technische universität
dortmund

Bachelorarbeit

Leistungsanalyse der Datenstromverarbeitung in kCQL

Tim Tannert
6. Januar 2017

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
M.Sc. Alexander Lochmann

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

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Last- und Geschwindigkeitsanalyse der Datenstromverarbeitung von *kCQL*. Für diese Untersuchung wird eine *kCQL*-Anfrage unter verschiedenen Lastszenarien betrachtet und es werden Verfahren entwickelt, die für die Untersuchung weiterer Anfragen verwendet werden können. Die Lastanalyse wird mit Hilfe des *Profilingwerkzeugs Perf* durchgeführt und die Geschwindigkeitsanalyse mit dem *Tracingwerkzeug LTTng*. Das Ziel dieser Arbeit ist die Lokalisierung von Programmteilen von *kCQL*, die für eine Verbesserung der erzeugten Last und der Verarbeitungsgeschwindigkeit, verändert werden können.

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel bietet eine kurze Einführung in die Thematik dieser Arbeit. Eine weiterführende Einführung wird in Kapitel 2 gemacht.
Zum Beginn wird in Abschnitt 1.1 die Gliederung der gesamten Arbeit betrachtet. Anschließend werden in Abschnitt 1.2 die allgemeinen Ziele dieser Arbeit beschrieben.

1.1. Gliederung

Diese Arbeit besteht aus sieben Kapiteln. In diesem Kapitel (Kapitel 1) werden die Ziele dieser Arbeit betrachtet. Anschließend wird in Kapitel 2 *kCQL* genauer beschrieben und dessen Funktion erläutert. Ferner wird in Kapitel 2 auf die Motivation für eine Leistungsanalyse von *kCQL* eingegangen. In Kapitel 3 werden verschiedene Werkzeuge zur Leistungsanalyse vorgestellt, aus denen im späteren Verlauf der Arbeit zwei Werkzeuge ausgewählt werden. Daraufhin werden in Kapitel 4 die Analyseverfahren für die Last- und Verarbeitungsgeschwindigkeitsanalysen entwickelt. In Kapitel 5 findet anschließend die Betrachtung der für die Analyse wichtigen Implementierungsdetails statt. Als nächstes wird aufbauend auf den vorangegangenen Kapiteln in Kapitel 6 die benötigten Messergebnisse für die Untersuchung von *kCQL* betrachtet und ausgewertet. Abschließend werden in Kapitel 7 die Ergebnisse dieser Arbeit zusammengefasst und ein kleiner Ausblick auf weitere Arbeiten und Untersuchungen - die auf diese Arbeit aufbauen können - gewährt.

1.2. Ziele

KCQL (Kernel Continuous Query Language) ist eine deklarative Anfragesprache, die Zugriff auf interne Zustände und Informationen des Betriebssystems gewährt. Dabei gibt es zwei Anforderungen, die an *kCQL* gestellt werden. Zum einen sollen Informationen so schnell wie möglich ausgegeben werden. Erhält *kCQL* neue relevante Eingabeinformationen, dann sollen diese so schnell wie möglich eine Auswirkung auf Ausgabe der *kCQL*-Anfrage haben. Zum anderen ist es wichtig, dass *kCQL* so wenig Last wie möglich erzeugt. Dies ist ein weitverbreitetes Ziel bei der Entwicklung von Programmen. Andere Prozesse des Systems sollen nämlich so wenig wie möglich beeinflusst werden. Besonders

die Ausgaben von *kCQL* können durch eine zu hohe Systemlast auf verschiedene Weisen beeinflusst werden.
Deswegen werden in dieser Arbeit Verfahren für eine Analyse der Last und der Verarbeitungsgeschwindigkeit entwickelt. Mithilfe dieser Verfahren sollen anschließend Programmteile von *kCQL* gefunden werden, die für eine Verbesserung der Last und der Verarbeitungsgeschwindigkeit angepasst werden müssen. Diese beiden Untersuchungen werden in dieser Arbeit getrennt voneinander betrachtet.
In Abschnitt 2.3 wird diese Einführung mit der Betrachtung von *kCQL* und der Motivation dieser Arbeit, fortgesetzt.

2. Kernel-Continuous-Query-Language

Dieses Kapitel bietet eine kurze Einführung in das Werkzeug *Kernel Continuous Query Language (kCQL)* und dessen Implementierung. Am Anfang wird in Abschnitt 2.1 die Grundidee und die Architektur der *kCQL*-Implementierung erläutert. Anschließend befasst sich Abschnitt 2.2 mit der *kCQL*-Anfragesprache. Dabei wird sowohl auf die Syntax der Sprache als auch auf die interne Verarbeitung einer Anfrage eingegangen. Am Ende dieses Kapitels werden in Abschnitt 2.3, im Hinblick auf den weiteren Verlauf dieser Arbeit, die Motive einer genaueren Leistungsanalyse betrachtet.

2.1. Architektur

Das Werkzeug *Kernel-Continuous-Query-Language (kCQL)* wird von der Arbeitsgruppe *Eingebettete Systemsoftware* an der TU-Dortmund entwickelt und stellt eine deklarative Schnittstelle zum Linux-Kernel bereit [1]. *KCQL* ermöglicht den deklarativen Zugriff auf Zustandsdaten und Ereignisse aus dem Linux-Kernel. Dabei soll *kCQL* einen einheitlichen und leichten Zugriff auf wichtige Betriebssystemdaten ermöglichen, der beispielsweise Programmierern und Systemadministratoren bei ihrer täglichen Arbeit hilft. Dieser Zugriff geschieht über sogenannte *kCQL*-Anfragen (vgl. Abschnitt 2.2).
Einige *kCQL*-Anfragen stellen Daten bereit, auf die aus dem Useradressbereich nicht direkt zugegriffen werden kann. Andere *kCQL*-Anfragen stellen wiederum Daten bereit, auf die aus dem Kerneladressbereich nicht ohne weiteres zugegriffen werden kann. Deshalb besitzt die *kCQL*-Implementierung mehrere Instanzen in verschiedenen Adressbereichen. Einige Anfragen benötigen allerdings lediglich eine User- und eine Kerneladressbereichsinstanz. Die Kerneladressbereichsinstanz der *kCQL*-Implementierung ist als Linux-*Kernelmodul* implementiert. Zum Zeitpunkt dieser Arbeit ist der Useradressbereichsteil aus Testzwecken eine Useranwendung. Momentan können sowohl die Useranwendung als auch das *Kernelmodul* lediglich fest vordefinierte Anfragen verarbeiten. Dies wird sich allerdings im weiteren Verlauf der Entwicklung ändern.

In Abbildung 2.1 ist die Architektur der *kCQL*-Implementierung erkennbar. Sowohl die Useranwendung als auch das *Kernelmodul* erzeugen aus der zu verarbeitenden Anfrage

einen Anfrageplan. Dieser entspricht in der Abbildung jeweils einer Instanz (*instance*) im Useradressbereich und einer im Kerneladressbereich. Der Anfrageplan besteht aus Operatoren, die von einem Scheduler ausführt werden (vgl. Unterabschnitt 2.2.4). Außerdem besitzt ein Anfrageplan asynchrone Quellpuffer, die beispielsweise im *Kernelmodul* durch *KProbes* befüllt werden. Diese Quellpuffer beinhalten die Daten, die von der gewählten Anfrage weiterverarbeitet werden müssen. Eine Anfrage wird so ausgewertet, dass die Kommunikation zwischen den Instanzen so gering wie möglich ist. Die Ausgabe einer Anfrage kann sowohl im User- als auch im Kerneladressbereich liegen. In jedem Adressbereich existieren Datenquellen (*source*). Diese Datenquellen werden auch Provider genannt und versorgen den Anfrageplan mit Daten. Sollte eine Anfrage beispielsweise Informationen über Netzwerkpakete benötigen, wird ein Provider im Kerneladressbereich gestartet, dessen Aufgabe das Sammeln der Metadaten aller ein- und ausgehenden Datenpakete ist.
Die Kommunikation zwischen den Instanzen läuft über virtuelle Dateien (*transport*) (vgl. Unterabschnitt 2.2.3). Auch die Kommunikation zwischen zwei Operatoren des Anfrageplans geschieht über Puffer, in die der eine Operator Daten schreibt und aus denen der andere Daten liest.

2.2. Anfragesprache

Der Teil von *kCQL*, der für die Anfrageverarbeitung zuständig ist, wird *STREAM* genannt und wurde in Stanford entwickelt [2]. Die Anfragen, welche die *STREAM*-Implementierung verarbeiten kann, sind sogenannte *CQL*-Anfragen. *CQL*-Anfragen haben eine Ähnlichkeit mit *SQL*[1]-Anfragen. Schlüsselwörter, die in beiden Anfragesprachen vorkommen, sind beispielsweise *SELECT*, *JOIN*, *WHERE*, *AS* und einige weitere [1, 3]. Allerdings können *CQL*-Anfragen sowohl mit Relationen als auch mit Datenströmen arbeiten. Die Grundidee der beiden Konzepte und deren Implementierung werden in den folgenden Unterabschnitten besprochen. Anders als in der *SQL* dient der Linux-Kernel für *kCQL* als Datenquelle. Die *kCQL*-Implementierung stellt diese Daten nach außen hin als Relation oder Datenstrom dar.

2.2.1. Relationen

Wie auch in *SQL*-Anfragen, gibt es in *CQL*-Anfragen Relationen. Im Kontext der *kCQL*-Implementierung können Relationen wie folgt interpretiert werden:
Relationen besitzen Elemente, die wiederum Attribute besitzen. Jedes Attribut eines Elements hat eine eindeutige Bezeichnung und einen Wert. Die Relation kann auch als

[1] Structured-Query-Language

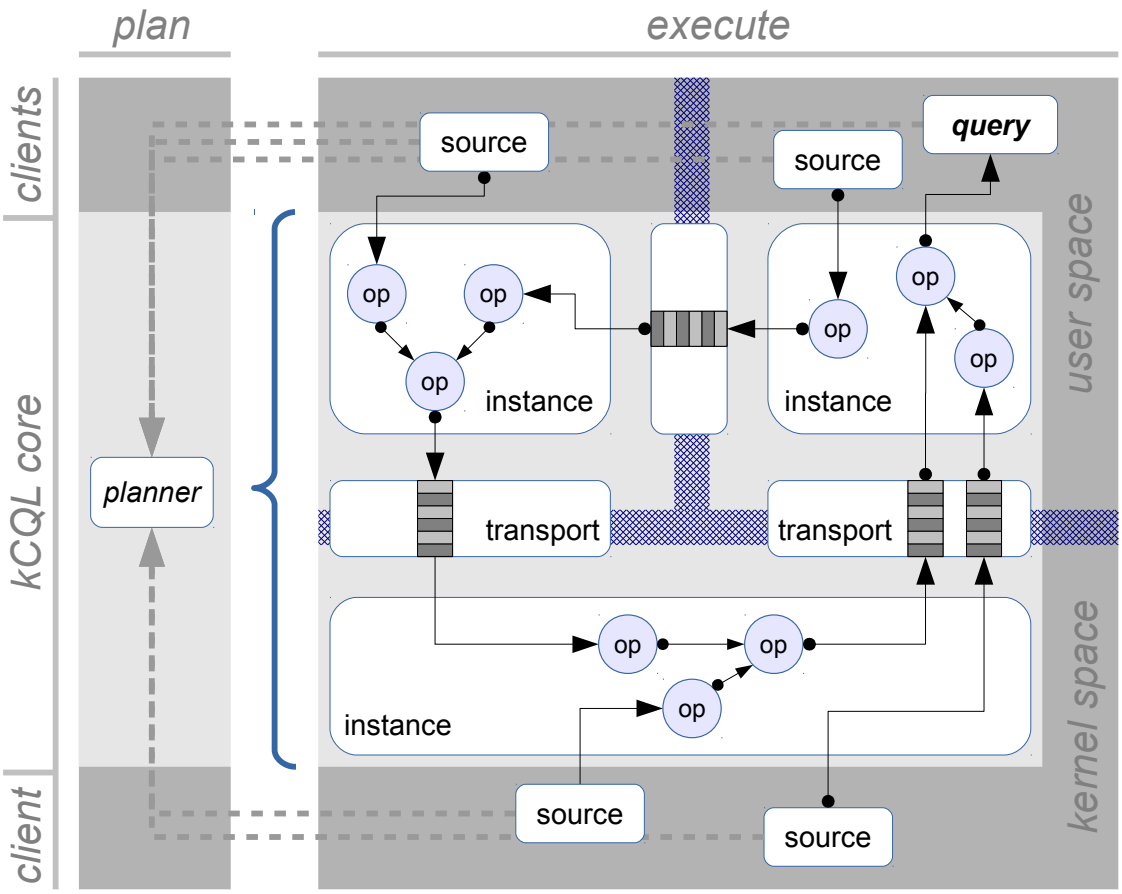


Abbildung 2.1.: Die Abbildung beschreibt die Architektur der *kCQL*-Implementierung. Erkennbar ist hierbei der fundamentale Aufbau und dessen Gliederung in User- und Kerneladressbereich [1].

Tabelle dargestellt werden.

Im Kontext der *kCQL* werden Informationen, die einen Zustand des Systems beschreiben, wie z. B. die laufenden Prozesse, offenen Sockets oder die geöffneten Dateien, als Relationen dargestellt [1]. In Wirklichkeit liegen diese Informationen in Datenstrukturen, die durch *kCQL* als Relationen dargestellt werden.
Damit die *kCQL*-Implementierung eine Möglichkeit hat diese Informationen intern abzubilden, existieren Plus-, Minustupel und die *Synopsis* in *STREAM* [4]. Besitzt nun ein Operator (vgl. Unterabschnitt 2.2.4) eine Relation als Eingabe, baut dieser sich seine eigene Sicht auf die eigentliche Datenstruktur in seiner *Synopsis* auf. Bei einer Zustandsänderung erhält er Plus- oder Minustupel. Ein Plustupel erzeugt ein neues Element in der *Synopsis*, ein Minustupel entfernt ein existierendes Element aus der *Synopsis*. Plustupel werden an der Informationsquelle erzeugt, wenn eine neue Information in die ursprüngliche Datenstruktur aufgenommen wird, ein Minustupel wenn sie aus der Datenstruktur genommen wird. Somit stellt die Synopsis die Informationen der Datenstruktur als Relation dar. Es existieren noch weitere Tupelarten, auf die in dieser Arbeit nicht weiter eingegangen wird.

Beispiel

In Abbildung 2.2 werden die Plus- und Minustupel anhand eines Beispiels in mehreren Schritten erläutert. Der aktuelle Status der Relation ist immer in der Synopsis abgebildet. Neben der *Synopsis* ist ihr Soll-Zustand abgebildet, welcher später ihr Ist-Zustand sein soll. Zu Beginn wird der Socket mit Port 42 dem System hinzugefügt und der Socket mit Port 24 aus dem System entfernt. Diese Aktionen führen zu zwei nacheinander gesendete Informationen, die im ersten Teil der Abbildung zu erkennen sind. Zum einen wird ein Plustupel für Port 42, zum anderen ein Minustupel für Port 24 erzeugt. Anschließend wird im zweiten Teil der Abbildung dargestellt, wie die Plus- und Minustupel vom Operator an die *Synopsis* weitergegeben werden. Zuletzt ist im dritten Teil der Abbildung erkennbar, wie der Socket mit dem Port 42 der *Synopsis* hinzugefügt und der Socket mit dem Port 24 aus der *Synopsis* entfernt wird. Anschließend kann der Operator die Informationen aus der *Synopsis* für seine Aufgabe verwenden. Wichtig ist hierbei, dass die in diesem Beispiel in einem Schritt verschickten Plus- und Minustupel eigentlich in zwei unabhängigen Schritten aus dem Puffer genommen und weiter verarbeitet werden.

2.2.2. Datenströme

Damit *kCQL* nicht nur zustandsbasierte Informationen verarbeiten kann, sondern auch kontinuierlich auftretende Daten wie z. B. Netzwerkpakete, werden zusätzlich zu den Relationen auch Datenströme verwendetet. *STREAM* ermöglicht es indirekt mit Da-

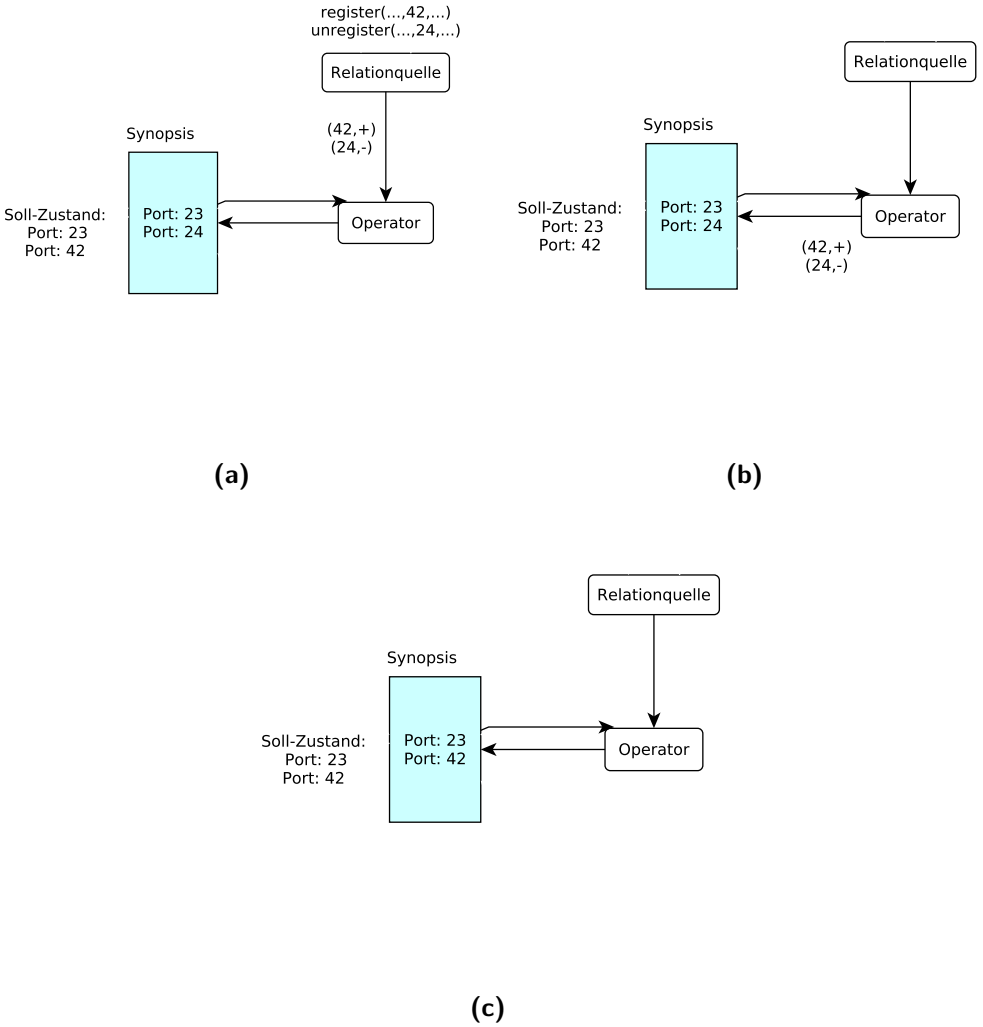


Abbildung 2.2.: Visualisierung der internen Vernwendung der Synopsis, Plus- und Minusinformationen in STREAM anhand eines Beispiels.

tenströmen umzugehen [1]. *CQL* bietet eine Möglichkeit, Datenströme mit einem sogenannten Fenster in eine Relation zu überführen. Das Fenster gibt entweder eine Anzahl der letzten n Informationen an, die in die Zielrelation aufgenommen werden, oder einen Zeitraum aus dem alle Informationen in die Zielrelation aufgenommen werden.
Außerdem bietet *CQL* zusätzlich Möglichkeiten eine Relation in einen Datenstrom umzuwandeln. Dies geschieht meistens bei den Ausgaben einer Anfrage. Hierfür existieren die Schlüsselwörter *ISTREAM*, *DSTREAM* und *RSTREAM*:

- **ISTREAM:** Hängt jedes der Eingaberelation hinzugefügte Element dem Datenstrom an.
- **DSTREAM:** Hängt jedes Element, dass aus der Eingaberelation gelöscht wird, dem Datenstrom an.
- **RSTREAM:** Hängt bei jeder Änderung der Eingaberelation jedes Element dem Datenstrom an.

Die interne Darstellung von Datenströmen wird in Unterabschnitt 2.2.4 genauer betrachtet.

2.2.3. Anfragestruktur

Die Struktur einer *CQL*-Anfrage ähnelt sehr stark dem Aufbau einer *SQL*-Anfrage [1, 3]. Der grobe Aufbau einer einfachen *CQL*-Anfrage ist in Listing 2.1 dargestellt. Das Schlüsselwort *FROM* gibt an, aus welchen Relationen die Informationen beschafft werden. Sollen Informationen aus einem Datenstrom beschafft werden, muss dieser zuvor mit einem Fenster in eine Relation überführt werden.
Das *WHERE* Schlüsselwort kann Bedingungen festlegen, die für Elemente aus den angefragten Relationen gelten müssen, um ausgegeben zu werden. Beispielsweise kann *FROM* aus zwei Relationen lesen, die beide ein Attribut "Name" besitzen. Das *WHERE* Schlüsselwort kann in diesem Zusammenhang dazu verwendet werden nur Elemente auszugeben, in denen die Werte des Attributes "Name" identisch sind.
Im *SELECT* Teil der Anfrage wird angegeben, welche Attributwerte der ermittelten Elemente ausgegeben werden sollen. Diese Spalteninformationen werden auch Attribute genannt.
Die in Listing 2.2 gezeigte Anfrage gibt beispielsweise die Datenpaketgröße, Datenpaketrichtung und Prozessid aller ein- und ausgehenden Datenpakete als Datenstrom aus. Die Anfrage in Listing 2.3 gibt dagegen das Transportprotokoll, die Datenpaketlänge und den dazugehörigen Prozessnamen aller gesendeten oder empfangenen Netzwerkpakete als Datenstrom aus. Die zweite Anfrage ist die *Netsimple*-Anfrage, die im weiteren Verlauf dieser Arbeit näher betrachtet wird.

```
SELECT  Spaltennamen
FROM    Tabellen
WHERE   Bedingung
```

Listing 2.1: Abstrakter Aufbau einer CQL-Anfrage

```
Packets: RSTREAM (
    SELECT  packet.datalen AS len, packet.direction AS
       dir, process.pid AS pid,
                process.name AS pname
    FROM    packet [now], socket, process
    WHERE   packet.sid = socket.sid AND socket.pid =
       process.pid
);
```

Listing 2.2: Package-Anfrage: Gibt die Datenpaketgröße, Datenpaketrichtung und Prozessid eines jeden Prozesses der ein Datenpaket empfängt oder sendet aus.

```
Netsimple: RSTREAM (
    SELECT  process.name,net.macprot,net.datalength
    FROM    net [now],sockets,process
    WHERE   net.socket=sockets.sid and
       sockets.pid=process.pid;
);
```

Listing 2.3: *Netsimple*-Anfrage: Gibt die Länge, die Richtung, die Prozessid und den Prozessnamen aller Datenpakete aus [1].

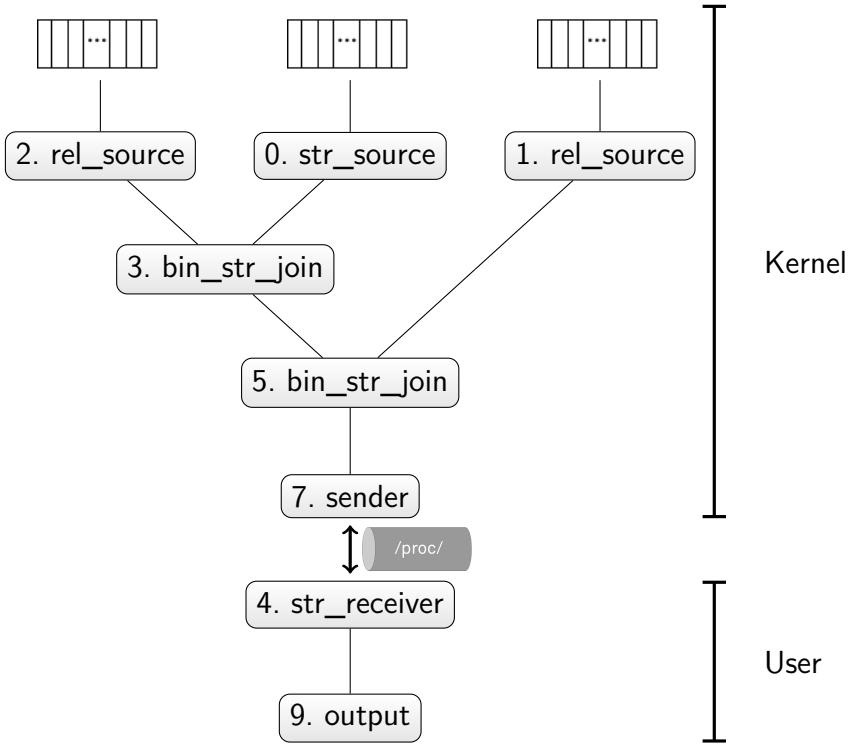


Abbildung 2.3.: Abstrakte Darstellung des aus der *Netsimple*-Anfrage (vgl. Listing 2.3) erzeugten Anfrageplans.

2.2.4. Interne Verarbeitung

Nach dem Einlesen einer Anfrage wird sie von der *kCQL*-Implementierung optimiert und anschließend in einen Anfrageplan überführt. Ein Anfrageplan besteht aus Operatoren, Puffer und *Synopsis*. Jede *Synopsis* gehört zu einem Operator, wohingegen jeder Transport Puffer zu zwei Operatoren gehört. Lediglich die asynchronen Quellpuffer besitzen keine *Synopsis*. *STREAM* sieht vor, dass fast jeder Teil einer Anfrage zu einen Operator überführt wird. Der *WHERE*-Teil wird in den meisten Fällen zu einen *Join*-Operator, der *FROM*-Teil zu *Source*-Operatoren und Fenster, *RSTREAM*, *DSTREAM* und *ISTREAM* werden zu eigenen Operatoren. Allerdings optimiert die *kCQL*-Implementierung einige Fenster weg und ersetzt diese durch eigene Operatoren, die mit Datenströmen statt Relationen arbeiten. Unter gewissen Umständen kann die *kCQL*-Implementierung auch *RSTREAM*-, *DSTREAM*- und *ISTREAM*-Operatoren wegoptimieren. Lediglich der *SELECT*-Teil wird nicht direkt in Operatoren überführt. Er nimmt Einfluss auf die Struktur der anderen Operatoren.
Folglich bestimmt der Anfrageplan, wie eine Anfrage verarbeitet wird. Er kann für jede

beliebige Anfrage erzeugt und für eine nähere Betrachtung in Textform ausgegeben werden. Während der Verarbeitung einer Anfrage werden von einem *Scheduler* die Operatoren des Anfrageplans ausgeführt, welche die über ihre Eingabepuffer erhaltenen Daten verarbeiten müssen. Je nachdem ob der Operator eine Relation oder einen Datenstrom erwartet, verwaltet er eingehende Plus- und Minustupel in Verbindung mit der *Synopsis* oder er verarbeitet direkt die eingehenden Plustupel. Beim *Scheduling* wird darauf geachtet, dass kein Operator ausgeführt wird, wenn er keine aktuellen Eingabedaten besitzt.
In Abbildung 2.3 sind die Operatoren und einige asynchrone Quellpuffer aus dem Anfrageplan der Netsimple-Anfrage abgebildet. Für eine nähere Betrachtung dieser Abbildung müssen erst einmal die verwendeten Operatoren näher erläutert werden:

Relation-Quelle (rel_source)

Relation-Quellen lesen die Daten aus den mit ihnen verbundenen asynchronen Quellpuffern, die wiederum von bestimmten Providern befüllt werden und versorgen die folgenden Operatoren mit den relevanten Daten. Welcher Provider den asynchronen Puffer befüllt, ist abhängig von der Anfrage. Die empfangenen Daten sind entweder Plus- oder Minustupel.

