FACHBEREICH 12 INFORMATIK UND MATHEMATIK

INSTITUT FÜR INFORMATIK

Bachelorarbeit

Optimierung der Datenprozessierung bei der Zeitkalibration von Detektordaten des Mini-CBM-Experiments

Sebastian Heinemann

Studiengang: Bachelor Informatik

Matrikelnummer: 6586667

Frankfurt am Main

31.10.2020

Eingereicht bei:

PD Dr. Andreas Redelbach

Abstract

Das Compressed Baryonic Matter-Experiment ist ein Teilchenbeschleuniger-Experiment an der Facility for Antiproton and Ion Research (FAIR) in Darmstadt. Es untersucht die Kollisionen von hochenergetischen Teilchen, um das Verhalten kleinster Teilchen besser verstehen zu können. Dazu werden mithilfe von verschiedenen Detektoren die Flugbahnen der einzelnen Teilchen erfasst und aufgezeichnet, sodass daraus die Kollisionsevents errechnet werden können. Ereignisse, die von einem Detektor erfasst werden, werden mit einem Zeitstempel versehen, sodass eine zeitliche Zuordnung im späteren Analyseprozess möglich wird. Ein Problem dabei ist allerdings, dass die Zeitstempel der Detektoren nicht auf die Nanosekunde genau synchronisiert sind, was die Event-Rekonstruktion erschwert. Um dieses Problem zu beheben, gibt es in der Prozesskette den sogenannten CheckTiming-Algorithmus, der diese Zeitverschiebungen errechnen kann. In der aktuellen Implementierung hat dieser Algorithmus jedoch eine viel zu hohe Laufzeit, als dass er effizient in der Live-Datenauswertung genutzt werden könnte.
Um die langsamsten Elemente, neben dem CheckTiming-Algorithmus im Prozess der Live-Datenauswertung identifizieren zu können, werden in dieser Arbeit zunächst die Laufzeiten des CheckTiming- sowie der Unpacker-Algorithmen analysiert. Die Ergebnisse der Analyse ergeben, dass es einer Optimierung des CheckTiming-Algorithmus bedarf.
Durch die Anpassung der verwendeten Datenstruktur, dem Entfernen von nicht essenziellen Programmteilen sowie dem Verringern der verwendeten Datenmenge kann ein signifikanter Performanzgewinn verzeichnet werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Eines der faszinierendsten und am schwersten zu erforschenden Themengebiete der Physik ist der Aufbau und die Entwicklung unseres Universums. Es herrschen Verhältnisse, die man sich auf der Erde fast nicht vorstellen kann. Nur eines der vielen extremen Beispiele sind Neutronensterne und die Masse, aus der sie bestehen. Es ist bereits bekannt, dass dort durch die großen Dichten „die positiven Protonen und negativen Elektronen zu neutralen Neutronen förmlich zusammengepresst werden“ [1]. Allerdings stellt sich die Frage was mit den Teilchen im Kern des Sterns passiert, wo der Druck am höchsten ist [1].

Das Compressed Baryonic Matter-Experiment (CBM) an der Facility for Antiproton and Ion Research (FAIR) soll Einblicke in das Verhalten von Materie unter extremen Bedingungen geben [2]. Ziel des Projekts ist es, das QCD-Phasendiagramm im Bereich hoher baryonischer Dichten zu erforschen [2]. Mithilfe des Teilchenbeschleunigers SIS100 und dem CBM-Experiment sollen die Bedingungen im inneren eines Neutronensterns simuliert werden [1]. Um diese besonderen Zustände mit möglichst hoher Präzision beobachten und auswerten zu können, können im CBM-Experiment Kern-Kern Kollisionen mit einer Interaktionsrate von bis zu 10 MHz aufgezeichnet werden. Dabei können bis zu 1 TB Daten pro Sekunde entstehen [3]. Datenmengen dieser Größe verarbeiten zu können erfordert nicht nur performante Hardware, sondern auch effiziente Software und Algorithmen.

Bis zur geplanten Fertigstellung des CBM-Projekts im Jahr 2025 gilt es also, die vorhandenen Hard- und Softwareteile bestmöglichst zu optimieren. Die Möglichkeit dazu bietet das bereits 2019 in Betrieb genommene mini CBM (mCBM) Experiment. Durch intensive Testläufe können in dieser vereinfachten Umgebung des CBM-Experiments Probleme behoben und Verbesserungen implementiert werden.

1.1 Problemstellung

Der CheckTiming-Algorithmus ist ein wichtiger Teil im Datenverarbeitungsprozess des mCBM-Experiments. Er dient dazu, ungewollte Verschiebungen in den Zeitstempeln (Time Offsets) der verschiedenen Detektoren zu errechnen. Die Korrektur dieser Time Offsets findet bei der Umwandlung der Rohdaten in C++ Datenstrukturen, also den sogenannten Unpacker-Algorithmen statt. Aufgrund der Annahme, dass die Time Offsets der Detektoren zwischen verschiedenen Runs[1] nicht sonderlich variieren und es Zeitsynchronisationen beim Start eines jeden Runs gibt, wird der CheckTiming-Algorithmus bisher nur selten eingesetzt. Stattdessen werden bei früheren Runs errechnete Time Offsets übernommen und genutzt. Durch Funktionsstörungen in den Detektoren oder Probleme beim Synchronisierungsprozess kann es allerdings vorkommen, dass Schwankungen in den Time Offsets auftreten [3]. Da der CheckTiming-Algorithmus eine zu hohe Laufzeit hat, damit er in der Echtzeit-Datenverarbeitung eingesetzt werden kann, ist das Ziel dieser Arbeit den Algorithmus soweit zu optimieren, dass er effizient zur Berechnung der Time Offsets genutzt werden kann.

Um die Time Offsets für einen Run zu berechnen, reicht es grundsätzlich aus, den CheckTiming-Algorithmus auf ein entpacktes Timeslice[2]-Archive-File[3] (TSA-File)anzuwenden. Trotzdem kann es zu Laufzeiten von bis zu einer Stunde kommen.

Eine Rolle dabei spielt hierbei die Hardware. Die compute nodes des First-level Event Selectors (FLES) auf denen momentan die entsprechenden Berechnungen ausgeführt werden, sind veraltet und werden erst mit dem Bau des CBM-Projekts erneuert.

Allerdings nicht nur die Hardware ist optimierungsfähig, sondern auch die Software. Ein Verbesserungsansatz ist es, den CheckTiming-Algorithmus nicht über ein gesamtes entpacktes TSA-File laufen zu lassen, sondern nur über einen Bruchteil der Daten. Dies ist möglich, da bereits durch eine geringe Datenmenge der korrekte Time Offset berechnet werden kann. Wichtig ist dabei allerdings, dass die Qualität des Ergebnisses prüfbar aufrechterhalten werden kann. Weiterhin werden im Rahmen dieser Arbeit die im Algorithmus verwendeten Datenstrukturen untersucht und optimiert.

[1] Ein Run ist eine etwa 10-15 minütige Datennahme im aktiven Experimentbetrieb.
[2] Ein Timeslice enthält die Rohdaten aus verschiedenen Detektoren zu einem bestimmten Zeitpunkt.
[3] Timeslice-Archive-Files werden als Datenformat verwendet und kann bis zu 400 Timeslices enthalten. Es werden bis zu 25 dieser Files in einem Run erzeugt.

2 Grundlagen

Das CBM-Experiment von FAIR ist ein sich im Bau befindliches Teilchenbeschleuniger-Experiment auf dem Gelände des GSI Helmholtzzentrum für Schwerionenforschung in Darmstadt. Neben dem neuen Synchrotron SIS100 sollen bis zum Jahr 2025 25 Bauwerke für 4 verschiedene Forschungsgebiete entstehen, wie in *Abbildung 2.1* zu sehen ist. SIS100 wird an den bereits existierenden Ringbeschleuniger SIS18 anschließen und diesen erweitern [4]. Somit sollen Strahlen aus schweren Goldionen mit einer Energie von bis zu 11 AGeV erzeugt werden können [2].

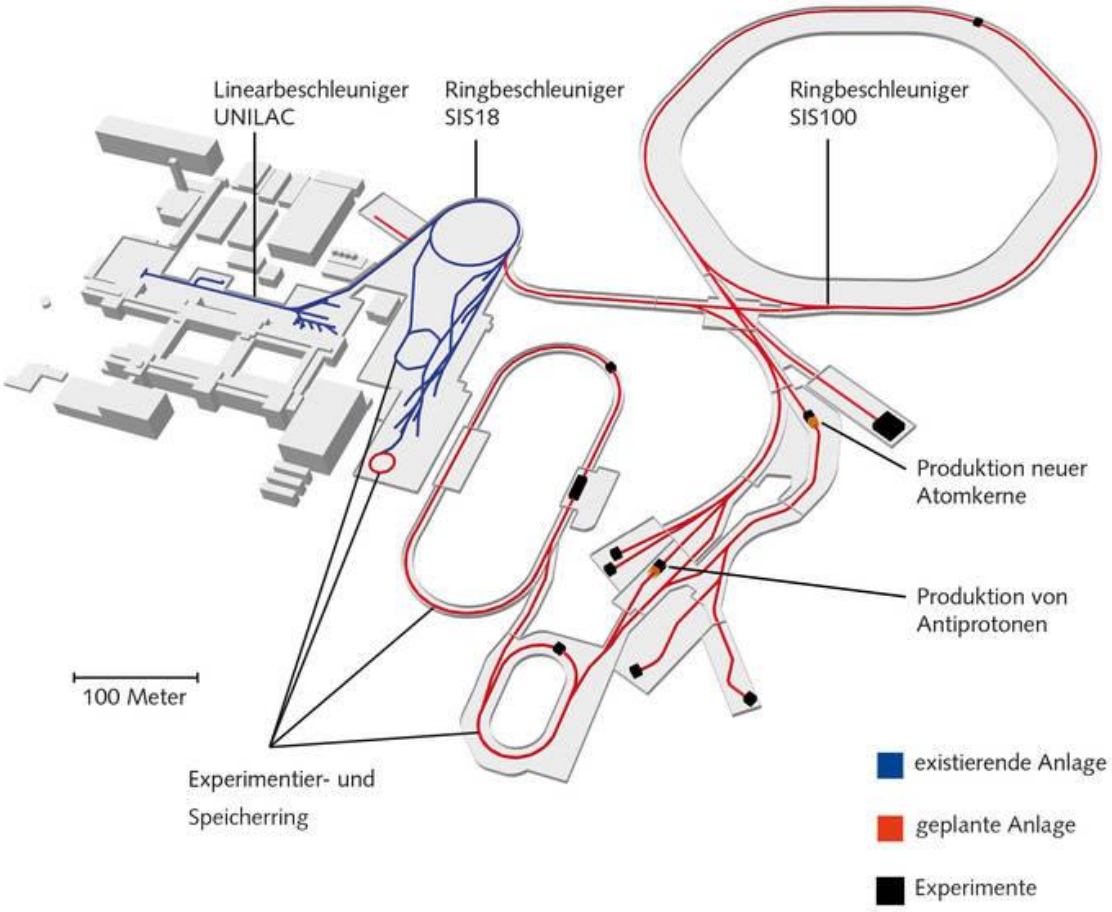


Abbildung 2.1 Übersicht der geplanten FAIR Anlage bei GSI [5].

2.1 mini Compressed Baryonic Matter-Experiment

Das mCBM-Experiment, welches als Prototyp zum Testen der verschiedenen Systemteile des CBM-Experiments entworfen wurde, ist an den SIS18 angeschlossen. Um die gewünschten Teilchenkollisionen für das mCBM zu erzeugen wird der Ionenstrahl des SIS18 auf ein festes Target geschossen [3]. In einem Winkel von 25° dazu befinden sich die aktuell sechs verschiedenen Detektor-Subsysteme mSTS, mMUCH, mTRD, mTOF, mRICH und mPSD. Im späteren Verlauf des Projekts soll noch ein siebter Detektor (mMVD) hinzugefügt werden. Der Testaufbau befindet sich in der mCBM-Cave und ist in *Abbildung 2.2* zu sehen, mSTS ist der Detektor am nächsten zum Target, welches auf dem Bild nicht zu sehen ist. Zusätzlich zu den sechs Detektoren befindet sich etwa 20cm vor dem Target noch der „Time Zero Counter“ (mT0), welcher als Vergleichswert in der Berechnung der Time Offsets genutzt wird.

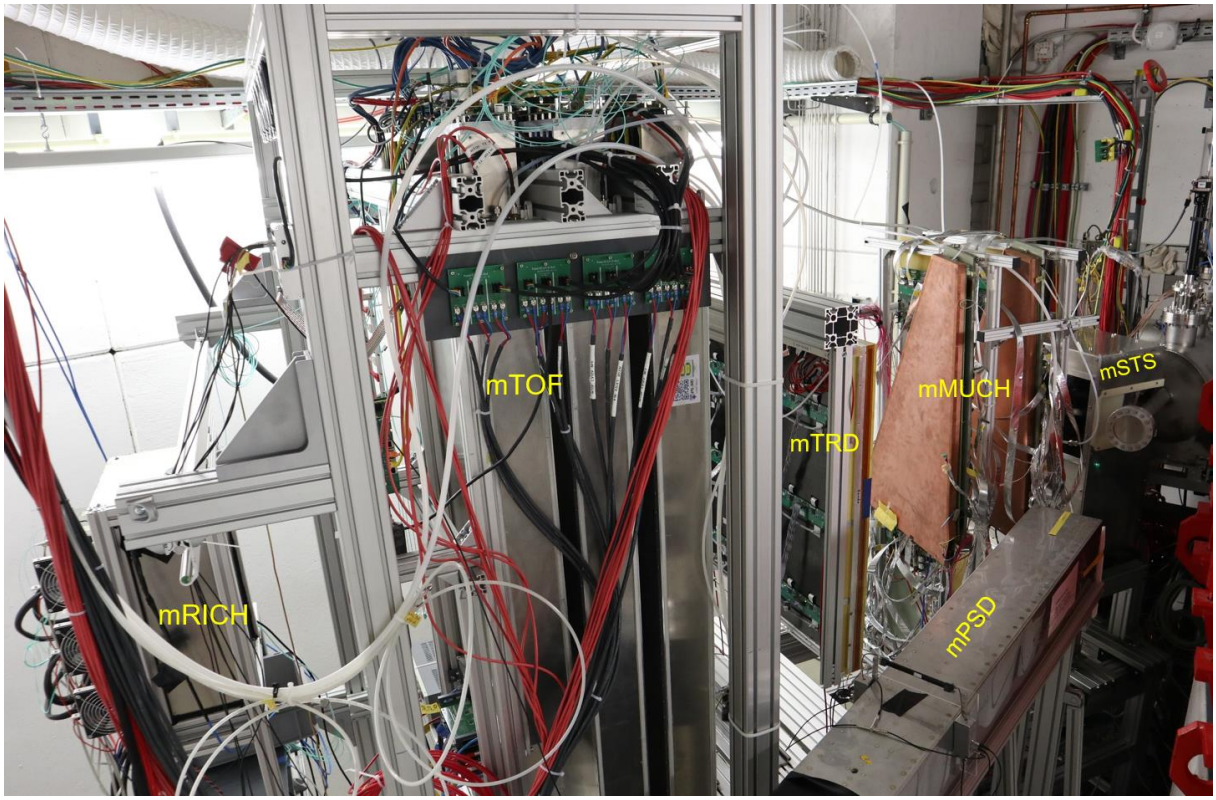


Abbildung 2.2 mCBM Aufbau in der mCBM-Cave [3].

Die von den Detektoren erfassten Daten werden über Gigabit Transceiver aus der mCBM-Cave in den Data Acquisition Container zu den Data Processing Boards (DPBs) weitergeleitet. Von dort aus geht es per optical Link weiter zum Green IT Cube, wo sie schließlich vom FLES input node und den FLES compute nodes verarbeitet werden. Der Prozessierungsweg der Rohdaten aus den Detektoren ist in *Abbildung 2.3* zu sehen.

