Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN
MATEMATICA E INFORMATICA

CICLO XVIII

COORDINATORE Prof. Massimiliano Mella

Adaptive Scheduling Applied to Non-Deterministic Networks of Heterogeneous Tasks for Peak Throughput in Concurrent Gaudi

Settore Scientifico Disciplinare **INF/01**

Dottorando
Dott. Illya Shapoval

Tutore
Dr. Luca Tomassetti

Cotutore al CERN
Dr. Marco Clemencic

Anni **2013/2015**

To my family

Contents

Abbreviations

CCS concurrency control system. 23–30, 39, 40, 42, 67, 73, 75

CERN European Laboratory for Particle Physics. 1–3, 87

CF control flow. 13–19, 21–24, 27, 28, 30, 41, 42, 44, 71, 72, 75, 78, 87, 88

CFS Completely Fair Scheduler. 55, 56

CPM critical path method. 45, 47, 48, 76

CPU central processing unit. 3, 36, 55–57, 59, 60, 63, 64, 83, 85, 88, 89

CR Control Ready. 10, 17, 24, 41, 42, 75

CS context switching. 56, 60

DF data flow. 13–15, 17, 19–24, 27, 30, 40, 41, 44, 47, 71, 72, 74, 78, 87, 88

DM decision making. xi, 19, 26, 27

DR Data Ready. 10, 17, 24, 37, 38, 40–42, 74, 75

EX Executed. 11

F Failed. 10

FSM finite-state machine. 17, 24, 71, 72

GPGPU general-purpose graphics processing units. 54

HEP high energy physics. 1, 2, 5, 31, 36, 37, 45, 53, 54, 56, 63, 71

I Initial. 10, 17, 24, 41, 42, 75

I/O input/output. 3, 36, 54

LHC Large Hadron Collider. 2, 36, 87

OS operating system. 55

RAM random-access memory. 3, 36, 54

SCH Scheduled. 10, 37

TBB Intel® Threading Building Blocks. 8, 9, 55–57, 60–64, 78

List of Deliverables

Introduction

As much the e-Science revolutionizes the scientific method in its empirical research and scientific theory, as it does pose the ever growing challenge of accelerating data deluge. Among such data intensive fields of science as genomics, connectomics, astrophysics, environmental research and many others is the field of modern high energy physics (HEP). The immense data sets, generated in collisions of beams at particle accelerators, require scalable high-throughput software that is able to cope with associated computational challenges.

Historically, a lot of elaborate HEP software were designed being inherently sequential. As in many other fields, this was a result of so called *serial illusion*, being cultivated by the serial semantics of computer design despite the parallel nature of the very hardware. For decades, this illusion has been successfully maintained by computer architects until the rapid growth of internal processors parallelism started. By this time, though, serialization wove into the very fabric of models, languages, practices and tools the programmers use [1].

One such striking example is represented by Gaudi [2] – a software framework for building HEP data processing applications, developed at the European Laboratory for Particle Physics (CERN). The principles of composability and reusability the framework was built upon were the key factors of its success. The following HEP experiments settled on the track of Gaudi:

- HARP – Hadron Production experiment @CERN (2000–2002);
- FGST – Fermi Gamma-ray Space Telescope, formerly known as GLAST (2008–);

- MINERνA – Main Injector experiment for ν-A @Fermilab (2010–);
- DayaBay – Daya Bay Reactor Neutrino experiment (2011–);
- ATLAS – A Toroidal LHC ApparatuS experiment @CERN (2009–);
- LHCb – Large Hadron Collider beauty experiment @CERN (2009–);
- LZ – LUX-ZEPLIN dark matter experiment @SURF/LBNL (under construction).

The LHCb [3] and ATLAS experiments (see figure 1) are the two of four major experiments at the Large Hadron Collider (LHC) at CERN. The LZ – a next generation dark matter experiment – has become a new customer of GAUDI recently [4].

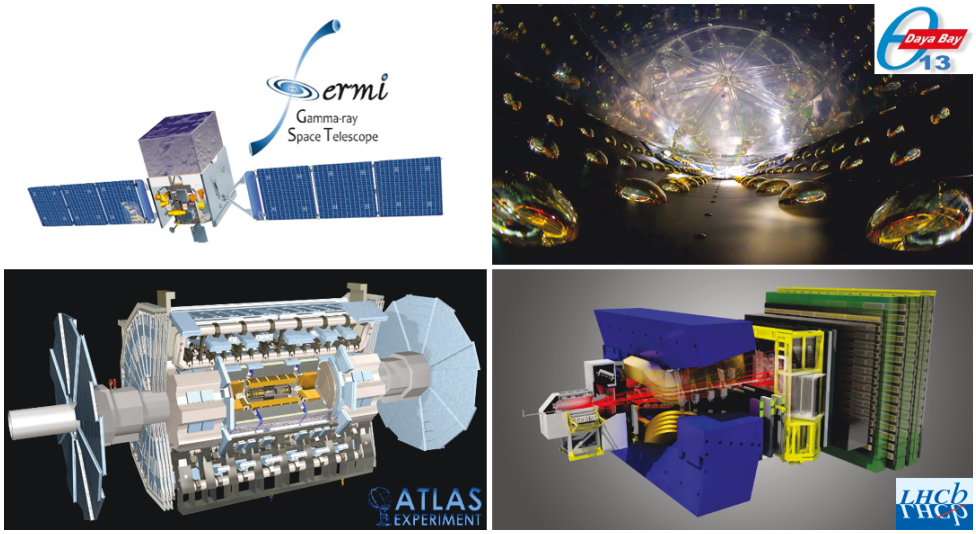

Figure 1: Some of the HEP experiments, using the GAUDI framework for event data processing.

In the late 90's, GAUDI was designed as a sequential framework. However, in order to address the challenges, posed by the current and future HEP experiments, GAUDI must evolve through a paradigm shift towards a concurrent architecture.

In 2010, the first GAUDI parallelization project was started at CERN by Mato and Smith [5], applying the *multiprocessing* (MP) paradigm to the

framework. The approach has been shown to be efficient if used on a *multi-core* central processing unit (CPU). However, the extensive levels of modern hardware parallelism, as well as the heterogeneity of computing units in general, make the application of this paradigm to Gaudi limited. This manifests itself in weak throughput scaling on the *many-core* systems, where the random-access memory (RAM) and the disk input/output (I/O) constraints become significant. Furthermore, the paradigm has been known to be not latency tolerant, thus lacking support for efficient handling of I/O-bound operations and computations offloading.

To leverage the full extent of hardware parallelism suggested by the industry, the Concurrent Framework Project [6] has been launched at CERN. The context of the project encompassed evaluation of the *multithreading* paradigm – the second major paradigm for high-throughput and many-task computing. It was believed to be capable of solving some, or all, of the problems of the multiprocessing paradigm.

A minimal Gaudi Hive prototype was developed in 2012 [7], demonstrating the basic principles of concurrent Gaudi of a new generation.

In the thesis, I consider a complex of non-intrusive task scheduling solutions for throughput maximization. After a short introduction to the architecture of the Gaudi framework in chapter 1, I present the new concurrency control system in chapter 2 where I compare its graph-based decision making to the previously adopted catalog-based one. I then devote chapter 3 to demonstrate the potential of predictive task scheduling approach. In chapter 4, I introduce the latency oblivious task scheduling for Gaudi Hive and prove a significant throughput maximization potential of this scheduling technique.

Chapter 1

The Gaudi Framework: overview

Definition 1.1

In the context of HEP, an event is a high energy interaction of elementary particles and the interaction aftermath, occurring in a well-localized region of space and time.

1.1 Sequential Gaudi

1.1.1 Architecture

GAUDI [8, 2, 9] is an object-oriented software framework, implemented in C++ and PYTHON, that provides a common infrastructure and environment for building HEP event data processing applications. Such applications cover the full range of HEP computing tasks, for instance: event and detector simulation, event triggering, reconstruction and analysis, detector alignment and calibration, event display, etc.

GAUDI is designed on the principles of composability, providing a way to construct applications of any complexity by combining general-purpose and specialized components. The main components of the GAUDI software architecture can be seen in the object diagram shown in figure 1.1.

All applications, based on GAUDI, are written in the form of special-

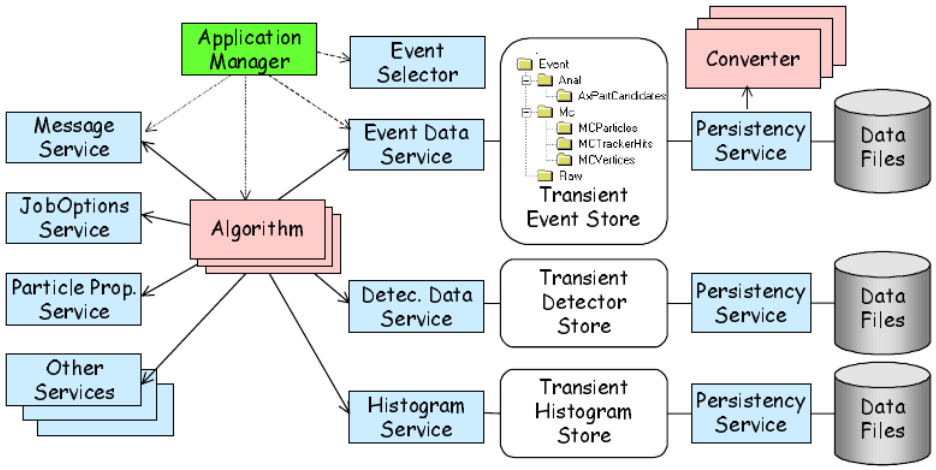

Figure 1.1: GAUDI framework object diagram [10]. It represents a hypothetical snapshot of the state of the system, showing various components of the framework, as well as their relationships in terms of ownership and usage.

izations of standard framework components, complying to the well-defined abstract interfaces. Strict interfacing rules of the GAUDI components were a key to a highly flexible framework and foreordained its success.

1.1.2 Data processing model

An important underlying principle of GAUDI is the separation of the concepts of data and procedures used for its processing [10]. Broadly speaking, event reconstruction and physics analysis consist of manipulation of mathematical or physical quantities (such as points, vectors, matrices, hits, momenta, etc.) by algorithms which are generally implemented in terms of equations and natural language. In the context of the framework, such procedures are realized as GAUDI ALGORITHMS (see figure 1.1).

The GAUDI architecture foresees either explicit unconditional invocation of ALGORITHMS by the framework, or by other ALGORITHMS. This latter possibility is very important as it allows to address a common use case of a physics application to execute different ALGORITHMS depending on the physics signature of each event, which might be determined at run time as a result of some reconstruction. This capability is supported in GAUDI through

sequences, branches and filters. A sequence is a list of ALGORITHMS. Each ALGORITHM may make a filter decision, based on some characteristics of the event, which can either allow or bypass processing of the downstream ALGORITHMS in the sequence. The filter decision may also cause a branch whereby a different downstream sequence of ALGORITHMS will be executed for events that pass the filter decision relative to those that fail it [10].

Historically, GAUDI was designed as a sequential framework. It means that it is only able to processes events sequentially. Furthermore, it means that the sequence hierarchies, as well as all ALGORITHMS within a sequence, are also executed sequentially for each event (see figure 1.2).

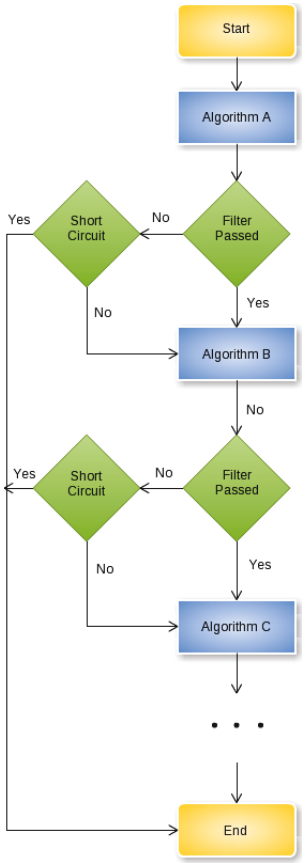

Figure 1.2: An example of conjunctive control flow inside a GAUDI sequence.

Lastly, all GAUDI components including ALGORITHMS were designed and implemented for single-threaded execution only.

1.2 Concurrent Gaudi (a.k.a. Gaudi Hive)

To leverage the full extent of hardware parallelism suggested by the industry, a minimal GAUDI HIVE prototype was developed [7]. The prototype was called to demonstrate, in the context of GAUDI, the basic principles of the *multithreading* paradigm – the second major paradigm for high-throughput and many-task computing. It was believed to be capable of solving some, or all, of the problems of the *multiprocessing* paradigm.

1.2.1 Architecture

An important novelty the GAUDI HIVE prototype employed together with the multithreading paradigm is the model of *task-based* programming. In the context of GAUDI, a task can be defined as follows:

Definition 1.2

Task – a GAUDI ALGORITHM, *a part of it or any other unit of computation packaged according to logical, efficiency or other suitable considerations.*

The main advantage of formulating computations in terms of tasks, not threads, is that they favour a well-organized work partitioning. Other advantages include efficiency considerations [11]:

- faster task startup and shutdown,
- improved load balancing,
- matching parallelism to available resources.

The task-based approach also allows concentrating on dependencies between tasks, leaving load-balancing issues to a back-end scheduler. Within the GAUDI HIVE project, the Intel® Threading Building Blocks (TBB) library was chosen as such back-end.

Support for multidimensional concurrent data processing in GAUDI required substantial revision of the framework. However, the principle of composability GAUDI was designed on made it possible to avoid any fundamental architectural change in the framework. Instead, the components necessary for the first minimalistic prototype of the multithreaded GAUDI were developed and used to augment the framework via its standard interfaces in a pluggable fashion [7].

The architectural peculiarities of the GAUDI HIVE prototype are outlined in figure 1.3. The event loop manager feeds events to the scheduler, which requests ALGORITHMS' instances from the ALGORITHM POOL, wraps them into tasks and submits them further to the TBB runtime. The GAUDI WHITEBOARD is the component responsible to hold the event data stores for all the events being treated concurrently [12].

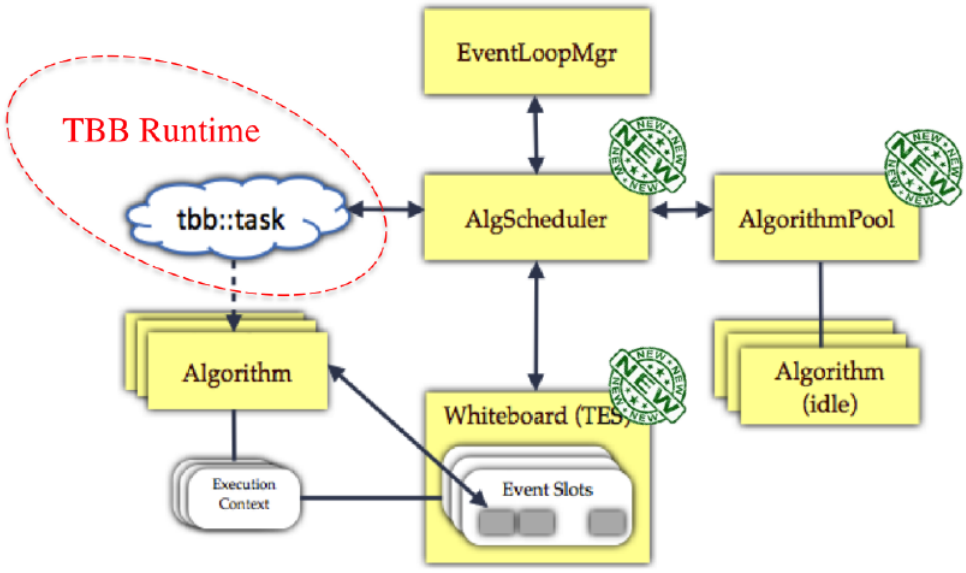

Figure 1.3: A diagram of architectural changes brought by the GAUDI HIVE prototype. The components marked as *new* were added to support the inter- and intra-event levels of concurrency [12].

1.2.2 Data processing model

The multithreaded task-based approach enables three levels of concurrency in data processing:

- inter-event (concurrent processing of multiple events);
- intra-event (concurrent processing of tasks on an event);
- intra-task (concurrency, internal to a task).

The second concurrency level required an introduction of a finite-state automaton (see figure 1.4) for the GAUDI ALGORITHM.

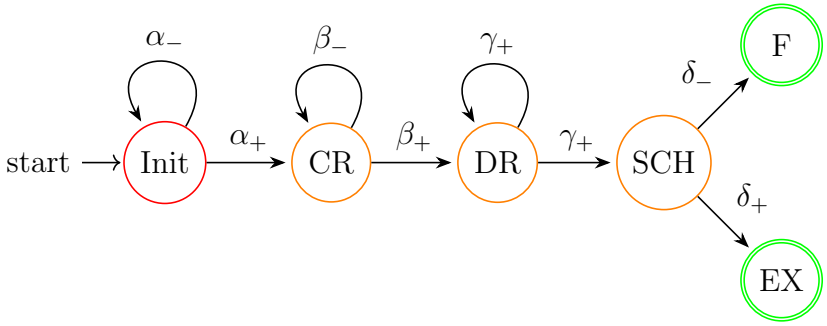

Figure 1.4: Decision-driven finite-state automaton for a GAUDI HIVE ALGORITHM. The evolution of the ALGORITHM state is handled by either positive or negative decisions. The decisions are produced by various components of the framework.

In particular, the following states need to be tracked within the GAUDI HIVE framework:

- Initial (I)
- Control Ready (CR) – required by concurrency control;
- Data Ready (DR) – required by concurrency control;
- Scheduled (SCH) – required to track the fact of scheduling;
- Failed (F) – occurs if an ALGORITHM failed to process an event;

- Executed (EX) – occurs upon successfull completion of event processing.

The states required for concurrency control are discussed in more detail in section 2.1.

