technische universität
dortmund

Bachelorarbeit

Interaktive Hardware-Fehlerinjektion mit FAIL*

Housssaine Siti
3. August 2020

Betreuer:
Dr.-Ing. Horst Schirmeier
M.Sc. Ulrich Thomas Gabor

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
`https://ess.cs.tu-dortmund.de`

Inhaltsverzeichnis

1 Einleitung

In der heutigen Zeit spielen computergesteuerte Systeme eine essentielle Rolle und sind aus der modernen Gesellschaft kaum wegzudenken. Im Laufe der Jahre entwickeln sich Softwaresysteme mit rasanter Geschwindigkeit und steigender Komplexität. Da sich heutzutage ebenfalls lebenswichtige Dinge, wie etwa Verkehrsmittel, medizinische Geräte oder Verwaltungsprogramme nahezu ausschließlich softwaregesteuert verhalten, ist es umso wichtiger, deren Zuverlässigkeit zu gewährleisten.

Die reibungslose Funktionstüchtigkeit wird zudem durch externe Faktoren, wie etwa physikalische Einflüsse auf die Hardware, beeinflusst. Demzufolge besteht eine weitere Herausforderung darin, diese möglichen Störungspotentiale zu reproduzieren, um das Verhalten des Systems unter diesen Bedingungen zu bewerten. Die erhöhte Komplexität sowie äußere Einflussfaktoren erhöhen entsprechend die Fehleranfälligkeit eines Systems. Deswegen sollten Fehlertoleranzmechanismen entwickelt werden, welche beim Auftreten solcher Störungen die Funktionstüchtigkeit des Systems aufrecht erhalten.

1.1 Motivation

Zur Untersuchung der Zuverlässigkeit - bezüglich der Fehlertoleranz - gilt die Fehlerinjektion als bewährte Methode. Das Werkzeug FAIL* - „FAult Injection Leveraged" - [9], das am Lehrstuhl 12 entwickelt wird, ist ein Werkzeug für Fehlerinjektionskampagnen, die eine Reihe von Fehlerinjektionsexperimenten bei dem Zielprogramm bzw. Zielsystem durchführen. Die Laufzeit der Fehlerinjektionskampagnen hängt von der Größe des zu untersuchenden Zielsystems ab und kann Stunden oder Tage dauern. Der Entwickler kann die Ergebnisse der Fehlerinjektionskampagnen mithilfe von diversen Analysewerkzeugen erst abrufen, wenn die Kampagnen alle Fehlerinjektionsexperimente abgeschlossen haben. Damit ist eine Bewertung der Zuverlässigkeit zur Laufzeit der Kampagnen nicht möglich, da keine Zwischenergebnisse geliefert werden.

1.2 Ziele

Ziel der vorliegenden Arbeit ist die Erweiterung und Weiterentwicklung der Analyseverfahren in FAIL*. Hierfür gilt die Integration bereits vorhandener Analy-

sewerkzeuge in eine gemeinsame Benutzeroberfläche, welche einen schnellen und einfachen Zugriff auf die implementierten Metriken und Analysen zulässt und einen Vergleich der Programmvarianten ermöglicht. Die FAIL*-Erweiterung soll als Webanwendung erfolgen.

Es sollen diesbezüglich zwei Möglichkeiten erprobt werden, um die Entwicklungs-Iterationen zeitlich zu verkürzen. Zuerst wird untersucht, inwieweit Schätzungen und Zwischenergebnisse während der Laufzeit der Kampagne durchgeführt werden können. Das soll dem Entwickler die Möglichkeit bieten, eine Bewertung der Zuverlässigkeit der abgeschlossenen Fehlerinjektionsexperimente zur Laufzeit der Kampagne vorzunehmen. Des Weiteren soll dem Entwickler über die Benutzeroberfläche die Möglichkeit geboten werden, relevante Programmteile zu selektieren und diese zu priorisieren, indem die Fehlerinjektionsexperimente dieser Programmteile zuerst durchgeführt werden. Insbesondere die Skalierbarkeit der Erweiterungen soll mittels geeigneter Benchmarkprogramme untersucht werden.

1.3 Aufbau der Arbeit

Neben diesem Kapitel besteht die vorliegende Arbeit aus 5 weiteren Kapiteln:

- In Kapitel 2 werden die Grundlagen der Arbeit sowie sämtliche Begriffe definiert, die zum Verständnis der Arbeit nötig sind.
- In Kapitel 3 wird eine Problemanalyse durchgeführt, aus der sich Anforderungen ergeben. Anschließend wird ein Lösungsansatz entworfen und die mit ihm verbundenen Algorithmen vorgestellt.
- In Kapitel 4 wird die Realisierung der entworfenen Anwendung beschrieben.
- In Kapitel 5 werden die in Kapitel 3 vorgestellten Algorithmen untersucht und ausgewertet.
- In Kapitel 6 werden die Ergebnisse dieser Arbeit zusammengefasst und zum Schluss werden mögliche Verbesserungen bzw. Erweiterungen vorgeschlagen.

2 Grundlagen

2.1 Fehlertoleranz

Neben Verfügbarkeit, Zuverlässigkeit und Sicherheit spielt die Fehlertoleranz eine wichtige Rolle, insbesondere in eingebetteten Systemen, bei denen ein Fehler oder ein Ausfall katastrophale Folgen haben kann. Um auf Fehlertoleranz (engl.: fault tolerance) einzugehen, müssen die folgenden Begriffe definiert werden:

- Fehler (engl.: fault) oder auch Defekt: Ist die Ursache für einen Fehlzustand (engl.: error). Diese treten in zwei Formen auf, entweder als aktive (activ) Fehler, wenn sie Fehlerzustände verursachen oder als latente Fehler (dormant), wenn sie keinen Einfluss auf das Ergebnis des Systems haben [1].
- Fehlzustand (engl.: error): Ist die Folge der Aktivierung oder Manifestation des Defektes oder Fehlers im System [6].
- Ausfall (engl.: failure): Wenn das gelieferte Ergebnis des Systems von dem erwarteten Verhalten abweicht [1].
- Fehlerkette: Ist der Übergang von einem Fehler bzw. Defekt zu einem Fehlzustand, was zu einem Ausfall propagiert wird [6].

Unter Fehlertoleranz versteht man die Fähigkeit eines Systems [1], einen Ausfall zu vermeiden, auch wenn dieses Fehler enthält. Mit anderen Worten: Ein fehlertolerantes System ist ein System, das seine Dienste bzw. Funktionen gewährleistet, auch wenn Fehler vorliegen. Also ist es in der Lage, mit Störungen und Ausfällen umzugehen, ohne dass sie extern bemerkbar auffallen.

Im folgenden Abschnitt wird eine Methodik zur Bewertung von Fehlertoleranzmechanismen erläutert.

2.2 Fehlerinjektion

Um die eingesetzten Fehlertoleranzmechanismen zu bewerten, muss das System belastet werden. Diese Last kann in Form natürlicher Fehlerquellen, wie etwa elektromagnetische Strahlung, oder Störungen dargestellt werden, um das Systemverhalten unter diesen Bedienungen zu reflektieren und mit dem erwarteten

Verhalten zu vergleichen bzw. bewerten. Eine Möglichkeit für die Bewertung der Fehlertoleranz besteht in dem Einsatz der Fehlersimulation [8]. Bei der Fehlersimulation wird eine Anpassung der Systemmodelle durchgeführt, um den Effekt eines bestimmten Fehlers auf das Systemverhalten zu reflektieren. Dabei werden alle Fehler aus dem Fehlermodell simuliert. Ein Nachteil dieses Verfahrens liegt in dem hohen Zeitaufwand.

Als Alternative zur Fehlersimulation bietet sich die Fehlerinjektion (engl.: Fault injection) als ein Verfahren zur Bewertung der Fehlertoleranz eines Systems an [8]. Bei der Fehlerinjektion können nicht nur Fehler aus dem Fehlermodell reproduziert werden, sondern auch solche, die vom Fehlermodell nicht prognostiziert werden. Dabei werden in dem System bestimmte Fehlerarten zu einem bestimmten Zeitpunkt an einem bestimmten Ort gezielt verursacht.

2.3 FAIL*

FAIL* [9] ist ein Werkzeug zur Fehlerinjektion, das am Lehrstuhl 12 der technischen Universität Dortmund entwickelt wird. Neben der hohen Skalierbarkeit und der Beschreibung von Kampagnen unterstützt FAIL* verschiedene Hardwareplattformen. Des Weiteren bietet FAIL* die Möglichkeit, Fehlertoleranzmechanismen gründlich und zuverlässig zu bewerten. Dabei werden alle möglichen Szenarien abgedeckt und reproduziert, bei denen es u. a. zu einem Systemausfall kommen kann. Außerdem stellt FAIL* feine detaillierte Analyseergebnisse dem Entwickler zur Verfügung. Bei diesen Analysen wird dem Entwickler nicht nur eine globale Einsicht über die Fehlerinjektion des kompletten Fehlerraumes geliefert, sondern auch die Häufigkeit des Auftretens möglicher Fehlerarten in Abhängigkeit von der dafür zuständigen Quelle, wie z. B. Funktion oder Übersetzungseinheit. Somit hat der Entwickler eine vielseitige und umfangreiche Bewertung für die entwickelten Fehlertoleranzmechanismen und wird darüber informiert, welche von ihm entwickelten Programmteile fehlertolerant und welche fehleranfällig sind.

Der Fehlerinjektionszyklus beinhaltet die folgenden Schritte, wie in Abbildung 2.1 beschreiben ist [6], [9]:

1. Definition des Experiments: In diesem Schritt wird ein Experiment definiert.

2. Vorbereitung: Nach der Festlegung der Experimentbeschreibung findet ein Referenzlauf (engl.: golden run) statt. Dabei wird ein Ablaufprotokoll (engl.: trace) erstellt, das u. a. Laufzeit und Instruktionen des Testprogramms beinhaltet. Dieses Ablaufprotokoll wird in die Datenbank importiert und dient als Basis für den Vergleich für die späteren Fehlerinjektionsergebnisse. Aus dem Ablaufprotokoll folgt eine Liste an Fehlerinjektionsexperimenten, die durchgeführt werden sollen.

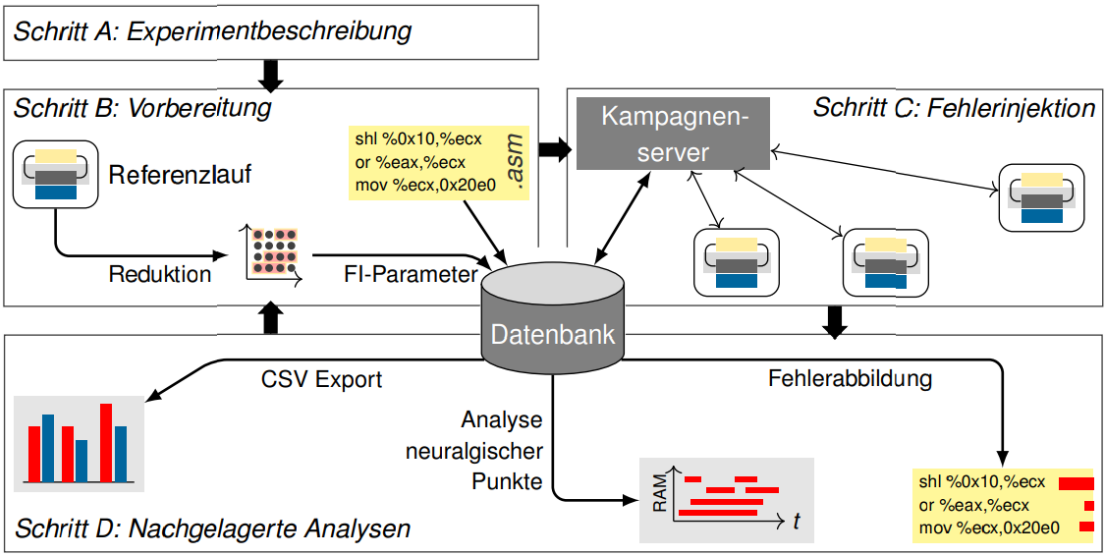

Abbildung 2.1: Der Ablauf des Fehlerinjektionszyklus, entnommen aus [6].

3. Fehlerinjektion: In diesem Schritt verteilt der FAIL*-Jobserver Experimente auf FAIL*-Clients, sammelt die Ergebnisse der zugeteilten abgeschlossenen Experimente und speichert diese in der Datenbank.

4. Analyse: Zum Schluss werden die Ergebnisse der Fehlerinjektion ausgewertet, die in der Datenbank gespeichert sind.

Die Ergebnisse der Fehlerinjektion des zu testenden Programms sind erst nach dem Ablauf der kompletten Fehlerinjektion verfügbar und sind wie folgt unterteilt:

- Gesamtes Auftreten (engl.: global occurrences): Gibt eine globale Einsicht über die eingetroffenen Fehlerarten und deren Häufigkeit.

- Auftreten pro Symbol (engl.: symbol occurrences): Liefert die Häufigkeit des Vorkommens auf Symbolebene.

- Auftreten pro Funktion (engl.: function occurrences): Liefert die Häufigkeit des Vorkommens auf Funktionsebene.

- Auftreten pro Übersetzungseinheit (engl.: translation unit occurrences): Liefert die Häufigkeit des Vorkommens pro Übersetzungseinheit. Dient dazu, welche Übersetzungseinheit welche Fehlerart verursacht hat.

