A new scheduling algorithm for the LHCb upgrade trigger application

Ekaterina Govorkova[1], Christoph Hasse[2,3], Rosen Matev[2], Niklas Nolte[2,3], Sebastien Ponce[2], Gerhard Raven[1] and Sascha Stahl[2]

[1]Nikhef, Amsterdam, Netherlands
[2]CERN, Geneva, Switzerland
[3]TU Dortmund, Dortmund, Germany

E-mail: `nnolte@cern.ch`

Abstract. During Run 3 of the LHC the LHCb detector will process a 30 MHz event rate with a full detector readout followed by a software trigger. To deal with the increased computational requirements, the software framework is reviewed and optimized on a large scale. One challenge is the efficient scheduling of $O(10^3)$-$O(10^4)$ algorithms in the High Level Trigger (HLT) application. This document describes the design of a new algorithm scheduler which allows for static-order intra-event scheduling with minimum complexity while still providing the required flexibility.

1. Introduction

During Run 3 of the LHC the LHCb experiment [1] will operate at a luminosity of $2 \times 10^{33} cm^{-2} s^{-1}$, which is a fivefold increase with respect to Run 2 [2]. The hardware trigger that currently reduces the data rate from 30 to 1 MHz will be removed. The LHCb experiment is going to operate a full detector readout followed by a software trigger [3].

Its main task is to select signal events and candidates based on reconstructed objects. It comprises two stages: HLT1 and HLT2. During the first stage a partial reconstruction of the event is performed and, if selected, the event is deferred to a buffer. The alignment and calibration of the detector are performed on the buffered data. Subsequently, the event undergoes full offline quality reconstruction in HLT2, where it is either rejected or sent to permanent storage for further analysis.

Compared to Run 2, HLT1 will need to process a 30 times higher event rate on a similarly sized farm, estimated to comprise about 1000 CPU nodes [4]. These conditions require significant optimizations in the software framework. An algorithm scheduler implementation to be used in the upgrade trigger is described in this document. It allows static intra-event algorithm ordering and inter-event concurrency.

The document is structured as follows: Section 2 contains a general description of the trigger selection framework of the LHCb experiment and Section 3 explains the scheduler implementation in detail.

2. Trigger Selection Framework

Both software trigger stages make use of trigger lines to record interesting events. Trigger lines are decision chains that are defined by a user. A chain is a logical conjunction of components, henceforth referred to as algorithms. An algorithm defines input(s) and/or output(s) and performs some calculation such as selection or reconstruction of a subdetector. Each individual line decision is based on a different criteria, with some overlaps, but logically independent and self-contained.

To define a line in the current framework the user needs to list a complete sequence of algorithms. Each algorithm has to have data inputs defined and producers of these inputs have to be configured and explicitly added to the sequence before the consuming algorithm.

This changes in the upgrade framework. The goal is to give users general building blocks that are well defined, multithreading friendly and automatically handle the data flow between algorithms. At application initialization a static data and control flow graph can be generated from those blocks. The scheduler, a detailed description of which can be found in Section 3, is responsible for building the static graph, scheduling and conditionally executing algorithms in a correct order to meet data and control flow dependencies.

3. The scheduling algorithm

The scheduler uses two general types of nodes: basic and composite. A basic node manages one underlying algorithm, in this document referred to as top level algorithm (TopAlg), by ensuring the existence of its inputs and keeping track of execution states. TopAlgs may define a control flow decision by checking if some condition is fulfilled, e.g. whether there is a particle with $P_T > 5\,\mathrm{GeV}$ present. Composite nodes are of a specific logical type and define evaluation manners of their children depending on control flow decisions. These can either be basic or other composite nodes themselves. The following types are currently implemented, providing the required flexibility to properly define trigger lines.

- LAZY_AND - boolean conjunction; short-circuit evaluation
- NON_LAZY_AND - boolean conjunction; not short-circuit evaluation
- LAZY_OR - boolean disjunction; short-circuit evaluation
- NON_LAZY_OR - boolean disjunction; not short-circuit evaluation
- NOT - boolean negation

Fig. 1 shows a small example setup to define two trigger lines. **Decision** and **Line_1(2)** are composite nodes. Basic nodes are displayed in green. Both lines share the **G** node, which could in a real setup correspond to the global event cut. **P** and **F** nodes are possible prescales and additional filters.

The work of the scheduler is divided into two parts. The first part prior to the execution includes ordering the basic nodes with specific control flow constraints and resolving data dependencies. The second part is the ordered, conditional execution of TopAlgs and their data producers. Information about the nodes' execution statuses and control flow decision are propagated to their parents to be able to determine whether following basic nodes are requested to be executed.