Chapter 2

Gaudi Hive Concurrency Control System

2.1 Problem formulation

The GAUDI framework is responsible for event data processing by means of task ensembles, which either transform the event data, produce filter decisions upon its processing, or do both. The tasks cannot be executed in arbitrary order since they always exhibit dynamic precedence constraints, arising from user-specified precedence rules of two categories:

- Control flow (CF) rules,
- Data flow (DF) rules.

The CF rules describe which tasks in a given ensemble must be executed for each event. This is controlled by filter decisions, produced by preceding tasks. The DF rules describe which data entities each task was designed to process, and if these data entities are mandatory or optional for execution of a task.

Let me define a task schedule as follows:

Definition 2.1

Task schedule - a concrete task precedence pattern.

Building a valid task schedule constitutes a serialization problem, and requires resolution of precedence rules for each task in a given ensemble. There is no single schedule pattern, though, which is valid across all types of events, since filter decisions depend on event type. Moreover, the event type, in general, is not known prior to its processing. Thus, the task schedules can only be built at run time.

The case of sequential data processing significantly simplifies the run time building of task schedules. This is a consequence, on one hand, of linear resolution of CF rules at run time, and, on the other, of implicit linear resolution of DF rules, accomplished at configuration time by tasks ordering.

For the case of concurrent data processing, such simplifications do not hold any more. The complexity of the serialization problem drastically increases due to loss of linearity in precedence rules resolution (see Table 2.1).

Table 2.1: Resolution conditions of task precedence rules

	Sequential Gaudi		**Concurrent Gaudi**	
Rules	CF	DF	CF	DF
Resolution	@run time	@configuration time	@run time	@run time
Complexity	Low	None	High	High
Mechanism	Trivial	None	Required	Required

The loss of linearity follows from the fact that subsets of concurrently executing tasks complete, in general, in non-predictable order, or, possibly, even worse – at the same time. This leads, on one hand, to increased complexity of CF rules resolution, and, on the other, to displacement of DF rules resolution from configuration time to run time. Sequential Gaudi does not have any means to perform decision making of such type, because within the sequential paradigm it is unnecessary. Not even has it a knowledge base, such decisions can be made over.

Thus, concurrent Gaudi required a new component, which would solve the problem of concurrency control.

2.2 Requirements

The first and the most significant requirement for any concurrency control system is, by definition, to guarantee correctness of ordering of concurrent operations.

Secondly, the system has to ensure decisions on concurrency control are made as quickly as possible. This constitutes a soft real-time requirement [13] for the system, since the usefulness of decisions on concurrency control degrades with time. In section 2.5.2 I will show how delaying decisions can affect framework's performance.

Furthermore, multiple valid decisions on concurrency control are possible in a concurrent environment. Thus, a third requirement of choosing, according to given criteria, the best decisions might be optionally set.

Finally, all previous requirements have to be fulfilled without sacrificing scalability of the concurrency control system.

2.3 Catalog-based decision making

In this section I will describe the approach, which was used for concurrency control in the first prototype of Gaudi Hive.

2.3.1 Metadata organisation

The catalog-based approach relies on disjoint representations of CF and DF rules, and thus - disjoint resolution of corresponding precedence rules.

The DF rules are expressed in the form of a catalog of tasks with a set of corresponding data entities each task requires as input. At any moment of time the framework contains a dynamic set of available data entities, tracked in a separate catalog.

The CF rules, are represented as a hierarchical rooted tree. There are two types of nodes within the hierarchy: tasks and decision hubs.

Every task node represents a Gaudi task. It has at least one parent node of decision hub type, and can not be itself a parent to any other node.

The decision hub node represents a GAUDI Sequence. It can be parental to both task and decision hub nodes.

Each decision hub node implements the CF logic, according to which its direct child nodes are coordinated. It also serves as an aggregation point for filter decisions, flowing up the tree from all subordinate nodes. The hierarchy is said to be completely resolved when the most superior, i.e. root, decision hub forms its filter decision.

An example of such tree is shown in figure 2.1 for the case of typical LHCb event reconstruction workflow.

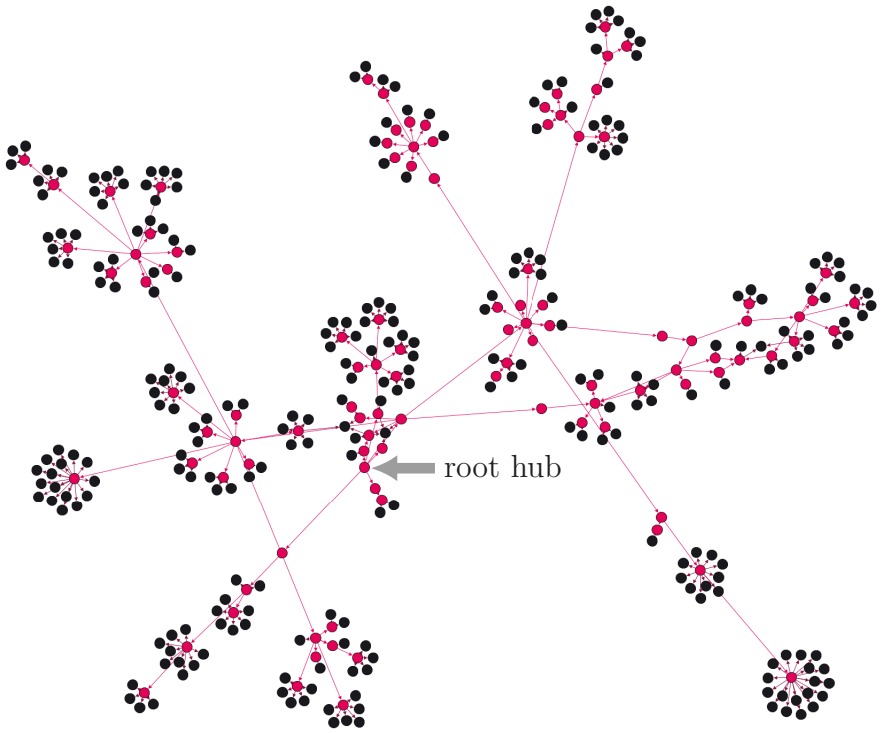

Figure 2.1: A rooted tree, representing the control flow rules in a typical workflow of physics data reconstruction in the LHCb experiment. Black nodes represent tasks (280 nodes), while red ones - decision hubs (110 nodes).

Despite the fact the CF rules were represented as a graph, its inefficient processing (see the next subsection) used little of its natural properties.

2.3.2 Processing of metadata

A schematic design of the catalog-based system is shown in Figure 2.2.

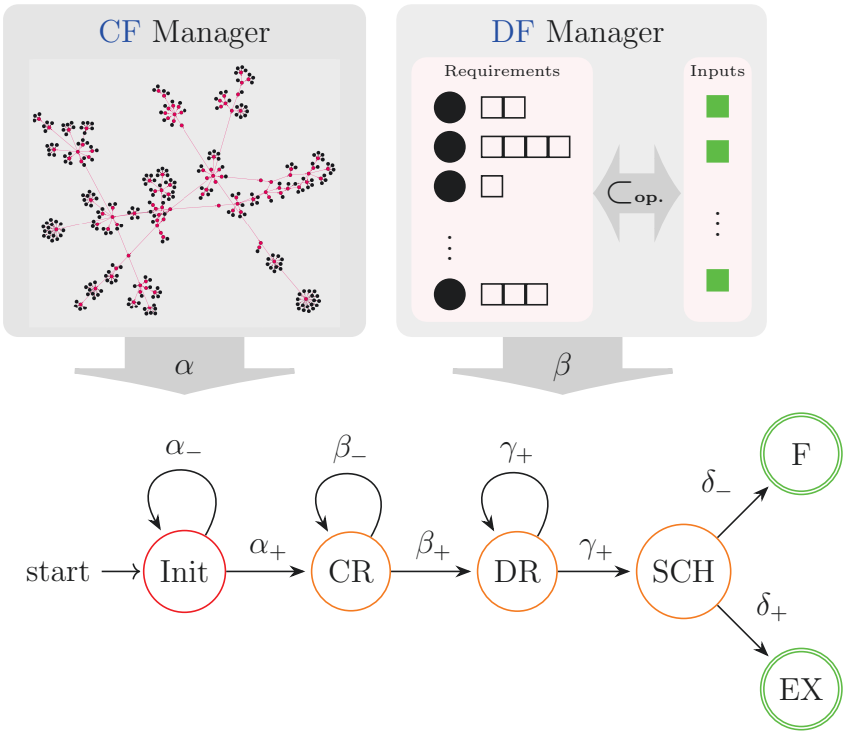

Figure 2.2: A schematic design of the catalog-based concurrency control along with the decision-driven finite-state machine (FSM) for a GAUDI ALGORITHM being executed in a task. In each decision making cycle, the evolution of the ALGORITHM state is handled by either positive, or negative, concurrency control decisions. The decisions are produced independently by the CF and DF managers.

It is a bipartite system, consisting of isolated components for resolution of CF and DF rules. Each component is designed to yield a single decision in one decision making cycle, governing a corresponding FSM transition for a task. Thus, a minimum of *two* decision making cycles are needed to perform a $I \to CR \to DR$ transition for a task in its FSM.

The DF component is responsible for resolution of DF rules by means of matching data input requirements of a task with data entities, declared as currently available in the GAUDI WHITEBOARD. This is achieved by means of regular index searching and matching operations.

The CF component resolves the relevant rules by performing global traversals of the tree, accomplished in a top-down manner – starting at the root CF hub down to all task nodes of the tree.

The use of the top-down type of tree query in the CF component has an important consequence. It allows two modes of decision making – *aggressive* and *conditional*. The former implies enqueuing a decision making cycle every time a task is executed. The latter utilizes a more optimized process by cancelling a decision making cycle, in case the decision making queue contains another cycle, launched by a different task earlier. The conditional mode of operation is possible due to the fact that CF traversal is global in this approach, and always revises the entire state of filter decisions in the system.

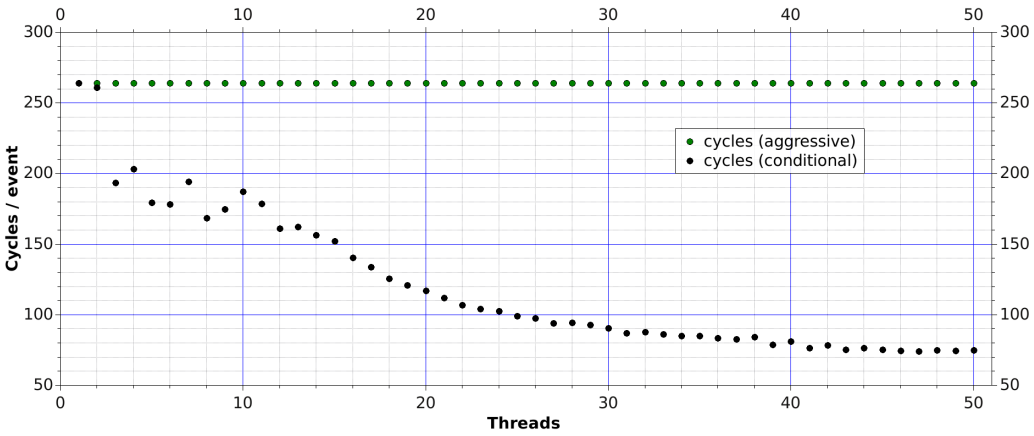

Figure 2.3: Count of decision making cycles per event. Configuration: LHCb Reconstruction @ Machine-2S48T.

Conditional mode of decision making demonstrates an interesting and useful characteristic, when considered in the context of scalability. The number of decision making cycles, completed per event, has been measured versus the number of threads, used by the framework.

Figure 2.3 reveals the following observation:

Observation 2.1 (*Dynamics of DM cycles*)

The number of decision making cycles, completed per event in the catalog-based approach, quickly decreases as the number of threads, used by the framework, grows.

This downtrend effect is caused by accompanying uptrend behaviour of the task completion rate as the number of threads increases.

2.4 Graph-based decision making

The concept of a graph has been around since the late 19th century, however, only in recent decades has there been a strong resurgence in the development of both graph theories and applications. In contrast to the index-intensive, set-theoretic operations of the catalog-based approach, the graph-based approach make use of index-free traversals.

2.4.1 Metadata organisation

Deliverable 2.1 (*Graph-based DM*)

An alternative concurrency control system, based on graph-based decision making, was suggested for Gaudi Hive *[14].*

The approach is based on two main pillars. First, a unified knowledge space for both CF and DF precedence rules. Second, a representation of the knowledge space in the form of a graph.

At first glance, explicit representation of DF rules as a graph does not require any auxiliary concepts, and can be done by establishing direct relationships between task nodes. From such simplistic viewpoint, the graph of data dependencies for LHCb event reconstruction would take the form, shown in figure 2.4).

However, in addition to basic data flow operations, tasks in sequential Gaudi can:

- generate output data items *conditionally*;

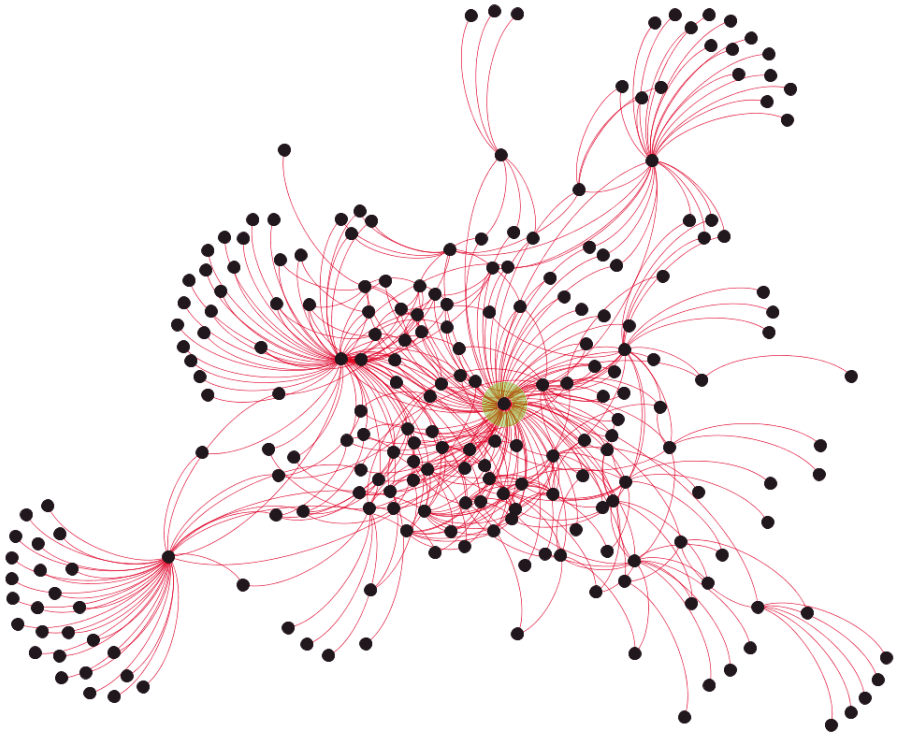

Figure 2.4: Graph of data flow between tasks in a typical workflow of physics data reconstruction in the LHCb experiment. Black nodes represent tasks, while curved edges – the data flow. The curve indicates direction of data flow: read an edge clockwise from a source node to a target node. The task node, highlighted in green, is virtual and represents the framework, which loads data from disk for subsequent processing.

- require *alternative* input data items.

To support this functionality in concurrent Gaudi greater flexibility is required in representation of DF rules. To achieve this, I augmented the representation with additional type of node to express the notion of a data item. Each such node has to have relationships with all possible actors it can originate from, as well as with the actors it may be consumed by. As a result, the data flow realm, shown in figure 2.4, extends to realm, presented in figure 2.5.

As a side note, the idea of explicit embedding data items into the concurrency control knowledge space can potentially have wider applications.

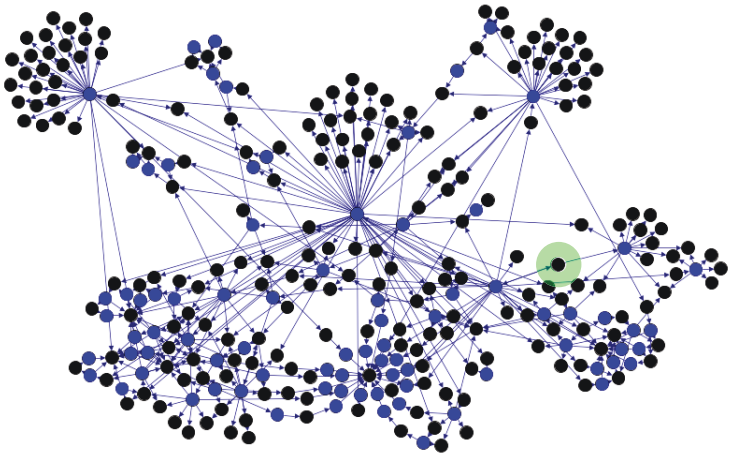

Figure 2.5: Augmented graph of data flow. In addition to figure 2.4, contains nodes, representing data entities. Each data entity node has a producer, and at least one consumer. Black nodes represent tasks, while blue ones denote data items. The task node, highlighted in green, is virtual and represents the framework, which loads data from disk for subsequent processing.

Such network of data items, enriched with knowledge of its flow patterns and placed in immediate vicinity to consumers, can serve for reduction of data access time as:

Deliverable 2.2 (*Distributed Transient Event Store*)

A short circuit, distributed variant of the Gaudi *Transient Event Store.*

As both CF and DF realms have common entities – tasks – it becomes natural to unify them in a common knowledge space. Such unification of CF and DF precedence rules is demonstrated in figure 2.6 for the case of LHCb event reconstruction workflow.

Such approach has several important properties, among which:

Property 2.1. Self-contained knowledge space;

Property 2.2. Perfect information partitioning;

Property 2.3. Global scope of precedence constraints.