Stream-Quelle (str_source)

Auch die Stream-Quellen erhalten ihre Daten aus den mit ihnen verbundenen asynchronen Puffern. Allerdings erhalten sie nur Plustupel.

Join (bin_str_join)

Die *Join*-Operatoren *joinen* einen Datenstrom mit einer Relation. Im Zusammenhang mit Datenbanken wird von einem Join gesprochen, wenn Informationen aus zwei verschiedenen Relationen zusammengeführt werden sollen, die auf gemeinsamen Informationen basieren [5]. Dafür werden oft sogenannte Schlüsselwerte verglichen. Diese Erklärung trifft auch auf das *Stream-Join* von *kCQL* zu. Es vergleicht gleichartige Attribute aus den Informationen des Datenstroms mit einer Relation und ermittelt ein Ergebnis. Die Form des Ergebnisses und welche Attribute verglichen werden, legt die zugrunde liegende Anfrage fest (vgl. Listing 2.3).

Sender (sender)

Der Sender schreibt die erhaltenen Daten in eine virtuelle Datei, die im */proc/*-Dateisystem liegt. Dies geschieht, da auch die Useranwendung Zugriff auf diese virtuelle

Datei hat. Ist die virtuelle Datei voll, dann benachrichtigt der Sender die Programminstanz des Empfängers und teilt ihr mit, dass Daten zum Lesen bereitstehen.

Empfänger (str_receiver)

Der Empfänger liest kontinuierlich aus der virtuellen Datei im /proc/-Dateisystem und gibt diese Informationen an den nächsten Operator weiter. Somit empfängt er Daten vom Sender. Ist die virtuelle Datei leer, dann arbeitet der Empfänger nicht. Seine Programminstanz erwartet anschließend eine Nachricht vom Sender, um weiter zu arbeiten.

Ausgabe (output)

Die Ausgabe erhält die Informationen einer vollständig verarbeiteten Anfrage und muss diese lediglich ausgeben. Was danach mit den Informationen geschieht, hängt von dem dahinter liegenden Programm ab. Die aktuelle *kCQL*-Implementierung gibt die Daten auf der Konsole aus.

Die in der Abbildung oben liegenden Quelloperatoren beziehen ihre Daten direkt aus den asynchronen Puffern, die wiederum von Providern befüllt werden. Daraufhin werden die Daten von allen Operatoren verarbeitet. Sie werden auf dem Weg zum Sender vom *Kernelmodul*, nach denen im *WHERE*-Teil der Anfrage gesetzten Bedingungen, *gejoint*. Anschließend werden die Ergebnisse zur Useranwendung transportiert und dort weiter an die Ausgabe gegeben, welche am Ende die Antwort der Anfrage ausgibt.

Aktueller Stand

In der aktuellen Version von *kCQL* existiert noch kein Werkzeug und keine *API*, mit dessen Hilfe frei wählbare *kCQL*-Anfragen formuliert werden können. Momentan existiert die Useranwendung *inmem.elf*, die im Useradressraum arbeitet und das *Kernelmodul kcpp*, das im Kerneladressraum arbeitet. Sowohl beim Start der Useranwendung als auch beim Laden des *Kernelmoduls* muss eine fest definierte Anfrage ausgewählt werden. Die auszuwertende Anfrage kann nur durch ein Neuladen des *Kernelmoduls* und durch einen Neustart der *Clientsoftware* geändert werden.
Eine auswählbare Anfrage ist beispielsweise die Netsimple-Anfrage. Die Leistungsanalyse in dieser Arbeit bezieht sich ausschließlich auf diese. Allerdings sind die erarbeiteten Analyseverfahren auch auf andere Anfragen anwendbar.

	DTrace	SystemTap	Fay	PiCO QL	kCQL
1) Deklarativ	Teilweise	Teilweise	Teilweise	Ja	Ja
2) Datenstromverarbeitung	Ja	Ja	Ja	Nein	Ja

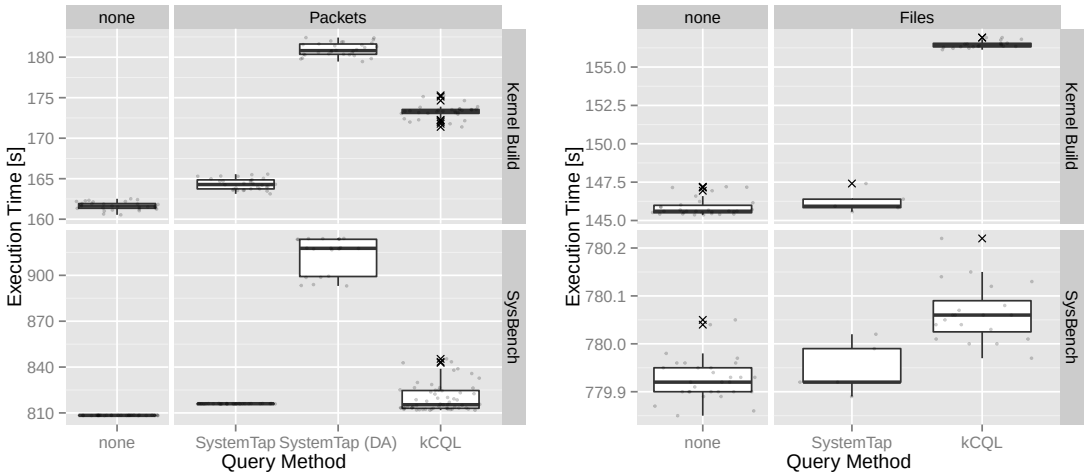
Tabelle 2.1.: Kleiner Vergleich der *kCQL* ähnlichen Werkzeuge [1, 6, 7, 8]. Verglichen wird zum einen, ob Datenströme verarbeitet werden können und zum anderen, ob das Werkzeug deklarative Anfragen verarbeiten kann.

2.3. Motivation für eine nähere Untersuchung

Neben *kCQL* existieren ähnliche Werkzeuge, die Zugriff auf betriebssysteminterne Daten gewähren. Da wären z. B. *SystemTap* (vgl. Abschnitt 3.3), *Fay*, *Dynamic-Trace (DTrace)* und *Pico-Collections-Query-Library (PiCO-QL)*, um nur einige zu nennen [1, 6, 7, 8]. Durch einen Vergleich mit *kCQL* ist allerdings erkennbar, dass sie nicht die gesamte Funktionsvielfalt von *kCQL* besitzen.
Eine solche Gegenüberstellung wird in Tabelle 2.1 gemacht. Es ist möglich jede denkbare *kCQL*-Anfrage in eine äquivalente Form zu überführen, so dass die oben besprochenen Werkzeuge eine vergleichbare Ausgabe wie *kCQL* liefern. *Pico QL* ist dabei eine Ausnahme. Da es keine Datenströme verarbeiten kann, muss eine äquivalente *kCQL*-Anfrage in eine *Pico QL*-Anfrage überführt werden und anschließend regelmäßig für eine Datenstromsimulation ausgeführt werden.
Eine wichtige Eigenschaft, die diese Werkzeuge mit *kCQL* gemeinsam haben, ist, dass sie den Prozessor so wenig wie möglich auslasten sollen. Dies ist sehr wichtig, damit die Werkzeuge einen so geringen Einfluss wie möglich auf das System und dessen Prozesse haben. Im weiteren Verlauf dieser Arbeit werden die Begriffe Systemlast und Prozessorauslastung gleichgesetzt und beschreiben das selbe.
Es ist also sinnvoll einen Vergleich der erzeugten Systemlast dieser Werkzeuge anzustellen, um zu überprüfen, ob die Verwendung von *kCQL* angebracht ist. In dem Papier über *kCQL* [1] werden zwei *kCQL*-Anfragen sowohl in *SystemTap* als auch in *SystemTap(Direct Access (DA))* implementiert. Anschließend werden die Systemlasten, die *SystemTap*, *SystemTap(DA)* und die *kCQL*-Implementierung in der Ausführung erzeugen, verglichen.
Die Anfragen sind in *SystemTap* so implementiert, wie *kCQL* diese intern verarbeitet. Das Skript baut sich eine interne Sicht der benötigten Kerneldatenstruktur auf. Hierfür passt es seine Daten immer an, wenn Ereignisse eintreten, die einen relevanten Zustand in dieser Datenstruktur verändern. Falls ein *Join* dies vorsieht, werden Datenströme einfach im Skript durch Vergleiche mit der entsprechenden internen Sicht der Systemzustände verarbeitet und es wird eine Ausgabe erzeugt.
SystemTap (DA) ist kein eigenständiges Werkzeug, sondern ein *SystemTap*-Skript,

welches direkt auf die Kerneldatenstrukturen zugreift. Dadurch, dass dieses Skript keine eigene Sicht auf die Datenstruktur hat, muss im schlechtesten Fall für jedes Element aus einem Datenstrom durch die gesamte Kerneldatenstruktur iteriert werden.
Für den Vergleich der Systemlast werden die Ausführungszeiten zweier Systembenchmarks in Boxplots dargestellt. In Abbildung 2.4b und 2.4a werden jeweils mehrere Messungen der *Benchmarklaufzeit* gemacht. Eine Messung wird gemacht, während nur der entsprechende Benchmark läuft, eine während das *SystemTap*-Skript läuft, eine während das *SystemTap(DA)*-Skript läuft und eine während *kCQL* läuft.
Die Abbildung 2.4a beschreibt die Laufzeiten für die *Packets*-Anfrage (vgl. Listing 2.2) wohingegen die Abbildung 2.4b die Laufzeiten der *Files*-Anfrage beschreibt. Die Form der *Files*-Anfrage ist an dieser Stelle unwichtig und wird nicht betrachtet. Die Systemlast der einzelnen Werkzeuge kann anhand der Differenz zwischen den Laufzeiten eines Benchmarks berechnet werden. Dafür wird die Laufzeit während kein Werkzeug arbeitet und die Laufzeit während das betrachtete Werkzeug arbeitet miteinander verrechnet.
In dem zusammenfassenden Diagramm Abbildung 2.4 ist zusätzlich erkennbar, dass die Implementierung von *kCQL* sowohl mit der *Packets*-Anfrage als auch mit der *Files*-Anfrage mindestens genauso viel Last erzeugt wie *SystemTap* und *SystemTap(DA)*.
Da die Geschwindigkeiten mit denen *SystemTap*, *SystemTap (DA)* und *kCQL* arbeiten nicht verglichen werden, sondern ausschließlich die Mehrlast für das System, ist nicht genau bestimmbar, ob die Implementierung von *kCQL* ineffizienter ist. Es wäre auch möglich, dass die *kCQL*-Implementierung mehr Last im System erzeugt, da sie deutlich schneller als *SystemTap* und *SystemTap(DA)* arbeitet. Im Allgemeinen ist es wünschenswert die Systemlast, die *kCQL* generiert weiter zu reduzieren. Allerdings sollte sich die Verarbeitungsgeschwindigkeit dabei höchstens verbessern.

Neben der Reduzierung der Systemlast, ist die schnelle Verarbeitung von Anfragen wünschenswert. Sobald sich Informationen im System ändern oder neue Informationen in einem Datenstrom vorliegen, soll dies Auswirkungen auf die Antwort einer Anfrage haben. Dieser Punkt wurde im zuvor betrachteten Papier nicht untersucht. Daher ist das Ziel dieser Arbeit, Optimierungsansätze für die Verbesserung der Systemlast und der Verarbeitungsgeschwindigkeit der *kCQL*-Implementierung zu finden. Im optimalen Fall kann eine Reduzierung der Systemlast auch zu einer besseren Bearbeitungsgeschwindigkeit führen. Im weiteren Verlauf wird das Problem der Lastverringerung getrennt vom Problem der Geschwindigkeitsverringerung betrachtet. Es wird ein Vorgehen für die Lastanalyse und eines für die Geschwindigkeitsanalyse entwickelt.



(a) **(b)**

Abbildung 2.4.: In Abbildung 2.4a und 2.4b sind Benchmarklaufzeiten abgebildet. Dabei werden die Laufzeiten der *Files*- und der *Packets*-Anfrage, ohne Anwendung in Ausführung, mit *kCQL* in Ausführung und mit SystemTap in Ausführung betrachtet.

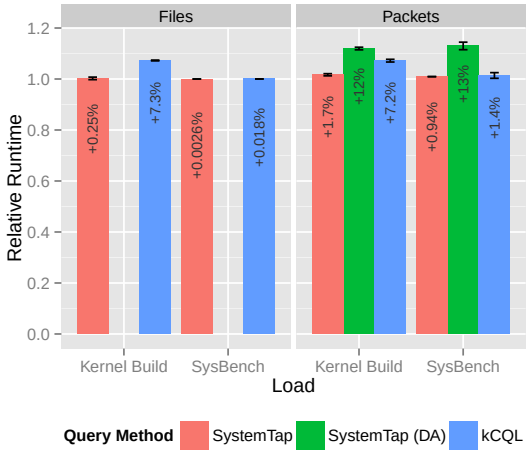


Abbildung 2.5.: Relative Benchmarklaufzeiten bei der Ausführung der *Files*- oder *Packets*-Anfrage durch *kCQL*, *SystemTap* und *SystemTap (DA)* im Hintergrund.

3. Existierende Werkzeuge

In diesem Kapitel werden die Werkzeuge vorgestellt, die im weiteren Verlauf dieser Arbeit zur Leistungsanalyse von *kCQL* verwendet werden können. Zuerst werden in Abschnitt 3.1 zwei Paradigmen für die Leistungsanalyse vorgestellt, mit deren Hilfe die betrachteten Werkzeuge unterschieden werden. Daraufhin wird in Abschnitt 3.2 eine Vorauswahl der zu betrachtenden Werkzeuge getroffen. Anschließend wird in Abschnitt 3.3 die Funktionalität aller in der Vorauswahl ermittelten Werkzeuge aufgeführt.

3.1. Paradigmen der Leistungsanalyse

Es existieren zwei Paradigmen der Leistungsanalyse, die für eine Kategorisierung der Werkzeuge relevant sind. Zum einen ist da die *Untersuchung zur Laufzeit* und zum anderen gibt es die *manuelle Instrumentierung*. Diese Paradigmen werden sehr oft durcheinander geschmissen und kaum voneinander abgegrenzt, weshalb die Definitionen aus [9] verwendet werden. Keines dieser Paradigmen ist für diese Arbeit dem anderen vorzuziehen. Deswegen werden im folgenden beide betrachtet.

3.1.1. Untersuchung zur Laufzeit

Um einen Prozess genauer zu untersuchen, muss die Ausführung eines Programms als eine Sequenz von Aktionen gesehen werden. Eine Aktion kann dabei der Eintritt in eine Programmroutine, eine Synchronisation, eine Kommunikation oder die Ausführung einer Programmzeile sein [9]. Jede Aktion wird als Ereignis interpretiert. Dadurch wird ein Programm durch eine Aufzeichnung dieser Ereignisse untersuchbar.
Die Grundidee der Untersuchung zur Laufzeit (auch *Profiling* genannt) ist das Sammeln und das Aggregieren von Ereignissen. Für die Untersuchung zur Laufzeit ist in der Regel keine Veränderung oder das vorhanden sein des Quellcodes notwendig. Somit können selbst Programme, dessen Quellcode nicht verfügbar ist, untersucht werden.
Die *Profiling*-Werkzeuge sammeln Statistiken, die für eine Leistungsanalyse eines Programmes verwendet werden können. Beispielsweise existieren Werkzeuge, welche die Häufigkeit von Funktionsaufrufen zählen.
Wie die entsprechenden Werkzeuge Informationen über Ereignisse sammeln, hängt von

der Implementierung des Werkzeugs, der verwendeten Hardware und dem verwendeten Betriebssystem ab.

3.1.2. Manuelle Instrumentierung

Manuelle Instrumentierung (auch *Tracing* genannt) beschreibt das manuelle Einfügen von Ereignisse in ein zu untersuchendes Programm. Die eingefügten Ereignisse werden *Tracepoints* genannt und geben Zeitstempel mit den Zeitpunkten ihrer Ausführung und zusätzliche Argumente aus dem Programm aus.
Die manuelle Instrumentierung wird für eine genaue Untersuchung des Programmablaufes und des Programmverhaltens verwendet [9]. Für dieses Paradigma ist das Vorhandensein des Quellcodes und eine erneute Übersetzung essentiell wichtig.

3.2. Vorauswahl

Für die Leistungsanalyse von Programmen gibt es eine Vielzahl an nützlichen Werkzeugen. Die große Anzahl der Werkzeuge ist unüberschaubar, weshalb bereits eine kleine Vorauswahl getroffen wurde. Aufgrund der großen Anzahl an verfügbaren Werkzeugen kann nicht garantiert werden, dass alle zu diesem Zeitpunkt existierenden Werkzeuge in die Betrachtung einbezogen wurden. Es kann aber gesagt werden, dass alle oft genutzten und im Internet weit verbreiteten Werkzeuge in die Betrachtung einbezogen[1] worden sind.
Alle ungeeigneten Werkzeuge werden nicht betrachtet. Als ungeeignet werden alle Werkzeuge angesehen, die nicht mit Linux-*Kernelmodulen* arbeiten können oder die kostenpflichtig sind. Letztere Einschränkung wurde gemacht, da die betrachteten kostenpflichtigen Werkzeuge keinen Funktionalität bieten, die eine Anschaffung für diese Arbeit rechtfertigen würde. Letztendlich sind folgende Werkzeuge übriggeblieben:

1. Perf [10]
2. LTTng [11]
3. PAPI [12]
4. SysProf [13]
5. oProfile [14]

[1] Die Implementierung von *DTrace* für Linux kann in dieser Arbeit leider aufgrund der späten Veröffentlichung nicht mehr betrachtet werden: http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html

Profiling	Tracing
Perf	Perf
oProfile	LTTng
SystemTap	-
SysProf	-
Offcputime	-
PAPI	-

Tabelle 3.1.: Zuordnung der Werkzeuge zu den Analysestrategien

6. SystemTap [15]
7. Offcputime [2]

Um sich hier einen Überblick zu verschaffen, werden in Tabelle 3.1 die Werkzeuge nach den bereits besprochenen Paradigmen der Leistungsanalyse: *Profling* und *Tracing* kategorisiert. Da kein Paradigma dem anderen vorzuziehen ist, wird keines von beiden ausgeschlossen.

3.3. Werkzeuge

Um zu entscheiden, welche Werkzeuge für die Untersuchung der *kCQL*-Implementierung geeignet sind, werden im Folgenden ihre wichtigsten Funktionen vorgestellt. Hierbei wird nicht auf die volle Funktionalität eingegangen, sondern nur auf die wichtigsten Eigenschaften und Funktionen eines jeden Werkzeugs.

3.3.1. Linux Trace Toolkit: next generation (LTTng)

LTTng ist eine Weiterentwicklung des *Linux Tracing Toolkits (LTT)*. Es ist ein Werkzeug, das besonders dazu geeignet ist, *Tracepoints* in Programme einzufügen. Hierbei wird die Minimierung der Beeinträchtigung des Systems - also der erzeugten Systemlast - angestrebt. Wichtige Eigenschaften von *LTTng* sind die Skalierbarkeit, Präzision, Erweiterbarkeit, Modularität und die einfache Anwendbarkeit [11]. Es ist gleichermaßen für eine Untersuchung von *Kernelmodulen* und Useranwendungen geeignet. *LTTng* besitzt einen *Session-Daemon*, der Hauptsächlich für die Aufzeichnung der *Tracepointinformationen* zuständig ist. Die genauere Vorgehensweise des *Session-Daemon* wird später betrachtet.
LTTng ist besonders dazu geeignet C und C++ Anwendungen zu instrumentieren. Dafür

[2] https://github.com/iovisor/bcc

```
TRACE_EVENT(
    /* "cql" ist der Subsystemname, "operator" ist der Eventname
       */
    cql_operator,

    /* Tracepoint Funktionsprototype */
    TP_PROTO(int id, const char *pos),

    /* Argumente für diesen Tracepoint */
    TP_ARGS(id, pos),

    /* LTTng doesn't need those */
    TP_STRUCT__entry(),
    TP_fast_assign(),
    TP_printk("",0)
);
```

Listing 3.1: Beispielhafter Tracepointdefinition für CQL-Operatoren.

müssen die verwendeten *Tracepoints* definiert werden. Das bedeutet ein *Tracepoint* erhält eine Bezeichnung und es wird definiert wie viele Argumente welchen Typs er besitzt. In Listing 3.1 wird die Definition des *Tracepoints "cql_operator"* dargestellt, die im späteren Verlauf dieser Arbeit verwendet wird. Wenn ein *Kernelmodul* untersucht werden soll, dann muss diese *Tracepoint*-Definition zusätzlich im Quellcode des *Session-Daemon* gemacht werden. Das bedeutet der *Session-Daemon* muss neu übersetzt und mit speziellen Parametern gestartet werden, damit die *Tracepoints* des untersuchten *Kernelmoduls* aufgezeichnet werden können.

Session-Daemon

Der *Session-Daemon* von *LTTng* verwaltet die *Tracingsitzungen*. Eine *Tracingsitzung* hat verschiedene Eigenschaften. Sie hat einen Namen, eine eigene Menge an *Tracedateien* (Ausgabedateien), einen Zustand (gestartet/gestoppt), einen Modus und einen eigenen Kanal.
Für die weitere Betrachtung ist der *Livemodus* der *Tracingsitzungen* interessant. In diesem Modus werden alle aktuell aufgerufenen *Tracepoints* aufgezeichnet.
Der Kanal einer *Tracingsitzung* besteht aus einer Menge an Ringpuffern. Die instrumentierte Anwendung schreibt beim Aufruf eines *Tracepoints* die entsprechenden Informationen in einen Ringpuffer der aktuellen *Tracingsitzung*. Ein Kanal hat mindestens einen *Ringpuffer* pro Prozessorkern, der wiederum aus mehreren Unterpuffern besteht. Im Standartfall sind es vier Unterpuffer. Ein spezieller Konsumprozess *des Session-Daemon* ist dafür zuständig die Unterpuffer der ihm zugeordneten gestarteten Sitzungen in

eine Datei zu schreiben. Dabei versucht der *Session-Daemon* so wenig Systemlast wie möglich zu erzeugen. Durch die Auswahl der Unterpuffergrößen und Unterpuffermengen kann die erzeugte Systemlast gesteuert werden.
Weitere Informationen zu *LTTng* können der Projektwebseite[3] entnommen werden.

3.3.2. Offcputime

Offcputime ist ein Werkzeug aus der *BPF Compiler Collection*[4]. Es wird von Brendan Gregg entwickelt und dient der Bestimmung der Prozessorabgabedauer. *Offcputime* zählt im Kernelkontext die Prozessorabgaben und summiert ihre Dauer auf. Lediglich die aufsummierten Ergebnisse werden an die Useranwendung weitergegeben. Das verringert die Daten, welche das Skript ausgeben muss. Damit *Offcputime* im Kernelkontext arbeiten kann, verwendet es *eBPF (extended BPF)*.
Zusätzlich zu der Dauer der Prozessorabgabe zeichnet *Offcputime* die Funktion, die vor der Prozessorabgabe in Ausführung war und dessen Stacktrace auf.
Aus der Ausgabe von *Offcputime* kann auch ein *Flame-Graph* erzeugt werden. Dieser wird in der Regel blau gefärbt und entsprechend seiner kalten Farben *Cold-Graph* genannt (vgl. Abbildung 3.2).
Die grobe Syntax eines Aufrufs von *Offcputime* ist in Listing 3.2 abgebildet.
Weitere Informationen zu *Offcputime* können auf der Webseite von Brenddan Gregg[5] eingeholt werden.

```
offcputime [--verbose] [--folded] [--useronly |
   --pid PID [Zeit] | [command [args]]]
```

Listing 3.2: Syntax von Offcputime.

3.3.3. OProfile

OProfile ist ein *Profilingwerkzeug*, das mithilfe von *Performance-Countern* arbeitet. Es bietet Zugriff auf die Performance-Monitoring-Counter, welche die Aufzeichnung von Hardwareevents wie *Cache-Misses*, Speicher-Referenzen, zurückgezogene Instruktionen (*instruction retired*) und Prozessor-*Clock-Cycles* ermöglichen. Diese Eigenschaften machen *oProfile* zu einem geeigneten Werkzeug, um auf Leistungsprobleme von Software zu schließen [14]. Ins besonders die Funktionsaufrufe und deren *Stacktraces* können von *oProfile* aufgezeichnet werden.
Durch die Zugriffe auf die *Performance-Counter* und *Hardwareevents* müssen untersuchte

[3] https://lttng.org
[4] https://github.com/iovisor/bcc
[5] http://brendangregg.com

```
ocount [options] [--system-wide | --process-list <pids>
   | --thread-list <tids> | --cpu-list <cpus> |
   [command [args]]]
operf [options] [--system-wide | --pid <pid> | [command
   [args]]]
```

Listing 3.3: Syntax von ocount und operf.

Programme nicht neu übersetzt werden. *OProfile* ist sehr ressourcensparend. Es erzeugt je nach Konfiguration 1-8% Mehrlast im System.
Um mit *oProfile Hardwareevents* aufzuzeichnen kann das Werkzeug *ocount* oder *operf* verwendet werden. Die aufgezeichneten *Hardwareevents* von operf müssen mit einem Werkzeug wie *opreport* oder *opannotate* nachbearbeitet werden. Die grobe Syntax eines Aufrufes von *ocount* oder *operf* ist in Listing 3.3 dargestellt. Im Gegensatz zu *ocount* kann *operf* entweder nur das gesamte System, oder einen bestimmten Prozess beobachten. Dagegen kann *ocount* mehrere Prozesse, *Threads* oder Prozessoren beobachten. Beide Werkzeuge können auch die *Hardwareevents*, die gezählt werden sollen, übergeben bekommen. Dann zeichnen sie nur die zu betrachtenden Events auf. Ein elementarer Unterschied zwischen *operf* und *ocount* ist, dass bei einer Aufzeichnung mit *operf* ein *Hardwareevents* häufiger als ein vordefinierter Wert auftreten muss, bevor es aufgezeichnet wird, wohingegen *ocount* jedes Auftreten zählt.
Weitere Informationen über *oProfile* können der Projektwebseite[6] und den *Man-Pages* von *operf* und *ocount* entnommen werden.