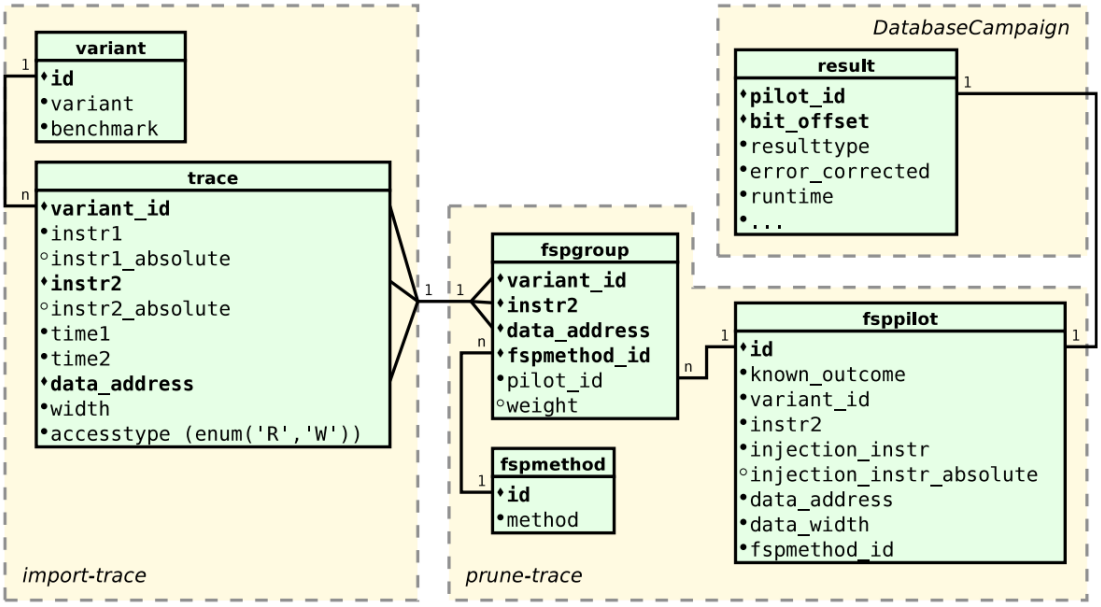

Abbildung 2.2: FAIL*-Datenbankschema, entnommen aus [9].

- Dauer der Variante (engl.: variationduration): Gibt die Laufzeit des Benchmarks sowie die Anzahl der dynamischen Instruktionen an.
- Fehlerraumplot (engl.: Faultscapeplot): Zeigt einen zweidimensionalen Fehlerraum an, bei dem die erste Dimension für einen Zyklus steht und die zweite für die verwendete Speicheradresse. Dabei steht jede Farbe für eine Fehlerart.

Die FAIL*-Datenbank besteht aus mehreren Tabellen, wie in Abbildung 2.2 abgebildet ist [9]. Die Tabelle `variant` dient der Speicherung von Programmvaianten, sie besitzt einen primären Schlüssel, der das untersuchte Zielsystem eindeutig identifiziert. Die Tabelle `result` enthält die Ergebnisse der abgeschlossenen Fehlerinjektionsexperimenten.

2.4 Softwarearchitektur im Web

Eine Webanwendung (engl.: Webapplikation) ist ein Anwendungsprogramm, deren Benutzeroberfläche von einem Browser bereitgestellt wird, während die Logik oder die Datenverarbeitung auf einem entfernten Webserver ausgeführt wird. Anders als klassische Desktop-Anwendungen, bei denen meistens eine Installation auf

einem Betriebssystem oder einer Laufzeitumgebung notwendig wird, sind Webanwendungen unabhängig von Betriebssystemen [5]. Eine Webanwendung wird in Frontend und Backend aufgeteilt, wobei das Backend die Datenverarbeitung, wie etwa Überprüfung von Benutzernamen und Passwörtern, übernimmt. Neben der Darstellung des Verarbeitungsresultats, z. B. erfolgreicher Login-Vorgang, stellt das Frontend, zwecks leichter Bedienbarkeit, u. a. Interaktionselemente zur Verfügung.

Zudem kann eine Webanwendung anhand einer 3-Schichten-Architektur in die folgenden Komponenten unterteilt werden:

- Darstellungsschicht oder Präsentationsschicht: Stellt Inhalte in einer Benutzeroberfläche dar und bietet dem Anwender die Möglichkeit, auf Benutzereingaben zu reagieren [4]. Weiterhin kann diese Schicht über eine Client-Prozesslogik verfügen, bei der meistens eine clientseitige Validierung der Eingaben erfolgt, bevor diese an die Webserverschnittstelle weitergeleitet sind.
- Anwendungsschicht: Verfügt über Schnittstellen, die Geschäftsprozesse, wie etwa Datenübertragung an die Darstellungsschicht oder Verarbeitung und Verwaltung der Benutzereingaben, umsetzen [4]. Diese Schnittstellen finden in einem entfernten Webserver statt.
- Datenbanksicht oder Persistenzschicht: Stellt der Anwendungsschicht die angefragten Daten zur Verfügung und dient der Speicherung der Daten [4].

JavaScript ist eine clientseitige Programmiersprache, die direkt von dem Webbrowser interpretiert werden kann [3]. Durch das Einsetzen von JavaScript ist es möglich, dynamische Aktionen durchzuführen, etwa Benutzereingaben zu validieren oder Elemente zur Laufzeit zu generieren.

JavaScript ist eine Single-Threaded-Umgebung, d.h. es können nicht ohne Weiteres gleichzeitig mehrere Skripts ausgeführt werden, die Ausführung erfolgt in einem einzigen Thread [10]. Hierbei werden alle Befehle sequentiell ausgeführt. Eine weitere Charakteristik von JavaScript besteht darin, dass der Typ einer Variable, anhand des in ihr gespeicherten Wertes, zur Laufzeit bestimmt bzw. geändert wird.

Ein Anwendungsbeispiel für JavaScript ist, die Kommunikation über AJAX mit dem Webserver zu initialisieren und Daten mit diesem auszutauschen, ohne dass der Nutzer diese Interaktion mitbekommt.

AJAX, Asynchronous JavaScript and XML dient der Kommunikation zwischen dem Client und Server. Dabei wird mittels JavaScript eine HTTP-Anfrage an den Server übermittelt. Dieser Vorgang kann asynchron verlaufen, wie in Listing 2.1 zu sehen. Das Warten auf die Serverantwort führt nicht zu einer Blockade

der Anwendung. Hierbei werden die beteiligte Prozesse wie zuvor erhalten. Die Verarbeitung der Rückgabefunktion wird erst erfolgen, wenn der Server eine Antwort an den Client zurückgibt. Hierbei findet ein versetzter Datenaustausch statt, der eine partielle Aktualisierung der Anwendung ermöglicht.

```
<script>
  $(document).ready(function() {
    function sendRequest()  {
      // Ziel url der Anfrage definieren
      AJAX_url = "server.php";
      // die zu sendenden Daten vorbereiten
      var data = {
        name : "Max Muster",
        address : "Musterweg 8"
      };
      // HTTP-POST-Anfrage senden
      $.post(AJAX_url, data, function(response){
        // einfache Ausgabe
        alert("Mich sehen Sie, wenn der Server antwortet");
      }
      // einfache Ausgabe
      alert("Mich sehen Sie sofort nach dem Versand der Anfrage");
    });
  });
</script>
```

Listing 2.1: Einfache HTTP-POST-Anfrage mit AJAX und jQuery

2.5 Fazit

In diesem Kapitel wurden die Grundlagen eingeführt, die zum Verständnis dieser Arbeit benötigt werden. Im Abschnitt 2.3 wurde der Fehlerinjektionszyklus von dem Werkzeug FAIL* verdeutlicht. Im nächsten Kapitel wird eine Ist-Zustandsanalyse durchgeführt, um die Anforderungen an die FAIL*-Erweiterung zu definieren. Anschließend wird anhand der imAbschnitt 2.4 vorgestellten Softwarearchitektur die FAIL*-Erweiterung vorgestellt.

3 Problemanalyse und Entwurf

Dieses Kapitel befasst sich mit der Problemanalyse und dem Entwurf der FAIL*-Erweiterung. Zuerst wird die Ausgangslage analysiert, dann werden die aus ihr ergebenen Anforderungen beschrieben. Anschließend werden die Architektur sowie Anwendungsfälle präsentiert.

3.1 Anforderungsanalyse

3.1.1 Ist-Zustandsanalyse

Für die Durchführung der Anforderungsanalyse wird zuerst die Ausgangslage bzw. der Ist-Zustand von FAIL* aus der Sicht des Benutzers betrachtet.

Wie im Abschnitt 2.3 erwähnt wurde, erhält der Benutzer die Ergebnisse der Fehlerinjektion erst nach dem kompletten Ablauf der Fehlerinjektionskampagne. Während der Laufzeit der Kampagne sind keine Ergebnisse der abgeschlossenen Fehlerinjektionsexperimente zu sehen. Der Benutzer soll also warten, bis die Fehlerinjektionskampagne beendet ist, dann kann er über unterschiedliche Skripte die Ergebnisse der Fehlerinjektion erhalten. Für diesen Zweck stellt FAIL* diverse Skripte zur Verfügung, die die einzelnen Analysen aus der Datenbank abrufen.

Nach der Beschreibung des Ist-Zustandes kommt es zu den folgenden Schlussfolgerungen:

- Wenn die Kampagne abgeschlossen ist, muss der Benutzer mehrere Skripte ausführen, um an die Ergebnisse zu gelangen.
- Während der Laufzeit der Kampagne werden keine Informationen über die bereits abgeschlossenen Experimente geliefert.
- Der Benutzer hat keine Möglichkeit, die Fehlerinjektion bestimmter Programmteile vorzuziehen.
- Die FAIL*-Datenbank gibt keine direkte Auskunft über den Kampagnenzustand.

3.1.2 Anforderungen

Aus der Schlussfolgerung der Ist-Zustandsanalyse ergeben sich die folgenden Anforderungen und Ziele:

Integration: Die Anwendung soll bereits abgeschlossene oder vorhandene Analyseverfahren in eine gemeinsame Benutzeroberfläche integrieren. Es sollen also die folgenden Bewertungsverfahren dargestellt werden:

- Gesamtes Auftreten
- Auftreten pro Symbol
- Auftreten pro Funktion
- Auftreten pro Übersetzungseinheit
- Dauer der Variante
- Fehlerraumplot
- Vergleich Benchmarks
- Variantenvergleich

Zwischenergebnisse: Die Anwendung soll zur Laufzeit der Kampagne bzw. während der Abarbeitung der Experimente die bereits vorliegenden Ergebnisse dem Benutzer zur Verfügung stellen.

Schätzung: Die Anwendung soll während der Kampagnenlaufzeit eine Schätzung bzw. Approximation, sowie eine Prognose über die Häufigkeit des Auftretens der Fehlerarten anzeigen.

Priorisieren: Die Anwendung soll dem Benutzer die Möglichkeit bieten, bestimmte oder für ihn relevante Programmteile zu priorisieren, sodass die Kampagne die ausgewählten Experimente zuerst durchführt.

3.2 Entwurf

3.2.1 Architektur

Neben den Komponenten der Darstellungs- und Anwendungsschicht beschreibt Abbildung 3.1 das Verhältnis zwischen der Anwendungsschicht und der bestehenden Architektur von FAIL*. Dabei wird die vorhandene FAIL*-Datenbank als Grundlage der Datenbankschicht für die Anwendung verwendet. **Die Datenbankschicht** dient der Bereitstellung der Daten, die von der Anwendungsschicht angefordert werden sowie die Speicherung der Ergebnisse von FAIL*-Experimenten.

Die Anwendungsschicht wird hauptsächlich von einem PHP-Skript betrieben, das für die Bereitstellung der benötigten Daten zuständig ist. Dieses Skript kommuniziert mit der Datenbank via SQL-Anfragen und reagiert auf AJAX-Anfragen, die von dem Client bzw. der Darstellungsschicht ausgelöst werden.

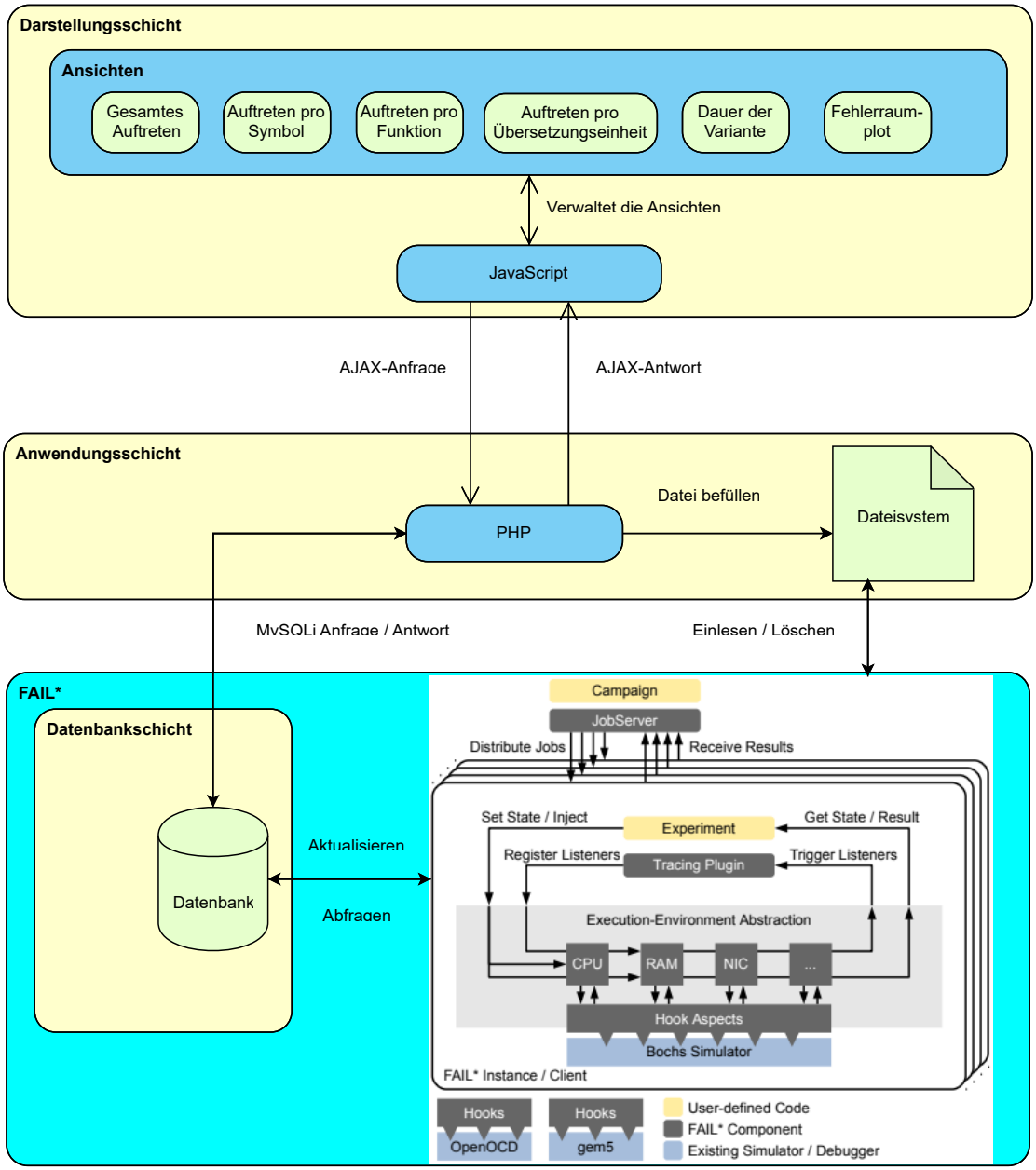

Abbildung 3.1: Architektur der Anwendung. Beschreibt die Komponenten der einzelnen Schichten und das Verhältnis zwischen der Anwendungsschicht und der bestehenden Architektur von FAIL*. Das Bild von FAIL* ist aus [9] entnommen.