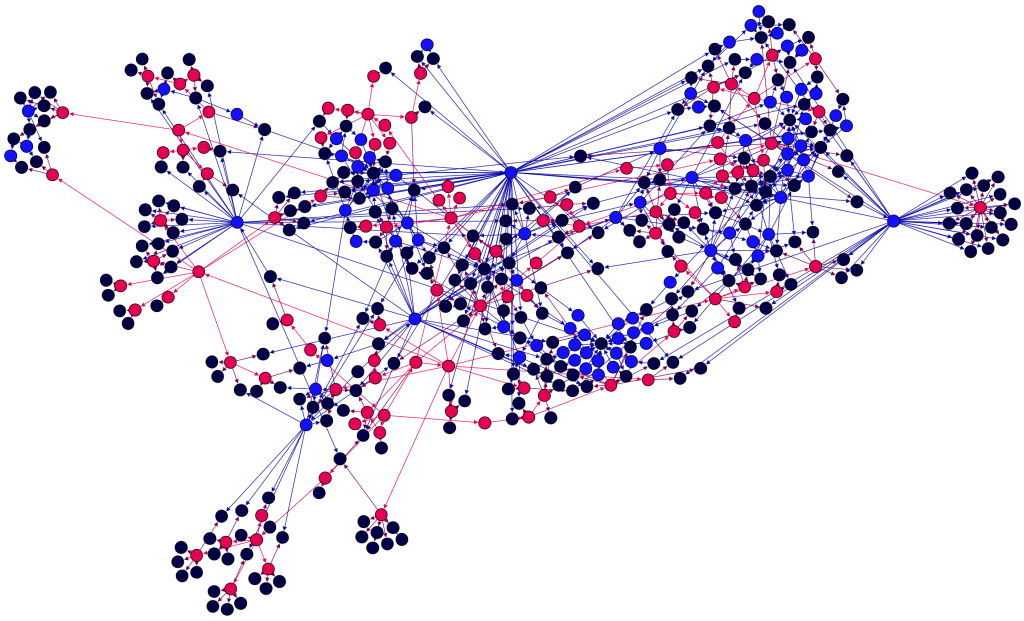

Figure 2.6: Graph of control and data flow rules between tasks in a typical data reconstruction of the LHCb experiment. Black nodes represent tasks (263 nodes), while blue ones (85 nodes) denote data items, produced or consumed by the tasks. The red nodes (105 nodes) represent the CF decision hubs.

These properties lead to several important implications on a concurrency control mechanism. For instance, property 2.1 has a consequence in architecture design. In particular, it allows to have:

Corollary 2.1. A single component for resolution of both CF and DF rules.

Property 2.2 provides ideal information partitioning, which means that to reason about a given task one needs to process only those entities that are directly related to it, and nothing else. Non-relevant information is naturally isolated. This leads to the following corollary:

Corollary 2.2. Good algorithmic complexity of the decision making system.

The latter is discussed in detail in section 2.5.3. Likewise, corollaries 2.1 and 2.2 result in better response time:

Corollary 2.3. Low latency response from the decision making system.

The graph-based decision making has even wider implications beyond the problem space of concurrency control. In particular, property 2.3 provides a convenient medium for:

Corollary 2.4. Topological analysis of precedence rules.

This topological analysis has at least two areas of application. First, it unleashes efficient implementation of configuration-time mechanisms for verification of workflow correctness, discussed in section 2.5.5. Second, being more sophisticated, it enables evaluation of potential intra-event concurrency dynamics and its degrees of freedom. This knowledge can then be used as foundation for various techniques of throughput maximization, not available before to the framework. Such techniques are investigated in Chapter 3.

2.4.2 Processing of metadata

Deliverable 2.3 (*Graph-based concurrency control*)

An alternative concurrency control system, operating on the principles of graph-based decision making, was developed [15] for Gaudi Hive.

A schematic design of the graph-based system is shown in figure 2.7.

As implied by property 2.1 and corollary 2.1, the new approach provides a single component concurrency control system (CCS).

Meta data processing has changed in two essential aspects.

Firstly, as the entire space of precedence rules in this approach is represented as a graph, a new decision making cycle comprises a pure graph traversal, including the DF realm.

Secondly, I changed the graph traversing algorithm in the part of CF processing:

Deliverable 2.4 (*Vicinity graph traversing strategy*)

An alternative, bottom-up CF graph traversing strategy was suggested and implemented.

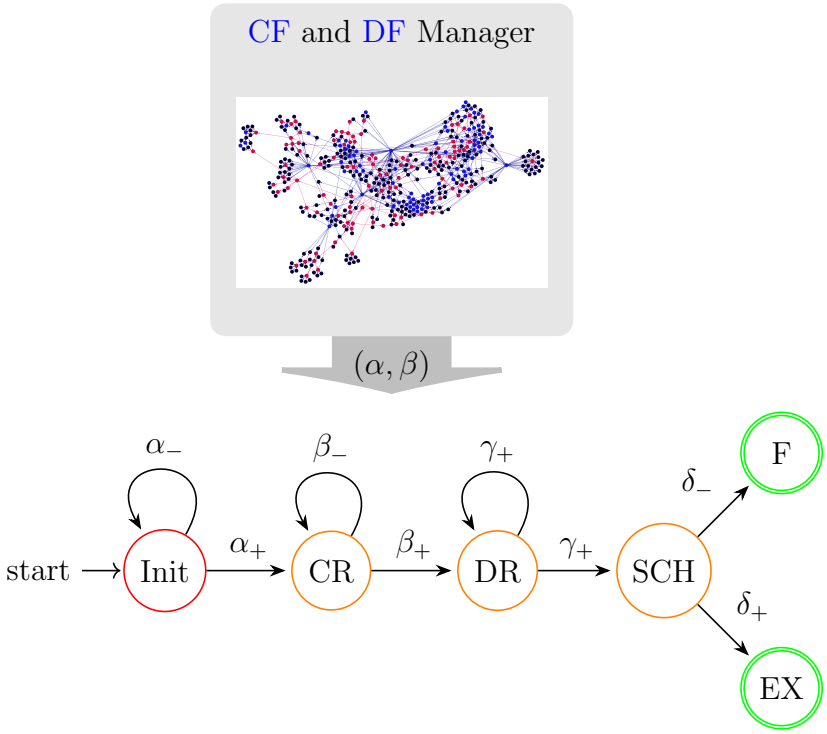

Figure 2.7: A schematic design of the graph-based concurrency control along with the decision-driven FSM for a Gaudi Algorithm being executed in a task. The FSM mechanism remains the same as in the catalog-based approach. Contrary to the catalog-based decision making, the graph-based one yields a pair of concurrency control decisions in a single, intrinsically efficient traversal of the graph of unified CF and DF rules.

It allows to reduce the scope of a graph traversal. In contrast to making graph-wide traversals, the new strategy reduces the traversal scope to only a neighborhood of a task. The DF part of decision making features the same – localized – graph traversing behavior. The aforementioned change of traversing style means that a decision making cycle must be triggered every time a task is executed.

Unlike the catalog-based CCS, the graph-based system is designed to yield a sequence of two decisions for both $I \to CR$ and $CR \to DR$ FSM transitions upon a single query. This became possible because a single graph traversal covers both CF and DF realms of a task.

Another peculiarity is that the graph-based CCS can only operate in

aggressive mode of decision making (see section 2.3.2). This characteristic might potentially lead to additional pressure in terms of decision making time and its scalability. However, as I will show in section 2.5.4, this effect is not significant due to the intrinsic efficiency of the graph-based approach.

2.5 Comparison of graph-based and catalog-based decision making

2.5.1 Primary implications on global performance

The deepest difference between the catalog-based and graph-based CCS is in the gamut of objectives the systems are able to pursue.

The base, and the only, objective the catalog-based system is designed to successfully attain is searching for all tasks that are permitted to execution in a given context of precedence rules. Following this objective guarantees the fact of completion of event processing. Note that no other guarantees are provided. This represents *reactive* concurrency control.

The graph-based system, though, is capable of *proactive* behavior. The latter involves acting in advance of a future situation, rather than just reacting. In addition to the objective of reactive concurrency control, the system is able to pursue the objective of minimizing the event processing time by taking control of decision consequences and by adjusting its decisions to profit from these consequences in the best possible way. Implementing such behavioral features requires deeper scope for analysis of precedence consequences. This is provided by the distinctive properties of the graph-based approach, described in section 2.4.

Chapter 3 covers in significant detail the framework performance limitations, that originate from reactive concurrency control. This is then followed by detailed investigation of several concrete proactive scheduling techniques.

2.5.2 Secondary implications on global performance

In concurrent environment, the problem of concurrency control is one of those that creates a new item in the account of overhead. In GAUDI, as explained in section 2.1, this overhead is exhibited at run time.

In order for the Gustafson-Barsis' [16] observation to hold, it is critical to ensure that all forms of overhead, including the one of concurrency control, grow as slow as it is possible with respect to the growth of parallel work, supplied to the framework. In other terms, the overhead has to be scalable. If no attention is paid to this strategically important aspect, the framework-wide performance will most definitely loose its scalability as well.

There is also another, more subtle, interplay between concurrency control and framework as a whole. It occurs irrespective of whether or not the concurrency control system is scalable, and originates from the fact that the CCS response is never instantaneous. This latency reduces the average number of tasks being concurrently executed by the framework, which results, according to Little's formula [17], in degradation of task and, hence, event throughput of the framework.

In sections 2.5.3 and 2.5.4, I will scrutinize various dimensions of the overhead, accompanying each type of decision making.

2.5.3 Algorithmic complexity of decision making

In assessment of efficiency of various approaches to the problem of concurrency control, it is of uttermost importance to quantify the corresponding amount of resources required. An important, machine model independent aspect of efficiency is the algorithmic time complexity of a decision making cycle.

Measurements 2.1 (*DM algorithmic complexity*)

> *The per-task algorithmic complexities of the catalog-based and graph-based decision making were estimated.*

In the context of concurrency control in GAUDI, the time complexity is a function of the number of elementary operations, performed on the entities

of concurrency control – tasks and decision hubs. I denote their respective numbers, engaged in a given ensemble, as n_a and n_d.

As one can see from table 2.2, the differences in approaches, discussed in sections 2.3 and 2.4, lead to diverse characteristics.

Table 2.2: Comparison of time complexities of one decision making cycle in catalog-based and graph-based approaches to concurrency control

	Average		**Worst**	
Approach	**CF**	**DF**	**CF**	**DF**
Catalog-based	$O(n_a + n_d)$	$O(1)$	$O(n_a + n_d)$	$O(n_a)$
Graph-based	$O(1)$	$O(1)$	$O(n_a)$	$O(1)$

With the graph-based approach, the average complexity improves from being linear to constant. As already mentioned in corollary 2.2, this result builds on property 2.2, and also on the localized, bottom-up, form of the graph traversal. Therefore, as the graph grows in size, the cost of traversal of a node neighborhood remains the same.

Even though only amortized time complexity is of practical importance for us, it is worth mentioning that the worst complexity also improves when the graph-based technique is used. It follows from the fact that a localized graph traversal engages only limited number of decision hubs, located in close proximity to a task node.

2.5.4 Decision making time and scalability

Another, more influential, dimension in assessment of computational characteristics of a CCS is evaluation of its response time. In the context of concurrent computing, as pointed out in section 2.5.2, this aspect can affect framework-wide performance.

Measurements 2.2 (*DM time and scalability*)

A series of benchmarks were carried out to evaluate the per-task and per-event decision making time and its scalability, maintained by the catalog-based and graph-based CCS.

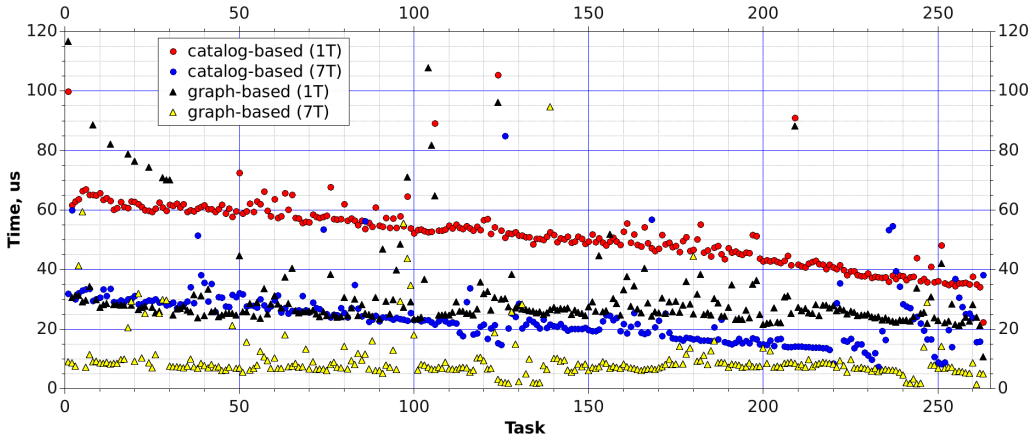

Figure 2.8: CCS response time, spent for each task in an ensemble, as a function of natural execution order of tasks. The response time for each task is averaged over 100 events. Two pair of curves are presented, each describing the impact of a chosen approach to concurrency control for 1- and 7-threaded operation of the GAUDI framework. Configuration: LHCb Reconstruction @ Machine-1S8T.

Figure 2.8 demonstrates the evolution of per-task response time within an event, examined for the cases of the two approaches described in the previous sections.

The first thing to note is the pair of curves, measured for the case of sequential execution, i.e., for one thread only. From the figure it becomes evident that the graph-based technique of concurrency control has a clear supremacy. It is also worth mentioning that the graph-based approach yields stable response time at all stages of event processing, whereas the catalog-based approach improves its response time all way through to the end of event processing. This can be explained by the fact that with time, as tasks yield filter decisions, the CF graph becomes more and more resolved, thus shortening the top-down graph traversal with every next decision making cycle. By the end of event processing the improvement in response time brings about 35%, staying, nevertheless, around 1.4 times slower than the corresponding graph-based decision making.

Furthermore, it is very important whether the response time is scalable across the number of threads granted to the concurrent framework. However, unlike the scalability across the number of entities engaged in the problem

of concurrency control, the scalability across the number of threads can be adequately estimated only empirically. Figure 2.8 provides the response time profiles for the case of seven threads. As one can see, the response time is systematically lower with both approaches to concurrency control, if more threads are used.

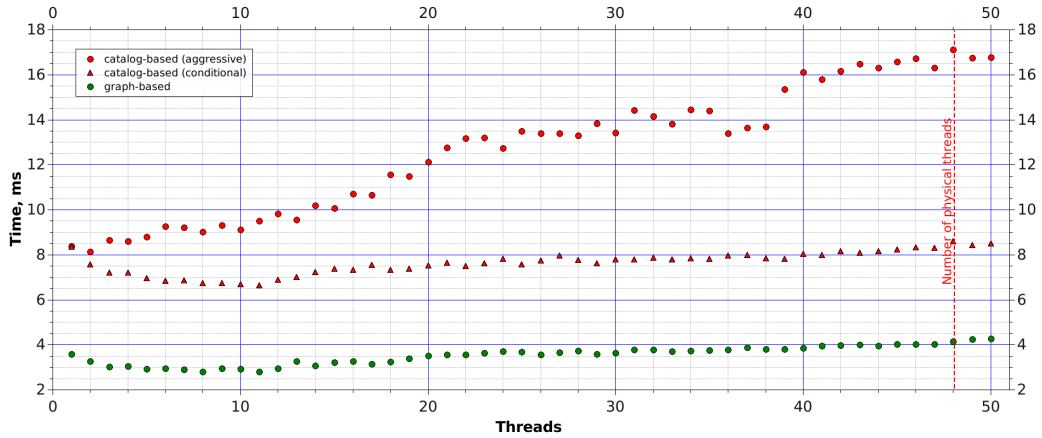

Figure 2.9: Cumulative decision making time, spent per processed event, as a function of number of CPU threads, used by the GAUDI framework. Graph-based and catalog-based implementations of CCS are compared. Configuration: LHCb Reconstruction @ Machine-2S48T.

The latter observation generates further interest in a more detailed profile of scalability of decision making time across the number of threads. Considering now the cumulative time of decision making, spent on concurrency control in processing of one event, figure 2.9 reveals that the downtrend of response time is persistent in the range of up to eleven threads for both concurrency control techniques (except for the case of *aggressive* catalog-based mode, see section 2.3.2). In the region above that value, though, this tendency turns to uptrend. From figure 2.9 it also becomes apparent, that the catalog-based approach, that operates in aggressive mode, not only yields the highest response time, but also shows no scalability across the number of workers, granted to the framework.

Finally, it is useful to measure the ratio of total decision making time, spent per event, to event processing time. Figure 2.10 reformulates the results, presented in figure 2.9, to this perspective. In all three cases, the ratio

is growing rapidly in the low range of threads. This is caused by the fact that in this very region the event processing exhibits almost linear speedup.

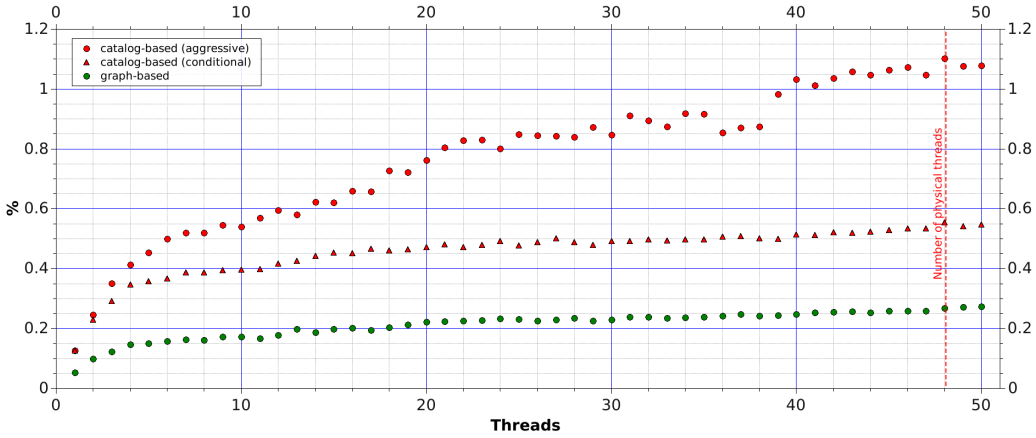

Figure 2.10: Ratio of total decision making time, spent on one event, to the event processing time. The upper limit of event processing speedup amounts to 4x with the chosen configuration. Configuration: LHCb Reconstruction @ Machine-2S48T.