3.3.4. SystemTap

SystemTap stellt eine simple Skriptsprache bereit, die Ähnlichkeiten zu awk und C aufweist. Mit dieser Sprache können Skripte geschrieben werden, um ein Programm zur Laufzeit zu untersuchen [15]. Die von *SystemTap* angebotene Skriptsprache erlaubt die Verwendung von *KProbes* des Linux-Kernels ohne viel Aufwand. Es muss kein zusätzliches *Kernelmodul* geschrieben werden und die Verwendung von *SystemTap* kann nicht zu Systemabstürzen führen.
Durch *SystemTap* können des Weiteren Informationen über ein laufendes Linux-System gesammelt werden. Diese Eigenschaft ist bei der Untersuchung von Programmen sehr hilfreich. Davon profitieren auch die Skripte, die zur Untersuchung einer Anwendung zur Laufzeit (*profiling*) geschrieben werden. Somit kann zur Laufzeit des Programmes auf für diese Anwendung wichtige Informationen zugegriffen werden. Anschließend können

[6] http://oprofile.sourceforge.net

```
global syscalls

probe begin {
    printf("Starte das Sammeln von Daten... \n")
}

probe syscall.* {
    syscalls[name]++
}

probe end {
    foreach (name in syscalls- limit 10)
        printf("%10d %s\n",syscalls[name],name)
}
```

Listing 3.4: *SystemTap*-Skript "syscalls.stp" aus https://sourceware.org/systemtap/wiki/OLS2007SystemTapTutorial?action=AttachFile&do=get&target=SystemTap-Tutorial-OLS2007.pdf.

diese ausgegeben und ausgewertet werden.
Als Beispiel wird in Listing 3.4 ein *SystemTap*-Skript gezeigt, welches alle auftretenden *Systemcalls* zählt.
Weitere Informationen über *SystemTap* können der Projektwebseite[7] entnommen werden.

3.3.5. SysProf

Das *SysProf-Toolkit* ist besonders für eine Untersuchung von verteilten Systemen mit Client-Server-Architektur geeignet. Dabei ist das Ziel von *SysProf*, verteilte Software zu untersuchen, ohne genauere Kenntnisse über deren Architektur zu besitzen.
Es zeichnet alle Aktivitäten einer verteilten Anwendung auf, wobei eine Aktivität ein Systemcall eines Clients oder eine spezifische *Request-Response*-Situation zwischen einen Client und einem Server sein kann. Der Quellcode der untersuchten Anwendung wird dazu nicht benötigt und es ist keine erneute Übersetzung erforderlich.
Zur Aufzeichnung der Aktivitäten verwendet *SysProf* das Monitoring-Interface *Kprof*. Der Linux-Kernel wurde dahingehend angepasst, dass er an einigen Stellen Informationen an *Kprof* bereitstellt, die verwendet werden, um die Aktivitäten aufzuzeichnen [13].

[7] https://sourceware.org/systemtap/

3.3.6. Perf

Perf beinhaltet eine Sammlung von *Profiling*- und *Tracingwerkzeugen*. Es bietet die Möglichkeit das Auftreten bestimmter *Hardwareevents* aufzuzeichnen. Von der Funktionalität ähnelt es sehr *oProfile*. Somit können beispielsweise *Cache-Misses*, Speicher-Referenzen oder das Blockieren eines Prozesses aufgezeichnet werden [10]. Des Weiteren kann mit *Perf* bestimmt werden, welche Funktion einer Anwendung wie oft ausgeführt wird. Der zu der entsprechenden Funktion gehörige *Stacktrace* kann auch aufgezeichnet werden [16]. Mithilfe dieser Information kann *Perf* den prozentualen Anteil bestimmen, den jede einzelne Funktion in Ausführung ist. Die Zuverlässigkeit der Messung ist abhängig von der Frequenz, welche *Perf* nutzt, um zu prüfen welche Funktion momentan in Ausführung ist.
Perf kann zusätzlich sogenannte *Tracepoint*-Events aufzeichnen, die manuell in ein Programm eingefügt werden können. Dies bedeutet, dass Perf auch für eine Instrumentierung von Software verwendet werden kann.
Eine *Perfaufzeichnung* kann mit dem Befehl *perf record* gestartet werden. Die von *Perf* erzeugte Ausgabe kann nicht direkt vom Menschen gelesen werden und muss von einem weiteren Werkzeug wie z. B. *perf script* weiterverarbeitet werden. Die Ausgabe von Perf ist auch durch sogenannte *Flame-Graphs* darstellbar (vgl. Abbildung 3.1).
Weitere Informationen können dem kernel.org-Wiki[8] entnommen werden.

3.3.7. Performance Application Interface (PAPI)

Moderne Mikroprozessoren besitzen eine kleine Anzahl an Registern, die Hardwareevents zählen. Diese Register werden *Performance-Counter* genannt.
Prozessoren mit Intels *IA-32*-Architektur besitzen beispielsweise einen *Performance-Counter* namens "*L2_Reject_Cycles*"[9], welcher die Zyklen zählt, in denen der *L2-Cache* alle Zugriffsanfragen verwirft.
PAPI bietet für diverse Prozessorarchitekturen eine Benutzerschnittstelle und ermöglicht Programmen den Zugriff auf die *Performance-Counter*. Dadurch ist es möglich sehr ressourcensparend Informationen über *Hardwareevents* zu erhalten. Die *Performance-Counter* sind sehr geeignet für die Leistungsanalyse von Software auf tieferer Ebene, besonders für Übersetzeroptimierung, Verbesserung, *Benchmarking*, Fehlererkennung und *Profiling* [12]. Mit der Hilfe von *PAPI* können auch Programme geschrieben werden, die Informationen über eine zu untersuchende Anwendung sammeln, die für eine Leistungsanalyse hilfreich sind. *PAPI* kann aber auch beim *Tracing* unterstützend verwendet werden, um Zugriff auf zusätzliche Informationen zu erhalten, die anschließend

[8] https://perf.wiki.kernel.org/index.php/Tutorial
[9] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

ausgegeben werden.
Weitere Informationen über *PAPI* können der Projektwebseite[10] entnommen werden.

3.3.8. Flame-Graph

Flame-Graphs werden für eine bessere Darstellung und leichtere Analyse von *Profiling*-Ergebnissen verwendet. Die Ausgaben verschiedenster *Profiling*-Werkzeuge können mithilfe von Brendan Greggs Skript *flamegraph.pl* in *Flame-Graphs* umgewandelt werden. Von den bereits betrachteten Werkzeugen kann die Ausgabe von *Perf* in *Hot-Graphes* überführt werden, welche die Aufrufhäufigkeit aller Funktionen darstellen und die Ausgabe von *Offcputime* kann in *Cold-Graphs* überführt werden, welche die Prozessorabgabedauer während aller Funktionsaufrufe darstellen.
Weiterführende Informationen sind der Webseite von Brendan Gregg[11] und den folgenden Unterabschnitten zu entnehmen.

Hot-Graphs

Die mit der Hilfe von *Perf* generierten *Hot-Graphs* geben Auskunft darüber, wie häufig eine bestimmte Funktion aufgerufen wurde und woher dieser Funktionsaufruf kam. Da *Perf* Informationen über mehrere Prozesse aufzeichnen kann, können die einzelnen Funktionsaufrufe auch den entsprechenden Prozessen zugeordnet werden. In Abbildung 3.1 ist ein *Hot-Graph* abgebildet. Aufgrund der Skalierung dieser Arbeit ist der *Hot-Graph* schwierig zu lesen. Für eine bessere Betrachtung der Messergebnisse sei an dieser Stelle für alle *Flame-Graphs* auf den digitalen Anhang verwiesen.
Die Y-Achse bildet den *Stacktrace* einer jeden Funktion ab. Ein Rechteck im Graphen bildet einen Funktionsaufruf ab. Ein Rechteck auf einem Rechteck sagt aus, dass die obere Funktion von der darunterliegenden Funktion aufgerufen wurde. Die Funktion, die somit ganz oben auf allen Funktionen liegt, ist somit die Funktion, die zum Zeitpunkt der Untersuchung in Ausführung war.
Die X-Achse bildet die Häufigkeit eines jeden Funktionsaufrufs ab. Je breiter ein Rechteck ist, desto häufiger wurde die dargestellte Funktion aufgerufen. Alle Funktionsaufrufe mit gleichem *Stacktrace* werden zusammengeführt. Somit gibt es keinen Funktionsaufruf (mit gleichem *Stacktrace*) doppelt in einem *Flame-Graph*.
Die verschiedenen Farben des *Hot-Graphs* sagen nichts über die gemessenen Daten aus. Liegt die digitale Version des *Hot-Graphs* vor, sind genauere Informationen ermittelbar. Dabei sind exaktere Angaben über die absolute Anzahl und die relative Anzahl der Funktionsaufrufe ablesbar (vgl. Abbildung 3.3a). Die abgebildeten Funktionsaufrufs-

[10] http://icl.cs.utk.edu/papi/
[11] http://www.brendangregg.com/flamegraphs.html

häufigkeiten entsprechen allerdings nur den gemessenen Aufrufen und nicht der realen Anzahl der Funktionsaufrufe. Diese ist in der Praxis viel größer.

Cold-Graphs

Die mit der Hilfe von *Offcputime* generierten *Cold-Graphs* geben Auskunft darüber, wie lange der Prozessor während eines bestimmten Funktionsaufrufs abgegeben wurde. Aufgrund der Skalierung dieser Arbeit ist der *Cold-Graph* schlecht zu lesen.
Die Y-Achse bildet - genau wie beim *Hot-Graph* - den *Stacktrace* einer jeden Funktion ab. Die Funktion, die somit ganz oben aufliegt, ist die Funktion, die während der Prozessor entzogen wurde, in Ausführung war.
Die X-Achse bildet anders als bei den *Hot-Graphs* nicht die Häufigkeit, sondern die Prozessorabgabedauer einer bestimmten Funktion ab. Je breiter ein Rechteck, desto länger wurde der Prozessor abgegeben. Auch bei diesen *Flame-Graphs* werden die gemessenen Prozessorabgabezeiten einer Funktion aufsummiert.
Die verschiedenen Farben des *Hot-Graphs* sagen nichts über die gemessenen Daten aus. Liegt die digitale Version des *Cold-Graphs* vor, sind genauere Informationen ermittelbar. Dabei sind exaktere Angaben über die absolute Abgabezeit und die relative Abgabezeit ablesbar (vgl. Abbildung 3.3b).

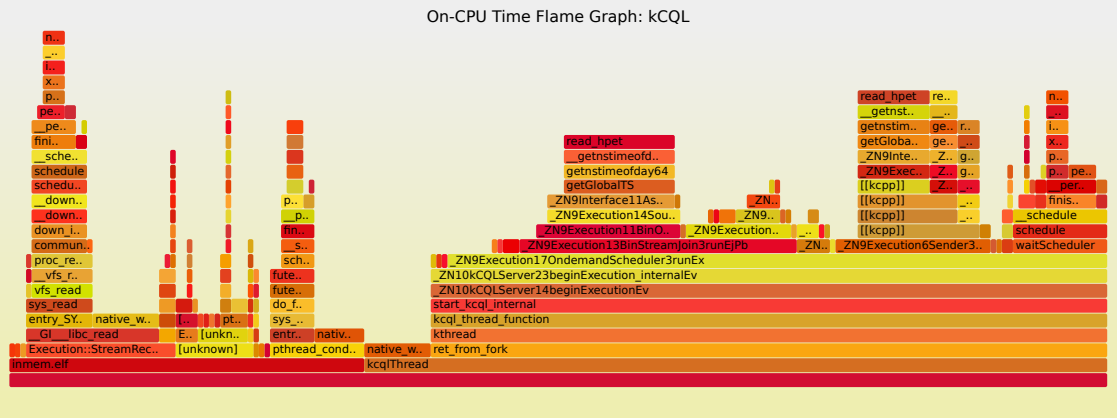


Abbildung 3.1.: Der abgebildete *Hot-Graph* wurde durch eine *Perfaufzeichnung* von kCQL erzeugt.

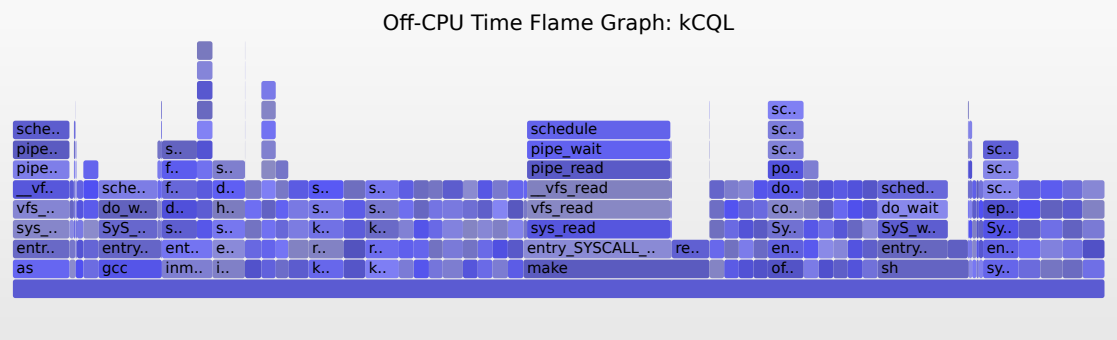
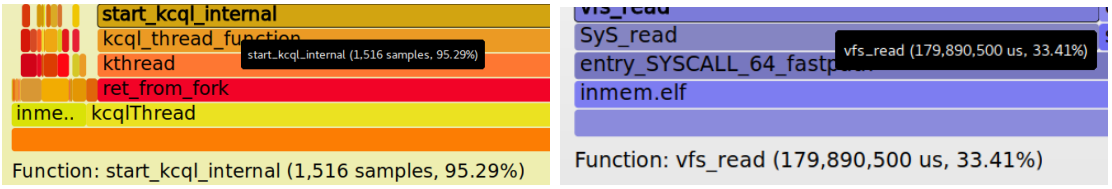


Abbildung 3.2.: Der abgebildete *Cold-Graph* wurde durch eine systemweite Offcputimeaufzeichnung erzeugt.



(a) *Hot-Graph* **(b)** *Cold-Graph*

Abbildung 3.3.: Darstellung der Informationen, die aus der digitalen SVG-Version entnehmbar sind und in der PDF-Version fehlen.

4. Durchführung der Leistungsanalyse

In diesem Kapitel wird die Durchführung der Leistungsanalyse von *kCQL* erörtert. Dabei wird in Abschnitt 4.1 eine kleine Vorbetrachtung der Analyse und deren Auswertung gemacht, welche für die spätere spezifischen Analysen wichtig ist.
Das restliche Kapitel besteht aus zwei Teilen. Zum einen wird bei der Durchführung zwischen der Lastanalyse in Abschnitt 4.2 und der Verarbeitungsgeschwindigkeitsanalyse in Abschnitt 4.3 unterschieden. Diese Kapitel sind wiederum in einige Unterkapitel unterteilt, auf die im weiteren Verlauf eingegangen wird.

4.1. Vorbetrachtung

Aus zeitlichen Gründen beziehen sich alle Untersuchungen in dieser Arbeit ausschließlich auf die *Netsimple*-Anfrage (vgl. Listing 2.3) von *kCQL*. Diese Anfrage verarbeitet Netzwerkpakete und gibt die dazugehörigen Prozesse und Sockets aus. Das betrachtete Vorgehen kann allerdings auch auf andere Anfragen übertragen werden. Eine Analyse der verwendeten Analyseverfahren geschieht nicht im Rahmen dieser Arbeit.
Wichtig für die Durchführung der Leistungsanalyse ist die Unterscheidung zwischen interner und externer Last. Zum einen beschreibt interne Last die Belastung, welche in der *kCQL*-Implementierung durch Daten die verarbeitet werden, erzeugt wird. Zum anderen beschreibt externe Last die Belastung des Systems durch andere Programme, die nicht direkt mit der *kCQL*-Implementierung in Verbindung stehen. Es wird vermutet, dass die Systemlast, welche *kCQL* erzeugt, von der internen Last generiert wird. Die folgende Lastanalyse untersucht unter anderem die Auswirkungen interner Last auf die von *kCQL* erzeugte Systemlast. Die externe Last hat nur einen indirekten Einfluss auf die von *kCQL* erzeugte Last.

4.2. Analyse der Last

Für eine Analyse der Last ist es sinnvoll Programmteile aufzuspüren, die häufige oder lange in Ausführung sind. Im speziellen wird angestrebt die Funktionen, welche viel Last

erzeugen zu erkennen. Dann können diese für eine Lastverringerung optimiert werden.

Welches Werkzeug für dieses Vorgehen verwendet wird, wird in Unterabschnitt 4.2.1 erörtert.
Zudem ist für eine Untersuchung der erzeugten Last das Vorhandensein interner Last wichtig, da *kCQL* vermutlich durch diese letztendlich Systemlast erzeugt. Diese Last und dessen Auswahl, wird in Unterabschnitt 4.2.2 genauer betrachtet.
Zum Schluss wird in Unterabschnitt 4.2.3 eine allgemeine Durchführung der Lastanalyse beschrieben, die auf beliebige Anfragen angewendet werden kann.

4.2.1. Auswahl des verwendeten Werkzeugs

Wegen des begrenzten Umfangs dieser Arbeit, wird zugunsten der Lastanalyse ein heuristischer Ansatz gesucht. Deswegen wird ein Werkzeug gewählt, welches ohne großen zusätzlichen Implementierungsaufwand häufig oder lange ausgeführte Programmteile erfassen kann. Um die Lastanalyse mit wenig Aufwand zu verbinden, wird die Auswahl auf Werkzeuge aus der Kategorie *Profiling* (vgl. Abschnitt 3.1) eingeschränkt, da diese in der Regel wenig Implementierungsaufwand erzeugen.
Für die Analyse wird *SystemTap* und *PAPI* ausgeschlossen, da das Schreiben von extra Skripten bzw. Programmen den zuvor beschriebenen Aufwand erzeugt, der eigentlich vermieden werden soll.
Auch *SysProf* wird nicht verwendet, da dieses Werkzeug seine volle Funktionalität erst in verteilten Client-Server-Anwendungen entfaltet. *KCQL* entspricht allerdings nicht dieser Struktur.
Offcputime wird nicht verwendet, da eine Abgabe des Prozessors in vielen Fällen keine Last erzeugt. Eine nicht optimale Abgabe verschiebt im schlimmsten Fall die Systemlast zeitlich nach hinten. Dies kann dazu führen, dass es lastintensivere Zeitpunkte gibt, die es sonst nicht gäbe. Dafür sind aber andere Zeitpunkte weniger lastintensiv und die Summe der erzeugten Systemlast bleibt unverändert. Ob diese Zeitpunkte der Mehrlast existieren ist mit *Offcputime* sehr schwer zu bestimmen.
Für die Lastanalyse sind besonders die Funktionen von *OProfile* und *Perf* interessant, die es ermöglichen regelmäßig den aktuellen *Stacktrace* eines Prozesses aufzuzeichnen. Durch eine regelmäßige Aufzeichnung des *Stacktraces*, können häufig verwendete und zeitaufwendige Funktionen ermittelt werden. Eine Funktion, die häufig in Ausführung ist, erzeugt auch mehr Systemlast als eine Funktion, die seltener in Ausführung ist. Bei der gemessenen Aufrufhäufigkeit der Funktion, kann somit auch von gemessener Last gesprochen werden.
Letztendlich fällt die Auswahl auf *Perf*, da dessen Ausgaben kompakter durch die *Hot-Graphs* dargestellt werden können. Könnten die Ausgaben von *oProfile* ohne Aufwand in *Flame-Graphs* überführt werden, wäre auch die Verwendung von *oProfile* denkbar.

4.2.2. Betrachtung der verwendeten Last

Um möglichst viele lastintensive Programmteile aufzuspüren, werden mehrere Messungen mit unterschiedlicher interner Last, mit und ohne externe Last durchgeführt. Es ist nämlich möglich, dass lasterzeugende Programmteile von der internen Last abhängig sind und somit mehrere Messungen gemacht werden müssen, um möglichst allgemeingültige Aussagen treffen zu können. Außerdem kann die Auswirkung externer Last auf die von *kCQL* erzeugte Last untersucht werden.

Das Vorgehen beim Erzeugen der internen Last ist abhängig von der untersuchten Anfrage. Damit *kCQL* unter Verwendung der *Netsimple*-Anfrage viel Last erzeugt, sollten die Quelldaten durch den Anfrageplan bis zum Ausgabeoperator laufen und von allen Operatoren auf dem Weg verarbeitet werden. Somit müssen für eine Analyse Daten erzeugt werden, welche diesen Weg durchlaufen.
Im Hinblick auf die *Netsimple*-Anfrage ist aus dem Anfrageplan erkennbar, dass die *Join*-Operatoren neue Daten zur Verfügung gestellt bekommen müssen, die mit dem *Join* kompatibel sind. Kompatibel sind Daten, welche die *Join*-Bedingung erfüllen. Die *Netsimple*-Anfrage besitzt drei verschiedene asynchrone Quellpuffer im *Kernelmodul*, die durch *KProbes* befüllt werden. Um also interne Last für die betrachtete Anfrage zu erzeugen, existiert die Möglichkeit den Prozess-, Socket- oder den Netzwerkpaket-Quellpuffer mit Daten zu befüllen. Dies geschieht durch die Erzeugung von Netzwerkpaketen, Sockets oder Prozessen. Dabei ist ersteres, die Erzeugung von Netzwerkpakete, den beiden anderen Varianten vorzuziehen. Ein Netzwerkpaket, dass zu einen Prozess und Socket passt, erzeugt nämlich den gewünschten Informationsdurchlauf von Quelle bis Ausgabe. Die Registrierung eines Sockets oder Prozesses alleine führt zu keiner Ausgabe. Dennoch erzeugt die Registrierung bzw. Deregistrierung von Sockets und Prozessen eine Systemlast die untersucht werden kann. Hierfür kann eine geschickt gewählte externe Last verwendet werden.
Um Netzwerkpakete zu generieren gibt es entweder die Möglichkeit echte Netzwerkpakete zu erzeugen oder die *kCQL*-Implementierung dahingehend abzuändern, dass der Netzwerk-Provider (vgl. Abschnitt 2.1) künstliche Datenpakete - sogenannte *Dummy*-Daten - erzeugt, die anschließend verarbeitet werden. Die Betrachtung beider Arten der internen Last ist interessant, um zu bestimmen, ob echte Daten höhere Last erzeugen als unechte. Eine genauere Implementierung der internen Lasterzeugung wird in Abschnitt 5.1 betrachtet.

Zusätzlich zur internen Last ist es ebenfalls interessant, die erzeugte Last der *kCQL*-Implementierung bei hoher externer Last zu betrachten. Durch diese macht der Prozessor viele Kontextwechsel, durch die *kCQL* den Prozessor oft abgeben muss. Dies führt zu einer geringeren Verarbeitungsgeschwindigkeit. Liefen nun Puffer bei einer Ausführung ohne externe Last leer, geschieht dies unter Umständen nicht mehr mit externer Last.

Außerdem erzeugt eine gut gewählte externe Last auch interne Last, denn sie kann dazu führen, dass beispielsweise Prozesse erzeugt werden und somit einige *Provider* von *kCQL* arbeiten müssen (vgl. Abschnitt 2.1). Dies zeigt dann ein etwas realistischeres Bild der erzeugten Systemlast.
Die benötigte externe Last wird in dieser Arbeit durch eine Übersetzung des Linux-Kernels erzeugt. Zum übersetzen des Kernels, werden so viele Prozesse wie Prozessoren verfügbar sind erzeugt, die jeweils einen Kern voll auslasten. Somit sind alle Prozessorkerne voll ausgelastet. Eine zusätzliche Messung der Übersetzungszeiten liefert eine Messgröße für die durch *kCQL* erzeugte Last.

Auswahl der Datenraten

Wie bereits angesprochen ist für die Lastanalyse das Verhalten von *kCQL* mit unterschiedlicher interner Last zu betrachten. Es werden Messungen mit drei verschiedenen Lastszenarien gemacht. Die echten Netzwerkpakete werden mit 10 Mbit/s, 100 Mbit/s und 800 Mbit/s empfangen. Die Übertragungsraten von 10 Mbit/s und 100 Mbit/s werden gewählt, da dies gängige Übertragungsraten in lokalen Netzwerken sind. Die dritte Übertragungsrate 800 Mbit/s wird gewählt, da sie Messungen zufolge die maximale Datenübertragungsrate ist, die durch den gegebenen Testaufbau möglich ist.
Um die Messergebnisse später besser vergleichen zu können, werden die Übertragungsraten der *Dummy*-Daten äquivalent zu den Übertragungsraten 10 Mbit/s, 100 Mbit/s und 800 Mbit/s gewählt. Hierfür wird die *Maximum Transfer Unit (MTU)* des sendenden Systems ermittelt, welche die maximale Größe eines Datenpakets in Byte angibt [17]. Diese liegt für das gegebene Testsystem bei 1500 Byte. Da die Datenpakete vom sendenden System immer voll sind, können die Frequenzen mit folgenden Formeln aus den Datenübertragungsraten berechnet werden:

$$dr[MB/s] = \frac{\ddot{U}bertragungsrate[Mbit/s]}{8}$$

$$Frequenz[Pakete/s] = \frac{dr}{MTU[Byte/Paket]} * 10^6[Byte/MB]$$

Durch diese Formel und Runden auf ganze Zahlen ergeben sich die Frequenzen 833 Pakete/s, 8333 Pakete/s und 66667 Pakete/s für die *Dummy*-Daten.

4.2.3. Durchführung

Im Hinblick auf die vorherigen Abschnitte ergibt sich folgendes Schema für eine Lastanalyse einer beliebigen *kCQL*-Anfrage:

1. Auswahl der zu untersuchenden Anfrage.

2. Bestimmung passender interner und externer Last.
3. Erzeugung der *Hot-Graphs*.
   - Mit externer Last.
   - Ohne externer Last.
4. Betrachtung der Funktionen, die am häufigsten ausgeführt werden.

Zuallererst wird die Anfrage ausgewählt, die untersucht werden soll. Anschließend wird die interne und externe Last so gewählt, dass sie zu der betrachteten Anfrage passt. Danach werden die *Hot-Graphs* entsprechend der zuvor bestimmten Lastszenarien erzeugt. Zum Schluss werden die erzeugten *Hot-Graphs* im Hinblick auf Funktionen mit häufiger oder langer Ausführung betrachtet.

Aus diesem allgemeinen Schema und den zuvor getroffenen Entscheidungen ergibt sich folgendes spezifischeres Vorgehen für die *Netsimple*-Anfrage, welches in Kapitel 6 angewendet wird:

1. Erzeugung der *Hot-Graphs* mit 10/100/800 Mbit/s und 833/8333/66667 Pakete/s.
   - Mit Linux-Kernel-Übersetzung.
   - Ohne Linux-Kernel-Übersetzung.
2. Betrachtung der am häufigsten ausgeführten Funktionen.

