Status of the calibration and alignment framework at the Belle II experiment

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 032032 doi:10.1088/1742-6596/898/3/032032

Status of the calibration and alignment framework at the Belle II experiment

D Dossett[1], M Sevior[1], M Ritter[2], T Kuhr[2], T Bilka[3] and S Yaschenko[4] for the Belle II Software Group

[1]School of Physics (David Caro Building), The University of Melbourne, VIC 3010, Australia
[2]Ludwig-Maximilians University Munich, Excellence Cluster Universe, Boltzmannstr. 2, 85748 Garching, Germany
[3]Charles University, Ovocn trh 3-5, Prague 1, 116 36, Czech Republic
[4]DESY, Platanenallee 6, 15738 Zeuthen, Germany

E-mail: david.dossett@unimelb.edu.au, martines@unimelb.edu.au, martin.ritter@lmu.de, thomas.kuhr@lmu.de, tadeas.bilka@gmail.com, sergey.yaschenko@desy.de

Abstract.
The Belle II detector at the Super KEKB e^+e^-collider plans to take first collision data in 2018. The monetary and CPU time costs associated with storing and processing the data mean that it is crucial for the detector components at Belle II to be calibrated quickly and accurately. A fast and accurate calibration system would allow the high level trigger to increase the efficiency of event selection, and can give users analysis-quality reconstruction promptly.

A flexible framework to automate the fast production of calibration constants is being developed in the Belle II Analysis Software Framework (basf2). Detector experts only need to create two components from C++ base classes in order to use the automation system. The first collects data from Belle II event data files and outputs much smaller files to pass to the second component. This runs the main calibration algorithm to produce calibration constants ready for upload into the conditions database. A Python framework coordinates the input files, order of processing, and submission of jobs. Splitting the operation into collection and algorithm processing stages allows the framework to optionally parallelize the collection stage on a batch system.

1. Belle II Detector

The Belle II detector [1] and the SuperKEKB accelerator are currently under construction at the KEK laboratory in Tsukuba, Japan. The aim of this next generation B factory experiment is to collect 50 times more data than its predecessor Belle [2] and to use this data to search for new physics in a variety of B meson, charm hadron, or τ lepton decays with unprecedented precision.

2. Belle II Analysis Software Framework

The Belle II Analysis Software Framework (basf2) [3] is a C++/Python framework to process the events recorded by the Belle II detector. Events are grouped into runs which mark a data-taking period with stable operating conditions. Recorded events are processed one by one using a sequential set of algorithms called *modules*. Each module can produce data objects ready for

subsequent modules to use. Chaining multiple modules together allows processing tasks such as track fitting, vertex reconstruction, or ntuple creation to be performed.

The Belle II Conditions Database [4] manages conditions data on run granularity. Runs are expected to have a maximum length of around 8 hours of data taking. However since they are meant to define a period of stable conditions, runs will be changed more frequently if the Belle II online monitoring requests a run change. Payload files containing calibration constants which are defined for a certain interval of validity (IoV) are grouped into so-called global tags. A global tag can be specified for a basf2 process, allowing different payloads to be downloaded and matched to the IoV used in the data. The database client [5] can also use a local, file-based backend which reads the list of payloads from an index file and the payload contents from files in a given directory. The client allows the creation of local payload files for IoVs, which can be later uploaded to the main conditions database.

3. Calibration Processing

Having the most accurate calibration constants available for the first (prompt) reconstruction of raw data is essential to several experiment goals. The High Level Trigger (HLT) relies on accurate alignment information for track reconstruction. Less accurate constants leads to a less efficient trigger, and a reduction in the useful data for analyses. Additionally, the PXD vertex detector produces far more information than can be saved. Therefore only information contained in calculated Regions-of-Interest (RoI) is saved by the PXD vertex detector. Reconstructed HLT track information is used to make these RoI calculations and it is not yet clear what margin for error exists in the ROI finding algorithm. As any data not in the RoI is unrecoverable, planning from the start to have the most accurate alignment constants possible is paramount.