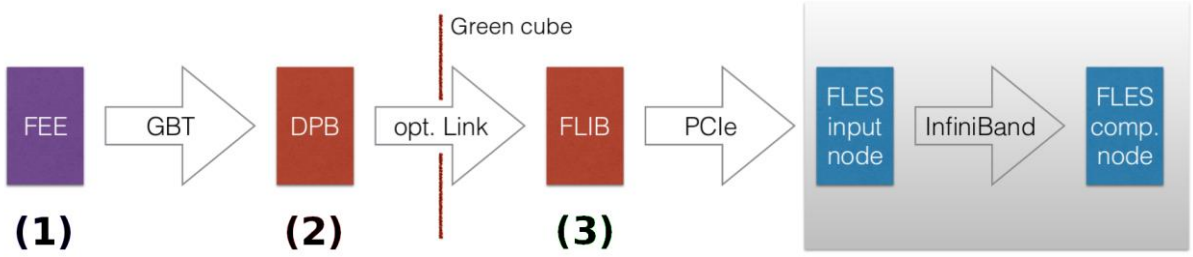


Abbildung 2.3 Datenfluss der Detektordaten [3].

2.2 First-level Event Selector (FLES)

Das CBM-Experiment besitzt im Gegensatz zu anderen Experimenten ähnlicher Art keinen Auslöser zum Auslesen der erzeugten Daten. Stattdessen geben die Detektoren einen kontinuierlichen Datenstream aus, der alle aufgezeichneten Daten enthält. Dadurch entstehen Datenraten von bis zu 1 TB/s [6].

Für die Bewältigung dieser Datenmengen ist der FLES zuständig. Er ist der letzte Knoten der CBM Datenauslesekette, seine Aufgabe ist es, informationstechnisch wertvolle Events zu selektieren und abzuspeichern, indem eine online-Analyse[4] der Detektordaten durchgeführt wird [7].

Die Eingangsdaten des FLES liegen in Form von Microslices vor, die von den subsystemspezifischen DPBs generiert werden. Ein Microslice ist eine global definierte Struktur, die für alle verschiedenen Detektoren gleich aufgebaut ist. Sie enthält die von den Detektoren erzeugten Daten zusammen mit dem zugehörigen Zeitstempel [7]. Eine der zwei wichtigsten Eigenschaften der Microslices ist, dass sie in sich abgeschlossen sind, also grundsätzlich keine Abhängigkeiten zu anderen Microslices besitzen. Dadurch können die Daten zur Verarbeitung im FLES auf die FLES compute nodes aufgeteilt werden, um so die Arbeitslast zu verteilen. Die andere Eigenschaft ist, dass sie immer ein festes Zeitintervall erfassen, welches sich im µs Bereich bewegt. Das bedeutet, dass ihre Größe von der Anzahl der Events im erfassten Zeitraum abhängt [7].

Mehrere Microslices werden zu einem Timeslice (TS) zusammengefasst, wobei ein Timeslice Microslices von verschiedenen Detektoren enthalten kann. Timeslices wiederum werden zur Speicherung in TSA-Files, mit der Dateiendung *.tsa* vereinigt.

2.3 Offline Datenverarbeitung

Die vollständige Analyse der Daten erfolgt im mCBM-Experiment aktuell offline, da der gesamte Prozess, insbesondere die einzelnen Algorithmen, zu viel Zeit in Anspruch nimmt, dass er online durchgeführt werden könnte. Mit dem Voranschreiten der Optimierungen an den

[4] Die online-Analyse ist eine Echtzeit-Analyse, die sofort mit Empfang der Daten durchgeführt wird.

verschiedenen Algorithmen ist allerdings in naher Zukunft geplant, dass die vollständige Analyse der Daten ebenfalls online vom FLES durchgeführt werden kann.

Die offline-Analyse knüpft in der Prozesskette unmittelbar an das Abspeichern der Daten im *.tsa* Format durch den FLES an. Sie beginnt mit der Umwandlung der Rohdaten aus den TSA-Files in C++ Datenstrukturen, die sogenannten Digis. Dieser Schritt wird von den Unpacker-Algorithmen (s. Kap. 2.4) durchgeführt. Bei dieser ersten Ausführung der Unpacker können die Time Offsets der Digis (s. ebenfalls Kap. 2.4) nur korrigiert werden, wenn diese bereits in einem vorherigen Run verlässlich berechnet werden konnten. Liegen keine errechneten Time Offsets vor, so können diese im ersten Lauf der Unpacker auch nicht korrigiert werden. In diesem Fall wird im zweiten Schritt der CheckTiming-Algorithmus mit den Daten aus den Unpackern ausgeführt (s. Kap. 2.5). Dieser ist für die Errechnung der fehlenden Time Offsets zuständig. Sobald die Ergebnisse vorliegen, werden die Unpacker erneut mit den errechneten Time Offsets laufen gelassen.

Mit den kalibrierten Daten werden durch das Makro *build_events.C* mithilfe von Track-Finding Algorithmen die Kollisionsevents rekonstruiert. Anschließend können auf Grundlage der gefundenen Events genauere Analysen durchgeführt werden [8].

2.4 Unpacker-Algorithmen

Die Unpacker-Algorithmen werden dafür genutzt, die in TSA-Files zusammengefassten Detektor-Rohdaten in C++ Datenstrukturen umzuwandeln und zu speichern. Pro Detektor Subsystem gibt es einen Unpacker-Algorithmus, da jeder Detektor die Daten in unterschiedlichem Format ausgibt. Außerdem gibt es das Makro *unpack_tsa_mcbm.C*, das die verschiedenen Unpacker-Algorithmen aufruft, sodass zum entpacken der Daten nicht alle Unpacker einzeln ausgeführt werden müssen.

Die Unpacker-Algorithmen basieren auf dem CbmRoot Framework, welches auf dem FairRoot Framework basiert, welches speziell für FAIR Experimente entwickelt wurde. Das FairRoot Framework ist ein objektorientiertes Simulations-, Rekonstruktions- und Datenanalyseframework [9]. Es basiert wiederum auf dem vom CERN entwickelten Root Framework, was speziell für den Einsatz in physikalischen Experimenten und somit dem Umgang mit großen Datenmengen ausgelegt ist.

Als Eingangsdaten für die Unpacker dienen die TSA-Files, die wiederum mehrere Timeslices enthalten. Aus den Timeslice Rohdaten werden CbmDigi (Digi) C++ Objekte erzeugt, wobei jedes Digi als genau ein von einem Detektor erfasstes Ereignis zu einer bestimmten Zeit an einem bestimmten Ort zu verstehen ist. Ein Digi Objekt enthält also einen Zeitstempel, die Adresse, an welcher Stelle im Detektor das Ereignis registriert wurde, und die Intensität der

Messung. Da die Eigenschaften der Digis sich von Subsystem zu Subsystem unterscheiden gibt es für jeden Detektor eine eigene Form der Digis. Entsprechend gibt es die Klassen *CbmStsDigi*, *CbmMuchDigi*, *CbmTrdDigi*, *CbmTofDigi*, *CbmRichDigi*, *CbmPsdDigi*, *CbmT0Digi*. Im Folgenden wird „Digi“ als Verallgemeinerung der verschiedenen Digi-Arten genutzt.

Wie bereits in Kap. 1.1 erläutert, müssen die Zeitstempel aus den Timeslices um die Ungenauigkeiten in der Zeitmessung korrigiert werden. Dafür werden die mittels CheckTiming errechneten Time Offsets händisch in das Unpacker-Makro *unpack_tsa_mcbm.C* eingetragen (s. Anh. *Abbildung 7.1*). Die aus den Timeslices ausgelesenen Zeitstempel können somit angepasst werden, bevor die Umwandlung in Digi-Objekte erfolgt.

Die Ausgangsdaten des Unpacker-Algorithmus werden in Form eines *.root* Files gespeichert. Es beinhaltet die verarbeiteten Timeslices mit den jeweils zugehörigen Digis.

Abbildung 2.4 zeigt die durchschnittlichen Datenraten aller Detektoren pro Sekunde für Run 831. Run 831 lief für etwa 10 Minuten, was eine Gesamtdatenmenge von ~100 GB ergibt, die von den Unpacker-Algorithmen verarbeitet werden müssen. Aufgrund des immensen Datendurchsatzes der Unpacker ist es wichtig, dass die einzelnen Algorithmen so gut wie möglich optimiert sind. Teil dieser Arbeit ist deshalb eine Laufzeitanalyse der verschiedenen Unpacker-Algorithmen (s. Kap. 3.1).

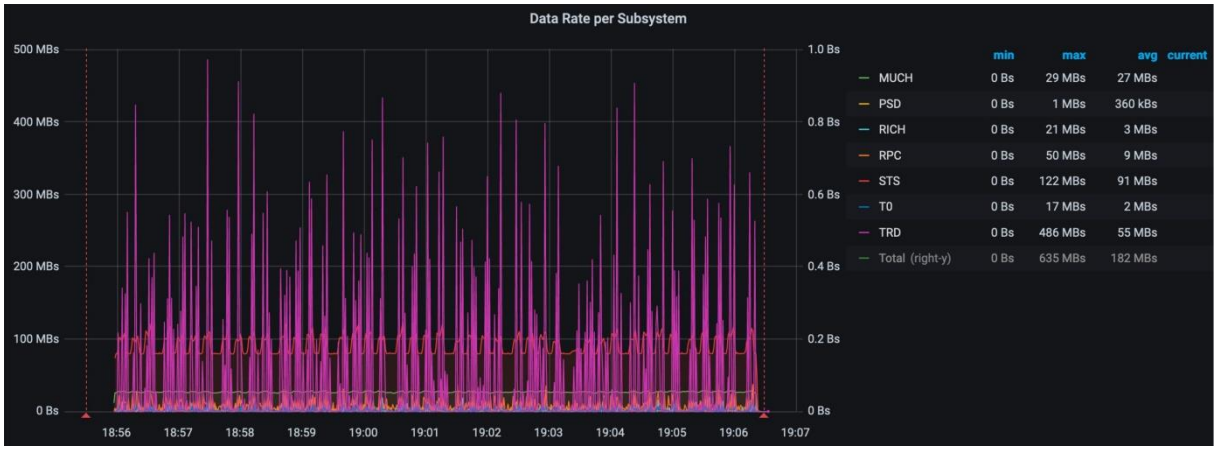


Abbildung 2.4 Datenraten pro Subsystem Run 831 FLES Übersicht

2.5 CheckTiming-Algorithmus

Wie die Unpacker-Algorithmen basiert auch der CheckTiming-Algorithmus auf dem CbmRoot Framework. Er besteht aus der Klasse *CbmCheckTiming* und dem C++ Makro *check_timing.C*, welches für die Initialisierung des FairRoot Frameworks und der *CbmCheckTiming* Klasse genutzt wird. Der Algorithmus wird dazu genutzt, die Time Offsets zwischen den Detektoren zu berechnen (s. Kap. 2.4).

Als Eingangsdaten werden Files des *.root* Typs genutzt, die in diesem Fall Daten enthalten, die bereits von den Unpackern prozessiert wurden. Es werden also nicht die rohen Detektordaten, sondern Timeslices mit den entsprechenden Digis verarbeitet. Standardmäßig nutzt der Algorithmus alle im *.root* File enthaltenen Timeslices zur Berechnung der Time Offsets. Über einen Funktionsparameter im *check_timing.C* Makro ist es allerdings möglich die Anzahl der Timeslices zu beschränken, um die Menge der zu verarbeitenden Daten verringern zu können.

Aufgrund des FairRoot Frameworks ist die *CbmCheckTiming* Klasse in drei grundlegende Funktionen unterteilt. Dabei handelt es sich um die Funktionen *CbmCheckTiming::Init*, *CbmCheckTiming::Exec* und *CbmCheckTiming::Finish*.

Die *Init*-Funktion dient dazu, zu prüfen, ob alle benötigten Daten vorhanden sind, und alle verwendeten Histogramme zu initialisieren.

Die *Exec*-Funktion dient zur Ausführung der Berechnungen. Die Berechnung der Time Offsets findet in der Funktion *CbmCheckTiming::CheckInterSystemOffset* statt, welche durch *Exec* aufgerufen wird. Die grundlegende Funktionsweise von *CheckInterSystemOffset* ist es, den Zeitstempel jedes *CbmT0Digi* mit den Zeitstempeln jedes anderen Digis in einer bestimmten Offset Range[5] zu vergleichen und die Zeitdifferenz der betrachteten Digis in ein entsprechendes Histogramm einzutragen. Jedes Subsystem hat eine eigene Offset Range, die zwischen 500 ns für die *CbmTrdDigi*s und für 50000 ns *CbmMuchDigi*s variiert. Es werden also beispielsweise die Zeitdifferenzen zwischen einem *CbmT0Digi* und allen *CbmTrdDigis* vermerkt, deren Zeitstempel ±500 ns Unterschied zum *CbmT0Digi* haben. Implementiert ist diese Funktionsweise über zwei for-Schleifen. Die erste iteriert über alle *CbmT0Digis* und die zweite über alle verschiedenen Digis innerhalb ihrer respektiven Offset Ranges.

Neben der *CbmCheckTiming::CheckInterSystemOffset* Funktion ruft die *Exec*-Funktion noch die *CbmCheckTiming::CheckTimeOrder*-Funktion auf. Alle Digis eines Subsystems sind zeitlich sortiert in einem *std::vector* gespeichert. *CheckTimeOrder* untersucht die Korrektheit der zeitlichen Reihenfolge der einzelnen Digis in den Vectoren und verzeichnet alle Fehlstände die dabei auftreten in Histogrammen.

Die Ergebnisse der oben genannten Berechnungen werden im *Finish* Teil des Programms in Form von eindimensionalen (1D) Histogrammen abgespeichert und ausgegeben. Die Time Offsets der Subsysteme können anhand von Peaks in den entsprechenden Histogrammen erkannt werden. Ein Zusammenschnitt der subsystemspezifischen Peaks ist in *Abbildung 2.5*

[5] Die Offset Range ist eine für jedes Subsystem festgelegte Zeitspanne, in der zusammengehörige Subsystem und T0 Digis vermutet werden können

zu sehen. Aus verschiedenen Gründen[6] kann es vorkommen, dass die Berechnung des Time Offsets eines Detektors fehlschlägt, was sich dadurch äußert, dass im entsprechenden Histogramm kein Peak zu erkennen ist. Das Erkennen der Korrektheit der Peaks sowie das Auslesen der Ergebnisse ist zurzeit nur visuell über den Graphen des Histogramms möglich.

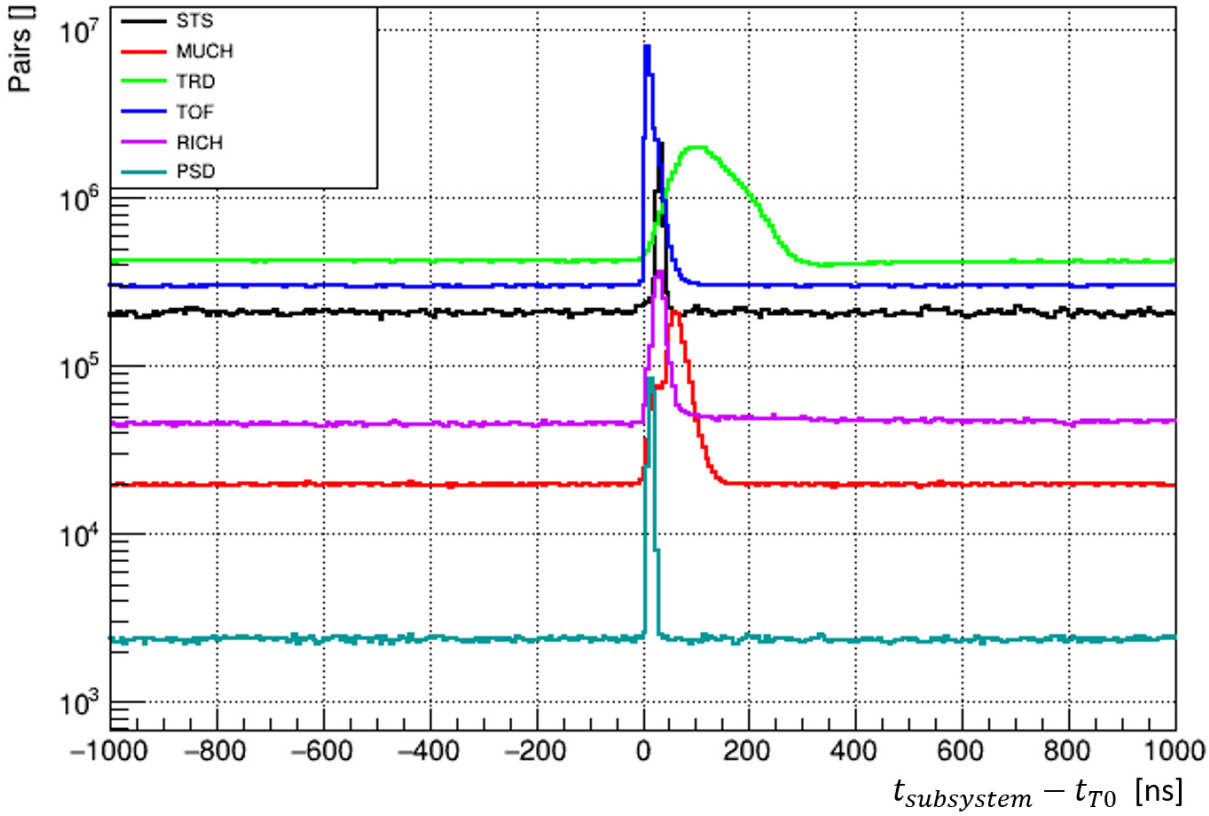


Abbildung 2.5 Darstellung aller (bereits korrigierten) subsystemspezifischen Peaks für den relativen Zeitversatz [3]

Die Peaks der verschiedenen Subsysteme haben teilweise sehr unterschiedliche Formen. Dies liegt daran, dass die Subsysteme zum Teil unterschiedliche Zeitauflösungen haben. Je höher die Zeitauflösung eines Detektors ist, desto schmaler ist ein Peak in der Histogrammdarstellung. Der in *Abbildung 2.5* abgebildete Peak für den Offset des STS-Detektors ist sehr hoch und schmal, mit einer Breite von etwa 75 ns. Die Zeitauflösung des TRD-Subsystems hingegen liegt im Bereich von etwa 300 ns. Dementsprechend ist der Peak im TRD Histogramm flacher und breiter.

