and has not been generalized. If multiple toolchains are expected, the classes need modifications to generalize to provide interfaces for allocation and isolation. For our PAI project, if only Xilinx FPGAs are of concern, then we can use a similar structure to implement everthing while the toolchain is replaced by those for Xilinx. If we want to implement from scratch, the architecture of Hadoop YARN is a good starting point.

4 Official Test Report [2]

In the recent Hadoop YARN update, support for FPGA has been added. With this, the big data and AI application can request FPGA resource easily and evolve with the “App + Accelerator” trend. Currently, only Intel FPGA is supported. The following functionalities have been implemented:

- Discover one type of vendor FPGA devices
- Allocate FPGA devices with local scheduler in NM
- Isolate vendor FPGA devices with cgroups (native FPGA module)
- Launch FPGA application with simple container under cgroup
- Reacquire container after NM restart
- One default vendor plugin

FPGA devices should be re-programmed before container launch with correct IP file so that can the application invoke native library/runtime to work. Environment variable “REQUESTED_FPGA_IP_ID” is provided for such task. If set, the value of it should be one string like “matrix_mul” indicating an ID of the IP file, which FPGA devices should be re-programmed before container launch with correct IP file so that can the application invoke native library/runtime to work.

The following test cases has also been applied and passed for this update:

- Test FPGA demo application with FPGA and IP env specified
- Test FPGA demo application with FPGA and IP already programmed
- Test FPGA demo application with FPGA but no IP env specified
- Test demo application with FPGA but no IP file uploaded
- Test demo application without FPGA devices requested
- No FPGA resource plugin configuration

References

[1] Apache Hadoop Development, “Add FPGA resource provider for YARN-3926.” https://jira.apache.org/jira/secure/attachment/12864436/YARN-5983-Support-FPGA-resource-on-NM-side_v1.pdf, 2017.

[2] Z. Tang, “YARN-5983 end-to-end test report.” https://jira.apache.org/jira/secure/attachment/12899283/YARN-5983_end-to-end_test_report.pdf, 2017.