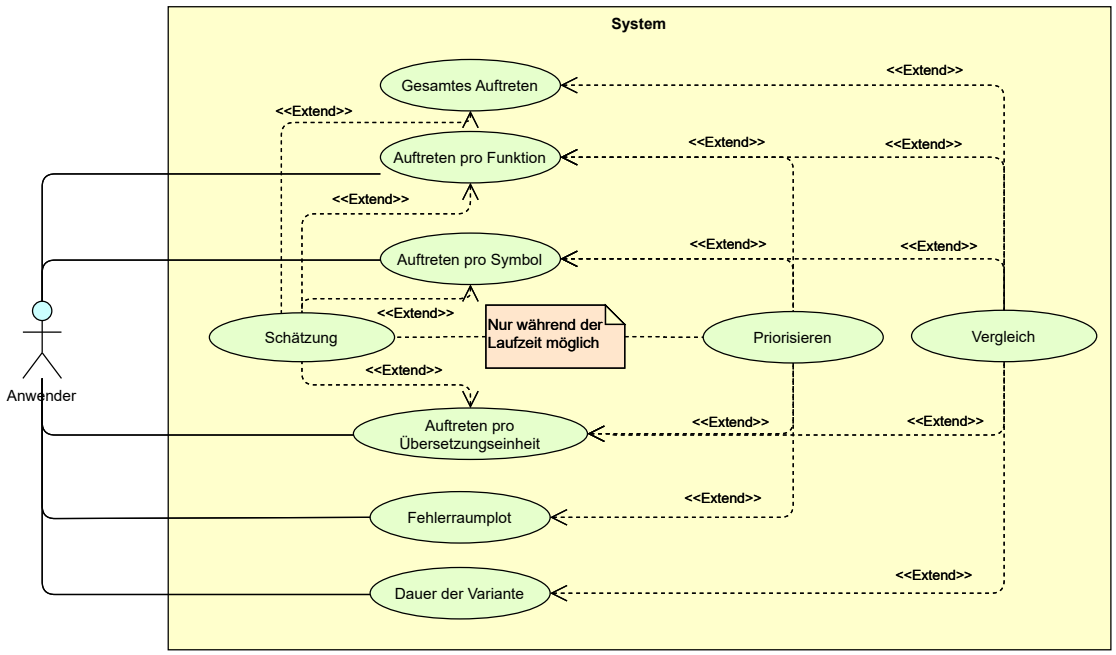

Abbildung 3.2: Anwendungsfalldiagramm beschreibt die mögliche Benutzerinteraktionen mit dem System.

Weiterhin findet eine indirekte Kommunikation über ein Dateisystem zwischen der Anwendungsschicht und der Fehlerinjektionskampagne statt.

Die Darstellungsschicht dient als Vordergrund der Anwendung und hat eine Benutzeroberfläche, die über verschiedene Ansichten verfügt. Diese Ansichten spiegeln die Ergebnisse der Fehlerinjektionsexperimente wider. Weiterhin bietet die Benutzeroberfläche dem Anwender die Möglichkeit, Interaktivitäten durchzuführen, z. B. Navigation zwischen den Ansichten.

3.2.2 Anwendungsfälle

Das dargestellte Anwendungsfalldiagramm in Abbildung 3.2 beschreibt die möglichen Interaktionen, die von dem Anwender durchgeführt werden können, um die Anwendungsziele erreichen zu können bzw. Anforderungen zu erfüllen.

Das System bzw. die Anwendung ermöglicht dem Benutzer, zwischen unterschiedlichen Analyseverfahren zu wählen. Eine Schätzung wird bereitgestellt, falls die Fehlerinjektion nicht abgeschlossen ist. Diese Schätzung hängt von dem gewählten Analyseverfahren ab und ist nur für die folgende Verfahren vorhanden:

- Gesamtes Auftreten.
- Auftreten pro Symbol.

- Auftreten pro Funktion.
- Auftreten pro Übersetzungseinheit.

Während der Kampagnenlaufzeit soll das Priorisieren eines bestimmten Programmabschnitts möglich sein, das eine Funktion, ein Symbol, eine Übersetzungseinheit oder einen Fehlerraumbereich bevorzugt.

3.2.2.1 Analoge Anwendungsfälle

Bei den Anwendungsfällen **Gesamtes Auftreten, Auftreten pro Symbol, Auftreten pro Funktion, Auftreten pro Übersetzungseinheit und Dauer der Variante** handelt es sich um analoge clientseitige sowie serverseitige Vorgehensweisen. Der Unterschied liegt lediglich in den SQL-Anfragen, die zum Großteil aus den vorhandenen FAIL*-Tools übernommen worden sind.

Mit den oben genannten Anwendungsfällen wird wie folgt vorgegangen:

1. Wahl einer Analyse: Bei der Wahl einer Analyse wird die Sicht auf diese umgeschaltet, indem alle anderen Analysen versteckt werden und nur der Bereich dieser Analyse in den Vordergrund gezogen wird.

2. HTTP-Anfrage: Bevor die HTTP-Anfrage an den Server geschickt wird, wird der Datenfeldparameter dieser Anfrage bereitgestellt, indem neben Datenbank, Benchmark und Variante eine Aktion definiert werden muss.

3. Serververarbeitung: Nach Empfang der Client-Anfrage wird eine Reihe an Verarbeitungen durchgeführt:
   - Aktion Auswertung: Zuerst wird die empfangene Aktion mit den bereitgestellten Aktionen verglichen.
   - Parameterüberprüfung: Falls die empfangene Aktion mit einer der bereitgestellten Aktionen übereinstimmt, wird das Vorhandensein der dafür benötigten Parameter für die Ausführung der Anweisung überprüft.
   - Datenabfrage: Bei Erfüllung aller Bedingungen wird eine Funktion aufgerufen, die eine SQL-Anfrage an den Datenbankserver schickt, um Daten abzufragen.
   - Client Antwort: Anschließend wird das Ergebnis der SQL-Anfrage an den Client weitergeleitet.

4. HTTP-Antwort: Nach Empfang der Serverantwort wird das Ergebnis des Analyseverfahrens in dem dafür gedachten Bereich aktualisiert.

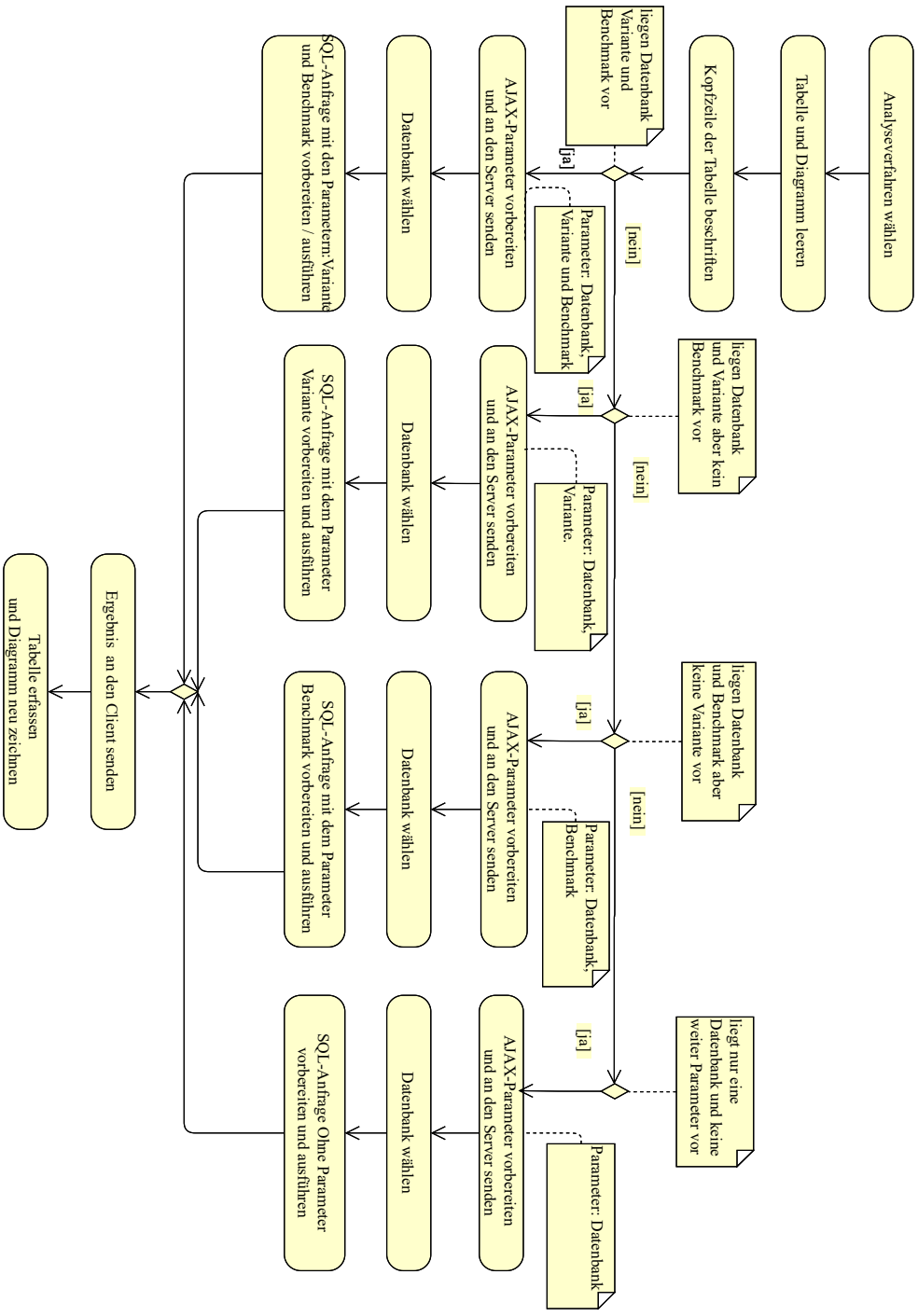

Abbildung 3.3: Ablaufdiagramm beschreibt die Abfolge von Verarbeitungen bei der Wahl einer Analyse.

3.2.2.2 Vergleich

Bei dem Anwendungsfall **Vergleich**, im Bezug auf eine vom Anwender gewählte Analyse, wird eine Fallunterscheidung (siehe Abbildung 3.3) zwischen den folgenden Fällen durchgeführt:

- Vergleich von Programmvarianten, die einen bestimmten Benchmark in der ausgewählten Datenbank enthalten.
- Vergleich von Benchmarks, die zu einer bestimmten Variante in der ausgewählten Datenbank gehören.
- Vergleich von allen Varianten und Benchmarks, die in der ausgewählten Datenbank vorliegen.
- Aufruf der Daten eines Analyseverfahrens für eine bestimmte Variante und einen bestimmten Benchmark in der ausgewählten Datenbank.

Für diesen Zweck kann der Benutzer neben der Datenbank eine Variante oder einen Benchmark wählen. Dabei führt die Wahl eines Benchmarks zum Varianten-Vergleich und die Wahl einer Variante zum Benchmarks-Vergleich, die zu dieser Variante gehören bzw. die dieses Benchmark enthalten. Außerdem kann der Anwender für einen allgemeinen oder globalen Vergleich nur eine Datenbank und eine Analyse wählen. Hierbei werden alle Benchmarks und Varianten bereitgestellt, die in dieser Datenbank verfügbar sind.

Für die Verarbeitung dieses Anwendungsfalls wird eine AJAX-Anfrage von dem Client an den Server geschickt, bei welcher der Client Analysedaten für die gewählte Analyse - anhand der verfügbaren Wahloptionen, die von dem Benutzer festgelegt sind - anfordert. Der Server fordert durch eine SQL-Anfrage mit den Parametern, der Variante, dem Benchmark oder ohne Parameter Daten von der Datenbank an und leitet diese an den Client weiter, woraufhin eine Ergebnisdarstellung erfolgt.

3.2.2.3 Fehlerraumplot

Beim Anwendungsfall **Fehlerraumplot** wird zuerst der Zustand der Kampagne abgefragt. Für diesen wird eine HTTP-Anfrage an den Server geschickt, um festzustellen, in welchem Zustand sich die Kampagne aktuell befindet.