Neben den sechs 1D-Histogrammen zum Ablesen der Time Offsets erzeugt der CheckTiming-Algorithmus noch über 50 weitere 1D- und 2D-Histogramme, wobei der Anteil der 2D-Histogramme überwiegt. Sie bieten hauptsächlich spezielle Informationen, die zur Fehlerfindung bei Problemen genutzt werden können. Entsprechend sind diese Histogramme nicht essenziell für die Bestimmung der Time Offsets.

[6] Z.B. zu wenig, bzw. fehlerhafte Input Daten usw.

2.6 CBM Datenraten

In diesem Kapitel werden die voraussichtlichen Datenraten des CBM-Experiments erläutert, da diese in späteren Abschnitten als Referenz genutzt werden.

Das CBM-Experiment soll nach seiner Fertigstellung im Jahr 2025 mit drei verschiedenen Detektorkonfigurationen betrieben werden können:

Die „Hadron Konfiguration“ besteht aus den Detektoren STS, TRD, TOF und PSD. Sie ist speziell darauf ausgelegt, das Verhalten von Hadronen genauer zu beobachten.

Die „Elektron Konfiguration“ besteht aus den Detektoren STS, TRD, TOF, PSD, MVD und RICH. Mit dieser Zusammenstellung der Detektoren soll nicht nur das Verhalten der Hadronen, sondern auch das von Elektronpaaren beobachtet werden.

Die „Myon Konfiguration“ besteht aus den Detektoren STS, MUCH, TRD und TOF. Sie ist ausschließlich dazu da, das Verhalten von Myonpaaren zu beobachten.

Um die Datenraten des CBM-Experiments herauszufinden wurden im Jahr 2018 Au+Au Kollisionen mit einem Impuls von 12 AGeV/c simuliert. Durch die Simulation kann die erwartete Anzahl von aufgezeichneten Treffern (Messages) an jedem Subsystem nachgebildet werden. *Tabelle 2.1* zeigt die dokumentierten Ergebnisse für alle verschiedenen Konfigurationen [10].

Um den Datenanteil der einzelnen Subsysteme nicht nur konfigurationsspezifisch, sondern für das Gesamtexperiment zu zeigen, wird die Summe aller Messages eines Subsystems durch die Summe der Gesamtmessages aller Konfigurationen geteilt. Dies ermöglicht einen Vergleich der Datenmengen zwischen CBM und mCBM in Kap. 3.1.

Subsystem	Hadron Konf. Messages	Elektron Konf. Messages	Myon Konf. Messages	Anteil am ges. Experiment
STS	5395	4779	5668	58,72%
MUCH	0	0	926	03,43%
TOF	670	1079	126	06,95%
RICH	0	425	0	01,58%
PSD	0	403	0	01,49%
TRD	1810	2487	55	16,13%
MVD	0	3156	0	11,70%
Gesamt	7875	12329	6775	26979

Tabelle 2.1 Simulierte Trefferraten des CBM-Experiments [10]

Die Daten zeigen, dass der STS-Detektor voraussichtlich mit einem Gesamtanteil von 58,72% die meisten Daten erzeugen wird. An zweiter und dritter Stelle folgen der TRD mit 16,13% und der MVD mit 11,7%.

3 Laufzeitanalysen

Um herauszufinden, welcher Teil der Software den dringendsten Verbesserungsbedarf hat, werden Laufzeitanalysen für die verschiedenen Unpacker sowie den CheckTiming-Algorithmus im mCBM-Experiment durchgeführt.

3.1 Unpacker Laufzeitanalyse

Um die Laufzeiten der einzelnen Unpacker-Algorithmen betrachten zu können, muss das in Kap. 2.4 beschriebene Makro *unpack_tsa_mcbm.C* abgeändert werden. Es wird aufgeteilt, sodass anschließend ein Makro pro Subsystem (STS, MUCH, TOF, TRD, RICH, PSD) existiert, welches nur den jeweiligen Unpacker-Algorithmus ausführt.

Die Laufzeitanalyse wird anhand von vier verschiedenen, zufällig ausgewählten Runs durchgeführt, damit die Ergebnisse nicht von Unregelmäßigkeiten in einem bestimmten Run beeinträchtigt werden. Zusätzlich zu den vier zufällig ausgewählten Runs 812, 827, 854 und 903 wird als fünftes noch Run 831 hinzugezogen, da dieser bereits als Musterbeispiel in der „Application for beam time at GSI/FAIR“ [3] genutzt wurde. Von jedem Run werden nicht alle zugehörigen TSA-Files genutzt, sondern auch diese werden zufällig, stichprobenartig ausgewählt, sodass insgesamt 21 Files untersucht werden. Die Analyse wird auf dem process node[7] pn06 durchgeführt. Die Laufzeiten werden auf ganze Sekunden gerundet, da bei der Wiederholung der Tests mit gleichen Daten Abweichungen im Bereich der Nachkommastellen auftreten.

Für jedes TSA-File wird die Anzahl der verwendeten Timeslices und Digis sowie die benötigte Laufzeit (CPU Time) des Programms betrachtet. *Tabelle 3.1* zeigt eine Zusammenfassung der aufgezeichneten Daten aus allen 21 TSA-Files.

Die in diesem Test erfassten Daten von RICH und PSD sind aufgrund der niedrigen aufgezeichneten Datenmengen als insignifikant zu betrachten. Die Laufzeiten dieser Unpacker-Algorithmen waren so gering, dass ein Großteil der Laufzeit dem Overhead durch das Framework zuzuordnen ist.

Es werden also nur die Unpacker von STS, MUCH, TOF und TRD, wie in *Abbildung 3.1* dargestellt, näher betrachtet.

[7] Als process node werden die FLES compute nodes bezeichnet.

Detektor	TS	CPU Time (s)	Digis	CPU Time (µs) / Digis	Digis / CPU Time (s)	Gesamtanteil Daten
STS	6152	1146	2141340688	0,54	1867524	78,30%
MUCH	6152	226	236292572	0,96	1045449	8,64%
TOF	6152	189	150304369	1,25	794504	5,50%
RICH	6152	36	545139	65,20	15321	0,02%
PSD	6152	20	1872432	10,60	94139	0,07%
TRD	6152	453	204560118	2,21	450870	7,48%
Gesamt	6152	2070	2734915318	0,76	1320583	-

Tabelle 3.1 Laufzeiten der verschiedenen Unpacker-Algorithmen auf pn06

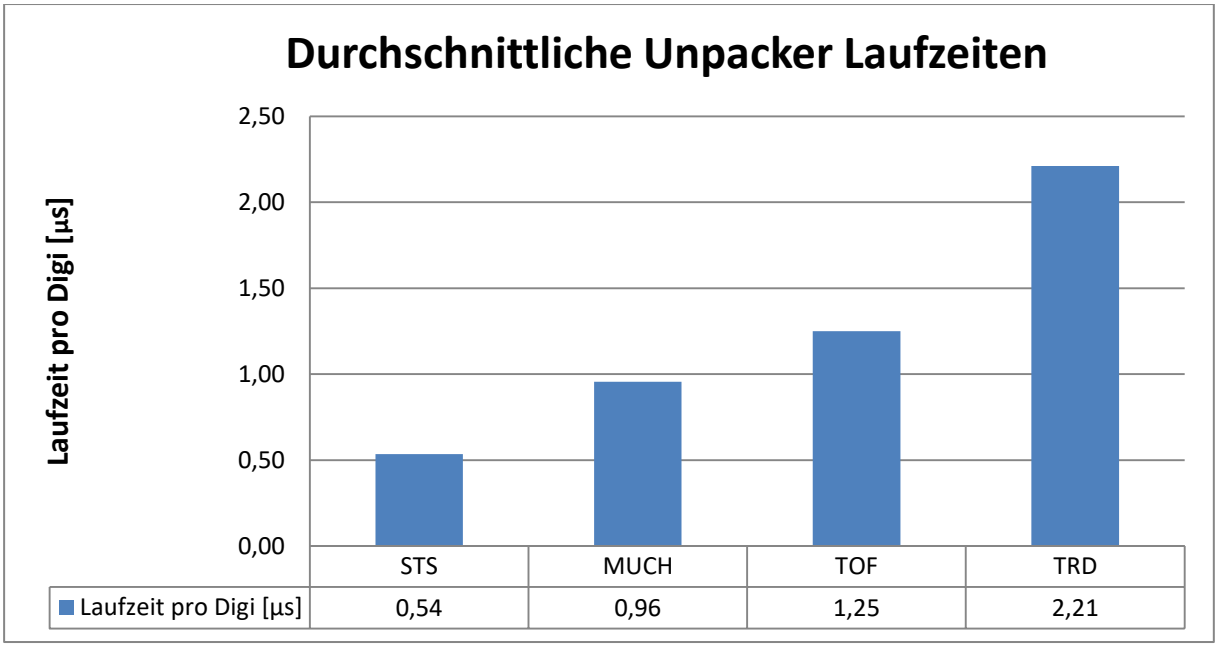


Abbildung 3.1 Laufzeitverhältnis der vier wichtigsten Unpacker auf pn06

Dabei fällt auf, dass die Laufzeit des STS, gemessen in Laufzeit pro Digi, deutlich schneller ist, als die der anderen Unpacker. Der MUCH-Unpacker ist um etwa einen Faktor 2 langsamer, der TRD-Unpacker sogar um etwa einen Faktor 4. Auch der TOF-Unpacker ist um einen Faktor 2,3 langsamer als der STS-Unpacker, wurde aber bereits in der Bachelorarbeit von Tim Geier [11] verbessert. Allerdings ist das Update des TOF-Unpackers zum Stand dieses Tests noch nicht implementiert worden.

Um herausfinden zu können, welcher Unpacker-Algorithmus den größten Verbesserungsbedarf hat, ist es sinnvoll, die Anteile der einzelnen Unpacker an der Gesamtdatenmenge in Betracht zu ziehen. Dazu werden die Datenraten dieser Laufzeitanalyse aus dem mCBM-Experiment mit den geschätzten Datenraten für das CBM-Experiments aus Kap. 2.6 verglichen.

Da in den in *Tabelle 2.1* beschriebenen Daten der MVD mit einbezogen wurde, dieser aber im mCBM-Experiment noch nicht vorhanden ist, sind Abweichungen in den Datenanteilen der anderen Subsysteme zu erwarten.

Für das CBM-Experiment wird erwartet, dass ein Großteil (58,72%) der Daten vom STS-Detektor erzeugt wird. Dies spiegelt sich auch in den aufgezeichneten Daten des mCBM-Experiments wieder: Dort ist der Anteil des Detektors mit 78,3% um etwa 20% höher, als zu erwarten wäre.

Das einzige andere Subsystem, das im mCBM-Experiment mehr Daten produziert, als für das CBM-Experiment vorhergesagt wird, ist der MUCH-Detektor. Der Anteil ist etwa 2,5-mal so hoch als geplant. Diese Abweichungen liegen vermutlich daran, dass die Daten in Tabelle 2.1 über alle drei Konfigurationen gemittelt sind, der MUCH-Detektor im mCBM-Experiment aber dauerhaft in Verwendung ist.

Die Subsysteme RICH und PSD werden nach *Tabelle 2.1* die geringsten Datenmengen, mit jeweils etwa 1,5% im CBM-Experiment erzeugen. Auch diese Verhältnisse konnten grob in der Laufzeitanalyse beobachtet werden. Nach *Tabelle 3.1* beträgt der Anteil des RICH-Detektors jedoch nur etwa ein Hundertstel und der des PSD etwa ein Fünfzigstel der erwarteten Mengen. Diese Beobachtung unterstreicht noch einmal, dass die Laufzeiten dieser beiden Subsysteme nicht repräsentativ genutzt werden sollten.

Die Anteile des TOF-Detektor an der im mCBM-Experiment erzeugten Datenmenge liegen mit 5,5%, unter Beachtung des Fehlens des MVD Subsystems und der daraus folgenden Abweichungen, sehr nah an den Erwartungen für das CBM-Experiment.

Mit 16,13% soll das TRD-Subsystem den zweithöchsten Anteil der Daten, nach dem STS, im CBM-Experiment liefern. In der Laufzeitanalyse im mCBM-Experiment erzeugt das System mit 7,48% allerdings sogar weniger Daten als das MUCH System, welches im CBM-Experiment für nur etwa 3% der Daten verantwortlich sein soll.

Unter Berücksichtigung der Verhältnisse zwischen den gemessenen und erwarteten Datenanteilen der Subsysteme werden die beobachteten Laufzeiten der Unpacker-Algorithmen noch einmal betrachtet, um zu erschließen, welcher Algorithmus den größten Verbesserungsbedarf hat.

Der STS-Unpacker hat sowohl den höchsten Anteil an den erzeugten Daten als auch die beste Geschwindigkeit unter den getesteten Algorithmen. Der MUCH-Unpacker ist um einen Faktor 2 langsamer als der des STS, soll aber im CBM-Experiment vergleichsweise nur 6% der Daten zu verarbeiten. Im Gegensatz dazu ist der TRD-Unpacker um einen Faktor 4 langsamer

als der STS-Unpacker, muss aber etwa 27,5% der Daten des STS, bzw. 16,13% der Gesamtdaten verarbeiten.

Demnach sollte bei einer weiteren Optimierung der Unpacker-Algorithmen zunächst der TRD-Unpacker verbessert werden.

3.2 CheckTiming Laufzeitanalyse

Im Zuge der Unpacker Laufzeitanalyse wird zu den ungepackten Daten jedes TSA-Files auch das CheckTiming-Makro laufen gelassen, damit die Vollständigkeit der Daten stichprobenartig mithilfe der Histogramme überprüft werden kann. Dabei fällt auf, dass der CheckTiming-Algorithmus eine immense Laufzeit hat, die mit der Anzahl der verarbeiteten *CbmT0Digis* skaliert. Wie bereits in Kap. 2.5 erklärt, kommt der Zusammenhang zwischen Laufzeit und Anzahl der T0 Digis daher, dass der Algorithmus eine for-Schleife durchläuft, die über alle T0 Digis iteriert. *Abbildung 3.2* zeigt die Laufzeit des CheckTiming-Algorithmus von verschiedenen TSA-Files aus den Runs 812, 831 und 854. Die Tests werden aus Zeitgründen nicht auf dem process node pn06, sondern auf dem login node[8] durchgeführt. Vergleichsberechnungen, bei denen CheckTiming mit den gleichen Daten auf den verschiedenen Servern ausgeführt wird, zeigen einen 2,7- bis 3-fachen Speedup des login nodes gegenüber pn06. Das bedeutet, dass die in *Abbildung 3.2* dargestellten Laufzeiten unter realen Bedingungen in der offline-Datenaufbereitung des mCBM-Experiments etwa dreimal so hoch sind.