3.1. Static node ordering during initialization time

All constraints on control flow and data dependencies are set at configuration time. Therefore, we can create a flat ordered list of basic nodes before any event loop.

All composite nodes are fully defined in the configuration, including the logical type, execution behaviour and the list of children by name. Basic nodes are inferred from the lists of children names.

Composite nodes may enforce a preserved order of their children by defining constraints in the

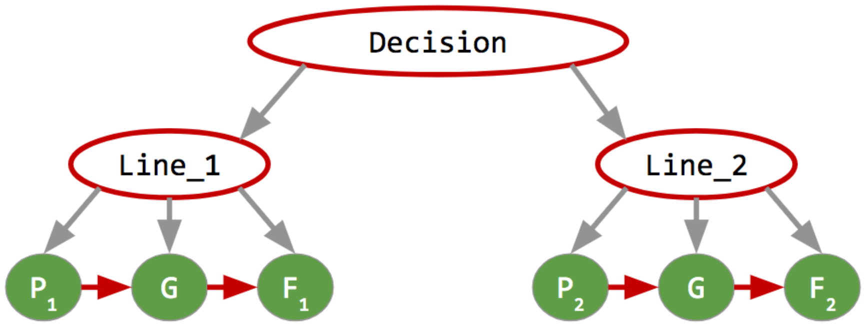

Figure 1. A small dependency tree with exemplary two trigger lines: Line_1 and Line_2. Both lines comprise prescales ($P_{1(2)}$), a global event cut (G) and a line-specific filters ($F_{1(2)}$). Line_1(2) ensure a specific order of their children as indicated by the red arrows.

form of control flow edges. These are displayed as red arrows in Fig. 1. Each edge guarantees a specific execution order between the corresponding nodes. Edges between composite nodes are interpreted as the complete set of pairwise edges between their respective children.
An ordered list for execution of basic nodes is constructed by moving a node from an unordered list if all its control flow dependencies are met. This loop continues until the unordered list is cleared. In the example shown in Fig. 1, the initial list of the basic nodes contains P_1, G, F_1, P_2 and F_2. Edges in Fig. 1 require P_1 (P_2) to be executed before G to be executed before F_1 (F_2). This results in the ordering of the list of all basic nodes as shown in Fig. 2. In addition to the edges that arise directly from the composite nodes, the scheduler framework allows additional edges defined by a user to control the execution order.

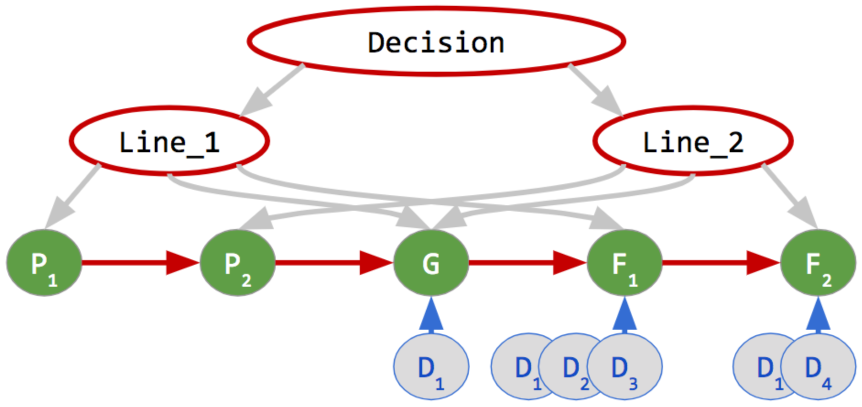

Figure 2. The example configuration scheme with two trigger lines after their children has been ordered according to the edges.

Once the list of all control flow nodes is ordered, the data dependencies need to be resolved. There is a one-to-one correspondence between the output and the algorithm that produces it. This allows to restore the chain of data producers from the input(s) specified for each given algorithm. An ordered list of data producers for each basic node is generated at initialization time.

3.2. Conditional algorithm execution in the event loop

After the static execution order is determined, the second part of the scheduler work is performed within the event loop.
We iterate over the ordered list of basic nodes and perform the following steps for each node:

(i) Is the node **requested**? If not, continue with the next node
  (a) A node is requested if at least one parent is not yet finished and requested itself. This recursion continues to the highest node, which is always requested.

(ii) Each algorithm in the ordered list of data producers for this node is **executed** if that did not happen already within another node.

(iii) The control flow **decision** of the TopAlg **is propagated** to all parents of the node.

(iv) All parents **update** their own state depending on their composite node type. If one parent is considered finished, this information is propagated further up the control flow tree:
  (a) LAZY_OR (LAZY_AND) nodes short-circuit as soon as the first child returns a positive (negative) decision.
  (b) NON_LAZY nodes do not short-circuit. They finish as soon as all children finished.
  (c) The NOT node as unary operator finishes directly after its child.