During the first years of data taking, Belle II will be in direct competition for new physics results with LHCb and its coming upgrade. Predictions suggest that on the current schedule, the critical factor for first discovery and world's best measurements will be the time taken to go from data taken to analysis completed. Therefore, in order for Belle II to be competitive in its physics programme, analyses should have the best quality data immediately in order to begin as soon as possible. Delivering analysis quality datasets promptly, without the need to wait for lengthy reprocessing, is therefore a key goal.

Figure 1 shows a simplified overview of the necessary steps in data production before physics analysis can begin. The mDST data contains reconstructed objects that depend on calibration constants and is the base data type that physics skims will use as input. However, the Raw data files will have to be moved to tape storage within a few days of the data being taken. Attempting to run mDST reconstruction after moving the Raw files to tape is a much slower process as they must be re-staged back to disk.

It should be clear that the creation of analysis quality calibration constants is a potential bottleneck to the physics goals of the experiment which should be addressed. An additional benefit of producing analysis quality constants for the prompt reconstruction is that there is less need to perform reprocessing of the data. This gives reductions in CPU time and data storage which can be freed up and used to expand the reach of experimental goals.

4. Belle II Calibration Framework

At Belle II, the design of a calibration framework (CAF) to calculate calibration constants had several initial requirements:

(i) It should be able to process multiple different calibrations and run them in a predefined order.

(ii) The input data for each calibration should not necessarily be the same.

(iii) Parallel processing should be used wherever appropriate to decrease the processing time.

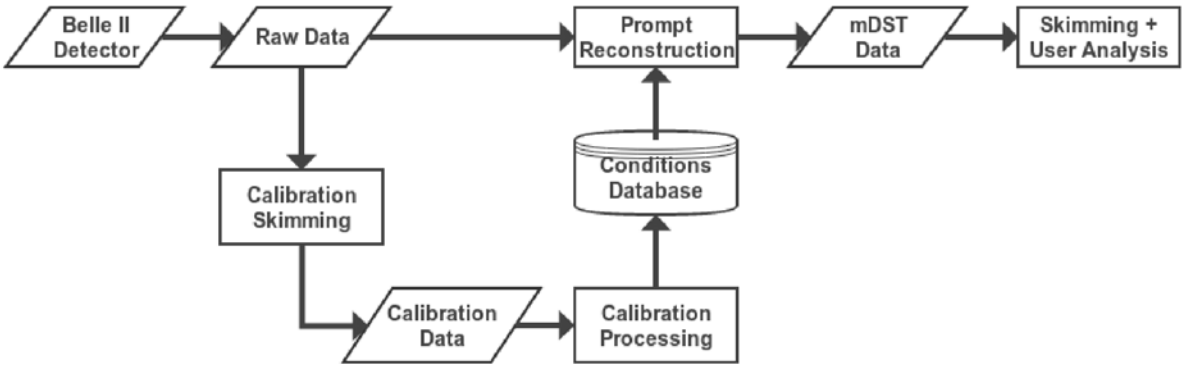

Figure 1. A simplified overview of the processing required for mDST production at Belle II.

(iv) Calls to the main conditions database should be minimised to reduce the access load.

(v) Detector experts should be encouraged to write algorithms with similar structure so that the processing framework needs to account for fewer special cases.

(vi) Detector experts should be able to test their algorithms in the framework with minimal knowledge and set up time.

Python steering files are widely used by Belle II analysts to configure and run basf2 event processing. For easier developer adoption of the CAF it should not be necessary to learn to use a complex external system to test new calibrations. Therefore, to satisfy (vi) the decision was made to write the main user interface to the CAF using Python in the basf2 environment. This additionally provides immediate access to both Python's extensive standard library and many useful basf2 specific extensions.

Since basf2 is used within the Belle II experiment to perform event processing, any CAF must include a basf2 process at some point in the workflow to extract the relevant data from the event. Each event is independent from all others, meaning that parallelizing any event processing is a relatively simple task. basf2 already provides local multiprocessing over multiple cores via the use of a ring buffer system. Events read from input files can be sent to separate processes waiting to run parallel copies of the basf2 module sequences. They then pass required data to a single output file. basf2 provides the tools to do this seamlessly with little knowledge required from a developer.