Die in *Abbildung 3.2* stark ansteigende Anzahl der T0 Digis in den TSA-Files ist dadurch zu erklären, dass Runs 812 und 831 eine durchschnittliche Kollisionsrate von etwa 2 kHz hatten, Run 854 hingegen eine von etwa 100 kHz [12], wodurch deutlich mehr T0 Digis erzeugt wurden.

Die Schwankungen im Bereich bis etwa 1.200 T0 Digis sind dadurch zu erklären, dass die Laufzeit des Algorithmus nicht ausschließlich von der Anzahl der T0 Digis abhängt. Wie bereits in Kapitel 2.5 beschrieben spielen die Anzahl der anderen Digis und die Offset Ranges der Subsysteme eine weitere Rolle. Dies bedeutet also, dass die Tests mit relativ niedrigen Laufzeiten in diesem Bereich vergleichsweise wenige Detektor Hits, also auch wenige Subsystem Digis verarbeiten und die Tests mit vergleichsweise hoher Laufzeit entsprechend viele.

[8] Der login node ist der Server am GSI, über den eine SSH Verbindung zu den process nodes aufgebaut werden kann.

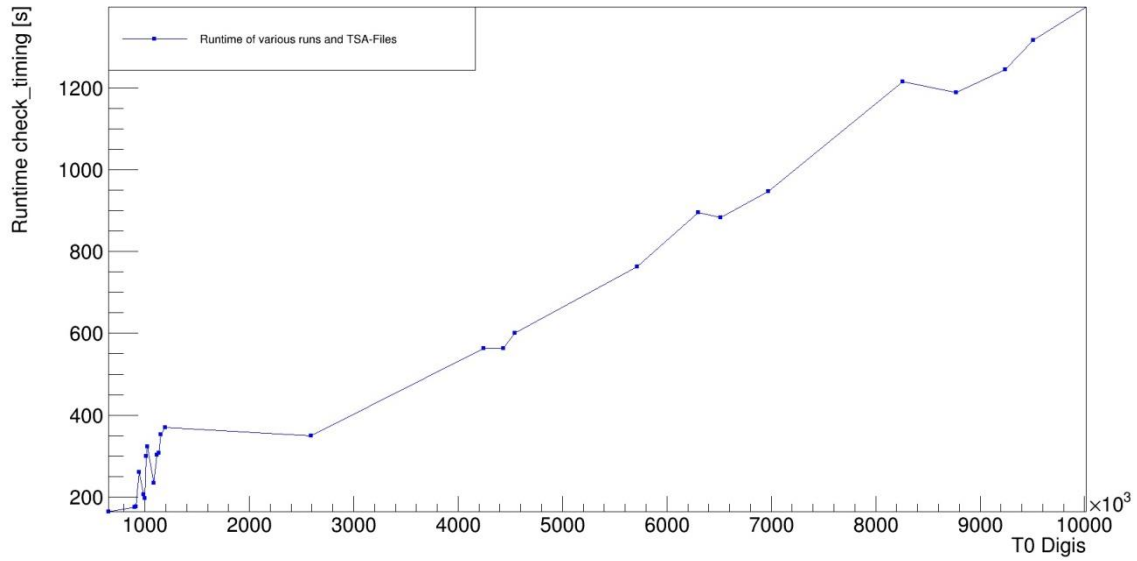


Abbildung 3.2 CheckTiming Laufzeiten verschiedener Runs in Abhängigkeit der Anzahl der T0 Digis, gemessen auf login node

Diese Laufzeitanalyse zeigt, dass die aktuelle Laufzeit zur effizienten Nutzung des Algorithmus, mit bis zu über einer Stunde auf den FLES compute nodes, ungenügend ist.

3.2.1 Funktionslaufzeiten

Um herauszufinden, wie der CheckTiming-Algorithmus am besten optimiert werden kann, werden die Laufzeiten der wichtigsten einzelnen Funktionen näher betrachtet. Jede in *Tabelle 3.2* aufgeführt Funktion wird dafür um eine eigene *TStopwatch* ergänzt, welche die beanspruchte CPU Zeit misst. Die Testläufe werden auf dem FLES compute node pn06 ausgeführt, um möglichst praxisnahe Ergebnisse zu erzielen. Run 856 hat, wie Run 854, ebenfalls eine Kollisionsrate von 100 kHz und unterscheidet sich damit in der erzeugten Anzahl der T0 Digis deutlich von Run 831. Dem entsprechend eignen sich diese Runs gut, um die Laufzeiten, insbesondere in Bezug auf die verschiedenen T0 Zahlen, genauer zu betrachten. Die Laufzeiten aus Run 831 ergeben sich durch die Verwendung des gesamten ersten TSA-Files, für Run 856 hingegen ist es aus technischen Gründen nicht möglich, das gesamte TSA-File zu nutzen. Deshalb wird der Test mit nur 30 von etwa 50 Timeslices für Run 856 durchgeführt.

Bei der Wiederholung der Tests treten Laufzeitschwankungen im niedrigen, einstelligen Sekundenbereich auf, die ausschließlich auf die *CheckInterSystemOffset* Funktion zurückzuführen sind. Aus diesem Grund sind auch diese Ergebnisse auf volle Sekunden gerundet.

Funktionsname	Laufzeit (s) für Run 831 TSA-File 0000, 384 TS, 1.022.038 T0 Digis	Laufzeit (s)für Run 856 TSA-File 0000, 30 TS, 2.358.246 T0 Digis
Init()	4	4
CheckTimeOrder()	28	10
CheckInterSystemOffset()	952	1571
Finish()	35	34
Zusammen	1019	1620
Gemessene Gesamtlaufzeit	1037	1627
Overhead	18	9

Tabelle 3.2 Laufzeitmessung der wichtigsten Funktionen der *CbmCheckTiming* Klasse auf pn06

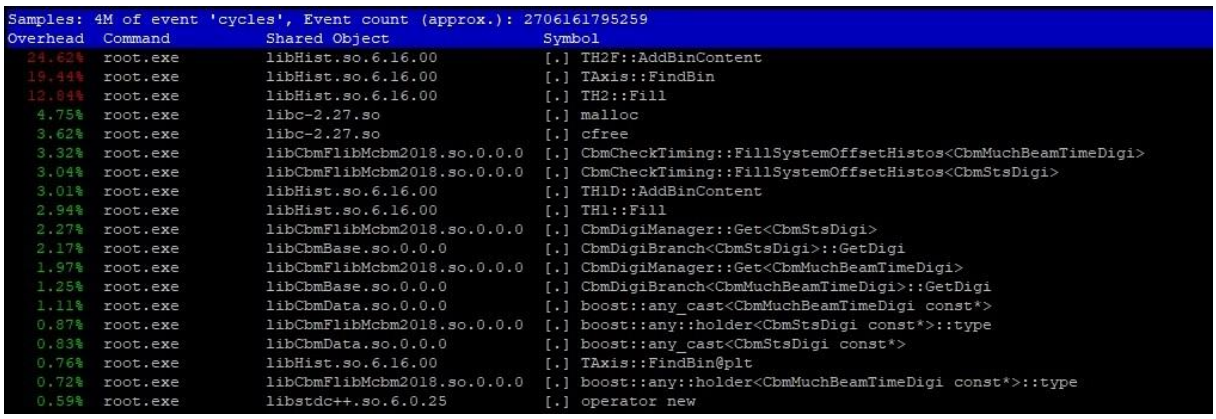
Die Ergebnisse zeigen, dass die Funktion *CheckInterSystemOffset* mit einem Anteil von über 90% in beiden Tests den größten Teil der Laufzeit erzeugt. Dies ist auch der Programmteil, welcher mit der Anzahl von T0 Digis skaliert. Die Funktionen *Init* und *Finish* haben konstante Laufzeiten, die sich auch in anderen Tests, wie z.B. ein Ausführen mit nur einem Timeslice, gleich verhalten. Schwankungen im Bereich der Nachkommastellen sind bei diesen Funktionen normal und werden daher vernachlässigt. Auffällig in der Gegenüberstellung von *Tabelle 3.2* ist, dass die *CheckTimeOrder*-Funktion in Run 856 eine niedrigere Laufzeit hat, als in Run 831, obwohl die Gesamtlaufzeit des Algorithmus dort höher ist. Eine Vermutung ist, dass die Laufzeit dieser Funktion nicht von der Anzahl der T0 Digis abhängt, sondern von der Anzahl der verwendeten Timeslices. Obwohl die Laufzeit aller essenziellen Funktionen der CheckTiming Klasse dokumentiert wurde gibt es Differenzen zwischen der zusammengerechneten und der vom *check_timing.C* gemessenen Gesamtlaufzeit. Diese werden hier als Overhead bezeichnet und sind ebenfalls abhängig von der Anzahl der verwendeten Timeslices. Der Overhead stammt nicht direkt aus der *CbmCheckTiming* Klasse, sondern aus dem FairRoot Framework, welches z.B. die Daten der Timeslices aus dem *.root* File entgegen nimmt und an die nachfolgenden Instanzen der *CbmCheckTiming* Klasse übergibt.

3.2.2 Erweiterte Laufzeitanalyse mit perf

Das Linux Performance Tool „perf“ ist ein in den Linux-Kernel integriertes Tool zur Analyse der Performanz eines Programms. Durch die Möglichkeiten des Profilings und Tracings, die perf bietet, ist es möglich, die Laufzeit des CheckTiming-Algorithmus näher zu betrachten und herauszufinden wie diese optimiert werden kann.

Dazu wird zunächst auf dem FLES compute node pn06 der Befehl *perf record root -l –b './check_timing.C("/scratch/heinemann/my_CbmRoot_Aug20/cbmroot/Makro/be amtime/mcbm2020/data/run_831/0000/unp_mcbm_831.root", 831)'* ausgeführt. Dies führt dazu, dass der CheckTiming-Algorithmus, der mit dem ersten TSA-File von Run 831 ausgeführt wird, von perf aufgezeichnet wird. Die Aufzeichnungen werden in Form einer *.data* Datei gespeichert und enthalten Informationen darüber, welche Funktionen wie viel Laufzeit in

Anspruch nehmen. Durch den Befehl *perf report* lassen sich diese Informationen, wie in *Abbildung 3.3* gezeigt, einsehen.



```
Samples: 4M of event 'cycles', Event count (approx.): 2706161795259
Overhead  Command          Shared Object                  Symbol
  24.62%  root.exe         libHist.so.6.16.00             [.] TH2F::AddBinContent
  19.44%  root.exe         libHist.so.6.16.00             [.] TAxis::FindBin
  12.84%  root.exe         libHist.so.6.16.00             [.] TH2::Fill
   4.75%  root.exe         libc-2.27.so                   [.] malloc
   3.62%  root.exe         libc-2.27.so                   [.] cfree
   3.32%  root.exe         libCbmFlibMcbm2018.so.0.0.0    [.] CbmCheckTiming::FillSystemOffsetHistos<CbmMuchBeamTimeDigi>
   3.04%  root.exe         libCbmFlibMcbm2018.so.0.0.0    [.] CbmCheckTiming::FillSystemOffsetHistos<CbmStsDigi>
   3.01%  root.exe         libHist.so.6.16.00             [.] TH1D::AddBinContent
   2.94%  root.exe         libHist.so.6.16.00             [.] TH1::Fill
   2.27%  root.exe         libCbmFlibMcbm2018.so.0.0.0    [.] CbmDigiManager::Get<CbmStsDigi>
   2.17%  root.exe         libCbmBase.so.0.0.0            [.] CbmDigiBranch<CbmStsDigi>::GetDigi
   1.97%  root.exe         libCbmFlibMcbm2018.so.0.0.0    [.] CbmDigiManager::Get<CbmMuchBeamTimeDigi>
   1.25%  root.exe         libCbmBase.so.0.0.0            [.] CbmDigiBranch<CbmMuchBeamTimeDigi>::GetDigi
   1.11%  root.exe         libCbmData.so.0.0.0            [.] boost::any_cast<CbmMuchBeamTimeDigi const*>
   0.87%  root.exe         libCbmFlibMcbm2018.so.0.0.0    [.] boost::any::holder<CbmStsDigi const*>::type
   0.83%  root.exe         libCbmData.so.0.0.0            [.] boost::any_cast<CbmStsDigi const*>
   0.76%  root.exe         libHist.so.6.16.00             [.] TAxis::FindBin@plt
   0.72%  root.exe         libCbmFlibMcbm2018.so.0.0.0    [.] boost::any::holder<CbmMuchBeamTimeDigi const*>::type
   0.59%  root.exe         libstdc++.so.6.0.25            [.] operator new
```

Abbildung 3.3 Ausgabe des Befehls *perf report* für Run 831 TSA-File 0 mit dem originalen Algorithmus

Die Informationen zeigen, dass die drei Funktionen, die den meisten Arbeitsaufwand erzeugen, zusammen für 56,9% der Laufzeit verantwortlich sind. Weiterhin lässt sich erkennen, dass alle drei Funktionen Teil von Histogrammklassen sind. Diese drei Funktionen *TH2F::AddBinContent*, *TAxis::FindBin* und *TH2::Fill* sind für das Füllen von Histogrammen mit Daten zuständig. Auffällig dabei ist, dass die Funktionen *AddBinContent* und *Fill* aus der Klasse *TH2* stammen, also ausschließlich zum Füllen von 2D-Histogrammen genutzt werden. Die Funktion *FindBin* hingegen ist Teil der TAxis Klasse und kann somit beim Füllen von 1D- und 2D-Histogrammen genutzt werden. Die Nutzung der 2D-Histogramme erzeugt allein durch die beiden Funktionen *AddBinContent* und *Fill* einen Anteil an der Gesamtlaufzeit von über 37%.

Histogramme sind in der Regel in viele kleine, gleich große Wertebereiche (Bins) eingeteilt. Ein Bin fungiert als Zähler, der die Summe der Anzahl der Werte enthält, die in das zugehörige Intervall fallen. Im Verlauf dieser Arbeit wird insbesondere die Bin Breite näher betrachtet, welche die Größe des Wertebereichs vorgibt.

Über die *annotate* Funktion im *perf report* ist es möglich, die Assemblerbefehle zu den entsprechend aufgeführten Funktionen anzeigen zu lassen.

So zeigt Abbildung 3.4 die Instruktionen, die in der Funktion *TAxis::FindBin* den größten Arbeitsaufwand erzeugen. Daraus geht hervor, dass in dieser Funktion die Berechnung, in welchen Bin ein Wert eingetragen werden soll, sehr aufwändig ist.

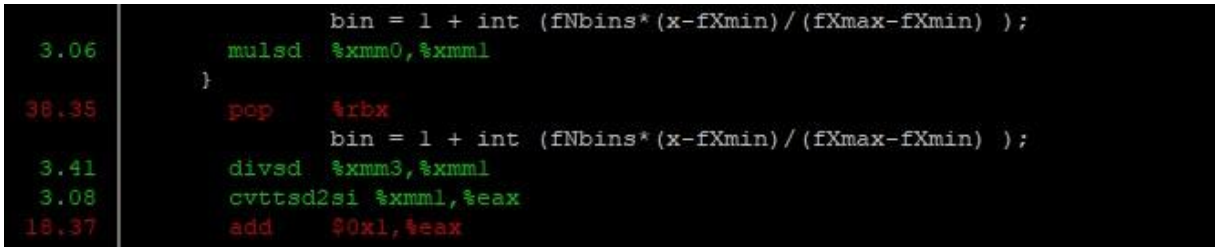


Abbildung 3.4 Assemblerbefehle zur Funktion *TAxis::FindBin*

Insgesamt wird klar, dass ein Großteil der entstehenden Laufzeit der Datenstruktur der Histogramme, insbesondere aber den zweidimensionalen Histogrammen zuzuordnen ist.

Aufgrund dieser Beobachtungen wird, wie in Kap. 4.1 erklärt, zunächst die Nutzung aller zweidimensionalen Histogramme aus dem Algorithmus entfernt. Anschließend wird erneut ein *perf report* für Run 831 TSA-File 0 erstellt, um zu sehen, welche Anteile die übrigen Instruktionen an der Gesamtlaufzeit haben. Die Ergebnisse sind in *Abbildung 3.5* zu sehen und zeigen, dass noch immer Histogramm-Funktionen eine große Rolle spielen. Zusammen erzeugen die Histogramm-Funktionen *TAxis::FindBin*, *TH1::Fill*, *TH1D::AddBinContent* und *TH1F::AddBinContent* 26,84% der Gesamtlaufzeit.

