UNIVERSITÄT
DUISBURG
ESSEN

Bachelorprojekt

Optimierungsverfahren zur Konfiguration von prioritätsbasierten QoS Verfahren

Michael Krane
2233018

Essen, 01.10.2015

Betreuung: Johannes Specht

Studiengang: Angewandte Informatik – Systems Engineering

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Essen, am 01.10.2015

Zusammenfassung

Prioritätsbasierte QoS Verfahren ermöglichen es, die Latenz eines Datenstroms präzise einzustellen. Das Problem ist, dass es oft nur schwer möglich ist, von einer Latenzvorgabe auf eine gültige Prioritätskonfiguration zu schließen. In diesem Bachelorprojekt wird das Tool "UBSOpti" designt und vorgestellt, welches genau dieses Problem adressiert.

Mit verschiedenen Algorithmen ist es möglich, aus den gegebenen Latenzen eine gültige Konfiguration abzuleiten. UBSOpti ist außerdem in der Lage, die verschiedenen Algorithmen miteinander zu vergleichen und den Prozess der Optimierung visuell darzustellen.

Inhaltsverzeichnis

1 Einleitung und Motivation

In den letzten Jahren ist das Volumen der Datenströme in Netzwerken immer weiter angestiegen[1]. Insbesondere in Autos werden immer mehr Sensoren verbaut[2], die miteinander vernetzt werden müssen. Um diese Datenströme zeitnah verarbeiten zu können, sind zwei grundsätzlich verschiedene Ansätze möglich.

Zum einen ist es möglich, die Bandbreite der Netzwerke durch die Verbesserung der Hardware zu erhöhen. Da die Menge der benötigten Hardware von der Netzwerkgröße abhängt, kann eine Erneuerung oder Neuentwicklung der Hardware jedoch kostspielig sein.

Ein anderer Ansatz ist es, die bereits vorhandene Hardware durch verbesserte Software optimal auszunutzen Eine Möglichkeit dafür sind Quality of Service (QoS) Verfahren.

Eine Untergruppe der QoS-Verfahren sind prioritätsbasiert, z.B. RCSP[3] oder UBS [4]. Diese Verfahren ermöglichen es durch die Zuweisung von Prioritäten, die Verbindungsqualität der einzelnen Datenströme festzulegen. Insbesondere ist dadurch möglich, aus der Prioritätskonfiguration eines Netzwerkes auf die Latenz einer Verbindung zu schließen. Oft ist aber der umgekehrte Fall vorhanden: Die Latenzanforderungen an eine Verbindung sind bekannt und es wird nach einer gültigen Prioritätskonfiguration gesucht, die alle Anforderungen erfüllt.

Im Zuge dieser Arbeit wurde deshalb das Programm UBSOpti entworfen und implementiert. Das Tool ermöglicht es, aus Latenzbedingungen für verschiedene Datenströme eine gültige Prioritätskonfiguration in einem beliebigen Netzwerk zu berechnen. Als Grundlage werden dafür verschiedene Optimierungsalgorithmen verwendet. Darüber hinaus ist es möglich, die Effektivität der Algorithmen zu erfassen und zu vergleichen.

2 Theoretische Grundlagen

2.1 Prioritätsbasierte QoS-Verfahren

Mit Quality of Service [6] wird die Qualität einer Verbindung beschrieben. Die Qualität wird dabei von den folgenden Parameter definiert: Datendurchsatz, Jitter, Latenz, Verfügbarkeit und Paketverlustrate. Anhand dieser Parameter lässt sich die Qualität einer Verbindung in verschiedenen Kategorien einteilen. Innerhalb dieser Kategorien werden Grenzwerte für die genannten Parameter festgelegt und zwar für alle Verbindungen

QoS-Verfahren[7] werden als Oberbegriff für Funktionen genutzt, die es ermöglichen, die oben genannten Parameter festzulegen, und können so garantieren, dass die vorhandene Bandbreite optimal genutzt wird.

Prioritätsbasierte QoS-Verfahren sind eine spezielle Variante, die es ermöglicht, dass nicht alle Verbindungen in einem Netzwerk gleich behandelt werden müssen. Es ist dadurch möglich, dass bestimmte Verbindungen gegenüber anderen bevorzugt werden.

Im Folgenden werden zwei solcher prioritätsbasierten QoS-Verfahren vorgestellt und deren Funktionsweise erklärt.

2.1.1 Rate Controlled Static Priority

RCSP[3] ist ein in den 90er Jahren entwickeltes QoS-Verfahren. Wie in Abbildung 1 zu erkennen, besteht RCSP aus zwei verschiedenen Komponenten:

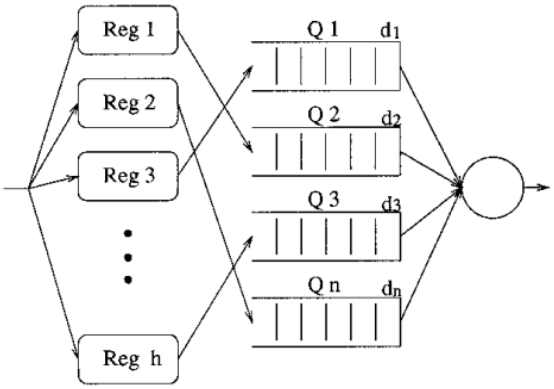


Abbildung 1 - Aufbau RCSP [3]

Die Aufgabe der 1. Komponente, des Regulators ist es, den Zeitpunkt zu bestimmen, an dem ein ankommendes Paket an den "Scheduler" weiter gesendet werden soll. In[3] werden zwei verschiedenen Möglichkeiten für den RCSP-Regulator vorgestellt: "Leaky-Bucket" und "Delay-jitter". Beide

Methoden erlauben es, die Datenströme in ein bestimmtes Muster zu formen.