Mithilfe zweier *Bash-Skripte* werden *Hot-Graphs* mit den in Unterabschnitt 4.2.2 bestimmten Lasten erzeugt (vgl. Abschnitt 5.2). Anschließend werden die Ergebnisse im Hinblick auf Funktionen mit häufiger und langer Ausführung betrachtet. Diese Funktionen geben Aufschluss über Programmteile, die für eine Lastverringerung von *kCQL* optimiert werden können. Dies geschieht für die *Netsimple*-Anfrage in Kapitel 6.

4.3. Analyse der Verarbeitungsgeschwindigkeit

Durch eine genaue Betrachtung der *kCQL*-Implementierung ist erkennbar, dass ihre Verarbeitungsgeschwindigkeit sehr eng mit der Zeit verknüpft ist, die eine neue Information aus einem Quellpuffer (vgl. Abschnitt 2.1) benötigt, um Einfluss auf die Ausgabe einer Anfrage zu nehmen. Diese Zeit wird in dieser Arbeit genauer untersucht. Des Weiteren ist bekannt, dass das *Kernelmodul* während der Verarbeitung der *Netsimple*-Anfrage den Prozessor abgibt, wenn alle Puffer einer *kCQL*-Instanz voll oder leer sind. Bei leeren Puffern kann das *Kernelmodul* logischerweise nicht arbeiten, da

keine Quellinformationen vorliegen und keine Informationen zu verarbeiten sind. Bei vollen Puffern ist besonders der Puffer zwischen *Kernelmodul* und Useranwendung voll. Das bedeutet, dass keine Daten zur Useranwendung transportiert werden können, was Platz für eine weitere Verarbeitung der restlichen Puffer machen würde. Somit bleiben alle Puffer voll und für neue Informationen ist in den Quellpuffern kein Platz. Da eine Prozessorabgabe sehr teuer ist, ist eine Untersuchung der Abgabehäufigkeit interessant. Zu häufige Prozessorabgaben sollten nämlich vermieden werden.

In Unterabschnitt 4.3.1 wird zuallererst basierend auf diesen Annahmen eine kleine Vorüberlegung gemacht, welche die Auswahl des verwendeten Analysewerkzeugs in Unterabschnitt 4.3.2 beeinflusst.
Anschließend wird in Unterabschnitt 4.3.3 betrachtet, wie die Operatoren instrumentiert werden müssen, damit aussagekräftige Ergebnisse erzielt werden.
Daraufhin werden in Unterabschnitt 4.3.4 sinnvolle Lastszenarien festgelegt, die für die Analyse verwendet werden.
Zum Schluss wird in Unterabschnitt 4.3.5 erläutert, wie das allgemeine Vorgehen bei der Analyse der Verarbeitungsgeschwindigkeit aussieht.

4.3.1. Vorüberlegung

Um die Durchlaufzeit einer speziellen Anfrage zu untersuchen, ist es sinnvoll ihren erzeugten Anfrageplan zu betrachten. Die Zeit, welche eine Information vom asynchronen Quellpuffer (grafisch ganz oben) bis zum Ausgabe-Operator (grafisch ganz unten) benötigt (vgl. Kapitel 2), ist dabei besonders interessant. Diese Zeit wird im weiteren Verlauf dieser Arbeit Durchlaufzeit genannt. Es gilt diese Zeit zu minimieren, um die Verarbeitungsgeschwindigkeit zu verbessern. Dafür werden die Laufzeiten, aus denen die Durchlaufzeit besteht untersucht. Es muss die Zeit, welche jeder Operator benötigt, um eine einzelne Eingabe zu verarbeiten und die Zeit, welche benötigt wird, um eine Informationen zwischen beliebigen Operatoren zu transferieren, untersucht werden. Dabei wird zwischen Transporten zwischen Operatoren einer *kCQL*-Instanz (vgl. Abschnitt 2.1), den sogenannten *synchronen Transporten* und den Transporten zwischen asynchronen Quellpuffern bzw. den Transporten zwischen verschiedenen *kCQL*-Instanzen, den sogenannten *asynchronen Transporten* unterschieden.
Die Durchlaufzeit entspricht der Summe der Verarbeitungszeiten aller Operatoren und Transporte, die im Anfrageplan zwischen Ausgabe und asynchroner Quelle liegen. Die Operatoren und Transporte, die die längste Zeit benötigen, beeinflussen somit die Verarbeitungsgeschwindigkeit am stärksten und sollten optimiert werden.
Für die zuvor betrachtete Bestimmung der Häufigkeit der Prozessorabgaben ist eine Betrachtung des *kCQL*-Quellcodes wichtig. Es gibt einen Programmteil, der zu dieser

Prozessorabgabe führt. Diesen gilt es ausfindig zu machen, um die Ausführungshäufigkeit zu bestimmen.

4.3.2. Auswahl des verwendeten Werkzeugs

Grundsätzlich könnten viele Werkzeuge aus Kapitel 3 für eine Verarbeitungsgeschwindigkeitsanalyse verwendet werden. Wie bereits betrachtet muss das verwendete Werkzeug die Ausführungszeit bestimmter Programmteile bestimmen können. Da diese Programmteile recht unterschiedlich sein können, ist eine geschickte Änderung des *kCQL*-Quellcodes nötig. Dafür sind alle Werkzeuge aus der Kategorie manuelle Instrumentierung geeignet. Im speziellen kann *LTTng* dafür verwendet werden. Es verfälscht die Messdaten durch die bereitgestellten Infrastruktur nur geringfügig. Es ermöglicht die Bestimmung der genauen Zeit für die Verarbeitung eines eingehenden Datenpakets und der Teilverarbeitungszeiten der einzelnen Operatoren und Transporte auf dem Weg von Quellpuffer zum Ausgabe-Operator. Somit ist LTTng ein geeignetes Werkzeug.
Offcputime ist für die bereits angesprochene Untersuchung der Prozessorabgabe eine gute Wahl. Die Ausgabe mittels *Cold-Graph* würde für diesen Anwendungsfall allerdings wegfallen, da die aufsummierte Prozessorabgabezeit keinen Aufschluss über die Häufigkeit der Prozessorabgabe gibt. Allerdings kann auch mithilfe von *LTTng* die Häufigkeit der Prozessorabgabe bestimmt werden. Dies ist möglich, indem an der Programmstelle, an welcher der Prozessor abgegeben wird, immer ein *Tracepoint* ausgegeben wird. Diese ausgegebenen *Tracepoints* können gezählt werden und liefern somit die Häufigkeit der Prozessorabgaben. *Offcputime* bietet somit keinen nennenswerten Vorteil und wird nicht verwendet.

4.3.3. Instrumentierung von kCQL

Um mithilfe von *Tracepoints* eine Aussage über die Verarbeitungszeit einzelner Anfragen zu erhalten, muss jede Information, die den Anfrageplan durchläuft, eine eindeutige Identifikationsnummer (*ID*) zugeordnet bekommen. Diese *ID* wird anschließend beim Transport zwischen den Operatoren den transportierten Informationen mitgegeben. Erreicht eine Information einen Operator mit zwei Eingängen, dann werden beide *IDs* ausgegeben. Anschließend wird für alle weiteren Operatoren stellvertretend für diese zwei *IDs* eine neue *ID* generiert. Folglich kann eine Information, die über einen asynchronen Quellpuffer bis zum Ausgabe-Operator läuft auf dem Weg verschiedene *IDs* besitzen. Um alle Verarbeitungszeiten eines Tupels zwischen Ausgabe und Quellpuffer zu erhalten sind diese *IDs* essentiell wichtig. Ebendiese Verarbeitungszeiten sind für die Bestimmung der Durchlaufzeit einzelner Pakete, der Streuung der Durchlaufzeit und für die Darstellung einzelner Ergebnisse als Boxplot wichtig.
Jeder Operator arbeitet eine gewisse Anzahl an Informationen ab, wenn er ausgeführt

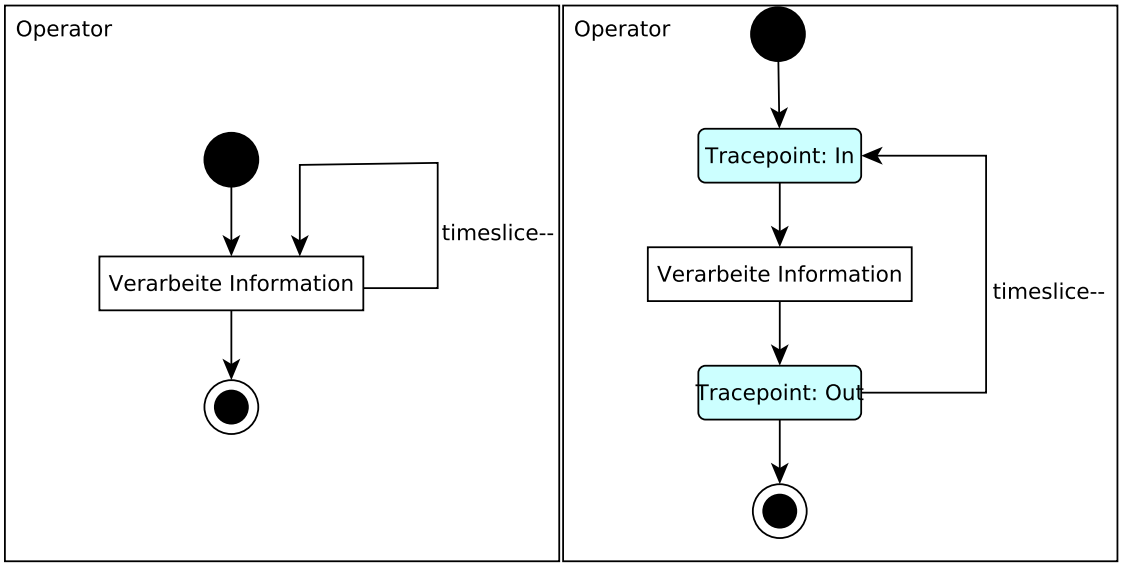


Abbildung 4.1.: Abstrakte Darstellung der Arbeit, die ein Operator verrichtet, wenn er vom *Scheduler* ausgeführt wird. In Abbildung 4.1b werden die Programmstellen blau dargestellt, die mit *Tracepoints* versehen werden.

wird. Wie viele Informationen ein Operator aus seinen Eingabe-Puffern verarbeiten kann, hängt von der Größe des vom *Scheduler* definierten *Timeslices* ab. Um an die Verarbeitungszeit einer Information zu gelangen, muss ein *Tracepoint* vor der Verarbeitung und ein *Tracepoint* nach der Verarbeitung dieser Information platziert werden. Diese *Tracepoints* geben die aktuelle *ID* der verarbeiteten Information aus. Sie sind im zweiten Teil von Abbildung 4.1 in blau dargestellt.

Jeder *Tracepoint* wird zusammen mit einen Zeitstempel ausgegeben. Es werden *Out*- und *In-Tracepoints* in *kCQL* eingefügt. Ein *Out-Tracepoint* ist ein *Tracepoint* am Ende der Verarbeitung einer Information, ein *In-Tracepoint* dagegen ist ein *Tracepoint* vor der Verarbeitung einer Information. Der *In-Tracepoint* wird ausgegeben nachdem die Informationen aus dem Eingabepuffer gelesen wird. Der *Out-Tracepoint* wird ausgegeben nachdem die Information in den Ausgabepuffer geschrieben wird. Die Verarbeitungszeit eines Operators ist ganz einfach mit folgender Differenz zu berechnen, wobei die Verarbeitungszeit t_v, der *Out*-Zeitstempel t_{out} und der *In*-Zeitstempel t_{in} entspricht.

$$t_v = t_{out} - t_{in}$$

Auch die Transportzeit zwischen zwei Operatoren aus dem Anfrageplan lässt sich auf ähnliche Weise berechnen. Hierfür beschreibt t_{tran} die Transportzeit von Operator 1

nach Operator 2, t_{out1} den *Out*-Zeitstempel von Operator 1 und t_{in1} den *In*-Zeitstempel von Operator 2:

$$t_{trans} = t_{in2} - t_{out1}$$

Die Transportzeit ist abhängig von der Dauer aller für den Transport notwendiger Operationen und vom *Scheduler*. Das schreiben in den Ausgabepuffer gehört aus technischen Gründen nicht zu der Transportzeit, sondern zur Verarbeitungszeit des Operators. Das Lesen aus dem Eingabepuffer gehört allerdings zur Transportzeit. Die Zeit, die zwischen einer Ausführung von Operator 1 und Operator 2 vergeht, ist abhängig von der *Scheduling*-Strategie und kann unter Umständen noch verbessert werden.

4.3.4. Lastszenarien

Eine geschickte Wahl der internen Last ist für eine detaillierte Untersuchung der Verarbeitungsgeschwindigkeit unerlässlich. Da der Anfrageplan von einem *Scheduler* ausgeführt wird, hat dieser auch einen großen Einfluss auf die Verarbeitungsgeschwindigkeit. Der *Scheduler* beeinflusst die Transportzeit, die ein Tupel von einem Operator zum nächsten benötigt. Zum jetzigen Zeitpunkt wird ein *Ondemand-Scheduler* verwendet. Jeder Operator aus dem Anfrageplan wird vom *Scheduler* betrachtet und ausgeführt. Hat ein Operator keine Informationen in seinen Eingabepuffern oder volle Ein- und Ausgabepuffer, dann wird er vom *Scheduler* übersprungen. Kann kein Operator abgearbeitet werden, dann gibt der *Scheduler* den Prozessor ab, bis er die Benachrichtigung erhält, dass es wieder ausführbare Operatoren gibt. Wie optimal das *Scheduling*-Verfahren in verschiedenen Lastsituationen für die Geschwindigkeit ist, lässt sich entweder durch eine detaillierte Analyse dieses Verfahrens, oder durch gezielte Messungen der Transportzeiten in unterschiedlichen Lastszenarien, prüfen. Im weiteren Verlauf dieser Arbeit wird der zweite Ansatz verfolgt.

Die Analyse der Verarbeitungsgeschwindigkeit wird grundsätzlich mit *Dummy*-Daten durchgeführt, da mit diesen die Übertragungsrate besser gesteuert und auch über das Limit einer Netzwerkkarte hinaus erhöht werden kann. Außerdem wird die Analyse ohne externe Last durchgeführt, da diese keinen Mehrwert bei der Analyse der Geschwindigkeit bietet.
Für die Ermittlung der Verarbeitungsgeschwindigkeit werden - wie unter anderem bei der Lastanalyse - *Dummy*-Datenpakete erzeugt. Es werden 500000 Eingaben erzeugt, die verarbeitet werden und zu einer Ausgabe führen. Zusätzlich werden zwei *Dummy*-Sockets und zwei *Dummy*-Prozesse erzeugt. Jedes zweite eingehende *Dummy*-Datenpaket gehört zu einem anderen Prozess und Socket. Die Transportzeiten und Verarbeitungszeiten dieser *Dummy*-Daten werden anschließend gemittelt und analysiert. Netzwerkpakete die aufgrund voller Puffer verworfen werden, werden nicht zu den 500000 Paketen gezählt.

Wichtig ist dabei zu beachten, dass alle Operatoren und Transporte, welche Sockets und Prozesse verarbeiten in diesem Fall lediglich zwei Messwerte besitzen, aus denen der Mittelwert berechnet wird. Zusätzlich wird die mittlere Durchlaufzeit pro Paket bestimmt.
Des Weiteren wird gezählt, wie oft der *kCQL-Scheduler* den Prozessor abgegeben hat. Die Messungen werden einmal mit 1000 Pakete/s, 10000 Pakete/s und 100000 Pakete/s gemacht. Das 1000 Pakete/s und 10000 Pakete/s Lastszenario sind zwei Szenarien, die vorigen Tests zufolge *kCQL* nicht voll auslasten. Es kommt somit zu Prozessorabgaben seitens *kCQL*, da es nichts zu tun gibt. Es ist interessant diese Lastszenarien mit dem 100000 Pakete/s Lastszenario zu vergleichen. Dieses lastet nämlich *kCQL* voll aus. *KCQL* erzeugt somit während der Verarbeitung 100% Prozessorauslastung. Im optimalen Fall sollten die Verarbeitungszeiten aller Lastszenarien gleich sein, bei steigender Last könnte sie eventuell etwas ansteigen. Es wird geprüft, ob sich die *kCQL*-Implementierung optimal verhält.
Zusätzlich zu den bereits besprochenen Lastszenarien, wird eine Messung gemacht, bei der 1000 *Dummy*-Pakete so schnell wie möglich erzeugt werden und anschließend 1 Sekunde gewartet wird, bevor die nächsten 1000 *Dummy*-Pakete erzeugt werden. Dieses Lastszenario wird *Burstmodus* genannt. Dieses Lastszenario verarbeitet rechnerisch Pakete mit einer Geschwindigkeit von maximal 1000 Pakete/s. Es unterscheidet sich vom zuvor betrachteten Szenario, da die Übertragungsrate stark schwankt und eher eine obere Schranke ist. Anhand dieses Lastszenarios kann bestimmt werden, ob sich schwankende Geschwindigkeiten auf die Verarbeitungsgeschwindigkeit auswirken.
Für die Untersuchung der Prozessorabgabehäufigkeit wird zusätzlich zu den betrachteten Lastszenarien eine Messung mit 150000 Pakete/s gemacht. Dieses Lastszenario lastet genau wie das 100000 Pakete/s Szenario den Prozessor voll aus. Es bietet allerdings nochmals eine Steigerung. Dies wird gemacht, um zu betrachten, wie sich die Häufigkeit der Prozessorabgaben bei steigender interner Last verhält.

4.3.5. Durchführung

Aufgrund der bereits betrachteten Vorüberlegungen ergibt sich für die Untersuchung der Verarbeitungsgeschwindigkeit mithilfe von *LTTng* folgendes allgemeine Vorgehen:

1. Auswahl der zu untersuchenden Anfrage.
2. Erzeugung des Anfrageplans.
3. Instrumentierung aller vorhandener Operatoren.
4. Auswahl geeigneter Last.
5. Analyse der Ergebnisse

Zuallererst muss die Anfrage, die untersucht werden soll ausgewählt werden. Abhängig von dieser wird anschließend der Anfrageplan betrachtet und alle Operatoren, die aus diesem ersichtlich sind werden instrumentiert. Anschließend müssen geeignete Lastszenerien gewählt werden. Zum Schluss werden die Ergebnisse analysiert. In dieser Analyse geht es darum die Verarbeitungszeiten und Transportzeiten aller Operatoren zu bestimmen und zusätzlich die Häufigkeit der Prozessorabgaben vom *kCQL-Scheduler* zu ermitteln.

Die Leistungsanalyse ist in dieser Arbeit auf die *Netsimple*-Anfrage beschränkt. Von daher fällt die Auswahl der zu untersuchenden Anfrage weg. Alle anderen Schritte bleiben gleich.

5. Implementierung

Dieses Kapitel beschäftigt sich mit einigen Implementierungsdetails, die für eine Leistungsanalyse interessant sind. Dabei erläutert Abschnitt 5.1 wie *kCQL* abgeändert wird, damit *Dummy-Daten* erzeugt werden können. In Abschnitt 5.2 wird anschließend auf das Skript eingegangen, welches bei der Lastanalyse für die *Flame-Graph*-Erzeugung zuständig ist. Abschließend beschreibt Abschnitt 5.3 die Implementierung der *Tracepoints* in *kCQL* und zum Teil dessen Auswertung.

5.1. Erzeugung der Last

Im folgenden wird auf die Implementierung der Erzeugung der Netzwerkpakete eingegangen. Dabei wird zwischen der Erzeugung echter und künstlicher Netzwerkpakete unterschieden.

Dummy-Daten

In Abbildung 5.1 wird der schematische Ablauf für die *Dummy*-Datenerzeugung abgebildet. Beim Start des *kCQL-Kernelmoduls* werden Informationen über *Dummy*-Sockets in den Quellpuffer für die Sockets und Informationen über *Dummy*-Prozesse in den Quellpuffer für die Prozesse geschrieben. Alle Puffer sind in der Abbildung gelb dargestellt. Anschließend wird ein *hrtimer* erzeugt. Dieser ist in der Abbildung grün dargestellt. Er führt in einem regelmäßigen Zeitintervall eine Funktion aus. Der *hrtimer* wird verwendet, um in einem regelmäßigem zeitlichen Abstand, der abhängig von der gewählten Frequenz ist, *Dummy*-Netzwerkpakete zu generieren und diese in den Quellpuffer für eingehende Netzwerkpakete zu schreiben. Diese erzeugten Netzwerkpakete erhalten als zugehörigen Prozess und Socket einen zuvor generierte *Dummy*-Prozess und *Dummy*-Socket. Im speziellen werden in dieser Arbeit zwei *Dummy*-Sockets und zwei *Dummy*-Prozesse erzeugt. Jedes eingehende Datenpaket wird dem Socket und Prozess zugeordnet, dem das vorangegangene Datenpaket nicht zugeordnet war.

Als Alternative zur Verwendung eines *hrtimers*, wäre auch die Erzeugung eines neuen Prozesses denkbar, der mit Hilfe von aktiven oder passiven Wartens die gewünschten Informationen in den Zielpuffer schreibt. Allerdings ist eine Implementierung mit aktiven

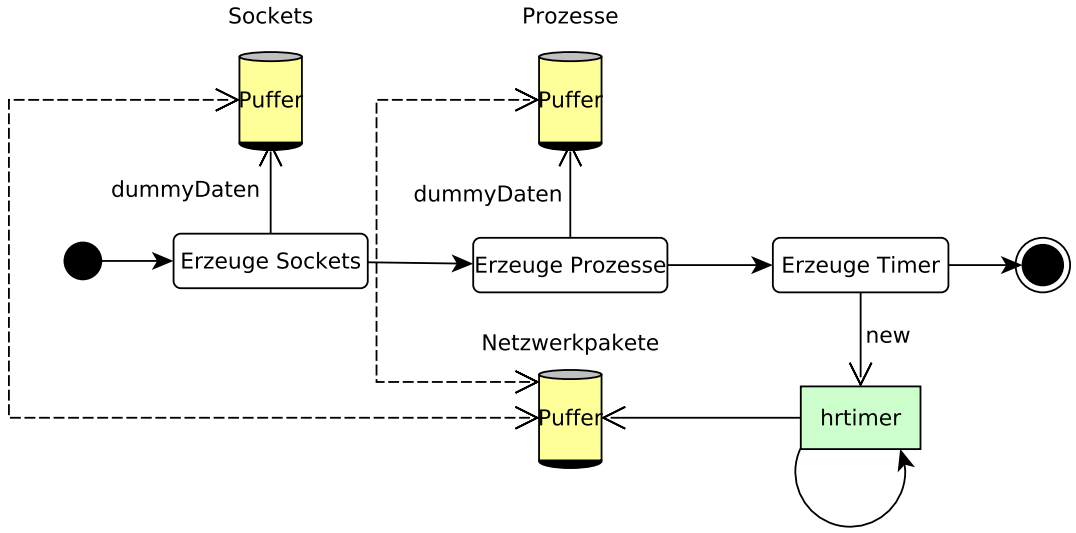


Abbildung 5.1.: Pseudo UML-Aktivitätsdiagramm, dass die Generierung der internen Last mit *Dummy*-Daten darstellt.

Warten, die einen Prozessorkern blockiert, nicht akzeptabel. Außerdem ist die Verwendung eines *hrtimer* einer manuellen regelmäßigen Prozessorabgabe vorzuziehen, falls regelmäßig eine sehr kurze Zeit gewartet werden muss [18].

Netzwerkpakete

Das Vorgehen beim Generieren der echten Netzwerkpakete ist aus Abbildung 5.2 ersichtlich. Das Testsystem, auf dem alle Leistungsanalysen durchgeführt werden und auf dem die *kCQL*-Implementierung läuft, wird *Eval1* genannt. Das System, dass die Netzwerkpakete an *Eval1* sendet wird *Eval2* genannt. Dieses System ist mittels einer *LAN*-Verbindung mit dem *Eval1*-System verbunden und wartet ununterbrochen auf einen Verbindungsaufbau. Wird eine Verbindung aufgebaut, sendet das *Eval2*-System so viele Daten wie möglich über die aufgebaute Netzwerkverbindung. Bei dieser handelt es sich um eine *TCP*-Verbindung. Als Nutzdaten werden kontinuierlich Nullen gesendet. Dem *Eval2*-System kann eine Datenübertragungsrate vom *Eval1*-System zugewiesen werden. Das *Eval2*-System sendet anschließend Pakete mit maximal der gesetzten Datenübertragungsrate.

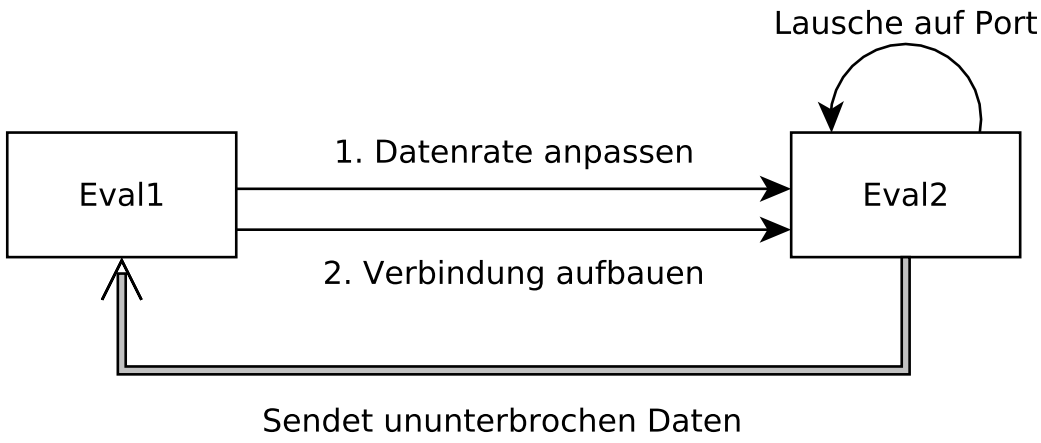


Abbildung 5.2.: Generierung der internen Last mit echten Daten.

5.2. Erzeugung der Flame-Graphs

Während der Lastanalyse mit externer Last, zeichnet Perf im Verlauf der gesamten Übersetzung des Linux-Kernels Daten auf. Zusätzlich wird die Dauer der Kernelübersetzung aufgezeichnet. Ohne externe Last dauert die Perfaufzeichnung 180 Sekunden. Dies entspricht ungefähr der mittleren Übersetzungszeit des Kernels. Somit können Messungen ohne externe Last trotzdem mit den Messungen mit externer Last verglichen werden. Eine Perfaufzeichnung zeichnet 500 *Stacktraces* pro Sekunde auf. Dabei gehören nicht alle *Stacktraces* zur untersuchten Anwendung.
Das Vorgehen bei einer Leistungsanalyse mit *Perf* ist in Abbildung 5.2 als Pseudo-*UML*-Aktivitätsdiagramm dargestellt. Zuallererst muss entschieden werden, ob die *kCQL*-Implementierung mit erzeugten *Dummy*-Daten, oder mit echten Netzwerkpaketen als interne Last untersucht werden soll. Dementsprechend wird die *kCQL*-Implementierung anschließend übersetzt und im Falle der Verwendung echter Datenpakete wird die Datenübertragung mit der ausgewählten Frequenz angestoßen. Anschließend wird eine Perfaufzeichnung gestartet. Woraufhin entschieden werden muss, ob externe Last während der Aufzeichnung existieren soll oder nicht. Diese Entscheidung beeinflusst, ob eine Linux-Kernel-Übersetzung gestartet wird oder lediglich 180 Sekunden gewartet wird. Zum Schluss muss die gestartete Aufnahme gestoppt werden und der entsprechende *Flame-Graph*-Typ aus der Ausgabe generiert werden.
Diese Schritte sollten schnell hintereinander ausgeführt werden, weswegen ein *Bash-Skript* verwendet wird. Es existieren zwei unterschiedliche *Bash-Skripte* für die Untersuchung mit *Dummy*-Daten und echten Daten. Das Skript für die Untersuchung mit echten Da-

ten ist eine abgeänderte Version eines vom Lehrstuhl bereitgestellten Evaluierungsskripts. Das zweite Skript basiert auf diesem Evaluierungsskript. Beide Skripte erwarten als Argument Informationen über die Anzahl der Durchläufe und ob externe Last vorhanden sein soll.