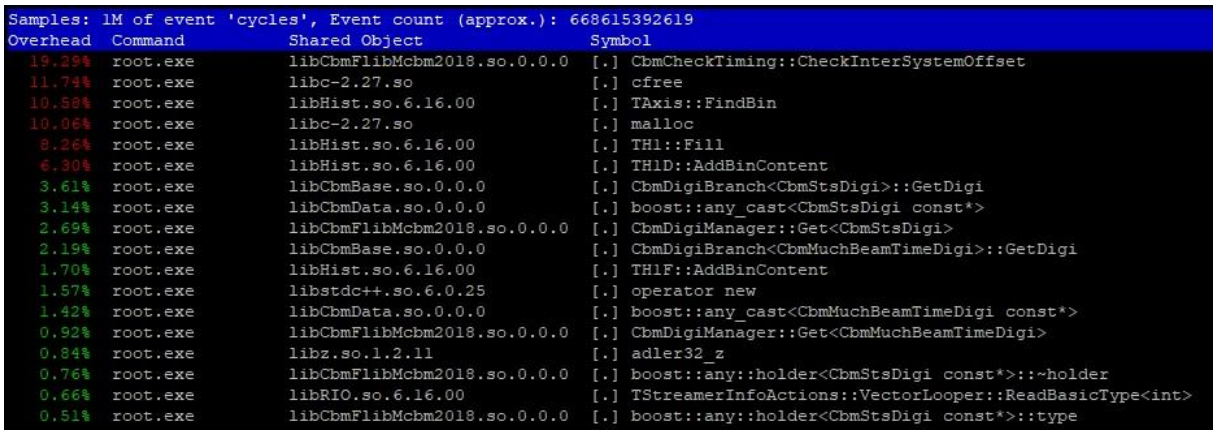


Abbildung 3.5 *perf report* von Run 831 TSA-File 0 ohne Nutzung von 2D-Histogrammen im Algorithmus

4 Optimierung des CheckTiming-Algorithmus

Die Optimierung des CheckTiming-Algorithmus basiert auf drei grundlegenden Ideen. *Abbildung 3.2* zeigt, dass die Laufzeit des Algorithmus proportional zur verwendeten Anzahl von T0 Digis ansteigt, was eine Einschränkung der verwendeten Datenmenge essenziell macht. Zudem wird die in Kapitel *3.2.2* diskutierte Verwendung der Histogramm-Datenstruktur geändert, indem sie durch die Datenstruktur *std::vector* ersetzt wird. Außerdem werden nicht essenzielle Programmteile, wie die oben beschriebenen Histogramme zur Fehlersuche, entfernt.

4.1 Entfernen der nicht essenziellen Histogramme

Wie in Kapitel 2.5 beschrieben, werden während der Berechnung der Time Offsets viele Histogramme mit unterschiedlichen Informationen erstellt, die hauptsächlich der Problembehebung bei Fehlern in den Subsystemen dienen. Da es in dieser Arbeit darum geht, die Berechnung der Time Offsets so effizient wie möglich zu gestalten, wird die Erstellung dieser Histogramme aus dem Code entfernt. Dies betrifft alle Histogramme, außer *fT0StsDiff*, *fT0MuchDiff*, *fT0TrdDiff*, *fT0TofDiff*, *fT0RichDiff* und *fT0PsdDiff*, die alleinig die Daten der Time Offsets enthalten. Da auf diese Weise nur noch eindimensionale Histogramme der TH1D Klasse zurückbehalten werden, fallen alle 2D-Histogramme weg, die für die in Kap. 3.2.2 besprochenen hohen Laufzeitanteile verantwortlich waren. Diese Änderung hat zur Folge, dass von den über 50 erzeugten Histogrammen des Algorithmus noch 6 weiterhin genutzt werden.

Durch das Entfernen der Histogramme wird auch die Funktion *CheckTimeOrder* überflüssig, da die Funktion ausschließlich für das Füllen von nicht essenziellen Histogrammen zuständig ist. Dementsprechend wird die Funktion vollständig entfernt.

4.2 Änderung der genutzten Datenstruktur

In der ursprünglichen Implementierung des CheckTiming-Algorithmus ist die Wahl der Datenstruktur zur Speicherung der Daten auf Histogramme gefallen. Am Ende von Kap. 3.2.2 wird gezeigt, dass die 1D-Histogramm nach der Entfernung der 2D-Histogramme noch immer einen Anteil an der Laufzeit von 26,84% haben.

Da die verbliebenen Histogramme alle eindimensional sind, können diese durch die Datenstruktur *std::vector<Int_t>* ersetzt werden. Ein 1D-Histogramm kann speichertechnisch wie ein Vector betrachtet werden, indem man einen Bin als einen Eintrag eines Vectors sieht. Wird im Fall eines Histogramms also ein Wert einem Bin zugeordnet und dieser erhöht, so wird im Vector ein bestimmter Indexeintrag inkrementiert. Es bietet sich an für die Vectoren eine Bin Breite von 1ns zu wählen. So kann die Größe des Vectors eines Subsystems einfach

durch die zugehörige Offset Range bestimmt werden. Da ein Histogramm mit einer Offset Range ***x*** den Bereich von ***–x*** ns bis ***+x*** ns abdeckt und zusätzlich der Wert der 0 beachtet werden muss, entspricht die Größe eines Vectors $\boldsymbol{x} * \mathbf{2} + \mathbf{1}$. Das Mapping zwischen Index und tatsächlichem Wert ist durch diese Wahl der Bin Breite ebenfalls sehr einfach. Der Index 0 des Vectors entspricht dem Wert ***–x*** ns, der Index ***x*** dem Wert 0 ns und der Index 2***x*** dem Wert ***x***. Durch diese einfache Art des Mappings, wird auch das durch *Abbildung 3.4* beschriebene Problem der teuren Berechnung bei der Bin Findung behoben. Muss der Zähler bei einem Wert ***y*** erhöht werden, ist der zugehörige Index ***i*** im Vector durch die Formel $\boldsymbol{i} = \boldsymbol{y} + \boldsymbol{x}$ zu finden.

Jedes der 6 Subsystem Histogramme wird durch einen Vector ersetzt. Damit die Vectoren global in der Klasse verfügbar sind, werden sie in der Header Datei initialisiert und im *Init* Teil des Programms mit $\boldsymbol{x} * \mathbf{2} + \mathbf{1}$ Einträgen gefüllt, die den Integer Wert Null enthalten.

4.3 Verringerung der genutzten Datenmenge

Wie in Kapitel 2.5 beschrieben besteht die Möglichkeit, die Anzahl der verwendeten Timeslices des genutzten *.root* Files durch einen Funktionsparameter im *check_timing.C* Makro einzuschränken. Sobald der Parameter mit einer Zahl ***k*** verwendet wird, werden nur die ersten ***k*** Timeslices des *.root* Files zur Berechnung genutzt. Ein Problem dabei ist allerdings, dass es Timeslices gibt, die keine T0 Digis enthalten, die zur Berechnung der Time Offsets aber notwendig sind. Dies ist der Fall, wenn ein Timeslice einen Zeitbereich eines Runs abdeckt, in dem eine Spill Pause war. Eine Spill Pause ist ein Zeitabschnitt eines Runs, in dem kein Strahl auf das Target geschossen wird und somit keine Kollisionen erzeugt werden.

In *Abbildung 4.1* ist die Verteilung der T0 Digis im Bezug zur Zeit von Run 831 zu sehen. In den weißen Spalten (z.B. um 100 s) werden keine T0 Digis erzeugt, weil eine Spill Pause anliegt.

Wenn also wenige Timeslices zur Berechnung der Time Offsets verwendet werden und die ersten Timeslices eine Spill Pause enthalten, kann es vorkommen, dass die Berechnung aufgrund zu weniger T0 Digis ungenau wird oder komplett fehlschlägt. Aufgrund der Beschaffenheit des *check_timing.C* Makros und des FairRoot Frameworks ist es nicht möglich, die Anzahl der verwendeten Timeslices bei gegebener Notwendigkeit innerhalb des Programms zu erweitern.

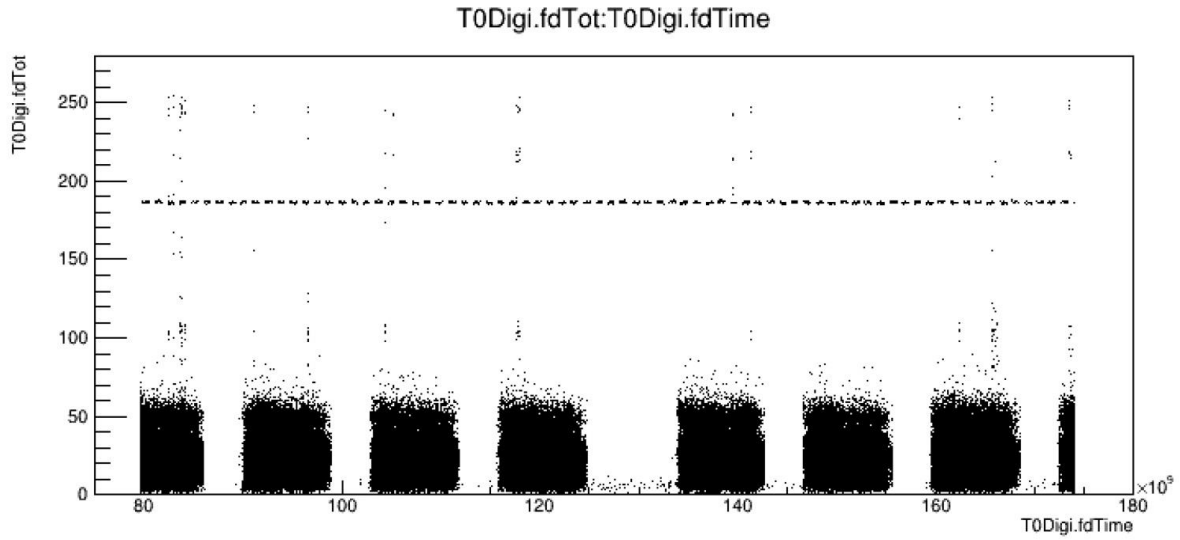


Abbildung 4.1 Zeitabschnitt aus Run 831, der die gemessene Intensität (Tot) der T0 Digis im Verhältnis zur Zeit zeigt

4.3.1 Implementierung

Deshalb wird eine andere Implementierung gewählt. Ebenfalls aufgrund des FairRoot Frameworks, ist die *CbmCheckTiming* Klasse in die drei grundlegenden Funktionen *Init*, *Exec* und *Finish* unterteilt. *Init* wird nur einmal bei der Initialisierung des Programms ausgeführt und *Finish* einmal am Ende. Gesteuert durch das *check_timing.C* Makro wird für jedes verwendete Timeslice einmal die *Exec*-Funktion ausgeführt, welche die Berechnung der Time Offsets enthält. Diese Struktur lässt sich so nutzen, dass in die *Exec*-Funktion eine Abfrage eingebaut werden kann, wie viele T0 Digis bereits insgesamt zur Berechnung verwendet wurden. Ist ein vorgegebener Schwellenwert erreicht, so wird der Inhalt der *Exec*-Funktion für kein weiteres Timeslice mehr ausgeführt. Dafür wird eine if-Abfrage in die *Exec*-Funktion eingebaut, die den globalen Zähler für T0 Digis, *fNrOfT0Digis* mit einem neu implementierten Parameter *fT0DigiLimit* vergleicht (s. Anh. *Abbildung 7.2*). Durch eine Ergänzung im Funktionsaufruf des Makros kann der *fT0DigiLimit* Parameter gesteuert werden(s. Anh. *Abbildung 7.3*). Da diese Implementierung nur vorsieht, dass der Inhalt der *Exec-Funktion* übersprungen wird, führt das FairRoot Framework trotzdem die *Exec*-Funktion für jedes Timeslice des TSA-Files aus. Dies hat zur Folge, dass ein Overhead entsteht, der sich nur durch Änderungen an der Struktur des FairRoot Frameworks verhindern lässt, was im Rahmen dieser Arbeit allerdings nicht möglich ist.

4.3.2 Kontrolle der Korrektheit

Die korrekte Berechnung der Time Offsets benötigt eine Mindestanzahl von T0 und allen anderen Subsystem Digis. Da diese nicht pauschal festgelegt werden kann und sich von Run zu Run unterscheidet, kann es passieren, dass eine Berechnung mit geringer Datenmenge ungenaue, falsche oder gar keine Ergebnisse liefert. Deshalb wird eine zweistufige Qualitäts-

sicherung für die Berechnungen eingeführt, die den Nutzer über die Echtheit des Time Offset-Peaks informiert.

Grundlegend für die Qualitätserkennung ist dabei, dass der Peak automatisch aus den Vectoren ausgelesen werden kann. Dies wird durch die Funktion *CbmCheckTiming::GetPeakIndex* implementiert, die den Index des Vectoreintrags mit dem größten Wert zurückgibt.

4.3.2.1 Erweiterung der Bin Breite

In der ursprünglichen Implementierung des CheckTiming-Algorithmus haben alle Histogramme eine Bin Breite von 6,25 ns, was dazu führt, dass ein Bin die Summe aus 6,25 Werten enthält. Dies hat den Effekt, dass Peaks, deren Breite 6 ns oder größer ist, besser erkennbar sind. Unechte Peaks mit einer geringeren Breite, die durch Zufall entstanden sind, können durch umliegende Werte ausgeglichen werden.

Die Anpassung der verwendeten Datenstruktur von Histogrammen zu Vectoren hat zur Folge, dass die standardmäßige Bin Breite der Vectoren nur noch 1 ns statt 6,25 ns beträgt. Dies führt dazu, dass vermehrt unechte Peaks auftreten, die teilweise sogar höher sein können als der Peak des eigentlichen Time Offsets, was wiederum zu Problemen in der automatischen Erkennung des echten Peaks führt.

Um diesem Problem entgegenzuwirken, wird grundsätzlich die Bin Breite auf 2 ns erhöht, bevor ein Peak ausgelesen wird. Dies reduziert zwar die Wahrscheinlichkeit auf einen zufällig entstandenen Peak, ohne dass große Einbußen bei der Genauigkeit gemacht werden müssen. Da allerdings auch mit einer Bin Breite von 2 ns zufällig entstandene Peaks nicht vollkommen ausgeschlossen werden können, wird die Funktion *CbmCheckTiming::CheckIndexPeakStability* implementiert (s. Anh. *Abbildung 7.7*). In dieser Funktion wird die Bin Breite der Vectoren erneut erweitert, um zufällige Peaks auszugleichen. Aus einem gegebenen Vector mit einer Bin Breite von 1 ns werden zwei neue Vectoren mit einer Bin Breite von 2ns und 8ns erstellt. Das bedeutet, dass ein Eintrag des neuen Vectors die Summe von 2 bzw. 8 Einträgen des originalen Vectors enthält.

Anschließend werden mithilfe der Funktion *CheckIndexPeakStability* die höchsten Peaks der beiden neuen Vectoren ausgelesen und miteinander verglichen. Liegen die Peaks nicht an der gleichen Stelle, so gibt die Funktion aus, dass der Peak des originalen Vectors nicht echt ist. Dabei wird berücksichtigt, dass sich der Peak auch durch die Aufsummierung der einzelnen Einträge verschieben kann.

Die Notwendigkeit, dass ein Peak aus einem Vector mit einer Bin Breite mit 2 ns statt aus einem mit Bin Breite 1 ns ausgelesen wird, kommt daher, dass der Peak des TRD Offsets sehr

flach ist, wenn die Anzahl der verwendeten Digis gering ist. Bei einem Testlauf von Run 831 mit einer Beschränkung von 100.000 T0 Digis verzeichnen die 1 ns breiten Bins des TRD Vectors durchschnittlich Einträge der Höhe 500. Im Bereich des etwa 100 ns breiten Peaks liegt der durchschnittliche Wert bei 600. Sowohl im Bereich des Peaks als auch außerhalb gibt es einzelne Bins, die einen Wert von 800 überschreiten. Beim Auslesen des Peaks bei einer Bin Breite von 1 ns kommt es also schnell vor, dass der Peak an einer Stelle erkannt wird, wo lediglich durch Zufall ein hoher Wert entstanden ist. Wenn die Bin Breite auf 2 ns erweitert wird, korrigieren sich die enthaltenen Werte insgesamt in Richtung des Durchschnitts und es wird weniger wahrscheinlich einen falschen Peak auszulesen.