However, a calibration may need to extract data from a large number of input events, O(10M), before it can succeed. Even with basf2 multiprocessing on several cores this could take up to an hour. If iteration is required or special reconstruction is added to the process, it could take O(Day) to finish. Therefore it is necessary to employ a computing cluster with a batch queue system; this allows a large number of input events to be split between many smaller independent basf2 processes. Splitting the overall extraction of data from basf2 events over multiple input files is achieved by the Python framework, described in 4.3.

In order to satisfy (v) two C++ virtual base classes were developed, the *CalibrationCollectorModule* and the *CalibrationAlgorithm*. The collector module is a basf2 module with some extra functionality, described in 4.1. The base class additionally ensures that any data objects produced in separate jobs can be merged later. Therefore a developer writing a collector module need not know how to parallelize code in any way. The Python framework, see 4.3, organizes the submission of separate basf2 jobs and merging of collected data invisibly.

The algorithm class is a simple C++ virtual base class, described in 4.2. It uses the output data from the collector step and creates the final database constants. The base algorithm class CAF provides no extra help parallelizing the algorithm code. However, in general any calibration that takes a long time to process is spending most of its time in the collector module event processing step.

The goal is for developers writing a new calibration to only have to create two classes, inheriting from these base ones. They should be able to concentrate on the specifics of their own calibration, while allowing the framework to organize the constants and data into the correct IoVs. The Python framework then provides a user interface allowing multiple calibrations to be configured, chained together, and to use a batch system to parallelize the collector step.

4.1. Collector Module

This is the first C++ class which detector developers write. The collector module is used to create simple data objects that a calibration algorithm can use later e.g. Ntuples or histograms. From a developer perspective, when creating a new collector module derived from the base class there are only two member functions that have to be implemented:

(i) *prepare()*
This method runs before the event loop of the basf2 process begins. It is used to create the persistent data objects that will be filled during the event loop.

(ii) *collect()*
This method runs once per event during the basf2 process. It is used to fill the persistent data objects with data from the event.

A developer can make their module as complex or simple as they wish, so long as these two member functions are implemented. The base class automatically creates separate data objects for each run in the input data. The object corresponding to the current run is returned whenever the code asks for the data object created in the prepare() function. In this way, the collect() function can be written without the need to query which object to fill with the event data.

4.2. Algorithm Class

Every calibration at Belle II requires the production of constants, which can be loaded by the conditions database interface. Developers create a C++ class that derives from the CalibrationAlgorithm base class. This is not a basf2 module looping over events, but a C++ class which uses the data objects produced by the collector module. Each new algorithm must implement the *calibrate()* member function. This function can request and automatically merge the correct data objects corresponding to the IoV being calibrated. The calibrate function returns one of four possible codes corresponding to the results:

OK:
The algorithm was successful and there is no need to iterate the collector stage again with the new constants. The constants for this calibration are saved to the current local database.

Iterate:
The algorithm was successful but iteration of the collector stage (and algorithm) with the new constants is requested. The constants for this calibration are saved to the current local database.

Not enough data:
The algorithm did not have enough data collected in the requested IoV to succeed. No constants are saved.

Failure:
The algorithm failed for any other reason. Logging messages should indicate the cause. No constants are saved.

These return codes allow an automated framework to make informed decisions about how to proceed with the requested operation.

4.3. Python Framework

Running the collector module and algorithm class separately is easy to do for developers using normal basf2 Python steering files. However it is potentially difficult to create scripts for more complex tasks. For example, a developer might want to iterate a calibration, or pass constants between dependent calibrations.