Probleme

Bei der Erzeugung der *Hot-Graphs* taucht das Problem auf, dass einige Funktionen im *Stacktrace* nicht korrekt benannt sind. Der Grund für diese Fehlbenennung hängt mit der Verwendung des C++- und C-Übersetzers und der anschließenden Zusammenführung beider Ausgaben zusammen. Es wird angenommen, dass diese Fehlbenennung keine kritischen Auswirkungen auf die Analyse der Ergebnisse hat, da die betroffenen Funktionen nicht am oberen oder unteren Ende des *Stacktraces* sind. Sollte der genaue *Stacktrace* für eine Analyse relevant sein, wird dieser Anhand des restlichen *Stacktraces* rekonstruiert.

5.3. Implementierung der Tracepoints

In diesem Abschnitt wird auf die in Kapitel 4 besprochene Instrumentierung der *kCQL*-Implementierung eingegangen. Die *kCQL*-Implementierung wird dahingehend verändert, dass jedes Tupel, das von einem Operator zu einen anderen transportiert wird, einen Integerwert mehr enthalten darf. Diese Erweiterung ist nötig, um jedem Tupel eine *ID* mitzugeben. Diese *IDs* erlauben das Nachvollziehen des Tupelverlaufes durch den von *kCQL* erzeugten Anfrageplan.
Wenn eine Information in *kCQL* entsteht, erhält diese eine *ID*. Wird eine Information lediglich weitergegeben, dann behält sie in der Regel die vorige *ID*. Besonders wenn ein Operator zwei Eingaben zusammenführt, wird eine neue *ID* für dessen Ausgabe erzeugt. Da die Erzeugung einer neuen *ID* in jeder *kCQL*-Instanz geschehen kann, es aber sehr schwierig ist die zuletzt vergebene *ID* einer jeden *kCQL*-Instanz allen anderen Instanzen mitzuteilen, können *IDs* mehrfach in verschiedenen Instanzen vergeben werden und haben nur innerhalb einer Instanz eine Aussagekraft. Deswegen erhält eine Information nachdem sie von einer Instanz zur nächsten transportiert wird eine neue *ID*.
Mithilfe der *IDs* kann die Durchlaufzeit einer Information ermittelt werden. Dies ermöglicht bei Bedarf die Betrachtung jeder einzelnen Durchlaufzeit.
Jeder Operator besitzt eine Funktion *run*, die vom *Scheduler* ausgeführt wird. In dieser Funktion wird eine bestimmte Anzahl von Eingaben bearbeitet. Diese Eingaben hängen mit dem definierten *Timeslice* des *Schedulers* zusammen. In der momentanen Implementierung werden zehn Eingaben verarbeitet, wenn ein Operator vom *Scheduler* ausgeführt wird. Um nun die Verarbeitungszeit einer einzelnen Eingabe zu erhalten, werden zwei

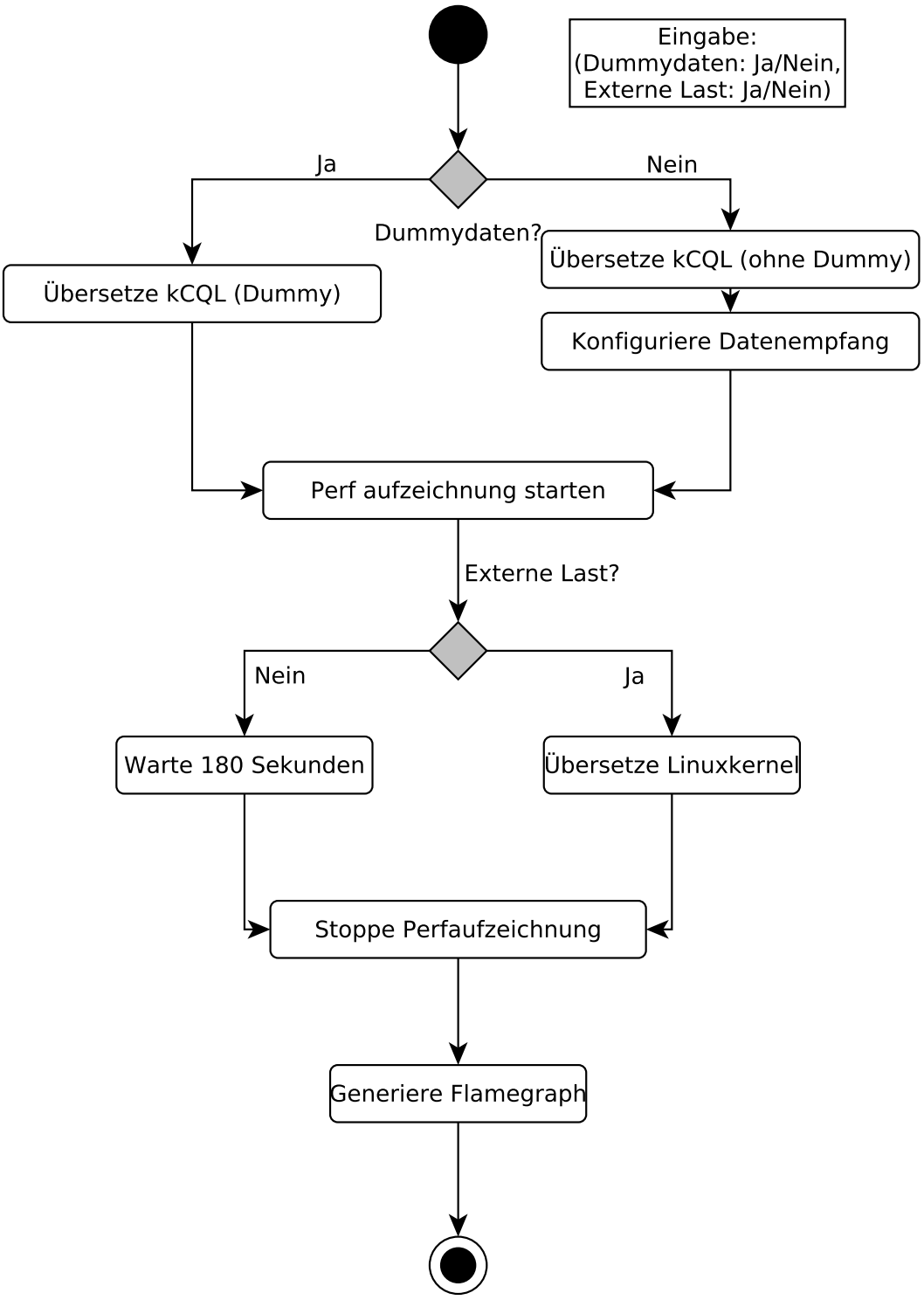


Abbildung 5.3.: Vorgehensweise für die Erzeugung der *Hot-Graphs*.

Tracepoints in der Funktion *run* eines jeden relevanten Operators implementiert: Ein *In-Tracepoint* an der Stelle, nachdem eine Eingabe zur weiteren Verarbeitung eingelesen wird und ein *Out-Tracepoint* nachdem die Ausgabe in den Ausgabepuffer geschrieben wird.

Es werden 4 verschiedene Arten von *Tracepoints* in der *kCQL*-Implementierung definiert und verwendet. Dabei handelt es sich um jeweils einen *binären* und *unären Tracepoint* in der Useranwendung und einen *binären* und *unären Tracepoint* im Kernelmodul. Ein *unärer Tracepoint* wird bei den Operatoren als *In-Tracepoint* verwendet, die lediglich einen Eingabepuffer besitzen. Er gibt zusätzlich zu der Information zu welchem Operator er hört die *ID* der momentan zu verarbeitenden Eingabe aus. Ein binärer *Tracepoint* wird dagegen verwendet, wenn ein Operator zwei Eingabepuffer besitzt. Dabei liefert er die Information zu welchem Operator er gehört, gibt die beiden *IDs* seiner zu verarbeitenden Eingaben und eine neu erzeugte *ID* aus.

Der *kCQL-Scheduler* besitzt einen Programmteil, der den Prozessor abgibt, wenn zuvor eine Variable gesetzt wurde. Zusätzlich zu der Prozessorabgabe wird in diesem Programmteil ein *Tracepoint* hinzugefügt. Folglich wird immer wenn *kCQL* den Prozessor freiwillig abgibt ein *Tracepoint* ausgegeben.

Die *Tracepointausgaben* werden in eine Textdatei geschrieben, die von einem selbstgeschriebenen Programm ausgewertet wird. Dieses Programm erzeugt eine Ausgabe, die zu jedem Operator und jedem Transport im Anfrageplan die durchschnittliche Verarbeitungszeit in Nanosekunden berechnet. Dafür wird der Mittelwert der Verarbeitungsgeschwindigkeit aller verarbeiteten Informationen gebildet. Zusätzlich gibt dieses Programm zu jedem Mittelwert eine Varianz als Maß der Streuung und die Häufigkeit der Prozessorabgaben aus. Die Prozessorabgaben vom Kernelmodul und von der Useranwendung können sehr einfach ermittelt werden, indem die *Tracepoints*, welche vor der Abgabe ausgegeben wurden, gezählt werden.
Dieses Programm kann mit einigen Abänderungen von jeder Anfrage die *Tracepointausgabe* verarbeiten. Hierfür benötigt es zusätzlich den Anfrageplan der untersuchten Anfrage als Eingabeparameter.

6. Evaluierung

Dieses Kapitel beschäftigt sich mit der Auswertung der in Kapitel 4 beschriebenen Messungen. Die Testkonfiguration des Systems auf dem die Messungen durchgeführt werden, wird in Abschnitt 6.1 beschrieben. Des Weiteren werden die Messungen für die Lastanalyse in Abschnitt 6.2 und die Messungen für die Analyse der Verarbeitungsgeschwindigkeit in Abschnitt 6.3 betrachtet.

6.1. Testkonfiguration

Auf dem für die Untersuchung verwendetem Testsystem läuft das Betriebssystem "Ubuntu 14.04.5 LTS". Es hat einen "Intel(R) Core(TM) i5-3570" Prozessor mit 3.40 GHz für jeden seiner vier Kerne und ihm stehen 7,5 GB Arbeitsspeicher zur Verfügung.
Das Testsystem ist mit 1 Gbit/s Netzwerkkarten an einem weiteren Testsystem angeschlossen. Dieses zweite System wurde zuvor in dieser Arbeit als Eval2-System bezeichnet und dient zur Netzwerkpaketerzeugung.

6.2. Analyse der Last

Dieser Abschnitt beschäftigt sich mit der Lastanalyse der *Netsimple*-Anfrage von *kCQL*. In Unterabschnitt 6.2.1 wird begründet, wieso alle gemessenen Daten korrekt und geeignet für die Untersuchung sind. Danach wird in Unterabschnitt 6.2.2 auf die gemessenen Daten eingegangen. Anschließend werden in Unterabschnitt 6.2.3 die verschiedenen Messdaten betrachtet. Am Ende wird in Unterabschnitt 6.2.4 auf die Ergebnisse der Messungen eingegangen und Funktionen genannt, die optimiert werden können.

6.2.1. Korrektheit

Die Messdaten der Lastanalyse werden mit *Perf* aufgezeichnet und durch *Hot-Graphs* dargestellt. *Hot-Graphs* stellen lediglich die Summe aller Funktionsaufrufe dar. Das bedeutet, dass aus diesen lediglich die Summe der gemessenen Last, die eine Funktion erzeugt, ablesbar ist. Die zeitliche Verteilung der gemessenen Last kann nicht abgelesen

werden.
Die Messungen mit *Perf* können dazu führen, dass *kCQL* der Prozessor häufiger entzogen wird. Dies kann die zeitliche Verteilung der erzeugten Last beeinflussen. Die Summe der erzeugten Last bleibt allerdings identisch und wird nicht beeinflusst. Deswegen wirkt sich die *Perfaufzeichnung* selbst kaum auf das gemessene Ergebnis aus.
Wenn *Perf* einen oder mehrere Prozesse erzeugt, werden Einträge in der Prozessrelation von *kCQL* hinzugefügt. Da diese aber keinen großen Einfluss auf die Verarbeitung nimmt und in allen Messungen im gleichen Maße auftritt, kann dieses Detail vernachlässigt werden.
Alles in allem kann gesagt werden, dass die betrachten *Hot-Graphs* ein gutes Mittel zur Bestimmung der erzeugten Systemlast sind. Durch die relativen Messergebnisse der *Hot-Graphs* können trotz geringer Belastung des Systems lasterzeugende Teile der *kCQL* Implementierung ermittelt werden.

6.2.2. Metriken

Eine vollständige statistische Analyse und die damit zusammenhängende Entwicklung der dazu benötigten Evaluierungswerkzeuge würde den Umfang dieser Arbeit übersteigen. Die Erzeugung einer repräsentativen Anzahl an Messdaten würde länger als der Zeitrahmen dieser Arbeit dauern. Deswegen wird zugunsten der Lastanalyse darauf verzichtet. Stattdessen werden für jedes Lastszenario fünf *Perfaufzeichnungen* gemacht, die mittels *Hot-Graphs* visualisiert werden. Für jedes zu untersuchende Lastszenario wird ein *Hot-Graph* als Repräsentant ausgewählt. Für die Auswahl dieses Repräsentanten wird die absolute Häufigkeit aller Funktionsaufrufe des Prozess *kcqlThread* betrachtet. Hier wird der *Hot-Graph* des Lastszenarios mit der mittleren absoluten Aufrufhäufigkeit[1] als Repräsentant verwendet.

6.2.3. Messdaten

Für die Untersuchung der Systemlast wird die Häufigkeit aller Funktionsaufrufe der *kCQL*-Implementierung während ihrer Ausführung der *Netsimple*-Anfrage betrachtet. Dabei erzeugt eine häufig aufgerufene Funktion im System mehr Last, als eine Funktion, die selten aufgerufen wird. Es kann somit auch von gemessener Last gesprochen werden. Im weiteren Verlauf wird zwischen absoluter und relativer Last unterschieden. Wenn Prozentangaben zur Beschreibung der gemessenen Last verwendet werden, dann ist immer die relative Last gemeint. Werden anderenfalls Zahlen zur Beschreibung der Last verwendet, wird von Funktionsaufrufen und somit von der absoluten Last gesprochen. In den meisten Fällen werden allerdings die relativen Lasten miteinander verglichen, da

[1] Median

die absoluten Werte aus unterschiedlichen Gründen voneinander abweichen können.
Zu jeden der in Unterabschnitt 4.2.2 vorgestellten Lastszenarien wurde eine Messung gemacht, die im folgenden betrachtet und verglichen wird. Die *Hot-Graphs*, die für die Lastszenarien als Repräsentanten ausgewählt werden sind im Anhang A zu finden.
Bereits bei einer groben Betrachtung der *Hot-Graphs* aller Lastszenarien fällt auf, dass bei der *Netsimple*-Anfrage der Prozess "inmem.elf", welcher der Useradressbereichs-Instanz der *kCQL*-Implementierung entspricht, sehr selten in Ausführung ist. In einigen *Hot-Graphs* ist der Prozess nicht einmal erkennbar, da die Beschriftung nicht mehr in das zu kleine, umliegende Rechteck passt. Dies lässt den Schluss zu, dass dieser Prozess für die *Netsimple*-Anfrage nur eine verschwindend geringe Last erzeugt. Somit wird er im folgenden nicht weiter betrachtet.
Des Weiteren ist auffällig, dass der Prozess "*kcqlThread*" am häufigsten die Funktion *run*[2] vom *Scheduler* ausführt. Diese wiederum führt die Funktion *run* von verschiedenen Operatoren aus dem Anfrageplan aus, welche zu der Last der *Scheduler-run*-Funktion führen. Diese Funktionen bieten somit einen ersten Ansatz für die Betrachtung.

Ohne externe Last

Im folgenden werden die Messungen betrachtet, die ohne externe Last gemacht wurden. Dabei werden zum einen die *Flame-Graphs* aus Abbildung A.1, Abbildung A.2 und Abbildung A.3 mit echter Last und zum anderen die *Flame-Graphs* aus Abbildung A.4, Abbildung A.5 und Abbildung A.6 mit *Dummy*-Daten als interne Last betrachtet.
Zuerst werden die *Perfaufzeichnungen* mit echter Last und anschließend die Aufzeichnungen mit *Dummy*-Daten ausgewertet.

Bei den Messungen mit 10 Mbit/s und 100 Mbit/s erzeugt die Funktion *run* der beiden *Join*-Operatoren des *Netsimple*-Anfrageplans 34%-35% der gemessenen Last. Bei der Messung mit 800 Mbit/s erzeugt die Funktion *run* des Sender-Operators wiederum eine größere relative Last. Bei einer genaueren Betrachtung der weiteren *Stacktraces* fällt allerdings auf, dass die Funktion *getGlobalTS* in jedem Lastszenario sehr oft aufgerufen wird. Sie ist an der Spitze verschiedener *Stacktraces* und fällt somit erst bei genauerem Hinsehen auf. Bei einer Aufsummierung der relativen Häufigkeit aller *getGlobalTS* Aufrufe ist erkennbar, dass diese Funktion mehr gemessene Last erzeugt, als die Funktion *run* eines speziellen Operators. Bei der Messung mit 10 Mbit/s interner Last erzeugt sie 27,50%, bei der Messung mit 100 Mbit/s interner Last 44,00% und bei der Messung mit 800 Mbit/s 51,62% der gemessenen Last. Die gemessene relative Last steigt sogar bei steigender interner Last. Dies ist ein Anzeichen dafür, dass diese Funktion für einen großen Teil der von *kCQL* erzeugten Systemlast verantwortlich ist.
Zusätzlich fällt die Funktion *waitScheduler* auf. Sie erzeugt in allen Lastszenarien mit

[2] _ZN9Execution17OndemandScheduler3runEx

echten Datenpaketen zwischen 15% und 21% der gemessenen Last. Das ist fast ein Viertel der erzeugten Last und macht diese Funktion für eine weitere Betrachtung sehr interessant.

Die Messergebnisse mit *Dummy*-Daten als interne Last sind denen der Messungen mit echten Netzwerkpaketen sehr ähnlich. Auch hier sind die Funktionen *getGlobalTS* und *waitScheduler* auffällig. Zusätzlich ist erkennbar, dass die gemessene absolute Gesamtlast größer ist, als bei den Messungen mit echten Daten. Die Herkunft dieses Unterschiedes ist nicht bekannt. Es könnte zwar sein, dass der *kCQL*-Prozess bei echten Daten öfter durch Systemunterbrechungen oder *KProbes* Verdrängt wird, die für das Empfangen von eingehenden Paketen zuständig sind. Da das Testsystem allerdings mehrere Prozessorkerne besitzt, die diese Aufgabe bewältigen könnten, ist diese Erklärung unwahrscheinlich. Außerdem kann dies auch durch die bei den *Dummy*-Daten verwendeten *hrtimer* geschehen.
Zusätzlich ist erkennbar, dass die gemessene relative Last des Prozesses "*inmem.elf*" bei den Messungen mit *Dummy*-Daten minimal kleiner ist. Das ist ein Effekt der bereits betrachteten größeren absoluten Last des *Kernelthreads*. Weiterhin fällt auf, dass bei der Messung mit 66667 Pakete/s im Gegensatz zu der Messung mit 800 Mbit/s die Funktion *service_schedule* mit 34.38% sehr viel Last erzeugt. Dagegen tauchen die *getGlobalTS* und *waitScheduler* Funktion überhaupt nicht auf. In allen anderen Lastszenarien dagegen verursacht *service_schedule* weniger als 7% der gemessenen Last. Um einen Messfehler handelt es sich dabei allerdings nicht, da die Messungen mehrfach mit ähnlichen Ergebnissen wiederholt wurden. Deshalb wird angenommen, dass die Messung korrekt ist. Dies lässt den Schluss zu, dass *service_schedule* bei größeren Lasten zu einem Problem führt und viel Systemlast erzeugt.

Mit externe Last

Im folgenden werden die erzeugten *Hot-Graphs* untersucht, die durch Messungen mit externe Last entstanden sind. Dabei werden zum einen die *Flame-Graphs* aus Abbildung A.7, Abbildung A.8 und Abbildung A.9 mit echter und zum anderen die *Flame-Graphs* aus Abbildung A.10, Abbildung A.11 und Abbildung A.12 mit künstlicher interne Last betrachtet. In Tabelle 6.1 können die Laufzeiten der Kernel-Übersetzung eingesehen werden. Dabei wird zwischen den verschiedenen Lastszenarien unterschieden. Anhand dieser Tabelle ist erkennbar, dass die von *kCQL* erzeugte Last bei steigender interner Last auch ansteigt. Somit kann nun mit Sicherheit gesagt werden, dass die interne Last und die erzeugte Last miteinander in Verbindung stehen. Für die *Netsimple*-Anfrage und den betrachteten Lastszenarien trifft diese Aussage definitiv zu.

Auch bei den hier vorliegenden Messungen existiert der in Unterabschnitt 6.2.3 beschriebene Unterschied der absoluten Last zwischen den Lastszenarien. Bei einem Vergleich der

	10 Mbit/s	100 Mbit/s	800 Mbit/s	833 Pakete/s	8333 Pakete/s	66667 Pakete/s
Laufzeit in Sekunden	145,51	147,95	210,12	145,94	157,20	199,54

Tabelle 6.1.: Laufzeiten der Linux-Kernel-Übersetzung in Sekunden.

Messungen mit 10 Mbit/s bzw. 833 Pakete/s mit externer Last und den entsprechenden Messungen ohne externe Last sind kaum nennenswerte Unterschiede in den relativen Lasten erkennbar. Die absolute Häufigkeit der Funktionsaufrufe ist in einigen Fällen ohne externe Last größer. Dies liegt vermutlich daran, dass *kCQL* ohne externe Last den Prozessor häufiger zur Verfügung steht. Dies liegt daran, dass ohne externe Last weniger Prozesse um den Prozessor konkurrieren, was dazu führt, dass der Prozessor je nach *Schedulingverfahren* seltener abgegeben werden muss.
Die relative Häufigkeiten bewegen sich allerdings in einem ähnlichen Bereich. Zusätzlich fällt auf, dass die Funktion *waitScheduler* bei vorhandener externer Last etwas weniger Last verursacht. Dies geschieht vermutlich, da die externe Last dazu führt, dass *kCQL* seltener den Prozessor verwenden kann. Somit wird *kCQL* oft dazu gezwungen den Prozessor abzugeben und muss diesen seltener von selbst abgeben. Folglich gibt *waitScheduler* den Prozessor weniger ab und erzeugt seltener Last. Das verringert die von *kCQL* erzeugte Last, erhöht allerdings die Verarbeitungsdauer einzelner eingehender Informationen. Dieser Effekt tritt verstärkt bei kleinerer interner Last auf.
Auch die Messungen mit 100 Mbit/s und 8333 Pakete/s interner Last unterscheiden sich kaum von den Messungen ohne externe Last. Lediglich die Differenz der gemessenen relativen Last der Funktion *waitScheduler* ist geringer.
Die 800 Mbit/s Messung unterscheidet sich von der entsprechenden Messung ohne externe Last und ähnelt eher der 66667 Pakete/s Messung ohne externe Last. Die relativen Werte der 66667 Pakete/s Messung wiederum ähneln ihrer äquivalenten Messung ohne externe Last.

6.2.4. Ergebnisse

Die durchgeführten Messungen haben einige Funktionen aufgezeigt, die für eine Verringerung der von der *Netsimple*-Anfrage erzeugten Systemlast optimiert werden können. Auffällig sind die Funktionen *getGlobalTS*, *service_schedule* und *waitScheduler*. Entweder kann die Implementierung der entsprechenden Funktion verbessert werden oder die Funktion sollte seltener verwendet werden. Bei der Funktion *getGlobalTS* sollte beispielsweise die interne Verwendung von *getnstimeofday* überdacht werden, da diese Funktion bei häufiger Nutzung viel Last erzeugt.
Eine ausgewogene Mischung der Vermeidung und der Optimierung ist vermutlich das beste Vorgehen. Da diese Funktionen in den *run*-Funktionen verschiedenster Operatoren auftauchen, kann davon ausgegangen werden, dass eine Optimierung dieser eine Verbes-

serung der erzeugten Systemlast aller *kCQL*-Anfragen bewirkt.
Zusätzlich lassen die Messergebnisse die Vermutung zu, dass externe Last wiedererwartens in einigen seltenen Fällen die direkt von *kCQL* erzeugte Systemlast verringern kann, da die Funktion *waitScheduler* in einigen Fällen etwas weniger Last erzeugt. Die zuvor gesparte Systemlast taucht allerdings vermutlich beim Betriebssystem-*Scheduler* wieder auf.

Es bleibt die Frage offen, wieso bei den Messungen mit *Dummy*-Daten fast in allen Fällen eine größere absolute Last gemessen wurde, als bei den Messungen mit echten Daten.

Kritik

Hot-Graphs zur Visualisierung der Messergebnisse zu verwenden ist (im Nachhinein betrachtet) keine gute Variante für schriftliche wissenschaftliche Arbeiten. Im allgemeinen kann mit den *Hot-Graphs* eine schnelle Analyse durchgeführt werden und auch für Präsentationen sind sie gut geeignet. In gedruckten Arbeiten gehen allerdings einige Informationen der *Hot-Graphs* verloren. Auch die formalen Rahmenbedingungen wie z. B. Randgrößen schränken eine lesbare Skalierung der *Hot-Graphs* ein.
Im Nachhinein wäre für diese Arbeit eine absteigend sortierte Betrachtung aller Funktionsaufrufe (unabhängig vom Stacktrace) sinnvoller gewesen. Auf diese Weise sind auch lastintensive Funktionen wie z. B. *getGlobalTS*, die in mehreren *Stacktraces* auftauchen, schneller zu erkennen.

6.3. Analyse der Verarbeitungsgeschwindigkeit

In diesem Abschnitt wird die Verarbeitungsgeschwindigkeit der *Netsimple*-Anfrage eingehend untersucht.
Dazu wird in Unterabschnitt 6.3.1 erläutert, weshalb die gemessenen Daten für eine Analyse korrekt sind und verwendet werden können. Daraufhin wird in Unterabschnitt 6.3.2 auf die verwendeten Metriken eingegangen. In Unterabschnitt 6.3.3 werden anschließend die gemessenen mittleren Verarbeitungszeiten und Prozessorabgaben betrachtet. Zum Schluss werden in Unterabschnitt 6.3.4 Schlüsse aus den Messungen gezogen.