Beim Leaky-Bucket werden die ankommenden Pakete zuerst in einer Queue gesammelt, um dann mit einer einstellbaren Rate weitergeleitet zu werden. Damit wird am Ausgang ein gleichmäßiger Datenstrom erzeugt.

Beim Delay-Jitter hingegen wird versucht die ankommenden Pakete in derselben Reihenfolge wieder auszugeben. Dies wird mittels einer für jedes Paket einzeln berechneten Verzögerung erreicht.

Die zweite Komponente von RCSP ist der Scheduler, der die verfügbaren Datenpakete weiterleitet. Der Scheduler besteht dabei aus mehreren FIFO-Queues, welche untereinander eine strikte Priorität besitzen. So werden z.B. Pakete in Q1 gegenüber von Paketen in Q2 immer bevorzugt.

Um RCSP in einem Netzwerk einzusetzen, wird pro Hop und pro Datenstrom ein Rate-Controller und pro Priorität eine FIFO-Queue benötigt.

2.1.2 Urgency Based Scheduler

Die UBS-Familie von QoS-Verfahren wird zurzeit an der Universität Duisburg-Essen in Kooperation mit General Motors entwickelt[4]. UBS ist vom Aufbau her ähnlich wie RCSP: Ankommende Pakete werden anhand ihrer Datenstrom-Zugehörigkeit in FIFO-queues einsortiert. Jede dieser FIFIO-Queues ist dabei einem Leaky-Bucket-Shaper zugeordnet. Diese Shaper fügen mittels einer individuell festgelegten Priorität die Pakete in eine Priorityqueue ein.

Aktuell gibt es zwei verschiedene Varianten, die sich hauptsächlich in der Anzahl der benötigten FIFO-Queues unterscheiden:

UBS-V0

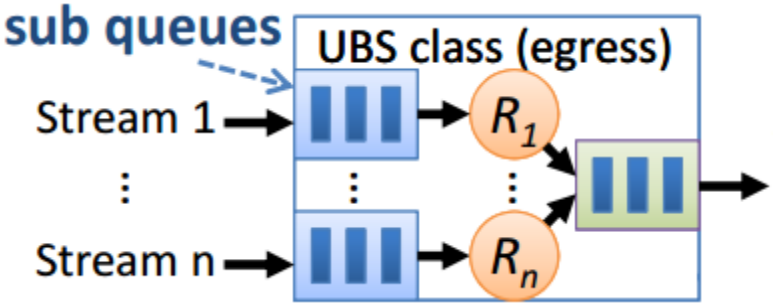
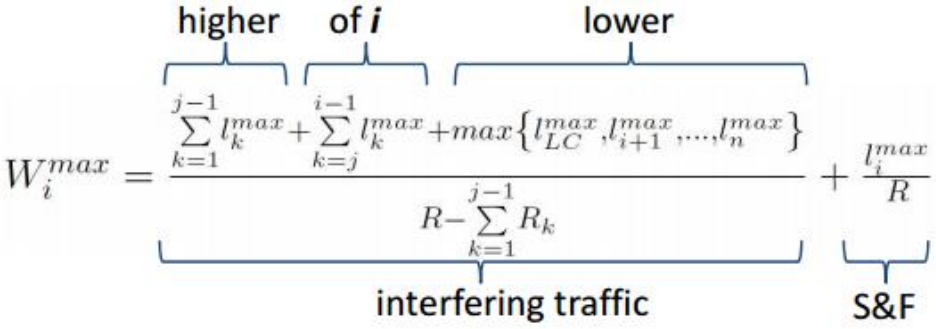


Abbildung 2 – Aufbau UBS-V0 [4]

Wie in Abbildung 2 zu erkennen, erhält in der V0 Variante [4] jeder Datenstrom eine eigene FIFO-Queue und einen Shaper. In dieser Konfiguration lässt sich die maximale Latenz für einen Datenstrom pro Hop durch Formel

1 beschreiben. Es ist jedoch anzumerken, dass die Formel lediglich eine obere Schranke beschreibt, die nur durch Experimente bestätigt wurde, nicht aber durch einen strikten mathematischen Beweis.