Ist die Kampagne bereits fertig, wird wie folgt vorgegangen:

- Erste HTTP-Anfrage: Es wird eine erste Anfrage an den Server geschickt, die Zeitzyklen-Intervall und Adressen-Intervall anfordert.
- Initialisierung des Plots: Nach Empfang der Antwort auf die erste Anfrage werden die x-Achse und y-Achse initialisiert.

- Zweite HTTP-Anfrage: Nach der Initialisierung des Plots werden die zu zeichnenden Daten bzw. Bereiche angefordert. Hierfür wird eine HTTP-Anfrage an den Server geschickt. Auf der Serverseite wird eine SQL-Anfrage vorbereitet, die ebenfalls aus den vorhandenen FAIL*-Werkzeugen übernommen wurde, bei der u. a. die Zeitzyklen, die Adressen der Instruktionen sowie der Fehlerarttyp als kodierte Farbe für eine bestimmte Datenbank, Variante und einen bestimmten Benchmark zurückgegeben werden.

- Zeichnen des Plots: Nach Empfang der Antwort der zweiten Anfrage werden die Bereiche des Fehlerraumes gezeichnet.

Befindet sich die Kampagne derzeit im laufenden Zustand, wird wie folgt vorgegangen:

- Initialisierung des Plots: Die Initialisierung des Plots erfolgt analog zu der des anderen Falls.

- Zustandsanfrage: Es wird eine periodische HTTP-Anfrage an den Server geschickt, solange die Kampagne sich im laufendem Zustand befindet. Der Zweck dieser Anfrage besteht darin, den Zustand der Kampagne ständig zu überprüfen und damit bei einer Zustandsänderung den Server zu entlasten, indem keine weiteren HTTP-Anfragen erzeugt werden. Somit erfüllt sich die Abbruchbedingung der periodischen Anfrage.

- Datenanfrage: Anhand des Ergebnisses der Zustandsanfrage wird eine weitere HTTP-Anfrage erzeugt, um die aktuellen Ergebnisse der Fehlerinjektion für den Fehlerraumplot anzufordern. Diese HTTP-Anfrage wird auch periodisch in Abhängigkeit von der ersten Anfrage hergestellt. Wird die Abbruchbedingung der ersten Anfrage erfüllt, werden beide Anfragen nicht mehr erzeugt.

Das Zeichnen größerer Datenmengen kann zeitaufwendig werden. Um den Zeitaufwand zu reduzieren, werden die zu zeichnenden Daten komprimiert. Hierfür werden zwei Vorgehensweisen erprobt:

- **ServerKompact:** Ein serverseitiges Komprimierungsverfahren.

- **ClientKompact:** Ein clientseitiges Komprimierungsverfahren.

Das Zeichnen des Plots kann trotz Komprimierung viel Zeit in Anspruch nehmen, was in manchen Fällen zum Absturz des Browsers führen kann. Um diesen unerwünschten Effekt zu vermeiden, soll das Zeichnen des Fehlerraumes in kleine Partitionen erfolgen, wobei die angefragten Daten nach und nach gezeichnet werden.

Außerdem soll für die Zwischenergebnisse des Fehlerraumes keine ständige Aktualisierung des Plots im Sinne „Löschen" und „neu zeichnen" stattfinden, stattdessen soll nur die Differenz zwischen den bereits gezeichneten Fehlerraumbereichen und den neu dazukommenden Bereichen gezeichnet werden. Für diesen Zweck sollen zwei Vorgehensweisen untersucht werden:

- **ClientDiff:** Bei diesem Verfahren soll die Differenz anhand der gezeichneten Daten und neuen Daten gebildet werden.
- **LFDDiff:** Bei dieser Variante wird die Tabelle `result_GenericExperimentMessage` um eine laufende Nummer erweitert. Dabei wird anhand dieser laufenden Nummer die Differenz bereits bei der SQL-Anfrage gebildet, indem nur Daten ab einer bestimmten laufenden Nummer angefordert und nur diese zurückgegeben werden.

3.2.2.4 Schätzung und Zwischenergebnisse

Die Anwendungsfälle **Zwischenergebnisse und Schätzung** sind voneinander abhängig und werden beide gleich verarbeitet. Sie unterscheiden sich nur in dem Fehlerraumplot.

Bei diesem Vorgehen wird von dem Anwender erwartet, dass er eine Datenbank, eine Variante, einen Benchmark und eine Analyse wählt. Anhand der Datenbank, Variante und dem Benchmark wird der Zustand der Kampagne festgelegt. Auch hier werden, wie bei Fehlerraumplot, zwei HTTP-Anfragen - nämlich Zustandsanfrage und Datenanfrage - an den Server geschickt. Nach Antwort der Datenanfrage wird die Tabelle neu erfasst und das Diagramm neu gezeichnet.

Zwischen der Anfrage-Antwort-Zeit muss diese Anwendung auf die Benutzerinteraktionen reagieren, z. B. andere bereits geladene Analyseverfahren wählen. Dies erfordert eine asynchrone Verarbeitung, was durch AJAX realisierbar ist. Um den Server zu entlasten, wenn die Kampagne fertig ist, ist eine Überprüfung des Kampagnenzustandes nötig. Hierfür wird die Datenbankerweiterung nötig, die eine zusätzliche Tabelle beinhaltet, in welcher der Zustand der Kampagne gespeichert wird. Dieser wird bei der Initialisierung der Kampagnen auf „running" gesetzt und auf „done", wenn diese fertig ist.

3.2.2.5 Priorisieren

Bei dem Anwendungsfall **Priorisieren** soll dem Anwender die Möglichkeit geboten werden, bestimmte Programmteile bereits zur Laufzeit der Kampagne zu selektieren, um diese zu priorisieren. Dafür kann der Benutzer eine bestimmte Funktion, ein Symbol oder einen Bereich des Fehlerraums selektieren. Dabei initialisiert der Client eine Verbindung mit dem Namen der selektierten Funktion oder Übersetzungseinheit bzw. das Adressenintervall und Zeitzyklen-Intervall zum

Server. Der Server fordert die laufende Nummer (pilot_id) zu priorisierender Experimente mittels einer SQL-Anfrage von der Datenbank. Anschließend speichert er diese in einer Datei. Bei dem FAIL*-Jobserver findet eine ständige Überprüfung statt, ob zu priorisierende Experimente vorliegen. Falls ja, werden diese unmittelbar verarbeitet, indem sie am Anfang der Durchführungs-Warteschlange umsortiert werden, was dazu führt, dass die Ergebnisse der Fehlerinjektion von diesen Experimenten als nächstes vorhanden sein werden.

3.2.3 Fazit

In diesem Kapitel wurde nach der Durchführung der Ist-Zustandsanalyse die Anforderungen an die FAIL*-Erweiterung festgestellt, die im Rahmen dieser Arbeit zu realisieren ist. Weiterhin wurde der Entwurf der in dieser Arbeit entwickelten Erweiterung dargelegt. Anschließend wurden Anwendungsfälle erläutert und diverse Verfahren präsentiert. Im nächsten Kapitel werden die Entworfenen Ansätze realisiert.

4 Implementierung

In diesem Kapitel wird die Realisierung der entworfenen Anwendung beschrieben. Zuerst werden die verwendeten Technologien kurz beschrieben, danach wird die Implementierung der einzelnen Komponenten erläutert.

4.1 Verwendete Technologien

4.1.1 Bibliotheken

Data Driven Documents D3[1] ist eine JavaScript-Entwicklerbibliothek, die die Möglichkeit bietet, Daten als grafische Elemente darzustellen [2]. Hierbei handelt es sich um keine Chart-engine, bei der man fertige Layouts auswählt und konfiguriert, sondern um ein JavaScript-Framework, das dem Entwickler die Möglichkeit bietet, HTML-Dokumente auf der Basis von Daten zu manipulieren. D3 bindet Daten an das DOM, indem zuerst eine Selektion eines Bereiches an der Stelle durchführt wird, an der die Daten dargestellt werden sollen. In der Data Operator werden die Daten anschließend als ein Feld (engl.: Array) übergeben. Mithilfe der Enter-Selektion kann man Elemente erstellen.

jQuery[2] ist ein JavaScript-Framework [7], das die Möglichkeit bietet, auf die DOM-Elemente zuzugreifen und diese zu manipulieren. Zudem vereinfacht jQuery das Schreiben von JavaScript und die Wartung des Quellcodes. Um gezielt Elemente oder Elementgruppen anzusprechen, werden in jQuery Selektoren verwendet.

Für das Zeichnen der Diagramme wurde die Open-Source-Bibliothek **chart.js**[3] verwendet und für die Erstellung der Tabellen wurde die Bibliothek **Bootstrap Table** [4] eingesetzt.

4.2 Struktur und Komponenten

Tabelle 4.1 verschafft einen groben Überblick über die implementierten Komponenten.

[1] Data Driven Documents, https://d3js.org/
[2] https://jquery.com/
[3] https://www.chartjs.org/
[4] https://bootstrap-table.com/

```
1     <body>
2       <script>
3       // Daten bereitstellen
4       var data =  [{ x0: 50,
5                      y0: 50,
6                      x1: 55,
7                      y1: 55,
8                      fillcolor: "red"}];
9         // SVG-Element an das body-Element anhaengen
10        var svg = d3.select("body")
11              .append("svg")
12              .attr("width", 200)
13              .attr("height", 200);
14        // data als Rechteck an das SVG-Element anhaengen
15      svg.append("g")
16        .selectAll("rect")
17        .data(data)
18        .enter()
19        .append("svg:rect")
20        .attr("x", function (d) { return x(d.x0); })
21        .attr("y", function (d) { return y(d.y1); })
22        .attr("width", function(d) {
23          return Math.abs(x(d.x1)-x(d.x0));
24         })
25        .attr("height", function(d) {
26          return Math.abs(y(d.y1)-y(d.y0));
27         })
28        .attr("fill", function(d) {
29          return d.fillcolor;
30         });
31       </script>
32    </body>
33
```

Listing 4.1: Zeichnen eines Rechteckes mithilfe von D3. Zuerst wird ein SVG-Element an das body-Element angehängt, danach werden die Daten als Rechteck dem SVG-Element hinzugefügt.

Datei	*Schicht*	*Codezeilen*	*Dateityp*
compactfaultspaceplot_Live.js	Darstellungsschicht	694	JavaScript
function_Occurrences_Live.js	Darstellungsschicht	524	JavaScript
global_Occurrences_Live.js	Darstellungsschicht	478	JavaScript
symbol_Occurrences_Live.js	Darstellungsschicht	549	JavaScript
translation_Unit_Occurrences_Live.js	Darstellungsschicht	593	JavaScript
variation_Durations_Live.js	Darstellungsschicht	492	JavaScript
start.js	Darstellungsschicht	87	JavaScript
server.php	Anwendungsschicht	2076	PHP
connection.php	Anwendungsschicht	6	PHP
index.html	Darstellungsschicht	268	HTML
style.css	Darstellungsschicht	88	CSS

Tabelle 4.1: Überblick über die Komponenten

4.2.1 Darstellungsschicht

Die Darstellungsschicht besteht aus den folgenden Komponenten:

index.html: Ein HTML-Dokument, welches die Struktur des Inhaltes der Anwendung gliedert und benötigten Bereiche bzw. Elemente bereitstellt, die mit Identifikatoren versehen sind. Das Grundgerüst besteht aus 3 Hauptbereichen, wie in Abbildung 4.1 abgebildet ist:

- Ein Bereich für FAIL* Parameter, in dem der Anwender die Datenbank, Variante und den Benchmark wählen kann.
- Ein Bereich für die Auswahl des anzuzeigenden Analyseverfahrens, der dem Benutzer die Möglichkeit bietet, zwischen Analyseverfahren zu navigieren. Hierfür wird eine Navigationsleiste (engl.: Navigation bar) mit einer Liste an Analyseverfahren bereitgestellt, wobei jedes Verfahren ein id-Attribut hat. Diese dient dazu, eine Selektion zu ermöglichen, um die angeforderte Analyse durch Klicks zu identifizieren.
- Ein Bereich für die Anzeige, in der die Ergebnisse der Analyseverfahren integriert werden. Jede Analyse - außer Fehlerraumplot - besitzt jeweils einen Bereich für ein Diagramm und eine Tabelle, welche mit eindeutigen id- Attributen versehen sind. Die Analyse „Fehlerraumplot" besitzt nur einen Bereich für den Plot, der ebenfalls mit einem id-Attribut versehen ist. Diese Bereiche werden von den dafür gedachten Skripten verwaltet. Dabei wird entschieden, welche Tabellen, Diagramme oder Plot anhand Benutzerinteraktionen angezeigt werden.

index.html besteht aus 268 Zeilen.