To prevent each detector group from spending time to create their own scripts and macros to solve these same problems, a common Python calibration framework (CAF) has been written. It has been designed to run from a basf2 Python script, and to present user friendly options for common operations. An example of the simplest possible starting code is given below.

```
# import statements
...
alg = TestAlgorithm()
cal = Calibration(name = "Test1",
                  collector = "TestCollectorName",
                  algorithms = [alg],
                  input_files = ["/path/to/test1.root",])
cal_framework = CAF()
cal_framework.add_calibration(cal)
cal_framework.run()
```

In this case, the input data corresponds to a particular IoV and the framework creates calibration constants valid for the entire IoV (if possible). The collector module and algorithm are grouped together into a *Calibration* object, which is added to the *CAF* object. The CAF class controls the overall setup of the environment and monitors the calibrations as they run. The CAF uses a configurable *Backend* object to run the collector basf2 jobs in different ways. By default the CAF runs the collector in a local subprocess allowing for multiprocessing of different calibrations.

As shown below, the CAF, Backend, and Calibration objects can be combined in different ways. In this case multiple calibrations are set up, where one depends on the constants from another. Additionally the CAF Backend is set to use the PBS (qsub) batch system for collector job submission. This creates the option to massively parallelize the collector stage over large datasets.

```
# Assume basic set up
...
cal_a.pre_collector_path = my_path # Can run any basf2 modules prior to collector
cal_b.depends_on(cal_a) # cal_a MUST complete before cal_b starts
cal_framework = CAF()
cal_framework.add_calibration(cal_a)
cal_framework.add_calibration(cal_b)
cal_framework.backend = PBS() # Configurable if needed
cal_framework.run()
```

A brief overview of the CAF is shown in Figure 2. Each calibration is added to an explicit state machine object, which contains the logic of when a state can transition, and which callback functions to execute when it does. The CAF class creates a thread for each requested calibration,

creates the state machine, and then attempts to progress the state forwards until an end point is reached. Calibrations cannot move from their initial state until all calibrations that they depend on have succeeded. By moving to a simple but explicit state machine framework, the program logic became much easier to both understand and develop.

An approximation of the flow diagram for each of the calibration state machines is shown in Figure 3. Note that when an algorithm calls for iteration the collector step and any reconstruction must be re-run with the new intermediate constants on the same data. Also, the CAF never uploads the final database payloads to the central database. Once the final iteration finishes they are instead saved locally, ready for an expert or an external validation job to check for consistency before upload.

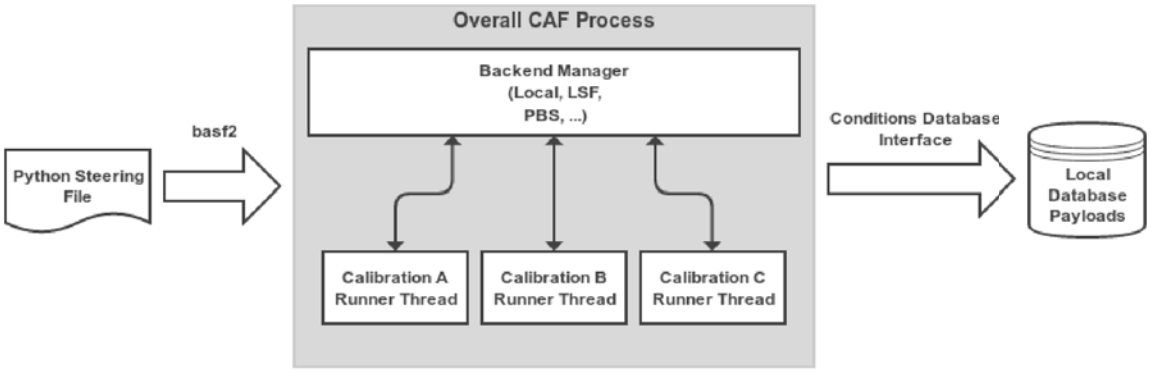

Figure 2. A simplified overview of the CAF. A Python steering file is used to describe which calibrations should be run (e.g. A, B, C) and their configuration. The basf2 CAF process creates a thread to monitor and run each requested calibration through to completion. The threads communicate with the overall framework and a configurable interface to various job submission backends. Once all calibrations return their success, local database payloads are created that can be later uploaded to the conditions database.