Formel 1 - UBSV0 Latenz [4]

UBS-V3

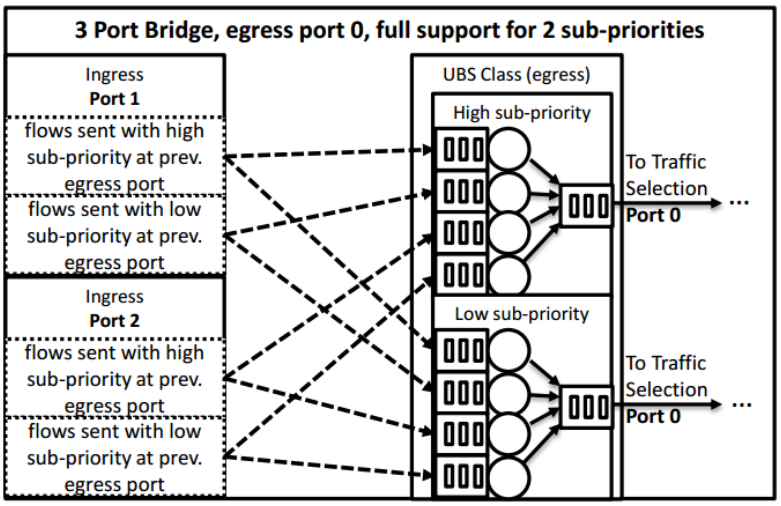


Abbildung 3 – Aufbau UBS-V3 [5]

Abbildung 3 beschreibt den Aufbau der V3-Variante[5], anders als in V0 erhält hier nicht jeder Datenstrom eine eigene FIFO-Queue. Es werden immer mehrere Datenströme anhand verschiedener Kriterien in einer FIFO-Queue zusammengefasst:

- Ingress Port: Der Port, über den der Datenstrom im aktuellen Hop angekommen ist.
- Egress Port: Der Port, über den der Datenstrom den aktuellen Hop verlässt.
- Priorität: Die Priorität im aktuellen und auch die Priorität im vorherigen Port

Dadurch, dass die Streams zusammengefasst werden, verringert sich die Performanz, wie auch in Formel 2 zu erkennen ist. V3 besitzt allerdings gegenüber V0 den großen Vorteil, dass die Anzahl der benötigten FIFO-Queues nicht von der Anzahl der Datenströme abhängig ist. Deshalb ist es möglich V3 ohne eine Veränderung der Hardware beliebig zu skalieren.

$$W_f^{max} \leq \underbrace{\max_{\forall f' \, in \, Q(f)}}_{\text{Max. over all flows sharing The queue with f in the destination port}} \left(\underbrace{\frac{\sum_{i \in H} l_i^{max} + \sum_{i \in S} l_i^{max} + \max_{i \in L}(l_i^{max})}{R - \sum_{i \in H} R_i}}_{\text{Interference by all competing flows In the source egress port (not only The ones in the same queue like f).}} + \underbrace{\frac{l_{f'}^{max}}{R}}_{\text{S\&F}} \right)$$

Formel 2 - UBSV3 Latenz [5]

2.2 Optimierungsverfahren

Als Grundlage für alle Algorithmen wird die Java Bibliothek "goataa"[8] genutzt. Mittels dieser Bibliothek ist es möglich, Optimierungsprobleme durch die Anwendung von verschiedenen Algorithmen zu lösen.

Damit goataa ein Netzwerk optimieren kann, müssen zunächst alle Prioritäten der Datenströme in ein Integer-Array kopiert werden. Dieser Array wird im nachfolgenden "Konfiguration" oder "Prioritätskonfiguration" genannt. Jeder Algorithmus braucht zudem eine Startkonfiguration, die als Startpunkt für die Optimierung dient. Hierfür wird die Konfiguration genommen, in der alle Datenströme überall die höchste Priorität haben.

Zusätzlich muss es eine Möglichkeit geben, Konfigurationen zu bewerten und miteinander zu vergleichen. Dafür wird eine Fitness-Funktion benötigt. Diese Funktion bewertet eine Konfigurationen auf einer Skala von $[0;\infty)$, wobei 0 die beste Bewertung ist. Die benutzte Funktion ist in Abschnitt 3.2 beschrieben.

2.2.1 Brute-Force

Der Brute-Force-Algorithmus dient als Maßstab für die anderen Algorithmen. Es wird nacheinander jede mögliche Konfigurationen probiert. Wird eine gültige Kombination gefunden, bricht der Algorithmus ab.

2.2.2 Hillclimbing

Hillclimbing ist in [8] bereits implementiert. Von der Startkonfiguration ausgehend werden alle möglichen Konfigurationen erzeugt, die sich in einer Priorität von der Startkonfiguration unterscheiden. Ist dabei eine Konfiguration besser als die aktuelle Konfiguration, wird diese übernommen und der Prozess beginnt von vorne. Dabei ist es möglich, dass der Algorithmus stecken bleibt, d.h. keine der neu erzeugten Konfiguration besser ist als die aktuelle.

2.2.3 BackTracking

BackTracking ist eine Möglichkeit, den Hillclimbing-Algorithmus zu verbessern. Es wird zunächst wie beim Hillclimbing vorgegangen. Bleibt der Algorithmus stecken, wird eine Backtracking-Strategie betrieben, d.h. der Algorithmus bricht nicht ab, sondern geht einen Schritt zurück und probiert die nächstbeste Änderung.

2.2.4 Simulated Annealing

Simulated Annealing ist eine weitere Möglichkeit, den Hillclimbing-Algorithmus zu verbessern. Simulated Annealing erlaubt es, eine neu erzeugte Konfiguration auch dann zu übernehmen, wenn diese schlechter als die aktuelle ist. Dieses Verhalten wird über eine "Abkühlfunktion" beschrieben, die zufällig die schlechte Konfiguration übernimmt. Die Wahrscheinlichkeit dafür sinkt aber, je länger der Algorithmus läuft.