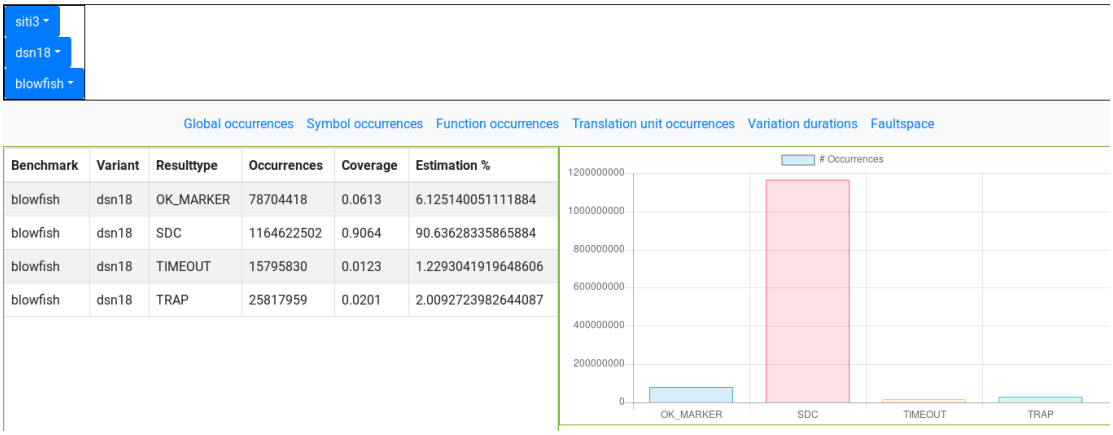

Abbildung 4.1: Der Grundaufbau für die Benutzeroberfläche. Er besteht aus mehreren Klapplisten, einer Navigationsleiste und einem Bereich für die Tabellen und Diagramme.

```
1  $(document).ready(function(){
2    var ajaxurl = 'server.php',
3    data = {'action' : "showDB"}; // Aktion festlegen
4    // POST-Anfrage an server.php mit der Aktion 'showDB' senden
5    $.post(ajaxurl, data, function (response){
6      // Antwort parsen
7      var result = JSON.parse(response);
8      // Die geparste Antwort durchlaufen, um die Klappliste zu
       erweitern.
9        for(i=0;i<result.length;i++){
10         $('#dbase').append('<a class="dropdown-item" href="#" >' +
       result[i] + '</a>');
11       }
12   });
13 });
14
```

Listing 4.2: Abfrage der verfügbaren Datenbanken. Es wird eine HTTP-POST-Anfrage an den Server („server.php") mit der Aktion „showDB" geschickt. Nach Empfang der Serverantwort wird die Klappliste erweitert.

style.css: Ein CSS-Datei, die für das Aussehen sorgt. Sie besteht aus 88 Zeilen.

start.js: Eine in JavaScript implementierte Komponente, die aus 87 Codezeilen besteht und der Bereitstellung der benötigten Parameter dient. Beim Laden der Seite wird das Skript **start.js** die verfügbaren Datenbanken in einer Klappliste (engl.: Dropdown) zur Auswahl stellen. Nach der Wahl einer Datenbank werden die in ihr vorhandenen Benchmarks bzw. Varianten in eine dafür gedachte Klappliste geladen. Listing 4.2 zeigt eine HTTP- POST-Anfrage mit der Aktion „showDB", bei der die verfügbaren Datenbanken abgefragt werden. Das Ergebnis dieser Anfrage wird der Klappliste hinzugefügt.

compactfaultspaceplot_Live.js: Stellt den Fehlerraumplot dar und besteht aus 694 Codezeilen.

Die folgenden Komponenten sind in JavaScript implementiert und dienen u. a. der Integration und dem Vergleich der vorhandenen Analyseverfahren:

global_Occurrences_Live.js: Setzt die Integration und Vergleich der vorhandenen Ergebnisse für den Anwendungsfall „Globales Auftreten" um und besteht aus 478 Codezeilen.

function_Occurrences_Live.js: Setzt die Integration und Vergleich der vorhandenen Ergebnisse für den Anwendungsfall „Auftreten pro Funktion" um und besteht aus 524 Codezeilen.

symbol_Occurrences_Live.js: Setzt die Integration und Vergleich der vorhandenen Ergebnisse für den Anwendungsfall „Auftreten pro Symbol" um und besteht aus 549 Codezeilen.

translation_Unit_Occurrences_Live.js: Setzt die Integration und Vergleich der vorhandenen Ergebnisse für den Anwendungsfall „Auftreten pro Übersetzungseinheit" um und besteht aus 593 Codezeilen.

variation_Duration_Live.js: Setzt die Integration der vorhandenen Ergebnisse für den Anwendungsfall „Dauer der Variante" um und besteht aus 492 Codezeilen

Mithilfe der `on()` Methode bindet jedes Skript genau ein Element aus der Navigationsleiste an ein Klick-Event, und jedes Element aus der Navigationsleiste entspricht einem Analyseverfahren. Beim Eintreten eines Klick-Ereignisses auf ein Element aus der Navigationsleiste wird in dem dafür gedachten Skript eine Funktion ausgeführt. Währenddessen wird jeder Parameter mithilfe des jQuery-$-Selektors ausgelesen und in einer Variable gespeichert. Danach findet die Überprüfung statt, bei der jeder Wert mit dem initialen Wert verglichen wird, um eine

```
1  ...
2  // Der Benutzer waehlt die Analyse function occurrences
3  $("body").on('click', '#functionOccurrences', function () {
4    // Der Bereich mit der id 'divTabelFunctionOccurrences' in den
     Vordergrund ziehen.
5    $('#divTabelFunctionOccurrences').show();
6    ...
7  });
8
9
```

Listing 4.3: Klick-Event auf das Element mit der id „functionOccurrences" aus der Navigationsleiste.

Aktion (engl.: action) zu definieren. Diese wird neben den FAIL*-Parametern an den Server übermittelt und sorgt dafür, welche Funktion in dem Server ausgeführt werden muss und somit auch, welche SQL-Anfrage ausgelöst wird. Der Wert der Aktion richtet sich zum Einen nach der gewählten Analyse und zum Anderen nach dem Vorhandensein des FAIL*-Parameters.

Listing 4.3 zeigt einen Code-Ausschnitt, der das Element mit dem id-Attribut „functionOccurrences" mit einem Klick-Ereignis bindet. Beim Klicken auf dieses Element wird die dafür implementierte Rückgabe-Funktion aufgeführt, in der alle anderen Bereiche versteckt werden und der Bereich mit dem id-Attribut „divTabelFunctionOccurrences" angezeigt wird.

4.2.2 Anwendungsschicht

Die Anwendungsschicht besteht aus 2 Dateien, die in PHP implementiert sind:

connection.php: Besteht aus 6 Codezeilen und dient dem Verbindungsaufbau mit dem Datenbankserver.

server.php: Umfasst die gesamte serverseitige Verarbeitung und besteht aus 2076 Codezeilen. Diese Komponente hat die Hauptaufgabe, auf clientseitige HTTP-Anfragen zu reagieren und die von dem Client angeforderten Daten von der Datenbank bereitzustellen. Dabei wird bei jeder empfangenen HTTP-POST-Anfrage geprüft, welche Aktion der Client fordert und ob alle dafür benötigten Parameter vorliegen. Sind alle Bedingungen erfüllt, wird für die empfangene Aktion die entsprechende Funktion aufgerufen. In diesen Funktionen wird meistens jeweils eine SQL-Anfrage an den Datenbankserver geschickt. Nach der Antwort des Datenbankservers wird das Ergebnis der Anfrage in einem zweidimensionalen Feld (engl.: Array) gespeichert und als JSON-Format an den Client zurückgegeben. Listing 4.4 beschreibt die Definition der Funktion `showSymbolOccurrences`, wobei die Wahl der Datenbank

```
1  function showSymbolOccurrences($bench,$db,$variant){
2    global $mysqli;
3    // Datenbank waehlen.
4    mysqli_select_db($mysqli ,$db);
5    ...
6    // SQL-Anfrage wird vorbereitet und geschickt.
7    ...
8    $array = array();// Feld dient der Rueckgabe
9    $sum = 0;
10   // Ergebnis der SQL-Anfrage durchlaufen.
11   while($row =$result->fetch_assoc())
12   { // die einzelnen spalten bilden in dem Hilfsfeld speichern
13     $subArray['benchmark']=$row["benchmark"];
14     $subArray['variant']=$row["variant"];
15     $subArray['resulttype']=$row["resulttype"];
16     $subArray['occurrences']=$row["occurrences"];
17     $subArray['address']=$row["address"];
18     $subArray['name']=$row["name"];
19     $sum = $sum + $row["occurrences"];
20     // Das Hilfsfeld an das Rueckgabe-Feld haengen
21     array_push($array,$subArray);
22   }
23   // Schaetzung berechnen
24   for ($i=0; $i < count($array); $i++){
25     $array[$i]['estimation'] = (100 * $array[$i]['occurrences'])
     / $sum;
26   }
27   // Antwort an den Client als JSON senden
28   echo json_encode($array);
29 }
30
```

Listing 4.4: Definition der Funktion showSymbolOccurrences. Es wird zuerst eine Datenbankselektion erfolgen, darauffolgend eine SQL-Anfrage an den Datenbankserver geschickt. Das Ergebnis der Anfrage wird in einem Feld gespeichert und an den Client als JSON weitergeleitet.

zuerst erfolgt, gefolgt von einer SQL-Anfrage. Daraufhin wird das Ergebnis dieser Anfrage in einem zweidimensionalen Feld gespeichert und anschließend an den Client weitergeleitet.

4.3 Zwischenergebnisse und Schätzung

Für diesen Zweck werden mehrere Skripte bereitgestellt, die die entworfenen Anwendungsfälle umsetzen und auf Benutzerinteraktionen reagieren.

4.3.1 Frontend

Die Implementierung von Zwischenergebnissen und Schätzung wird einen ähnlichen Verlauf nehmen. Dabei löst das Klicken eines Elementes aus der Navigationsleiste eine Reihe an Verarbeitungen aus:

1. Zunächst wird für das ausgewählte Element der dafür gedachte Bereich in den Vordergrund gezogen, alle anderen Bereiche werden versteckt.

2. Um das mehrfache Versenden von HTTP-POST-Anfragen zu vermeiden, wird zunächst überprüft, ob ein Element zuvor bereits in der Vergangenheit ausgewählt würde. Ist dies der Fall, werden keine überflüssigen Anfragen versendet. Anderenfalls würde der Client für jeden durchgeführten Klick entsprechend HTTP-POST-Anfragen erzeugen und folglich viele Antworten erhalten.

3. Danach werden die vom Benutzer gewählten FAIL*-Parameter mithilfe von jQuery-$-Selektor ausgelesen und in dafür gedachte Variablen gespeichert.

4. Überprüfung des Kampagnenzustandes: Die Funktion `checkCampaign` versendet eine HTTP-POST-Anfrage an den Server, um den Kampagnenzustand abzufragen. Ist die Kampagne im laufenden Zustand, wird „running" zurückgegeben und die Daten müssen mithilfe einer Funktion (Listing 4.5 wird pollGlobals aufgerufen.) von der Datenbank abgefragt werden. Anschließend wird anhand eines Timers nach 2,5 Sekunden die Funktion `checkCampaign` aufgerufen.

5. Parsen der Serverantwort in einer JSON-Datenstruktur.

6. Anschließend wird der Tabelleninhalt anhand des geparsten Ergebnisses der Anfrage erfasst.

7. Zum Schluss werden vier Funktionen aufgerufen, die das Diagramm zeichnen sowie die mit ihm gebundenen Labels erstellen. Diese bieten dem Benutzer

```
1
2   var dbase,benchmark,variant,campaignChecked;
3   $("body").on('click', '#globalOccurrences', function () {
4     // show(); und hide();
5     ...
6     dbase = $('#databaseBtn').text();
7     benchmark = $('#benchBtn').text();
8     variant = $('#vairantBtn').text();
9     // Pruefen, ob fuer diese Analyse die Funktion 'checkCampaign'
      bereits aufgerufen wurde
10    //-> falls nicht aufrufen und vermerken als aufgerufen.
11    if(!campaignChecked){
12      checkCampaign();
13      campaignChecked = true;
14    }
15    // Falls die Kampagne fertig ist
16    //-> Daten abfragen.
17    if(completed){
18      pollGlobals();
19    }
20      ...
21    // checkCampaign wird alle 2,5 s aufgerufen,
22    //solange das Ergebnis der Anfrage mit der Aktion 'checkState'
      gleich 'running' ist
23    function checkCampaign() {
24      var resp;
25      var data = {'action' : "checkState", 'benchmark' : str, '
     database' : dbase, 'variant' :vart}
26      $.post(ajaxurl, data, function (response){
27        resp = JSON.parse(response);
28        if(resp == "running"){
29          pollGlobals();
30          setTimeout(checkCampaign,2500);
31        }
32        else{
33          completed = true;
34          pollGlobals();
35        }
36      });
37    }
38  });
39
```

Listing 4.5: Periodische Überprüfung des Kampagnenzustandes. Zuerst wird überprüft, ob die Funktion „checkCampaign" bereits aufgerufen wurde. Ist dies nicht der Fall, wird sie aufgerufen. Bei der Funktion „checkCampaign" wird ständig eine HTTP-POST-Anfrage an den Server geschickt, um den Kampagnenzustand abzufragen. Ist die Kampagne im laufenden Zustand, wird die Funktion „pollGlobals" aufgerufen und nach 2500 Millisekunden wird die Funktion „checkCampaign" aufgerufen.

```
1  function checkState($bench,$db,$variant){
2    ...
3    $sql ="SELECT c.state
4      FROM campaign_State c
5      INNER JOIN variant v
6      ON v.id = c.variant_id
7      WHERE v.variant = '$variant' AND v.benchmark ='$bench'
8      ";
9    // sql-Statement ausfuehren
10   $state_array = array();
11   if($row =$result->fetch_assoc())
12   { //benchmarks in bench_array an clien send
13     array_push($state_array,$row["state"]);
14   }
15   echo json_encode($state_array);
16   exit;
17 }
```

Listing 4.6: Definition der Funktion „checkState". Es wird in der Tabelle „campaign_State" nach dem aktuellen Zustand der Kampagne für eine bestimmte Variante und einen bestimmten Benchmark abgefragt.

die Möglichkeit, bestimmte Diagrammbalken in den Vordergrund zu ziehen, indem andere durch Klicken versteckt werden.

4.3.2 Backend

Auch bei der serverseitigen Verarbeitung handelt es sich um eine analoge Vorgehensweise. Nach Erhalt der ersten HTTP-POST-Anfrage findet eine Überprüfung der angeforderten Aktion sowie des Vorhandenseins der benötigten Parametern für diese Anfrage statt.