6.3.1. Korrektheit

Das einfügen der *Tracepoints* in die *kCQL*-Implementierung und das interne Mitführen einer *ID* beeinflusst die Verarbeitungszeit von *kCQL*. Die gemessenen Zeiten entsprechen somit nicht den realen Verarbeitungszeiten. Allerdings betreffen alle Änderungen jeden Operator gleichermaßen und sind für alle Operatoren ähnlich, weshalb die gemessenen

Zeiten dennoch dazu geeignet sind, die Verarbeitungszeiten der Operatoren miteinander zu vergleichen.

6.3.2. Metriken

Um die verschiedenen Lastszenarien einander gegenüberzustellen, muss anhand der 500000 gemessenen Laufzeiten ein Vergleichswert ermittelt werden. Dabei muss beachtet werden, dass nicht zu jedem Operator und Transport 500000 Laufzeiten gemessen wurden. Alle Transporte und Operatoren, die mit Relationen zusammenhängen besitzen lediglich 2 Laufzeiten.
Um einen Vergleichswert zu erhalten, wird ein Mittelwert benötigt. Dafür wird sowohl das arithmetische Mittel $\bar{x}$ als auch die geschätzte mittlere Varianz $\hat{\sigma}_{ML}$ verwendet. Die geschätzte mittlere Varianz wird benötigt, um eine Information über die Streuung der Messdaten zu erhalten und um zu wissen, ob Messungen mit ähnlichen Mittelwert, wirklich ähnlich sind.
Da keine genauere Informationen über die Verteilung bekannt sind und jede Verteilung mit vielen Messdaten gegen eine Normalverteilung konvergiert [19], wird für die Bestimmung der Varianz davon ausgegangen, dass die Messdaten normalverteilt sind. Anhand dieser Annahme kann eine mittlere Varianz wie folgt bestimmt werden [20]:

$$\hat{\sigma}_{ML} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

x_i : i-ter Messwert, N : Anzahl der Messungen

Die Varianz wird allerdings lediglich bestimmt, wenn genügend Messdaten vorliegen. Zusätzlich werden einige Transportzeiten durch Boxplots dargestellt, um die Laufzeiten der verschiedenen Lastszenarien leichter zu vergleichen.

6.3.3. Messdaten

Die Rohdaten der Messergebnisse der Verarbeitungsgeschwindigkeitsanalyse sind in Output A.1, Output A.2, Output A.3, Output A.4 und Output A.5 einsehbar. Bei der Untersuchung der Verarbeitungsgeschwindigkeit werden drei getrennte Lastszenarien gemessen. Einmal werden die Verarbeitungszeiten der Messungen mit 1000, 10000 und 100000 Pakete pro Sekunde verglichen. Dann wird die Häufigkeit der Prozessorabgabe untersucht und es werden die Verarbeitungszeiten im *Burstmodus* betrachtet, welche mit dem 1000 Pakete/s Lastszenario verglichen werden.

Untersuchung der Lastszenarien

Bei allen Messungen fällt auf, dass die mittlere Verarbeitungszeit der Operatoren recht gering ist (vgl. Tabelle 6.2, Tabelle 6.4 und Tabelle 6.6) und diese somit kaum eine Auswirkung auf die Durchlaufzeit haben. Viel stärker fallen dagegen die Verarbeitungszeiten der Transporte zwischen den Operatoren ins Gewicht. Deswegen wird der Fokus dieser Betrachtung auf die Transporte gelegt.
Bei der Untersuchung der Messungen mit 1000 Pakete/s (vgl. Tabelle 6.2 und Tabelle 6.3) und 10000 Pakete/s (vgl. Tabelle 6.4 und Tabelle 6.5) interner Last fällt auf, dass aus einem nicht bekannten Grund die Verarbeitungszeiten der Operatoren und Transporte trotz größerer Last kleiner werden.
In Tabelle 6.10 ist die mittlere Durchlaufzeit aller Lastszenarien dargestellt. Daraus ist erkennbar, dass die mittlere Gesamtdurchlaufzeit eines Pakets für eine interne Last, die *kCQL* nicht voll auslastet, im Bereich um 10^6 Nanosekunden bewegt. Im speziellem fällt auf, dass die mittlere Durchlaufzeit vom 10000 Pakete/s Lastszenario etwas geringer ist, als die des 1000 Pakete/s Lastszenario. Allerdings ist die Streuung der Messdaten beim 10000 Pakete/s Lastszenario größer. Wird *kCQL* allerdings voll ausgelastet, dann wird die mittlere Gesamtdurchlaufzeit schlechter und bewegt sich um die 10^9 Nanosekunden. Auch die Streuung wird bei den Lastszenarien, die *kCQL* voll auslasten, viel größer.

In Tabelle 6.3, Tabelle 6.5 und Tabelle 6.7 sind die Verarbeitungszeiten aller Transporte der verschiedenen Lastszenerien abgebildet. Dabei sind auffällige Verarbeitungszeiten fett gedruckt. Es ist erkennbar, dass in den beiden kleineren Lastszenarien die asynchronen Transporte, das bedeutet die Transporte zwischen Useranwendung und *Kernelmodul* oder zwischen asynchronen Quellen und Operatoren, auffällige Verarbeitungszeiten aufweisen. Diese Verarbeitungszeit ist ungefähr um den Faktor 10^4 größer, als die der synchronen Transporte zwischen den Operatoren einer Instanz. Lediglich bei großen Lastszenarien, die *kCQL* voll auslasten wie z. B. das 100000 Pakete/s Szenario, wird die Verarbeitungszeit der synchronen Transporte zwischen allen Operatoren im *Kernelmodul* größer, als die Verarbeitungszeit der asynchronen Transporte.
Die *asynchronen Transporte* zwischen asynchronen Quellen und *Relation-Source*-Operatoren werden in den vorliegenden Lastszenarien sehr selten ausgeführt und haben praktisch keine Auswirkung auf die Durchlaufzeit eines Netzwerkpakets. Das liegt daran, dass die Registrierung der Sockets und Prozesse vor dem Erhalt eines Datenpakets geschieht. Deshalb werden diese Verarbeitungszeiten nicht weitergehend betrachtet.

Die Transporte zwischen Kernel- und Useranwendung werden für einen genaueren Vergleich der Lastszenarien in Abbildung 6.1 als *Boxplots* dargestellt. Dabei werden die Verarbeitungszeiten aller Lastszenarien als Boxplot dargestellt. Anhand der Boxplots ist erkennbar, dass die Transportzeit bei hoher Last, die *kCQL* voll auslastet (100000 Pakete/s), sehr stark zu schwanken beginnt und deutlich schlechter wird.

Außerdem fällt auf, dass die betrachtete Transportzeit bei 10000 Pakete/s minimal geringer ist als bei 1000 Pakete/s.
Es sind einige Ausreißer zu erkennen. Der 1000 Pakete/s Boxplot hat ungefähr 44 und der 10000 Pakete/s Boxplot hat ungefähr 620 Ausreißer. Das sind so wenige, dass sie für die Betrachtung vernachlässigt werden können. Diese Ausreißer sind lediglich Randerscheinungen, die vermutlich durch ein unglückliches *Scheduling* der Useranwendung entstanden sind.

Untersuchung der Messungen im Burstmodus

In Tabelle 6.9 sind die mittleren Laufzeiten der Operatoren und Transporte abgebildet, die durch eine Messung im *Burstmodus* entstanden sind. Im *Burstmodus* werden 1000 Pakete so schnell wie möglich in die Eingabe von *kCQL* geschrieben und anschließend eine Sekunde gewartet. Dies wiederholt sich solange bis 500000 Pakete verarbeitet worden sind. Die Verarbeitungszeiten sind am ehesten mit denen der Messung mit 1000 Pakete/s vergleichbar, da die Frequenz vom *Burstmodus* rechnerisch in einem ähnlichen Bereich liegt.
Im Vergleich ist auffällig, dass einige synchronen Transporte zwischen den Operatoren eine um den Faktor 10 größere Transportzeit haben. Ein ähnliches Verhalten konnte bereits bei anderen Lastszenarien, die *kCQL* voll auslasten beobachtet werden. Die Transportzeiten führen letztendlich zu einer um den Faktor 10 größere mittlere Durchlaufzeit der Pakete des *Burstmodus* (vgl. Tabelle 6.10). Dies zeigt, dass die Verteilung der internen Last wichtig ist und eine Auswirkung auf die Verarbeitungsgeschwindigkeit hat.

Untersuchung der Prozessorabgaben

Während aller Messungen wird gezählt, wie häufig das *Kernelmodul* und die Useranwendung den Prozessor abgeben. Diese Prozessorabgaben sind in Tabelle 6.11 aufgelistet. Dabei kann die Prozessorabgabehäufigkeit der verschiedenen Lastszenarien verglichen werden.
Aus der Prozessorabgabehäufigkeit im *Burstmodus* ist erkennbar, dass die zuvor betrachtete größere mittlere Durchlaufgeschwindigkeit dieser Messung nicht durch eine zu häufige Prozessorabgabe verursacht wird. Der Prozessor wird nämlich im *Kernelmodul* im *Burstmodus* seltener abgegeben als bei der 1000 Pakete/s Messung.
Weiterhin fällt auf, dass bei einer steigenden Last die Prozessorabgabehäufigkeit abnimmt. Besonders bei den Lastszenarien, die *kCQL* voll auslasten geschieht es nicht, dass größere Lastszenarien zu einer häufigeren Prozessorabgabe führen als nötig. Die Abgabehäufigkeit der Prozessoren nimmt nämlich bei 150000 Pakete/s im Vergleich zu der Messung mit 100000 Pakete/s ab. Im zweiten Lastszenario ist die Wahrschein-

lichkeit voller Puffer höher, die zu einer Prozessorabgabe führen könnten. Dennoch nimmt die Abgabehäufigkeit nicht wie vermutet zu. Dies liegt daran das der Transport zwischen *Kernelmodul* und Useranwendung schnell genug ist, damit kein Stau des Datenflusses erzeugt wird, welcher eine Prozessorabgabe des *Kernelmoduls* zur Folge hätte. Außerdem sind die Abgabehäufigkeiten mit 1524 und 57 sehr gering. Sie werden die Durchlaufgeschwindigkeit der Pakete nicht ausschlaggebend beeinträchtigen, da der Prozessor immer so schnell wie möglich wieder beansprucht wird, wenn *kCQL* weiter arbeiten kann.
Die Abgabehäufigkeiten des *Kernelmoduls* liegen bei den Messungen mit 1000 Pakete/s und 10000 Pakete/s über 500000. Das lässt die Vermutung zu, dass der Prozessor etwas zu häufig abgegeben wird. Denn wenn die Verarbeitungsgeschwindigkeit von *kCQL* so schnell ist, dass ein komplettes Datenpaket verarbeitet werden kann, bevor das nächste im System eintrifft, dann wäre zu erwarten, dass der Prozessor nach jedem abgearbeiteten Paket abgegeben wird. Ist die Geschwindigkeit geringer, dann sollte der Prozessor seltener abgegeben werden. Eine schnellere Geschwindigkeit sollte keine Auswirkung auf die Abgabehäufigkeit des Prozessors haben. Somit sollte der Prozessor nicht häufiger als 500000-mal abgegeben werden.
Vermutlich kommt die zu hohe Abgabehäufigkeit daher, dass zusätzlich zu den 500000 Netzwerkpaketen, die zu einer Ausgabe führen noch weitere Netzwerkpakete in das System eintreffen, Sockets registriert/deregistriert werden oder Prozesse erzeugt/zerstört werden. Die Änderung der Socket- und Prozessrelation ist nicht zu umgehen, allerdings ist es nicht nötig, dass *kCQL* bei jedem eingehenden Netzwerkpaket aufgeweckt wird, wenn dieses anschließend wieder verworfen wird. Eine Optimierung an dieser Stelle führt nicht zu einer verbesserten Durchlaufgeschwindigkeit. Sie verbessert ausschließlich die von *kCQL* erzeugte Last.

6.3.4. Ergebnisse

Es gibt einige Optimierungsansätze, die aus den Messungen geschlossen werden können. Allerdings sind diese Ansätze in erster Linie für die *Netsimple*-Anfrage und nur bedingt für andere Anfragen gültig.
Zum einen muss der Transportzeit zwischen asynchronen den Quellpuffern und den Operatoren im Anfrageplan verringert werden, damit die Durchlaufzeit für Lastszenarien, die *kCQL* nicht voll auslasten verbessert wird. Zum anderen müssen die Transportzeiten zwischen den Operatoren optimiert werden, damit sie bei großer Belastung von *kCQL* die Durchlaufzeit nicht so negativ beeinflussen, wie es aktuell der Fall ist. Dies kann durch verschiedene Ansätze erreicht werden. Die schlechte Transportzeit entsteht nicht durch die Schreib- oder Lesedauer aus den Transportpuffern, sondern durch die Verzögerung zwischen dem Schreiben in und dem Lesen aus einen Puffer. Die Verzögerung der synchronen Transporte entsteht durch das *Schedulingverfahren* des *kCQL-Schedulers*.

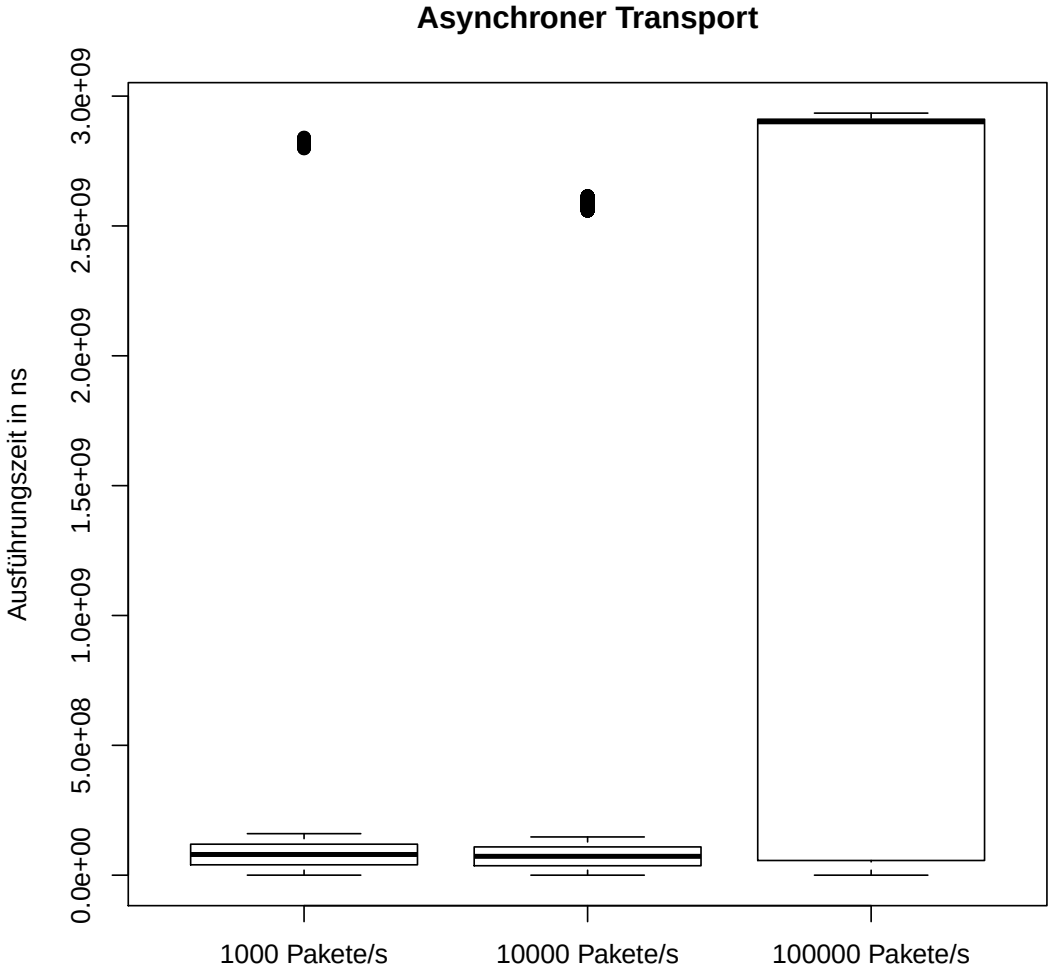


Abbildung 6.1.: Vergleich der Verarbeitungszeiten des Transports zwischen Kernelanwendung und Useranwendung. Dabei werden die Verarbeitungszeiten der 3 Lastszenarien als Boxplots dargestellt.

Operator	Mittlere Verarbeitungszeit (ns)	Varianz
Asynchrone Quelle von 0.	$6384,098$	$93,213 * 10^4$
Asynchrone Quelle von 1.	$2373,999$	-
Asynchrone Quelle von 2.	$2409,499$	-
0. str_source	$2026,080$	$74,810 * 10^3$
1. rel_source	$978,005$	-
2. rel_source	$733,504$	-
3. bin_str_join	$2010,336$	$76,327 * 10^3$
5. bin_str_join	$980,806$	$11,467 * 10^4$
7. sender	$864,023$	$78,363 * 10^3$
4. str_receiver	$1457,020$	$42,267 * 10^5$
9. output	$4152,495$	$16,213 * 10^6$

Tabelle 6.2.: Mittlere Verarbeitungszeit der Operatoren aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei wurden je Operator 500000 Netzwerkpakete verarbeitet und das arithmetische Mittel der Verarbeitungszeiten gebildet. Die verarbeiteten Daten werden mit einer Frequenz von 1000 Pakete/s in die Quellpuffer von *kCQL* geschrieben. Zusätzlich ist die Varianz als Maß für die Streuung angegeben. Die Varianz der Relationsoperatoren wird aufgrund der wenigen Messdaten nicht berechnet.

Dieser sollte für lastintensive Szenarien optimiert werden. Die Verzögerung der asynchronen Transporte hat mehrere Ursprünge. Zum einen braucht ein Signal, dass den *Scheduler* zum Lesen aus einem asynchronen Quellpuffer anweist etwas Zeit. Zum anderen priorisiert der *kCQL-Scheduler* einen Operator, der aus einer asynchronen Quelle liest nicht höher als andere. Dies kann sich negativ auf die Reaktionszeit auswirken. Die Reaktionszeit zwischen asynchronen Lesen und Schreiben ist eventuell durch ein Signal beim Schreiben in den Puffer mit anschließender höherer Priorisierung der Operatoren, die aus dem Puffer lesen, möglich.
Des Weiteren sollte versucht werden die Prozessorabgaben bei geringen Lasten wie 1000 Pakete/s und 10000 Pakete/s zu reduzieren. Dies kann z. B. dadurch erreicht werden, dass ein lastschonendes Verfahren entwickelt wird, dass beim Erhalt eines Netzwerkpaket entscheidet, ob dieses Netzwerkpaket für die Anfrage relevant ist und der *Scheduler* aufgeweckt werden muss. Dabei sollte untersucht werden ob eine Reduzierung der Prozessorabgaben eine positive Auswirkung auf die mittlere Durchlaufzeit hat, ohne dabei die Last für das System deutlich zu verschlechtern. Vermutlich hat diese Optimierung einen stärkeren Einfluss auf die erzeugte Systemlast, als auf die Verarbeitungsgeschwindigkeit.
Weiterhin sollte weitere Ursachenforschung im Hinblick auf die schlechteren synchronen

Transport	Mittlere Verarbeitungszeit (ns)	Varianz
Von Quelle nach 0.	$8600,473$	$57,699 * 10^6$
Von Quelle nach 1.	$\mathbf{2171,938 * 10^4}$	-
Von Quelle nach 2.	$\mathbf{5579,215 * 10^4}$	-
Von 2. nach 3.	$3233,650 * 10^1$	-
Von 0. nach 3.	$5051,943$	$50,445 * 10^4$
Von 3. nach 5.	$3403,057$	$27,728 * 10^4$
Von 1. nach 5.	$1368,900 * 10^1$	-
Von 5. nach 7.	$1627,428$	$97,896 * 10^3$
Von 7. nach 4.	$\mathbf{7973,254 * 10^4}$	$27,710 * 10^{14}$
Von 4. nach 9.	$1882,563 * 10^1$	$79,955 * 10^6$

Tabelle 6.3.: Die beiden Tabellen beschreiben die mittlere Verarbeitungszeit aller Transporte aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei werden die verarbeiteten Informationen mit einer Frequenz von 1000 Pakete/s den asynchronen Quellen hinzugefügt. Die Varianz der Relationstransporte wird aufgrund der wenigen Messdaten nicht berechnet.

Transportzeiten lastintensiver Szenarien betrieben werden.

Operator	Mittlere Verarbeitungszeit (ns)	Varianz
Asynchrone Quelle von 0.	$3545,006$	$12,947 * 10^5$
Asynchrone Quelle von 1.	$2479,501$	-
Asynchrone Quelle von 2.	$2444,001$	-
0. str_source	$1286,883$	$37,067 * 10^4$
1. rel_source	$907,501$	-
2. rel_source	$733,504$	-
3. bin_str_join	$997,456$	$29,957 * 10^4$
5. bin_str_join	$909,133$	$26,940 * 10^4$
7. sender	$821,220$	$34,656 * 10^4$
4. str_receiver	$1720,644$	$28,721 * 10^5$
9. output	$4108,212$	$31,996 * 10^6$

Tabelle 6.4.: Mittlere Verarbeitungszeit der Operatoren aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei wurden je Operator 500000 Netzwerkpakete verarbeitet und das arithmetische Mittel der Verarbeitungszeiten gebildet. Die verarbeiteten Daten werden mit einer Frequenz von 10000 Pakete/s in die Quellpuffer von *kCQL* geschrieben. Zusätzlich ist die Varianz als Maß für die Streuung angegeben. Die Varianz der Relationsoperatoren wird aufgrund der wenigen Messdaten nicht berechnet.

Transport	Mittlere Verarbeitungszeit (ns)	Varianz
Von Quelle nach 0.	$5248,103$	$41,777 * 10^7$
Von Quelle nach 1.	$\mathbf{2175,203 * 10^4}$	-
Von Quelle nach 2.	$\mathbf{5611,241 * 10^4}$	-
Von 2. nach 3.	$3816,850 * 10^1$	-
Von 0. nach 3.	$2430,760$	$30,360 * 10^5$
Von 3. nach 5.	$2067,147$	$29,500 * 10^5$
Von 1. nach 5.	$1403,850 * 10^1$	-
Von 5. nach 7.	$1118,074$	$16,764 * 10^5$
Von 7. nach 4.	$\mathbf{7562,512 * 10^4}$	$95,793 * 10^{14}$
Von 4. nach 9.	$2062,734 * 10^1$	$97,212 * 10^6$

Tabelle 6.5.: Die beiden Tabellen beschreiben die mittlere Verarbeitungszeit aller Transporte aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei werden die verarbeiteten Informationen mit einer Frequenz von 10000 Pakete/s den asynchronen Quellen hinzugefügt. Die Varianz der Relationstransporte wird aufgrund der wenigen Messdaten nicht berechnet.

Operator	Mittlere Verarbeitungszeit (ns)	Varianz
Asynchrone Quelle von 0.	2041.102	$39,358 * 10^4$
Asynchrone Quelle von 1.	2374.501	-
Asynchrone Quelle von 2.	2409.499	-
0. str_source	2090.046	$35,985 * 10^6$
1. rel_source	977.998	-
2. rel_source	3107.503	-
3. bin_str_join	2105.308	$28,794 * 10^6$
5. bin_str_join	1964.920	$26,611 * 10^6$
7. sender	1856.827	$25,736 * 10^6$
4. str_receiver	1705.590	$68,733 * 10^4$
9. output	3829.013	$76,439 * 10^7$

Tabelle 6.6.: Mittlere Verarbeitungszeit der Operatoren aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei wurden je Operator 500000 Netzwerkpakete verarbeitet und das arithmetische Mittel der Verarbeitungszeiten gebildet. Die verarbeiteten Daten werden mit einer Frequenz von 100000 Pakete/s in die Quellpuffer von *kCQL* geschrieben. Zusätzlich ist die Varianz als Maß für die Streuung angegeben. Die Varianz der Relationsoperatoren wird aufgrund der wenigen Messdaten nicht berechnet.

Transport	Mittlere Verarbeitungszeit (ns)	Varianz
Von Quelle nach 0.	$\mathbf{4322,257 * 10^7}$	$41,374 * 10^{19}$
Von Quelle nach 1.	$2299,646 * 10^4$	-
Von Quelle nach 2.	$5626,208 * 10^4$	-
Von 2. nach 3.	$8506,700 * 10^1$	-
Von 0. nach 3.	$\mathbf{4611,625 * 10^5}$	$10,525 * 10^{17}$
Von 3. nach 5.	$\mathbf{4613,116 * 10^5}$	$10,524 * 10^{17}$
Von 1. nach 5.	$5950,450 * 10^1$	-
Von 5. nach 7.	$\mathbf{4598,612 * 10^5}$	$10,513 * 10^{17}$
Von 7. nach 4.	$\mathbf{1918,427 * 10^6}$	$17,913 * 10^{17}$
Von 4. nach 9.	$8587,734 * 10^2$	$81,705 * 10^{13}$

Tabelle 6.7.: Die beiden Tabellen beschreiben die mittlere Verarbeitungszeit aller Transporte aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei werden die verarbeiteten Informationen mit einer Frequenz von 100000 Pakete/s den asynchronen Quellen hinzugefügt. Die Varianz der Relationstransporte wird aufgrund der wenigen Messdaten nicht berechnet.

Operator	Mittlere Verarbeitungszeit (ns)	Varianz
Asynchrone Quelle von 0.	2860.314	$24,058 * 10^5$
Asynchrone Quelle von 1.	2339.504	-
Asynchrone Quelle von 2.	2269.502	-
0. str_source	1308.577	$13,189 * 10^5$
1. rel_source	1117.500	-
2. rel_source	1152.497	-
3. bin_str_join	1295.938	$13,280 * 10^5$
5. bin_str_join	1291.832	$12,322 * 10^5$
7. sender	1263.874	$12,687 * 10^5$
4. str_receiver	1727.765	$21,372 * 10^5$
9. output	3860.372	$46,944 * 10^5$

Tabelle 6.8.: Mittlere Verarbeitungszeit der Operatoren aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei wurden je Operator 500000 Netzwerkpakete verarbeitet und das arithmetische Mittel der Verarbeitungszeiten gebildet. Die verarbeiteten Daten im *Burstmodus* in die Quellpuffer von *kCQL* geschrieben. Zusätzlich ist die Varianz als Maß für die Streuung angegeben. Die Varianz der Relationsoperatoren wird aufgrund der wenigen Messdaten nicht berechnet.