2.2.5 Evolutionary Algorithm

Der evolutionäre Algorithmus implementiert eine Vererbungsstrategie. In dieser wird die erste Generation komplett zufällig erzeugt, jede darauf folgende Generation setzt sich aus einer Kombinationen der besten Elemente der aktuellen Generation zusammen.

3 UBSOpti

In diesem Abschnitt wird das Tool UBSOpti vorgestellt und die verschiedenen Features erklärt.

UBSOpti benutzt als Grundlage für die Latenzberechnungen nur die UBS-Familie. Es ist möglich festzulegen, welche Variante genutzt wird (siehe Abbildung 4). RCSP ist nicht implementiert, da der Aufbau im Vergleich zu UBS sehr ähnlich ist, die Formeln zur Latenzberechnung aber deutlich komplexer sind[3].

3.1 Optimierungsmodul

Insgesamt sind drei verschiedenen Möglichkeiten gegeben, um ein Netzwerk zu erzeugen und zu optimieren.

3.1.1 Netzwerkgenerator

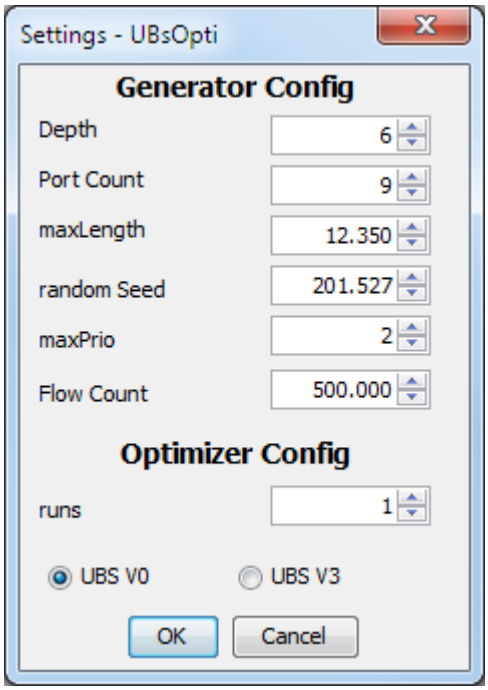


Abbildung 4 - UBSOpti Einstellungen

Es ist möglich, ein zufälliges Netzwerk anhand der gegebenen Parameter zu erzeugen. Dafür wird ein bereits vorhandener "directed acyclic graph"-Generator genutzt, welcher am Lehrstuhl für "Dependability of Computing Systems" der Universität Duisburg-Essen entwickelt wurde. Die für die Optimierung benötigten Latenzanforderungen werden dabei durch eine zufällig erzeugte Prioritätskonfiguration vorgegeben.

3.1.2 .ubsNetwork Dateien

```
{
    "connections" : [
        {"n0" : {"dest" : "b0", "linkSpeed" : "1Gbps"}},
        {"b0" : {"dest" : "b1", "linkSpeed" : "1Gbps"}},
        {"b1" : {"dest" : "b2", "linkSpeed" : "1Gbps"}},
        {"b2" : {"dest" : "n1", "linkSpeed" : "1Gbps"}},
    ],
    "streams" : [
    {
        "streamID" : 1,
        "route"  : {"dijkstra" : { "n0" : ["n1"] }},
        "tspec"  : {"leakRate" : "100Mbps", "maxPacketLength" : "1b", "maxLatency" : "250ns"}
    },
    {
        "streamID" : 2,
        "route"  : {"dijkstra" : { "n0" : ["n1"] }},
        "tspec"  : {"leakRate" : "150Mbps", "maxPacketLength" : "4b", "maxLatency" : "500ns"}
    },
    {
        "streamID" : 3,
        "route"  : {"dijkstra" : { "n0" : ["n1"] }},
        "tspec"  : {"leakRate" : "200Mbps", "maxPacketLength" : "3b", "maxLatency" : "500ns"}
    },
    {
        "streamID" : 4,
        "route"  : {"dijkstra" : { "n0" : ["n1"] }},
        "tspec"  : {"leakRate" : "250Mbps", "maxPacketLength" : "7b", "maxLatency" : "500ns"}
    }
    ]
}
```

Abbildung 5 - Beispiel .ubsNetwork

Es ist möglich, ein Netzwerk aus einer Datei einzulesen. In dieser Datei sind die Topologie und die Streams des Netzwerkes spezifiziert. Des Weiteren ist es möglich, pro Stream eine optionale Latenz-Anforderung anzugeben, fehlt diese, wird der Stream bei der Bewertung einer Prioritätskonfiguration ignoriert.

3.1.3 Massentest

```
X:\Users\tan\Documents\sync\Studium\FS_6\Bachelor-Projekt\UBSOpti>java -jar UbsOpti.jar -help
java UbsOpti [options...] arguments...
 -BT           : BackTrack enabled (Vorgabe: false)
 -HC           : HillClimbing enabled (Vorgabe: false)
 -SA           : Simulated Annealing enabled (Vorgabe: false)
 -factor N     : factor that makes the optimazation easier (Vorgabe: 1.0)
 -help         : displays this hlep message (Vorgabe: true)
 -maxFlow N    : max amount of Flows (Vorgabe: 6)
 -maxPort N    : max amount of Ports (Vorgabe: 20)
 -maxPrio N    : max amount of Prio (Vorgabe: 2)
 -maxStep WERT : max amount of steps before the algo stops trying (Vorgabe:
                 2000000)
 -minFlow N    : min amount of Flows (Vorgabe: 3)
 -minPort N    : min amount of Ports (Vorgabe: 17)
 -minPrio N    : min amount of Prio (Vorgabe: 2)
 -runs N       : runs for each Configuration (Vorgabe: 100)
 -sEA          : simple EvolutionAlgo enabled (Vorgabe: false)
 -seed N       : seed for the random generator (Vorgabe: 4919)
```

Abbildung 6 - Parameter Massentest

Es ist möglich, über ein CLI einen Massentest durchzuführen. Dabei ist es möglich, die Eigenschaften der zu erzeugenden Netzwerke anhand der in Abbildung 6 - Parameter Massentestbeschriebenen Parameter festzulegen.