2.5.5 Auxiliary considerations

The graph-based approach to concurrency control enables efficient implementation of several auxiliary mechanisms.

In sequential Gaudi, the resolution of the DF rules is accomplished implicitly, at configuration time, by ordering tasks. The absence of a dedicated mechanism for resolution of these rules was making their verification inconvenient. The graph-based CCS, though, makes such verification straightforward, as the system has the whole data flow realm assembled as a graph.

A similar search for inconsistencies becomes possible for the CF realm.

Another application area concerns the intra-event concurrency constraints analysis. A unified space of precedence rules makes it possible to determine the asymptotic speedup limits of a given data processing workflow.

Chapter 3

Scheduling of non-deterministic task networks in Gaudi Hive

In the pursuit of throughput, the independent nature of HEP events enables the possibility of saturating available hardware resources by using inter-event concurrency. This property of a problem domain is sometimes referred to as *pleasingly parallel*.

In this chapter I will show how a blind belief in the possibility mentioned above can lead to significant flaws in framework-wide performance. I will also show what alternatives are available to secure against those risks, and what performance effects do they bring in the context of non-deterministic task networks.

3.1 Limitations of reactive scheduling

This section covers my investigation of Gaudi performance limits, when its task scheduling is based on reactive concurrency control. I also point out and investigate the factors those limits originate from, providing thus a foundation for systematic techniques of workflow independent performance improvement.

3.1.1 Performance limits

As described in section 1.2, GAUDI HIVE profits from both the intra- and inter-event concurrency.

Measurements 3.1 (*Throughput scalability*)

> *A series of benchmarks were carried out to evaluate the scalability of throughput, maintained by* GAUDI HIVE *under reactive concurrency control in the intra- and inter-event operation modes [18].*

The curve in figure 3.1 reveals the extent of scalability of the intra-event speedup of GAUDI HIVE, that operates under *reactive* concurrency control. As one can see, despite the ample hardware resources, the framework is significantly limited in its ability to scale up in this mode. The speedup starts to degrade at around 7 threads and shows its developing downtrend up until 38 threads, where after a 5.5% saltus it finally saturates and hits a plateau at around 18.3.

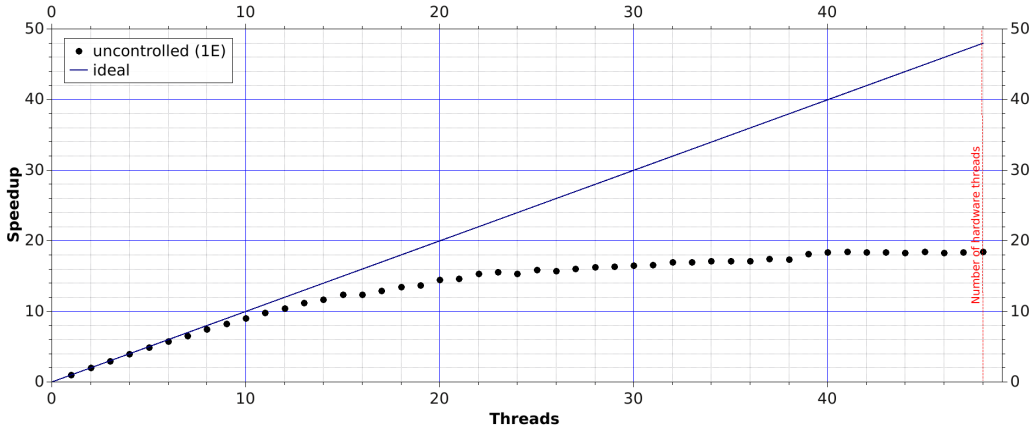

Figure 3.1: Speedup in processing of a single event under *reactive* concurrency control (hence, *uncontrolled* decisions) as a function of the number of threads granted to the framework. The intra-event speedup starts to degrade at around 7 threads, and saturates and hits a plateau above 18 at 40 threads. Configuration: LHCb Reconstruction[U–10ms] @ Machine-2S48T. 1k events processed.

The degradation of speedup scalability, shown in figure 3.1, is driven

by the intra-event concurrency constraints, imposed by the task precedence rules, used in the LHCb Reconstruction[U–10ms].

In order to achieve better scalability, Gaudi Hive can leverage on the inter-event dimension of concurrency. Figure 3.2 demonstrates the scalability of speedup in the inter-event mode, depending on the number of events being processed at the same time. It is notable, that as the number of events in flight grows, the utility of associated boost in speedup rapidly degrades. The observation can be rephrased in the following way:

Observation 3.1 (*Inter-event throughput scalability*)

In the LHCb Reconstruction[U–10ms], the framework is weakly scalable across the number of concurrently processed events.

Further increase in the number of events, processed concurrently, does not improve the throughput. Thus, for the taken configuration and concurrency control technique, the curve corresponding to 4 events describes a saturated mode of framework operation.

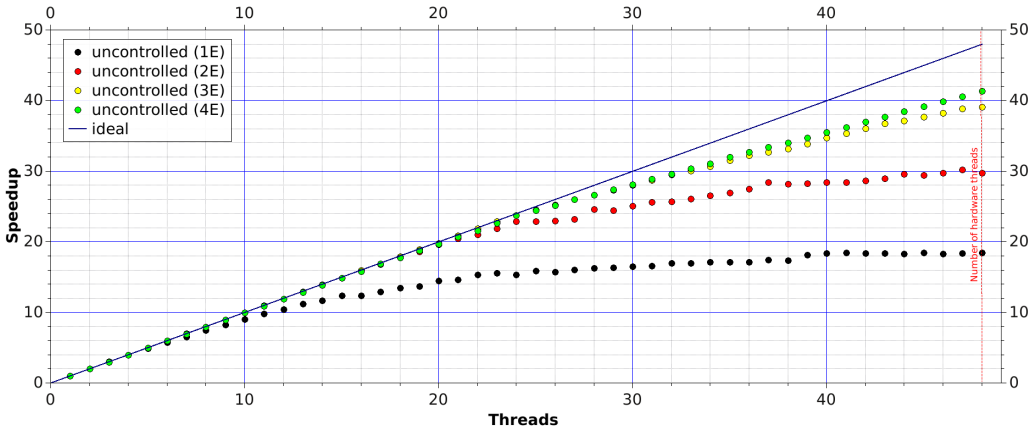

Figure 3.2: Speedup in processing of multiple events in flight under *reactive* concurrency control as a function of the number of threads granted to the framework. With all hardware threads being allotted to the framework, the speedup gets saturated with **4** events in flight. The saturated speedup starts to degrade at around 15 threads, still exhibiting linear growth up to the maximum hardware capacity of the machine. Configuration: LHCb Reconstruction[U–10ms] @ Machine-2S48T. 1k events processed.

Developing the previous observation, I suggest another set of benchmark results in figure 3.3. Its corresponding benchmark setup – the LHCb Reconstruction[N] @ Machine-2S48T – differs from the one used in figure 3.2 in only the utilized task time mapping. The set of presented curves indicates an even more rapid fall of speedup boost gained with every new event being added to the stack of concurrently proccessed ones. This strengthens observation 3.1 to the following observation:

Observation 3.2 (*Inter-event throughput scalability*)

Weak scalability across the number of concurrently processed events persists across varying mappings of task execution times in general, and across varying length of a critical path in a task network in particular, with other conditions being equal.

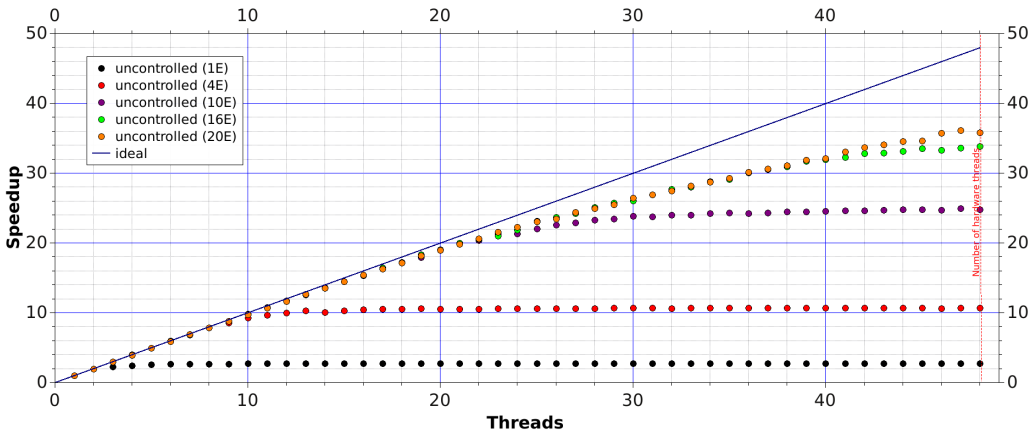

Figure 3.3: Speedup in processing of multiple events in flight as a function of the number of threads granted to the framework. With all hardware threads being allotted to the framework, the speedup gets saturated with **20** events in flight. The saturated speedup starts to degrade at around 25 threads, still exhibiting linear growth up to the maximum hardware capacity of the machine. Configuration: LHCb Reconstruction[N] @ Machine-2S48T. 1k events processed.

Figure 3.3 makes it also evident that the used non-uniform task time mapping (see C.3) yields significantly lower levels of available intra-event concurrency, thus greatly affecting the intra-event speedup. This provokes the

higher demand in the number of events, required to saturate the throughput. The net result is that, in the LHCb Reconstruction[N] @ Machine-2S48T, throughput saturates at only 20 events, giving it *many-event* nature.

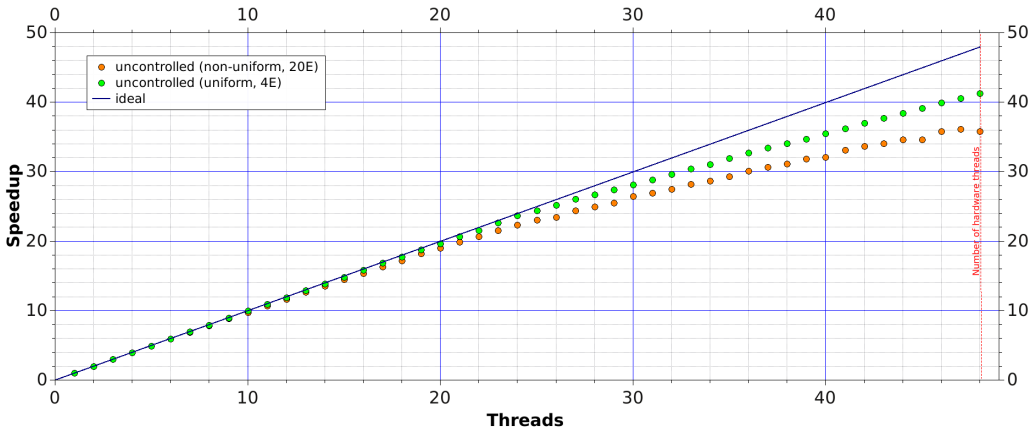

Figure 3.4: A comparison of saturated throughputs, achieved with distinct numbers of events, required for this saturation. The comparison is a function of the number of threads granted to the framework. Configuration: LHCb Reconstruction[N] @ Machine-2S48T is compared against the LHCb Reconstruction[U-10ms] @ Machine-2S48T. 1k events processed.

The many-event case allows to reveal another fundamental limitation, accentuated in figure 3.4. The latter provides an explicit comparison of the saturated regimes, showed separately in figures 3.2 and 3.3, and conventionally referred to as multi-event and many-event cases. One can observe a notable degradation of the many-event speedup: it drops by 14.2% at the maximum hardware capacity of the machine. The observation can be summarized in the following conclusion:

Observation 3.3 (*Throughput and inter-event concurrency*)

The peak achievable throughput degrades as the number of events required to saturate the throughput grows.

A possible reason for that is the increase of the overhead pressure, originating from the inter-event concurrency management.

One can anticipate at least two other harmful factors that increase their contribution as the number of events being processed concurrently increases.

First, the increase of the requirements on RAM and the increase of the load on the CPU data cache hierarchy. Second, the increase of disk I/O rates. However, realistic evaluation of associated performance degradation will only be possible when all GAUDI components are refactored for the use in the multithreaded environment.

The transition from multi- to many-event nature of concurrent data processing will intesify with the continuing growth of the hardware parallelism, suggested by the industry. Thus, systematic solutions are needed to address the limiting factors outlined in this section.

3.1.2 Intra-event concurrency dynamics

A standard approach to the problem of interplay between scalability and overhead of concurrency management is to mitigate the latter. Indeed, being an important aspect in any parallelization problem, this can facilitate some improvement in throughput scalability. However, in the context of HEP, as well as in any other field with the similar layout of concurrency dimensions, the problem of the overhead is rather a symptom, than a prime cause.

The root of the problem that obliges to use the *many-event* policy resides in low-level intra-event concurrency. This is corroborated by the results of measurements, presented in figure 3.3.

In this section I present my investigation of the factors that limit the intra-event concurrency.

Measurements 3.2 (*Reactive intra-event dynamics*)

The intra-event concurrency dynamics was measured under reactive concurrency control for the case of task precedence rules of the LHCb Reconstruction, used in sequential event reconstruction of LHCb during Run-1 and Run-2 phases of LHC.

Figure 3.5 exposes the task concurrency dynamics yielded by an event, being processed on 8 threads. It occurs that in this case the concurrency is disclosed quickly enough, inducing a regular load on available computing resources. The profile contains only a couple of discontinuities at the middle

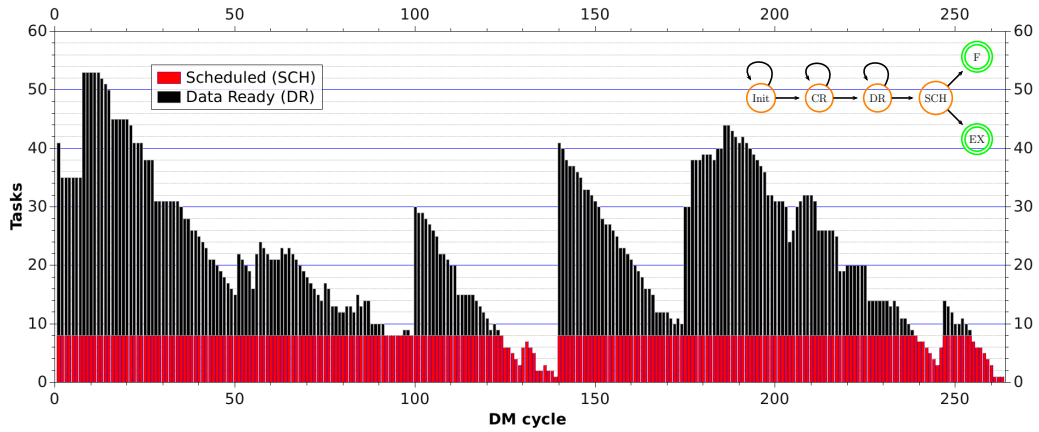

Figure 3.5: Intra-event concurrency dynamics as a function of *reactive* decision making cycle in natural execution order. **8** threads are granted to the framework. The DR task queue length varies from 0 to 31, with the average approaching 15. Configuration: LHCb Reconstruction @ Machine-2S48T. Single event profile.

and at the end of the processing range. Such concurrency pattern yields a speedup of 7.45 per event, missing only 7% of the theoretical maximum, which is lost due to the discovered irregularities.

A similar profile, in figure 3.6, considers the dynamics achieved with 20 threads. In this case the dynamics of concurrency disclosure do not keep anymore the available computing resources busy. In this case the observed speedup amounts to 14.4, lacking more than 28% of theoretical speedup.

In the view of presented results, it becomes conclusive that:

Observation 3.4 (*FSM transition imbalance*)

The intra-event concurrency dynamics, being extracted in the LHCb Reconstruction under reactive concurrency control, exhibits a developing imbalance in the $DR \rightarrow SCH$ transition phase as the number of threads, granted to the framework, grows.

In the LHCb Reconstruction, as well as in any other similar data processing scenarios of a HEP experiment, the organisation of tasks and associated precedence rules are constrained by the physics requirements. In pursuit of better layout of the intra-event concurrency, a scenario reorganisation is

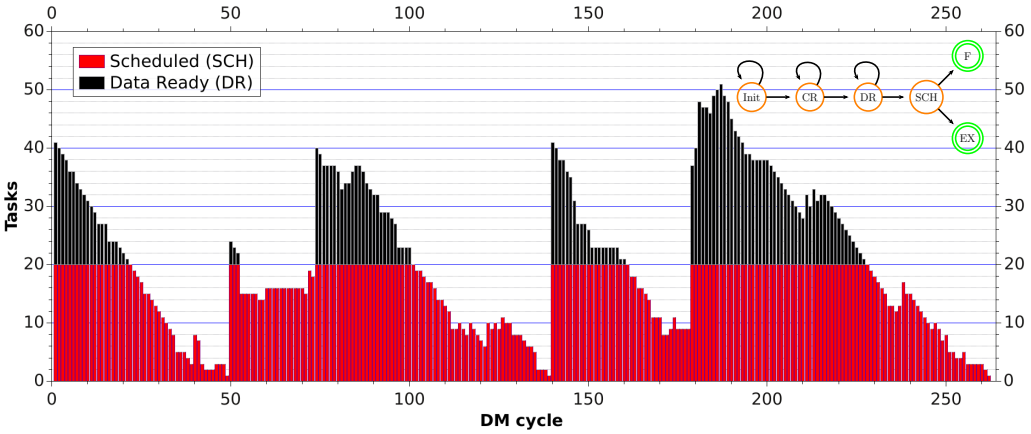

Figure 3.6: Intra-event concurrency dynamics as a function of *reactive* decision making cycle in natural execution order. **20** threads are granted to the framework. The DR task queue length varies from 0 to 31, with the average approaching 6. Configuration: LHCb Reconstruction @ Machine-2S48T. Single event profile.

sometimes feasible. In a settled experiment, however, the practical possibilities for that are rather limited.