Transport	Mittlere Verarbeitungszeit (ns)	Varianz
Von Quelle nach 0.	$3405,188 * 10^3$	$55,607 * 10^{11}$
Von Quelle nach 1.	$\mathbf{2241,703 * 10^4}$	-
Von Quelle nach 2.	$\mathbf{5622,642 * 10^4}$	-
Von 2. nach 3.	$6076,199 * 10^1$	-
Von 0. nach 3.	$7726,367 * 10^1$	$90,012 * 10^7$
Von 3. nach 5.	$8595,354 * 10^1$	$36,960 * 10^8$
Von 1. nach 5.	$4162,550 * 10^1$	-
Von 5. nach 7.	$7661,527 * 10^1$	$90,297 * 10^7$
Von 7. nach 4.	$\mathbf{5135,388 * 10^5}$	$2.563 * 10^{16}$
Von 4. nach 9.	$3712,046 * 10^1$	$11,927 * 10^7$

Tabelle 6.9.: Die beiden Tabellen beschreiben die mittlere Verarbeitungszeit aller Transporte aus dem *Netsimple*-Anfrageplan (vgl. Abbildung 2.3). Dabei werden die verarbeiteten Informationen im *Burstmodus* den asynchronen Quellen hinzugefügt. Die Varianz der Relationstransporte wird aufgrund der wenigen Messdaten nicht berechnet.

Lastszenario	Mittlere Durchlaufzeit	Varianz
1000 Pakete/s	$79,788 * 10^6$ ns	$27,708 * 10^{14}$
10000 Pakete/s	$75,670 * 10^6$ ns	$95,791 * 10^{14}$
100000 Pakete/s	$46,524 * 10^9$ ns	$41,334 * 10^{19}$
150000 Pakete/s	$44,441 * 10^9$ ns	$40,181 * 10^{19}$
Burstmodus	$51,722 * 10^7$ ns	$25,503 * 10^{16}$

Tabelle 6.10.: Diese Tabelle bildet die mittlere Durchlaufzeit der verschiedenen Lastszenarien ab. Dabei wurden 500000 Durchläufe erzeugt und daraus der angegebene Mittelwert berechnet.

Lastszenario	Prozessorabgaben Kernel	Prozessorabgaben User
1000 Pakete/s	501551	3322
10000 Pakete/s	501368	548
100000 Pakete/s	1524	46
150000 Pakete/s	57	41
Burstmodus	760	254

Tabelle 6.11.: Diese Tabelle stellt die Häufigkeit der Prozessorabgaben der Useranwendung und der Kernelanwendung für die Lastszenarien 1000 Pakete/s, 10000 Pakete/s, 100000 Pakete/s und 150000 Pakete/s dar.

7. Zusammenfassung

Diese Arbeit verfolgt das Ziel Programmteile von *kCQL* zu bestimmen, die für eine Leistungsverbesserung optimiert werden können. Dazu zählt die Verringerung der erzeugten Systemlast und die Verbesserung der Verarbeitungsgeschwindigkeit. Für die Umsetzung dieser Ziele musste zuerst *kCQL* genauer betrachtet werden. Dies ist für eine spezifische Analyse unumgänglich. Anschließend wurden verschiedene Werkzeuge verglichen, die für Leistungsanalysen in Frage kommen. Daraufhin wurden allgemeine Verfahren für die Last- und Verarbeitungsgeschwindigkeitsanalyse entwickelt, die weiterhin in dieser Arbeit verwendet wurden (vgl. Unterabschnitt 4.3.5 und Unterabschnitt 4.2.3). Diese Analyseverfahren können auch mit kleineren Anpassungen auf beliebige andere *kCQL*-Anfragen angewendet werden.
Zur Optimierung der von *kCQL* erzeugten Systemlast, wurden einige Ansätze entwickelt. Da sind zum einen die Funktionen *getGlobalTS*, *service_schedule* und *waitScheduler*, die seltener verwendet werden sollten oder dessen Implementierung überdacht werden muss. Bei *getGlobalTS* sollte besonders von der Verwendung der Funktion *getnstimeofday* abgesehen werden. Die Optimierung dieser Funktionen hätte eine Verbesserung der erzeugten Systemlast aller *kCQL*-Anfragen zur Folge (vgl. Unterabschnitt 6.2.4).
Zusätzlich sollte für eine Reduzierung der erzeugten Last ein Verfahren für eingehende Informationen entwickelt werden, dass kostengünstig prüft, ob der *Scheduler* aufgeweckt werden muss (vgl. Unterabschnitt 6.3.3).
Ferner wurden auch Optimierungsansätze für die Verbesserung der Verarbeitungsgeschwindigkeit ermittelt. Zielführend dafür ist eine Verbesserung der Transportzeiten. Die Optimierung der Verarbeitungszeit der asynchronen Transporte muss getrennt von der Optimierung der Verarbeitungszeit der synchronen Transporte betrachtet werden. Um die Durchlaufzeit für den Fall, dass *kCQL* nicht voll ausgelastet wird, zu optimieren, müssen die asynchronen Transporte verbessert werden. Eine Verbesserung der synchronen Transporte dagegen führt zu einer besseren Durchlaufzeit im Falle großer Lastszenarien, die *kCQL* voll auslasten (vgl. Unterabschnitt 6.3.4). Außerdem sollten die in Unterabschnitt 6.3.4 erwähnten Verbesserungen des *kCQL-Schedulers* in Betracht gezogen werden. Diese sind nämlich für eine Verbesserung der Transportzeiten nötig.

Ausblick

Weitere Arbeiten oder Untersuchungen können sich mit einer Analyse der in dieser Arbeit verwendeten Messverfahren beschäftigen. Dabei sollte weitergehend untersucht werden, ob es bessere Messverfahren für die Last oder die Geschwindigkeit gibt.
Außerdem sind weitere Untersuchungen anderer *kCQL*-Anfragen oder der *Netsimple*-Anfrage unter anderen Lastszenarien für eine stetigen Verbesserung der erzeugten Systemlast und der Verarbeitungsgeschwindigkeit zielführend. Dafür kann beispielsweise eine erschöpfende Analyse mit Hilfe der in dieser Arbeit entwickelten Analyseverfahren gemacht werden. Das bedeutet, dass nachdem die in dieser Arbeit genannten Programmteile verbessert wurden, die gemachten Untersuchungen erneut durchgeführt werden können. Sie liefern dann neue Programmteile, die es zu optimieren gilt.
Grundsätzlich ist *Perf* in Verbindung mit *Hot-Graphs* für die Lastanalyse eine gute Wahl. Allerdings sind sie für wissenschaftliche Arbeiten nicht so geeignet wie zuvor erwartet. Dadurch das die *Hot-Graphs* in gedruckter Form einige wichtige Informationen verlieren, sollten für eine detaillierte Analyse ergänzende Ausgabeverfahren in Betracht gezogen werden.
Im Bezug zu der bereits gemachten Lastanalyse ist es auch wünschenswert weiter zu erkunden, weshalb die in den Lastszenarien mit *Dummy*-Paketen in den meisten Fällen deutlich mehr absolute Funktionsaufrufe auftauchen, als in denen mit echten Netzwerkpaketen (vgl. Unterabschnitt 6.2.3).
Diese Arbeit fokussiert sich im Bezug auf die Verarbeitungsgeschwindigkeitsanalyse größtenteils auf die Durchlaufzeit. Auch andere Verarbeitungszeiten wurden betrachtet. Von einer weitergehenden Untersuchungen wurde allerdings abgesehen. Diese sollten dennoch weitergehend betrachtet werden. Beispielsweise kann die Zeit, die eine eingehende Information benötigt bis sie in einer *Synopsis* gespeichert oder entfernt wird, weiterführend betrachtet werden.

Literatur

[1] J. Streicher, A. Lochmann und O. Spinczyk. "kCQL: Declarative Stream-based Acquisition and Processing of Diagnostic OS Data". Englisch. In: *Proceedings of the Conference on Timely Results in Operating Systems (TRIOS)*. ACM, 2015. ISBN: 978-1-4503-3941-4/15/10. DOI: 10.1145/2834590.2834598.

[2] A. Arasu u. a. "STREAM: the stanford stream data manager (demonstration description)". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, S. 665–665.

[3] J. Melton. "Sql language summary". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), S. 141–143.

[4] A. Arasu, S. Babu und J. Widom. "The CQL continuous query language: semantic foundations and query execution". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 15.2 (2006), S. 121–142.

[5] D. Maier. *Theory of relational databases*. Computer Science Pr, 1983.

[6] M. Fragkoulis u. a. "Relational access to Unix kernel data structures". In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, S. 12.

[7] B. Gregg und J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[8] Ú. Erlingsson u. a. "Fay: extensible distributed tracing from kernels to clusters". In: *ACM Transactions on Computer Systems (TOCS)* 30.4 (2012), S. 13.

[9] S. Shende. "Profiling and tracing in linux". In: *Proceedings of the Extreme Linux Workshop*. Bd. 2. Citeseer. 1999.

[10] V. M. Weaver. "Linux perf_event features and overhead". In: *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. 2013, S. 80.

[11] M. Desnoyers und M. R. Dagenais. "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux". In: *OLS (Ottawa Linux Symposium)*. Bd. 2006. Citeseer. 2006, S. 209–224.

[12] P. J. Mucci u. a. "PAPI: A portable interface to hardware performance counters". In: *Proceedings of the department of defense HPCMP users group conference*. 1999, S. 7–10.

[13] S. Agarwala und K. Schwan. "Sysprof: Online distributed behavior diagnosis through fine-grain system monitoring". In: *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE. 2006, S. 8–8.

[14] W. E. Cohen. "Tuning programs with OProfile". In: *Wide Open Magazine* 1 (2004), S. 53–62.

[15] F. C. Eigler und R. Hat. "Problem solving with systemtap". In: *Proc. of the Ottawa Linux Symposium*. Citeseer. 2006, S. 261–268.

[16] B. Gregg. *Linux Performance Analysis and Tools*. Techn. Ber. Technical report, Joyent, 2013.

[17] W. Wu, M. Crawford und M. Bowden. "The performance analysis of Linux networking–packet receiving". In: *Computer Communications* 30.5 (2007), S. 1044–1057.

[18] T. Gleixner und D. Niehaus. "Hrtimers and beyond: Transforming the linux time subsystems". In: *Proceedings of the Linux symposium*. Bd. 1. Citeseer. 2006, S. 333–346.

[19] R. K. Wolfgang Viertl und R. K. Wolfgang Viertl. "Zentraler Grenzverteilungssatz". In: *Einführung in die Stochastik: Mit Elementen der Bayes–Statistik und der Analyse unscharfer Information* (2003), S. 105–109.

[20] K. Stange und K. Stange. "Die Schätzung der Varianz σ 2 einer Normalverteilung mit bekanntem Mittelwert μ bei „geeigneten Vorinformationen "über σ 2". In: *Bayes-Verfahren: Schätz-und Testverfahren bei Berücksichtigung von Vorinformationen* (1977), S. 78–95.

A. Anhang

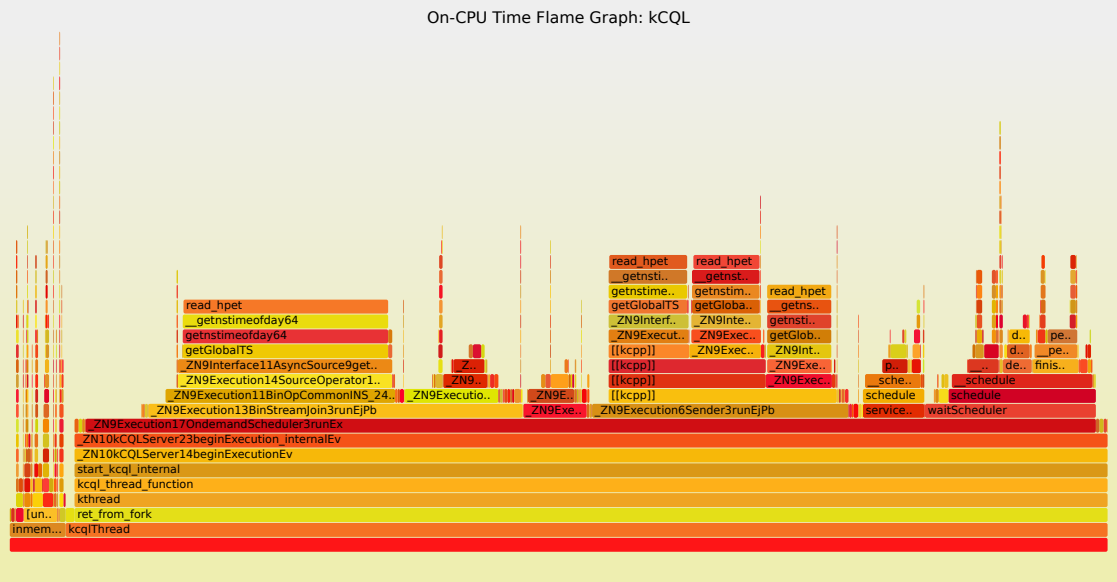


Abbildung A.1.: Ergebnis der 180-sekündigen Perfaufzeichnung ohne externe Last. Dabei wurde eine echte interne Last von 10 Mbit/s verwendet.

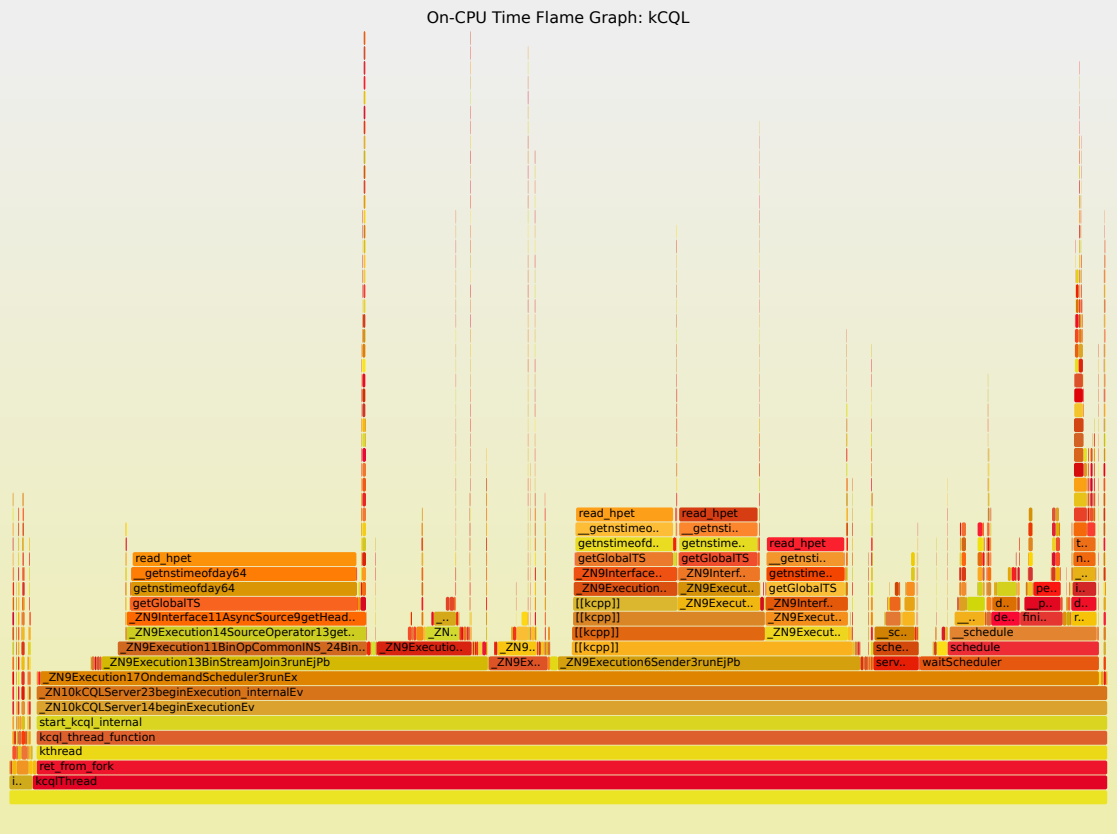


Abbildung A.2.: Ergebnis der 180-sekündigen Perfaufzeichnung ohne externe Last. Dabei wurde eine echte interne Last von 100 Mbit/s verwendet.

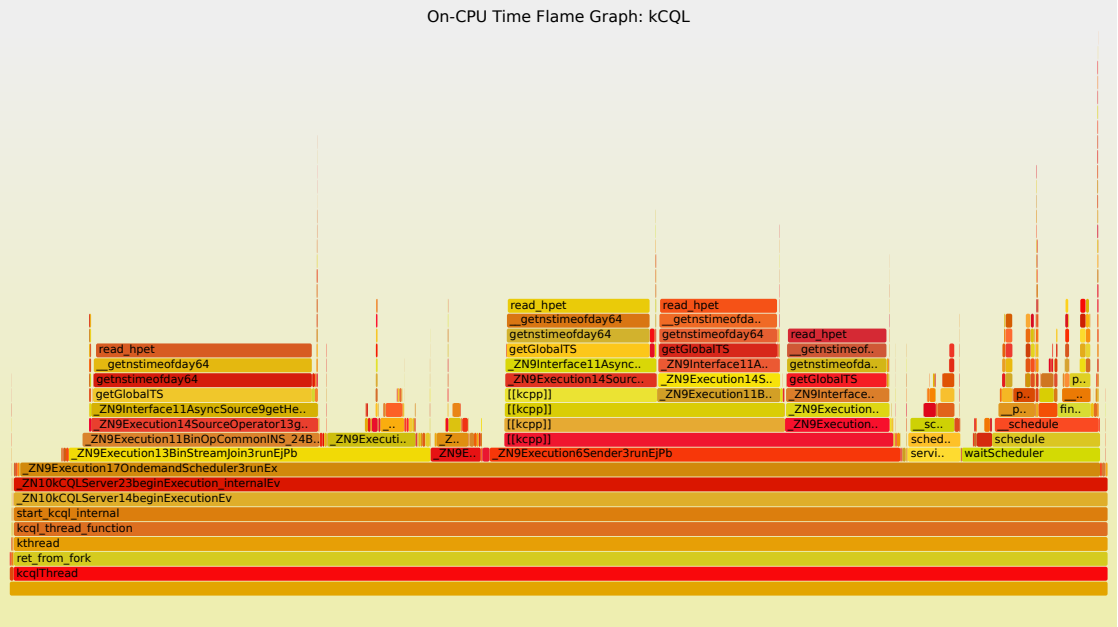


Abbildung A.3.: Ergebnis der 180-sekündigen Perfaufzeichnung ohne externe Last. Dabei wurde eine echte interne Last von 800 Mbit/s verwendet.

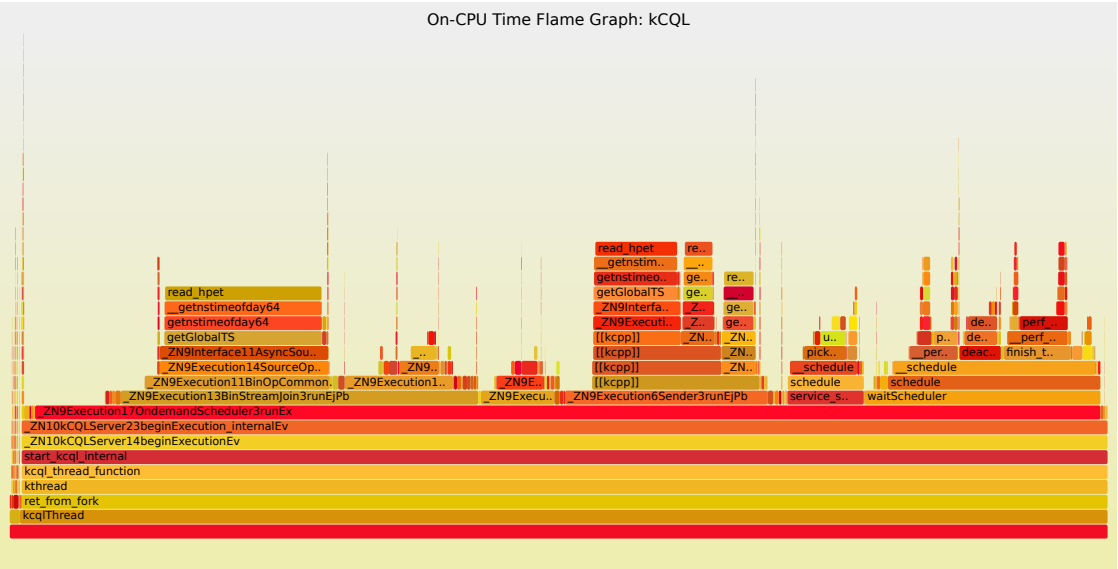


Abbildung A.4.: Ergebnis der 180-sekündigen Perfaufzeichnung ohne externe Last. Dabei wurden *Dummydaten* mit 833 Pakete/s als interne Last erzeugt.

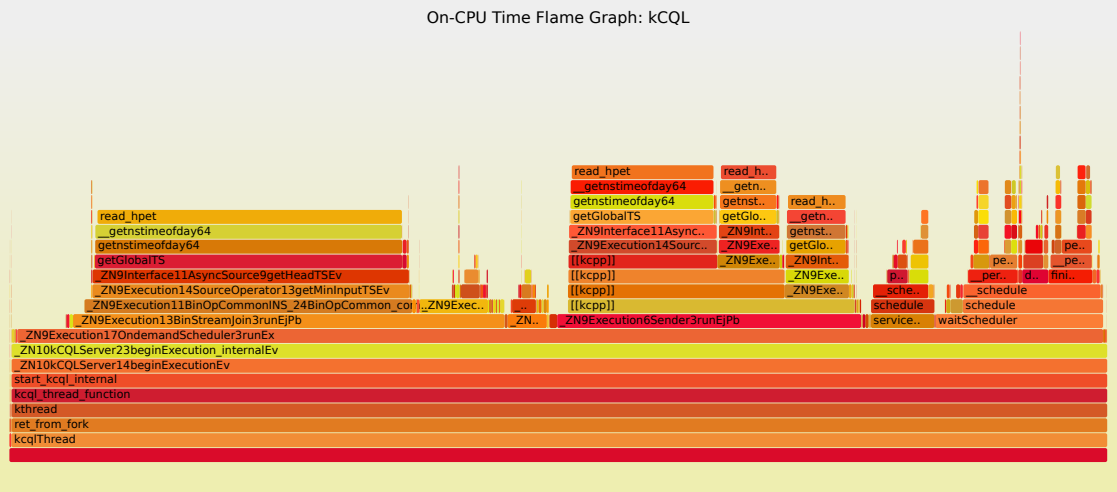


Abbildung A.5.: Ergebnis der 180-sekündigen Perfaufzeichnung ohne externe Last. Dabei wurden *Dummydaten* mit 8333 Pakete/s als interne Last erzeugt.

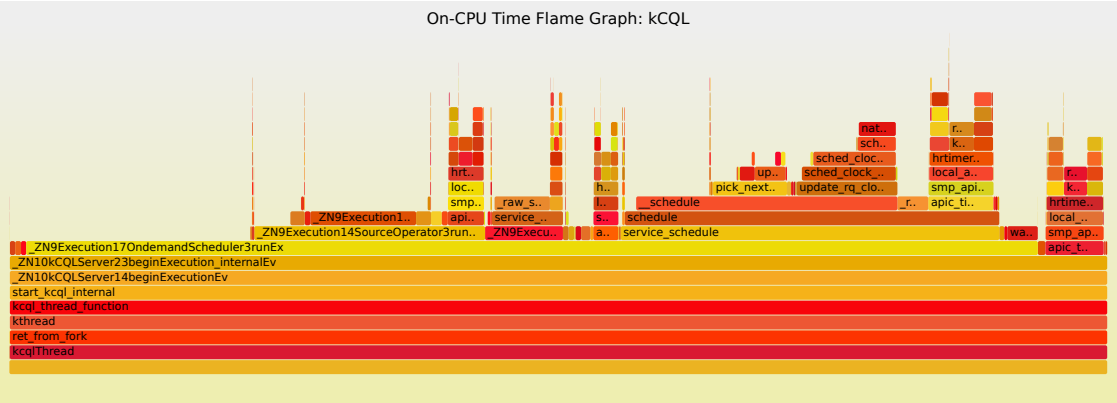


Abbildung A.6.: Ergebnis der 180-sekündigen Perfaufzeichnung ohne externe Last. Dabei wurden *Dummydaten* mit 66667 Pakete/s als interne Last erzeugt.

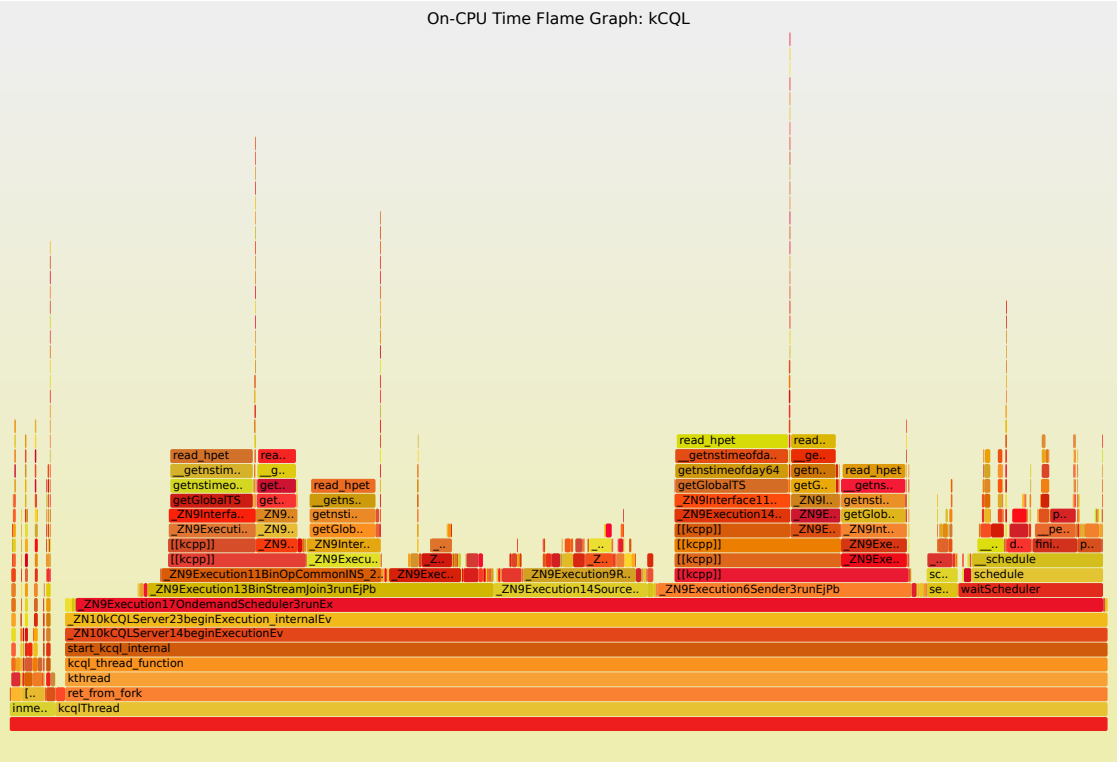


Abbildung A.7.: Ergebnis der Perfaufzeichnung mit einer Linux-Kernel-Übersetzung als externe Last. Dabei wurde eine echte interne Last von 10 Mbit/s verwendet. Die Übersetzung des Linux-Kernels dauerte 145,51 Sekunden.