3.2 Fitnessfunktion

Wie in Abschnitt 2.2 beschrieben, benutzt goataa eine Fitnessfunktion, um eine Konfiguration zu bewerten. UBSOpti benutzt die in Formel 33 beschrieben Funktion, um den Fitnesswert einer Konfiguration zu berechnen.

$$\sum_{\forall f \,\epsilon\, network} \left|delay_f - delaySoll_f\right| + \begin{cases} 2 * \left|delay_f - delaySoll_f\right|, & delay_f > delaySoll_f \\ 0, & sonst \end{cases}$$

Formel 3 – Fitnessfunktion UBSOpti

Der Fitnesswert setzt sich dabei aus der Summe der Differenz der Latenz und der maximalen Latenz aller Ströme zusammen. Zusätzlich wird ein Malus vergeben, falls ein Datenstrom in der Konfiguration die maximale Latenz überschreitet.

3.3 Anzeigemöglichkeiten

3.3.1 Netzwerkdarstellung

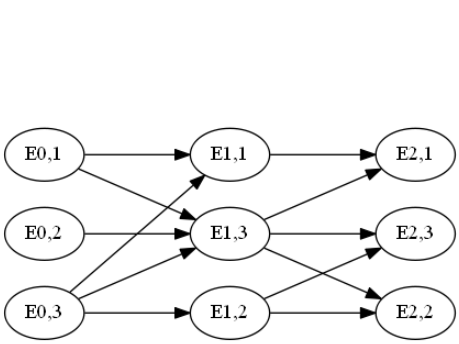


Abbildung 7a – Portdarstellung

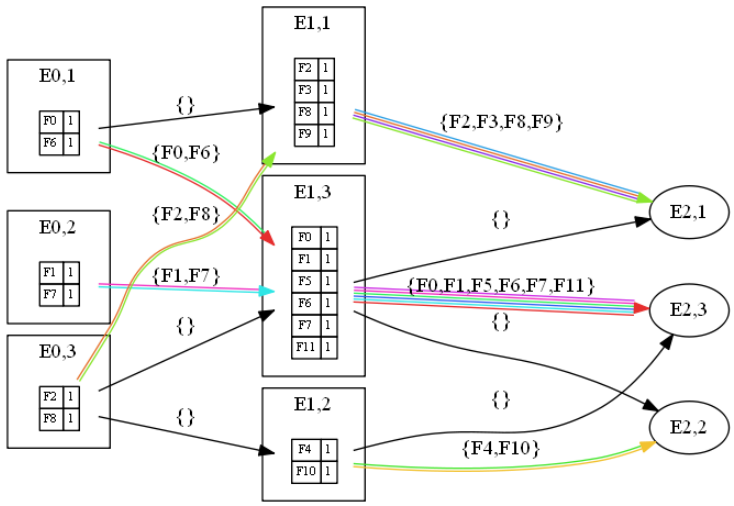


Abbildung 7b - Datenstromdarstellung

UbsOpti kann das geladene Netzwerk auf zwei verschiedene Arten darstellen. Zum einen ist es möglich, alle Verbindungen zwischen den Ports anzuzeigen (siehe Abbildung 7(a)). Zum anderen ist es möglich, die Datenströme einzublenden. In diesem Modus wird zusätzlich die aktuell beste Prioritätskonfiguration angezeigt (Abbildung 7(b)). Die Darstellung basiert dabei, auf einem .dot-File Renderer, welcher am Lehrstuhl für "Dependability of Computing Systems" der Universität Duisburg-Essen entwickelt wurde.

3.3.2 Ergebnisdarstellung

UbsOpti kann die Ergebnisse der Optimierung auf verschiedene Arten grafisch darstellen.

- Es ist möglich, pro Algorithmus die Entwicklung der besten gefundenen Konfiguration darzustellen(siehe Abbildung 8). Für die Darstellung wird die Java-Bibliothek JFreeChart [9] genutzt.