5. Millepede alignment example

An integral part of calibrating the Belle II detector will be to accurately calculate the alignment of detector sensors with respect to each other. Alignment constants are used to correct the simulated geometry, used in reconstruction, to match the real positions and rotations of the physical detector during the run.

At Belle II the current track-based alignment uses the General Broken Lines (GBL) method and Millepede II program [6, 7]. They have already been successfully incorporated into the CAF and tested on simulated e^+e^-collision and cosmic ray events. Figure 4 shows the distribution of the difference between the true and reconstructed Δt. Where Δt is the difference between the decay times of the two B mesons from $\Upsilon(4S)$ decays. The distribution is shown for events reconstructed in a range of deliberately misaligned detector geometries. The idealized geometry (black) occurs when no misalignment is present in the simulation. The misaligned distributions are the results when the alignment constants of the sensor positions and rotations were deliberately made worse according to a Gaussian spread. The increasing width of the distribution shows the potential negative impact on the precision of physics analysis quantities, if the vertex detectors are not aligned correctly.

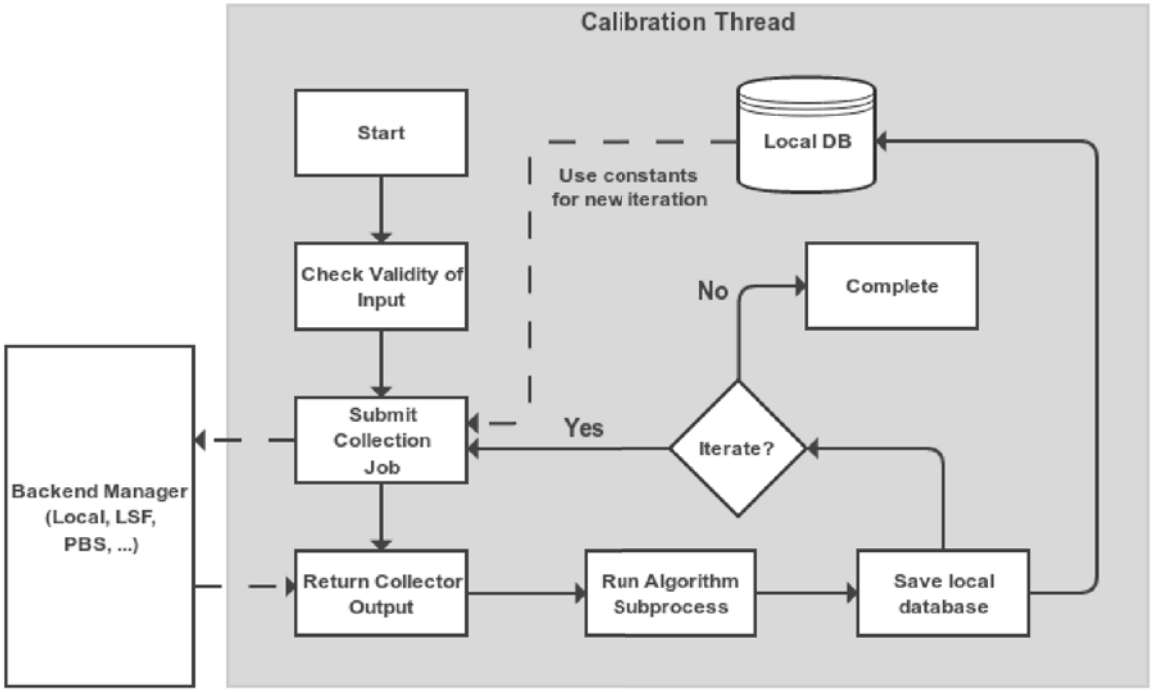

Figure 3. Simple overview of a calibration state machine logic being run in a thread. The collector stage is sent to a submission backend, while the algorithm stage runs in a subprocess. Calibrations can iterate using the saved constants if requested.