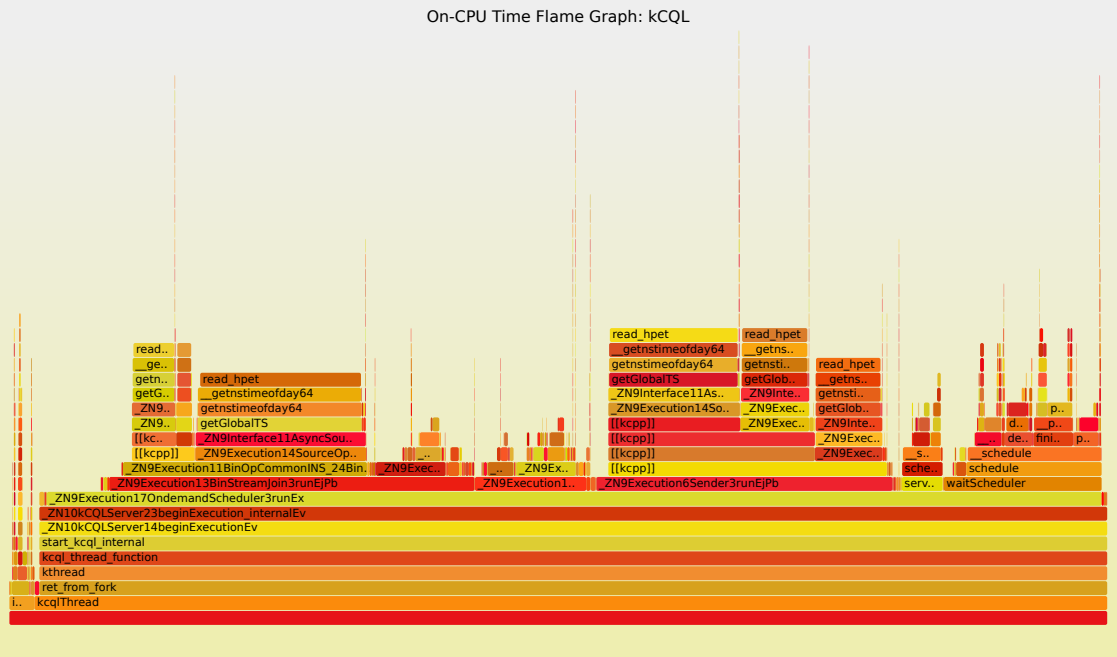


Abbildung A.8.: Ergebnis der Perfaufzeichnung mit einer Linux-Kernel-Übersetzung als externe Last. Dabei wurde eine echte interne Last von 100 Mbit/s verwendet. Die Übersetzung des Linux-Kernels dauerte 147,95 Sekunden.

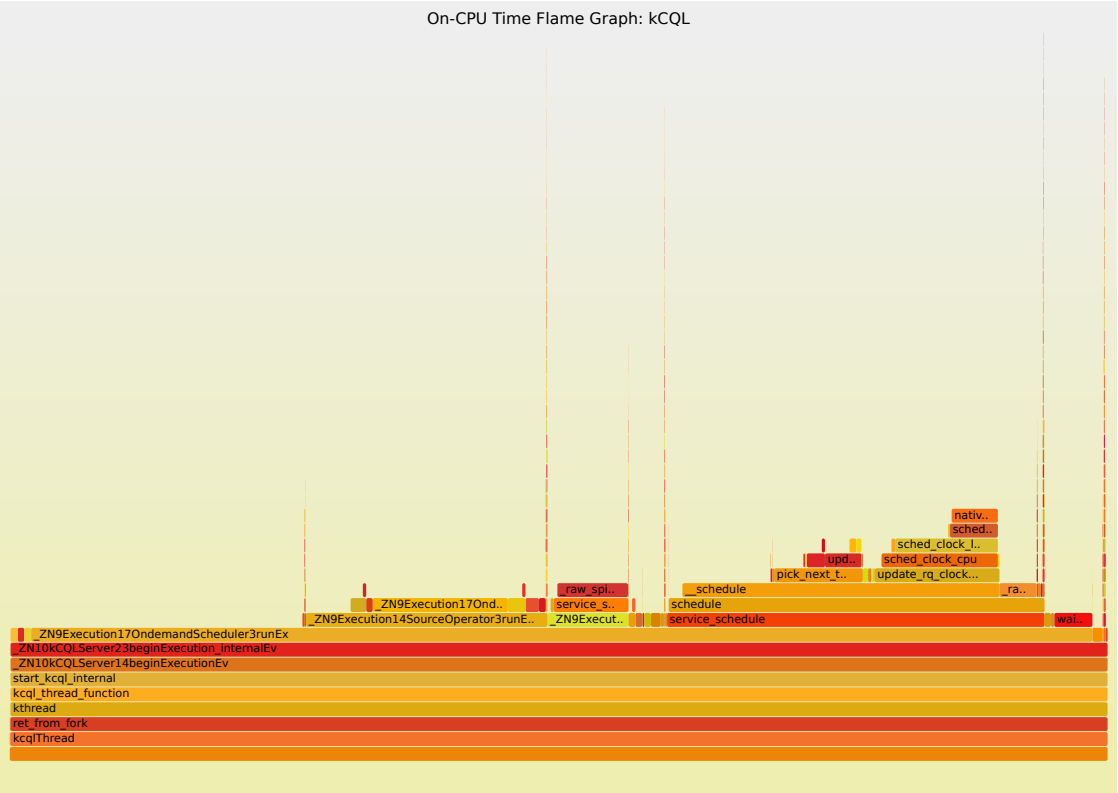


Abbildung A.9.: Ergebnis der Perfaufzeichnung mit einer Linux-Kernel-Übersetzung als externe Last. Dabei wurde eine echte interne Last von 800 Mbit/s verwendet. Die Übersetzung des Linux-Kernels dauerte 210,12 Sekunden.

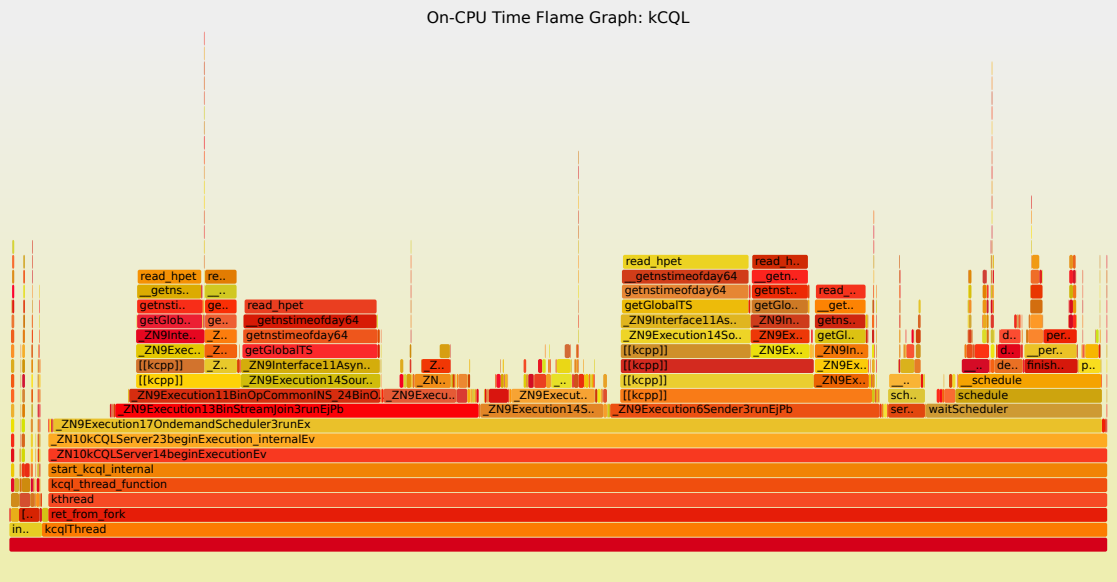


Abbildung A.10.: Ergebnis der Perfaufzeichnung mit einer Linux-Kernel-Übersetzung als externe Last. Dabei wurden *Dummydaten* mit 833 Pakete/s als interne Last erzeugt. Die Übersetzung des Linux-Kernels dauerte 145,94 Sekunden.

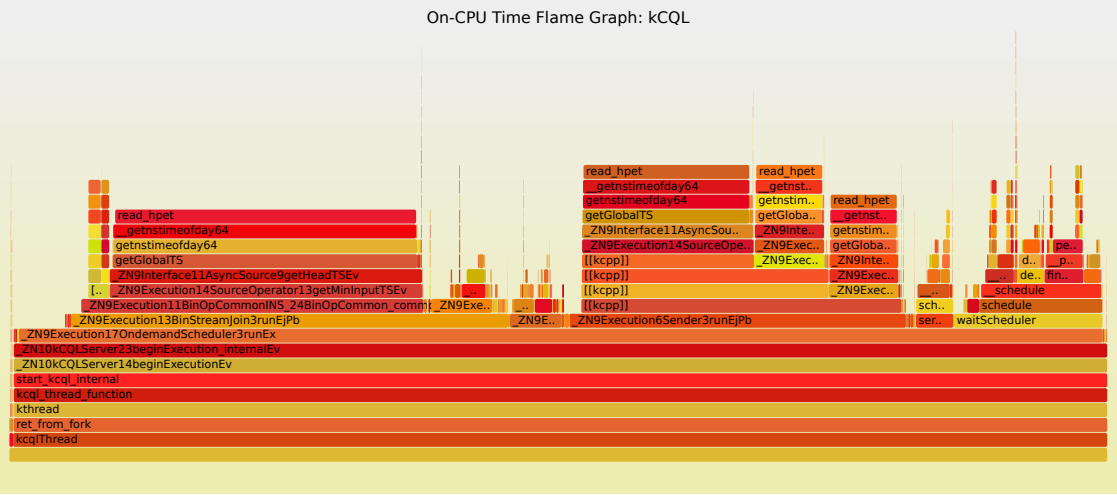


Abbildung A.11.: Ergebnis der Perfaufzeichnung mit einer Linux-Kernel-Übersetzung als externe Last. Dabei wurden *Dummydaten* mit 8333 Pakete/s als interne Last erzeugt. Die Übersetzung des Linux-Kernels dauerte 157,20 Sekunden.

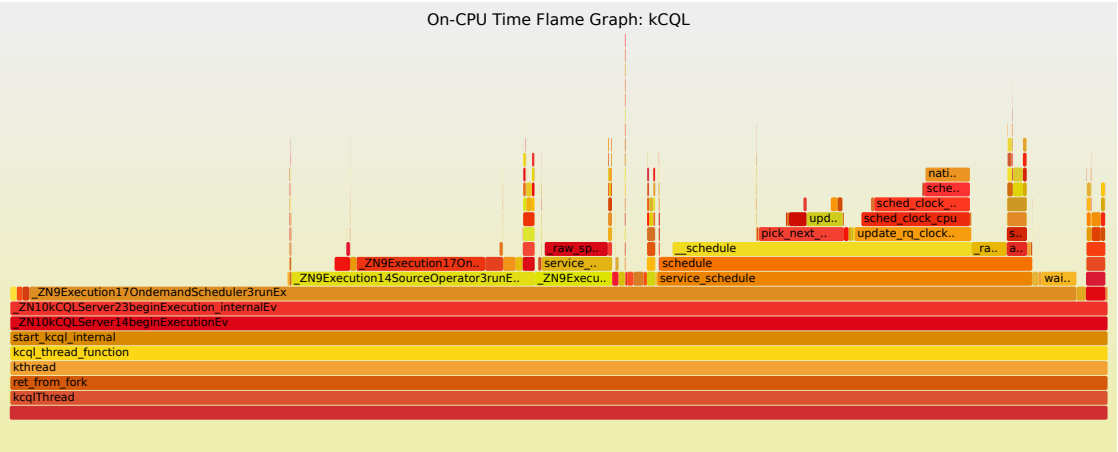


Abbildung A.12.: Ergebnis der Perfaufzeichnung mit einer Linux-Kernel-Übersetzung als externe Last. Dabei wurden *Dummydaten* mit 66667 Pakete/s als interne Last erzeugt. Die Übersetzung des Linux-Kernels dauerte 199,54 Sekunden.

```
{
Kernel-CPUAbgaben: 501551
User-CPUAbgaben: 3322
{({AsyncSource 2., ,1328266.415365751,2409.4988475545524} , {2.
    rel_source , ,1190.4287861594046,733.5038390050541})->2 :
    78119.48740911146,5.579215049475637E7},
{({AsyncSource 0., ,932125.6458662039,6384.097890431979} , {0.
    str_source , ,74809.71811162453,2026.0796435810958})->0 :
    5.769906390322969E7,8600.473035763522},
{({AsyncSource 1., ,121800.01088060245,2373.999450366404} , {1.
    rel_source , ,0.0,978.0051186680794})->1 :
    702245.9144258733,2.1719381999018796E7},
{({2. rel_source , ,1190.4287861594046,733.5038390050541} , {3.
    bin_str_join , ,76327.12574845091,2010.3360507977006})2->3 :
    121454.91102326522,32336.4947690409},
{({0. str_source , ,74809.71811162453,2026.0796435810958} , {3.
    bin_str_join , ,76327.12574845091,2010.3360507977006})0->3 :
    504450.0597547352,5051.943182591415},
{({1. rel_source , ,0.0,978.0051186680794} , {5. bin_str_join ,
    ,114670.84274981834,980.806454318143})1->5 :
    239123.5030360606,13689.001207240719},
{({3. bin_str_join , ,76327.12574845091,2010.3360507977006} , {5.
    bin_str_join , ,114670.84274981834,980.806454318143})3->5 :
    277282.4288079752,3403.057414767789},
{({5. bin_str_join , ,114670.84274981834,980.806454318143} , {7.
    sender , ,78362.83789734532,864.023078438224})5->7 :
    97895.94567329163,1627.4283701789798},
{({7. sender , ,78362.83789734532,864.023078438224} , {4.
    str_receiver , ,4226718.524612684,1457.019563329497})7->4 :
    2.770989216974696E15,7.973253908909619E7},
{({4. str_receiver , ,4226718.524612684,1457.019563329497} , {9.
    output , ,1.621257258450443E7,4152.495293790522})4->9 :
    7.995523945838393E7,18825.632751395475},
}
Mittlere Durchlaufzeit: 7.978792248182687E7
Varianz der Durchlaufzeit: 2.7707949057518415E15
```

Output A.1: Laufzeiten der Operatoren und Transporte zwischen den Operatoren für 1000 Pakete/s interne Last. Die Zeiten sind in Nanosekunden. Die erste Zahl ist immer die Varianz und die zweite Zahl das arithmetische Mittel. Die Bezeichnungen der Operatoren entsprechen denen aus dem Netsimple-Anfrageplan (vgl. Abbildung 2.3) und die Transportbezeichnungen entsprechen den Nummern vor den Operatoren. Zusätzlich ist oben die Häufigkeit wie oft das *Kernelmodul* und die Useranwendung den Prozessor abgegeben hat aufgelistet. Unten ist die mittlere Durchlaufzeit eines Datenpakets und die Varianz der Durchlaufzeit angegeben.

```
{
Kernel-CPUAbgaben: 501368
User-CPUAbgaben: 548
{({AsyncSource 2., ,1247683.944398212,2444.0014385657755} , {2.
    rel_source , ,1260.72383869088,733.5038390105074})->2 :
    99228.344665589,5.611240799790547E7},
{({AsyncSource 0., ,1294701.063850679,3545.0060682230715} , {0.
    str_source , ,370670.1703205647,1286.8833290136722})->0 :
    4.1777487219593745E8,5248.102579007726},
{({AsyncSource 1., ,59780.87575901514,2479.5008357723523} , {1.
    rel_source , ,0.24479252175636265,907.5010893863791})->1 :
    762139.0820206779,2.175203299705303E7},
{({2. rel_source , ,1260.72383869088,733.5038390105074} , {3.
    bin_str_join , ,299569.2882781606,997.4556521653176})2->3 :
    98912.30550284771,38168.502215063636},
{({0. str_source , ,370670.1703205647,1286.8833290136722} , {3.
    bin_str_join , ,299569.2882781606,997.4556521653176})0->3 :
    3036042.1712529557,2430.7595531625493},
{({1. rel_source , ,0.24479252175636265,907.5010893863791} , {5.
    bin_str_join , ,269399.1296627589,909.1332664551967})1->5 :
    175147.1096631944,14038.501831118589},
{({3. bin_str_join , ,299569.2882781606,997.4556521653176} , {5.
    bin_str_join , ,269399.1296627589,909.1332664551967})3->5 :
    2950094.1727726366,2067.1472958622617},
{({5. bin_str_join , ,269399.1296627589,909.1332664551967} , {7.
    sender , ,346562.7104976456,821.2203801412313})5->7 :
    1676360.6651993669,1118.073910740183},
{({7. sender , ,346562.7104976456,821.2203801412313} , {4.
    str_receiver , ,2872129.4867508984,1720.644094779632})7->4 :
    9.579340999755166E15,7.562511572903332E7},
{({4. str_receiver , ,2872129.4867508984,1720.644094779632} , {9.
    output , ,3.1996326275326237E7,4108.212214893162})4->9 :
    9.721166161737207E7,20627.340972919377},
}
Mittlere Durchlaufzeit: 7.566999570835972E7
Varianz der Durchlaufzeit: 9.579052438876224E15
```

Output A.2: Laufzeiten der Operatoren und Transporte zwischen den Operatoren für 10000 Pakete/s interne Last. Die Zeiten sind in Nanosekunden. Die erste Zahl ist immer die Varianz und die zweite Zahl das arithmetische Mittel. Die Bezeichnungen der Operatoren entsprechen denen aus dem Netsimple-Anfrageplan (vgl. Abbildung 2.3) und die Transportbezeichnungen entsprechen den Nummern vor den Operatoren. Zusätzlich ist oben die Häufigkeit wie oft das *Kernelmodul* und die Useranwendung den Prozessor abgegeben hat aufgelistet. Unten ist die mittlere Durchlaufzeit eines Datenpakets und die Varianz der Durchlaufzeit angegeben.

```
{
Kernel-CPUAbgaben: 1524
User-CPUAbgaben: 46
{({AsyncSource 2., ,1171795.7410338116,2409.4988475537875} , {2.
    rel_source , ,5473245.55492561,3107.50328934965})->2 :
    98596.77042985914,5.626207799652899E7},
{({AsyncSource 0., ,393576.1759918352,2041.1024479933653} , {0.
    str_source , ,3.5985301353274725E7,2090.045823746694})->0 :
    4.137362702312176E20,4.322256529052346E10},
{({AsyncSource 1., ,78115.42022160259,2374.501491427632} , {1.
    rel_source , ,5.29395592035119E-5,977.9978427069581})->1 :
    762113.6744210223,2.2996464998986937E7},
{({2. rel_source , ,5473245.55492561,3107.50328934965} , {3.
    bin_str_join , ,2.8793972072667263E7,2105.3075403035446})2->3 :
    7043716.675293191,85066.99668990774},
{({0. str_source , ,3.5985301353274725E7,2090.045823746694} , {3.
    bin_str_join , ,2.8793972072667263E7,2105.3075403035446})0->3 :
    1.05247344361939699E18,4.611625069196502E8},
{({1. rel_source , ,5.29395592035119E-5,977.9978427069581} , {5.
    bin_str_join , ,2.661084278370343E7,1964.9202528538133})1->5 :
    175141.01965584612,59504.49849499182},
{({3. bin_str_join , ,2.8793972072667263E7,2105.3075403035446} , {5.
    bin_str_join , ,2.661084278370343E7,1964.9202528538133})3->5 :
    1.05239426348760422E18,4.6131159127088326E8},
{({5. bin_str_join , ,2.661084278370343E7,1964.9202528538133} , {7.
    sender , ,2.5735685342466865E7,1856.8270039982633})5->7 :
    1.05125958010529318E18,4.5986124849653906E8},
{({7. sender , ,2.5735685342466865E7,1856.8270039982633} , {4.
    str_receiver , ,687331.5401850609,1705.5897107369553})7->4 :
    1.79130968226674893E18,1.9184266367802E9},
{({4. str_receiver , ,687331.5401850609,1705.5897107369553} , {9.
    output , ,7.643939466620187E7,3829.013184962609})4->9 :
    8.170517396606955E14,858773.4371615482},
}
Mittlere Durchlaufzeit: 4.652420164023609E10
Varianz der Durchlaufzeit: 4.1333599348638345E20
```

Output A.3: Laufzeiten der Operatoren und Transporte zwischen den Operatoren für 100000 Pakete/s interne Last. Die Zeiten sind in Nanosekunden. Die erste Zahl ist immer die Varianz und die zweite Zahl das arithmetische Mittel. Die Bezeichnungen der Operatoren entsprechen denen aus dem Netsimple-Anfrageplan (vgl. Abbildung 2.3) und die Transportbezeichnungen entsprechen den Nummern vor den Operatoren. Zusätzlich ist oben die Häufigkeit wie oft das *Kernelmodul* und die Useranwendung den Prozessor abgegeben hat aufgelistet. Unten ist die mittlere Durchlaufzeit eines Datenpakets und die Varianz der Durchlaufzeit angegeben.

```
Kernel-CPUAbgaben: 57
User-CPUAbgaben: 41
{({AsyncSource 2., ,1171811.4934692371,2409.5061235029434} , {2.
    rel_source , ,5.2939559203089924E-5,767.9991540513134})->2 :
    4102662.0323731853,5.628124949745318E7},
{({AsyncSource 0., ,518803.4401761679,2092.2047030263416} , {0.
    str_source , ,3.739029503470323E7,2781.476631874386})->0 :
    3.962993581652353E20,4.1408602196844215E10},
{({AsyncSource 1., ,59777.317850008636,2339.504135305544} , {1.
    rel_source , ,7230746.595589897,3597.000613808632})->1 :
    762126.3781738362,2.441955100221793E7},
{({2. rel_source , ,5.2939559203089924E-5,767.9991540513134} , {3.
    bin_str_join , ,2.787949427834872E7,2591.229930869071})2->3 :
    7230668.335445738,155850.99754507875},
{({0. str_source , ,3.739029503470323E7,2781.476631874386} , {3.
    bin_str_join , ,2.787949427834872E7,2591.229930869071})0->3 :
    9.3169711574754982E17,4.222133973858144E8},
{({1. rel_source , ,7230746.595589897,3597.000613808632} , {5.
    bin_str_join , ,2.695403948091818E7,2363.2180138374315})1->5 :
    9878465.88769551,96729.99294687796},
{({3. bin_str_join , ,2.787949427834872E7,2591.229930869071} , {5.
    bin_str_join , ,2.695403948091818E7,2363.2180138374315})3->5 :
    9.3276577121220006E17,4.2628990519558924E8},
{({5. bin_str_join , ,2.695403948091818E7,2363.2180138374315} , {7.
    sender , ,2.772226510846585E7,2407.209344558362})5->7 :
    9.3353416136862451E17,4.254130957250198E8},
{({7. sender , ,2.772226510846585E7,2407.209344558362} , {4.
    str_receiver , ,1.4700772094550362E7,2013.836910630502})7->4 :
    1.78576588600426214E18,1.7573178573630922E9},
{({4. str_receiver , ,1.4700772094550362E7,2013.836910630502} , {9.
    output , ,2.647870642290752E12,8127.859681115292})4->9 :
    7.532715004314601E14,809785.0743728272},
}
Mittlere Durchlaufzeit: 4.444066861462159E10
Varianz der Durchlaufzeit: 4.018066564483533E20
```

Output A.4: Laufzeiten der Operatoren und Transporte zwischen den Operatoren für 150000 Pakete/s interne Last. Die Zeiten sind in Nanosekunden. Die erste Zahl ist immer die Varianz und die zweite Zahl das arithmetische Mittel. Die Bezeichnungen der Operatoren entsprechen denen aus dem Netsimple-Anfrageplan (vgl. Abbildung 2.3) und die Transportbezeichnungen entsprechen den Nummern vor den Operatoren. Zusätzlich ist oben die Häufigkeit wie oft das *Kernelmodul* und die Useranwendung den Prozessor abgegeben hat aufgelistet. Unten ist die mittlere Durchlaufzeit eines Datenpakets und die Varianz der Durchlaufzeit angegeben.

```
Kernel-CPUAbgaben: 760
User-CPUAbgaben: 254
{({AsyncSource 2., ,1025171.862151718,2269.502147114421} , {2.
    rel_source , ,1260.723838688491,1152.4971341620899})->2 :
    19320.970604595004,5.622642399996189E7},
{({AsyncSource 0., ,2405836.3624023893,2860.3137913071055} , {0.
    str_source , ,1318864.2260272028,1308.5770502530168})->0 :
    5.560690048909525E12,3405188.2989482763},
{({AsyncSource 1., ,98912.3055028508,2339.504135305418} , {1.
    rel_source , ,0.2594038400984372,1117.499778039462})->1 :
    313047.6227035395,2.2417026499107935E7},
{({2. rel_source , ,1260.723838688491,1152.4971341620899} , {3.
    bin_str_join , ,1327988.2043283565,1295.9375718800554})2->3 :
    312486.0841708951,60761.994972843015},
{({0. str_source , ,1318864.2260272028,1308.5770502530168} , {3.
    bin_str_join , ,1327988.2043283565,1295.9375718800554})0->3 :
    9.001182067558714E8,77263.67376973282},
{({1. rel_source , ,0.2594038400984372,1117.499778039462} , {5.
    bin_str_join , ,1232181.0645226117,1291.8321707419782})1->5 :
    311933.17706175236,41625.49885229899},
{({3. bin_str_join , ,1327988.2043283565,1295.9375718800554} , {5.
    bin_str_join , ,1232181.0645226117,1291.8321707419782})3->5 :
    3.695966570673692E9,85953.54489880301},
{({5. bin_str_join , ,1232181.0645226117,1291.8321707419782} , {7.
    sender , ,1268743.8681306846,1263.8738088442851})5->7 :
    9.029694630033039E8,76615.27495408879},
{({7. sender , ,1268743.8681306846,1263.8738088442851} , {4.
    str_receiver , ,2137181.303171141,1727.7647606408652})7->4 :
    2.56253677506268064E17,5.135288339902778E8},
{({4. str_receiver , ,2137181.303171141,1727.7647606408652} , {9.
    output , ,4694434.71080713,3860.3723085524134})4->9 :
    1.1925940571429643E8,37120.45889621181},
}
Mittlere Durchlaufzeit: 5.1722458391318893E8
Varianz der Durchlaufzeit: 2.55029414810364192E17
```

Output A.5: Laufzeiten der Operatoren und der Transporte zwischen den Operatoren. Es werden 1000 Pakete so schnell wie möglich in die Quellpuffer geschrieben und anschließend eine Sekunde gewartet, bis dann wieder 1000 Pakete geschrieben werden. Dies wiederholt sich bis 500000 Pakete verarbeitet wurden. Die Zeiten sind in Nanosekunden. Die erste Zahl ist immer die Varianz und die zweite Zahl das arithmetische Mittel. Die Bezeichnungen der Operatoren entsprechen denen aus dem Netsimple-Anfrageplan (vgl. Abbildung 2.3) und die Transportbezeichnungen entsprechen den Nummern vor den Operatoren. Zusätzlich ist oben die Häufigkeit wie oft das *Kernelmodul* und die Useranwendung den Prozessor abgegeben hat aufgelistet. Unten ist die mittlere Durchlaufzeit eines Datenpakets und die Varianz der Durchlaufzeit angegeben.