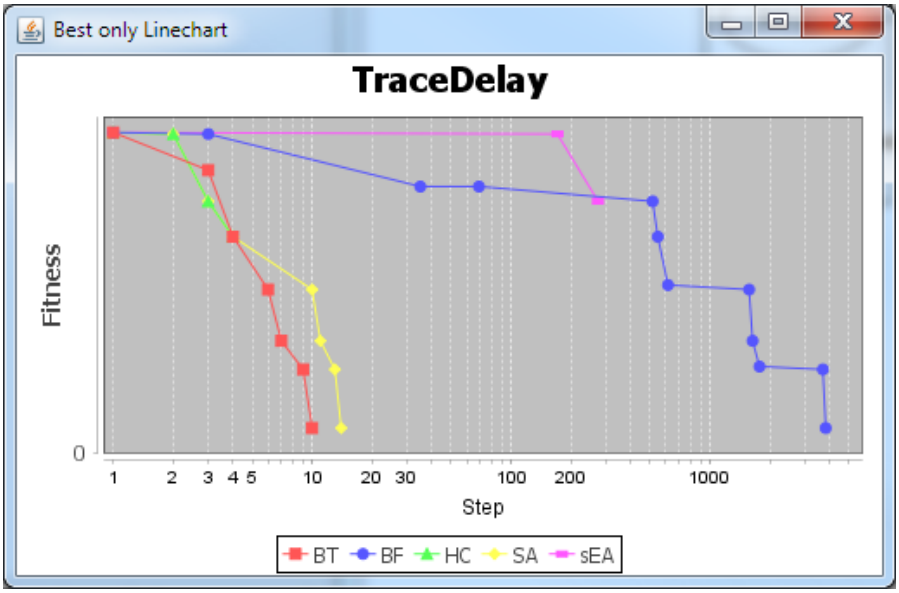


Abbildung 8 - Darstellung der besten gefundenen Konfigurationen für verschiedene Algorithmen

- Es ist möglich, sich jede von einem Algorithmus geprüfte Konfiguration anzusehen. Falls eine gültige Konfiguration gefunden wurde, wird diese in Grün angezeigt. Zusätzlich werden der Vorgänger und die Unterschiede zu der Konfiguration für einen auswählbaren Knoten dargestellt (siehe Abbildung 9). Die grafische Darstellung wird durch die Java-Bibliothek JUNG[10] erreicht.

- Die Ergebnisse des Massentestes werden als .csv-Dateien abgespeichert. Dabei werden die Daten anhand der Netzwerkkonfiguration und benutzen Algorithmen in verschiedene Dateien gespeichert.

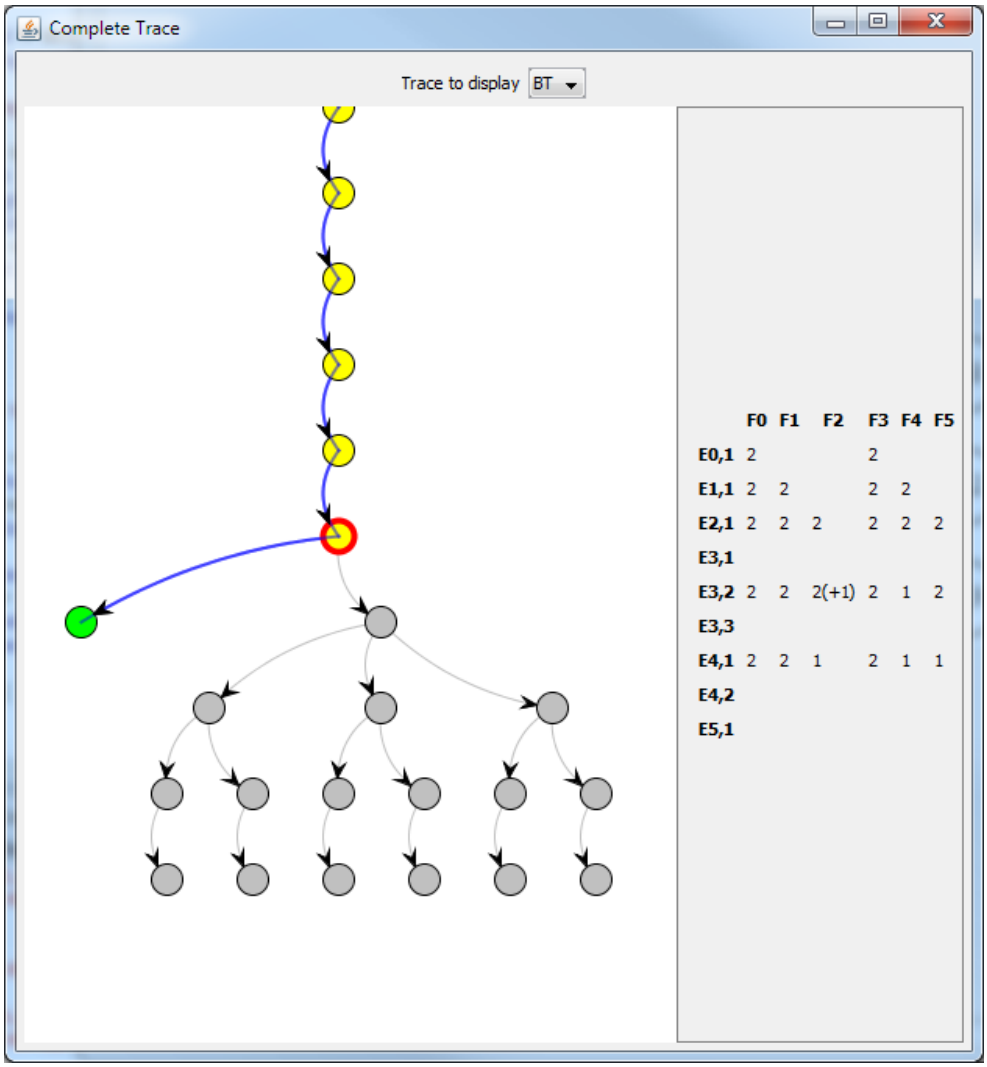


Abbildung 9 – Darstellung der Prioritätskonfiguration für einen Algorithmus.