Therefore, in the rest of the study I will always assume that the precedence rules can not be revised and will focus on examining other systematic concurrency leveling techniques that could be applied to and be useful in the context of GAUDI.

3.1.3 Degrees of freedom in concurrency control

The number of new tasks, qualified in a given decision making cycle as eligible for concurrent execution, depends on the concrete context the decision making is focused on. The context includes an executed task, the cycle was caused by, together with the precedence rules that task explicitly participates in.

The LHCb Reconstruction is characterized by significant heterogeneity of its precedence rules. This results in strong asymmetries across the number of tasks produced in each decision making cycle. That can be clearly seen in figures 3.5 and 3.6, where the number of tasks, released in a cycle, varies

from 0 to 40. When during a sufficiently long period of time no new tasks are emitted by the CCS, an intra-event task starvation period occurs.

It is not rare, however, that more tasks are emitted by a cycle than available computing resources can absorb. When this happens, the rejected part of the task emission is amortized in a queue for later execution, creating what I define as *situational* task excess. Nonetheless, the scenario under consideration shows that even for the moderate case of 8 threads, such task excess, in itself, does not always keep up in compensating the task shortcomings (see figures 3.5 and 3.6).

The situational task excess, however, can give new quality to task scheduling in Gaudi. When such excess occurs, it gives rise to vast degrees of freedom in controlling the future task excess rates. In particular, it becomes possible to induce more massive, *chain* task excess by initiating the avalanche task emissions, which can potentially be more efficient in leveling the intra-event task starvation periods. If successful, this will maximize the average intra-event task occupancy.

In the next section, I discuss various techniques for initiating the avalanche task emissions.

3.2 Predictive task scheduling for throughput maximization

In the context of concurrent Gaudi, an ability to maximize the intra-event occupancy allows to minimize the necessity in inter-event concurrency. As I demonstrated in section 3.1.1, this can help reducing pressure from concurrency management overhead, and result in higher data processing throughput in saturated regimes of data processing.

In order to provoke the avalanche task emissions, mentioned in section 3.1.3, two main mechanisms were required.

First, a task ranking mechanism was needed that could be able to automatically estimate the consequences the task can yield upon its execution. The first Gaudi Hive prototype [12] used the catalog-based CCS, capable of

only reactive decision making. The new – graph-based – CCS, as described in sections 2.4 and 2.5.1, enables proactive decision making and can be used for predictive scheduling. The graph-based CCS can estimate a task rank at either configuration, run time, or both.

Second, a mechanism for run time task reordering was needed to schedule the tasks with the highest rank first. The reordering is non-intrusive from the view point of task precedence rules and operates only within the queues of the DR tasks.

Deliverable 3.1 (*Predictive scheduling*)

A predictive scheduler, built on graph-based CCS and its proactive concurrency control, was developed.

Measurements 3.3 (*Predictive scheduling*)

A series of benchmarks were carried out to evaluate the scalability of throughput, maintained by Gaudi Hive *under proactive concurrency control in the intra- and inter-event operation modes.*

In the next sections I investigate several ranking strategies that can be applied in the context of Gaudi to arbitrary data processing scenarios and that can potentially influence the intra-event task emission dynamics.

3.2.1 Local task-to-task asymmetry

The simplest reasonable type of ranking involves *1-level task-to-task* relationships, established through their common data entities in the DF realm. The rank of a task is calculated as the total number of tasks that directly consume at least one of its data products. Since the precedence rules in general, and the DF ones in particular, are fixed across all types of events, the ranking of tasks can be completed at framework's configuration time.

Figure 3.7 demonstrates schematically one of the typical patterns that can occur within the DF realm of precedence rules. In this example, if both T_1 and T_3 happen to reside in the DR queue, it might be more profitable

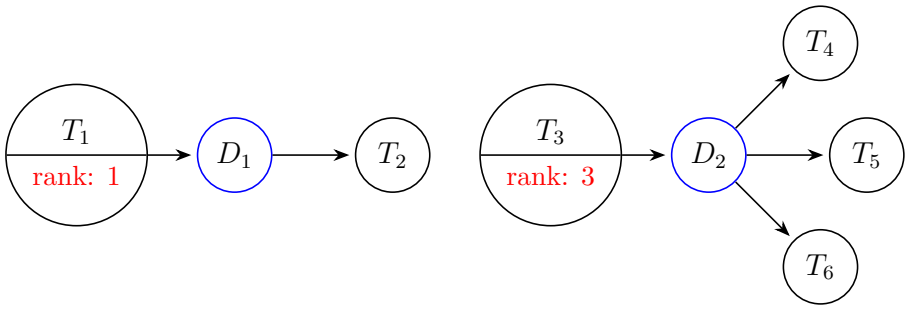

Figure 3.7: A schematic example of local task-to-task asymmetry in the DF precedence realm. Tasks T_1 and T_3 have distinct extent of consumption of the data products (D_1 and D_2) they produce. Thus, if both T_1 and T_3 reside in the DR queue, it might be more profitable to prioritize the execution of T_3.

to give T_3 a higher priority. The real asymmetries, exhibited by the data products from the viewpoint of their consumers can be seen in figure 3.8.

It should be stressed, however, that with such a limited decision scope it is not guaranteed that such prioritization is always profitable. This follows from the fact that any, or all, of T_{4-6} tasks might happen to be bound by other DF dependencies that will forbid their immediate scheduling for a period of time. Similarly, they might occur to be bound by the CF rules, preventing them from the $I \rightarrow CR$ state transition. The latter situation is clearly illustrated in figure 3.9.

In figure 3.10 I compare the data processing speedup, achieved with reactive and predicitive types of scheduling. One can observe that the 1-level task-to-task ranking technique allows to improve the speedup up to 30%, occurring at 19 threads.

It is notable that the area of ideal speedup has been tripled, when moving from reactive to predictive scheduling of 1-level type. An obvious consequenece is that, for this particular data processing scenario, there is no need to engage the inter-event concurrency at all for machines having less than 19 threads allotted to the framework.

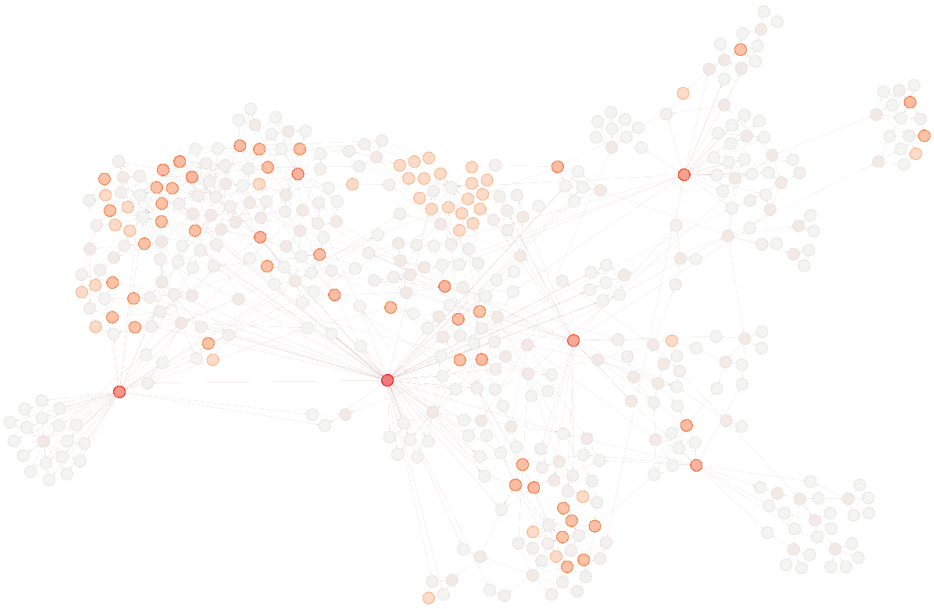

Figure 3.8: Task precedence rules of the LHCb Reconstruction (see figures 2.6 and C.1), with all, but data nodes, faded out. Color intensity of a data node represents the number of its direct task consumers.

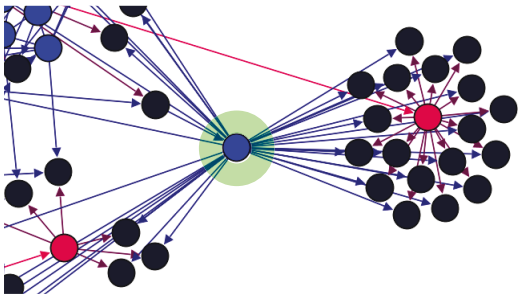

Figure 3.9: A topological feature, present in the LHCb Reconstruction (a zoom of an area from figure 2.6). One can see that for the CCS it is topologically profitable to prioritize production of the highlighted data item, because it enables the $CR \rightarrow DR$ transition for a whole cluster of tasks located on its right. On the other hand, it might well be that the decision hub that performs the CF handling of the cluster is still preventing all, or a subset, of its tasks from the $I \rightarrow CR$ transition, thus suppressing the rapid task emission the CCS aims to provoke.

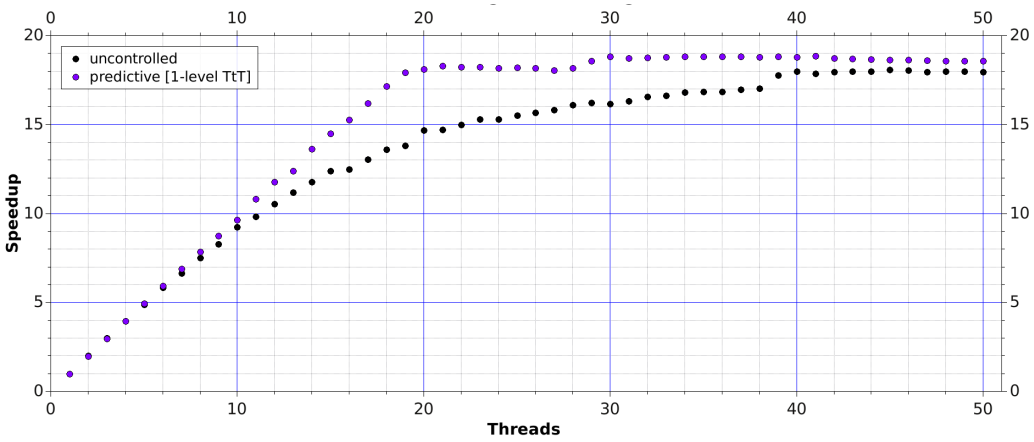

Figure 3.10: A comparison of intra-event speedups, achieved with reactive scheduling (thus *uncontrolled* task emissions) and predictive scheduling. The latter is based on **1-level task-to-task** (1-level TtT) ranking. The comparison is a function of the number of threads granted to the framework. Across the range, the improvement in speedup goes up to 30%. Configuration: LHCb Reconstruction[U-10ms] @ Machine-2S48T. 1k events processed.

3.2.2 Global task-to-task asymmetry

The ranking, based on global task-to-task asymmetry, is a generalization of the case outlined in the previous section. Contrary to 1-level ranking, it *cumulatively* engages all levels of task precedence rules, including the deepest ones. Thereby, the rank of a given task is calculated as the total number of the tasks that either directly, or indirectly (up to arbitrary level), consume at least one of its data products (see figure 3.11).

This type of ranking can also be completed at configuration time of the framework.

By definition, the cumulative task-to-task ranking provides a more comprehensive assessment of task importance. It was thus possible to anticipate that a supplementary improvement in speedup is feasible.

This, however, has not occured to be true. Figure 3.12 reveals the scalability of speedup in predictive mode based on cumulative task-to-task ranking. The measured speedup is clearly inferior to the one based on 1-level ranking. More so, it even goes below the one of reactive scheduling in the region

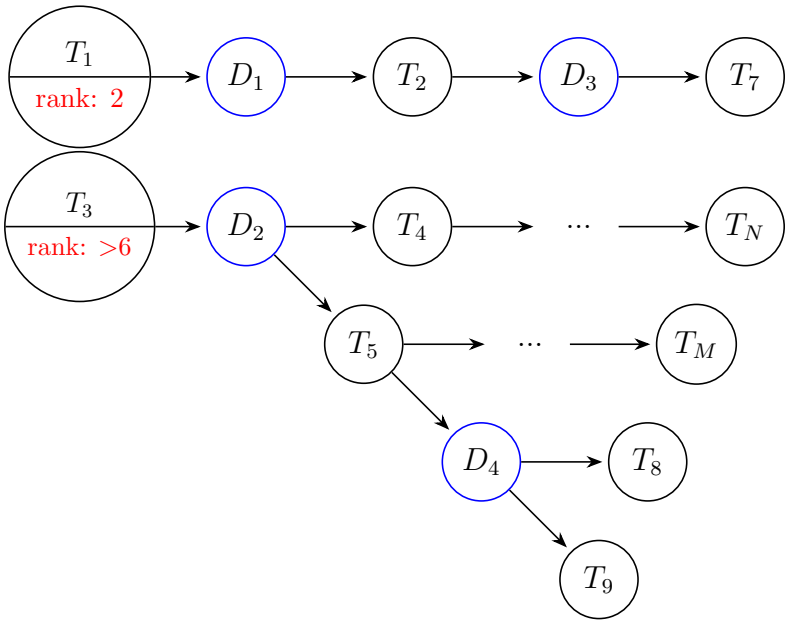

Figure 3.11: A schematic example of the cumulative task-to-task asymmetry. A generalization of the case, outlined in 3.7, up to the boundaries of the entire task ensemble.

between 10 and 15 threads, then raising the improvement to up to +9.8% at 30 threads.

A possible reason for such speedup degradation is related to the nature of cumulative task ranking. It is correct to assume that the task prioritizations, driven by this ranking strategy, selects the execution paths of greater task emission potential. However, it is wrong to assume that the potential is not deferred either by the CF rules, or by the topology of the DF path itself. There is no guarantee that a prioritized execution path will necessarily emit its related tasks immediately, or, at least, quickly enough to arrange the task emission avalanches the predictive scheduler is trying to provoke.

We can conclude that the cumulative task ranking, performed in the DF realm of precedence rules, has a limited application in the problem of throughput maximization.

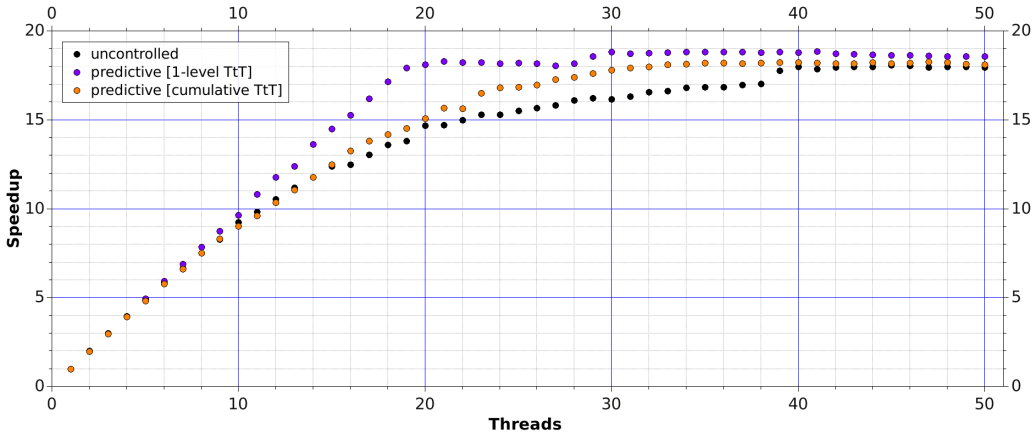

Figure 3.12: A comparison of intra-event speedups, achieved with reactive scheduling (thus *uncontrolled* task emissions) and predictive scheduling. For the latter, the case of the **cumulative task-to-task** (TtT) ranking is included with respect to figure 3.10. The comparison is a function of the number of threads granted to the framework. Across the range, the improvement in speedup of the new measurement shows an improvement of up to 9.8% with respect to reactive scheduling. Configuration: LHCb Reconstruction[U-10ms] @ Machine-2S48T. 1k events processed.

3.2.3 Critical path method

In this section I will apply the ideas of the critical path method (CPM), developed by Kelley and Walker [19], to the context of the intra-event concurrency in Gaudi.

In order to prioritize the execution of tasks along the critical path of task precedence patterns, a dedicated task ranking technique is required. In the context of Gaudi, the per-event precedence pattern is dynamic and unpredictable. Thus, the ranking has to combine the topological constraints of the task precedence rules and the task execution times.

Unfortunately, the very nature of the HEP domain makes it impossible to come up with a static task execution time map. The reason is that the task execution time depends on particular physics of an event. Moreover, the events are processed, in general, in unpredictable order. Another level of complication arises from the wide variations in hardware performance and computing environments being involved in the data processing. Hence, spe-

cial mechanisms are required to discover the execution times of tasks during data processing. This would allow to dynamically detect and follow the critical and sub-critical task precedence paths, reducing in this way the event processing times.