4.3.2.2 Fitten der Peaks

Durch Fehler in den Eingangsdaten des CheckTiming-Algorithmus oder durch die Berechnung mit zu wenigen Daten kann es passieren, dass zwar Werte in die Vectoren eingetragen werden, der Time Offset-Peak aber nicht vorhanden ist. Dieses Problem lässt sich nicht alleinig durch die Erweiterung der Bin Breite lösen, da sich der höchste Punkt des Vectors ohne das Vorhandensein eines echten Peaks nicht unbedingt verschiebt.

Die Echtheit des errechneten Peaks lässt sich mithilfe von Fitting verifizieren. Wie in *Abbildung 2.5* zu erkennen ist, besteht jedes Histogramm, welches einen echten Peak enthält, aus dem besagten Peak und relativ konstantem Hintergrundrauschen. Daher bietet es sich an, eine Funktion zu fitten, die aus einer Konstanten für das Hintergrundrauschen und einer Gauß-Kurve für den Peak besteht.

Das Root Framework bietet verschiedene Möglichkeiten zum Fitten an. Eine davon wird durch die Histogrammklasse selbst geboten. Sie beinhaltet eine Funktion zum Fitten von eindimensionalen Histogrammen, welche einfach aus den vorhandenen Vectoren erstellt werden können. Um diese Funktion zu nutzen wird zunächst für jedes Subsystem ein Histogramm erstellt, welches mit den Daten aus den Vectoren gefüllt wird. Dies hat keine signifikante Laufzeit, da für das Füllen lediglich einmal durch jeden Vector gelaufen wird und die Inhalte der Vectoreinträge mithilfe der Funktion *TH1::AddBinContent* in die Bins der entsprechenden Histogramme kopiert werden.

Die Funktionen *CbmCheckTiming::gauss* (s. Anh. *Abbildung 7.4*), *CbmCheckTiming::fitFunction* (s. Anh. *Abbildung 7.5*) und *CbmCheckTiming::background* (s. Anh. *Abbildung 7.6*) sind aus verschiedenen Beispiel-Fits auf der Root Webseite entnommen [13, 14], allerdings wurde die background Funktion angepasst, sodass diese nur noch einen Parameter enthält und somit konstant ist. Die Fit-Funktion besteht also insgesamt aus vier Parametern, einer für das Hintergrundrauschen und drei für die Gauß-Kurve. Parameter *p0* bestimmt die Höhe des Hintergrundrauschens, *p1* die Höhe des Peaks über dem Rauschen, *p2*

die Position des höchsten Punkts des Peaks und *p3* die Breite des Peaks. Die Parameter gelten dabei nur für den Fit und können von den realen Werten abweichen. Der Grund dafür ist die insgesamt hohe Anzahl an Datenpunkten und die Tatsache, dass die Fits immer einen Error[9] aufweisen.

Der STS-Detektor hat eine Offset Range von 20.000 ns, was bei einer Bin Breite von 6,25 ns zu insgesamt 6.400 Datenpunkten führt. Der Peak des STS Systems hat eine Breite von etwa 150 ns, also 24 Datenpunkten. Würde man also über alle Daten des Histogramms fitten, würde der Peak etwa $\frac{1}{267}$ der Datenpunkte ausmachen und nicht erkannt werden. Deshalb wird der Fit angepasst, sodass nur der Bereich von -1000 ns bis +1000 ns um den höchsten Punkt im Histogramm gefittet wird. Dadurch hat der Peak einen Anteil von $\frac{3}{40}$ an den gefitteten Datenpunkten und kann im Normalfall erkannt werden.

Abbildung 4.2 zeigt den magenta gefärbten Fit zum STS-Histogramm aus Run 831, TSA-File 0 mit einer Beschränkung von 100.000 T0 Digis. Es ist zu erkennen, dass der Fit nicht ganz akkurat ist. Dies liegt daran, dass der Peak im Vergleich zum Rauschen sehr schmal und hoch ist. Allerdings ist ein Fit dieser Art ausreichend, um die später beschriebenen Qualitätskriterien erfüllen zu können. Zur verbesserten Ansicht wird in *Abbildung 4.2* nur der Ausschnitt des Histogramms gezeigt, der den Fit enthält.

[9] Im Zusammenhang mit Fitting, wird ein Error als eine geschätzte Abweichung vom realen Wert bezeichnet.

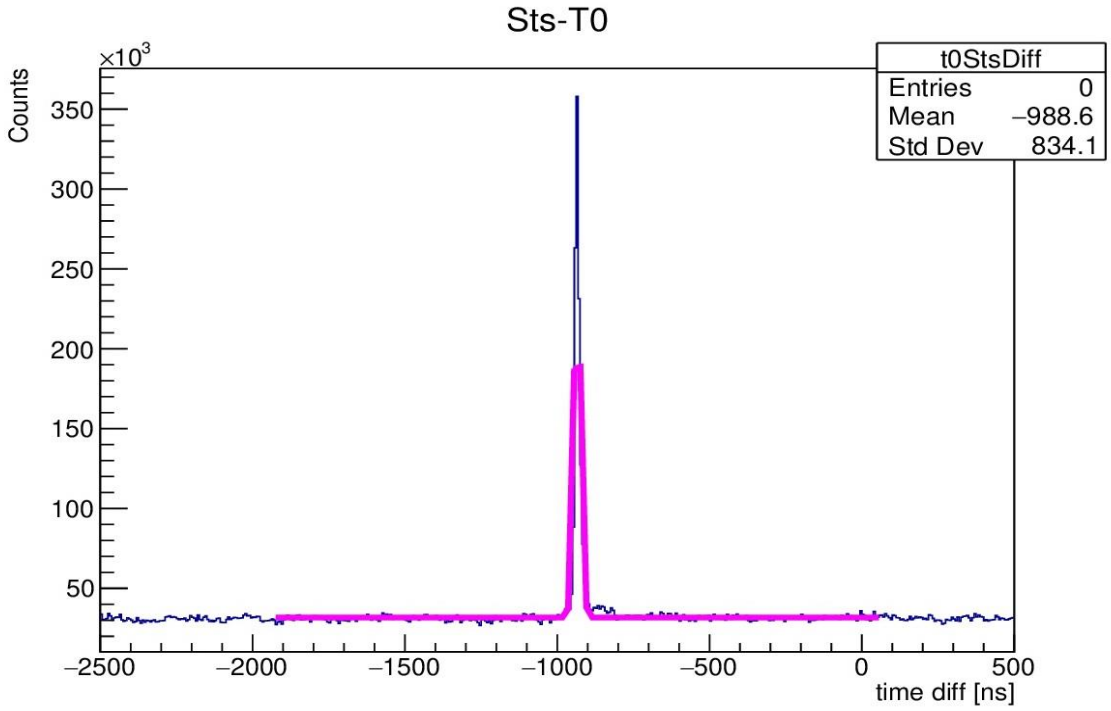


Abbildung 4.2 Vergrößerte Ansicht des Fits des STS-Histogramms zu Run 831, TSA-File 0000, 100.000 T0 Digis

Dass sich die Histogramme und entsprechend auch die Fits der verschiedenen Subsysteme unterscheiden, zeigt *Abbildung 4.3*. Da der abgebildete TRD-Peak im Vergleich zum Hintergrundrauschen sehr breit und nicht sonderlich hoch ist, kann er nahezu perfekt über eine Gauß-Kurve gefittet werden.

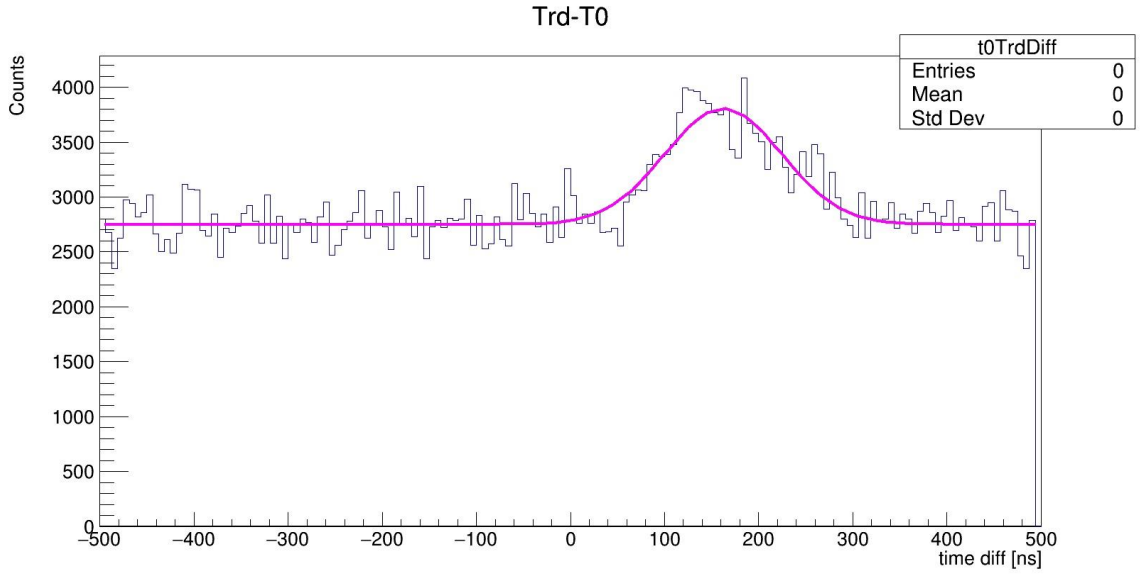


Abbildung 4.3 Fit des TRD-Histogramms zu Run 831, TSA-File 0000, 100.000 T0 Digis

Um die Qualität eines Fits und dementsprechend die des zugehörigen Peaks zu ermitteln, werden drei Kriterien für jeden Fit anhand der Parameter überprüft:

1. Die Höhe des gefitteten Peaks muss einen bestimmten Mindestwert überschreiten.
2. Der Mittelpunkt des gefitteten Peaks muss in der Nähe des höchsten Punkts des Histogramms liegen.
3. Die Breite des gefitteten Peaks darf einen bestimmten Wert nicht überschreiten.

Der in 1. beschriebene Mindestwert wird anhand der Höhe des Rauschens, also *p0* berechnet. Da die Peaks der verschiedenen Subsysteme verschieden hoch sind, erfolgt die Berechnung des Wertes individuell. Erfahrungswerte zeigen, dass ein echter Peak für STS, TOF, PSD und RICH eine Höhe *p1* von mindestens 50% bezüglich der Höhe des Rauschens *p0* erreicht. Für den MUCH-Peak hingegen wird eine Höhe *p1* von mindestens 25% von *p0* und für den TRD-Peak eine Höhe *p1* von mindestens 10% von *p0* erwartet. Zur besseren Funktionalität können diese Schwellenwerte über das *check_timing.C* Makro nach Bedarf angepasst werden (s. Anh. *Abbildung 7.3*). Dafür werden die Variablen *fStsMinHeight*, *fMuchMinHeight*, *fTrdMinHeight*, *fTofMinHeight*, *fRichMinHeight* und *fPsdMinHeight* implementiert, die im Folgenden durch eine Pseudovariable *fSystemMinHeight* ersetzt werden.

Die Implementierung dieser Schwellenwertvorgabe sieht vor, dass bei einem Fehler der boolsche Wert false zurückzugeben wird. Daraus ergibt sich die Abfrage:

$$\boldsymbol{if(p1 < p0 * fSystemMinHeight)\{return\ false;\}}$$

Als Referenzwert, wie weit der höchste Punkt des gefitteten Peaks *p2* vom höchsten Punkt des Histogramms (histoPeak) entfernt liegen darf, kann die Standardabweichung σ (*p3*) des gefitteten Peaks genutzt werden. Da es sich bei den gefitteten Peaks annähernd um eine Normalverteilung handelt, kann mithilfe der Standardabweichung die Halbwertsbreite (HB) des Peaks berechnet werden. Dabei gilt: $\boldsymbol{HB \approx 2,3548 * \sigma}$ [15]

Die Halbwertsbreite gibt die Breite des Peaks bei halber Höhe an. Sollte der höchste Punkt des gefitteten Peaks weiter als die Halbwertsbreite vom höchsten Punkt des Histogramms entfernt liegen, würde der Fit also nicht korrekt auf dem höchsten Punkt des Histogramms liegen. Entsprechend ergibt sich die Abfrage:

$$\boldsymbol{if(abs(p2 - histoPeak) > 2.3548 * p3)\{return\ false;\}}$$

Das 3. Kriterium sieht vor, dass die Breite des gefitteten Peaks einen bestimmten Schwellenwert nicht überschreiten darf. Dieser Schwellenwert wird über die Variable *fFitMaxPeakWidth* festgelegt, die ebenfalls über das *check_timing.C* Makro angepasst werden kann. Es wird vorausgesetzt, dass die echten Time Offset-Peaks wie in *Abbildung 2.5*, maxi-

mal eine Breite am Fuß des Peaks von 200 ns haben, weshalb *fFitMaxPeakWidth* standardmäßig auf den Wert 200 gesetzt wird. Die Breite des gefitteten Peaks wird wieder über die Halbwertsbreite bestimmt. Sollte die Halbwertsbreite eines Peaks den in *fFitMaxPeakWidth* festgelegten Wert überschreiten, welcher die Breite des Peaks auf Höhe des Rauschens beschreibt, ist der Fit fehlgeschlagen. Da es aufgrund der oben beschriebenen Error beim Fitting passieren kann, dass die Standardabweichung einen negativen Wert annimmt, wird in der Abfrage der Betrag der Halbwertsbreite genutzt. Dementsprechend folgt also die Abfrage:

$$\boldsymbol{if(abs(p3 * 2.3548) > fFitMaxPeakWidth)\{return\ false;\}}$$

Wird von einem dieser drei Kriterien der boolsche Wert false zurückgegeben, so wird eine Fehlermeldung erzeugt, die entsprechend spezifiziert welche der Abfragen nicht erfüllt werden konnte.

Die errechneten Time Offsets werden durch die Erweiterung der Bin Breite und das Fitting auf ihre Echtheit geprüft. Sollte eines der oben genannten Kriterien nicht erfüllt werden, wird eine Fehlermeldung ausgegeben. Da die beim Fitting entstandenen Histogramme abgespeichert werden, kann im Falle eines Errors visuell überprüft werden was das Problem ist. Wenn kein Error vorliegt, können die Time Offsets ohne Prüfen der Histogramme in die Unpacker übernommen werden. Die Funktionen für die Qualitätssicherung werden im *Finish* Teil des Programms aufgerufen.

4.3.2.3 Bekannte Fehler beim Fitten des Peaks

Durch das vermehrte Testen des Fittings an verschiedenen Runs mit verschieden vielen T0 Digis hat sich gezeigt, dass insbesondere in Runs mit hohen Kollisionsraten das Fitten der MUCH- und TRD-Peaks Probleme mit sich bringen kann. *Abbildung 4.4* zeigt den magentafarbenen, fehlgeschlagenen Fit des MUCH Histogramms von Run 856, TSA-File 0 mit 500.000 T0 Digis. Der Fit liegt grundsätzlich an der korrekten Stelle im Histogramm, da bei etwa 0 ns ein Peak zu erkennen ist. Allerdings wird vermutet, dass der Peak aufgrund der hohen Schwankungen im Hintergrundrauschen sowie seiner vergleichsweise niedrigen Höhe nicht erkannt werden kann. Die relative Höhe des Peaks zum Hintergrundrauschen steigt auch mit erhöhter Anzahl verwendeter T0 Digis nicht an. Zum Vergleich wird in *Abbildung 4.5* das MUCH Histogramm von Run 831, TSA-File 0 mit 100.000 T0 Digis gezeigt, in dem der Peak erfolgreich erkannt und gefittet wurde. Im Falle eines fehlgeschlagenen Fits ist es nur möglich den Peak visuell aus dem erzeugten Histogramm auszulesen. Ein entsprechender Hinweis wird mit der Fehlermeldung, dass der Fit fehlgeschlagen ist ausgegeben.

Ein weiteres Problem besteht beim Fitten des TRD-Peaks bei Runs mit hohen Kollisionsraten. Wie in *Abbildung 4.6* zu sehen ist, verliert der TRD-Peak vollständig die Form einer Normal-

verteilung und sammelt sich in einem Bin. Dadurch ist es für die Fitting Funktion nicht mehr möglich den Peak korrekt zu erkennen. Auch in diesem Fall ändert die Anzahl der verwendeten T0 Digis nichts am Aussehen bzw. an den Eigenschaften des Peaks.

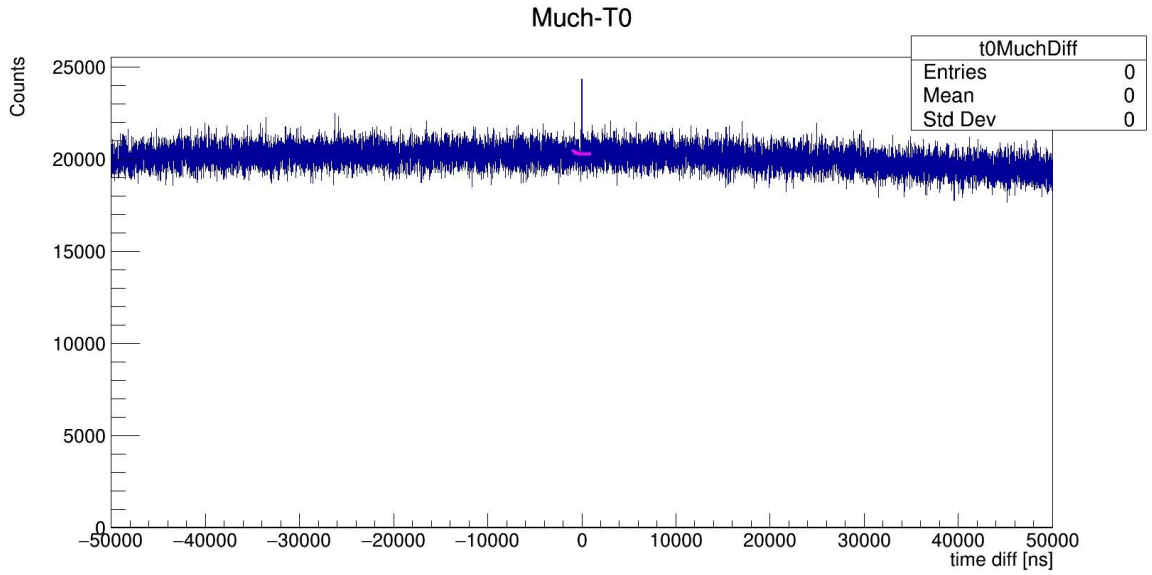


Abbildung 4.4 Fehlgeschlagener Fit des MUCH Histogramms für Run 856, TSA-File 0, 500.000 T0 Digis

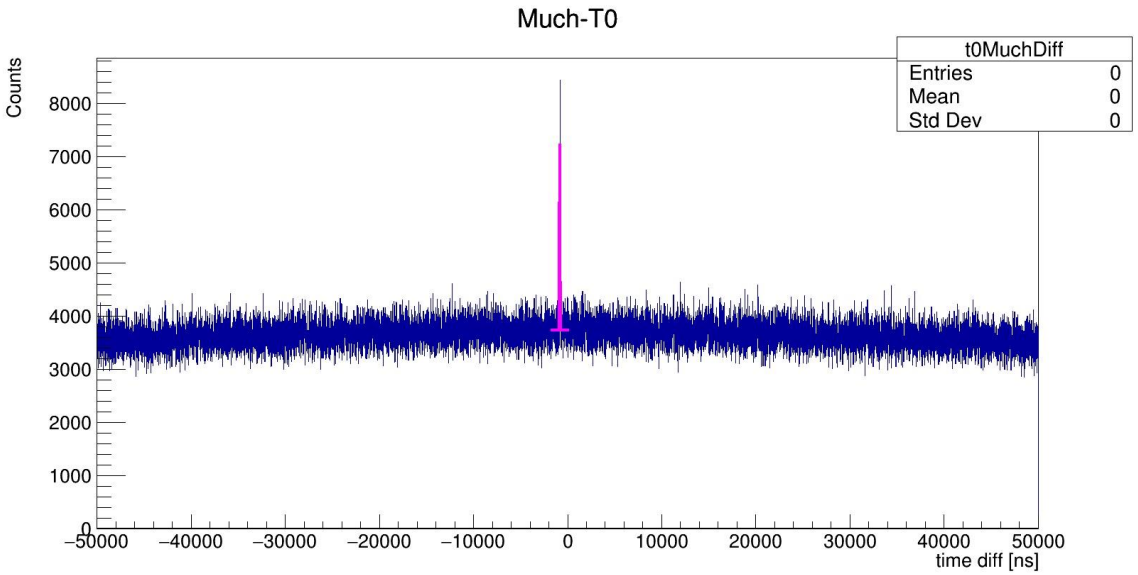


Abbildung 4.5 Erfolgreicher Fit des MUCH Histogramms für Run 831, TSA-File 0, 100.000 T0 Digis

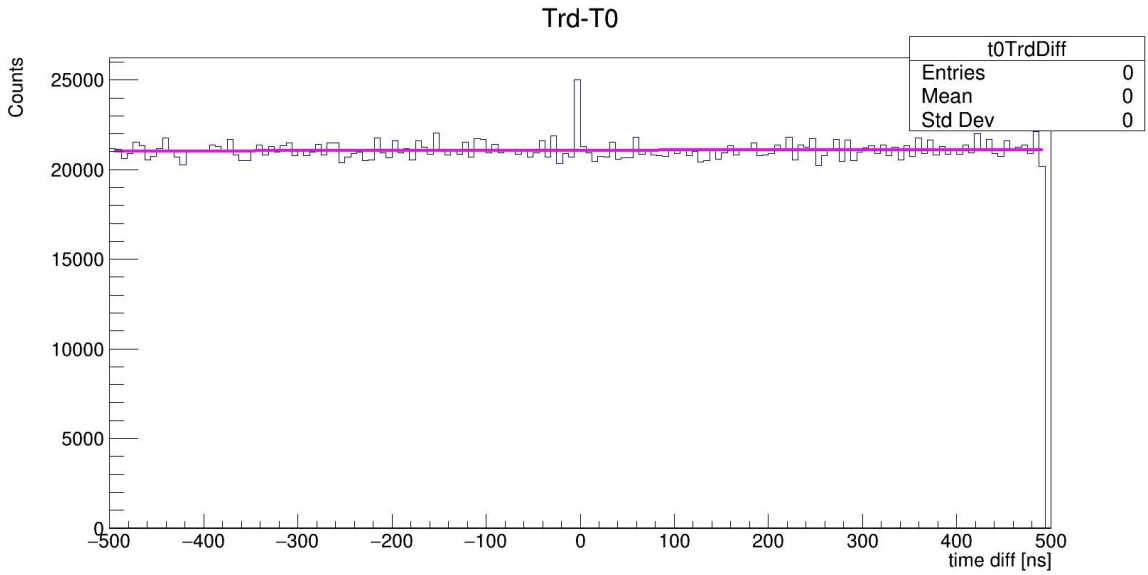


Abbildung 4.6 Fehlgeschlagener Fit des TRD Histogramms für Run 856, TSA-File 0, 500.000 T0 Digis

Anhand dieser Ausnahmefälle wird klar, dass bei einem fehlgeschlagenen Fit trotzdem ein Peak vorhanden sein kann. Aufgrund der speziellen Umstände ist es allerdings nicht möglich diese Peaks mit den gegebenen Mitteln zuverlässig zu verifizieren. Aus diesem Grund wird bei einem fehlgeschlagenen Fit eine Fehlermeldung ausgegeben, die darauf hinweist dass das Histogramm nochmal visuell auf einen Peak überprüft werden soll.

5 Ergebnisse

5.1 Probleme bei der Time Offset Berechnung

Die Berechnung der Time Offsets kann durch verschiedene Faktoren negativ beeinflusst werden. In allen Tests die im Zusammenhang dieser Arbeit durchgeführt werden, ist es nicht möglich, den Time Offset des RICH-Detektors zu errechnen. Der Grund dafür ist ein Problem im RICH-Unpacker in der aktuellen Softwareversion (commit cc19d3ab3a127ffd47d2dcbc8705cf212757b8a3 vom 10.08.2020 [16]), was dazu führt, dass die eigentlich zusammengehörigen T0 und RICH Digis nicht im gleichen TS gespeichert werden und entsprechend bei der Time Offset Berechnung nicht miteinander verglichen werden [17]. Der Fehler wurde gemeldet und ist behoben, allerdings noch nicht für die aktuell verwendete Version des RICH-Unpackers.

Grundsätzlich haben die durchgeführten Tests im Verlauf der Arbeit gezeigt, dass für die korrekte Berechnung der Time Offsets Subsysteme STS, TOF und PSD etwa 10.000 T0 Digis benötigt werden. Für den MUCH-Detektor scheint die Mindestanzahl der T0 Digis jedoch zu variieren. In Run 831 reichen z.B. ebenfalls 10.000 T0 Digis aus, in Run 812 hingegen 70.000 und in Run 856 müssen sogar etwa 500.000 T0 Digis verglichen werden, damit das Ergebnis eindeutig wird. Für den TRD Time Offset werden im Falle eines Runs mit niedriger Kollisionsrate etwa 30.000 T0 Digis zur Berechnung benötigt und im Falle eines Runs mit hoher Kollisionsrate etwa 70.000.

Aufgrund dieser stark schwankenden Mindestanforderungen von T0 Digis wird empfohlen, den Parameter *fT0DigiLimit* bei Runs mit niedriger Kollisionsrate (2 kHz) auf 100.000 und bei Runs mit hoher Kollisionsrate (100 kHz) auf 500.000 zu setzen.

5.2 Verwendete Strahlzeiten

Die zum Testen der Ergebnisse gewählten Runs sind Run 831 und Run 856. Run 831 wird genutzt, da dieser sich bereits als Musterbeispiel bewährt hat (s. Kap. 3.1) und verlässliche Daten liefert. Da Run 831 ein Run mit einer vergleichsweise niedrigen Kollisionsrate von etwa 2 kHz ist, ist es sinnvoll, außerdem einen Run mit einer vergleichsweise hohen Kollisionsrate von 100 kHz zu betrachten. Deshalb wird Run 856 zum Vergleich gewählt.

Zum Messen der Laufzeiten wurden die Berechnungen für möglichst aussagekräftige Ergebnisse auf dem FLES compute node pn06 durchgeführt. Die dokumentierten Laufzeiten entsprechen also denen, die bei einer tatsächlichen offline-Datenaufbereitung im mCBM-Experiment entstehen würden.

5.3 Speedup insgesamt

Aufgrund der Anforderungen an die Mindestanzahl der verwendeten T0 Digis variiert der Laufzeitgewinn durch die Verbesserungen von Run zu Run. In Tabelle 5.1 wird die Laufzeit des ursprünglichen Algorithmus aus der oben beschriebenen CbmRoot Version mit dem Algorithmus, der die in Kap. 4 beschriebenen Optimierungen enthält verglichen.

Run	Laufzeit des ursprünglichen Algorithmus mit allen TS (s)	Laufzeit des optimierten Algorithmus mit Mindestanzahl Digis (s)	Speedup
Run 831 TSA-File 0	1036s	30 s	34,5
Run 856 TSA-File 0	1626s	96s	17

Tabelle 5.1 Laufzeitvergleich zwischen dem ursprünglichen und optimierten Algorithmus, gemessen auf pn06

Es zeigt sich, dass im Falle eines Runs mit niedriger Kollisionsrate ein Speedup von bis zu 34,5 erreicht werden kann, bei einem Run mit hoher Kollisionsrate hingegen nur ein Speedup von 17. Für optimierte Version des Algorithmus wurde dabei *fT0DigiLimit* auf die empfohlene Mindestanzahl an T0 Digis aus Kap. 5.1 gesetzt.

Tabelle 5.2 zeigt die genauen Anteile der einzelnen Funktionen an der Gesamtlaufzeit. Diese können direkt mit den Werten aus Tabelle 3.2 verglichen werden, welche die Laufzeiten der einzelnen Funktionen des ursprünglichen Algorithmus zeigt.

Der Laufzeitgewinn in der *Init*- und *Finish*-Funktion kommt daher, dass die Histogramme entfernt wurden. Diese müssen also weder initialisiert noch abgespeichert werden. Die Laufzeit der beiden Funktionen nehmen somit nur noch einen sehr geringen Anteil an der Gesamtlaufzeit ein.

Die Funktion *CheckTimeOrder* wurde gänzlich aus dem Programm entfernt und beansprucht somit keinerlei Laufzeit mehr.

Der Großteil der eingesparten Laufzeit kommt aus der *CheckInterSystemOffset*-Funktion. Sowohl das Entfernen der Histogramme als auch die Verringerung der genutzten Daten sowie die Optimierung der Datenstruktur spielen dabei eine Rolle.

Auffällig ist, dass im Fall von Run 831 der Overhead einen Anteil von 61% an der Gesamtlaufzeit hat. Vergleicht man den Overhead mit dem des Ursprungsalgorithmus, kann man erkennen, dass er sich nicht verändert hat. Wie in Kap. 4.1 beschrieben hängt der Overhead von der Verwaltung der Timeslices durch das FairRoot Framework ab und wird somit durch keine der Änderungen beeinflusst.

Für Run 856 ist die Laufzeit des Overheads gestiegen, was daran liegt, dass in der optimierten Version des Algorithmus alle Timeslices verwendet werden, im Gegensatz zu den 30 Timeslices aus Tabelle 3.2. Der Anteil des Overheads beträgt in diesem Fall etwa 22%.

Funktionsname	Optimierte Laufzeit (s) für Run 831 TSA-File 0000, 100.000 T0 Digis	Optimierte Laufzeit (s) für Run 856 TSA-File 0000, 500.000 T0 Digis
Init()	0,2	0,2
CheckInterSystemOffset()	11,25	74
Finish()	0,2	0,2
Zusammen	11,65	74,4
Gemessene Gesamtlaufzeit	29,89	96,18
Overhead	18,24	21,78

Tabelle 5.2 Analyse der Laufzeiten des optimierten Algorithmus, gemessen auf pn06

Wie bereits in Kap. 4.1 beschrieben, ist es im Rahmen dieser Arbeit nicht möglich, den Overhead durch das FairRoot Framework zu verringern.

Zudem ist anzumerken, dass die Tests auf dem FLES compute node pn06 durchgeführt wurden und die Laufzeit entsprechend durch eine Verbesserung der Hardwarekomponenten weiter gesenkt werden kann. Tests auf dem login node zeigen, dass die Laufzeit von Run 831 mit 100.000 T0 Digis auf 13 Sekunden, mit einem Overhead Anteil von 65%, verringert werden kann. Die Ausführung von Run 856 mit 500.000 T0 Digis auf dem login node hat eine Laufzeit von 40 s mit einem Overhead Anteil von 25%.

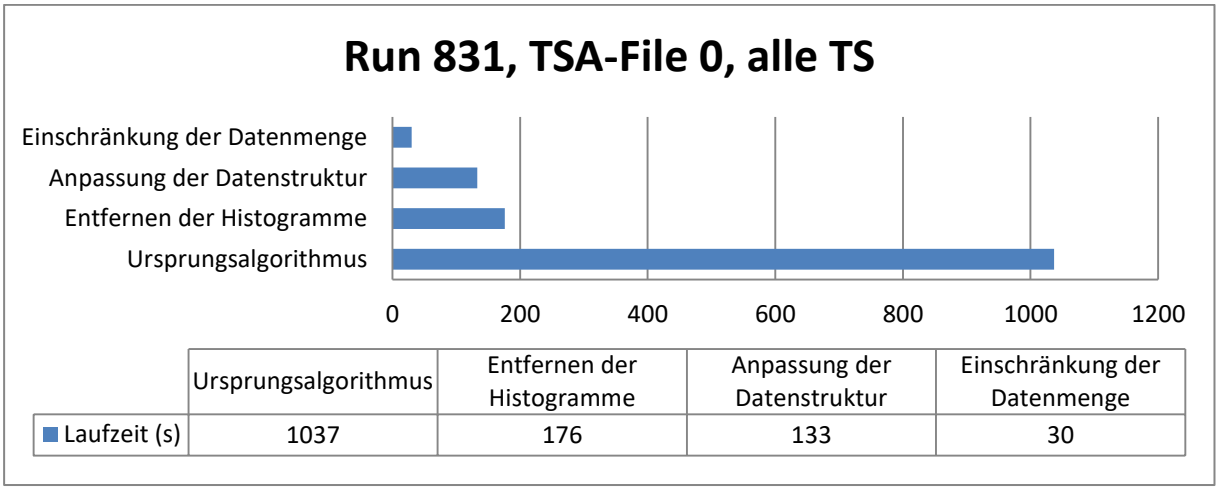
5.4 Speedup im Detail

Um herauszufinden, welche Änderungen welchen Anteil an den Laufzeitverbesserungen haben, wird nach jeder implementierten Verbesserung ein Laufzeittest gemacht. *Abbildung 5.1* und *Abbildung 5.2* zeigen die Anteile für Run 831 und 856. Der CheckTiming-Algorithmus wird für Run 831 in jeder Variante mit allen Timeslices ausgeführt. Bei Run 856 hingegen werden nur 30 Timeslices verwendet, außer bei der Einschränkung der Datenmenge. Dort wird der Algorithmus mit allen Timeslices aufgerufen, da die verwendeten Timeslices durch den Parameter *fT0DigiLimit* bestimmt werden. Dies hat zur Folge, dass der Overhead im Vergleich zum Ursprungsalgorithmus, dem Entfernen der Histogramme und der Anpassung der Datenstruktur etwas höher ist.