3.4 Abbruchkriterium

Alle Optimierungsalgorithmen brechen die Optimierung ab, sobald eine Konfiguration gefunden ist, welche alle Latenzbedingungen erfüllt.

Zusätzlich wird für alle Algorithmen eine maximale Schrittzahl festgelegt. Diese Grenze ist eine obere Schranke der Schritte vom BruteForce Algorithmus. Die genaue Anzahl aller gültigen Prioritätskonfigurationen ist nur sehr schwer zu berechnen, deshalb wird die Anzahl der möglichen Konfigurationen abgeschätzt. Dabei ist diese immer größer als die tatsächliche Anzahl.

Zusätzlich kann im Massentest eine weitere obere Schranke festgelegt werden, bei welcher die Optimierung erst gar nicht versucht wird. Dies ist notwendig, da besonders bei großen Netzen die Anzahl der möglichen Kombinationen extrem groß ist.

4 Ergebnisse & Ausblick

In diesem Kapitel werden einige Ergebnisse des Massentests diskutiert und ein Ausblick über eventuelle Verbesserungen von UBSOpti gegeben.

4.1 Ergebnisse Auswertung Massentest

Eine der grundlegenden Fragen, die UBSOpti versucht zu beantworten, ist die Frage: Wie gut lassen sich Optimierungsalgorithmen nutzen, um die Prioritätskonfiguration eines Netzwerkes an die Latenz-Voraussetzungen anzupassen? Daraus ergeben sich die beiden Fragestellungen:

1. Gibt es Unterschiede beim Erfolg der Algorithmen?

2. Gibt es Unterschiede in der Schnelligkeit der Algorithmen?

Um diese Fragen beantworten zu können, wurden mit Hilfe des Massentests Netzwerke mit unterschiedlichen Parametern erzeugt und optimiert. Um die erzeugten Daten auszuwerten, wurde die Scriptsprache R genutzt.

4.1.1 Unterschiede beim Erfolg

Eine Optimierung gilt als erfolgreich, wenn der zu testende Algorithmus weniger Schritte als die in 3.3 beschriebenen Abbruchkriterien gebraucht hat, um eine Lösung zu finden.

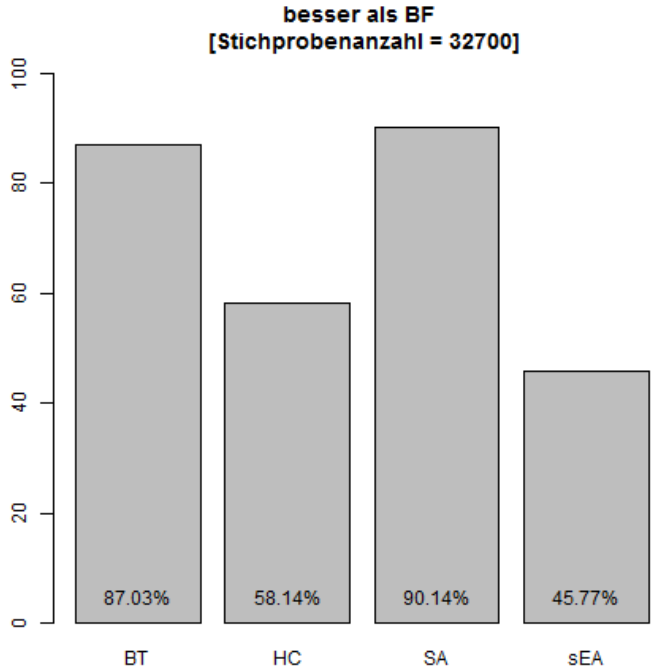


Abbildung 10 – Erfolgchance für die Optimierung

Es ist in der Abbildung 10 deutlich zu erkennen, dass es große Unterschiede zwischen den einzelnen Algorithmen gibt. Insbesondere der Evolutionäre-

Algorithmus (sEA) scheint nicht besonders gut zu funktionieren. BackTracking und Simulated Annealing hingegen schneiden sehr gut ab und scheinen für diese Art von Optimierungsproblemen gut geeignet zu sein.

4.1.2 Unterschiede in der Schnelligkeit

Um die Frage der Schnelligkeit beantworten können, werden nur noch erfolgreiche Optimierungs-Versuche gezählt. Um Abhängigkeiten von den Parametern auszuschließen, wurden die Ergebnisse pro Konfiguration ausgewertet.