In this section I investigate a special case of the critical path method and estimate its potential in the context of the LHCb Reconstruction. Let me point out the following observation:

Observation 3.5 (*Critical path and task eccentricities*)

A task precedence scenario, in which the task ensemble has a uniform time distribution, makes the problem of critical path detection independent of task execution times. The task node eccentricity becomes the only key parameter.

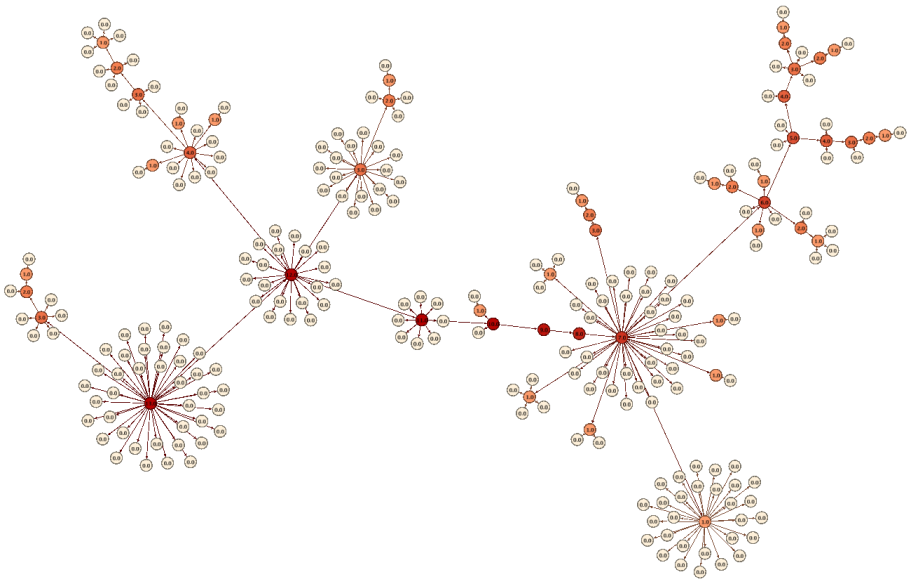

Figure 3.13: An example of the critical and sub-critical task paths, estimated using task nodes eccentricities in a task precedence pattern, materialized from the precedence rules of the LHCb Reconstruction of a concrete event. Color intensity represents eccentricity-based rank of a task. Configuration: LHCb Reconstruction.

An immediate consequence of observation 3.5 is:

Corollary 3.1. The main critical path of an equitime task precedence graph coincides with its *diameter*.

Figure 3.13 shows an example of the critical and sub-critical task paths, estimated using the DF realm eccentricities of the task nodes for the case of the equitime LHCb Reconstruction. For the precedence topology under consideration, the *diameter*, and thus its critical path, has the length of 17 task nodes.

An interesting topological feature is that the critical path experiences a triple split, taking place in the upper right corner of the graph. This makes it important to direct the task execution flow in this region to all three splitting paths, if it is possible. This equally refers to all sub-critical paths, present in the graph. If this prescription is intentionally ignored or is not holding for another reason, the event processing will take longer than the time of the critical path.

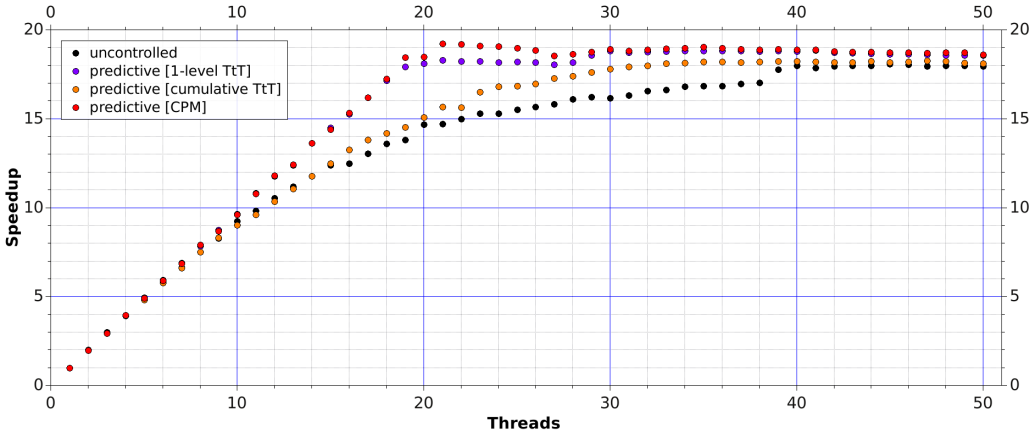

Figure 3.14: A comparison of intra-event speedups, achieved with reactive scheduling (thus *uncontrolled* task emissions) and predictive scheduling. For the latter, the case of the **critical path method** (CPM) is included with respect to figure 3.12. The comparison is a function of the number of threads granted to the framework. Across the range, the new measurement demonstrates the best throughput when compared to all other scheduling techniques. Configuration: LHCb Reconstruction[U-10ms] @ Machine-2S48T. 1k events processed.

If compared to reactive task scheduling (figure 3.14), the CPM one gives

an improvement of up to 34% when 21 threads are alloted to the framework. It also outperforms the 1-level task-to-task regime in the rest of the range.

Such significant improvement has been achieved due to successful arrangement of the the avalanche task emissions, mentioned in section 3.1.3. Figures 3.16 and 3.17 demonstrate how the intra-event concurrency dynamics evolves when the CPM-based predictive task scheduling is used.

Having demonstrated the supremacy of the CPM-based scheduling in the intra-event mode and being driven by the aim of maximizing the peak global throughput, let me point out the implications of the CPM-based scheduling on the framework speedup in many-event and saturated operation modes.

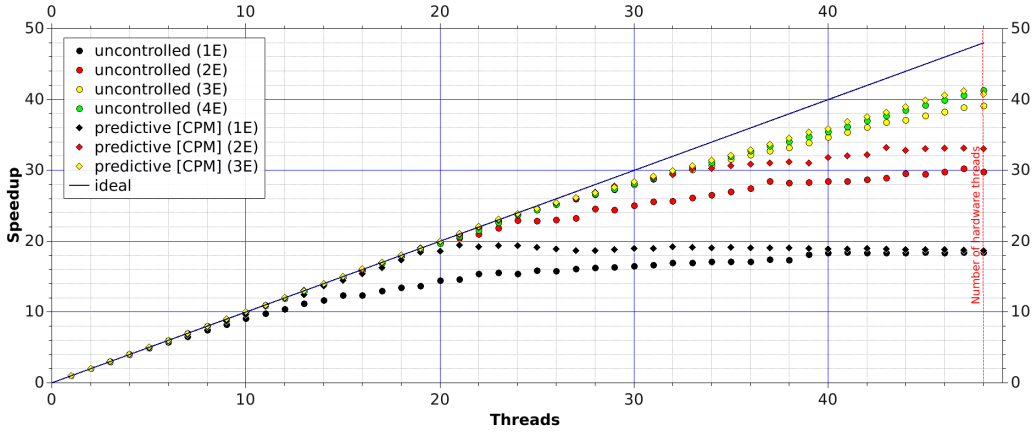

Figure 3.15: The speedups of multi-event and saturated data processing modes, operating on reactive scheduling, and predictive – CPM-based – scheduling. The comparison is a function of the number of threads, allotted to the framework. A one-to-one comparison of multi-event and saturated modes shows the total domination of the CPM method. Configuration: LHCb Reconstruction[U–10ms] @ Machine-2S48T. 1k events processed.

Firstly, a one-to-one comparison of multi-event modes (figure 3.15) shows the total domination of the CPM-based scheduling over reactive one across the entire range of threads. Secondly, the CPM-based mode of saturated operation outperforms the one, achieved in reactive mode, by almost 2% at the maximum hardware capacity of the machine. Thirdly, the new predictive mode gets saturated with 3 instead of 4 events being processed at the same time. Fourthly, it is notable that the area of ideal speedup in the CPM-based

saturated mode has extended from 15 threads by 66%, starting to degrade only at around 25 threads.

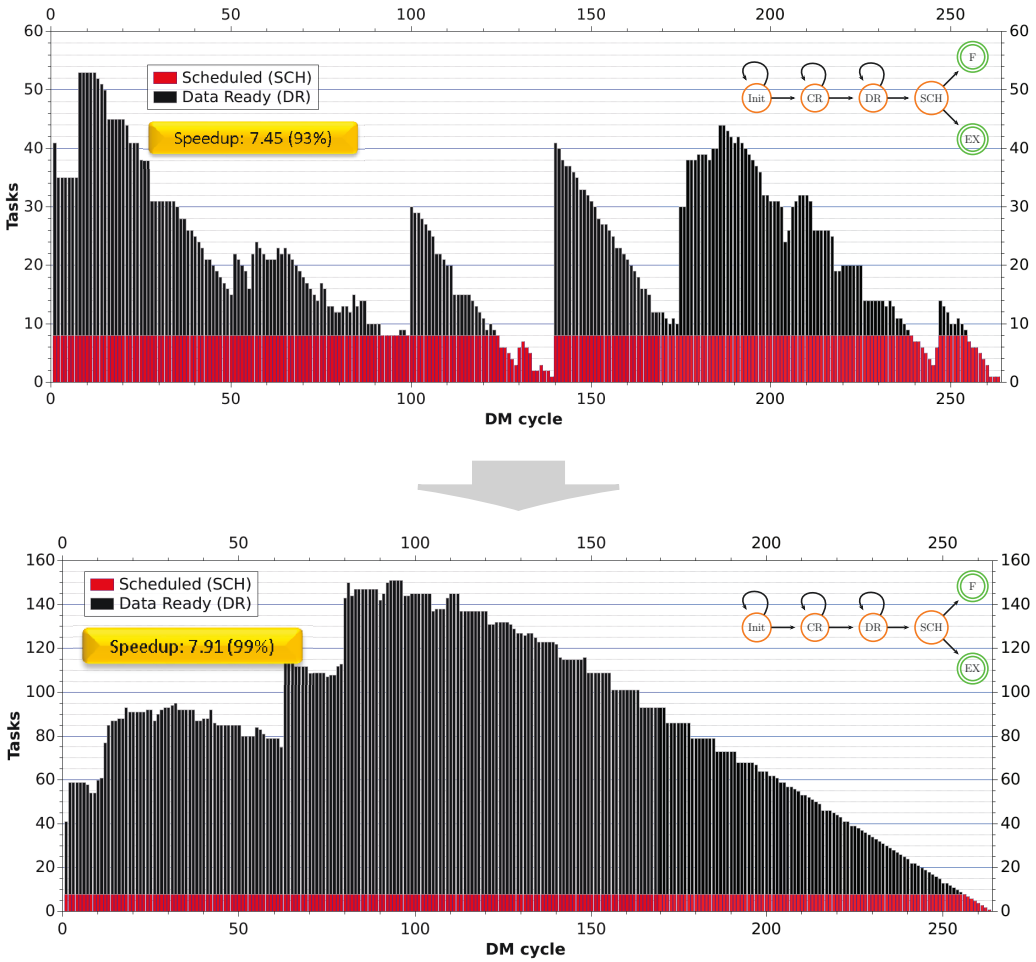

Figure 3.16: Comparison of intra-event concurrency dynamics for reactive (top) and predictive (CPM) (bottom) types of scheduling as a function of decision making cycle. **8** threads are granted to the framework. Extensive task emissions of almost tripled amplitude, arranged by predictive scheduling, mitigate the task starvation periods. Configuration: LHCb Reconstruction @ Machine-2S48T. Single event profile.

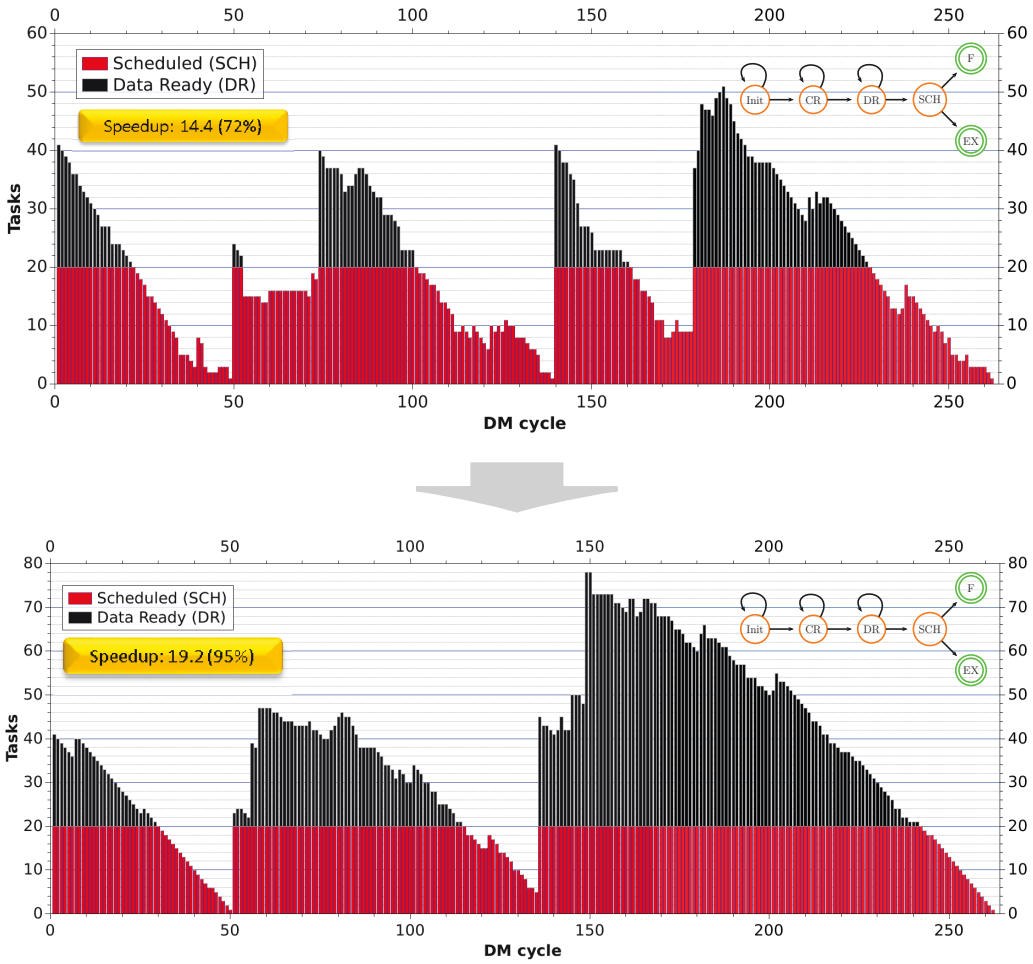

Figure 3.17: Comparison of intra-event concurrency dynamics for reactive (top) and predictive (CPM) (bottom) types of scheduling as a function of decision making cycle. **20** threads are granted to the framework. Task emission dynamics, improved by 40% in predictive scheduling, mitigate the task starvation periods, raising speedup from 72 to 95% of theoretical maximum. Configuration: LHCb Reconstruction @ Machine-2S48T. Single event profile.

Chapter 4

Scheduling of heterogeneous tasks in Gaudi Hive

In chapter 3, I investigated the topological factors of throughput limitations in Gaudi Hive, as well as suggested various scheduling techniques to mitigate them. These factors arise from the intra-event task precedence constraints and influence the balance between the inter- and intra-event concurrency dimensions. The precedence constraints are independent of the nature of computations performed by the tasks.

In this chapter I will focus on another aspect of throughput maximization that concerns the heterogeneous nature of tasks commonly used in HEP experiments. I will outline the limitations of task nature agnostic scheduling employed in the previous versions of the Gaudi Hive prototype. This will be followed by a quantitative evaluation of an alternative – task nature aware – scheduling approach, allowing to further increase the framework throughput.

4.1 Problem formulation

I define a heterogeneous set of tasks as follows:

Definition 4.1

A set of tasks is called heterogeneous if its tasks perform any of the following operations:

1. *disk I/O operations*
2. *network I/O operations*
3. *synchronization mechanisms (locks, condition variables, etc.)*
4. *offload computations*

In the context of a HEP experiment, the I/O-bound operations are a commonplace. For instance, in order to be processed an event has to be first loaded from persistent storage into RAM. Likewise, it must be written back to the storage when the processing is completed. Moreover, depending on event specifics, the framework may decide to acquire various auxiliary metadata in order to proceed with event processing. An example of this is loading the experiment's detector conditions associated to a given event time.

In multithreaded environment, the tasks are often based on synchronization mechanisms.

However, the category of tasks employing the offload computation are not common in the modern HEP data processing. At the same time, they are of a special interest for experiments [20, 21] from the viewpoint of the potential they have in massively parallel event processing (Single Task Multiple Event). In this context, general-purpose graphics processing units (GPGPU) currently get most of the focus among numerous R&D groups of the experiments. Furthermore, coprocessor devices arouse a growing interest in the HEP communities [22]. For instance, the Intel® Many Integrated Core Architecture [23] demonstrates a significant general potential for the Multiple Task Multiple Events strategy due to its support of both the offload and native execution modes, a more generic multithreading model and the ability to run on standard programming tools and methods [23].

A characteristic that makes all the heterogeneous operations common is that they rely on a system call that is either blocking (because it must precede the following steps), or that cannot gain much from being asynchronous (because the backlog of *out-of-order* steps allowed to concur it within the task is too low to hide the blocking latency). It is a consequence of that a GAUDI task has, typically, a very narrow specialization.

As described in section 1.2, GAUDI HIVE uses TBB for thread and task management. The library was designed for CPU-bound computations and thus employs non-preemptive task scheduling policy. This allows it to be very efficient with compute intensive tasks by avoiding costly context switches that occur in preemptive scheduling policy.

The non-preemptive scheduling policy, however, has a downside if used with heterogeneous tasks. Every blocking, non-preempted task wastes the logic thread it was scheduled to for the whole period of blocking [24], which results in CPU being starved of tasks. This results in suboptimal performance.

Since the aim of throughput maximization is so important, and the category of blocking operations is so wide, there was a clear necessity of a new mechanism for GAUDI HIVE that would allow to get control on and, possibly, even profit from the above described effect.

4.2 Tolerating task heterogeneity

A well-known, systematic technique to mitigate the effect of blocking operations relies on the mechanism of blocked thread handling, employed in the Linux process scheduler.

Since Linux 2.6.23, the operating system (OS) kernel uses the Completely Fair Scheduler (CFS) [25] – the first implementation of a weighted fair queuing process scheduler that is broadly used in a general-purpose OS [26].

In CFS, as soon as a thread is blocked and becomes waiting for control to return, it is removed from the red-black process tree – the run time allocation structure of CFS – so that no CPU time is allocated to it. When the thread is awoken by the system interrupt it is waiting for it is re-inserted to the red-

black tree and allocated its CPU share according to general fair share rules. The latter also include the concept of *sleeper fairness* [27], which considers the awakening threads equivalent to those on the process tree. This means that when a sleeper thread eventually returns, it is immediately prioritized and the CPU share it gets is proportional to its waiting time and, hence, fair.

To use this mechanism, a blocking task has to be scheduled to a full-blown thread [24], bypassing the TBB scheduler. This leaves a TBB thread, that would be blocked otherwise, for a CPU-intensive task, thus increasing the overall throughput. The blocking thread is delegated to the CFS and, being effectively hidden, does not affect any other thread.

By the latter, it is tacitly assumed that in order to gain in throughput such scheduling of blocking threads has to be made by oversubscribing a CPU. It is interesting to note, that, due to the nature of blocked threads and the efficient mechanism of its handling in the CFS, the depth of oversubscription is loosely constrained and can reach the values significantly exceeding the actual number of physical threads alloted to the framework.

It is of uttermost importance to realize that efficient oversubscription is only possible if associated context switching (CS) is rare enough. This is dictated by the fact that CS can be expensive, as it involves saving and restoring of thread's register state and cache. The requirement of rare context switching is satisfied if the amount of compute-intensive operations in a blocking task is significantly lower than that of blocking ones. Although such isolation of blocking operations could be a good guideline for future task developments, the reality is that most of the blocking tasks that are currently used in data processing can have a comparable share of both operation types.

Therefore, an investigation was needed to evaluate the technique, suggested in this section, under the conditions approximated to the ones of a HEP experiment.

4.3 Throughput maximization: CPU oversubscription

Measurements 4.1 (*Oversubscription*)

A series of benchmarks were carried out [28] to evaluate the scalability of speedup that can be maintained by Gaudi Hive *in processor oversubscription conditions. Several heterogeneous workflow of LHCb data processing were considered.*

4.3.1 Oversubscribing CPU with TBB

As a fiducial point, I chose the current Gaudi Hive task scheduling back-end as a mechanism for oversubscription studies.

Measurements 4.2 (*TBB-based oversubscription*)

The scalability of intra-event speedup, maintained by Gaudi Hive *in TBB-based oversubscription regime, was evaluated for non-blocking, partially blocking and blocking data processing scenarios of the LHCb experiment.*

All measurements of this subsection were taken for the case in which the scheduler makes no distinction between blocking and non-blocking tasks. This means that a CPU was oversubscribed with tasks of both types. Initially, such measurements were primarily meant to be referential and indicate the extent of speedup degradation the compute-intensive tasks can lead to in oversubscription. The results of the tests, though, brought few unexpected results.

Figure 4.1 shows intra-event speedup of data processing in the oversubscription domain. To simulate more realistic conditions, every blocking task executes a mixture of compute-intensive and blocking operations at proportion of 50%.

First of all, the *non-blocking* LHCb Reconstruction did not exhibit any significant degradation in oversubscription phase. This is still more unex-

pected given that the special GAUDI ALGORITHM, used to simulate the close to real load, was developed intentionally as compute-intensive and inefficient.

Consider the curve describing the speedup of the *mixed* LHCb Reconstruction. This scenario represents a moderate case in which only 10% of the tasks used in a given data processing workflow are blocking. The maximum associated improvement for such case goes up to 9.7% at 17 threads.

Finally, the *blocking* LHCb Reconstruction, in which all tasks are considered as blocking, shows an improvement of 119% at 17 threads.

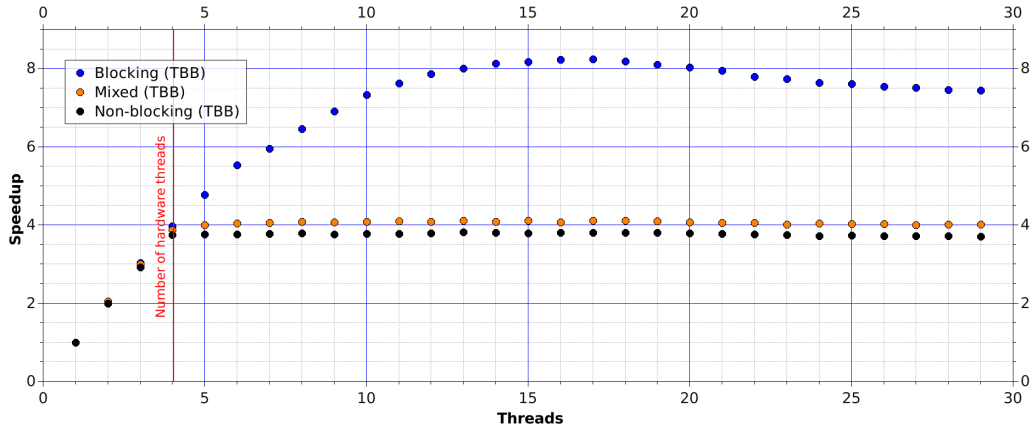

Figure 4.1: Intra-event speedup of data processing in oversubscription domain as a function of threads granted to the framework. The ***blocking*** scenario has all its tasks blocking, the ***mixed*** scenario contains only 10% of such tasks, while the ***non-blocking*** one has no blocking tasks at all. The blocking tasks spend 50% of their total run time in intensive computations. In the ***mixed*** scenario, the bloking tasks have a random distribution across the graph of precedence rules, shown in figure 2.6. The blocking scenario yields an improvement of up to 119% at 17 threads. Configuration: LHCb Reconstruction[U-20ms] @ Machine-1S4T. 1k events processed.

The results of the benchmark lead to the following observation:

Observation 4.1 (*Blocking extent of tasks ensemble*)

The more blocking tasks a given data processing scenario has, the higher intra-event speedup can GAUDI HIVE *maintain in oversubscription regime.*

The measurements presented in figures 4.2 and 4.3 quantify another dimension of throughput maximization:

Observation 4.2 (*Blocking extent of a task*)

The more blocking operations a blocking task has, the higher intra-event speedup can GAUDI HIVE *maintain in oversubscription regime.*

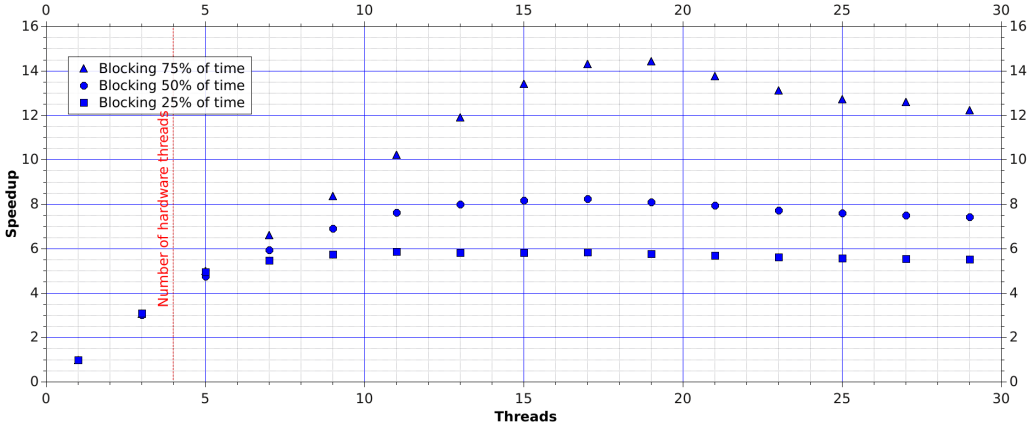

Figure 4.2: Intra-event speedup of data processing in oversubscription domain as a function of threads granted to the framework. Three ***blocking*** scenarios differ in the extent of blocking of the tasks. The latter spend 75%, 50% and 25% of their total run time in blocking operations, while the rest is spent in intensive computations. Configuration: LHCb Reconstruction[U-20ms] @ Machine-1S4T. 1k events processed.

Note that all benchmarks in this section were run on Machine-1S4T with Intel® Hyper-Threading technology disabled. This was an intentional measure aiming at isolating the oversubscription effects from distortions associated with simultaneous multithreading, which yields drastically distinct speedup improvement for compute-intensive and blocking load.

Clearly, there is no guarantee the real-life tasks will repeat the performance of the CPU Cruncher [29] in the oversubscription conditions. Moreover, there may be additional dimensions of performance degradation concerned with the effect of so called *mutual exclusion*. In particular, the real-life compute-intensive tasks can employ locks when dealing with various frame-

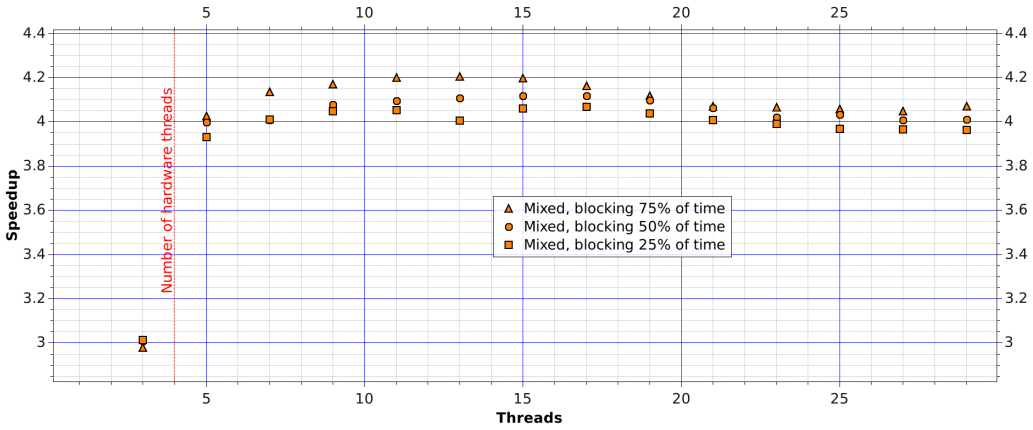

Figure 4.3: Intra-event speedup of data processing in oversubscription domain as a function of threads granted to the framework. The ***mixed*** scenario contains only 10% of blocking tasks distributed randomly across the graph of precedence rules, shown in figure 2.6. The presented scenarios differ in the extent of blocking of the tasks. The latter spend 75%, 50% and 25% of their total run time in blocking operations, while the rest is spent in intensive computations. Configuration: LHCb Reconstruction[U-20ms] @ Machine-1S4T. 1k events processed.

work components. Every time a task, holding a lock, is suspended due to CS, the other tasks holding the same lock are blocked.

It thus becomes clear that it would be a mistake to expect that the observed effectiveness of CPU Crunchers in oversubscription domain will extrapolate on real-life tasks. Hence, one must comply to the classic approach of avoiding oversubscription for non-blocking compute-intensive tasks.

For that purpose, the relatively recent concept of TBB task arenas [30] could be employed. In particular, two specialized arenas would be involved. The first, only for non-blocking tasks only with its maximum level of concurrency being fixed to the total number of available hardware threads (or, alternatively, to any other lower number). The second for blocking tasks only. However, it is clear that in order to run at peak data processing throughput the maximum level of concurrency of the second task arena is not known in advance and can only be *learned* at run time.

At the same time, the maximum level of concurrency of a TBB task

arena is designed to be fixed at configuration time, and, as of TBB version 4.4, can not be adjusted later on [30]. To bypass the problem, workarounds are possible, which require the task arena to be initialized with overflow concurrency levels. The latter, though, is not efficient. Hence, the use of the TBB-based oversubscription as a final solution for Gaudi Hive is currently problematic.

4.3.2 Composite scheduling

The TBB library was not designed for high-througput computing in conditions of heterogeneous load in general, and for oversubscription modes of operation in particular [31].

However, it was augmented it for more efficient handling of heterogeneous task sets in Gaudi Hive.

Deliverable 4.1 (*Composite scheduler*)

A composite oversubscribing scheduling mechanism was implemented for Gaudi Hive. *It is based on the collaboration of the TBB built-in scheduler and a specialized asynchronous scheduler aimed at oversubscription.*

In the mode of composite scheduling, the non-blocking tasks are handled by the TBB built-in scheduler. The blocking tasks, in turn, are managed by the specialized scheduler. It dynamically spawns a dedicated asynchronous thread for each such task, entering the oversubsciption domain only when needed.

In figure 4.4, I compare three scheduling regimes. First, the TBB in standard mode, where no oversubscription is involved. Second, the TBB scheduler, used with 15 logical threads oversubscribing 4 hardware threads alloted to the framework. Finally, the composite scheduler.

The measurements demonstrate the following:

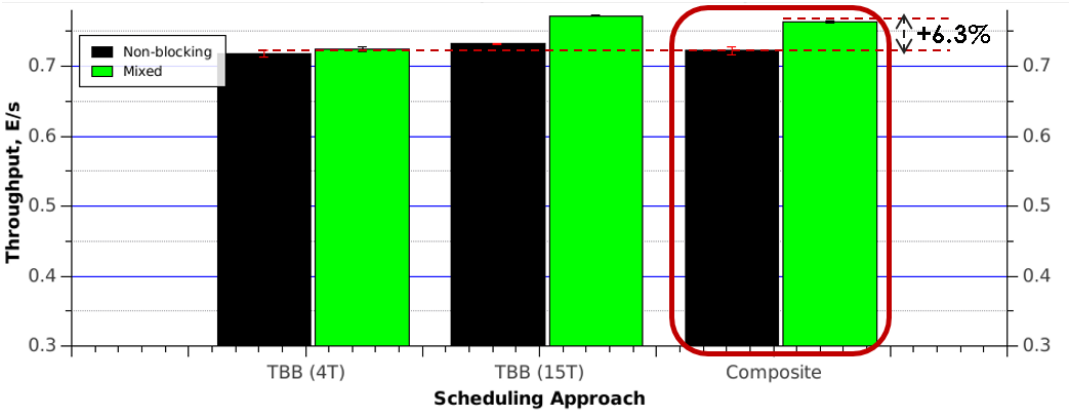

Figure 4.4: Intra-event data processing throughput of TBB (native mode), as well as both TBB and a new composite mechanism in oversubscription modes. ***Mixed*** scenario contains 10% of blocking tasks. The blocking tasks spend 50% of their total run time in intensive computations and have a random distribution across the graph of precedence rules, shown in figure 2.6. For heterogeneous tasks, the composite scheduling approach demonstrates an improvement of 6.3% in throughput relative to the non-oversubscription regime, maintained by TBB. Configuration: LHCb Reconstruction[U-20ms] @ Machine-1S4T. 1k events processed.

Observation 4.3 (*Composite scheduler*)

For heterogeneous tasks, the composite scheduling approach shows an improvement of 6.3% in throughput relative to the non-oversubscription regime, maintained by TBB.

The scheduler efficiency can be further improved. In particular, since starting and terminating a task in Linux is about 18 times faster when compared to a thread [32], a more advanced thread life-span management could reduce the scheduling overhead.

Note 4.1 (*Alternative scheduling systems*)

The mechanism for composite scheduling of heterogeneous tasks in Gaudi Hive*, and the technique behind it, are presented here as a proof of concept. Further studies can involve other, more advanced, scheduling systems that claim to have an efficient task latency hiding. One of them is* STE||AR *High Performance ParalleX (HPX) runtime scheduling system [33]. This option can potentially constitute a full-fledged solution for heterogeneous task scheduling in* Gaudi Hive*, is meant to replace the TBB scheduler and requires detailed investigations, planned for the near future.*

4.3.3 Framework throughput and offload computations

In the HEP experiments, the nature of some specific types of CPU-based tasks, used in various data processing scenarios, allows to reimplement these tasks in such a way that part of their computations is offloaded to a coprocessor. Typically, it is argued, though, that such approach is not efficient in most of the cases since the offload latency overrides the possible gains of the computation being offloaded to the coprocessor.

The infrastructure, developed for scheduling of heterogeneous tasks in Gaudi Hive (see section 4.3.2), can be used to simulate the impact of substitution of a subset of CPU-based tasks in a given data processing scenario with their coprocessor-based equivalents.

In figure 4.4, I have indirectly demonstrated that:

Deliverable 4.2 (*Offload substitution*)

Refactoring of 10% of tasks, used in LHCb Reconstruction[U-20ms], to offload 50% of their computations elsewhere, yields an improvement of 6.3% in the overall throughput in the composite regime of task scheduling.

In this simulation I assumed that 50% of computations, being offloaded from a task, take the same amount of time when executed on a coprocessor device.

In figure 4.5, I extend the throughput simulation described above. Specifically, the first two values from the left being earlier presented in figure 4.4 are augmented with four other throughput estimations for the pessimistic cases of offloaded computations being slowed down when compared to their CPU based equivalents.