Fordert der Client den Kampagnenzustand mit der Aktion „checkState" an, wird anhand dieser Aktion die Funktion `checkState` aufgerufen, bei der eine SQL-Anfrage mit den Parametern Benchmark und Variante an den Datenbankserver geschickt wird. Bei diesem Vorgang wird der Zustand aus der Tabelle state_campagn abgefragt und anschließend an den Client zurückgegeben, wie in Listing 4.6 zu sehen ist.

Fordert der Client ein Analyseverfahren, so wird dieses anhand der empfangenen Aktion festgestellt. Dann wird die dafür implementierte Funktion aufgerufen, bei der eine SQL-Anfrage vorbereitet und anschließend an den Datenbankserver geschickt wird. Nach Erhalt der Antwort wird ein zweidimensionales Feld angelegt, in welchem das Ergebnis der Anfrage gespeichert und an den Client übermittelt wird.

4.4 Fehlerraumplot

4.4.1 Komprimierung

Für die Komprimierung der Daten des Fehlerraumplots wurden in Abschnitt 3.2 zwei Alternativen vorgestellt: Zum Einen **ClientKompact** und zum Anderen **ServerKompact**. Bei den vorgestellten Vorgehensweisen handelt sich um ähnliche Vorgehensweisen. Demzufolge wird **ClientKompact** clientseitig in der Datei „compactfaultspaceplot_Live.js" implementiert, indem jeweils eine vertikale und horizontale Komprimierung durchgeführt werden. Für die vertikale Komprimierung werden die empfangenen Daten, also Fehlerraumbereiche, die als Rechtecke dargestellt werden, wie folgt behandelt:

1. Zunächst werden die empfangenen Rechtecke nach X0-, Y0- und Y1-Koordinaten sortiert.

2. Die benachbarten Rechtecke, die dieselbe Farbe und Y0- und Y1-Koordinaten haben, werden zusammengefügt, indem die Rechtecke mit den kleinen X0-Koordinaten mithilfe der Methode `delete()` gelöscht werden und deren direkte Nachbarn erweitert werden.

3. Das Feld wird erneut indiziert.

Nach der Durchführung der vertikalen Komprimierung wird die horizontale Komprimierung durchgeführt. Dafür werden die Daten erneut nach Y0-, X0- und X1-Koordinaten sortiert und die benachbarten Rechtecke mit derselben Höhe, Farbe zusammengefügt.

Bei der serverseitigen Variante **ServerKompact** wird dasselbe Verfahren angewendet, wie bei **ClientKompact**.

Listing 4.7 zeigt einen Abschnitt aus der Funktion `compactRect`, die in der Datei „server.php" implementiert ist, bei der eine vertikale, serverseitige, Komprimierung durchgeführt wird. Die Funktion `unset()` löscht ein Element aus einem Feld, ohne das Feld erneut zu indizieren, und mithilfe der Funktion `array_values()` wird das Feld erneut indiziert.

4.4.2 Differenzbildung

Für das Zeichnen des Fehlerraumplots während der Kampagnenlaufzeit soll eine Differenz zwischen gezeichneten Rechtecken und neu dazukommenden Rechtecken gebildet werden. Für diesen Zweck wurde in Abschnitt 3.2 zwei Verfahren präsentiert:

ClientDiff: Bei diesem Verfahren fordert der Client unkomprimierte Daten von dem Server an und verarbeitet diese wie folgt:

```
1  function compactRect($rectArray){
2    while (true)
3    { $countHorizontal=0;
4      for($i = 1; $i< count($rectArray) ; $i++){
5        if($rectArray[$i-1]['fillcolor'] == $rectArray[$i]['
     fillcolor']
6          &&$rectArray[$i-1]['x1'] == $rectArray[$i]['x0']
7          &&$rectArray[$i-1]['y1'] == $rectArray[$i]['y1']
8          &&$rectArray[$i-1]['y0'] == $rectArray[$i]['y0'] )
9        {
10         // x0 von dem aktuellen Rechteck auf x0 von dem vorherigen
     setzen
11         $rectArray[$i]['x0'] = $rectArray[$i-1]['x0'];
12         $rectArray[$i]['id'] = max($rectArray[$i],$rectArray[$i
    -1]);
13         unset($rectArray[$i-1]);
14           $countHorizontal++ ;
15       }
16     }
17     $rectArray = array_values($rectArray);// erneut indizieren
18     if($countHorizontal ==0){
19       break;
20     }
21   }
22 }
```

Listing 4.7: Abschnitt aus der Funktion „compactRect". Es zeigt den Ablauf einer horizontalen Komprimierung.

```
1  function pollDataToPlot(){
2    ...
3    var dat = {'action' : "plot", 'benchmark' : str, 'database' :
       dbase, 'variant' :vart, 'fromId' : fromID }
4    $.post(ajaxurl, dat, function (response){
5      dataIn = [];// dataIn fuer die Differenz
6      dataIn = JSON.parse(response);
7      //Aktualisierung der laufenden Nummer
8      for (i=0; i< dataIn.length; i++){
9        if(fromID<parseInt(dataIn[i].id)){
10         fromID = parseInt(dataIn[i].id);
11       }
12     }
13     data = data.concat(dataIn);// data beinhaltet alles
14     drawRectangles(scatter, data, partition_length);
15   }
16   ...
17 }
```

Listing 4.8: Definition der Funktion „pollPlotData". Die Funktion fordert Daten für das Zeichnen an.

1. Differenzbildung: Bei diesem Schritt werden die unkomprimierten Daten aus der vorherigen Anfrage mit den neu dazukommenden Daten verglichen und daraus wird die Differenz gebildet.

2. Aktualisierung der Gesamtdaten: Hierfür werden die Daten aus der vorherigen Anfrage durch die Daten der aktuellen Anfrage ersetzt.

3. Danach wird die Differenz mithilfe von **ClientKompact** komprimiert.

4. Anschließend wird die Differenz mithilfe der Funktion `drawRectangles` gezeichnet.

LFDDiff: Bei dieser Variante wird nur die Differenz - anhand der laufenden Nummer - zwischen der letzten Anfrage und der aktuell in der Datenbank vorhandenen Rechtecke abgefragt, indem bei der Funktion `pollPlotData` das Datenfeld der HTTP-POST-Anfrage um eine weitere Variable „fromId" erweitert wird, wie in Listing 4.8 zu sehen ist. Diese Variable hat bei der ersten Anfrage den Wert 0 und wird bei jeder Antwort aktualisiert.

```
1  $('#tableFunktion ').on('dblclick','tbody tr', function () {
2    // Name der Funktion anhand des Indizes und des Feldes 'name'
3    nameToPrio = result[$(this).data().index].name ;
4    dbase = $('#databaseBtn').text();
5    benchmark = $('#benchBtn').text();
6    variant = $('#vairantBtn').text();
7    var ajaxurl = 'server.php';
8    //Datenfeld fuer die POST-Anfrage vorbereiten
9    var data = {'action' : "prioritizeFromFunction", 'benchmark' :
      benchmark, 'database' : dbase, 'variant' :variant,
10     'name' : nameToPrio
11      }
12   //HTTP-POST-Anfrage an server.php senden
13   $.post(ajaxurl, data);
14 });
15
```

Listing 4.9: Das Priorisieren einer Funktion. Zuerst wird der Name der Funktion in einer Variable gespeichert und zum Schluss an den „server.php" versendet.

4.5 Priorisieren

4.5.1 Frontend

Das Priorisieren wird clientseitig entweder durch Doppelklick auf das zu priorisierende Element oder durch eine Selektion eines Bereiches des Fehlerraumplots ausgelöst. Für diesen Zweck wird die Ansicht des Fehlerraumplots erweitert. Diese Erweiterung bringt weitere Interaktionmöglichkeiten (etwa Bereich-Selektion und Bereich-Priorisierung) mit sich. Dafür werden zusätzliche Knöpfe in die Hauptseite eingebaut und vom Skript „compactfaultspaceplot_Live.js" mit Klick-Ereignissen gebunden.

Die Skripte „symbol_Ocurrences_live.js", „function_Occurrences_live.js" und „translation_Unit_Occurrences_live.js" binden die dazugehörigen Tabellen mit Klick-Ereignissen. Bei einem Doppelklick auf eine Zeile der Tabelle wird der Name der Funktion oder des Symbols oder der Pfad der Übersetzungseinheit in einer Variable gespeichert. Diese wird neben der Datenbank, der Variante, dem Benchmark und der Aktion an den Server mithilfe einer HTTP-POST-Anfrage übergeben. Der Wert der Aktion richtet sich nach der Ansicht bzw. nach dem Skript. Die Serverantwort wird in diesem Fall ignoriert, da keine direkte Rückgabe erfolgt, sondern eine weitere Serververarbeitung. Listing 4.9 zeigt das Priorisieren einer Funktion, bei welcher zuerst der Name ausgelesen wird und in einer Variable gelagert wird. Diese Variable wird neben der Aktion und den anderen FAIL*-Parametern an den

```
1  function prioritizeFromFunction ($bench,$db,$variant,$name){
2    global $mysqli;
3    mysqli_select_db($mysqli ,$db);
4    $sql = "SELECT g.pilot_id
5      FROM variant v
6      JOIN symbol s
7        ON s.variant_id = v.id
8      JOIN trace t
9        ON t.variant_id = s.variant_id AND t.instr2_absolute BETWEEN
       s.address AND s.address + s.size - 1
10     JOIN fspgroup g
11       ON g.variant_id = t.variant_id AND g.data_address = t.
      data_address AND g.instr2 = t.instr2
12       AND g.fspmethod_id = (SELECT id FROM fspmethod WHERE method =
       'basic')
13     JOIN result_GenericExperimentMessage r
14       ON r.pilot_id = g.pilot_id
15     WHERE v.benchmark = '$bench' AND v.variant = '$variant'
16       AND s.name= '$name';";
17   if (!($result=mysqli_query($mysqli,$sql))) {
18     printf("Error: %s\n", mysqli_error($mysqli));
19   }
20   $file = 'pilot.txt';
21   while($row =$result->fetch_assoc()){
22     $txt = $row['pilot_id'] . "\n";
23     file_put_contents($file, $txt, FILE_APPEND | LOCK_EX);
24   }
25 }
```

Listing 4.10: Definition der Funktion „prioritizeFromFunction". Es wird ein SQL-Statement vorbereitet und ausgeführt. Das Ergebnis dieser SQL-Anfrage wird in der Datei „pilot.txt" gespeichert.

Server weitergeleitet. Die HTTP-POST-Anfrage erwartet keine Rückgabefunktion.

4.5.2 Backend

Bei der serverseitigen Verarbeitung des Priorisierens wird neben der Aktion eine Variante, ein Benchmark und Name einer Funktion, eines Symbols oder ein Pfad einer Übersetzungseinheit erwartet. Mithilfe des Aktionswertes wird entschieden, welche Funktion aufgerufen wird, bei denen eine SQL-Anfrage vorbereitet und versendet wird. Diese Anfrage selektiert die pilot_ids, die zu der empfangenen Variante und dem Benchmark gehören. Daraufhin werden diese pilot_ids in einer Datei gespeichert, wie in Listing 4.10 zu sehen ist.

In dem Kampagnen-Server wird während der Laufzeit der Kampagne ständig überprüft, ob die zu priorisierende Datei vorliegt. Liegt diese vor, werden die in ihr liegenden pilot_ids eingelesen und in einer Datenstruktur gespeichert und anschließend in der Vorzugsliste befüllt.

4.6 Fazit

In diesem Kapitel wurde die Umsetzung der in Kapitel 3 entworfenen FAIL*-Erweiterung dargelegt. Demzufolge wurden die verwendeten Technologien zuerst beschrieben, danach wurden die implementierten Komponenten der in Abschnitt 3.2 vorgestellten Architektur zugeordnet und anschließend die wichtigen Punkte der Implementierung vorgestellt. Im folgenden Kapitel werden die implementierten Verfahren evaluiert.

5 Evaluation

Dieses Kapitel befasst sich mit der Evaluation der im Abschnitt 3.2.2.3 beschriebenen Algorithmen. Zuerst wird der Aufbau der Testumgebung für die Evaluation beschrieben, anschließend werden die Ergebnisse der einzelnen Algorithmen repräsentiert und verglichen.

5.1 Aufbau der Testumgebung

Zur Evaluation der implementierten Verfahren wurden diverse Benchmarks eingesetzt, die unterschiedliche Fehlerinjektionslaufzeiten haben. Somit liegt eine unterschiedliche Anzahl an Fehlerinjektionsexperimenten vor. Der Benchmark „bubblesort" wurde mehrmals mit unterschiedlichen Eingabemengen verwendet. Der Speicherverbrauch wurde mithilfe von Task Manager von Google Chrome gemessen.

5.1.1 Komprimierung

5.1.1.1 ServerKompact

Für die Evaluation dieser Vorgehensweise wurde eine Anpassung durchgeführt:

- **Serverseitig:** Bei dem Server wurde der Quellcode angepasst, sodass - bevor die Daten an den Client übermittelt werden - eine serverseitige Komprimierung durchgeführt wird. Die Komprimierung wurde, wie in Unterabschnitt 4.4.1 beschrieben ist, in der Funktion `ServerKompact` implementiert und wird nach Bereitstellung der zu komprimierenden Daten aufgerufen. Bevor die Funktion `ServerKompact` aufgerufen wurde, wurde mithilfe der Funktion `microtime` eine Zeitmessung gestartet, die erst nach der kompletten Ausführung der Funktion `ServerKompact` beendet wurde. Außerdem wurde die Datengröße vor und nach der Komprimierung gemerkt und in einer Datei, neben der gemessenen Zeit, gespeichert.

- **Clientseitig:** Bei dem Client wurde der Quellcode auskommentiert, der für die clientseitige Komprimierung sorgt.