4. A work sharing barrier

In an environment where multiple nodes invoke some very time consuming data producers processing different but intersecting subsets of the same data collection, one wants to avoid the same calculation being performed multiple times. It might then prove more efficient to first merge the subsets and afterwards invoke one execution of the expensive algorithm on the merged set. We implement this feature as control flow barrier, the schematic of which is shown in Fig. 3. The implementation comprises three steps: **gathering**, **executing** and **scattering**. The gather

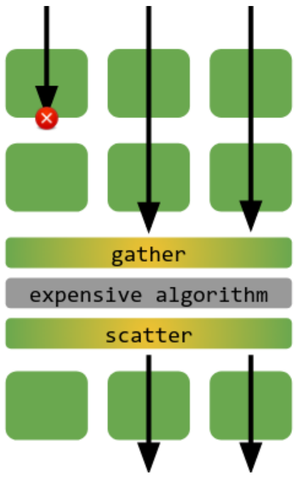

Figure 3. Schematic example of the barrier concept.

step creates a union of a variable amount of data subsets. The actual work is performed on the merged collection and is therefore blind to the possibility of a gather step beforehand. The scatter step uses the necessary information from the gatherer to split transformed data back into corresponding containers to be picked up by the trigger lines.
Via inter-line control flow edges the barrier ensures a basic node order where all trigger lines registered with it arrive at a certain point before gathering. Therefore the barrier is the (only) data producer that necessarily affects the order of control flow nodes.

5. Performance

The scheduler algorithm has been tested in multiple environments to ensure sufficient efficiency even in large and complex environments. Performance measurements were undertaken using the Intel® VTune^{TM} Amplifier [5] which allows for profiling of individual function calls in multi-threaded environments.
The first and trivial performance test is executing the HLT1 reconstruction which from the perspective of scheduling is a simple sequence of about 15 algorithms. The performance is compared to using the previously for Run 3 development purposes employed scheduling algorithm. This one is laid out for the specific task of executing linear sequences and implements no concept of control flow other than a full stop of an event. No difference in throughput could be observed comparing both algorithms.
The following tests described in this chapter are mock tests to quickly determine framework overhead when aiming for specific complexity and throughput. The next tests aims to emulate an HLT1-like environment. Test algorithms with controllable execution times are calibrated to achieve a throughput of about 30 kHz on one computing node, utilizing all 40 cores available on the given Intel Xeon E5-2630v4.
In such a scenario, comprising 10 lines with 5 basic nodes each, the relative self time consumed by the scheduler stays well under 0.5%. Doubling the number of lines while maintaining the same throughput effectively brings the self time up to nearly 1%.
The second HLT stage envisaged for the LHCb upgrade will comprise O(500) trigger lines, as it already did for Run 2. Thus, a dedicated test for this scenario is also performed. A throughput of about 1 kHz per computing node is expected for the HLT2 application. The mock test comprises 500 lines with 10 nodes each. The test shows a contribution of under 2% of the scheduling algorithm to the total execution time. However, other parts of the underlying framework that are not yet optimized obfuscate the results because they slow down the application severely when going to such a high number of algorithms.
The undertaken tests show promising results and the algorithm seems to perform sufficiently for the LHCb trigger use case in Run 3.

6. Conclusion and Outlook

A new scheduling algorithm for the upgrade software framework of LHCb was implemented. It can handle arbitrary data and control flow and supports multi-threading via inter-event concurrency.
It is the first feature complete implementation of a scheduling algorithm that seems to meet the performance requirements of LHCb's High Level Trigger. Thus, it is currently used as the default scheduling algorithm for all HLT development purposes and also planned to be used in production mode during Run 3.
In order to get a test of how this algorithm performs under realistic HLT2 conditions, a prototype of the HLT2 framework is currently built and benchmarked. The final test will be the fully fledged HLT2 configuration.

7. References

[1] The LHCb Collaboration 2008 The LHCb Detector at the LHC *JINST* **3** S08005–S08005
[2] The LHCb Collaboration 2011 Letter of Intent for the LHCb Upgrade Tech. Rep. CERN-LHCC-2011-001. LHCC-I-018
[3] The LHCb Collaboration 2018 Upgrade Software and Computing Tech. Rep. CERN-LHCC-2018-007. LHCB-TDR-017
[4] The LHCb Collaboration C 2014 LHCb Trigger and Online Upgrade Technical Design Report Tech. Rep. CERN-LHCC-2014-016. LHCB-TDR-016
[5] Intel® VTune^{TM} Amplifier 2019 URL `https://software.intel.com/en-us/vtune`