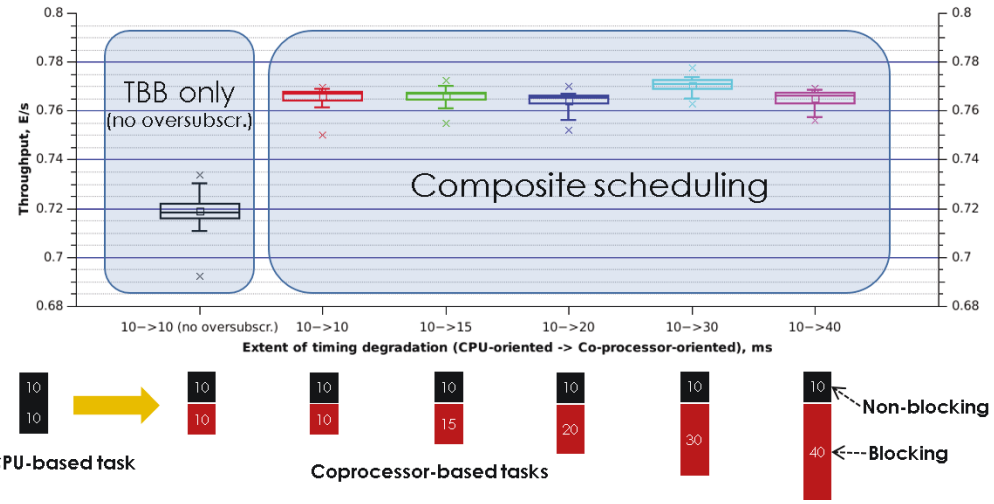

Figure 4.5: Intra-event data processing throughput in TBB- and composite-based scheduling regimes. The ***mixed*** scenario is used, which contains 10% of blocking tasks. Blocking tasks have a random distribution across the graph of precedence rules, shown in figure 2.6. Throughput is given as a function of duration of a blocking operation (e.g., of an offload computation). Configuration: LHCb Reconstruction[U-20ms] @ Machine-1S4T. 1k events processed.

Figure 4.5 proves the following observation:

Observation 4.4 (*Offload latency oblivious framework*)

The framework, when used in the composite scheduling regime, is oblivious of the latency of its offload computations.

Conclusion

This work marked a milestone in the evolution of the Gaudi framework towards massively concurrent and adaptive data processing. I suggested, implemented and benchmarked a number of qualitatively new techniques for generic and non-intrusive throughput maximization in concurrent Gaudi.

Below, I summarize my major contributions to Gaudi Hive.

Aggregate 1 (*Low latency, scalable concurrency control*)

> *I suggested, implemented and benchmarked an alternative approach for the concurrency control system (CCS) founded on the idea of graph-based decision making.*

The CCS is a key component of the Gaudi Hive task scheduler. It is essential for *reactive run-time* task precedence resolution in conditions of *unpredictable* data processing workflows. I demonstrated the following advantages of the graph-based decision making approach:

1. **Low response time** (reduction by 2x when compared to the previous approach);
2. **Constant amortized step complexity** by the number of entities of the concurrency control problem;
3. **Excellent scalability** by the number of threads allotted to data processing (tested in the *many-core* range);
4. Full-scale **precedence graph analysis** (e.g., topological consistency of task precedence rules, asymptotic intra-event concurrency analysis);

5. ***Proactive* concurrency control**.

The proactive concurrency control is the most outstanding from the viewpoint of throughput maximization, as it allowed to realize the new type of task scheduling in Gaudi Hive:

Aggregate 2 (*Predictive task scheduling*)

> *I suggested and implemented the predictive task scheduling in* Gaudi Hive. *Several look-ahead strategies were tested in a series of benchmarks to evaluate the impact of the technique on framework's throughput and scalability.*

In this context, application of the critical path look-ahead strategy to the LHCb reconstruction workflow demonstrated the best results, increasing the intra-event speedup by 34%.

The mechanisms for intra-event throughput maximization allowed to investigate the balance between the intra- and inter-event concurrency dimensions from the viewpoint of overall throughput of the framework. An important corollary from this study is:

Aggregate 3 (*Intra- and inter-event concurrency balance*)

> *With all other conditions being equal, higher peak throughput is proved in regimes that minimize the use of inter-event concurrency dimension by virtue of maximizing the intra-event one.*

Hence, the dominant strategy for throughput maximization is to prioritize the intra-event concurrency maximization. The predictive task scheduling helps to realize this strategy in generic and non-intrusive manner.

Finally, I explored the performance inefficiencies and architectural constraints that emerge in execution of heterogeneous task networks in Gaudi and delivered a solution to mitigate those limitations:

Aggregate 4 (*Infrastructure for latency oblivious task scheduling*)

> *I suggested, implemented and benchmarked a generic and adaptive mechanism for scheduling heterogeneous task networks in* Gaudi Hive.

This advancement brought the capacity to maximize the framework throughput by adapting to heterogeneous task networks of arbitrary topology in an automatic and non-intrusive manner, thus enabling the heterogeneous computing paradigm in the GAUDI framework.

List of Figures

List of Tables

Appendix A

Testbed for benchmarking: 2S-48T

Machine-2S48T specifications:

- Intel® Xeon® CPU E5-2695 v2 @ 2.40GHz
- 2 sockets, 12 cores, Hyper-Threading: **enabled**
- L2 256KB, L3 30 MB

Appendix B

Testbed for benchmarking: 1S-XT

Machine-1S8T specifications:

- Intel® Core™ i7-3770 CPU @ 3.40GHz
- 1 socket, 4 cores, Hyper-Threading: **enabled**
- L2 256KB, L3 8MB

Machine-1S4T specifications:

- Intel® Core™ i7-3770 CPU @ 3.40GHz
- 1 socket, 4 cores, Hyper-Threading: **disabled**
- L2 256KB, L3 8MB

Appendix C

Workflow scenario

The LHCb Reconstruction scenario represents a modified version of the workflow used in sequential reconstruction of LHCb event data during Run-I phase of the LHC project at CERN. It is also being used in the current Run-II phase.

Where it matters, a reference to the scenario is followed by a time mapping specification: *[U–Xms]* stands for the uniform time mapping of X milliseconds, while *[N]* – for non-uniform time mapping. Appendix C.3 contains more details about the time mappings.

C.1 Intra-event task precedence rules

The topology of the CF and DF task precedence rules, shown in figure C.1, is identical to the one, used in sequential event reconstruction, with one exception being that the indirect data dependencies between tasks were ignored. The latter simplification was a compulsory measure driven by the lack in sequential Gaudi of an explicit mechanisms for tracking data dependencies. This led to noticable difficulties in tracking the direct data dependencies (which were, nevertheless, taken into account), as well as the indirect ones.

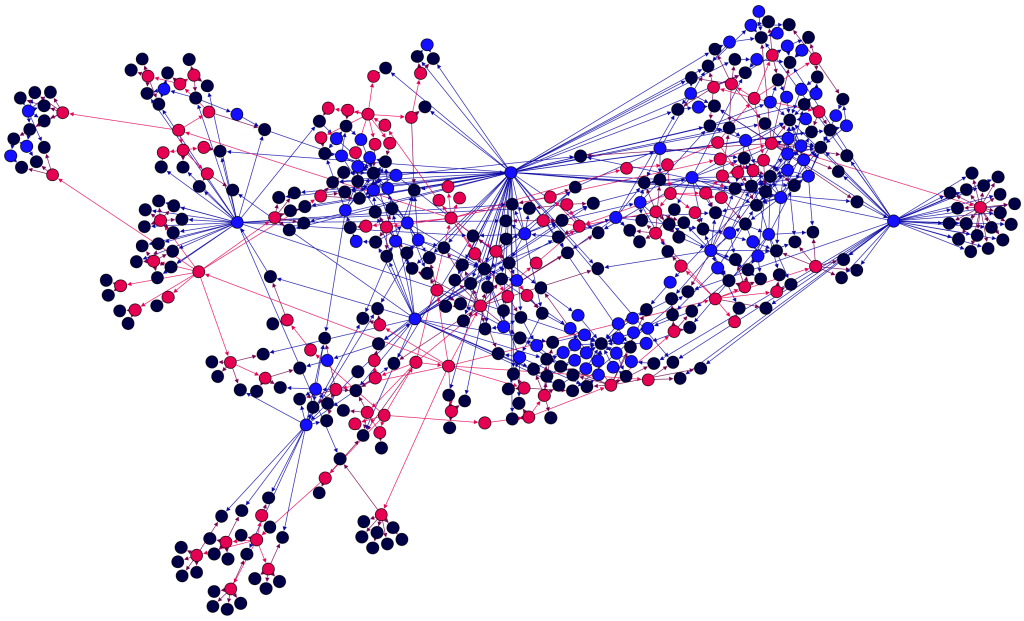

Figure C.1: Graph of CF and DF task precedence rules in reconstruction of LHCb event data. Black nodes represent tasks, blue nodes denote data items, while red ones indicate CF decision hubs.

C.2 Tasks

The Gaudi Algorithms and other components, used in sequential data processing, were designed for single-threaded execution. This makes their use impossible in the context of multithreaded Gaudi.

However, an acceptable alternative was chosen for the purposes of task scheduling studies and performance evaluation. In particular, in all benchmarks, discussed in this thesis, a special Gaudi Algorithm – *CPU Cruncher* [29] – was wrapped in a task. It was designed to perform *inefficient* and intensive use of the CPU, as well as to have configurable precedence constraints.

It is interesting to note that, in the intra-event throughput tests, the use of Gaudi Algorithms that are just sleeping in `std::this_thread::sleep_for` did not reveal any significant differences when compared to Gaudi Algorithms, performing heavy CPU crunching.

Nevertheless, in all benchmarks throughout the thesis I prefer to use the crunching load, which is still more realistic, in order to avoid too optimistic

benchmark results in situations, where the task load becomes of a greater importance (e.g., in inter-event throughput tests in conditions of machine saturation).

C.3 Task execution time mapping

Throughout the thesis, two types of time mappings are considered.

C.3.1 Uniform mapping

As the name states, the uniform mapping establishes an equal amount of execution time for each task. In each specific case, and where it matters, the actual time value is specified.

C.3.2 Non-uniform mapping

The non-uniform mapping assigns different time values for each task. In this schema, the time each CPU cruncher takes to execute is close enough to the actual execution time of its realistic task counterpart, used at the same slot of precedence rules in sequential event reconstruction.

Figure C.2 shows the orders of magnitude of non-uniform time mapping.

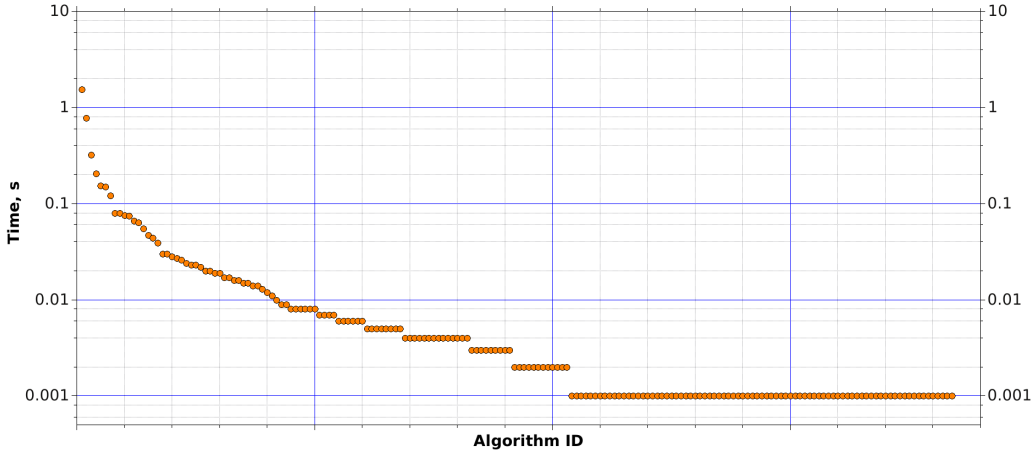

Figure C.2: Log-lin distribution of non-uniform task execution times. Note that the values represent the wall-clock time of sequentially executed tasks, thus including the I/O part. This is intentional.

Bibliography

[1] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[2] G. Barrand et al. GAUDI - a software architecture and framework for building HEP data processing applications. *Comput. Phys. Commun.*, 140:45–55, 2001.

[3] A. Augusto Alves, Jr. et al. The LHCb Detector at the LHC. *JINST*, 3:S08005, 2008.

[4] LZ Collaboration. LZ Conceptual Design Report, LBNL, 2015. http://hep.ucsb.edu/LZ/CDR/, section:15-5.

[5] P. Mato and S. Smith. User-friendly parallelization of gaudi applications with python. *Journal of Physics: Conference Series*, 219(4):042015, 2010.

[6] Concurrent Framework Project (CF4Hep). http://concurrency.web.cern.ch/GaudiHive, 2012. [Online; accessed 24-Feb-2016].

[7] B. Hegner, P. Mato, and D. Piparo. Evolving LHC data processing frameworks for efficient exploitation of new CPU architectures. In *IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, pages 2003–2007, Oct 2012.

[8] P. Mato. GAUDI-Architecture Design Document, LHCb@CERN, December 1998. v9.

[9] M. Clemencic, H. Degaudenzi, P. Mato, S. Binet, W. Lavrijsen, C. Leggett, and I. Belyaev. Recent developments in the LHCb software framework textscGaudi. *Journal of Physics: Conference Series*, 219(4):042006, 2010.

[10] M. Cattaneo and P. Maley. GAUDI – LHCb Data Processing Applications Framework. Users Guide., CERN, December 2001. v9.

[11] Task-Based Programming. https://software.intel.com/en-us/node/506100. [Online; accessed 26-Feb-2016].

[12] M. Clemencic, B. Hegner, P. Mato, and D. Piparo. Preparing hep software for concurrency. *Journal of Physics: Conference Series*, 513(5):052028, 2014.

[13] A. Srinivasan and J.H. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *15th Euromicro Conference on Real-Time Systems*, pages 51–59, July 2003.

[14] I. Shapoval. GAUDI HIVE: evolution of execution flow management. Technical report, CERN, April 2014. Annual Concurrency Forum.

[15] I. Shapoval et al. Graph-based decision making for task scheduling in concurrent gaudi. In *IEEE Nuclear Science Symposium & Medical Imagine Conference Record*, November 2015.

[16] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[17] J. D. C. Little. A proof for the queuing formula: $l = \lambda w$. *Operations Research*, 9(3):383–387, 1961.

[18] I. Shapoval and M. Clemencic. Graph-based scheduling in GAUDI HIVE. Technical report, CERN, February 2015. Forum on Concurrent Programming Models and Frameworks.

[19] J. E. Kelley, Jr and M. R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint*

IRE-AIEE-ACM Computer Conference, IRE-AIEE-ACM '59 (Eastern), pages 160–173, New York, NY, USA, 1959. ACM.

[20] A. Badalov, D. Cámpora, N. Neufeld, and X. Vilasís-Cardona. LHCb GPU acceleration project. *Journal of Instrumentation*, 11(01):P01001, 2016.

[21] D. Emeliyanov and J. Howard. GPU-Based Tracking Algorithms for the ATLAS High-Level Trigger. *Journal of Physics: Conference Series*, 396(1):012018, 2012.

[22] A. Nowak, G. Bitzes, A. Dotti, A. Lazzaro, S. Jarp, P. Szostek, L. Valsan, M. Botezatu, and J. Leduc. Does the Intel Xeon Phi processor fit HEP workloads? *Journal of Physics: Conference Series*, 513(5):052024, 2014.

[23] Intel® Many Integrated Core architecture. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html, 2010. [Online; accessed 26-Feb-2016].

[24] J. Reinders. *Intel® Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2007.

[25] I. Molnár. Completely Fair Scheduler. http://lwn.net/Articles/230501/, 2007. [linux-kernel (Mailing list); Online; accessed 27-Feb-2016].

[26] T. Li, D. Baumberger, and S. Hahn. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-robin. *SIGPLAN Not.*, 44(4):65–74, February 2009.

[27] T. Jones. Inside the Linux 2.6 Completely Fair Scheduler. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/, 2009. [IBM developerWorks; Online; accessed 27-Feb-2016].

[28] I. Shapoval and M. Clemencic. GAUDI HIVE on the landscape of heterogeneous computing. Technical report, CERN, September 2015. Forum on Concurrent Programming Models and Frameworks.

[29] D. Piparo. A Gaudi algorithm for intense CPU crunching. https://gitlab.cern.ch/gaudi/Gaudi/blob/master/GaudiHive/src/CPUCruncher.cpp, 2012. [Online; accessed 14-Feb-2016].

[30] Intel® TBB Task Arenas. https://software.intel.com/en-us/node/506359, 2014. [Online; accessed 28-Feb-2016].

[31] J. Reinders. *Intel® Threading Building Blocks*, chapter 9, pages 133–134. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2007.

[32] J. Reinders. *Intel® Threading Building Blocks*, chapter 9, page 135. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2007.

[33] STE||AR High Performance ParalleX. http://stellar.cct.lsu.edu/projects/hpx/, 2011. [Online; accessed 29-Feb-2016].

Acknowledgments

First of all, I am thankful to my supervisors for the full-fledged support they rendered in the course of my PhD. In particular, I thank Dr. Marco Clemencic for the numerous discussions, which helped to proof my ideas and results put forward in this work. I thank Dr. Luca Tomassetti for organizing the best UNIFE setting I could think of for my PhD.

I am expressing my gratitude to the LHCb Management, especially to Dr. Marco Cattaneo, for the most comprehensive support and for enabling the rare opportunity to do a PhD at CERN. It was an honor for me to be part of the LHCb Core Software team all these years.

I extend my appreciation to the leaders of the Ferrara team in LHCb – Dr. Stefania Vecchi, Dr. Concezio Bozzi and Dr. Wander Baldini – for the broad assistance in promoting the results of this work at several scientific events of the highest profile.

I also thank my colleagues at KIPT – Prof. Iurii Raniuk and Prof. Anatolii Dovbnya – for many valuable advices in the course of my PhD.

I dedicate this work to my family and address my special thanks to my parents and grandfather for the inexhaustible encouragement and confidence in me, and to my wife for standing side by side and being my Muse throughout this time.