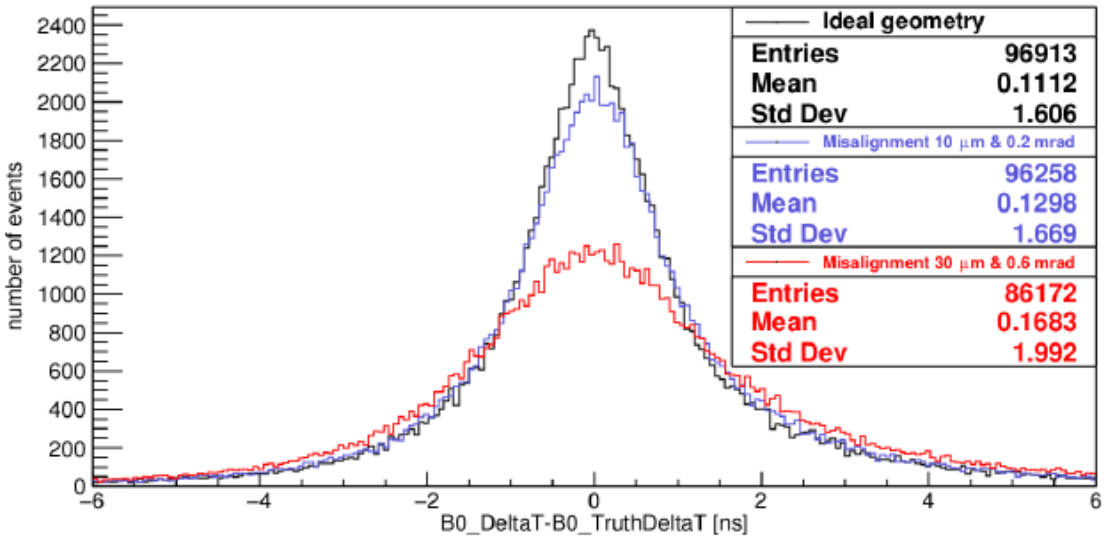

Figure 4. Plot of the difference between the true and measured $B^0\Delta t$ distributions from MC data. Each line corresponds to reconstruction being perform in increasingly misaligned vertex detector geometries. The distribution for the idealized geometry (black) shows the best possible resolution. Deliberate misalignments of $10\mu m$ (blue) and $30\mu m$ (red) show the potential for decreasing performance of the reconstruction with misalignment.

Figure 4 shows that the distribution after misalignments of the size, $10\mu m$ and $0.2mrad$ (blue line), was almost consistent with the idealized geometry. In current tests of the Millepede algorithm implementation at Belle II, the systematic uncertainty of the final constants were $< 5\mu m$ and $< 0.06mrad$. This indicates that the vertex detector alignment algorithm can reach the level where the physics impact of residual misalignment is negligible.

6. Conclusions

The Belle II experiment has created a flexible and powerful calibration framework. Leveraging the interface to the Belle II conditions database and software framework, basf2. The CAF has been designed to minimize the boilerplate code written by detector experts and to enforce a standard calibration procedure early in the development process. Several calibrations from different sub-detectors have already been written in the framework successfully, including the PXD, ECL, and VXD alignment. Development of the CAF will continue as more features are requested, and larger scale testing will be proceeding as Belle II gets ready to take data. Looking to the future, the automated running of the CAF on new data will require an external system of job submission and monitoring, and several options are currently being investigated.

References

[1] Abe T et al. 2010 Belle II Technical Design Report *Preprint* arXiv:1011.0352
[2] Abashian A et al. 2000 The Belle Detector *Nucl. Instrum. Meth.* A **479** 117
[3] Moll A 2011 The Software Framework of the Belle II Experiment *J. Phys. Conf. Ser.* **331** 032024
[4] Wood L, Stephan E, Schram M and Elsethagen T 2016 Conditions Database for the Belle II Experiment *also in this issue*
[5] Ritter M, Kuhr T and Stari M for the Belle II Software Group 2016 High Level Interface to Conditions Data at Belle II *also in this issue*
[6] Kleinwort C 2012 General Broken Lines as advanced track fitting method *Nucl. Instrum. Meth.* A **673** 107
[7] Blobel V 2006 Software alignment for tracking detectors *Nucl. Instrum. Meth.* A **566** 5-13