5.1.1.2 ClientKompact

Für die Evaluation dieses Vorgangs fand eine Quellcodeanpassung auf zwei Ebenen statt:

- **Serverseitig:** Bei dem Server bestand die Anpassung darin, dass die angefragten Daten direkt an den Client übertragen wurden.
- **Clientseitig:** Bei dem Client wurde nach Empfang der Daten eine Funktion aufgerufen, bei welcher die Komprimierung stattfand. Bevor jedoch diese Funktion aufgerufen wurde, wurde die unkomprimierte Datengröße in einer Tabelle erfasst und danach eine Zeitmessung gestartet. Nach der Komprimierung wurden die komprimierte Datengröße sowie die Zeit der Komprimierung in einer Tabelle erfasst.

5.1.2 Differenzbildung

5.1.2.1 LFDDiff

Die Evaluation von **LFDDiff** wurde während der Laufzeit der Fehlerinjektionskampagne durchgeführt. Für diesen Zweck wurde eine Quellcodeanpassung durchgeführt:

- **Serverseitig:** Bei dem Server wurde unmittelbar vor der SQL-Anfrage eine Zeitmessung gestartet, bei welcher die Zeit der SQL-Anfragen und der serverseitigen Datenverarbeitung gemessen wurde. Neben der gemessenen Zeit wurden die komprimierte und unkomprimierte Datengröße in einer Datei gepeichert.
- **Clientseitig:** Bei dem Client wurde bei jedem Zeichnungsvorgang des Fehlerraumes eine Zeitmessung vorgenommen.

5.1.2.2 ClientDiff

Das Ziel dieser Variante war, die Differenz zwischen den alten gezeichneten Daten und den neuen dazukommenden Daten zu bilden. Die Evaluation von **ClientDiff** erfolgte zur Laufzeit der Fehlerinjektionskampagne. Dabei wurde sowohl clientseitig als auch serverseitig der Quellcode angepasst. Die serverseitige Anpassung wurde von Abschnitt 5.1.2 übernommen. Bei dem Client wurden nach Empfang jeder Serverantwort die folgende Informationen in einer Tabelle zur Laufzeit erfasst:

- Zeit der Differenzbildung
- Größe aller unkomprimierten Daten, die bereits komprimiert gezeichnet wurden.

Benchmark	$D_{Original}$	D_{Komp}	T_{Komp}	*Speicherverbrauch*
basicmath	1013	142	6	47516
bubblesort	11443	3145	59	61452
qsort	23173	6863	139	79060
bitcount	105474	34387	1070	222400
bubblesort	111015	31213	926	205496
blowfish	218749	11864	1534	111392
bubblesort	313075	88399	2727	499816
bubblesort	2860695	811103	45334	3938916

Tabelle 5.1: Ergebnisse von **ServerKompact**

- Gesamtanzahl der neu angefragten Daten
- Zeit der Komprimierung der Differenz
- Größe der komprimierten gezeichneten Daten, die im Fehlerraumplot bereits gezeichnet wurden.
- Zeit des Zeichnens

5.2 Ergebnisse und Beobachtung

5.2.1 Komprimierung

5.2.1.1 ServerKompact

Tabelle 5.1 zeigt die Ergebnisse des Verfahrens **ServerKompact**, wobei T_{Komp} für die Zeit der Komprimierung in Millisekunden steht, während $D_{Original}$ für die Größe der unkomprimierten Daten und D_{Komp} für die Größe der komprimierten Daten steht. Bei etwa 100000 Datensätzen betrug die Zeit der Komprimierung knapp eine Sekunde und der Speicherverbrauch etwa 200000 KB, während bei etwa 2800000 Datensätzen die Dauer ca. die 45 Sekunden betrug und der Speicherverbrauch knapp 4000000 KB.

5.2.1.2 ClientKompact

Tabelle 5.2 zeigt die Ergebnisse des Verfahrens **ClientKompact**, wobei T_{Komp} für die Zeit der Komprimierung in Millisekunden steht, während $D_{Original}$ für die Größe der unkomprimierten Daten und D_{Komp} für die Größe der komprimierten Daten steht. Die Zeit der Komprimierung hängt von der Größe der zu komprimierenden Daten ab. Bei einer Datengröße von etwa 313000 Elementen betrug die Zeit der Komprimierung etwa 0,5 Sekunden und der Speicherverbrauch etwa 500000 KB,

Benchmark	$D_{Original}$	D_{Komp}	T_{Komp}	*Speicherverbrauch*
basicmath	1013	142	5	118756
bubblesort	11443	3145	34	140628
qsort	23173	6863	85	148328
bitcount	105474	34387	186	275624
bubblesort	111015	31213	213	260712
blowfish	218749	11864	406	173292
bubblesort	313075	88399	506	525880
bubblesort	2860695	811103	4655	4270752

Tabelle 5.2: Ergebnisse von **ClientKompact**

D_{Origin}	T_{Diff}	D_{Diff}	$T_{DiffKomp}$	D_{Plot}	T_{Plot}	T_{Server}
66559.55	38.03	348.99	2.37	31206.78	281.20	1155.51
51746.20	29.8	295.3	1.61	24893	237.86	18185.31
230629	178	1046	23	109053	2649	20907
624	0	0	0	402	29	16426

Tabelle 5.3: Ergebnisse von **ClientDiff**

während bei 2860695 Datensätzen die Zeit der Komprimierung knapp 5 Sekunden betrug und der Speicherverbrauch 4270752 KB.

5.2.1.3 Vergleich der Komprimierungsvarianten

Wie erwartet ergaben die beiden Verfahren die gleichen Ergebnisse, was die Komprimierung der Daten angeht. Allerdings lag der Unterschied in der Zeit der Komprimierung. Dabei war die clientseitige Variante zum Vegleich mit der serverseitigen Komprimierung wesentlich schneller, insbesondere bei größeren Datenmengen, z. B. bei der Komprimierung von 2860695 Datensätzen lag der Unterschied bei etwa 40 Sekunden und bei 11443 Datensätzen war die clientseitige Variante nur 25 Millisekunden schneller. Es kam zu der Schlussfolgerung: Je größer die Datensätze sind, desto deutlicher war, dass die clientseitige Komprimierung schneller als die serverseitige Alternative war.

Ein Nachteil der clientseitigen Komprimierung besteht darin, dass die zu übertragende Datenmenge größer war, als die von der serverseitigen Variante, was zu einer Verzögerung bei der Übertragung führen kann.

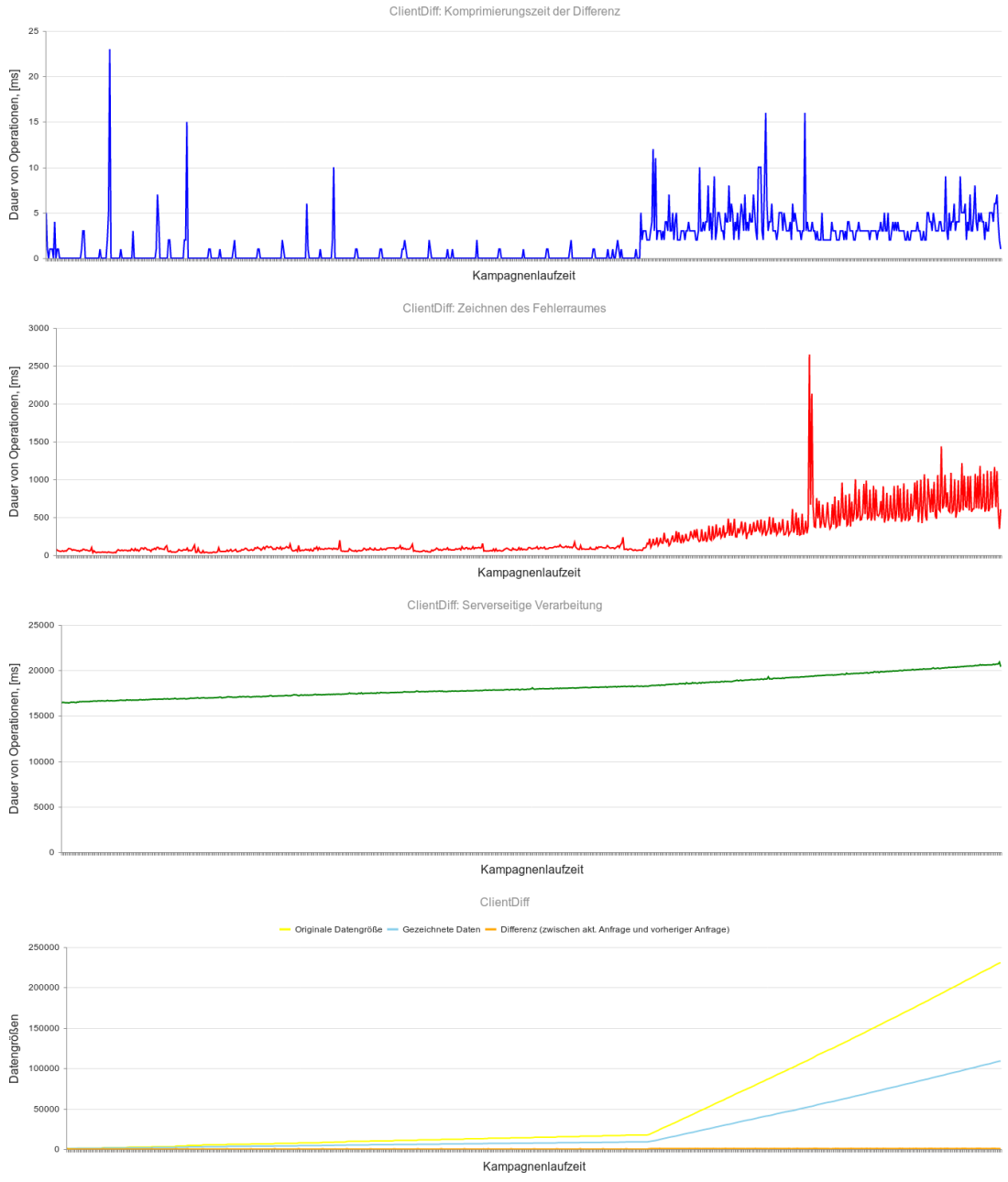

Abbildung 5.1: Veranschaulichung der Dauer von Operationen in Millisekunden und der Datengröße vor und nach der Komprimierung während der Kampagnenlaufzeit von der ClientDiff-Variante.

5.2.2 Differenzbildung

5.2.2.1 ClientDiff

Tabelle 5.3 zeigt die Ergebnisse der Evaluation von **ClientDiff**, wobei:

- D_{Origin} für die angefragte Datengröße steht
- T_{Diff} für die Zeit der Differenzbildung steht
- D_{Diff} für die Datengröße der Differenz aus der aktuellen Anfrage und der Datengröße aus der vorherigen Anfrage steht
- $T_{DiffKomp}$ für die Komprimierungszeit der Differenz steht
- D_{Plot} für die Datengröße steht, die bereits gezeichnet ist
- T_{Plot} für die Zeit des Zeichnens des Fehlerraumes steht
- T_{Server} für die serverseitige Verarbeitung steht

Abbildung 5.1 stellt vier Diagramme dar, die während der Kampagnenlaufzeit und des Zeichnens des Fehlerraumplots erfasst wurden. Für diesen Zweck wurde der Benchmark „dijkstra" eingesetzt. Während der Laufzeit der Kampagne wurden 781 AJAX-Anfragen verarbeitet. Die ersten drei Diagramme beschreiben die Dauer von Operationen und das vierte Diagramm die Datengröße während der Kampagnenlaufzeit. Mithilfe der Tabelle 5.3 lassen sich die Diagramme besser interpretieren:

- Das erste Diagramm beschreibt die Komprimierungszeit der Differenz, bei dem die x-Achse die Kampagnenlaufzeit und y-Achse die Dauer der Komprimierung darstellt. Dabei betrug die maximale Dauer 23 Millisekunden und minimale Dauer 0 Sekunden, was dadurch begründet sein könnte, dass die Datendifferenz zweier sequentieller AJAX-Anfragen gering war und es manchmal gar keine gab. Außerdem betrug der Mittelwert 1,61 Millisekunden und die Standardabweichung 2,37 Millisekunden.
- Das zweite Diagramm zeigt die Zeit des Zeichnens des Fehlerraumplots während der Kampagnenlaufzeit. Die maximale Dauer lag bei 2649 Millisekunden und die minimale bei 29 Millisekunden, während die durchschnittliche Verarbeitungszeit 18185 Millisekunden betrug.
- Das dritte Diagramm stellt die serverseitige Verarbeitungszeit - SQL-Anfrage und Datenverarbeitung - dar. Die maximale Verarbeitungszeit betrug 20907

Maß	D_{Plot}	T_{Plot}	T_{Server}	D_{Origin}	T_{Komp}	D_{Komp}
Standardabweichung	53864.85	239.75	1324.34	92.76	0.63	51.36
Mittelwert	45904.18	302.41	2495.11	80.87	0.42	41.70
Maximum	186705	1428	16664	734	6	489
Minimum	489	24	2	0	0	0

Tabelle 5.4: Ergebnisse von **LFDDiff**

Millisekunden und die minimale 16426 Millisekunden während der Mittelwert 18185 Millisekunden betrug. Im Allgemeinen nahm die Verarbeitungszeit langsam zu, indem die minimale Dauer zu Beginn der Kampagnenlaufzeit und die maximale gegen Ende der Laufzeit der Kampagne erreicht wurde.

- Das vierte Diagramm stellt die Entwicklung der Datengröße vor und nach der Komprimierung sowie die Differenz zwischen Daten aus der aktuellen Anfrage und der vorherigen Anfrage in Abhängigkeit von der Kampagnenlaufzeit dar. Die gelbe Kurve beschreibt die angefragte Datengröße, die hellblaue Kurve beschreibt die komprimierte Datengröße und die orangefarbene Kurve die Datendifferenz zu der vorherigen Anfrage. Die gelbe und hellblaue Kurve nahmen am Anfang einen schwachen zunehmenden Verlauf an. Ab etwa 60% der Kampagnenlaufzeit stiegen diese linear an. Die Datendifferenz nahm 0 Datensätze als minimalen Wert und 1046 Datensätze als maximalen Wert an.