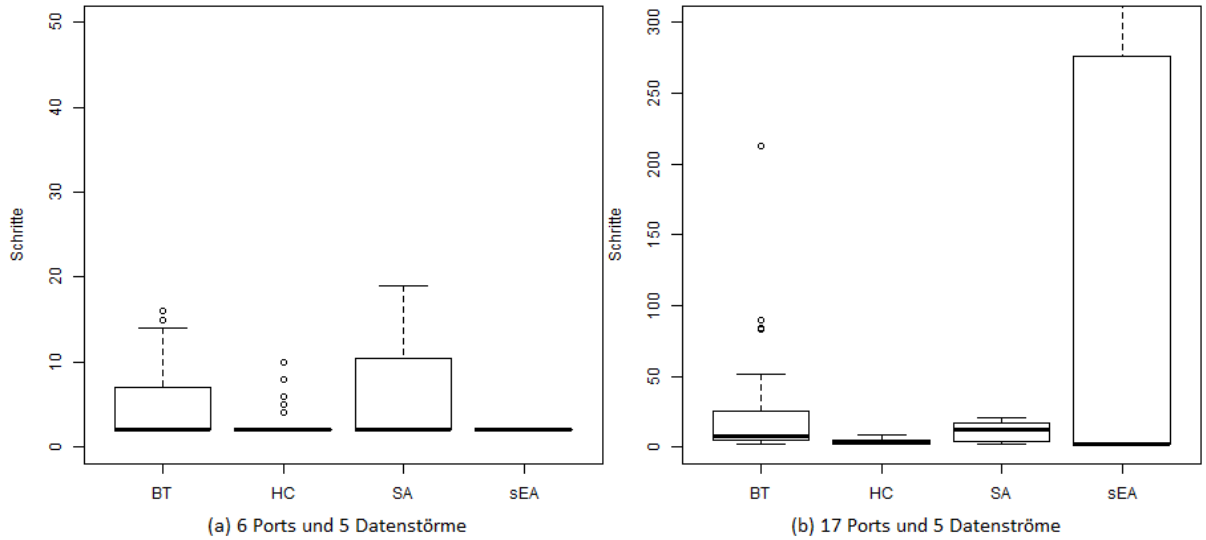


Abbildung 11 – Vergleich Schrittanzahl Optimierungsalgorithmen

Wie Abbildung 11 zeigt, scheint es jedoch keine Korrelation zwischen der Netzwerkgröße und der benötigten Schrittanzahl zu geben. Auch lässt sich kein großer Unterschied zwischen den Algorithmen feststellen. Allerdings weisen die Ergebnisse darauf hin, dass, wenn die Algorithmen eine gültige Lösung finden, dies sehr schnell und unabhängig von der Größe des Netzwerkes passiert.

4.2 Ausblick

Insbesondere an zwei verschiedenen Stellen sind Erweiterungen oder Verbesserungen von UBSOpti möglich:

1. Wahl der Abkühlfunktion von Simulated Annealing

   Wie in Abschnitt 2.2.4 beschrieben, ist es beim Simulated Annealing-Algorithmus möglich, dass der Algorithmus einen schlechteren Wert übernimmt. Dieses Verhalten wird durch die Abkühl-Funktion beschrieben. Die Auswahl dieser Funktion ist in der aktuellen Version fix. Es ist hingegen nicht klar, wie sich die Art der Funktion auf die "Güte" des Algorithmus auswirkt. Es sind weitere Experimente sinnvoll, um eine optimale Abkühlfunktionen zu finden, um den Algorithmus weiter zu verbessern.

2. Wahl der Fitnessfunktion

   Wie in Abschnitt 3.2 beschrieben, setzt sich die "Fitness" einer Konfiguration aus den Differenzen zwischen Soll- und Ist-Werten zusammen. Es ist möglich, dass diese Bewertung nicht optimal ist, sondern noch weitere Werte der Konfiguration berücksichtigt werden müssen, wie z. B. die Anzahl der Streams, welche die Ist-Werte erfüllen bzw. nicht erfüllen. Auch hier sind weitere Experimente sinnvoll, um die Algorithmen zu verbessern.

5 Literatur

[1] Cisco Systems: "Cisco Visual Networking Index: Forecast and Methodology, 2014-2019” (2015) [Online] Verfügbar:http://investor.cisco.com/files/doc_downloads/report_2014/white_paper_c11-481360.pdf

[2] Daimler AG: "Innovations in the new S-Class: Key new products and features at a glance". http://media.daimler.com/dcmedia/0-921-1549267-1-1608172-1-0-0-0-0-1-0-1549054-0-1-0-0-0-0-0.html (13.8.2015)

[3] Zhang, Hui, and Edward W. Knightly: "RCSP and stop-and-go: a comparison of two non-work-conserving disciplines for supporting multimedia communication". *Multimedia systems* 4.6 (1996): 346-356

[4] Specht, Johannes: "Urgency Based Scheduler Performance and Characteristics" (2013) [Online]. Verfügbar: http://www.ieee802.org/1/files/public/docs2013/new-tsn-specht-ubs-perfchar-1113-v1.pdf

[5] Specht, Johannes and Samii, Soheil: "Urgency Based Scheduler Scalability How many Queues?!" (2015) [Online]. Verfügbar: http://www.ieee802.org/1/files/public/docs2015/new-tsn-specht-ubs-queues-0521-v0.pdf

[6] Tanenbaum, Andrew. S. and Wetherall, David J.: "Computernetzwerke, 5. Auflage". Pearson Education (2011): 404-424

[7] Bloom, Andreas A.: "Quality of Service Eine Einführung"(2001) [Online] Verfügbar:http://faq.bintec-elmeg.com/download/de/faq/wp_qos_de.pdf

[8] Weise, Thomas: "Global optimization algorithms-theory and application." *Self-Published,* (2009).

[9] JFreeChar [Online]. Verfügbar: http://www.jfree.org/jfreechart/

[10] Java Universal Network/Graph Framework [Online]. Verfügbar: http://jung.sourceforge.net/