Die Optimierungen wurden in der gleichen Reihenfolge wie in Kap. 4 implementiert. Das Entfernen der Histogramme hat die Laufzeit von Run 831 von 1037s auf 176s verringert und somit einen Speedup von etwa einem Faktor 5,9. Für Run 856 bringt diese Änderung einen Speedup von etwa 6,5.

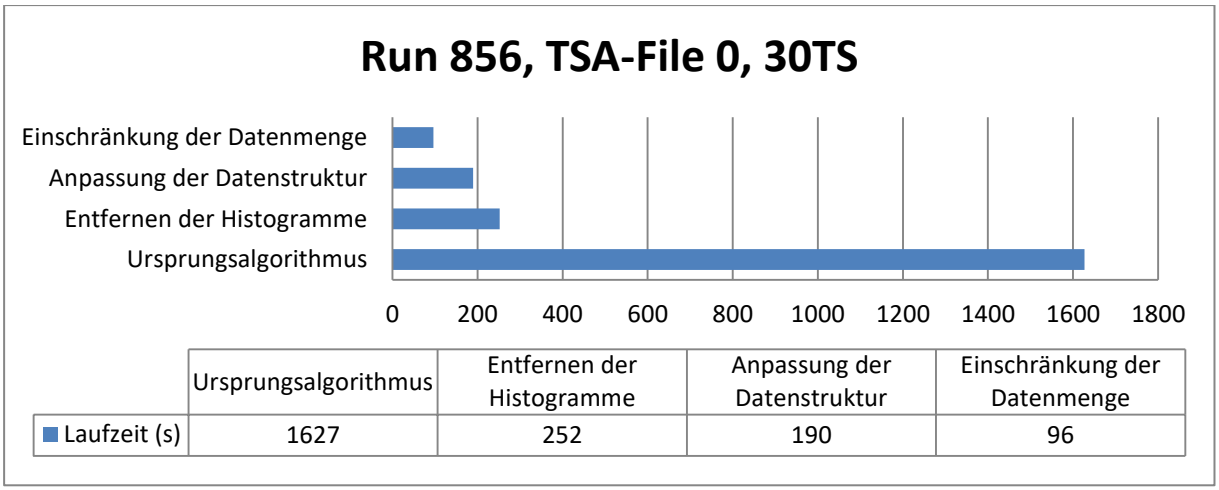
Die Anpassung der grundlegenden Datenstruktur von Histogrammen zu Vectoren wird ausgehend von der Optimierung der entfernten Histogramme gemessen. In beiden getesteten Runs ergibt sich ein Speedup von 1,3.

Durch die zusätzliche Einschränkung der Datenmenge kann für Run 831 ein Speedup von 4,4 und für Run 856 ein Speedup von 2 erreicht werden. Die beiden Runs unterscheiden sich bei dieser Optimierung aufgrund der verschiedenen Mindestanzahl von T0 Digis am meisten.



	Ursprungsalgorithmus	Entfernen der Histogramme	Anpassung der Datenstruktur	Einschränkung der Datenmenge
Laufzeit (s)	1037	176	133	30

Abbildung 5.1 Laufzeitverbesserungen der verschiedenen Optimierungen für Run 831, TSA-File 0, gemessen auf pn06



	Ursprungsalgorithmus	Entfernen der Histogramme	Anpassung der Datenstruktur	Einschränkung der Datenmenge
Laufzeit (s)	1627	252	190	96

Abbildung 5.2 Laufzeitverbesserungen der verschiedenen Optimierungen für Run 856, TSA-File 0, 30TS, gemessen auf pn06

6 Zusammenfassung

Der Datenanalyseprozess ist ein wichtiger Teil des mCBM-Experiments. Im Verlauf dieses Prozesses werden in mehreren Schritten aus den Detektor-Rohdaten Kollisionsevents rekonstruiert. Aktuell wird dieser Prozess offline durchgeführt, da die Laufzeiten der einzelnen Komponenten zu hoch sind. Da geplant ist, dass dieser Prozess online, direkt bei der Datenerfassung durchgeführt werden soll ist es notwendig die Laufzeiten der beteiligten Algorithmen zu optimieren.

Im Rahmen dieser Arbeit wurden die Laufzeiten der Unpacker-Algorithmen sowie die Laufzeit des CheckTiming-Algorithmus analysiert. Die Analysen ergaben, dass der Bedarf einer Verbesserung des CheckTiming Algorithmus am größten ist. Dementsprechend wurde die Optimierung dieses Algorithmus zum Schwerpunkt dieser Arbeit gewählt.

Durch Verbesserungen an der genutzten Datenstruktur, Entfernen von nicht essenziellen Berechnungen und dem Einschränken der genutzten Datenmenge konnte softwareseitig ein Speedup von bis zu 34,5 erreicht werden. Dies gilt allerdings nur für Runs mit einer niedrigen Kollisionsrate von 2 kHz. Um die Laufzeiten der Runs mit einer hohen Kollisionsrate von 100 kHz auf eine ähnlich geringe Laufzeit zu verbessern, ist es sicherlich in Zukunft lohnenswert, die genaue Funktionsweise der Berechnung des Algorithmus noch einmal näher zu betrachten. Kann dadurch herausgefunden werden, warum die benötigte Mindestanzahl von T0 Digis so stark schwankt, ist es möglich, die Laufzeit für diese Art von Run erneut zu optimieren. Zudem hat sich gezeigt, dass ein Upgrade der Hardware die Laufzeit nochmal um mindestens einen Faktor 2 verbessern kann.

Die Ergebnisse zeigen, dass der durch das FairRoot Framework erzeugte Overhead einen Anteil von bis zu über 50% der Laufzeit des optimierten CheckTiming-Algorithmus ausmacht. Daher ist es sinnvoll in folgenden Arbeitsschritten zu überprüfen, ob die Abhängigkeit der Algorithmen des offline-Analyseprozesses vom FairRoot Framework notwendig ist und diese entsprechend zu verbessern.

Neben den Laufzeitoptimierungen wurde mithilfe des Fittings außerdem ein Weg gefunden die Qualität der errechneten Time Offsets effizient überprüfen zu können. Die Qualitätskriterien können durch neue Makro-Parameter flexibel angepasst werden, falls hardwareseitige Änderungen an den Detektoren auftreten.

Damit der Prozess der offline-Datenanalyse mit in die online-Abläufe übernommen werden kann, müssen dennoch weitere Optimierungen vorgenommen werden. Insbesondere wird empfohlen, dass dafür der Unpacker des TRD-Detektors näher betrachtet und verbessert wird.

7 Anhang

```
1   void unpack_tsa_mcbm(TString inFile  = "",
2
3                        UInt_t uRunId   = 0,
4
5                        UInt_t nrEvents = 0,
6
7                        TString outDir  = "data",
8
9                        TString inDir   = "") {
10
11    ...
12    ...
13    ...
14
15    unpacker_sts->SetTimeOffsetNs(-936);   // Run 811-866
16    unpacker_much->SetTimeOffsetNs(-885);  // Run 811-866
17    unpacker_trdR->SetTimeOffsetNs(0);     // Run 811-866
18    unpacker_tof->SetTimeOffsetNs(25);     // Run 811-866
19    unpacker_rich->SetTimeOffsetNs(-310);  // Run 811-866
20    unpacker_psd->SetTimeOffsetNs(-225);   // Run 811-866
```

Abbildung 7.1 Quellcode Ausschnitt aus dem Unpacker-Makro *unpack_tsa_mcbm.C* [18]

```
1   void CbmCheckTiming::Exec(Option_t* /*option*/) {
2     if (fNrOfT0Digis < fT0DigiLimit){
3       LOG(debug) << "executing TS " << fNrTs;
4
5       CalcNoOfDigis();
6       if (fCheckInterSystemOffset) CheckInterSystemOffset();
7       fNrTs++;
8     }
9   }
```

Abbildung 7.2 Quellcode der Funktion *CbmCheckTiming::Exec* nach Optimierung

```
1   void check_timing(TString fileName,
2                     UInt_t uRunId  = 0,
3                     Int_t nDigis = 100000,
4                     Int_t nEvents  = 0,
5                     TString outDir = "data/") {
6
7   ...
8     CbmCheckTiming* timeChecker = new CbmCheckTiming();
9   ...
10    timeChecker->SetT0DigiLimit(nDigis);
11
12    timeChecker->SetStsFitMinHeight(0.5);
13    timeChecker->SetMuchFitMinHeight(0.25);
14    timeChecker->SetTrdFitMinHeight(0.1);
15    timeChecker->SetTofFitMinHeight(0.5);
16    timeChecker->SetRichFitMinHeight(0.5);
17    timeChecker->SetPsdFitMinHeight(0.5);
18    timeChecker->SetFitMaxPeakWidth(100);
19  ...
```

Abbildung 7.3 Quellcode der Ergänzungen im Makro *check_timing.C*

```
1   Double_t CbmCheckTiming::gauss(Double_t *x, Double_t *par) {
2    return par[0]*exp(-0.5*((x[0]-par[1])/par[2]) * ((x[0]-par[1])/par[2]) );
3   }
```

Abbildung 7.4 Quellcode der neuen Funktion *CbmCheckTiming::gauss* [14]

```
1   Double_t CbmCheckTiming::fitFunction(Double_t *x, Double_t *par){
2     return background(par) + gauss(x,&par[1]);
3   }
```

Abbildung 7.5 Quellcode der neuen Funktion *CbmCheckTiming::fitFunction* [13]

```
1   Double_t CbmCheckTiming::background(Double_t *par){
2     return par[0];
3   }
```

Abbildung 7.6 Quellcode der neuen Funktion *CbmCheckTiming::background* [13]

```
1   Bool_t CbmCheckTiming::CheckIndexPeakStability(std::vector<Int_t>* t0Diff, Int_t &peakIndex){
2     Int_t vecSize = t0Diff->size() - 1;
3     Int_t compBinSize = 8;
4
5     std::vector<Int_t> twoBinWidth = ChangeVecBinSize(t0Diff, 2);  // Initial Peak is measured
      with 2 binWidth, so no random errors occur
6     std::vector<Int_t> eightBinWidth = ChangeVecBinSize(t0Diff, compBinSize);
7
8     Int_t initialPeakIndex = GetPeakIndex(&twoBinWidth) * 2;
9     peakIndex = initialPeakIndex;
10    Int_t widerPeakIndex = GetPeakIndex(&eightBinWidth) * compBinSize; //multiply with binSize
      to compare it to original Index
11
12    std::cout << "Initial Peak: " << initialPeakIndex - vecSize/2 << std::endl;
13    std::cout << "Rougher Peak: " << widerPeakIndex - vecSize/2 << std::endl;
14
15    if(initialPeakIndex == 0){
16      std::cout << "No data" << std::endl;
17      return false;
18    }else
19    {
20      if(abs(initialPeakIndex - widerPeakIndex) < 2*compBinSize){
21        std::cout << "Good Peak" << std::endl;
22        return true;
23      }else{
24        std::cout << "Bad Peak, difference is: " << initialPeakIndex - widerPeakIndex <<
          std::endl;
25        return false;
26      }
27    }
28  }
```

Abbildung 7.7 Quellcode der neuen Funktion *CbmCheckTiming::CheckIndexPeakStability*

8 Abkürzungsverzeichnis

AGeV	Maßeinheit - Giga-Elektronenvolt pro Nukleon
CBM	Compressed Baryonic Matter
DPB	Data Processing Board
FAIR	Facility for Antiproton and Ion Research
FEE	Front-End-Electronics
FLIB	FLES Interface Board
GBT	Gigabit Transceiver
MUCH	Muon Chambers
MVD	Micro Vertex Detector
PSD	Projectile Spectator Detector
QCD	Quantenchromodynamik
RICH	Ring Imaging Cherenkov Detector
SIS100	Schwer-Ionen-Synchrotron 100
STS	Silicon Tracking System
T0	Time Zero Counter
TOF	Time-Of-Flight System
TRD	Transition Radiation Detector

9 Literaturverzeichnis

16. *CbmRoot Master Repository*. (10. August 2020). Abgerufen am 03. November 2020 von https://git.cbm.gsi.de/computing/cbmroot

13. Brun, R. (kein Datum). *FittingDemo.C File Reference.* Abgerufen am 27. Oktober 2020 von root.cern.ch: https://root.cern.ch/doc/master/FittingDemo_8C.html

12. CBM Softwaremeeting. (13. August 2020).

6. De Cuveland, J., Lindenstruth, V., & The CBM Collaboration. (Dezember 2011). A First-level Event Selector for the CBM-Experimentat FAIR. *Journal of Physics: Conference Series*.

10. Friese, V. (2018). *Estimate of the CBM computing requirements for operation at SIS-100.* Darmstadt.

11. Geier, T. (21. Mai 2020). Optimierung des Unpackers zur Datenprozessierung des TOF-Detektors am CBM-Experiment. Frankfurt am Main.

9. GSI Helmholtzzentrum für Schwerionenforschung GmbH. (10. April 2011). *FairRoot*. Abgerufen am 1. Oktober 2020 von https://cbmroot.gsi.de/?q=node/1

4. GSI Helmholtzzentrum für Schwerionenforschung GmbH. (2017). *cern.ch.* Abgerufen am 30. September 2020 von https://accelconf.web.cern.ch/ipac2017/papers/wepva030.pdf

1. GSI Helmholtzzentrum für Schwerionenforschung GmbH. (kein Datum). *www.gsi.de*. Abgerufen am 28. September 2020 von https://www.gsi.de/forschungbeschleuniger/fair/forschung/cbm_im_inneren_eines_neutronensterns.htm

5. GSI Helmholtzzentrum für Schwerionenforschung GmbH. (kein Datum). *www.gsi.de.* Abgerufen am 30. September 2020 von https://www.gsi.de/forschungbeschleuniger/fair/die_maschine.htm

3. Hermann, N., & Sturm, C. (2020). *Application for beam time at GSI/FAIR.*

7. Hutter, D., & De Cuveland, J. (2020). *The FLES Detector Input Interface, Version 1.*

18. Loizeau, P.-A., Singhal, V., Karpushkin, N., Toia, A., & Raisig, P. (31. Oktober 2020). *gsi.de.* Abgerufen am 1. November 2020 von https://lxcbmredmine01.gsi.de/projects/cbmroot/repository/revisions/master/entry/Makro/beamtime/mcbm2020/unpack_tsa_mcbm.C

14. Redelbach, A. (13. August 2020). private E-Mail Kommunikation.

2. The CBM Collaboration. (24. Februar 2012). *fair-center.eu.* Abgerufen am 28. September 2020 von https://fair-center.eu/fileadmin/fair/experiments/CBM/documents/CBMatSIS100.pdf

8. The CBM Collaboration. (2017). *Beamtime Application A CBM full system test-setup.*

17. Weber, A. (13. August 2020). private E-Mail Kommunikation.

15. Weisstein, E. W. (kein Datum). *MathWorld*. Abgerufen am 03. November 2020 von https://mathworld.wolfram.com/GaussianFunction.html