5.2.2.2 LFDDiff

Tabelle 5.4 zeigt die Ergebnisse der Evaluation von **LFDDiff**, wobei:

- T_{Komp} für die Zeit der Komprimierung steht
- T_{Plot} für die Zeit des Zeichnens des Fehlerraumes steht
- D_{Plot} für die Datengröße steht, die bereits gezeichnet ist
- T_{Server} für die Ausführungszeit der SQL-Anfrage und die serverseitige Datenverarbeitung steht
- D_{Origin} für die angefragte Datengröße steht
- D_{Komp} für die komprimierte Datengröße steht

Abbildung 5.2 und Tabelle 5.4 beschreiben die Dauer von Operation in Millisekunden und Datengröße während des Zeichnens des Fehlerraumplots zur Laufzeit

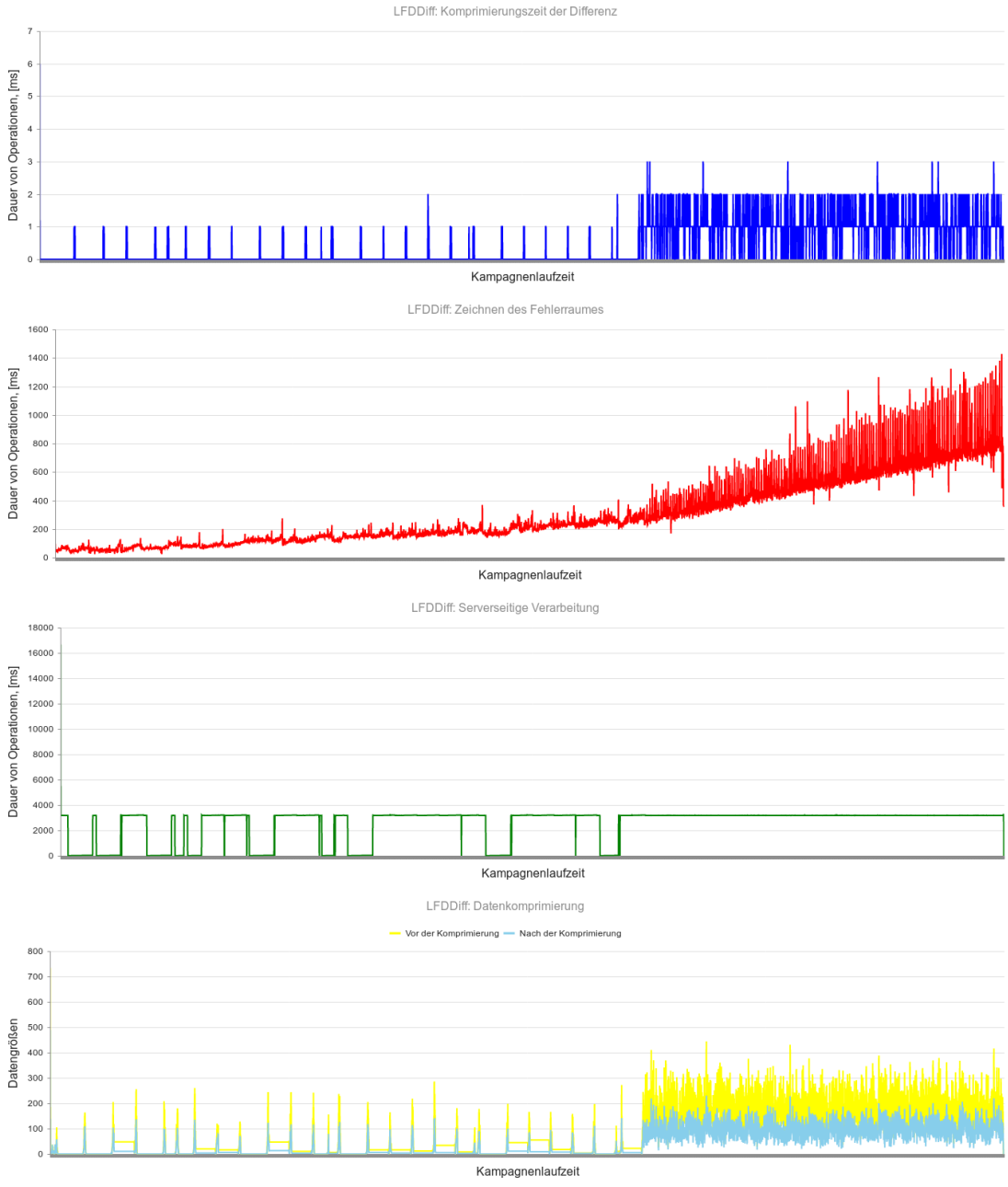

Abbildung 5.2: Veranschaulichung der Dauer von Operationen in Millisekunden und der Datengröße vor und nach der Komprimierung während der Kampagnenlaufzeit von der LFDDiff-Variante.

der Kampagne. Dabei wurde der Benchmark „dijkstra" als Zielprogramm eingesetzt. Während der Laufzeit der Kampagne wurden über 4000 AJAX-Anfragen verarbeitet. Abbildung 5.2 umfasst vier Diagramme, welche die folgende Informationen liefern:

- Das erste Diagramm zeigt die Dauer der Komprimierung der durch **LFDDiff** gebildete Differenz in Abhängigkeit von der Kampagnenlaufzeit bzw. AJAX-Anfragen in zeitlicher Reihenfolge. Die maximale Dauer der Komprimierung lag bei 6 Millisekunden und die minimale bei 0 Millisekunden während die Standardabweichung bei 0,63 Millisekunden und der Mittelwert bei 0,42 Millisekunden lag.
- Das zweite Diagramm beschreibt die Dauer des Zeichnens des Fehlerraumplots. Die Kurve nahm einen schwankenden Verlauf, was daran lag, dass die Datendifferenz zwischen zwei Anfragen immer unterschiedlich war. Im Allgemeinen ist zu sehen, dass die Kurve trotz der Schwankung einen wachsenden Verlauf nahm. Die minimale Dauer des Zeichnens lag bei 24 Millisekunden und die maximale Dauer bei 1428 Millisekunden während die durchschnittliche Dauer bei etwa 302 Millisekunden lag.
- Das dritte Diagramm stellt die Dauer serverseitige Verarbeitung während der Kampagnenlaufzeit dar. Die Kurve nahm einen schwankenden Verlauf zwischen 0 Millisekunden und etwa 3000 Millisekunden, bis auf die erste Anfrage, bei welcher die Dauer der serverseitigen Verarbeitung bei 16664 Millisekunden lag und somit als Maximum galt. Die durchschnittliche Dauer betrug 2495 Millisekunden.
- Das letzte Diagramm beschreibt die empfangene Datengröße vor und nach der Komprimierung. Die gelbe Kurve spiegelt die Datengröße vor der Komprimierung und die hellblaue nach der Komprimierung wieder. Vor der Komprimierung lag die maximale Datengröße bei 734 Datensätzen und nach der Komprimierung bei 489 Datensätzen, die minimale Datengröße betrug vor und nach der Komprimierung 0 Datensätze.

5.2.2.3 Vergleich der Differenzbildung-Alternativen

Im Vergleich zwischen **ClientDiff** und **LFDDiff** fällt auf, dass die Zeit der serverseitigen Verarbeitung bei der LFDDiff-Variante deutlich kleiner als die Zeit der ClientDiff-Variante war, was dadurch begründet sein könnte, dass sich die abgefragte Datenmenge bei der Einführung einer laufenden Nummer wesentlich reduziert hatte. Durch das Einsetzen von laufenden Nummern wurde die Differenz mit der SQL-Anfrage geliefert, was beim ClientDiff-Verfahren nicht der Fall war, bei welchem eine zusätzliche Differenzbildung stattfand, die mehr Variablen benötigte und mehr Zeit kostete.

Außerdem war zu beobachten, dass die Anzahl der Anfragen bei der ClientDiff-Variante 781 Anfragen und bei der LFDDiff-Variante über 4000 Anfragen betrug. Dieser Unterschied lässt sich dadurch erklären, dass bei dem ClientDiff-Verfahren die Dauer der SQL-Anfrage länger war als bei dem LFDDiff-Verfahren.

5.3 Fazit

Nach der Evaluation der vorgestellten Vorgehensweisen wurde im Rahmen dieser Arbeit das ServerKompact-Verfahren mit der LFDDiff-Variante kombiniert. Die serverseitige Komprimierung lieferte kompakte Datenmengen an den Client, während die clientseitige Komprimierung größere Datenmengen förderte. Die Differenzbildung durch die laufende Nummer leistete deutlich bessere Verarbeitungszeit.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Ziel dieser Arbeit war, eine FAIL*-Erweiterung zu entwickeln, sodass die unterschiedlichen Analyseverfahren in einer gemeinsamen Benutzeroberfläche integriert werden. Ein weiteres Ziel bestand darin, während der Laufzeit Zwischenergebnisse der bereits abgeschlossenen Fehlerinjektionsexperimente zu liefern und dem Benutzer eine Möglichkeit zu bieten, bestimmte Programmteile zu priorisieren. Die FAIL*-Erweiterung erfolgte in der Form einer Webanwendung.

Bei dem Entwurf wurde eine Client-Server-Architektur mit dem 3-Schichten-Modell präsentiert. Als Datenbank für die Datenbankschicht wurde die FAIL*-Datenbank übernommen. Die Darstellungsschicht stellte die Ergebnisse der unterschiedlichen Analyseverfahren dar, während die Anwendungsschicht Anfragen an die FAIL*-Datenbank schickte sowie mit FAIL* über ein Dateisystem kommunizierte.

Bei der Implementierung wurde die Umsetzung der entworfenen Anwendung realisiert. Für die Komprimierung wurden 2 Varianten erprobt: Zum Einen eine clientseitige Komprimierung und zum Anderen eine serverseitige Komprimierung. Außerdem wurden für das Zeichnen des Fehlerraumes während der Laufzeit der Kampagne 2 Alternativen untersucht. Es wurde zuerst das ClientDiff-Verfahren für Differenzbildung verwendet, welches als ineffizient galt und durch LFDDiff-Verfahren ersetzt wurde.

Bei der Evaluation wurden die im Entwurf vorgestellten Varianten miteinander verglichen und abschließend wurde die ServerKompact-Variante mit dem LFDDiff-Verfahren kombiniert.

6.2 Mögliche Erweiterungen

6.2.1 Kommunikation

Eine mögliche Verbesserung dieser Arbeit besteht darin, während der Laufzeit der Kampagne Websocket für die Kommunikation anstatt AJAX zu verwenden. Durch die periodische AJAX-Anfragen wird der Server ausgelastet, außerdem könnte die

Kommunikation zwischen der Anwendungsschicht und FAIL* mithilfe von Websocket statt Dateisystem realisiert werden.

6.2.2 Schätzung

Eine aussagekräftige Schätzung während der Kampagnenlaufzeit könnte erst gegen Ende der Laufzeit der Kampagne erreicht werden, da die Fehlerinjektionsexperimente nicht zufällig durchgeführt werden. Eine Verbesserungsmöglichkeit beinhaltet eine zufällige Durchführung der Fehlerinjektionsexperimente.

6.2.3 Fehlerraumplot

Das Zeichnen des Fehlerraumplots während der Kampagnenlaufzeit könnte verbessert werden, indem nach jeder Serverantwort eine Komprimierung der gesamten Daten durchgeführt wird, zu jedem Zeitpunkt wird nur die kleinste kompakte Datenmenge gezeichnet. Das hat den Effekt, dass eine Reaktion auf Benutzerinteraktionen, etwa Zoomen, schneller erfolgen wird. Außerdem könnte das Zeichnen des Fehlerraumes durch Verwendung anderer Bibliotheken beschleunigt werden.

Literatur

[1] A. Avižienis u. a. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), S. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[2] M. Bostock, V. Ogievetsky und J. Heer. “D3: Data-Driven Documents”. In: *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS* 17.12 (Dez. 2011), S. 2301–2309. ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.185.

[3] P. Bühler, P. Schlaich und D. Sinner. *JavaScript*. Berlin,Heidelberg: Springer Vieweg, März 2018, S. 32–53. DOI: https://doi.org/10.1007/978-3-662-54730-4_21.

[4] J. Dunkel und A. Holitschke. *Softwarearchitektur*. Berlin,Heidelberg: Springer, 2003, S. 1–30. DOI: https://doi.org/10.1007/978-3-642-55552-7_1.

[5] M. Fische u. a. *Entwicklung erfolgreicher Webanwendungen*. Website. Online erhältlich unter https://www.bitkom.org/sites/default/files/file/import/LF-Webanwendungen-150910-1.pdf; abgerufen am 1. August 2020. 2015.

[6] M. Hoffmann. “Konstruktive Zuverlässigkeit: Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware”. Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016.

[7] *jQuery in Action*. Manning Publications, 2010.

[8] P. Marwedel. “Evaluierung und Validierung”. In: *Eingebettete Systeme*. Springer, 2008. Kap. 6, S. 241–243.

[9] H. Schirmeier u. a. “FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance”. In: *Proceedings of the 11th European Dependable Computing Conference (EDCC ’15)* (Paris, France). Piscataway, NJ, USA: IEEE Press, Sep. 2015, S. 245–255. DOI: 10.1109/EDCC.2015.28.

[10] I. Zaglov. *Multithreading in JavaScript: So funktionieren Web-Worker*. Website. Online erhältlich unter https://t3n.de/news/web-worker-javascript-556677/; abgerufen am 1. August 2020.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis