Karlsruher Institut für Technologie

# Reproduzierbare Evaluationsumgebungen mit NixOS

Bachelorarbeit
von


## Maximilian Bosch

an der Fakultät für Informatik


| | |
|---|---|
| Erstgutachter: | Prof. Dr. Frank Bellosa |
| Zweitgutachter: | Prof. Dr. Wolfgang Karl |
| Betreuender Mitarbeiter: | Lukas Werling, M.Sc. |

23. Januar 2025 – 23. Mai 2025

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 23. Mai 2025

# Zusammenfassung

Das Nachbauen von Evaluationsumgebungen, um z.B. Ergebnisse aus Benchmarks zu reproduzieren ist häufig nicht-trivial aufgrund der gestellten Anforderungen: etwa spezifische Software, ein angepasster Kernel oder auch besondere Hardware, häufig mit bestimmter Konfiguration. Diese Schritte werden in langen Shellskripten oder Dokumentation abgebildet und der Nachbau ist dadurch häufig fehleranfällig.

Ich schlage in dieser Arbeit eine Architektur für die Zustandsverwaltung von Systemen vor und habe darauf aufbauend *Smash* implementiert, ein Framework für Zustandsverwaltung von NixOS, einer Linuxdistribution mit Fokus auf deklaratives System- und Konfigurationsmanagement.

Damit demonstriere ich, wie o.g. Anwedungsfälle – das Paketieren der benötigten Software, Verifikation der Anforderung & Aufbereitung der Konfiguration – mit NixOS einfacher zu erfüllen sind.

Das Ergebnis sind sicherere Anwendungsdeployments mit NixOS und wesentlich einfacher aufzusetzende Evaluationsumgebungen, sodass der Fokus bei Artefaktevaluationen weniger auf der Verwaltung der Umgebung und stärker auf der Überprüfung des Inhaltes liegen kann.

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Heutzutage ist es Standard, für Evaluationen von Einreichungen, den Code mitzuliefern, um diese nachzustellen. Ein klassisches Beispiel wäre, dass für ein verbessertes Verfahren der Code für den Benchmark und die Auswertung, bereitgestellt wird. Für große Konferenzen im Bereich Systems Engineering, beispielsweise die Usenix ATC [28], gibt es darüber hinaus Gremien, welche diese Reproduktion testen und dokumentieren [22].

## 1.1 Probleme der Artefaktevaluation

Zur Darstellung der aktuellen Probleme mit Evaluationen habe ich einige Einreichungen gesichtet, dieser Absatz enthält eine Zusammenfassung meiner Beobachtungen. Die Referenzen verweisen auf Beispiele für Artefakte, auf die die jeweilige Aussage zutrifft.

Mein Eindruck ist, dass sich die Reproduktionsarbeit häufig als kompliziert darstellt: die Anweisungen für die Durchführung der Evaluation sind entweder primär in Textform [43, 67, 91] oder in langen, selbstgeschrieben Skripten formuliert [44, 57, 58, 74, 91, 95], was verschiedene Probleme mit sich bringt: zum einen liegt bei ad-hoc geschriebener Automatisierung eine wesentlich höhere Fehleranfälligkeit vor. Solche Skripte sind meist nicht atomar gestaltet, d.h. falls die Installation in der Mitte mit einem temporären Fehler wie Netzwerkproblemen fehlschlägt, ist ein erneutes Ausführen nicht zwingend die Lösung, sondern ein manueller Eingriff nötig, um Zwischenzustände zu entfernen, damit die Vorannahmen des Skriptes wieder stimmen.

Noch dazu reicht es in manchen Fällen aus, dass ein Artefakt bereits als *funktional* gilt, wenn Dokumentation existiert die die Einrichtung ausreichend beschreibt [22].

Zusätzlich ist die Umgebung meistens unterspezifiziert: zum Beispiel durch

eine `requirements.txt` [43, 57] – eine Möglichkeit, um Abhängigkeiten für Python-Programme anzugeben – ohne zugehöriges Lockfile, also nur ein unvollständige Angabe aller benutzen Bibliotheken. Ein andere Fall sind nicht angegebene Versionsanforderungen an Komponenten der Umgebung – etwa dem Compiler oder des Kernels – oder nur die Version der benutzten Distribution [67].

Manuelle Anweisungen haben den Nachteil, dass jeder Tester die Korrektheit jedes einzelnen Schrittes verifizieren muss, was die Anfälligkeit für Fehler gegenüber fertigen Skripten nochmal erhöht, wobei o.g. Nachteile weiterhin gegeben sind. Des Weiteren ist die Korrektheit dieser Anweisungen nicht zwingend gegeben: nachdem die Entwicklung der Artefakte auch ein iterativer Prozess ist, ändern sich im Laufe der Entwicklung die genauen Schritte und es muss sichergestellt sein, dass im Nachgang die korrekten Schritte dokumentiert sind bzw. im Laufe der Entwicklung korrekt angepasst worden sind. Symptomatisch dafür sind Probleme im Nachgang wie z.B. tote Links zu anderen benötigten Repositories [44].

Ein Ansatz, um das Problem zu umgehen, ist die Vorbereitung fertiger Container oder sogar virtueller Maschinen [58]. Dies bringt den Vorteil, dass bestimmte Teile der Reproduktion nicht wiederholt werden müssen. Dies ist allerdings nicht unproblematisch: wenn auch der Code mit geprüft werden soll, muss auch verifiziert werden, dass auf der Maschine genau der veröffentlichte Code läuft. Generell ist es mit signifikant mehr Aufwand verbunden, Binärdateien zu prüfen im Vergleich zu Quellcode. Außerdem schränkt das die Möglichkeit zu experimentieren ein: wenn herausgefunden werden soll, wie z.B. eine Einstellung sich auf das Endergebnis auswirkt, ist es wesentlich einfacher das zu erreichen, indem der Code verfügbar ist und die relevanten Stellen mit der entsprechenden Änderung gebaut werden.

Insgesamt zeigt sich, dass solche Anweisungen nur schwer automatisiert testbar sind. In der Softwareentwicklung wird dies durch sog. kontinuierliche Integration (*continuous integration*), also das regelmäßige Zusammenführen von Änderungen und ausführen automatisierter Tests bei jeder Änderung gelöst. Dies ist für nur teilweise automatisierte Installationen kaum möglich. Dass diese Arbeit nicht-trivial ist, ist in der Informatik bereits lange bekannt: Traugott et al. stellten bereits vor über 20 Jahren fest, wie wichtig die korrekte Reihenfolge einzelner Schritte zur Installation eines Systems ist [103], eine Aufgabe, die – wie oben gezeigt – teilweise komplett manuell erledigt werden muss.

Ein Extremfall, den ich in dieser Arbeit gefunden habe, war ein Entwicklungszweig von LLVM [24] mit einem Syntaxfehler [18, 44]. Weitere Beispiele sind Programmierfehler in der Referenzimplementierung der Einreichung (vgl. Abschnitt 3.1.1 oder inkonsistenten Versionsangaben [58]: die Anwen-

dung ließ sich nur mit Java 8 kompilieren, zur Laufzeit lief diese ohne Abstürze nur mit Java 11.

## 1.2 Über die Möglichkeiten von NixOS

Die oben genannten Probleme erinnern an einige Aspekte der Motivation für die Doktorarbeit von Eelco Dolstra über das Nix-System [53], aus welcher u.a. die Linuxdistribution NixOS entstanden ist. NixOS, basierend auf dem Nix-Paketmanager, bietet deklarative Bauvorgänge, sowie Systemkonfiguration an und transktionale Updates inkl. Rollbacks. Zusammengefasst lassen sich damit gesamte Systemkonfigurationen inkl. aller Softwarepakete mit Code beschreiben. Aufgrund von isolierten Builds und explizit definierten Abhängigkeiten ist sichergestellt, dass sich das Ergebnis nicht implizit auf umgebungsbedingte Seiteneffekte verlässt. Durch die vollständige Beschreibung aller Bauvorgänge mit Code ist die Verteilung der Umgebung auf verschiedene Arten möglich: je nach Anwendungsfall kann die gesamte Umgebung aus den Quellen gebaut werden. Alternativ kann dies auch in einer CI-Umgebung geschehen, um Aufwand für die Prüfer zu sparen. Allerdings können dann einzelne Komponenten trotzdem anhand der deklarativen Konfiguration neugebaut und verändert werden. Generell ist durch Nix eine explizite Verknüpfung zwischen binärem Artefakt und der deklarativen Buildanweisung gegeben. Im Zweifelsfall ermöglicht die aus den o.g. Eigenschaften resultierende Wiederholbarkeit von Bauvorgängen eine Verifikation der binären Artefakte.

Aufbauen lässt sich dies auf `nixpkgs` [80], einer Sammlung von ca. 100000 Paketen und dem Quellcode der Distribution NixOS. Zuletzt zeigte sich auch, dass in aktuellen Versionen von `nixpkgs` über 90% der Pakete nicht nur wiederholt baubar sind, sondern auch bitweise reproduzierbar [75].

## 1.3 Ansätze für Zustandsverwaltung

Ein Anwendungsfall bei der Evaluation von Artefakten, der von NixOS nur unzureichend abgedeckt ist, ist die Verwaltung einzelner Aspekte des Systemzustandes. Dazu zählen beispielsweise das Erzeugen eines Dateisystems mit speziellen Einstellungen, die Verwaltung von Benutzern einer Datenbank oder die interaktive Konfiguration einer Software. Derartige Anwendungsfälle können in NixOS nicht sehr gut dargestellt werden, da solche Operationen in der Regel ein konvergierendes [103] Deploymentwerkzeug erfordert. Häufig werden solche Operationen ad-hoc in Hooks einzelner Systemdienste einge-

baut und haben aufgrund dieser Architektur häufig Probleme [48, 84, 86]. Dieser Vorgang wird auch Reconciliation genannt. Außerdem gibt es – im Vergleich zur Konfiguration und den Paketen – keine eingebaute Möglichkeit zum Zurückrollen.

Diese Problemstellung ist der Hauptfokus dieser Arbeit: ich modelliere und implementiere ein Verfahren zur Zustandsverwaltung im Zuge der Aktivierung einer NixOS-Konfiguration. Dies geschieht auf Basis des Aeolus [106]-Modells nach Zacchiroli et al., welches als konvergenter Ansatz zum Ausrollen von Paketen und dazugehörigen Diensten auf Debian [6] entwickelt wurde. Dabei stellt es ein System als Menge an Komponenten mit bestimmten Zuständen dar und modelliert Deployments als Abfolge bestimmter Zustandsübergänge. Zusätzlich werden Aussagen über die Erfüllbarkeit verschiedener Varianten des Modells getroffen [106]. Dieses wird angepasst, um die Aktivierung einer NixOS-Konfiguration darzustellen. Dabei wird auch ein mögliches Zurückrollen im Fehlerfall betrachtet, sowie Maßnahmen in diesem Fall, beispielsweise das Zurückrollen auf einen vorher angelegten Snapshot des Dateisystems.

Insgesamt zeige ich, dass die Paketierung und Aufbereitung der Artefakte zu leichter zu handhabenden Ergebnissen führt. Hierfür wurden zum einen Einreichungen evaluiert, welche keine besonderen Anforderungen an Zustandsverwaltung haben. Zusätzlich werden auf Basis der Modellierung und Implementierung der Zustandsverwaltung Artefakte, deren Aufbereitung auch Zustandsverwaltung benötigt, evaluiert. Langfristig besteht die Hoffnung, dass dieser Ansatz oder eine vergleichbare Lösung von NixOS selbst übernommen wird, damit für Autoren ausreichend Bordmittel zur Verfügung stehen und der Entwicklungsaufwand eigener Erweiterungen auf Basis dieser Arbeit möglichst gering ist.

# Kapitel 2

# Grundlagen

## 2.1 NixOS

NixOS [79] spielt in dieser Arbeit eine zentrale Rolle. Dieser Abschnitt gibt einen kurzen Überblick über die wichtigsten Aspekte des Nix-Paketmanagers und NixOS, sowie über bestimmte Idiome, die in dieser Arbeit verwendet werden.

Es ist eine Linuxdistribution, welche aus den Arbeiten von Eelco Dolstra und Armijn Hemmel hervorging [53, 60].

Diese zeichnet sich durch transaktionale Updates inkl. Rollbacks und deklarative Systemkonfiguration und Paketverwaltung auszeichnet. Realisiert wird das durch den Nix Store, wie gezeigt in Abbildung 2.1: jedes Buildartefakt - z.B. ein Softwarepaket - landet in einem eigenen Verzeichnis, welches eine Prüfsumme enthält, der aus einer Beschreibung des Buildprozesses, sowie aller Abhängigkeiten (*inputs*) erzeugt wird. Diese Beschreibung wird im Nix-Kontext auch Derivation genannt. Pakete werden dabei so gebaut, dass alle Laufzeitabhängigkeiten in anderen sog. Storepfaden enthalten sind. Auf diese Weise ist es möglich,

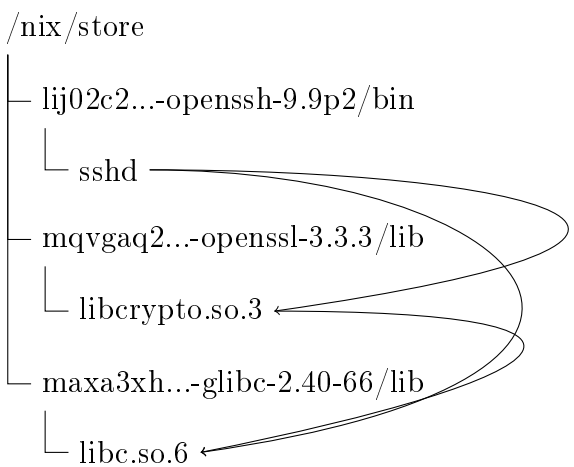


Abbildung 2.1: Auszug des Nix Stores mit Referenzen

mehrere Programme, welche dieselben Abhängigkeiten in unterschiedlichen Versionen haben, nebeneinander zu betreiben. Konfiguration und selbst das gesamte System ist ebenfalls ein Storepfad: dies ermöglicht es, das gesamte System in einer CI vorzubauen und - ähnlich wie einzelne Pakete - über einen sog. Binary Cache zu substituieren. Eine Konfigurationsänderung erzeugt hierbei einen neuen Storepfad. Die Aktivierung selbst ist hierbei atomar, es wird hauptsächlich ein Symlink ausgetauscht. Dadurch ist es möglich, schnell in eine ältere Generation zurückzuwechseln, etwa weil die aktuelle Konfiguration aufgrund eines Fehlers nicht mehr startet. Die Konfiguration wird in einer eigens dafür entwickelten funktionalen DSL geschrieben, der Nix Expression Language, häufig auch nur als Nix bezeichnet.

```
{ stdenv, fetchurl, zlib, openssl, libedit
, pkg-config, autoreconfHook, pam
, withPAM ? true
}:
stdenv.mkDerivation (finalAttrs: {
  pname = "openssh";
  version = "9.9p2";
  src = fetchurl {
    url = "mirror://.../openssh-${finalAttrs.version}.tar.gz";
    hash =
      "sha256-karbYD4IzChe3fll4RmdAlhfqU2ZTWyuW0Hhch4hVnM=";
  };

  buildInputs = [ zlib libedit openssl ];
  nativeBuildInputs = [ autoreconfHook pkg-config ];
  enableParallelBuilding = true;

  configureFlags = [
    (lib.withFeature withPAM "pam")
  ];

  meta = {
    license = lib.licenses.bsd2;
  };
})
```

Abbildung 2.2: Vereinfachte Paketdefinition von OpenSSH

Abbildung 2.2 zeigt eine typische Nix-Paketdefinition, hier eine verein-

fachte Variante von OpenSSH [97]. Das `stdenv` ist eine Buildumgebung, die – sofern nicht anders konfiguriert – davon ausgeht, dass das vorliegende Paket mit einer Abfolge aus `./configure`, `make` und `make install` gebaut werden kann. Es enthält die wichtigsten Werkzeuge aus dem GNU Projekt zum Bauen von Software, etwa den Compiler GCC [99] oder die Bash [98] als Shell.

Das Funktionsargument `finalAttrs` in Zeile 5 ist eine Referenz auf das finale Attributset, das an die Funktion `mkDerivation` gereicht wird. Dies ist möglich, weil Nix *lazy* evaluiert wird. Damit ist es möglich, dass die einzelnen Konfigurationsparameter auf andere Werte zugreifen können, z.B. um eine URL zu erzeugen, welche die Version enthält.

Weitere Programme für den Bauvorgang werden via `nativeBuildInputs` (vgl. Zeile 15) in die Umgebung hinzugefügt. Dies gilt auch für sog. Hooks, also kleine Skripte, die das Verhalten des Bauvorgangs verändern:

Der `autoreconfHook` erzeugt beispielsweise das `configure` Skript. Der Hook von `pkg-config` erzeugt die Umgebungsvariable `PKG_CONFIG_PATH`, welche die Pfade zu den `.pc`-Dateien der Laufzeitabhängigkeiten enthält.

Laufzeitabhängigkeiten werden mit `buildInputs` (vgl. Zeile 14) hinzugefügt. Diese Differenzierung ist für sog. *Crossbuilds* wichtig: wenn für eine andere Plattform gebaut wird, müssen die `nativeBuildInputs` der Plattform der Baumaschine entsprechen, die `buildInputs` der Plattform des Zielsystems.

Um Wiederholbarkeit zu garantieren, muss sichergestellt sein, dass alle Abhängigkeiten explizit definiert sind, d.h. OpenSSH [97] benötigt nur die Eingaben, die in Abbildung 2.2 formuliert sind. Um zu garantieren, dass es keine weiteren Seiteneffekte gibt, die einen Einfluss auf das Paket nehmen, werden die Pakete in einer sog. Sandbox gebaut, welche den Bauvorgang vom Rest des Systems isoliert. Konkret bedeutet das, dass ausschließlich auf die Storepfade der Eingaben zugegriffen werden kann und auf sonst keine weiteren Pfade des Dateisystems. Außerdem wird der Netzwerkzugang beschränkt.

Normalerweise ist der erste Schritt eines Bauvorgangs der Download eines Archivs mit den Quelldateien, wie auch in Abbildung 2.2 der Fall. Hierfür gibt es in Nix eine Sonderform, die sog. Fixed-Output-Derivation: diese bietet weniger Isolation und erlaubt beispielsweise Netzwerkzugriff. Um trotzdem Wiederholbarkeit zu ermöglichen, muss die Prüfsumme des Inhalts des daraus resultierenden Storepfades angegeben werden. Damit ist garantiert, dass das Ergebnis grundsätzlich deterministisch ist. Sollte die Prüfsumme nicht mehr übereinstimmen, schlägt der Bauvorgang fehl.

`nixpkgs` [80] ist ein Repository, welches Paketdefinitionen für 100000 Pakete enthält, sowie den Code für die darauf aufbauende Distribution, genannt NixOS [79].

Einige besondere Idiome aus dem Nix-Ökosystem werden in dieser Arbeit erwähnt:

**Overlays** sind eine Möglichkeit, um eine Menge an Paketen – z.B. `nixpkgs`– zu erweitern [82].

Grundsätzlich sind das Funktionen, welche als Argumente `self` – den Fixpunkt des finalen Paketsets – und `super` – dem Attributset, bevor das Overlay angewendet wurde – erhält.

```
1 self: super: {
2   openssh = super.openssh.override { openssl = self.libressl; };
3
4   was_neues = /* ein Paket */
5 }
```

Abbildung 2.3: Overlay, welches OpenSSH [97] mit einer alternativen Crypto-Bibliothek baut

In Abbildung 2.3 wird das OpenSSH [97] Paket modifiziert, sodass eine alternativen Crypto-Bibliothek benutzt wird. Das `self.libressl` stellt sicher, dass die finale Version des Pakets `libressl` benutzt wird, also das Paket mit allen Modifikationen späterer Overlays.

Die Funktion `.override` ermöglicht die nachträgliche Modifikation der Parameter des Pakets, welche in Abbildung 2.2 ab Zeile 1 definiert werden.

Zu beachten ist, dass das veränderte Paket zur Folge hat, dass jedes Paket in `nixpkgs`, welches OpenSSH [97] als Eingabe benutzt, verändert wird.

Overlays sind nicht nur zur Modifikation gut, in Zeile 4 wird ein neues Paket zu `nixpkgs` hinzugefügt.

**overrideAttrs** ist eine Funktion, um ein Paket zu modifizieren anhand einer Abbildung der alten Attribute, die an eine Derivation gereicht werden – vgl. Definitionen ab Zeile 5 aus Abbildung 2.2 – auf veränderte Attribute, wie dargestellt in Abbildung 2.4.

**NixOS-Konfiguration** ist eine Abbildung der gesamten Konfiguration eines Systems mit Nix-Code. Die Konfiguration setzt sich aus einer Menge an Attributsets zusammen, welche tief zusammengeführt werden. Die Optionen inkl. ihrer Datentypen werden dabei von sog. Modulen deklariert.

Die Konfiguration in Abbildung 2.5 wird u.a. dazu verwendet, um die Datei `/etc/fstab` zu erzeugen.

```
with import <nixpkgs> {};
openssh.overrideAttrs (old: {
  nativeBuildInputs = old.nativeBuildInputs ++ [ krb5 ];
  buildInputs = old.buildInputs ++ [ krb5 ];
  configureFlags = old.configureFlags ++ [
    "--with-kerberos5=${lib.getDev krb5}"
  ];
})
```

Abbildung 2.4: Eine Paketmodifikation mit *overrideAttrs* für Kerberos-Unterstützung

```
{
  boot.initrd.luks.devices.crypted.device =
    "/dev/disk/by-partlabel/nixos";
  fileSystems."/".device = "/dev/mapper/crypted";

  users.users.root.password = "aligator3";
}
```

Abbildung 2.5: Eine NixOS-Konfiguration, die das Gerät des mit LUKS verschlüsselten Rootdateisystems angibt und *root* ein Passwort gibt.

Das Gesamtsystem selbst ist wieder eine Derivation: diese wird im Quellcode von NixOS unter dem Attribute `system.build.toplevel` in diesem Modulsystem deklariert. Der daraus resultierende Storepfad enthält u.a. den Kernel und die Anwendungsverzeichnisse der global installierten Pakete, also `bin`, `lib`, `share` usw., sowie die Konfiguration, die unter `/etc` liegen soll. Aktiviert wird dieses System durch ein Skript aus diesem Storepfad, genannt `switch-to-configuration` [87].

Dieses prüft, welche Dienste hinzugefügt, entfernt bzw. geändert worden sind und startet, stoppt bzw. startet diese neu. Außerdem wird erhält der Bootloader einen weiteren Eintrag im Menü für die aktuelle Generation. Dies hat den Effekt, dass man beim Neustart in ältere Konfigurationsversionen des Systems booten kann. Für die Dateien in `/etc` werden Symlinks in den o.g. Storepfad angelegt. Des Weiteren gibt es noch die sog. Aktivierungsskripte: diese werden für imperative Änderungen genutzt, z.B. um die Benutzer zu aktualisieren und neuen Benutzern eine UID zuzuweisen, wobei keine UID verwendet werden darf, die z.B. ein bereits gelöschter Benutzer hatte, um zu vermeiden, dass der Benutzer auf Dateien ehemaliger Benutzer

zugreifen kann. In Grundzügen ist also bereits eine Reconciliation vorhanden. Allerdings wird hierbei das Zurückrollen nicht beachtet und letztendlich ist das ein relativ strukturloses Skript, in welches einzelne NixOS-Module Codeabschnitte hinzufügen können. Mit anderen Worten, ein ad-hoc Skript mit ähnlichen Problemen wie in Abschnitt 1 bereits angesprochen.

## 2.2 Konfiguration vs. Daten

Die Grenze zwischen Konfiguration und Zustand ist häufig fließend. Ein gutes Beispiel für einen solchen Fall ist das Ökosystem um OpenTofu [101], einem Werkzeug zur Provisionierung von Ressourcen. Hier wird ein Endzustand deklariert, gegen den konvergiert wird. Ressourcen können hierbei Objekte der von OpenTofu beschriebenen Infrastruktur, beispielsweise Cloud-VMs, anlegen, welche mit spezifischem Code – den Providern – erzeugt werden. Ein Beispiel hierfür ist die Integration für AWS [2].

```
1 resource "aws_instance" "ipv6_instance" {
2   ami = "ami-0822295a729d2a28e"
3   instance_type = "p3dn.24xlarge"
4   # ...
5 }
```

Abbildung 2.6: Ausschnitt einer Konfiguration zur Erzeugung einer VM in AWS [2] in OpenTofu.

Auch wenn die Deklarationen in Abbildung 2.6 den Eindruck vermitteln, dass es sich hierbei nur um Konfiguration handelt, wird hierbei der Zustand des eigenen AWS-Kontos verändert. Ein Prozess, der genauso gut interaktiv durchgeführt werden kann. Dieser Abschnitt grenzt Konfiguration und Zustand voneinander ab. Beide Begriffe werden in dieser Arbeit auf Basis dieser Unterscheidung verwendet.

Unter Konfiguration werden hier Strukturen, meistens Dateien, verstanden, welche durchgehend vom Konfigurationsmanagementwerkzeug, in diesem Fall NixOS, verwaltet werden. Dies beinhaltet auch, dass wenn die Konfiguration gelöscht wird, die Strukturen im System entfernt werden.

Ein Beispiel dafür ist die Konfiguration eines Systemdienstes, wie etwa `systemd-networkd` [20] zur Netzwerkkonfiguration. Sämtliche Parameter werden in Nix-Code formuliert, woraus dann eine Datei im `ini` Format für den Dienst generiert wird. Ein Administrator oder Benutzer editiert dieses

Ergebnis nicht manuell. Wenn der Dienst deinstalliert wird, verschwinden auch die dazugehörigen Konfigurationen.

Im Gegensatz dazu ist der Zustand des Systems etwas, das von Benutzern oder Programmen beeinflusst wird. Ein Beispiel wäre die Tabelle einer Datenbank: diese wird von einem Programm zur Laufzeit, etwa durch die Eingabe eines Benutzers, modifiziert. Diese Struktur kann auch Konfiguration enthalten, ist damit aber Teil des Zustandes. Ein Beispiel für einen solchen Grenzfall ist PowerDNS [64], mit welchem DNS Zonen mit beispielsweise einer Datenbank interaktiv konfiguriert werden können.

Unterschieden wird hier beispielsweise von Traugott et al. [103] zwischen konvergenter, divergenter und kongruenter Verwaltung. NixOS fällt in Sachen Konfigurationsmanagement in die Kategorie kongruent, d.h. die gesamte Konfiguration entspricht ihrer Beschreibung [103], dem Nix-Code. Eine Modifikation wird dadurch vermieden, dass in den Store nur der Daemon bei der Erzeugung eines Pfades schreiben darf und dieser sonst unveränderlich ist. Eine Abweichung ist per Definition ein Verfahrensfehler oder Sicherheitsvorfall [103].

In dieser Arbeit werden sog. Benutzerdaten vom Zustand des Systems ausgenommen. Wie oben ausgeführt, gibt es aber auch Grenzfälle, wo Konfiguration tatsächlich Daten sind. Dieser Aspekt wird unter NixOS tatsächlich divergent behandelt, d.h. initial gab es einen definierten Zustand, von dem kontinuierlich abgerückt wird [103].

Das bedeutet auch, dass es zwischen mehreren Konfigurationsaktivierungen beliebige Änderungen an solchen Zuständen geben kann: im Falle einer interaktiven Anwendung – wie o.g. PowerDNS – kann es zwischen Konfigurationen auch Änderungen einzelner Benutzer geben. Wenn dies nicht gewollt wäre, wäre es a priori sinnvoller, solche Aspekte über Storepfade darzustellen, welche nur Leseberechtigung haben und damit unveränderlich sind. Diese Abweichungen bedeuten, dass für dieses Problem keine kongruente Lösung nach Traugott et al. [103] existiert und dieser Abschnitt mit einer konvergenten Lösung implementiert werden muss.

Dass solche Grenzfälle ein reales Problem sind, zeigt sich beispielsweise an der o.g. Existenz von OpenTofu, sowie der großen Menge an verfügbaren Providern.

Ein weiterer Fall sind Aspekte des Systems, die nicht mit Softwarekonfigurationsmanagement verändert werden können. Ein Beispiel hierfür ist das Hyperthreading [65], eine Technologie, die es ermöglicht, auf einem physischen CPU-Kern mehr als einen Thread laufen zu lassen – eine Einstellung, die von der Firmware konfiguriert wird – oder allgemein die Eigenschaften der verbauten Hardware. In diesem Fall muss sich Zustandsverwaltung darauf beschränken, sicherzustellen, dass diese Aspekte den Erwartungen entsprechen

oder dies als Fehler in der Aktivierung der Konfiguration behandeln.

## 2.3 Vorhandene Umsetzungen von Zustandsverwaltung

In Abschnitt 2.1 habe ich erläutert, dass Zustandsverwaltung in NixOS nur schwach ausgeprägt ist. Dennoch gibt es bereits einige Umsetzungen dafür. Dieser Abschnitt gibt einen kurzen Überblick über die aktuellen Lösungen und deren Probleme.

Eine von den stärkeren Implementierungen ist die Behandlung von UNIX-Benutzern: die Herausforderung ist hier, dass es sowohl möglich ist, manuell Benutzer anzulegen, dies aber auch über die NixOS-Konfiguration zu erledigen. Dabei ist wichtig, dass Benutzer aus der Konfiguration kongruent behandelt werden, d.h. gelöscht werden, falls diese entfernt werden.

Gleichzeitig wird aber deren Zustand – z.B. das Homeverzeichnis – nicht gelöscht. Wenn nun aber ein neuer Benutzer angelegt wird, darf dieser keinen Zugriff auf die Daten eines anderen Benutzers erhalten. Gleichzeitig soll, wenn ein gelöschter Benutzer wieder angelegt wird, dieser wieder Zugriff erhalten. Um das zu realisieren, speichert NixOS die IDs von aktuell und ehemals verwalteten Benutzern in `/var/lib/nixos`. Dieser Ansatz ist die Inspiration für die Reaktivierung aus Abschnitt 4.1.

Streng genommen ist `switch-to-configuration` [87] auch ein Fall von Zustandsverwaltung: bei der Aktivierung einer Konfiguration ist das System in einem bestimmten Zustand, d.h. eine bestimmte Menge an Diensten ist zu der Zeit aktiv. Dieses Programm ist dafür zuständig, den Stand des Systems in Hinblick auf Dienste oder auch Geräte wieder dem gewünschten Zustand anzugleichen. Deshalb benutze ich dieses Programm auch in Smash weiter, siehe Abschnitt 5.1.1.

Ein negatives Beispiel für Zustandsverwaltung ist der Umgang mit der Datenbank PostgreSQL [71]: Datenbanken und deren Benutzer sind für vielen Anwendungen ein wichtiger Teil ihrer Konfiguration, sodass der Wunsch besteht, diese über NixOS konfigurieren zu können. Aus Sicht von Postgres handelt es sich nach Abschnitt 2.2 um Daten.

Aktuell wird das in `nixpkgs` [80] gelöst, indem ein Skript generiert wird, welches nach Start des Datenbankdienstes Benutzer und deren Inhaber anlegt. So gab es in der Vergangenheit die Möglichkeit, Rechte (*Permissions*) an Datenbanknutzer zu geben. Wenn diese aus der Konfiguration entfernt werden, werden diese Rechte nicht entzogen [36]. Das war nicht das einzige Problem mit diesem Ansatz, Rechte zu verteilen [84]. Auch wenn letzteres

Problem von mir beseitigt wurde, ist das nicht der einzige Mangel: zusätzlich blockiert diese Implementierung potentiell die WAL-Wiederherstellung [48] und das Anlegen von Datenbanken wird pro Neustart ausgeführt, was allerdings den Dienst bricht, wenn sich die Version der `libc`-Locale geändert hat [86]. Zuletzt ist zu bedenken, dass die Entfernung des Verteilens von Rechten zur Folge hat, dass es keine Möglichkeit gibt, feingranular Rechte an Benutzer zu verteilen.

Außerdem kam die Community in der Vergangenheit zu dem Ergebnise, dass es sich hier per se um den falschen Ansatz handelt, um komplexere Änderungen durchzuführen [78]. Zusammengefasst gibt es – von zwei positiven Ausnahmen abgesehen – entweder gar keine Möglichkeiten der Zustandsverwaltung oder schwach ausgeprägte Implementierungen mit vielen Nachteilen.

Ich persönlich halt das von mir hier geschilderte Vorgehen für den falschen Ansatz, was eine weitere Motivation für die Architektur von Smash ist.

## 2.4 Varlink

Varlink ist ein Protokoll und Format für die Beschreibung von Schnittstellen [93], welches – im Vergleich zu Alternativen wie DBus [54] – sehr einfach gehalten ist. Dieses wird in Smash für IPC benutzt, dieser Abschnitt gibt einen sehr kurzen Überblick darüber.

Das Prinzip von Varlink ist die Übertragung von Nachrichten im JSON-Format [7] über einen Socket, wobei einzelne Nachrichten mit einem Nullbyte terminiert werden [92,93]. Der Vorteil von diesem Ansatz ist, dass das wesentlich leichter zu debuggen ist, z.B. kann man die Nachrichten mit `strace` [90] direkt auslesen und benötigt nicht noch ein weiteres Werkzeug, um die Nachrichten zu dekodieren.

Ein Varlink-Dienst liegt in einem Namensraum. Es können auf der Gegenseite Methodenaufrufe durchgeführt werden, für die Parameter und Rückgabewerte definiert werden. Neben primitiven Datentypen ist es möglich, eigene Typen, sowie Fehlertypen zu definieren.

Abbildung 2.7 enthält ein minimales Schema für einen Varlink-Dienst zum Anfragen einer Abschlussarbeit. Die Abschlussarbeit ist in Zeile 3 ein eigener Datentyp. Der Integer ist zwar implementierungsabhängig, allerdings gehe ich in dieser Arbeit – der Empfehlung von Varlink folgend [93] – von einem 64 Bit Integer aus.

Ein Aufruf dieser Methode würde in JSON wie folgt aussehen:

Der Standardfall ist ein Methodenaufruf gefolgt von einer Antwort. Multiplexing wird nicht unterstützt, ein Dienst arbeitet die Anfragen in der Reihenfolge ab, in der diese empfangen worden sind [93]. Es ist zusätzlich mög-

```
interface edu.kit.service

type Abschlussarbeit (
  title: string,
  duedate: int,
)

error UngueltigeAnfrage (
  details: string
)

method AbschlussarbeitAnfragen(
  arbeit: Abschlussarbeit,
  pruefer: []string,
) -> (erfolg: bool)
```

Abbildung 2.7: Minimales Beispiel für ein Varlink-Schema

lich, über den Puffer des Socket mehrere Anfragen einzureihen und für einen Dienst mehrere Antworten zu versenden [93].

Analog zu Anfragen, wie in Abbildung 2.8 werden Antworten ebenfalls in ein `parameters`-Feld geschrieben.

## 2.5 Aeolus Modell

Die Ideen hinter diesem Modell stellen die Basis für die Modellierung der Zustandsverwaltung dar. Dieser Abschnitt gibt einen kurzen Überblick über die Ideen und verwendeten Formalismen.

Aeolus [106] stellt ein Deployment – welches auch mehrere Maschinen enthalten kann – über ein Komponentenmodell dar. Jede Komponente besitzt einen Zustandsautomat. Für einen Systemdienst wäre das beispielsweise $\{\text{uninstalled}, \text{installed}, \text{running}\}$. Das Ausrollen der Änderungen wird als Abfolge von Zustandsübergängen in diesem Modell dargestellt. In diesem Fall werden auch Pakete als Zustand behandelt, weil diese explizit die Zustände *uninstalled* und *installed* annehmen können. Das Modell wurde für auf Debian basierende Deployments entwickelt, wo dies notwendig ist, da kein funktionaler Paketmanager wie Nix benutzt wird [106]. Letztendlich handelt es sich also um eine Modellierung konvergenter Deployments. Nach Abschnitt 2.2 also genau das, was für dieses Problem benötigt wird.

```
{
  "method": "edu.kit.service.AbschlussarbeitAnfragen",
  "parameters": {
    "arbeit": {
      "title": "Reproduzierbare Evaluationsumgebungen mit NixOS",
      "duedate": 1748037599
    },
    "pruefer": ["Prof. Dr.-Ing. Frank Bellosa"]
  }
}
```

Abbildung 2.8: JSON-Daten für den Methodenaufruf aus Abbildung 2.7

Wie Abbildung 2.9 zu entnehmen ist, sind die Abhängigkeiten zwischen einzelnen Komponenten an die jeweiligen Zustände gebunden. Dies wird über sog. Ports oder Schnittstellen realisiert: beispielsweise bietet die Komponente *MySQL* im Zustand *running* die Schnittstelle *mysql_up*, welche eine Bedingung für das Erreichen des Zustandes *running* der Komponente *WordPress* ist. Entsprechend ergibt sich aus diesem Modell auch die Reihenfolge der Installationsschritte [106].

Formal wird eine Komponente als Quintupel $U := \langle Q, q_0, T, P, D \rangle$ beschrieben mit [106]

- der Menge aller möglichen Zustände $Q$
- dem Initialzustand $q_0$
- Der Menge aller Zustandsübergänge $T \subseteq Q \times Q$
- Dem Tupel $P := \langle \boldsymbol{P}, \boldsymbol{R} \rangle$ mit den Mengen aller angebotenen Schnittstellen, sowie aller benötigten Schnittstellen.
- Der Funktion $D : Q \to (\boldsymbol{P} \to \mathbb{N}_\infty) \times (\boldsymbol{R} \to \mathbb{N}_0)$, welche eine Abbildung eines Zustandes ist auf die angebotenen und benötigten Schnittstellen dieses Zustandes, sowie der minimal benötigten bzw. maximal angebotenen Menge.

Im obigen Fall würde für die Komponente *Load Balancer* gelten:

- $Q = \{\text{uninstalled}, \text{installed}, \text{running}\}$
- $q_0 = \text{uninstalled}$

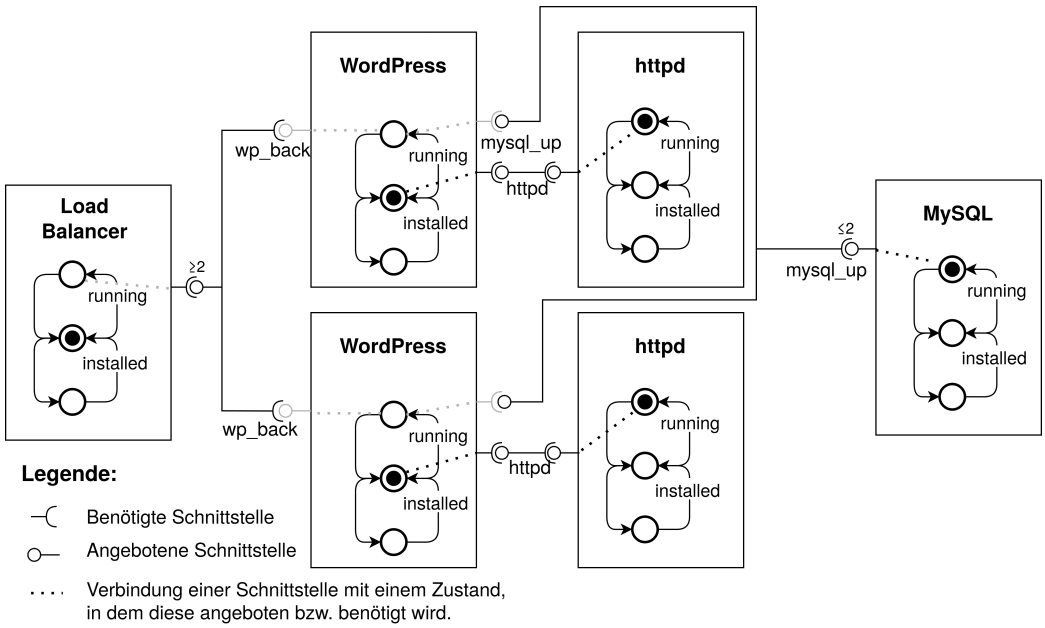


Abbildung 2.9: Beispiel eines Aeolus-Komponentenmodells

- $T = \{\text{uninstalled} \rightarrow \text{installed}, \text{installed} \rightarrow \text{running},$
  $\text{running} \rightarrow \text{installed}, \text{installed} \rightarrow \text{uninstalled}\}$
- $P = \{\{\}, \{\text{wp\_back}\}\}$
- $D = \{\text{uninstalled} \rightarrow \langle\emptyset, \emptyset\rangle, \text{installed} \rightarrow \langle\emptyset, \emptyset\rangle$
  $\text{running} \rightarrow \langle\{\}, \{\text{wp\_back} \rightarrow 2\}\rangle\}$

Die Angabe $\leq 2$ der MySQL-Komponente in Abbildung 2.9 bedeutet, dass höchstens zwei Komponenten sich an diesen Port binden können. Die Angabe $\geq 2$ für den Load Balancer bedeutet, dass mindestens zwei Instanzen laufen müssen. Diese Einschränkungen – formuliert durch $D$ – werden genutzt für Kapazitätsplanung [106]: die Anwendung wird hier erst als stabil bzw. fertig ausgerollt betrachtet, wenn eine bestimmte Menge an Instanzen in Betrieb sind. Es wird sichergestellt, dass die Datenbank nicht zu viele Clients hat und um Zweifelsfall muss hochskaliert werden.

Inkompatibilitäten zwischen Komponenten werden ebenfalls über diese Ports dargestellt: wenn eine Komponente von einer Schnittstelle 0 Verbindungen benötigt, bedeutet dies einen Konflikt, falls eine andere Komponente diese Schnittstelle anbietet. Daher ist der Rückgabewert der zweiten Funktion des Tupels von $D$ auch $\mathbb{N}_0$ [106].

Die Gesamtheit des Deployments wird als Konfiguration bezeichnet. Diese ist ein Quadrupel $C := \langle U, Z, S, B \rangle$ bestehend aus [106]:

- Der Menge $U$ aller existierender Komponenten
- Der Menge $Z$ aller aktuell ausgerollter Komponenten
- Die Beschreibungsfunktion $S : Z \to \langle T \in U, q \in Q \rangle$, welche eine aktuell ausgerollte Komponente assoziiert mit einem Tupel bestehend aus ihrem aktuellen Zustand $q$ und dem o.g. Quintupel, der Komponentenbbeschreibung
- Der Menge $B \subseteq I \times Z \times Z$, welche die Verbindungen zwischen Komponenten beschreibt: $I$ ist hierbei die Menge aller Schnittstellen – z.B. *mysql_ up* in Abbildung 2.9 – assoziiert mit der Komponente, die diese Schnittstelle benötigt und der Komponente, die diese anbietet.

Zusätzlich seien folgende Notationen definiert [106]:

- $C[z] := \langle T, q \rangle = S(z)$ als Aufruf einer Komponentendefinition aus einer Konfiguration $C$.
- Die Postfixoperatoren *.type* und *.state*, für jeweils $q$ bzw. $T$ aus $C[z]$.
- Die Postfixoperatoren *.states*, *.init* und *.trans* für $Q$, $q_0$ und $T$ einer Komponentendefinition.
- Die Postfixoperatoren *.prov* und *.req*, welche die Abbildungen auf die Anzahl der angebotenen bzw. benötigten Schnittstellen sind.

Beispielsweise ist also $C[z].type.req(q)$ die Menge aller Abbildungen von benötigten Schnittstellen auf die Anzahl dieser für die Komponente $z$ im Zustand $q$.

Eine Konfiguration wird als korrekt angesehen, wenn die Beschränkungen für Schnittstellen aus $D$ für alle Schnittstellen und alle Komponenten gegeben ist [106].

Ein Deployment ist die schrittweise Veränderung einer Konfiguration, um das System an den gewünschten Zustand anzugleichen. Folgende Schritte sind definiert [106]:

- *stateChange*$(z, q_1, q_2)$ mit $z \in Z$
- *bind*$(r, z_1, z_2)$ mit $z_1, z_2 \in Z, r \in I$

- *unbind*$(r, z_1, z_2)$ mit $z_1, z_2 \in Z, r \in I$
- *new*$(z, T)$ mit $z \in Z$ und Komponente $T$
- *del*$(z)$ mit $z \in Z$

Die ersten Schritte zum Erzeugen von der Zustandes von Abbildung 2.9 wären entsprechend

```
1 new(z1: wordpress)
2 new(z2: httpd)
3 stateChange(z2, uninstalled, installed)
4 bind(httpd, z1, z2)
5 stateChange(z1, uninstalled, installed)
6 new(z3: mysql)
7 stateChange(z3, uninstalled, installed)
8 stateChange(z3, installed, running)
```

Abbildung 2.10: Erste Schritte zum Deployment des Modells von Abbildung 2.9

Das volle Aeolus-Modell ist nicht praktikabel, weil dieses unentscheidbar ist [106]. Daher gibt es zwei mögliche Reduktionen:

- *Aeolus core*, in welchem

  $$\forall z \in Z : \forall q \in z.\text{states} : \forall p \in I : z.\text{prov}(q)(i) = \infty \wedge z.\text{req}(q)(i) \leq 1$$

  gilt, also jede Komponente eine Schnittstelle ausreichend oft anbietet und jede Komponente eine Schnittstelle höchstens ein Mal benötigen darf.

  Die Möglichkeiten zur Kapazitätsplanung fallen damit weg.

  Diese Reduktion ist Ackermann-schwer [106].

- *Aeolus*$^-$, in welchem auch Konflikte entfallen. Die Entscheidbarkeit ist hier polynomiell [106].

Effiziente Möglichkeiten, auf Basis dieses Modells konvergente Deployments darzustellen, wird u.a. in dem daraus hervorgegangenem Projekt Zephyrus [42, 106] untersucht.

Für die Anwendungsfälle dieser Arbeit ist die polynomiell entscheidbare Variante, Aeolus$^-$, ausreichend, welche die Basis für das vorgeschlagene Zustandsverwaltungsmodell von NixOS bildet.

# Kapitel 3

# Paketierung von Evaluationsumgebungen

Ziel der Arbeit ist es, zu zeigen, dass NixOS sich für den Einsatz in Evaluationsumgebungen eignet. Der Status Quo ist, dass Benchmarks, sowie deren Bauprozesse, in Shellskripten oder in Prosa in der Dokumentation formuliert werden. Diese treffen viele Vorannahmen, welche häufig nicht verifiziert werden. Nach Abschluss des Benchmarks wird oft nur unvollständig aufgeräumt.

NixOS zeichnet sich durch deklaratives Paket- und Konfigurationsmanagement, sowie ein sehr großes Paketset aus. Es fehlt noch an Funktionalität zur Verwaltung des Zustandes von System und installierten Anwendungen. Dieser Abschnitt soll zeigen, wie mit bereits existenten Bordmitteln Benchmarks vorbereitet werden können und welche Effekte das mit sich bringt.

Hierfür wurden die Artefakte zweier Einreichungen verifiziert. Ersteres stellt virtuelle Speicherverwaltung im Userspace [68] vor, letzteres optimierte Variante von TCP BBR, welche mit weniger Retransmissions [34] auskommt. Darüber hinaus weisen die Evaluationsumgebungen beider Arbeiten die o.g. Probleme auf.

## 3.1 ExtMem

„ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications“ [68] wurde auf der USENIX ATC2024 eingereicht.

Jalalian et al stellen vor, wie anwendungsspezifisch der virtuelle Speicher im Userspace verwaltet werden kann. Dafür wurde ein Framework entwickelt, welches Entwickler bei dieser Aufgabe unterstützen soll, gleichzeitig weitestgehend kompatibel mit heutigen Linuxinstallationen ist und auch bei

Multithreading gut funktioniert [68].

Das Framework baut auf `userfaultfd` [26] auf, einem Subsystem des Linux-Kernels, welches es ermöglicht, Seitenverwaltung im Userspace durchzuführen. Konkret werden neue Seiten angelegt, indem die Adressbereiche über den `UFFDIO_REGISTER ioctl` [25] registriert werden. Über einen Kernelpatch wurde `userfaultfd` so erweitert, dass Seitenfehler nicht über IPC-Mechanismen, sondern über einen Upcall mit `SIGBUS` an den Userspace gereicht werden. Dieser Ansatz ist mit einem geringeren Overhead und damit besserer Performance verbunden [68].

Systemaufrufe an `mmap` und `munmap` werden mit `syscall_intercept` [66] abgefangen und ebenfalls über ExtMem behandelt. Prozesse können dies benutzen, indem die C-Bibliothek mit `LD_PRELOAD` (siehe `ld.so(8)` [9]) geladen wird. Das Framework implementiert die grundlegenden Operationen für Speicherverwaltung und verwendet standardmäßig das LRU Verfahren mit je einer Warteschlange für genutzte bzw. ungenutzte Seiten, ähnlich der Implementierung des Linux-Kernels. Zu Demonstrationszwecken wurde eine zweite Bibliothek zur Verfügung gestellt, welche für Verarbeitung von Graphen optimiert wurde, indem die Seiten, die die Liste mit Knoten enthält, nie in den Swap kommt, aber immer nur der Abschnitt der Kantenliste, die derzeit verarbeitet wird, im RAM gehalten wird [68].

Die Evaluation vergleicht in verschiedenen Situationen den Durchsatz der virtuellen Seitenverwaltung des Linuxkernels mit der von ExtMem, sowie einer für Graphverarbeitung optimierten Implementierung von ExtMem [67].

### 3.1.1 Paketierte Komponenten

Das Framework wurde gemeinsam mit Tooling zum Ausführen von Benchmarks veröffentlicht [67]. Es enthält einen angepassten Kernel, welche den Upcall für Seitenfehlerbehandlung im Userspace implementiert, ExtMem und einige Programme für die Durchführung der Benchmarks:

- `mmapbench` [11], welches einen 16 GB großen Bereich im virtuellen Speicher alloziert und diverse Muster dort hineinschreibt [68], womit der Durchsatz der Speicherverwaltung gemessen werden kann. Insbesondere wird mit `cgroups` [4] der physische Speicher auf 8 GB begrenzt, sodass das Verhalten bei hohem Druck und viel Swapping nachvollzogen werden kann.

- GAP [32], ein Benchmarkingwerkzeug zur Evaluation von Graphverarbeitungsverfahren. Mit diesem wurde die dynamische Bibliothek (*shared Object*) evaluiert, welche auf ExtMem basiert und für die Verarbeitung von Graphen optimiert wurde. Als Graph werden auf Twitter

gesammelte Nutzer-IDs [70] und deren Follower verwendet. Im Benchmark wird auf diesen Graph PageRank ausgeführt, ein Verfahren, welches von Google zur Bestimmung der Popularität von Webseiten - den Knoten - anhand ihrer Hyperlinks - den Kanten - entwickelt wurde [89].

All diese Komponenten habe ich im Zuge dieser Arbeit paketiert.

Die Pakete in `nixpkgs` [80] sind in einem sog. Attributset angeordnet, welches mit sog. Overlays [82] erweitert werden kann.

Der Linux-Kernel wurde über die `buildLinux` [85]-API paketiert, welche ein Tarball mit dem Quellcode, Versionsinformationen und optional auch Kernelkonfiguration annimmt und daraus einen Kernel kompiliert. Die Konfiguration wird standardmäßig mit Anpassungen zusammengeführt, die von den Maintainern des Linuxkernel in nixpkgs gewartet werden. Für alle verbliebenen Parameter wird möglichst `m` – d.h. dass die Funktionalität hinter diesem Konfigurationsparameter nicht in den Kernel kompiliert wird, sondern ein Kernelmodul dafür erzeugt wird – gesetzt, außer es gibt einen anderen Standardwert.

Im Fall des Kernels von ExtMem sieht das wie folgt aus:

```
1  self: super: {
2    extmem = {
3      linux-src = self.applyPatches rec {
4        /* Kernel Quellcode mit Patches */
5      };
6
7      kernel = self.buildLinux {
8        pname = "linux-extmem";
9        version = "5.15.0";
10       src = self.extmem.linux-src;
11       modDirVersion = "5.15.0";
12       structuredExtraConfig = with lib; with kernel; {
13         DEBUG_INFO = mkForce no;
14       };
15     };
16   }
17 }
```

Wie in Zeile 4 erwähnt, waren folgende Patches notwendig:

- In einem Entwicklungswerkzeug gab es ein sog. Use-after-free, welches ab GCC 12 die Kompilierung bricht. Der Fix [50] dafür wurde auf spätere Patchlevel von 5.15 portiert, allerdings basiert die Evaluationsumgebung von ExtMem auf 5.15.0.

- In `pahole` [14] - einem Program zur Untersuchung von Datenstrukturen und Erzeugung von Debuginformationen für BPF Programme - ab Version 1.24 gab es eine Inkompatibilität mit älteren Ständen des Kernels [15]. Da die Patches nicht ohne Weiteres auf 5.15.0 anwendbar waren, habe ich die relevanten Stellen manuell editiert und daraus eine Patchdatei erzeugt.

Hier zeigt sich bereits eine Schwäche, Buildanweisungen in Prosa zu formulieren: mit der Zeit verändert sich die Umgebung und die Anweisungen werden unvollständig. Die o.g. Probleme waren weder dokumentiert noch in dem Quellcode bereits behoben. Mit Nix kann man solche Dinge zum einen deklarativ formulieren. Zum anderen kann man sicherstellen, dass sich keine weiteren Aspekte verändern, indem die Umgebung eingefroren (*Lockfiles*) wird: über z.B. Flakes [13] kann der Autor einer Evaluationsumgebung die Git Revision von nixpkgs [80] setzen, die genutzt wird. Beim Nachbauen der Umgebung wäre damit sichergestellt, dass dieselben Versionen von allen Komponenten des Betriebssystems benutzt werden und folglich auch u.a. der gepatchte Linuxkernel baubar bleibt.

In Zeile 13 wurden noch Debugsymbole im Kernel abgeschaltet, weil der gepatchte Kernel aus mir unbekannten Gründen sonst keine Kernelmodule laden konnte, was für diese Evaluation allerdings nicht kritisch ist.

Alle Pakete für ExtMem werden innerhalb des `extmem`-Attributes definiert, um im globalen Namensraum von nixpkgs [80] potentielle Konflikte zu vermeiden. Der Kernel kann in einer NixOS-Konfiguration nun wie folgt benutzt werden:

```
1  { pkgs, ... }:
2  {
3    boot.kernelPackages =
4      pkgs.linuxPackagesFor pkgs.extmem.kernel;
5  }
```

Die Paketdefinition von von ExtMem sieht wie folgt aus:

```
1  self: super: { extmem = {
2    kernel-headers = self.makeLinuxHeaders {
3      version = "5.15.0";
4      src = self.extmem.linux-src;
5    };
6
7    src =
```

```
8       /* Tarball von GitHub */
9
10    extmem_software = self.callPackage
11      ({ stdenv, extmem, liburing }: stdenv.mkDerivation {
12        pname = "extmem";
13        version = "unstable-2024-10-14";
14        inherit (self.extmem) src;
15        buildInputs = [
16          extmem.syscall_intercept
17          liburing
18        ];
19        sourceRoot = "source/src";
20        postPatch = ''
21          substituteInPlace core.c \
22            --replace-fail mmgr_stats '//mmgr_stats'
23        '';
24        NIX_CFLAGS_COMPILE = [
25          "-I${extmem.kernel-headers}/include"
26          "-g"
27        ];
28        makeFlags = ["all"];
29        dontStrip = true;
30        installPhase = ''
31          find . -name '*.so' \
32            -exec install -D {} $out/lib/{} \;
33        '';
34      }) {};
35  }; }
```

**Anmerkung:** der Code hier ist unvollständig. Auf Patches wird in Abschnitt 3.1.2 eingegangen, die Paketdefinitionen für `syscall_intercept` [66] und GAP [32] ähneln dem von `extmem_software` und bringen keine neuen Erkenntnisse, weshalb diese nur nicht in der Arbeit explizit erwähnt werden.

In Zeile 2 wird eine Funktion aus nixpkgs [80] benutzt, die aus einem Tarball mit dem Quellcode des Linuxkernels ein Paket erzeugt, das nur die Header enthält.

In Zeile 20 wird eine Zeile auskommentiert: diese enthält einen Aufruf an die Funktion `mmgr_stats()`. Die Funktion ist in einem Header deklariert, allerdings im Coderepository nicht implementiert [67]. Diese Funktion wird aufgerufen, nachdem die Statistiken geschrieben wurden, weshalb ich davon ausgehe, dass dies eine Art Hook sein soll, z.B. um die Statistiken zurückzu-

setzen, nachdem diese geschrieben wurden. Da die Existenz dieser Funktion vernachlässigbar scheint und ExtMem sonst nicht startet, weil das Symbol fehlt, habe ich entschieden, den Aufruf im Build rauszupatchen.

In Zeile 29 wird das Entfernen von Debugsymbolen abgeschaltet, was standardmäßig zur Einsparung von Festplattenplatz gemacht wird. Dies liegt daran, dass ich bei den Arbeiten zur Reproduktion ein Segmentierfehler entdeckt habe, welcher untersucht werden musste. Dazu mehr in Abschnitt 3.1.2.

Die Benchmarks sind auf mehrere Shellskripte verteilt, die von der Struktur im Coderepository [67] abhängen. Des Weiteren wurden `cgroups` und Swap manuell verwaltet und die Parameter für die Experimente waren hart kodiert. Nachdem Anpassungen notwendig waren für die Paketierung, habe ich alle Tests in einem Pythonprogramm zusammengefasst.

In Hinblick auf Zustandsverwaltung gibt es zwei erwähnenswerte Aspekte dabei:

- Die `cgroups` wurden in den Evaluationsscripten manuell angelegt und konfiguriert [67], was bereits ein möglicher Anwendungsfall für Zustandsmanagement wäre. Nativ unterstützt NixOS dies auch nicht.

  Es gibt allerdings eine andere Lösung, `systemd-run` [19]. Dies legt ad-hoc einen systemd-Service an und führt das dem Befehl übergebene Programm in diesem Service aus. Das hat den Vorteil, dass man alle Einstellungen von systemd-Services, inkl. `cgroup`-Einstellungen ad-hoc benutzen kann [21].

  Die oben erwähnte Begrenzung des physischen Speichers auf 8 GB kann dann wie folgt vorgenommen werden:

```
1 systemd-run --collect --pty --wait \
2   -p MemoryLimit=8G \
3   -- mmapbench args
```

  Die Optionen `--collect --pty --wait` stellen hierbei sicher, dass die temporäre systemd-Unit im Nachgang aufgeräumt wird, `stdout` und `stderr` weitergeleitet werden und der Aufruf solange blockiert, bis der Benchmark fertig ist.

  Des weiteren kann damit direkt die CPU-Affinität gesetzt werden, um genau einen NUMA-Knoten auszunutzen. Dies wird genauer in Abschnitt 3.1.2 beschrieben.

- Zustandsverwaltung wäre hier bereits im Umgang mit dem Swap notwendig gewesen: der Basistest (d.h. ohne ExtMem) mit `mmapbench` [11]

benötigt 8 GB Swap und 8 GB physischen Speicher, um das Verhalten von ExtMem bei viel Swapping zu messen [67, 68].

ExtMem selbst wird mit `mmapbench` [11] unter denselben Bedingungen getestet. Allerdings benötigt es seinen eigenen Swap, der nicht vom Linux-Kernel mitbenutzt wird. Daher ist es nötig, innerhalb der Tests den Swap zu deaktivieren und später neu anzulegen und zu formatieren.

Für diesen Fall habe ich das mit dem Kontextmanager-Feature aus Python gelöst:

```
from contextlib import contextmanager
from subprocess import check_call

@contextmanager
def swapoff(swap_path):
    check_call(["swapoff", swapdir])
    yield
    check_call(["mkswap", swapdir])
    check_call(["swapon", swapdir])

with swapoff("/dev/nvme0n1p1"):
    # führe extmem Tests aus
    pass
```

Der Vorteil hieran ist, dass nur innerhalb des `with`-Blocks der Swap frei für ExtMem ist. Der Kontextmanager ist äquivalent zu einer Konstruktion aus `try`/`finally`, d.h. selbst im Fehlerfall wird der vorherige Zustand wiederhergestellt.

Dies ist nur eine Hilfskonstruktion für fehlende Zustandsverwaltung, die in diesem Fall ausreichend ist, da die Testmaschine aufgrund der Notwendigkeit eines speziellen Kernels keine Produktivmaschine sein sollte, die von dem Swap abhängt und das erneute Erzeugen eines Swaps hinreichend trivial ist. Fehlerfälle bei `mkswap`/`swapon` werden aber nicht abgefangen und für komplexere Anforderungen an Zustandsverwaltung würde das auch nicht skalieren.

### 3.1.2 Ergebnisse

Im ExtMem Paper gab es drei Abbildungen, die den Durchsatz von ExtMem mit der virtuellen Speicherverwaltung des Linux-Kernels vergleichen. Reproduziert wurden die Evaluationen auf einer Maschine mit 110 GB DDR4 RAM

und einer Intel Xeon Gold 6138 CPU, beides verteilt auf zwei NUMA Knoten. Als Swap wurde eine Intel SSD 750 Series NVMe mit 400 GB verwendet, für das Betriebssystem eine Samsung 960 EVO mit 250 GB Kapazität.

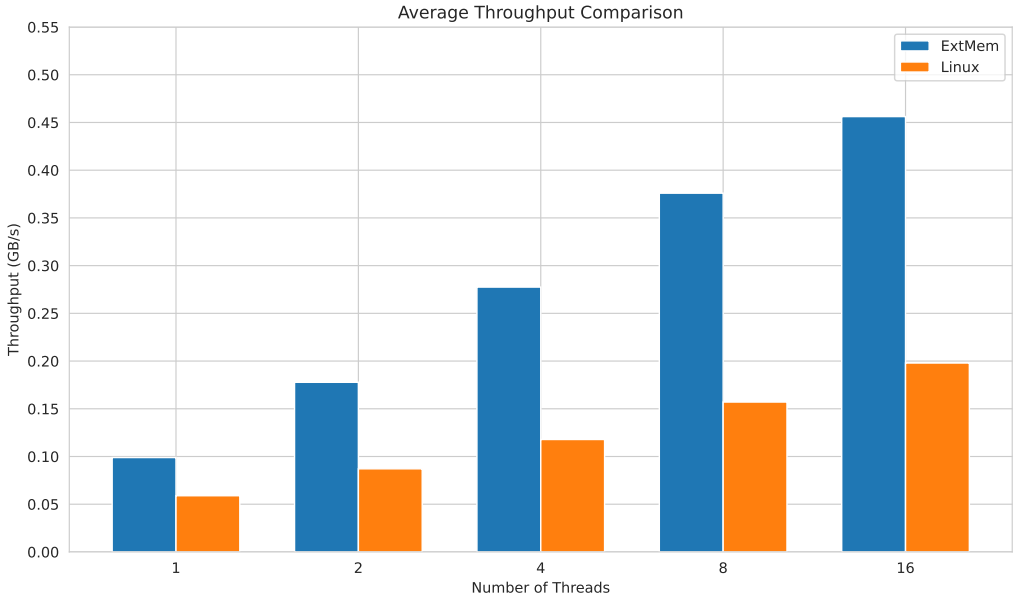


Abbildung 3.1: Vergleich des Durchsatzes bei Schreiben an zufälligen Stellen

In Abbildung 3.1 zu sehen ist der Durchsatz von `mmapbench` in GB/s bei einem 16 GB großem Bereich im virtuellen Speicher. Die Anzahl der schreibenden Threads ist auf der X-Achse angegeben. Zu beachten ist, dass den Prozessen nur 8 GB physischer Speicher und hinreichend viel Swap zur Verfügung steht. Wie in der Abbildung der Einreichung auch, ist der Durchsatz von ExtMem höher.

Ein vergleichbares Ergebnis - dass ExtMem einen höheren Durchsatz hat - konnte ich auch für Schreiben in sequentiellen Bereichen im virtuellen Speicher reproduzieren:

Auch bei Abbildung 3.2 konnte ich den höheren Durchsatz reproduzieren.

Im Originalpaper zeigt sich erst ab vier Threads, dass weitere Threads nicht den Durchsatz erhöhen [68], hier war das bereits ab einem Thread der Fall.

Auffällig ist allerdings, dass die absoluten Zahlen des Durchsatzes deutlich hinter denen im Paper sind. Bei zufälliger Beschreibung des virtuellen Speicherbereichs konnte bei 16 Threads ein Durchsatz von knapp 1,6 GB/s erzielt werden [68], wohingegen ich knapp 0,5 GB/s in Abbildung 3.1 reproduzieren konnte.

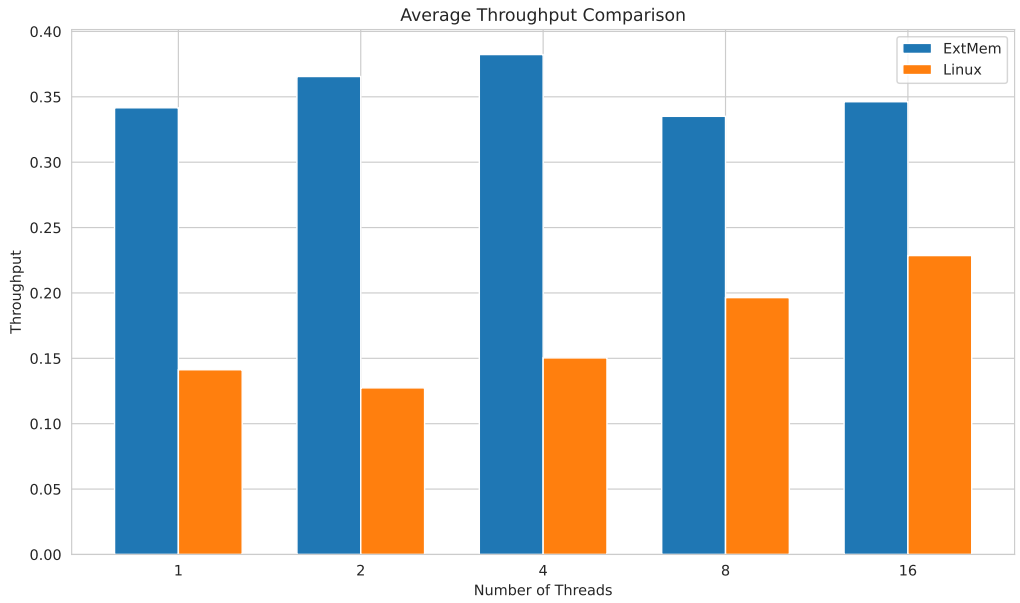


Abbildung 3.2: Vergleich des Durchsatzes bei Schreiben in sequentiellen Bereichen

Die Vermutung liegt nahe, dass die NVMe-SSDs der Testmaschine die Ursache sind: hier wird das Verhalten der Speicherverwaltung bei viel Swapping getestet und entsprechend ist für den Durchsatz auch die Geschwindigkeit der Festplatte relevant. Die o.g. Festplatten der Testmaschine sind beide in die Jahre gekommen: für den Swap wurde eine NVMe aus dem Jahr 2015 verwendet, welche für welche der Hersteller eine maximale Performance von 2500 MB/s für sequentielles Schreiben angibt, sowie für zufälliges Lesen von 4 KB großen Blöcken eine maximale Geschwindigkeit von 460 KB/s.

Als Vergleich habe ich den Basistest für `mmapbench` [11] auf einer Maschine mit 40 GB DDR4 RAM, Intel 11th Gen i7-1165G7 und 20 GB ZRAM Swap [27] ausgeführt. Damit konnte ich bei 16 Threads ca. 2,9 GB/s schreiben, der Peak lag sogar bei 3,9 GB/s. Zu beachten ist, dass die Maschine ein Laptop ist, welcher zu der Zeit nicht nur diese Messung durchführte, sondern u.a. einen Webbrowser und diverse andere Programme offen hatte. Außerdem ist der Durchsatz offensichtlich höher, weil auf dieser Maschine ZRAM Swap benutzt wurde, d.h. der Swap liegt als komprimierte RAM Disk wieder im RAM. Das bedeutet, dass der Flaschenhals von Schreibvorgängen auf eine NVMe SSD hier gar nicht vorhanden ist. Letztendlich spricht dieser Unterschied aber dafür, dass die Differenz auf die eingesetzte Hardware zurückzuführen ist.

CPU Pinning auf den ersten NUMA Knoten, an welchen die ersten 40 Threads und beide Festplatten angeschlossen sind, hat bei der Reproduktion dabei keine nennenswerten Unterschiede bewirkt.

Ein dritter, auf `mmapbench` [11] basierender Test schreibt an zufälligen Stellen, allerdings zu 90% der Zeit in 10% des virtuellen Speicherbereichs:

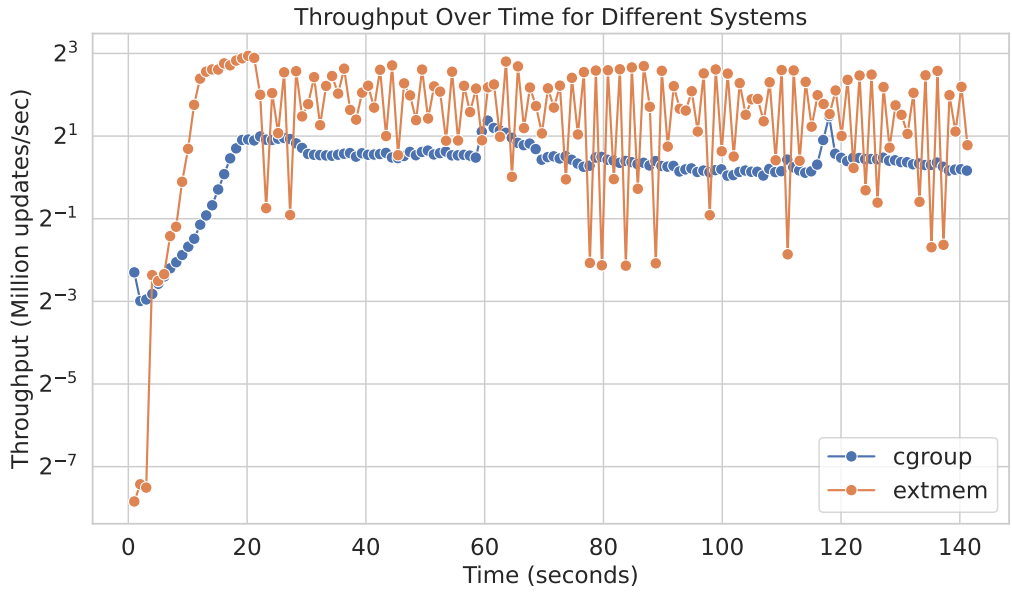


Abbildung 3.3: Messung des Durchsatzes bei zufälligen Beschreiben nah aneinanderliegender Bereiche.

Hier wurde über Zeit geplottet, wobei mit durchgehend acht Threads geschrieben wurde. Der relative Unterschied aus der Einreichung ist auch hier nachvollziehbar, wobei ExtMem bis zu doppelt so viel Durchsatz hat, wie in der Einreichung [68].

Dass die Spitze hier bei 8 GB/s liegt statt 16 GB/s führe ich ebenfalls auf Unterschiedliche Hardware zurück. Zu beachten ist allerdings, dass ich das Skript zur Erzeugung dieses Diagramms gepatcht habe: der Durchsatz von `mmapbench` [11] wird in der Einreichung in GB/s angegeben [68]. Allerdings ist die Y-Achse im reproduzierten Graph mit "Million Updates/sec"beschriftet. Der Code führt dabei folgende Berechnung durch:

$$\text{throughput} \cdot \frac{1024^3}{4096 \cdot 1000 \cdot 1000}$$

Damit sollte vermutl. der Durchsatz in Seiten pro Sekunde dargestellt werden. Nachdem die Einreichung in GB/s angegeben ist, habe ich entschieden, die Umwandlung zu entfernen.

Für die Evaluation des in Abschnitt 3.1.1 beschriebenen PageRank-Verfahrens gab es kein Skript zur Erzeugung eines Graphen. Getestet wurde mit GAP [32], wie schnell das Verfahren ohne Begrenzung des physischen Speichers; mit Begrenzung des physischen Speichers auf 8 GB; mit derselben Begrenzung, aber mit ExtMem; sowie mit derselben Begrenzung aber der für Graphverarbeitung optimierten dynamischen Bibliothek [68].

| | in Memory | Linux | ExtMem | ExtMem, optimiert |
|---|---|---|---|---|
| Reproduktion | 27,9s | 526,6s | 153,5s | 127s |
| Paper | ca. 30s | ca. 190s | ca. 180s | ca. 80s |

Abbildung 3.4: Ergebnisse der Reproduktion

**Anmerkung:** die Daten aus der Einreichung waren nur in Form eines Graphen verfügbar, weshalb ich diese abgelesen habe. Daher sind diese Werte mit „ca.“markiert.

Das Verfahren wurde hier zehn Mal durchgeführt, o.g. Wert ist die durchschnittliche Ausführungszeit. Die Dauer zum Einlesen der Graphdaten ist nicht Teil der Messung.

Auffällig hierbei ist, dass der Basistest ohne Begrenzung am nächsten an den Zahlen der Einreichung dran ist, was ein weiterer Grund dafür ist, dass der Unterschied in den absoluten Zahlen mit den unterschiedlich schnellen NVMe-SSDs zu erklären ist. Die Ergebnisse selbst sind der Einreichung ähnlich: ExtMem ist schneller als die virtuelle Speicherverwaltung des Linux-Kernels und die optimierte Implementierung auf Basis von ExtMem ist noch schneller. Allerdings ist das Performancegefälle zwischen Standard-ExtMem und dem Linux-Kernel wesentlich größer in der Reproduktion.

Hierbei zu erwähnen ist, dass die Ausführung dieses Benchmarks einen Patch in ExtMem erforderte:

```
--- a/src/storage_iouring.c
+++ b/src/storage_iouring.c
[...]
 void ioring_submit_all_reads(){
+    if(uring_read_init == false){
+        thread_init_read_iouring();
+        uring_read_init = true;
+    }
     if (io_uring_submit(&io_read_ring) < 0) {
        fprintf(stderr, "io_uring_submit");
        perror("io_uring_submit");
```

Nach Einlesen der Graphdaten schlug das Benchmark zuverlässig mit einem Segmentierfehler fehl: dieses Problem war die Motivation für das Kompilieren aller Programme mit Debugsymbolen, wie in Abschnitt 3.1.1 beschrieben. Der Coredump zeigte, dass der Fehler in einem Thread auftrat, wo `uring_read_init == false` war und entsprechend die Struktur unter `io_ring_ring` nicht initialisiert war, was den Segmentierfehler in `liburing` verursacht hat.

Warum das bei der Evaluation der Autoren nicht aufgefallen ist, kann ich nicht beantworten, möglicherweise tritt das Problem nur bei hoher Parallelität auf. Nachdem der Change das Problem behoben hat, alle Benchmarks problemlos damit liefen und die `mmapbench` [11] Benchmarks vergleichbare Zahlen lieferten, habe ich entschieden, mit diesem Patch weiterzuarbeiten.

### 3.1.3 Fazit

Den verbesserten Durchsatz verglichen mit dem virtuellen Speichermanagement des Linux-Kernels konnte ich in dieser Arbeit nachweisen. Die absoluten Zahlen wichen zwar spürbar von denen in der Einreichung ab, allerdings zeigen die Basistests ohne ExtMem, dass der Zuwachs des Durchsatzes auch bei weniger leistungsfähiger Hardware nachweisbar ist. Des weiteren weicht die konkret eingesetzte Hardware stark von der ab, die in der Einreichung verwendet wurde.

Die größten technischen Herausforderungen waren die Untersuchung des in Abschnitt 3.1.2 beschriebenen Segmentierfehlers und die in Abschnitt 3.1.1 beschriebenen Probleme bei der Paketierung des Linux-Kernels. Letzteres ist damit zu erklären, dass ich ein mehrere Jahre altes Patchlevel des Linux-Kernels mit heutigen Werkzeugen gebaut habe. Wenn man ähnlich wie in Abschnitt 3.1.1 beschrieben ganz `nixpkgs` [80] zu dem Zeitpunkt der Entwicklung eingefroren hätte, wäre aufgrund der Funktionsweise & Garantien von Nix der Kernel direkt baubar gewesen.

Der Nachteil von diesem Ansatz ist, dass mehrere Jahre an Sicherheitsupdates fehlen. Solange es nur um Evaluationsumgebungen, die nicht direkt aus dem Internet erreichbar sind, geht und nicht um Produktivsysteme, ist das aus meiner Sicht vernachlässigbar und in gewisser Weise auch genau so gewollt, weil abweichende Softwareversionen ein Faktor sind, der die Ergebnisse verfälschen könnte.

Damit zeigt sich also, dass die Wiederholbarkeit von NixOS in diesem Kontext auch eine wertvolle Eigenschaft ist: im Vergleich zu den Anweisungen im Coderepository von ExtMem muss man nicht selbst prüfen, ob die Kernel-Konfiguration stimmt und die richtigen Bibliotheken bzw. die richtige Version der Distribution installiert ist [67], sondern kann all diese Anforde-

rungen in Nix-Code formulieren, daraus ein NixOS-System erzeugen - siehe Abschnitt 3.3 dafür und dies bei Bedarf regelmäßig in einer CI-Umgebung testen.

Einzelne Pakete können auch außerhalb der Evaluationsumgebung benutzt werden, sofern Nix installiert ist: dabei gibt es im Userspace keine nennenswerten Abhängigkeiten zu den systemweit installierten Bibliotheken, da Nix Abhängigkeiten aus seinem Closure bezieht. Das war hilfreich, um `mmapbench` [11] mit den exakt gleichen Abhängigkeiten im Userspace auf einer anderen Maschine laufen zu lassen, um den Durchsatz mit schnellerem Swapspace zu messen, wie in Abschnitt 3.1.1 beschrieben.

## 3.2 oBBR

"oBBR: Optimize Retransmissions of BBR Flows on the Internet" [34] wurde auf der USENIX ATC2023 eingereicht.

Bi et al beschäftigen sich mit BBR [39], einem von Google entwickeltem Verfahren zur Staukontrolle von TCP-Verbindungen, sowie den Schwachstellen davon und stellen Möglichkeiten vor, diese zu verbessern.

Im Vergleich zu anderen Verfahren zur Staukontrolle sieht BBR Paketverlust nicht als Indikator für eine Stauung an [34]. Stattdessen wird die Durchsatzrate anhand der geschätzten Bandbreite und gemessenen Paketumlaufzeit bestimmt. Erstere wird bei einer neuen Übertragung initial ermittelt, indem sukzessive die Übertragungsrate erhöht wird (*Startup*). Sobald die Bandbreite nicht mehr mitwächst, wird die Übertragungsrate reduziert, um die Puffer auf den Routern zu entlasten (*Drain*). Daraufhin wird wiederholt die Übertragungsrate erhöht und verkleinert, um zu prüfen, ob die Bandbreite mitwächst und ggf. die Übertragungsrate dauerhaft erhöht werden kann (*ProbeBW*). Falls sich die Messergebnisse zur Paketumlaufzeit für ca. zehn Sekunden nicht verändern, werden genau vier Pakete gesendet und auf die Antworten gewartet, um die Umlaufzeit neu zu schätzen (*ProbeRTT*) [34].

Dieser Algorithmus hat den Nachteil, dass damit Übertragungen mit anderen, Paketverlust-basierten Verfahren zur Staukontrolle ausgestochen werden [62]. Dies wurde in BBRv2 [40], welches auf Kosten des Durchsatzes Paketverlust wieder als Indikator für Stauung ansieht, geändert.

In dieser Einreichung werden zwei Faktoren diskutiert, wodurch viele Neuübertragungen die Folge sind: zum einen wird die Bandbreite meist als zu hoch eingeschätzt [34], wodurch Pakete bei der Übertragung gepuffert werden müssen. Sollten die Puffer der Router auf der Übertragungsstrecke sehr klein sein, hat das Paketverlust und Neuübertragungen zur Folge [34].

Die geschätzte Bandbreite wird ermittelt, indem die maximale gemessene

Bandbreite in einem Zeitfenster von 10 Paketumläufen ermittelt wird [34]. Als Ausgleich zu diesem Maximum wird die kleinste gemessene Umlaufzeit gewählt. Wenn nun während einer Übertragung die Bandbreite sich verringert, müssen 10 Umläufe geschehen, bevor die Messungen zur neuen, veringerten Bandbreite überhaupt in die Berechnung der Übertragungsrate mit einfließen [34]. Dies hat zur Folge, dass mehr Pakete übertragen werden, als die Übertragungsstrecke verträgt, wodurch es zur Stauung und bei vollen Puffern zu Paketverlust mit Neuübertragungen kommt [34].

Das modifizierte Verfahren, genannt oBBR, adressiert diese Schwachstellen: sobald $k$ aufeinanderfolgende Messungen der Bandbreite geringer sind als 75% der aktuellen Bandbreitenschätzung, wird eine ein Abfall der Bandbreite angenommen [34]. In Testversuchen konnten zu keinem Zeitpunkt mehr als 22 aufeinanderfolgende Messergebnisse unterhalb von 75% der aktuellen Bandbreite bei konstanter Bandbreite gemessen werden [34]. Je größer $k$ bei dieser Abschätzung, desto sicherer ist ein Abfall der Bandbreite. Die Übertragungsrate wird bei Paketverlust verringert, in Abhängigkeit der Schätzung, wie ausgelastet die Puffer auf der Übertragungsstrecke sind [34].

Damit soll der Durchsatz von oBBR im Vergleich zu BBRv2 erhöht bzw. die Menge an Neuübertragungen wesentlich veringert werden [34].

### 3.2.1 Paketierte Komponenten

Ziel dieser Arbeit ist es, die Eignung von NixOS für Evaluationsumgebungen zu zeigen und hierfür die Unterstützung für Zustandsverwaltung zu verbessern. Die Reproduktion von Einreichungen mit Bordmitteln ist nur ein kleiner Teil davon, weshalb ich nicht alle Artefakte reproduziert habe. Da alle Tests mit denselben Bausteinen arbeiten, erwarte ich hier keinen weiteren, großen Erkenntnisgewinn.

Konkret werden folgende Programme für die Messungen von BBR und oBBR benötigt [33]:

- `quic_client` [47], ein Programm im Coderepository von Chromium zum Testen von QUIC [72] Verbindungen.

  QUIC ist ein auf UDP basierendes Transportprotokoll, welches intern mit sog. Streams arbeitet, durch welche das Problem der sog. HOL (*Head of line*) Blockierung vermieden wird [72]. Da auch dieses Protokoll Staukontrolle benötigt [72] und Anbieter, wie Youtube und Spotify bereits QUIC unterstützen [34], wird hier BBR bzw. oBBR im Kontext von QUIC statt TCP evaluiert [33].

- `nginx-quic` [33], eine modifizierte Variante von nginx [96], welche BBR

und oBBR über QUIC [72] unterstützt und Statistiken zur Übertragungsrate mitschneidet.

- Diverse Testscripte zur Reproduktion der Artefakte.

Ich habe damit die Artefakte reproduziert, die die Reaktion von BBR auf ein Absinken der Bandbreite visualisieren.

Sowohl `quic_client`, als auch `nginx-quic` stammen aus Projekten, welche bereits in `nixpkgs` [80] paketiert sind, welche zu diesem Zweck angepasst werden konnten. Die Anpassung für `nginx-quic` sieht hierbei wie folgt aus:

```
self: super: { obbr = {
  src =
    /* Tarball des Quellcodes für oBBR */

  nginxQuic = super.nginxQuic.overrideAttrs ({
    buildInputs ? [],
    configureFlags ? [],
    ...
  }: {
    src = self.obbr.src;
    sourceRoot = "source/nginx-quic";
    buildInputs =
      lib.filter (
        pkg: ! builtins.elem
          (lib.getName pkg)
          [ "openssl" "quictls" ]
      ) buildInputs
      ++ [
        self.obbr.boringssl
        (self.zlib-ng.override { withZlibCompat = true; })
      ];
    configureScript = "auto/configure";
    configureFlags = configureFlags ++ [
      "--with-cc-opt=-I${lib.getDev self.obbr.boringssl}/include"
      "--with-ld-opt=-L${lib.getLib self.obbr.boringssl}/lib"
    ];
  });
}; }
```

Die Einreichung benötigt eine modifizierte Variante von `boringssl` [33, 55], welches notwendig für die Unterstützung von QUIC ist [33]. Die Paketde-

finition für `boringssl` habe ich hier nicht abgebildet, weil diese hinreichend trivial ist.

In `nixpkgs` [80] wird `nginx` [96] normalerweise mit anderen Bibliotheken für TLS kompiliert. Um zu vermeiden, dass diese statt `boringssl` [55] verwendet werden, werden diese aus den Abhängigkeiten in Zeile 12 herausgefiltert. In Zeile 23 werden die Parameter zur Konfiguration des Buildvorgangs so angepasst, sodass die Header von `boringssl` [55] gefunden werden und dagegen gelinkt wird.

Eine größere Herausforderung war das Paket für `quic_client`: dies ist im Coderepository von Chromium enthalten, welches sehr groß ist und durch ein komplexes Buildsystem konfiguriert wird [46]. Die ursprüngliche Idee, hier das bereitgestellte Kompilat [33] zu benutzen, habe ich aus zwei Gründen verworfen: zum einen würde es der Idee wiedersprechen, NixOS zum bauen von Evaluationsumgebungen zu benutzen, wenn vor diesem Vorgang noch ein weiterer Buildprozess gebaut werden muss. Zum anderen müsste der ELF-Header modifiziert werden, da auf NixOS keine Bibliotheken bzw. der Interpreter unter `/usr/lib` existieren. Grundsätzlich ist dies war mit einem Programm namens `patchelf` [52] zwar möglich, allerdings schlug das bei dem mitgelieferten `quic_client` fehl, weil es Teile des ELF Headers nicht lesen konnte. Dies habe ich nicht weiter untersucht, sondern stattdessen mit Hilfe des Pakets für Chromium in `nixpkgs` [80] das Programm frisch kompiliert.

Die Derivation hierfür sieht wie folgt aus:

```
1  self: super: { obbr = {
2    quic_client = self.callPackage
3      ({ chromium }: chromium.mkDerivation (base: {
4        name = "quic_client";
5        packageName = "quic_client";
6        postPatch = base.postPatch + ''
7          pushd third_party/test_fonts &>/dev/null
8            mkdir test_fonts
9            pushd test_fonts &>/dev/null
10             touch ./DejaVuSans.ttf
11             # und noch mehr `touch Font.ttf` Aufrufe.
12           popd &>/dev/null
13         popd &>/dev/null
14       '';
15       buildTargets = [
16         "quic_client"
17       ];
18       installPhase = ''
```

```
19          runHook preInstall
20          mkdir -p $out/bin
21          cp out/Release/quic_client $out/bin/
22          runHook postInstall
23        '';
24        postFixup = "";
25      })) {};
26  }; }
```

An dieser Stelle bietet das Chromium Paket in `nixpkgs` [80] eine eigene API an, um im Coderepository von Chromium Buildvorgänge laufen zu lassen.

Problematisch war, dass bei dem Versuch, `quic_client` [47] so zu bauen, wie in der Dokumentation [47] beschrieben, es folgenden Fehler gab, zu welchem ich bei der darauffolgenden Recherche keine weiteren Informationen finden konnte:

```
ninja: error: '../../third_party/test_fonts/test_fonts/Ahem.ttf',
needed by 'test_fonts/Ahem.ttf', missing and no known rule to
make it
```

Aufgrund der Größe des Baumes mit Bauschritten war die Ursache davon nicht abzulesen. Beim Bauen von Zielen für Chromium selbst trat der Fehler nicht auf. Unter der Annahme, dass für den `quic_client` [47] selbst keine Schriftarten gebraucht werden, habe ich diese Dateien lediglich angelegt, siehe Zeile 6, was das Problem behoben hat.

Die `installPhase` ab Zeile 18 im Originalpaket war darauf ausgelegt, `chromium` in den Storepfad zu installieren. Daher habe ich diese komplett überschrieben, sodass nur `quic_client` [47] installiert wird.

Wie vorher in Abschnitt 3.1.1 beschrieben, ist die Zustandsverwaltung in NixOS noch nicht ausgereift. Hier wird diese für sog. Verkehrsformung (*Traffic shaping*) benötigt. Konkret soll für die Beobachtung, wie sich BBR bei einem Absinken der Bandbreite verhält, der Durchsatz für den involvierten Netzwerkadapter zuerst auf einen festen Wert gesetzt und dann abgesenkt werden [33]. Umgesetzt wird dies unter Linux mit `tc(8)` [23], mit welchem entsprechende Einstellungen im Netzsubsystem des Linux kernels vorgenommen werden können.

Ursprünglich war die Idee, mit `systemd.network(5)` dies zu konfigurieren:

```
1  [NetworkEmulator]
2  DelaySec=50msec
```

```
3  Handle=1
4  Parent=root
5
6  [TokenBucketFilter]
7  BurstBytes=100K
8  Handle=10
9  LimitBytes=500K
10 Parent=1:1 ; parent is the network emulator
11 Rate=20M
```

Dies hätte die initiale Verkehrsformung für den lokalen Netzwerkadapter eingerichtet und die Benchmarks hätten bei Bedarf die Werte ändern können. Allerdings war dies aufgrund von zwei Problemen nicht möglich: zum einen erkannte `systemd.network(5)` die Deklaration in Zeile 10 nicht als gültige Elternbezeichnung an, zum anderen war `systemd.network(5)` an dieser Stelle nicht konvergent: falls die Konfiguration hiervon geändert und dies aktiviert wurde, wurden die Veränderungen nicht konfiguriert, sofern bereits Verkehrsformung vorhanden war.

Diese Problematik ist damit eine weitere Motivation für bessere Zustandsverwaltung in NixOS. Um trotzdem Ergebnisse zu erhalten, habe ich beschlossen, wieder den gesamten Evaluationsvorgang in einem Pythonprogramm abzubilden, mit denselben Motivationen wie in Abschnitt 3.1.1. Konkret sieht der Kontextmanager für die Verkehrsformung wie folgt aus:

```
1  from contextlib import contextmanager
2
3  @contextmanager
4  def lo_setup():
5      # führe folgende Befehle aus:
6      # tc qdisc add dev lo root handle 1:0 netem delay 50ms
7      # sudo tc qdisc add dev lo parent 1:1 handle 10: tbf \
8      #  rate 20Mbit burst 100KB limit 500KB
9      yield
10     # tc qdisc del dev lo root
```

Für einzelne Evaluationen wurde dann in einem definierten Abstand die Bandbreite zwischen zwei Werten gewechselt:

```
1  @contextmanager
2  def shape_lo(
3      rate1: int, rate2: int, limit: int, sleep_s: int, burst: int
```

```
4  ):
5      def set_rate(rate: int):
6          # tc qdisc change dev lo parent 1:1 handle 10: \
7          #   tbf rate {rate}Mbit burst {burst}KB limit {limit}KB
8
9      pid = os.fork()
10     if pid == 0:
11         while True:
12             try:
13                 set_rate(rate1)
14                 time.sleep(sleep_s)
15                 set_rate(rate2)
16                 time.sleep(sleep_s)
17             except KeyboardInterrupt:
18                 sys.exit(0)
19     else:
20         try:
21             yield
22         except Exception as e:
23             print(e)
24         finally:
25             os.kill(pid, SIGINT)
26
27 with shape_lo(...):
28     # lasse tests laufen
```

Hier wird der Prozess geteilt und der Kindsprozess wechselt alle `sleep_s` Sekunden die Bandbreite zwischen zwei Werten, bis die Tests fertig sind und der Kontextmanager verlassen wird.

Der `exit(0)` in Zeile 18 ist notwendig, da sonst auch im Kindsprozess der Kontextmanager verlassen werden würde und die Tests laufen gelassen würden bzw. aufgrund des fehlenden `yield` in diesem Pfad ein Laufzeitfehler geworfen werden würde.

Innerhalb der Tests wird ein `nginx` [96] gestartet, welcher QUIC [72] mit BBR [39] für die Staukontrolle benutzt. Zur Messung des Durchsatzes wird eine 10 GB große Datei bereitgestellt und von dem `quic_client` [47] angefragt.

### 3.2.2 Ergebnisse

Die Evaluationen wurden auf derselben Hardware, wie in Abschnitt 3.1.2 durchgeführt. Die Einreichung hat mit wesentlich geringer ausgestatteter Hardware gearbeitet [34]. Nachdem nur über den lokalen Netzwerkadapter kommuniziert wird und die Bandbreite auf maximal 20 Mb/s gedeckelt wird, erwarte ich bei heutiger Hardware keine großen Abweichungen.

Abbildung 4 aus der Einreichung zeigt Paketumlaufzeit, -verlust und die Menge an aktuell übertragenen Paketen bei einer Bandbreite von konstant 20 Mb/s, sowie einem Abfall auf fünf bzw. zwei Mb/s bei ca. zehn Sekunden. Die Beobachtungen waren reproduzierbar, wie Abbildung 3.5 zu entnehmen ist.

Abbildung 3.5 zeigt, wie in der Einreichung, dass ab dem Abfall der Bandbreite es Paketverlust gibt, aber die Übertragungsrate - charakteristisch für BBR [39] - nicht absinkt, weil aus den letzten zehn Messungen der Bandbreite das Maximum gewählt wird [34, 39]. Erst nachdem es keine solchen Messungen mehr gibt, fällt die Menge der sich in Übertragung befindlichen Pakete (*Inflight*). Dies geschieht nach Ablauf von zehn Paketumläufen, ist also abhängig davon, wie sehr die Umlaufzeit in Folge des Abfalls der Bandbreite ansteigt. In der Einreichungen waren es ca. 700 ms [34], hier ein wenig mehr, wodurch der die zehn Umläufe erst nach ca. 8 bis 9 Sekunden erfolgt sind.

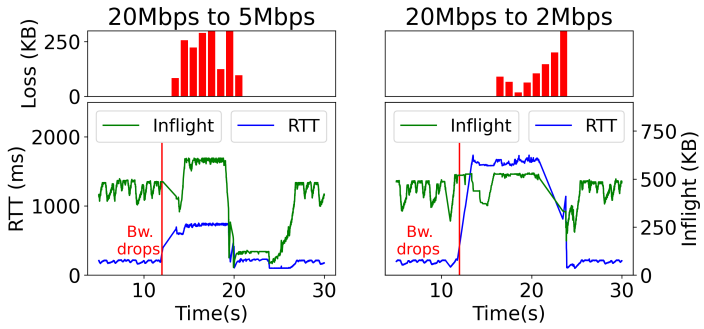


Abbildung 3.5: Abfall der Bandbreite bei BBR von 20 Mb/s auf 5 bzw. 2 Mb/s

Auffällig ist der rasante Anstieg der im sich im Umlauf befindlichen Pakete ab ca. 25 Sekunden: dies liegt daran, dass das Programm darauf ausgelegt ist, alle 12 Sekunden die Bandbreite zu verändern, von 20 Mb/s nach fünf bzw. zwei Mb/s usw.. Dies ist auch im Coderepository so implementiert (siehe `nginx-quic/scripts/nets/change.sh`) [33]. Aus der Historie ist nicht nachvollziehbar, dass dies verändert wurde, das Verhalten passt aber exakt zu den Daten: die Menge der im Umlauf befindlichen Paketen steigt stark, ohne Paketverlust.

Auf der rechten Seite ist der Abfall der in Umlauf befindlichen Pakete kaum ablesbar. Dies liegt daran, dass die so stark anstieg, dass zu dem Zeitpunkt, zu welchem die Pakete im Umlauf nach unten korrigiert wurden, die Bandbreite wieder erhöht wurde. Ich könnte mir vorstellen, dass zur Visualisierung des Abfalls auf der rechten Seite die Bandbreite nicht mehr nach oben gesetzt und diese Änderung nicht mehr eingereicht wurde.

Abbildung 5 der Einreichung besteht aus drei Einzelabbildungen: links die kumulierte Verteilung aller Messungen der ermittelten Bandbreite von BBR bei konstant 20Mb/s Bandbreite und 100ms Paketumlaufzeit, normalisiert zur tatsächlichen Bandbreite 20Mb/s. Die Kurve sieht der, der Einreichung ähnlich und auch hier kann man ablesen, dass mehr als 50% aller Messungen unter 95% der tatsächlichen Bandbreite liegen.

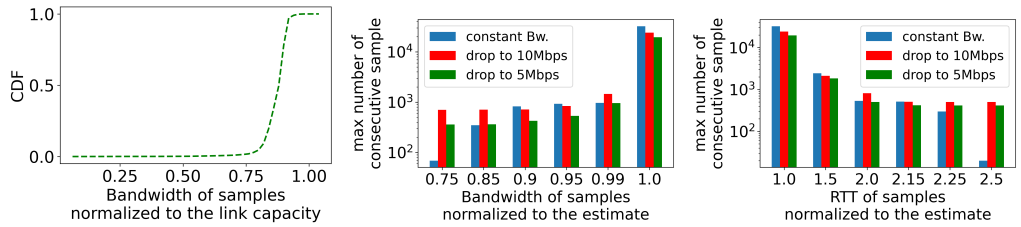


Abbildung 3.6: Normalisierte Bandbreiten- und Umlaufzeitmessungen

Damit soll gezeigt werden, dass Messungen von geringerer Bandbreite nicht zwingend einen tatsächlichen Abfall bedeuten [34], was hier nachvollziehbar ist.

Die mittige Abbildung soll zeigen, dass abweichende Messungen der Bandbreite selten aufeinanderfolgend sind: hier ist die größte Folge, aufeinander folgender und abweichender Messungen der Bandbreite gezeigt. Die Messungen sind auf der X-Achse danach gruppiert, um welchen Faktor sie sich von der tatsächlichen Bandbreite unterscheiden.

Bei konstanter Bandbreite wird angegeben, dass maximal 22 aufeinanderfolgende Messungen unter 75% der verfügbaren Bandbreite ergeben [34]. Die Größenordnung kann hier ebenfalls abgelesen werden. Die Größenordnung von $10^2 < k < 10^3$ für abweichende Messungen bei einem Abfall der Bandbreite sind ebenfalls ablesbar.

Analog wird rechts, gruppiert nach Faktor, gezeigt, wieviele aufeinanderfolgende Messungen der Paketumlaufzeit die tatsächliche Umlaufzeit überschreiten. Die Größenordnungen, die in der Einreichung angegeben sind [34], sind auch hier nachvollziehbar. Insbesondere die Beobachtung, dass bei konstanter Bandbreite max. 15 aufeinanderfolgende Messungen der Umlaufzeit die tatsächliche Umlaufzeit um das 2,5-fache überschreiten.

Wie in Abschnitt 3.2.1 beschrieben, sollen anhand dieser Messungen Heuristiken abgeleitet werden, mit welchen bereits in unter zehn Sekunden auf Veränderungen der Bandbreite reagiert werden sollen [34].

### 3.2.3 Fazit

Die wichtigsten Aussagen der Abbildungen vier und fünf der Einreichung [34] waren mit dem Code zur Reproduktion nachvollziehbar. Die einzige Abweichung ist bei Abbildung vier [34] bzw. in dieser Arbeit Abbildung 3.5, bei welcher die Übertragungsrate nach 24 Sekunden wieder angehoben wird. Allerdings ist das Verhalten, wie in Abschnitt 3.2.2 direkt aus dem Coderepository der Einreichung [33] ablesbar.

Eine weitere Erkenntnis ist, dass `nixpkgs` [80] nicht nur für Abhängigkeiten von Evaluationssoftware benutzt werden kann, sondern auch als Basis für die Software selbst dienen kann. So konnten die Definitionen für `nginx` [96] und `quic_client` [47] weiterverwendet werden und benötigten nur wenige Anpassungen. In Hinblick auf die Komplexität des Buildsystems in Chromium war die Existenz eines Pakets samt Schnittstelle zur Erweiterung dafür sehr hilfreich.

Leider hat sich auch gezeigt, dass die Unterstützung für Verkehrsformung in `systemd.network(5)` [20] in Hinblick auf Konvergenz nicht gut funktioniert und dies ebenfalls Funktionalität ist, die von besserem Zustandsmanagement profitieren würde.

## 3.3 Bereitstellung von NixOS Installationen

NixOS bietet fertige Funktionalität an, um in einem Build Artefakte, wie z.B. ein ISO oder ein `qcow2`-Image - einem Abbildformat von QEMU - zu erzeugen. Mit den Bordmitteln wäre es damit möglich, installierbare Evaluationsumgebungen anzubieten, welche z.B. von Prüfern selbst reproduziert werden kann.

Für obige Arbeiten habe ich das nicht genutzt, weil es sich für iterative Arbeit anbietet, auf einem fertigen NixOS System zu arbeiten und dort Schritt für Schritt die Umgebungen vorzubereiten. Zu Demonstrationszwecken, habe ich Builds für derartige Zwecke implementiert:

```
1 { pkgs ? import <nixpkgs> {}, lib ? pkgs.lib }:
2
3 {
4   qcow2 = import (pkgs.path + "/nixos/lib/make-disk-image.nix") {
```

```
5       inherit lib pkgs;
6       inherit (pkgs.nixos {
7         imports = [
8           ./evaluation-environment.nix
9           (modulesPath + "/profiles/qemu-guest.nix")
10        ];
11        networking.hostName = "evaluation-environment";
12        fileSystems."/" = {
13          fsType = "ext4";
14          device = "/dev/disk/by-label/nixos";
15          options = ["discard"];
16          autoResize = true;
17        };
18        fileSystems."/boot" = {
19          device = "/dev/disk/by-label/ESP";
20          fsType = "vfat";
21        };
22        boot.growPartition = true;
23        boot.loader.systemd-boot.enable = true;
24      }) config;
25      format = "qcow2";
26      partitionTableType = "efi";
27      copyChannel = false;
28      diskSize = 32768;
29    };
30  }
```

Mit diesem Ausdruck wird ein `qcow2` Artefakt erzeugt, also ein Abbild einer VM für den QEMU Hypervisor [16] für eine NixOS-Konfiguration. Die für die Umgebung relevanten Einstellungen können hier in der Datei `evaluation-environment.nix` vorgenommen werden, die in Zeile 8 importiert wird.

Die Konfigurationsparameter ab Zeile 12 dienen dazu, ein starttaugliches System zu erzeugen. So wird eine EFI Partition angelegt, sowie eine Hauptpartition für das System. Standartmäßig ist das Abbild 32 GB groß, kann aber beim hochfahren auch automatisch vergrößert werden.

Neben Unterstützung hierfür bietet `nixpkgs` [80] auch noch Builder für andere Formate, beispielsweise ISO-9660 Abbilder oder AMIs für die Amazon Cloud [77]. Damit ergibt sich ein Buildsystem, mit welchem die gesamte Software deklarativ beschrieben und gebaut werden kann, inkl. startbarer Umgebungen für Virtualisierungsumgebungen.

Die Erzeugung solcher virtueller Umgebungen lässt sich in vielen Fällen mit Provisionierungswerkzeugen, wie OpenTofu [101] erledigen. Eine weitere Möglichkeit ist `nixos-anywhere` [88]: das ist ein Werkzeug, welches deklarative Konfiguration von Festplattenlayouts mit dem Installer von NixOS vereint und es damit erlaubt, auch Hardware NixOS Installationen zu automatisieren.

Es zeigt sich, dass es hinreichend viele Lösungen gibt, um NixOS zu installieren. Die weiteren Abschnitte beschränken sich bzgl. Zustandsverwaltung auf laufende Systeme bzw. die Interaktion zwischen laufenden Systemen.

# Kapitel 4

# Entwurf von Smash

Im vorherigen Kapitel habe ich gezeigt, dass NixOS die Mittel bietet, um wiederholbar das gesamte System, d.h. Pakete und Konfiguration für verschiedene Ökosysteme als Code darzustellen, zu bauen und auszurollen. Dies ist bereits eine Verbesserung zum Status Quo für die Bereitstellung von Evaluationsumgebungen, bei denen penibel sichergestellt werden muss, dass die korrekte Software und Konfiguration benutzt wird.

## 4.1 Anforderungen

Allerdings gibt es einen Aspekt, den NixOS heutzutage nicht gut behandelt: die Verwaltung des Zustandes bzw. einiger Aspekte von diesem. Bei der Analyse der Anforderungen von Artefakten diverser Einreichungen [44, 58, 74] ergaben sich folgende Anforderungen:

**Provisionierung** von Daten, die als Konfiguration definiert sind. Beispielsweise das automatische Schemaupgrade einer Datenbank, das Anlegen von Zugängen für einen Speicherdienst – z.B. MinIO [74] – oder die Konfiguration einer Messengerqueue, z.B. das Anlegen von *Topics* in Kafka [58].

Hierbei müssen auch Änderungen, beispielsweise andere Parameter für das Kafka-Topic, berücksichtigt werden. Endgültige Löschung darf nicht automatisch Erfolgen, stattdessen muss entweder bestätigt werden oder die Daten vor Entfernung archiviert werden. Dies ist konsistent mit dem aktuell erwarteten Verhalten von NixOS, nicht von sich aus destruktive Operationen durchzuführen. Beispielsweise wird das Home-Verzeichnis eines Benutzers nicht gelöscht, wenn der Benutzer aus der Konfiguration entfernt wird. Es wird lediglich sichergestellt, dass kein anderer Benutzer diese UID zugewiesen bekommt.

**Behauptungen** (engl. *Assertions*) sind ein Weg, um zu Prüfen, ob bestimmte Eigenschaften des Systemzustandes gültig sind. Beispiele dafür sind Anforderungen an Hard- oder Firmware, wie in Abschnitt 2.2 beschrieben.

Falls eine solche Prüfung fehlschlägt, muss die Aktivierung der Konfiguration verweigert werden. Es muss eine klare Fehlermeldung geben, welche den Kern des Problems erläutert und idealerweise Wege, um das Problem zu lösen.

Im Falle des Beispiels von Hyperthreading hat das den Vorteil, explizit sichergestellt ist, dass die Anforderung erfüllt ist, statt das den Prüfern zu überlassen. In letzterem Fall würde der Fehler schlimmstenfalls erst bemerkt, wenn grobe Abweichungen oder zu starke Varianz der Resultate vorliegt bei z.B. Hardwarelastigen Evaluationen, wie z.B. der Prüfung moderner Isolationsmechanismen [45].

**Zurückrollen** ist etwas, das NixOS von Haus aus für Konfiguration unterstützt: sowohl im Bootmenü kann auf eine alte Generation zurückgerollt werden und somit die alte Konfiguration, Kernel usw. aktiviert werden.

Selbst für die beschriebenen Möglichkeiten, wie NixOS Zustandsverwaltung unterstützt (vgl. Abschnitt 2.3) ist dies allerdings nur spärlich umgesetzt. Sofern ein Schritt der Aktivierung, die Zustandsverwaltung betrifft, fehlschlägt, muss direkt auf die vorherige Generation zurückgerollt werden, inkl. eines Zurückrollens aller geänderten Aspekte des Systemzustandes.

Diese Anforderung geht auf zwei Probleme zurück:

- Wenn z.B. ein Teilaspekt des Zustandes einer Datenbank verwaltet wird – z.B. das Schemaupgrade – dann kann im Fehlerfall nicht mehr zwingend zu einem späteren Zeitpunkt ein altes Backup eingespielt werden, weil ggf. neue Benutzerdaten in der Datenbank gelandet sind.

- Teilweise ist das Deployen einer älteren Version der Software nicht mehr möglich, weil die Software keine Downgrades unterstützt.

Eine mögliche Umsetzung davon könnte mit dem Einsatz von Snapshot des Dateisystems erfolgen. Dies wird u.a. von ZFS [5] unterstützt.

**Reaktivierung** beschreibt die Möglichkeit, Zustände temporär zu entfernen. Ein Beispiel ist das in Abschnitt 2.3 beschriebene Verhalten, dass NixOS einmal benutze UIDs nicht noch einmal zuweist. Außerdem erhält der Benutzer, wenn dieser vorübergehend entfernt wurde, die vorherige UID zurück, selbst wenn diese nicht Teil der Konfiguration war, damit sichergestellt ist, dass nur dieser Benutzer Zugriff auf sein Homeverzeichnis erhält.

Dieses Verhalten ist allgemein wünschenswert, falls Parameter in der NixOS-Konfiguration verändert werden, deren Standartwert anderweitig gesetzt wird: beispielsweise die Aktivierung von Turboboost oder das Setzen der maximalen bzw. minimalen CPU-Frequenz [44]. Hierfür braucht es eine allgemeine Möglichkeit, um die vorherigen Werte eines Parameters zu sichern.

**Erweiterbarkeit** der Implementierung ist wichtig, damit dieser Ansatz die Chance hat, ein Standardwerkzeug für NixOS zu werden. Dadurch müssten die Autoren von Einreichungen diesen Teil der Artefakte nicht selbst implementieren, was bei der Verbreitung hinderlich wäre.

Eine Lösung, wo ein Programm die gesamte Zustandsverwaltung realisieren würde, würde nicht skalieren: aktuell enthält NixOS ca. 1700 Module, sowie ca. 100000 Pakete. Eine monolithische Lösung würde in einem sehr großen Paket enden. Entsprechend muss eine Implementierung über eine Art Pluginsystem möglichst leicht erweiterbar sein.

**Installation** von NixOS Systemen ist in dieser Arbeit vorerst keine Anforderung. Für dieses Problem gibt es hinreichend viele Lösungen, beispielsweise OpenTofu [101] zur Erzeugung von Cloud-VMs oder *disko* [83] für die Konfiguration des Disk-Layouts und Erzeugung eines Skripts zur Partitionierung mit NixOS-Konfiguration.

Es wird also eine lauffähige NixOS-Installation vorausgesetzt. Der Anwendungsfall, dass ein Dateisystem mit spezieller Konfiguration für eine Evaluation angelegt werden muss, ist valide, wird allerdings als ein „normaler“Anwendungsfall der *Provisionierung* angesehen.

## 4.2 Modellierung

Das Modell für die Verwaltung beliebiger Aspekte des Systemzustandes baut auf der Modellierung von Aeolus$^{-}$ [106] aus Abschnitt 2.5 auf. Die Abbildung konvergenter Deployments wird dabei auf die Aspekte angewendet, die auch bei Aktivierung in NixOS konvergent sein müssen.

### 4.2.1 Grundlagen

Für den Anwendungsfall der Zustandsverwaltung unter NixOS werden die Komponenten in die Typen *Root*, *Service*, *Upgrade* und *Check* eingeteilt mit verschiedenen Zuständen. Anhand der Typen der Komponente definiere ich später verschiedene Besonderheiten im Verhalten.

Der Typ *Root* besitzt die Zustände

$$Q_{\text{root}} = \{\text{old}, \text{new}\}$$

welche den allgemeinen Übergang zu einer neuen Systemkonfiguration repräsentieren. Alle Schritte sind Abhängigkeiten dieses Übergangs. Diese Komponente dient dazu, um alle weiteren Schritte als Teil des Übergangs einer alten in eine neue Konfiguration darzustellen. Entsprechend ist es durchaus möglich, dass spätere Iterationen einer Implementierung den Schritt *Old* → *New* nicht explizit komplett entfernen.

Der Typ *Upgrade* besitzt die Zustände

$$Q_{\text{upgrade}} = \{\text{wait}, \text{checkpoint}, \text{done}, \text{rollback}\}$$

Der Übergang wait → checkpoint wird zur Absicherung des Systems vor den eigentlichen Vorgängen benutzt: beispielsweise können hier alle Dienste gestoppt werden, die verändert werden oder eine Sicherung des Dateisystems kann hier vorgenommen werden.

Der Übergang checkpoint → done wird ausgeführt, falls alle vorherigen Übergänge erfolgreich waren. Dies kann beispielsweise genutzt werden, um veränderte Dienste wieder zu starten. Entsprechend wird das Verhalten der Aktivierung der aktuellen Konfiguration – nämlich der Implementierung von `switch-to-configuration.pl` [87] – in diese Übergänge eingebettet.

Im Fehlerfall wird der Übergang checkpoint → done ausgeführt, welcher den vorherigen Zustand des Systems wiederherstellen soll. Beispielsweise werden hier Sicherungen wiederhergestellt und vorherige Dienste wieder gestartet. Genaueres zum Zurückrollen wird in Abschnitt 4.2.2 beschrieben.

Der Typ *Service* besitzt die Zustände

$$Q_{\text{service}} = \{\text{active}, \text{inactive}, \text{upgrade}, \text{undo}\}$$

Es ist zu beachten, dass dieser Typ für beliebige Zustandsveränderungen an Komponenten im System benutzt werden kann. Die tatsächlichen Modifikationen am Systemzustand sollen hier im Übergang inactive → upgrade geschehen. Der *undo*-Zustand ist nur für das Zurückrollen 4.2.2 gedacht, da in diesem Fall die Zustandsbezeichnung *upgrade* irreführend erscheint.

Zuletzt gibt es noch den Typ *Check* mit den Zuständen

$$Q_{\text{check}} = \{\text{pending}, \text{verified}\}$$

wo im Übergang eine bestimmte Eigenschaft überprüft werden kann. Wenn der Übergang fehlschlägt, wird die Aktivierung der Konfiguration verweigert und es wird zurückgerollt. Sofern der Übergang erfolgreich ist, gibt es von Komponenten dieser Klasse keine weiteren Operationen.

Die Operationen *bind* & *unbind* geschehen nur implizit: die Schnittstellen sind primär dazu da, um Abhängigkeiten abzubilden – z.B. dass erst Migrationen ausgeführt werden, sobald ein Snapshot des Dateisystems gemacht wurde – enthalten allerdings keine eigene Logik. Diese wird nur bei *stateChange* ausgeführt. In der Implementierung haben diese beiden Operationen damit keine Bedeutung.

*del* & *new* werden in diesem Modell ausgeklammert: für Komponenten vom Typ *Check* & *Upgrade* gibt es keinen Grund für das Anlegen und Entfernen, da der gesamte Lebenszyklus über Zustandsveränderungen abgebildet werden kann.

Ich habe entschieden, analog bei Komponenten vom Typ *Service* zu verfahren: es stellt sich raus, dass das weniger Sonderfälle in der Implementierung ergibt. Beispielsweise existiert dann kein Unterschied aus Sicht der Implementierung, ob ein Paket soeben frisch installiert wurde und der Dienst noch nicht läuft oder ob der Dienst vor der Aktivierung z.B. manuell von einem Administrator gestoppt wurde. Wie Änderungen zu bestimmen sind, wird in Abschnitt 4.2.3 genauer erläutert.

Abbildung 4.1 liefert eine graphische Repräsentation des angepassten Komponentenmodells. Der Zustand *undo* der Komponente *PostgreSQL* ist ausgegraut, weil dieser nur für das Zurückrollen (vgl. Abschnitt 4.2.2) relevant ist. Zur Aktivierung eines neuen NixOS Systems würden sich nun Schritte ergeben, wie in Abbildung 4.2 angegeben.

Die Namen der Schnittstellen sind hier der Einfachheit halber nach dem Schema *Komponentenname_Zustandsname* benannt.

Die jeweiligen Zustandsübergänge werden hier durch die Abhängigkeiten Komponentenmodell ausgelöst. Daraus ergibt sich die Reihenfolge der einzelnen Schritte. Jeder *stateChange()* ist ein Schritt, zu welchem eine Komponente Code ausführen kann.

Sofern keine Abhängigkeiten über Komponentenverbindungen zwischen zwei Übergängen bestehen, ist die Reihenfolge undefiniert. In Abbildung 4.2 betrifft dies beispielsweise die Übergänge von *wait* zu *checkpoint*, sowie von *checkpoint* zu *done* der Komponenten *Snapshot* & *Switch Configuration*. Es ist auch erlaubt, beide Übergänge parallel laufen zu lassen.

Entsprechend werden die Zustandsübergänge nach topologischen Generationen [12] sortiert.

**Notation** Abweichend zu Aeolus [106] gibt es folgende zusätzliche Notationen:

- Übergang von einer Konfiguration zur nächsten, z.B. durch einen Zu-

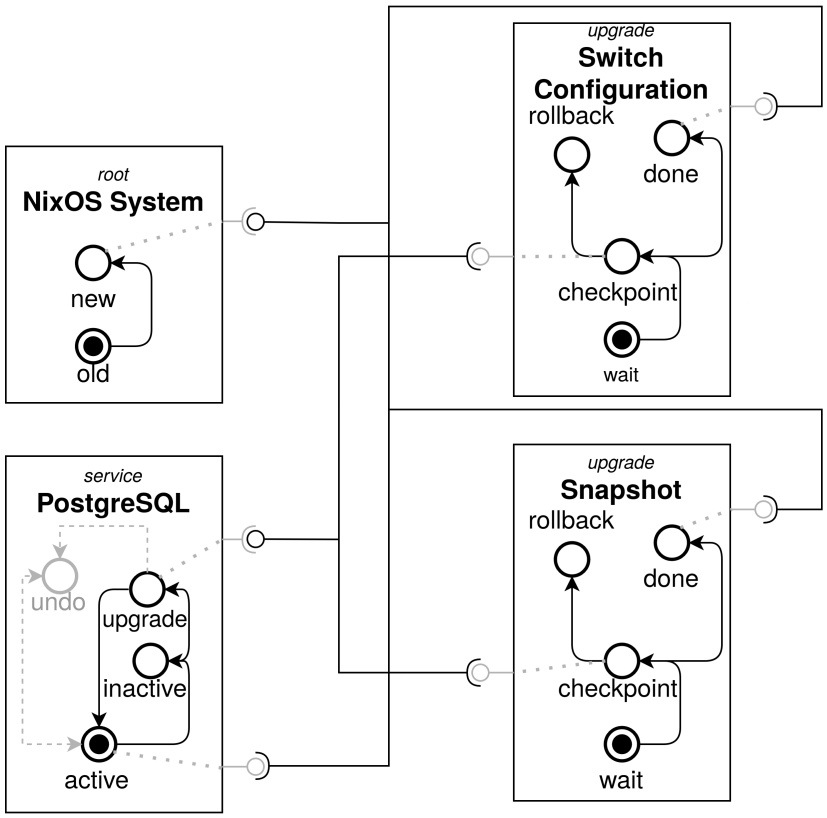


Abbildung 4.1: Beispiel eines Komponentenmodells für NixOS

standsübergang (*stateChange*):

$$C_n \Longrightarrow C_{n+1}$$

- Mehrere dieser Übergänge von $C_n$ zu $C_m$ $(m > n)$ werden als

  $$C_n \Longrightarrow_i C_m$$

  bezeichnet, wobei $i$ die Anzahl der Schritte ist. Wenn zwei Komponenten keine direkte Abhängigkeit haben und ggf. parallelisiert werden können, dann werden diese zwei Übergänge als ein Schritt und nicht als zwei Schritte gewertet.

- $C_N$ bezeichnet die finale Konfiguration, also die Konfiguration, wo die Komponente *NixOS System* in den Zustand *new* übergegangen ist.

- $C_f$ bezeichnet in weiteren Ausführungen die Konfiguration, bei der der letzte Zustandsübergang fehlgeschlagen ist.

```
1  stateChange(z1: postgresql, active, inactive)
2
3  bind(postgresql_inactive_switch, z2: switch_cfg, z1: postgresql)
4  bind(postgresql_inactive_snap, z3: snapshot, z1: postgresql)
5  stateChange(z2: switch_cfg, wait, checkpoint)
6  stateChange(z3: snapshot, wait, checkpoint)
7
8  bind(switch_checkpoint, z1: postgresql, z2: switch_cfg)
9  bind(snapshot_checkpoint, z1: postgresql, z3: snapshot)
10 unbind(postgresql_inactive_switch, z1: postgresql, z2: switch_cfg)
11 unbind(postgresql_inactive_snap, z1: postgresql, z3: snapshot)
12 stateChange(z1: postgresql, inactive, upgrade)
13
14 unbind(switch_checkpoint, z1: postgresql, z2: switch_cfg)
15 unbind(snapshot_checkpoint, z1: postgresql, z3: snapshot)
16 stateChange(z1: postgresql, upgrade, active)
17 bind(postgresql_active, z1: postgresql, z4: root)
18
19 stateChange(z2: switch_cfg, checkpoint, done)
20 stateChange(z2: snapshot, checkpoint, done)
21 bind(switch_done, z2: switch_cfg, z4: root)
22 bind(snapshot_done, z3: snapshot_cfg, z4: root)
23
24 stateChange(z4: root, old, new)
```

Abbildung 4.2: Schritte zur Aktivierung der Konfiguration aus Abbildung 4.1

- $C_n$.req $\subseteq Z \times Z$ bezeichne die Menge aller gebundenen, benötigten Schnittstellen in der aktuellen Konfiguration. Dabei ist das erste Element die Komponente mit der benötigten Schnittstelle und das zweite Element die Komponente, die diese Schnittstelle im aktuellen Zustand bindet.

  Umgekehrt sei $C_n$.prov $\subseteq Z \times Z$ die Menge aller angebotenen Schnittstellen der aktuellen Konfiguration, wobei das je erste Element eine vom zweiten Element angebotene Schnittstelle bindet.

  Das bedeutet auch, dass – wie in Abbildung 4.2 angedeutet – im Modell eine Schnittstelle pro Abhängigkeit existiert. Es binden also nicht mehrere Komponenten an eine Schnittstelle einer *Upgrade*-Komponente, sondern es werden pro Komponente Schnittstellen erzeugt.

### 4.2.2 Fehlerbehandlung & Zurückrollen

Es gibt zwei Aussagen zum Verhalten bei Zurückrollen

$$\forall z \in Z : C_f[z].\text{state} \in \{\text{checkpoint}, \text{done}\} \Rightarrow C_0'[z].\text{state} = \text{rollback}$$

$C_0'$ meint hier die ursprüngliche Konfiguration $C_0$, die durch Zurückrollen wieder erreicht wurde. Mit anderen Worten: jede Komponente vom Typ *Upgrade* muss, sofern der Zustand *Checkpoint* oder *Done* erreicht wurde, zurückgerollt werden.

Des Weiteren gilt:

$$\forall z \in Z : (\exists n > 0 : C_0[z] \Longrightarrow_n C_f[z]) \wedge (\exists m > 0 : C_f[z] \Longrightarrow_m C_0'[z]) \Rightarrow m = n$$

Mit anderen Worten: jeder einzelne Zustandsübergang hat ein Pendant im Fall eines Rollbacks. Die Anzahl der Übergänge einer Komponente vor und nach dem fehlgeschlagenen Übergang ist gleich:

$$\begin{array}{ccccc} \text{Active} & \rightarrow \text{Inactive} & \rightarrow \text{Upgrade} & \rightarrow \text{Active} \\ & \swarrow & \downarrow & \\ \text{Active} & \rightarrow \text{Inactive} & \rightarrow \text{Undo} & \rightarrow \text{Active} \end{array}$$

Diese Darstellung veranschaulicht, dass zu jedem Zustand die Schritte zurück exakt der Anzahl der durchgeführten Schritte entspricht. Zusätzlich werden nicht zwingend dieselben Zustände benutzt, z.B.

$$\begin{array}{ccccc} \text{Wait} & \rightarrow \text{Checkpoint} & \rightarrow \text{Done} \\ & & \downarrow \quad \downarrow & \\ & & \text{Rollback} & \leftarrow \text{Checkpoint} \end{array}$$

Auch von *Done* kann auf *Rollback* gewechselt werden. Intuitiv ist das wie folgt zu erklären: zwar sind alle Komponenten, die nach dem Checkpoint abgeschlossen wurden, erfolgreich, aber da die Aktivierung aber atomar sein soll, werden auch diese zurückgerollt.

Im Falle eines Fehlers wird niemals das neue System aktiviert. In einigen Fällen kann es dennoch sinnvoll sein, Fehler zu ignorieren und das System zu aktivieren. Ein klassisches Beispiel sind Dienste, die beim ersten Versuch fehlschlagen können, beispielsweise ACME [1] Automatisierung. Wie damit umzugehen ist, ist ein Implementierungsdetail der Komponenten. Eine Komponente, die Dienste aktiviert, könnte auch so implementiert werden, dass solche Fehler zwar geloggt werden, aber nicht als Fehler im Komponentensystem behandelt werden.

Auch beim Zurückrollen gilt, dass die Ausführungsreihenfolge zwischen zwei Komponenten, zwischen denen es keine explizite Abhängigkeit gibt undefiniert ist und diese potentiell parallelisiert werden dürfen.

Hier definiere ich nur das Verhalten im Fall von Fehlern, die abgefangen werden können. Es liegt im Ermessen der Implementierer zu entscheiden, auf welche Fälle das zutrifft. In nicht abfangbaren Fällen sollten aber Anweisungen ausgegeben werden, wie manuell eine Reparatur durchgeführt werden kann.

### 4.2.3 Änderungserkennung und Kompatibilität

Wenn eine Komponente initialisiert wird, wird immer ein Anfangszustand $q_0$ angenommen. Bei jeder Komponente vom Typ *Service* ist dieser *Active*, unabhängig davon, ob diese Eigenschaft tatsächlich zutrifft.

Im ersten Schritt werden Änderungen untersucht.

Dabei wird zwischen kompatiblen und inkompatiblen Änderungen unterschieden: kompatible Änderungen sind alle, die die Komponente wieder an den gewünschten Zustand angleichen kann. Wenn der Dienst, den eine *Service*-Komponente repräsentiert, nicht aktiv ist, wäre es eine kompatible Änderung, diesen im Laufe der Zustandsübergänge wieder zu starten.

Sofern es nur kompatible Änderungen gibt, kann die Zustandsverwaltung laufen. Sollten inkompatible Änderungen erkannt werden, wird die Aktivierung verweigert. Ein Beispiel wäre das Downgrade eines Dienstes, der keine Downgrades unterstützt.

Kompatible Änderungen werden zwischen *normal* und *indifferent* unterschieden. Die Eigenschaft *indifferent* ist relevant, um ggf. überflüssige Komponenten zu entfernen, siehe dazu Abschnitt 4.2.4.

Repräsentiert werden Änderungen über

$$c \in \{\text{indifferent}, \text{normal}, \epsilon\}$$

wobei $\epsilon$ „keine Änderung“ bedeutet.

Die Komponentenrepräsentation $U$ wird mit $c$ zu einem Sextupel erweitert. Es gilt die Notation $C[z]$.chg, um die Art der Änderungen einer Komponente aus einer Konfiguration zu bestimmen.

### 4.2.4 Abkürzungen

*Upgrade*-Komponenten sind potentiell nicht günstig: z.B. ist das Sichern von Daten auf Dateisystemen ohne Snapshot-Funktionalität teuer. Auch können diese Komponenten für Anwendungsfälle, wie das Entfernen der Maschine

aus dem Load-Balancer benutzt werden, was eine gewisse Disruption mit sich bringt.

Entsprechend sollten Schritte, die nicht erforderlich sind, übersprungen werden. Dies kann über sog. Abkürzungen (von mir *Short Circuit* genannt) abgebildet werden. Wie in Abschnitt 4.2.3 erwähnt, bestimmt jede Komponente die Änderungen, die es durchzuführen gilt.

Zuerst werden alle Komponenten entfernt, die keine Änderungen erkannt haben.

Dann werden die sog. *indifferenten* Änderungen untersucht: dieses Verhalten betrifft nur Komponenten vom Typ *Service* & *Upgrade*. Dort wird zwischen Änderungen und *indifferenten* Änderungen unterschieden.

Folgende Vorbedingungen gelten für eine Komponente $z_1$:

- $z_1$ hat nur indifferente Änderungen.
- Die Komponente hat nur Schnittstellen zu Komponenten $z_i$ mit folgenden Eigenschaften:
  - Alle $z_i$ haben einen Übergang, der ein direkter Nachfolger des Übergangs *Wait* $\rightarrow$ *Checkpoint* bzw. *Active* $\rightarrow$ *Inactive* von $z_1$ ist.
  - Alle $z_i$ haben einen Übergang, der ein direkter Vorgänger des Übergangs *Checkpoint* $\rightarrow$ *Done* bzw. *Upgrade* $\rightarrow$ *Active* von $z_1$ ist.
  - Alle $z_i$ haben keine oder nur indifferente Änderungen.

Falls dies zutrifft, kann die Komponente $z_1$ aus dem Graph entfernt werden.

Wichtig zu beachten ist, dass die Prüfung der direkten Nachbarn ausreicht: jede Komponente **muss** Abhängigkeiten zu allen anderen Komponenten enthalten, deren Zustandsübergänge voneinander abhängen.

Nach Entfernung aller indifferenten Komponenten gilt folgende Eigenschaft:

$$
\begin{aligned}
\nexists a : & C_n[a].\text{chg} = \text{indifferent} : \\
& \langle a, b\rangle \in C_n.\text{req} \wedge C_{n-1}[a].\text{state} \in \{\text{checkpoint}, \text{inactive}\} \\
& \quad \Rightarrow C_{n-1}[b].\text{state} \neq C_n[b].\text{state} \wedge C_n[b].\text{chg} = \text{normal} \\
& \wedge \langle a, b\rangle \in C_n.\text{prov} \wedge C_n[a].\text{state} \in \{\text{done}, \text{active}\} \\
& \quad \Rightarrow C_{n-1}[b].\text{state} \neq C_n[b].\text{state} \wedge C_n[b].\text{chg} = \text{normal}
\end{aligned}
$$

In Worten ausgedrückt: es existiert keine Komponente $a$ mit nur indifferenten Änderungen, die eine Schnittstelle zu einer Komponente $b$ mit normalen Änderungen hat, falls

- ein Zustandsübergang von $b$ ein direkter Nachfolger des Übergangs zu *Checkpoint* bzw. *Inactive* von $a$ ist
- und ein Zustandsübergang von $b$ ein direkter Vorgänger des Übergangs zu *Done* bzw. *Active* von $a$ ist

Falls in Abbildung 4.1 die Komponenten *Switch Configuration* und *Snapshot* indifferent sind und *PostgreSQL* keine Änderungen hat, würden alle drei Komponenten nicht ausgeführt werden.

### 4.2.5 Neustarts bei Aktivierung

Es kann vorkommen, dass manche Teilschritte der Zustandsverwaltung Neustarts des Systems erfordern. Dies wird nicht gesondert in dem Modell gehandhabt, sondern kann durch eine Komponente vom Typ *Upgrade* abgebildet werden. Diese kann im Übergang *Checkpoint* $\rightarrow$ *Done* einen Neustart durchführt und – je nach Anforderungen – dies erneut im Fall eines Zurückrollens wiederholt.

In diesem Fall kann der Rest der auszuführenden Komponenten auf dem Dateisystem serialisiert werden und gestartet werden, nachdem das `initrd` verlassen wurde und bevor alle Systemdienste gestartet werden. Im Fall von `systemd` [73] bevor das `multi-user.target` erreicht wird.

### 4.2.6 Entfernung alter Daten

Die Entfernung obsoleter Aspekte des Systemzustandes ist nicht explizit berücksichtigt: dies ist eine Änderung, wie in Abschnitt 4.2.3 beschrieben.

Eine Umsetzungsmöglichkeit ist es, alle Komponenten nicht nur in der NixOS-Konfiguration zu speichern, sondern auch z.B. auf dem Dateisystem des Zielsystems: dadurch kann bestimmt werden, welche Komponenten aus der Konfiguration entfernt worden sind. Damit kann die Entfernung des Zustandes in das Komponentenmodells mit aufgenommen werden. Dabei muss die Komponente ausgeführt werden, diese Änderung selbst erkennen und durchführen

Auf NixOS ist das möglich, weil die Komponenten der aktuellen Konfiguration ein Teil des Closure des Systems sind und damit nicht gelöscht werden können, solange die nächste Konfiguration nicht aktiviert wurde.

# Kapitel 5

# Implementierung

In Kapitel 4 habe ich einen Entwurf für Zustandsverwaltung zur Aktivierung eines NixOS [79]-Systems entworfen. In dieser Arbeit zeige ich die Tauglichkeit von NixOS für Evaluationsumgebungen, wobei der Aspekt der Zustandsverwaltung implementiert wurde. Die hierfür entwickelte Lösung, genannt. S.M.A.S.H. [37] bzw. Smash – State Management Avoiding Shell – wird in diesem Abschnitt vorgestellt, sowie einige Implementierungsdetails diskutiert.

## 5.1 Integration in NixOS

Um mögliche Adoption innerhalb des NixOS-Projektes selbst zu ermöglichen, wollte ich eine möglichst einfache Integration in den Prozess der Aktivierung.

### 5.1.1 Zusammenspiel mit der Aktivierung

Wie in Abschnitt 2.1 erläutert, wird die tatsächliche Aktivierung der Konfiguration – welche u.a. den Neustart von veränderten Systemdiensten beinhaltet – über das Skript `switch-to-configuration` [87] ab. Dies ist de facto ein Teil der API von NixOS, dessen Existenz und Verhalten von vielen anderen Werkzeugen aus dem Ökosystem als gegeben hingenommen werden. Entsprechend legt das Modul, das ich zur Integration von Smash in NixOS entwickelt habe, ein kleines Skript in *system.build.toplevel* ab, welches dieselben Argumente wie `switch-to-configuration` akzeptiert, aber intern Smash aufruft. Dadurch war es möglich, Smash in beliebige NixOS-Konfigurationen einzubetten und für die Evaluationen kein neues Deploymentwerkzeug zu benötigen. Ich konnte weiterhin `colmena` [107] benutzen, womit Deployments auf andere Systeme bzw. von mehreren Systemen angenehmer abzubilden sind.

```rust
use libstc::consts::*;
/* use ... */
fn main() -> anyhow::Result<()> {
  let current_system_bin = std::path::PathBuf::from(
    "/run/current-system/sw/bin"
  ).canonicalize()?;
  let dbus_conn = LocalConnection::new_system()?;
  let (systemd, _) = new_dbus_proxies(&dbus_conn);
  let settings = libstc::State::Settings { /* ... */ };
  // Erkennen, welche Dienste, Mounts, Swaps usw. sich
  // geändert haben.
  let (state_tracking, switch) = UnitTracker::init_from_maps(
    START_LIST_FILE, RESTART_LIST_FILE, RELOAD_LIST_FILE
  ).compute_changeset(
    &settings,
    &current_system_bin,
    Action::DryActivate,
    &systemd,
  )?;
  state_tracking.perform_dry_activation(
    &switch,
    &settings.out,
    &settings.toplevel,
    DRY_RESTART_BY_ACTIVATION_LIST_FILE,
    DRY_RELOAD_BY_ACTIVATION_LIST_FILE,
  )?;
  Ok(())
}
```

Abbildung 5.1: Implementierung der *dry Activation*, der reinen Erkennung, was sich geändert hat.

Ein Problem hierbei ist, dass `switch-to-configuration` ursprünglich ein Perl-Skript war, das Zeile für Zeile nach Rust [17] portiert wurde [69]. Das bedeutet, dass die Struktur von diesem Programm sehr monolithisch ist und folglich kaum wiederzuverwenden. Allerdings wollte ich diese Komponente nicht von Grund auf neuschreiben, da dieses Programm das das über Jahre gesammelte Wissen über die Feinheiten der korrekten Aktivierung von Systemdiensten und Geräten auf `systemd` [73]-basierten Systemen enthält. Entsprechend habe ich mich für einen Zwischenweg entschieden:

nachdem Smash in Rust implementiert ist, habe ich die Rust-Portierung von `switch-to-configuration` so umstrukturiert, sodass dies als Bibliothek wiederverwendet werden kann [38].

Diese ist sehr minimalistisch gehalten, reicht aber aus, um die einzelnen Schritte in einem anderen Programm anzusteuern.

Abbildung 5.1 zeigt, wie das ursprüngliche Skript nun als Bibliothek strukturiert ist und wie diese benutzt werden kann, um einzelne Teilschritte selbst durchzuführen bzw. in Anwendungen einzubetten für Recherchezwecke, ähnlich wie bei Smash. Hierbei handelt es sich nur um die reine Änderungserkennung.

Zu beachten ist, dass diese Abbildung stark vereinfacht wurde: so wurden z.B. die `use`-Deklarationen ausgelassen und einige Prüfungen vor dem Start des Programms ebenfalls. Der Pfad zu der neuen Konfiguration wird in der *Settings*-Struktur in Zeile 9 abgelegt.

Vor Zeile 9 wird der Ort bestimmt, an welchem die aktuelle Generation abgelegt ist, sowie eine IPC-Verbindung zu `systemd` eingerichtet. In Zeile 12 wird bestimmt, welche Dienste usw. sich eigentlich geändert haben. Diese Information wird in bestimmten Dateien abgelegt, die hier als Konstanten übergeben werden, um im Falle eines Absturzes diese Informationen wiederverwenden zu können.

In Zeile 20 wird das Ergebnis ausgegeben, d.h. welche Dienste gestartet bzw. gestoppt werden usw.. Die Ausgabe entspricht hier der Einfachheit halber exakt derjenigen von `switch-to-configuration`, da der Code bei der Umstrukturierung nicht weiter modifiziert wurde.

### 5.1.2 Aufbau des NixOS-Moduls

Ein Weiterer wichtiger Aspekt ist die Möglichkeit der Skalierung: aktuell enthält NixOS ca. 1700 Module. Auch wenn das nicht bedeutet, dass 1700 Komponenten benötigt werden, heißt das, dass der naive Ansatz – alle Komponenten in einem Paket zu implementieren – auf Dauer nicht skaliert.

Spätestens wenn es um den Anwendungsfall dieser Arbeit geht – die Aufbereitung von Evaluationsumgebungen mit NixOS – wird das deutlich: wenn für eine Umgebung eine angepasste Komponente benötigt wird, sollte diese nicht erst in Smash akzeptiert werden müssen.

Abbildung 5.2 zeigt eine Konfiguration mit Smash zur Aktivierung und zwei zusätzlichen Komponenten *requried-hyperthreading* vom Typ *Check* & *PostgreSQL* vom Typ *Service*. Automatisch wird bereits die Komponente, die die Aufgaben von `switch-to-configuration` übernimmt und vom Typ *Upgrade* ist, konfiguriert. Standardmäßig werden alle Komponenten vom Typ *Service* so eingereiht, sodass der Übergang *Inactive* $\rightarrow$ *Upgrade* geschieht,

```
1  { config, lib, pkgs, ... }:
2  {
3    system.state = {
4      enable = true;
5      components.check.required-hyperthreading = {
6        implementation = lib.getExe pkgs.component-hypth;
7      };
8      components.service = {
9        postgresql = {
10         implementation = lib.getExe pkgs.component-bar;
11         payload = {
12           inherit (config.services.postgresql.package) version;
13         };
14         requires.upgrade = {
15           required-hyperthreading = {
16             enable = true; state = "verified";
17           };
18         };
19       };
20     };
21   };
22 }
```

Abbildung 5.2: Konfiguration einer Komponente mit dem NixOS Modul von Smash

wenn alle Komponenten vom Typ *Upgrade* im Zustand *Checkpoint* sind. Konkret bedeutet das, dass die `switch-to-configuration`-Komponente alle sich geänderten Dienste gestoppt hat, wenn die Upgrades der *Service*-Komponenten laufen.

Der Payload in Zeile 11 hat nur die Anforderung, dass dieser nach JSON serialisierbar sein muss. Dies ist alles an Konfiguration, was eine Komponente erhält. Beliebige Werte aus der NixOS-Konfiguration können hier hineingereicht werden. In diesem Beispiel wird die neue PostgreSQL Version an die Komponente gereicht: damit kann z.B. geprüft werden, ob die Installation älter ist, als von der Konfiguration angefordert und ggf. ein Upgrade durchgeführt werden.

Über die Option *requires* wie in Zeile 14 können weitere Abhängigkeiten deklariert werden. Die Aussage an dieser Stelle ist, dass wenn die Komponente *postgresql* den Zustand *Upgrade* erreicht, muss die Komponente

*required-hyperthreading* bereits den Zustand *Verified* erreicht haben. Der Wert `enable = true;` ermöglicht es, in eigenen Modulen diese Abhängigkeiten zu deaktivieren: indem die Option `requires.upgrade.stc.enable` auf `false` gesetzt wird, wäre die Bedingung, dass die Komponente erst in *Upgrade* übergeht, sobald die Komponente *stc* in *Checkpoint* ist, aufgehoben. Der Einfachheit halber können Abhängigkeiten nur in eine Richtung angegeben werden.

Diese Abhängigkeiten repräsentieren die Schnittstellen aus dem Komponentenmodell aus Abschnitt 4.2.1. Dieser Aspekt des Modells existiert nur im Nix-Code, die Komponenten werden einen Graph umgewandelt, wobei anhand der Abhängigkeiten eine Adjazenzliste erzeugt wird.

Abbildung 5.3 enthält einen Ausschnitt des Codes zur Erzeugung der Daten – in Smash auch Manifest genannt – die vom Nix-Code an Smash selbst gereicht werden. Das beinhaltet zum einen eine Liste an Komponenten in Zeile 15, hier nicht nach Typ gruppiert, sondern als flaches Attributset, in dem der Typ eine weitere Eigenschaft ist. Wie bereits in Abschnitt 4.2.1 erwähnt, sind die Knoten des aus dem Modell resultierenden Graphen die jeweiligen Zustandsübergänge. Die Deklaration der `nodes` in Zeile 2 enthält genau dies, wobei die Variable `transitions` eine Zuordnung von des Typs – *Service*, *Upgrade* usw. – zu der Liste aller Übergänge im Format `{ from = ...; to = ...; }`.

Die Kanten aus Zeile 8 werden als Adjazenzliste angegeben, wobei jedes Element ein Tupel – in Nix als Liste repräsentiert – bestehend aus dem Index des ausgehenden Knotens und einer Liste aller Zielknoten als zweitem Element ist. Die hierfür relevante Funktion `mkEdgesFor` habe ich hier aus Platzgründen ausgelassen.

Nachdem Smash inkl. Konfiguration auch Teil des Systems ist, ist es nicht möglich, den Storepfad des Systems in Nix in den Payload einer Komponente zu reichen. Dieser Wert ist aber relevant für die Aktivierung, um u.a. zu bestimmen, welche Dienste sich geändert haben. An dieser Stelle gibt es folgendes Sonderverhalten: wenn der Platzhalter `@@OUT@@` im finalen Manifest vorkommt, wird dieser durch den Storepfad des Systems ersetzt. Dieser Schritt ist in den Code von `system.build.toplevel` integriert. An dieser Stelle ist der Storepfad bereits bekannt und liegt in der Umgebung unter `$out`, vgl. Abbildung 5.4, Zeile 11.

In Zeile 2 wird das JSON Manifest erzeugt. Die Variable `manifest` entspricht dabei dem Ergebnis aus Abbildung 5.3. Zur Buildzeit wird Smash einmal aufgerufen, um das Manifest zu prüfen. Das beinhaltet u.a. ob Smash das JSON korrekt einlesen kann und eine Überprüfung, ob der Graph Zyklen hat. Auf diese Weise wird bereits zur Bauzeit erkannt, ob eine Konfiguration gültig ist.

```
{ lib , allComponents, transitions, mkEdgesFor }: with lib; let
  nodes = foldlAttrs (acc: type: cmps: acc ++ foldlAttrs
    (acc: name: { ... }: acc ++ forEach transitions.${type}
      (transition: { inherit name transition; }))
    [ ]
    cmps
  ) [ ] allComponents;
  edges = foldlAttrs (
    acc: type: components:
    foldlAttrs (acc: name: { requires, ... }:
      acc ++ mkEdgesFor name requires) acc components
  ) [ ] allComponents;
in {
  inherit nodes edges;
  components = concatMapAttrs (
    type: concatMapAttrs (
      name:
      { implementation, payload, ... }: {
        ${name} = {
          inherit implementation type payload;
        };
      }
    )
  ) allComponents;
}
```

Abbildung 5.3: Vereinfachte Implementierung der Erzeugung einer Adjazenzliste aus dem Komponentenmodell in Nix.

## 5.2 Aufbau von Smash

Der folgende Abschnitt gibt einen Überblick über Smash selbst, die Implementierung von Abschnitt 4.

### 5.2.1 IPC mit Varlink

In Abschnitt 7.1.1 empfehle ich, größere Komponenten, etwa in der Größe eine Komponente pro Systemdienst, zu implementieren. Dies liegt u.a. daran, dass mit sehr vielen Komponenten die Interprozesskommunikation in Smash sehr teuer werden würde. Dabei ist jede Komponente ein eigener Prozess. Ein

```
{ config, lib, pkgs, ... }: {
  system.build.manifest =
    pkgs.runCommand "manifest.json" {
      nativeBuildInputs = [ pkgs.smash ];
      manifest = builtins.toJSON manifest;
      passAsFile = [ "manifest" ];
    } ''
      smash check-manifest -m $manifestPath
      cp $manifestPath $out
    '';
  system.activatableSystemBuilderCommands = lib.mkAfter ''
    cp ${config.system.build.manifest} $out/manifest.json
    substituteInPlace $out/manifest.json \
      --replace-fail "@@OUT@@" $out
  '';
}
```

Abbildung 5.4: Code zum Auflösen des Platzhalters für den Storepfad des Systems

Grund dafür ist, dass diese Komponenten mit sehr viel beliebiger Software interagieren müssen, teilweise über C-FFIs. Wenn es dabei z.B. zu einem Segmentierfehler kommt und alles in einem Prozess gehalten werden würde, dann ist keine sinnvolle Fehlerbehandlung wie in Abschnitt 4.2.2 beschrieben mehr möglich. Des Weiteren bedeutet das, dass die Wahl der Programmiersprache nicht festgelegt ist und somit die Sprache gewählt werden kann, die sich für eine Komponente am besten eignet.

### Grundlagen

Wie in Abschnitt 2.4 beschrieben, ist Varlink ein leichtgewichtiges Protokoll für die Kommunikation zwischen Prozessen. Es wird für die Kommunikation zwischen Komponenten und dem sog. Smash-Controller selbst benutzt. Die Aufgabe des Smash-Controller beschränkt sich darauf, Zustandsübergänge in Komponenten in der korrekten Reihenfolge auszuführen, sowie Logs von diesen entgegenzunehmen und rauszuschreiben. All diese Aufgaben sind stark IO-gebunden. Aufgrund dieser Eigenschaft, habe ich entschieden, den Smash-Controller in einer asynchronen Tokio-Laufzeitumgebung [41] laufen zu lassen.

Nachdem die Referenzimplementierung von Varlink für Rust [92] nicht

```rust
#[derive(Debug, Clone)]
pub struct FatalError { pub reason: String }
#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct ErrorResponse<E> {
    pub error: String,
    pub parameters: E,
}
#[derive(Debug, Clone)]
pub enum HandlerError<E> {
    ErrorResponse(ErrorResponse<E>),
    FatalError(FatalError),
}
pub trait Handler<T, V, E> {
    fn handle(
        &mut self,
        method: &str,
        parameters: T,
    ) -> impl Future<Output = Result<V, HandlerError<E>>>
    where
        T: serde::de::DeserializeOwned,
        V: serde::Serialize;
}
```

Abbildung 5.5: Struktur eines Nachrichtenverarbeiters für Varlink in Smash

asynchron, sondern blockierend ist, ist diese ungeeignet für den Controller von Smash. Konkret bedeutet das, dass eine Routine, die auf einem Socket auf eine Varlink-Nachricht wartet, nicht von anderen Routinen unterbrochen werden kann, obwohl die Aufgabe sich auf das reine Warten beschränkt. Entsprechend habe ich entschieden, eine sehr minimalistische Implementierung von Varlink selbst zu schreiben. Diese baut auf Serde [102] – einer Bibliothek zur Serialisierung von Strukturen in Rust in Textformate, wie z.B. JSON [7] – und der asynchronen Socket-API von Tokio auf. Der Einfachheit halber beschränke ich mich in dieser Arbeit auf die Unterstützung von UNIX Domain Sockets. Grundsätzlich ist es aber vorgesehen, beliebige Arten von Sockets zu unterstützen. Damit wäre es in Zukunft möglich, auch Komponenten auf anderen Maschinen laufen zu lassen und somit Zustandsübergänge abzubilden, die stärkere Koordination zwischen mehreren Maschinen benötigen.

Abbildung 5.5 zeigt die Serverseite von meiner Varlink-Implementierung: die Nachrichten müssen in eine Struktur vom Typ `T` deserialisiert werden und

nach Verarbeitung wird asynchron eine Struktur vom Typ `V` zurückgegeben.

```
1  dispatch! {
2      ("org.nixos.smash.component.", method, parameters) => {
3          ("Method"
4          , Methods::Method { foo, bar, baz }) => {
5              /* do something */
6
7              Ok(Response {
8                  /* ... */
9              })
10         }
11     }
12 }
```

Abbildung 5.6: Behandlung verschiedener Methoden in einem Varlinkserver

Bei der Fehlerbehandlung wird zwischen schweren Fehlern, der Struktur `FatalError`, und Logikfehlern unterschieden. Erstere sind ein Indikator, dass keine Erholung unmöglich ist, z.B. dass eine Nachricht nicht eingelesen werden kann. Letztere sind Fehler, welche z.B. durch ungültige Eingaben ausgelöst worden sind, aber keinen Abbruch des Servers erfordern. Bei diesem Fehlertyp können weitere strukturierte Daten vom Typ `E` an den Sender zurückgegeben werden.

Der Einfachheit halber muss die Implementierung alle Methoden selbst behandeln. Es gibt keinen Generator wie in der Referenzimplementierung. Um dies etwas zu vereinfachen, gibt es allerdings ein Makro, wie in Abbildung 5.6 gezeigt.

In Zeile 2 wird verifiziert, dass die Varlink-Methode im korrekten Namensraum – *org.nixos.smash.component.* – liegt. Mit der Bedingung in Zeile 3 wird ein Block geöffnet, der die Methode `Method` behandelt, sofern sich der Payload in die Struktur von `Methods::Method` deserialisieren lässt.

Sollte eine dieser Bedingungen nicht zutreffen, wird eine `ErrorResponse` mit der Struktur aus Abbildung 5.5 gesendet. Das aktuelle Stadium dieser Implementierung ist zwar wesentlich einfacher gehalten, als in der Referenzimplementierung, dafür laufen alle Vorgänge asynchron, was die Koordination im Smash-Controller vereinfacht.

Abbildung 5.7 enthält die vereinfachte Funktion, die mit Tokios Socket-API Nachrichten im Varlink-Protokoll versendet. Serialisierung und Deserialisierung wurden aus Gründen der Einfachheit in eigene Funktionen ausgelagert. Die Funktion `read_into_buffer` aus Zeile 17 wurde aus der Refe-

renzimplementierung [92] übernommen. Wie in Varlink vorgeschrieben, liest diese eine Zeichenkette aus dem Socket, bis ein Nullbyte empfangen wird.

In Zeile 13 wird der Socket aufgetrennt, sodass asynchron gelesen und geschrieben werden kann. Im Vergleich zu einer synchronen Implementierung braucht es dafür nicht zwei Socketverbindungen. Stattdessen wird asynchron bei Lese- bzw. Schreibvorgängen sichergestellt, dass es eingehende Verbindungen gibt bzw. dass in den Socket geschrieben werden kann und der Puffer dafür ausreicht und damit nicht blockiert wird. Damit wird dann das serialisierte JSON an die Gegenseite gesendet.

```
1  pub async fn send_varlink_unix<T, V, E>(
2    socket: &mut UnixStream, interface: &str,
3    method: &str, payload: T,
4  ) -> Result<V, RequestError<E>>
5  where
6    V: serde::de::DeserializeOwned, T: serde::Serialize,
7    E: serde::de::DeserializeOwned + Debug,
8  {
9    let request = Request {
10     method: format!("{}.{}", interface, method),
11     parameters: payload,
12   };
13   let (read, mut write) = socket.split();
14   { write_json(&request, &mut write).await?;
15     write.flush().await?;
16   };
17   let buf = read_into_buffer(read).await?;
18   let resp: MaybeResponse<V, E> = read_json(&buf).await?;
19   match resp {
20     MaybeResponse::Error(e) => Err(
21       RequestError::ErrorResponse(e)
22     ),
23     MaybeResponse::Success(s) => Ok(s.parameters),
24   }
25 }
```

Abbildung 5.7: Implementierung des Nachrichtenversandes über Varlink

Die Struktur `MaybeResponse` differenziert bei Antworten zwischen erfolgreichen Antworten und Fehlern, die behandelt werden müssen. Aktuell gelten Fehler von der Gegenseite auch als Fehler im `Result`-Typ: wenn die Struktur

`RequestError::ErrorResponse` zurückgegeben wird, dann handelt es sich um z.B. einen Logikfehler in der Anwendung.

Sollte z.B. ein Schreibvorgang fehlschlagen, weil keine Verbindungen angenommen werden, dann wird in `write_json` ein schwerer Fehler zurückgegeben, ähnlich wie für Abbildung 5.5 erläutert.

### Änderungserkennung

In Smash wird in beide Richtungen kommuniziert, d.h. der Controller schickt Methodenaufrufe an Komponenten und umgekehrt. Wenn eine Komponente gestartet wird, prüft diese anhand ihres Payloads, welche Änderungen durchzuführen sind. Dies geschieht über das Varlink-Schema aus Abbildung 5.8.

```
1  type Change (
2    type: (Normal, NeedsConfirmationAbort, NeedsConfirmationSkip),
3    description: string,
4  )
5  type ChangeReport (
6    strategy: (Normal, Indifferent)
7    changes: []Change,
8    incompatibilities: []string,
9  )
10 type Component (
11   component: string,
12 )
13
14 method ReportIn(
15   component: Component, changes: ChangeReport
16 ) -> ()
```

Abbildung 5.8: Varlink-Schema für die Änderungsanalyse

Dieser Ansatz hat drei Gründe: zum einen signalisieren Komponenten auf diese Weise, dass diese erfolgreich gestartet worden sind. Die Änderungsanalyse soll möglichst früh passieren, sodass dem Plan bei den Zustandsübergängen nurnoch gefolgt werden muss. Außerdem ermöglicht das eine Vorschau über alle geplanten Komponenten.

Die Differenzierung von Änderungstypen in Zeile 2 ist relevant für die Interaktion aus Abschnitt 5.2.5, d.h. um bestimmte Änderungen nur nach explizitem OK durchzuführen.

Die Struktur `ChangeReport` erlaubt es mit dem Tupel aus Zeile 6 anzugeben, ob es sich um indifferente Änderungen handelt, um potentiell im Ausführungsgraphen Abkürzungen zu gehen, wie in Abschnitt 4.2.4 beschrieben.

Neben der Liste an Änderungen, die die Komponente plant durchzuführen, gibt es eine Liste mit inkompatiblen Zuständen – vgl. Abschnitt 4.2.3 – aufgrund derer die Aktivierung fehlschlägt. Die Liste an Zeichenketten ist nur für eine Fehlermeldung da.

### Durchführung der Zustandsübergänge

Sobald alle Komponenten gestartet wurden, wird via Varlink ausschließlich in die andere Richtung kommuniziert. Für diese Kommunikation ist das Schema aus Abbildung 5.9 definiert.

```
1 method StateTransition(
2   to: (
3     Wait, Checkpoint, Done, Rollback,
4     ...
5   ),
6   kind: (Reconcile, Rollback)
7 ) -> ()
```

Abbildung 5.9: Varlink-Schema für die Zustandsübergänge

Im Protokoll ist jeder Zielzustand erlaubt, daher darf `to` aus Zeile 2 jeden Zustand einer Komponente, wie in Abschnitt 4.2 annehmen. Jede Komponente prüft selbst, ob der Übergang valide ist. Für Komponenten in Rust gibt es dafür eine fertige Bibliothek, vgl. Abschnitt 5.2.2 & Abschnitt 5.2.3.

Mit dem Parameter `kind` wird angezeigt, ob eine Konfigurationsaktivierung läuft oder ob zurückgerollt wird. Dies ist wichtig, da Zustände in beiden Fällen erreichbar sind. Aus den Aussagen aus Abschnitt 4.2.2 folgt direkt, dass der Zielzustand des Übergangs eindeutig bestimmt ist. Erst wenn der Übergang erfolgreich durchgeführt wurde, darf die Komponente den Methodenaufruf beantworten.

### Logging

Komponenten selbst senden keine Logs über Varlink. Stattdessen wird die gesamte Ausgabe der Prozesse in einen Dateideskriptor umgeleitet und eine Routine gibt zeilenweise die Logs aus diesem mit dem Komponentennamen als Präfix aus.

Dies geschieht, indem eine UNIX-Pipe angelegt wird, welche in Rust mit Tokio asynchron ausgelesen bzw. beschrieben werden kann. Wenn die Komponente gestartet wird, werden die Schritte aus Abbildung 5.10 durchgeführt.

```
1  let (r_chld_out, w_chld_out) = tokio_pipe::pipe()?;
2  match fork() {
3    Ok(Fork::Parent(_)) => {
4      tokio::spawn(async move {
5        let buf_r = tokio::io::BufReader::new(r_chld_out);
6        let mut lines = buf_r.lines();
7        while let Some(line) = lines
8          .next_line().await? {
9          /* write out */
10       }
11       Ok(())
12     });
13   }
14   Ok(Fork::Child) => {
15     dup2(w_chld_out.as_raw_fd(), 1)?;
16     dup2(1, 2)?;
17     execvp(
18       &CString::new(/* pfad zu komponente */),
19       &[/* ... */],
20     )?;
21   }
22 }
```

Abbildung 5.10: Fork der Komponente aus dem Smash-Controller

Der Code ist ein Auszug aus dem Startvorgang einer Komponente. In Zeile 19 wird im Kindprozess von Smash eine Komponente gestartet, wobei vorher die Dateideskriptoren von `stdout` & `stderr` weitergeleitet werden. Die Argumente, die an eine Komponente gereicht werden sind fest definiert und enthalten u.a. den JSON Payload und die Pfade zu den Sockets für Kommunikation via Varlink. Im Smash-Controller läuft eine asynchrone Routine, welche in Zeile 4 gestartet wird und die Logzeilen aus der Pipe ausliest und rausschreibt.

Aktuell werden keine strukturierten Daten auf diese Weise durchgereicht, die Logs werden als reiner Text behandelt. Ich habe im gesamten Code versucht, strukturiertes Logging zu benutzen, d.h. wichtigen Informationen über `schluessel=wert`-Zuordnungen darzustellen. Bei der Fehlerbehandlung ha-

be ich vorerst einen einfacheren Weg gewählt und lasse Rust aktuell die Datenstrukturen formatieren.

### Laufzeitumgebung

Das Entgegennehmen von Logs ist nicht die einzige Aufgabe zur Laufzeit im Controller. Insgesamt gibt es pro Komponente neben der o.g. Tokio-Routine noch zwei weitere: den Koordinator, der innerhalb des Controllers von der Hauptroutine Nachrichten entgegennimmt, um Zustandsübergänge einzuleiten. Diese ruft also in einer Komponente die Varlink Methode aus Abbildung 5.9 auf.

```
let len = cmps.len();
let (c_tx, mut c_rx) = mpsc::channel::<Coordination>(N);
let mut component_threads = HashMap::new();
for (name, exec_path, payload, r#type) in cmps {
  component_threads.insert(name.clone(),
    ComponentRuntime::new(...)?);
}
let mut result = HashMap::new(); let mut failed = false;
loop {
  if let Ok(Some(Coordination::Ready {
    component, changes, ..
  })) = timeout(from_secs(2), c_rx.recv()).await? {
    if !changes.incompatibilities.is_empty() {
      error!(/* info über Inkompatibilität */);
      failed = true;
    } else {
      let handle = component_threads.remove(&component)?;
      result.insert(component, (handle, changes.clone()));
    }
  } else { /* sind Komponenten abgestürzt? */ }
}
```

Abbildung 5.11: Logik zum Start aller Komponenten

Außerdem eine Routine, genannt Varlink-Server, welche zum Start der Komponente auf einem Socket lauscht, um die Meldung von Änderungen entgegenzunehmen. Diese nimmt also den Methodenaufruf aus Abbildung 5.8 entgegen.

Der Controller selbst läuft in der Hauptroutine und kommuniziert über Tokio-Pipes mit diesen Routinen. Dieser Zwischenschritt wurde eingebaut, um z.B. Fehlerbehandlung zu realisieren, d.h. der Koordinator kann bei bestimmten Fehlern einen neuen Versuch starten ohne dass die Logik dafür im Controller selbst implementiert werden muss.

Mit dem Vorgehen aus Abbildung 5.11 wird die Aktivierung erst gestartet, wenn alle Komponenten bereit sind. Die Abbildung ist dabei stark vereinfacht, da z.B. das Logging vieler Fälle aus Platzgründen ausgelassen wurde.

In Zeile 2 wird eine Tokio-Pipe angelegt. Diese ist dazu da, dass Komponenten Nachrichten an den Controller senden können, z.B. dass diese fertig initialisiert sind und welche Änderungen es gibt – vgl. Zeile 10. Wenn Änderungen inkompatibel sind, wird die Aktivierung in Zeile 14 als fehlgeschlagen markiert. Auf die verbliebenen Komponenten wird dennoch gewartet, alle Änderungen zu empfangen und auszugeben.

Ab Zeile 5 werden die einzelnen Laufzeitumgebungen für die Komponenten konfiguriert, wobei die entsprechenden Routinen für Logging und Koordination erzeugt werden. Die Konfiguration stammt dabei aus dem NixOS-Modul, wie in Abschnitt 5.1 beschrieben. Dabei wird auch eine weitere Tokio-Pipe zurückgegeben, welche zum Initiieren von Zustandsübergängen in Komponenten benutzt werden kann.

Damit wächst die Menge der Routinen pro Komponente linear. Ein Test in Smash stellt sicher, dass es keine blockierenden IO-Operationen gibt und folglich Smash mit nur einem Thread ausführbar ist.

### Überprüfung der Komponenten

Ein mögliches Problem ist, dass ein Bestandteil der Komponente zur Laufzeit mit einem Fehler stoppt, wodurch entweder die gesamte Aktivierung fehlschlägt oder festhängt, weil alle verbliebenen Teile auf eine Nachricht von der fehlgeschlagenen Komponente warten.

Dies wird in Abbildung 5.11 in Zeile 14 durchgeführt. Dabei wird sichergestellt, dass alle Tokio-Routinen noch aktiv sind und mit `waitpid` auch, dass die Komponente selbst noch läuft. Dies geschieht nur, wenn Komponenten zu lange zur Aktivierung brauchen. Wenn alle Komponenten ohne Verzögerung melden, dass diese bereit sind – d.h. kein Timeout beim Warten in Zeile 10 entsteht, wird die Überprüfung nie durchgeführt.

### 5.2.2 Verarbeitung des Ausführungsgraphen

Um die Logik für die Validierung von Zustandsübergängen zu minimieren, habe ich eine Bibliothek, *rust-fsm* [31], benutzt, welche es erlaubt, in einer eingebetteten domänenspezifischen Sprache die Zustandsübergänge zu formulieren.

```
state_machine! {
  #[state_machine(
    input(crate::graph::RunType),
    state(crate::graph::ValidStates))
  ]
  pub state_transitions(Wait)
  // upgrade
  Wait(Reconcile) => Checkpoint,
  Wait(Rollback) => Wait,
  Checkpoint(Reconcile) => Done,
  Checkpoint(Rollback) => Rollback,
  Done(Rollback) => Checkpoint,
  // check
  Pending(Reconcile) => Verified,
  Verified(Rollback) => Verified,
  // ...
}
```

Abbildung 5.12: Formulierung der Zustandsübergänge aller Komponenten

Der Zustand und die Zustandsmaschine werden von einer Struktur mit dem Namen `CurrentState<T>` umschlossen, wobei `T` der Typparameter für das Trait der Zustandsmaschine aus `rust_fsm` ist.

Die Struktur nimmt einen Übergang vom Typ *Reconcile* bzw. *Rollback* entgegen und prüft, ob der Zustandsübergang zu dem erwarteten Zustand erfolgt. Das ist ein Integritätscheck, um Implementierungsfehler zu vermeiden.

Die Knoten des Ausführungsgraphen repräsentieren Zustandsübergänge. Die Reihenfolge und die Knoten selbst sind Teil des Manifests, welches das NixOS Modul erzeugt, siehe Abschnitt 5.1. Für jeden auszuführenden Zustandsübergang geht die Zustandsmaschine in `CurrentState<T>` einen Schritt weiter und prüft, ob der geplant Übergang gültig ist.

Der Graph selbst wird über die Bibliothek *petgraph* [8] repräsentiert. Weil diese keine eigene Implementierung für topologische Generationen ent-

hält [12], benutze ich in Smash eine von mir darum erweiterte Version von *petgraph* [35].

### 5.2.3 Struktur einer Komponente

Abbildung 5.13 enthält die Grundstruktur für eine in Rust geschriebene Komponente. In Zeile 12 wird nur das Logging initialisiert, sodass – wie im Smash-Controller auch – über die Umgebungsvariable `RUST_LOG` das Loglevel gesteuert wird.

```
1  struct Varlink<T: StateMachineImpl<
2    Input = RunType, State = ValidStates
3  >> {
4    name: String,
5    state: CurrentState<T>,
6    // ...
7  }
8  #[derive(Serialize, Deserialize, Debug)]
9  struct Payload { /* ... */}
10 #[tokio::main]
11 async fn main() -> Result<()> {
12   setup_default_helpers()?;
13   cmp_init_and_run::<Payload, Varlink<Impl>, Impl>(Some(
14     smash::graph::Type::Service
15   )).await?;
16   Ok(())
17 }
```

Abbildung 5.13: Grundstruktur einer Komponente in Rust

In Zeile 13 werden die Argumente aus der Kommandozeile der Komponente verarbeitet. Dabei wird sichergestellt, dass diese tatsächlich zu der Erwartung passen, dass die Komponente vom Typ *Service* ist. Der Payload wird in die Struktur in Zeile 9 eingelesen. Im Anschluss werden die Änderungen über den Varlink Methodenaufruf aus Abbildung 5.8 an den Smash-Controller gesendet.

Sollten sich bei der Änderungsüberprüfung inkompatible Änderungen ergeben, wird die Komponente danach angehalten.

Die Varlink-Struktur aus Zeile 1 enthält den Namen und aktuellen Zustand der Komponente in Abhängigkeit der in Abschnitt 5.2.2 definierten Zustandsmachine.

```
1  impl<T: StateMachineImpl<Input=RunType,State=ValidStates>>
2    Handler<ComponentCommands,(),ProtocolError> for Varlink<T>
3  {
4    async fn handle(&mut self, _: &str, _: ComponentCommands)
5      -> varlink_protocol::Result<()>
6    where ComponentCommands: serde::de::DeserializeOwned,
7      (): serde::Serialize,
8    { }
9  }
10 impl VarlinkIface<Payload, Impl> for Varlink<Impl> {
11   fn from_payload(
12     name: String, payload: Payload, state: CurrentState<Impl>
13   ) -> Result<Box<Self>> {
14     Ok(Box::new(Self { /* ... */ }))
15   }
16   async fn changes(&mut self) -> Result<ChangeReport> {
17     Ok(ChangeReport { /* ... */ })
18   }
19 }
```

Abbildung 5.14: Struktur für Initialisierung & Übergänge in einer Komponente

Abbildung 5.14 enthält die notwendigen Trait-Implementierungen für eine Komponente. In Zeile 8 werden die jeweiligen Zustandsübergänge durchgeführt. Dabei benutze ich normalerweise das Hilfsmakro aus Abbildung 5.6.

Für die Umwandlung des Payloads in die Varlinkstruktur wird in bei der Initialisierung die Methode `from_payload` benutzt. In vielen Fällen ist das eine triviale Abbildung in die Varlinkstruktur. Hier ist der Methodenkörper in Zeile 11.

In Zeile 16 ist der Körper für die Methode zur Änderungserkennung. Der Rückgabewert muss dabei die Struktur aus Abbildung 5.8 enthalten. Ich habe bei der Implementierung der Komponenten folgendes Muster entdeckt: ich erzeuge einen Ausführungsplan, der deklarativ die Änderungen beschreibt und füge diesen in einem optionalem Feld der Varlinkstruktur hinzu. Aus diesem Grund ist `self` auch veränderlich.

Abbildung 5.15 ist ein Beispiel für dieses Vorgehen bei der Änderungserkennung in der `changes`-Methode. Dort werden die Änderungen in die Varlinkstruktur geschrieben und diese werden dann im Übergang *Inactive* → *Upgrade* durchgeführt bzw. in *Upgrade* → *Undo* zurückgenommen.

```
1  let existing_topics = kafka.topics();
2  for t in existing_topics {
3    let name = t.name();
4    if self.payload.topics.contains_key(name) {
5      if t.partitions().len()
6        != self.payload.topics[name].partitions {
7        changes.push(Change { ... });
8        todo.push(Operation::Update(
9          name.to_string(),
10         self.payload.topics[name].clone(),
11       ));
12     }
13     self.payload.topics.remove(name).unwrap();
14   } else {
15     changes.push(Change { ... });
16     todo.push(Operation::Delete(name.to_string()));
17   }
18 }
19 for (new, config) in &self.payload.topics {
20   changes.push(Change { ... });
21   todo.push(Operation::Create(new.clone(), config.clone()));
22 }
23 self.changes = Some(todo);
```

Abbildung 5.15: Vereinfachter Auszug aus der Änderungserkennung einer Komponente zum Anlegen von Topics für Kafka

Dazu werden in Zeile 1 die aktuellen Topics aus Kafka [29] ausgelesen. Der Sonderfall, dass Kafka in der letzten Konfigurationsgeneration nicht aktiv war, wird hier der Einfachheit halber ausgelassen.

In Zeile 6 wird geprüft, ob sich die Konfiguration eines Topics geändert hat. Falls dem so ist, wird eine *Update*-Operation hinzugefügt. Sofern das entsprechende Topic nicht in der Liste der Topics aus der Konfiguration enthalten ist, wird dieses zur Löschung freigegeben. Alle hier behandelten Topics werden aus dem Payload gelöscht. Die verbliebenen Topics existieren noch nicht in Kafka und in Zeile 20 als neu markiert.

Hier werden Topics als kongruent nach Traugott et al [103] behandelt. Manuell angelegte Topics werden – sofern diese nicht Teil der Konfiguration sind – wieder gelöscht. Wie dieser Fall behandelt werden soll, ist Implementierungs- bzw. Konfigurationssache. Eine Alternative wäre es, die Komponente spei-

chern zu lassen, welche Topics von dieser Komponente verwaltet wurden und nur Topics zu löschen, wenn diese von dieser Komponente angelegt worden sind und nicht mehr Teil des aktuellen Payloads sind. Siehe dazu auch Abschnitt 5.2.6.

### 5.2.4 Zurückrollen im Fehlerfall

Ich habe in Smash das in Abschnitt 4.2.2 beschriebene zurückrollen implementiert. Dabei gilt die Aussage, dass jeder Zustandsübergang eine Umkehrung beim Zurückrollen hat. In Smash speichere ich die Schritte pro Generation – dargestellt als Menge – an Übergängen in einer `VecDequeue`, einer Warteschlangenstruktur, welche rückwärts abgearbeitet wird.

```
1  while let Some(steps) = rollback_plan.pop_front() {
2    for step in steps {
3      let cmp = self.library.get_mut(&step).unwrap();
4      let cm_runtime = self.processes.0.get(&step).unwrap();
5      if !cmp.state.rollback() {
6        bail!("Transition rejected")
7      }
8      cm_runtime.0.to_component
9        .send(Worker::OpStateTransition {
10         to: cmp.state.state().clone(),
11         kind: RunType::Rollback,
12       }).await?;
13     match self.recv.recv().await.unwrap() {
14       Coordination::Success { .. } => { /* continue */ }
15       x => bail!("Cannot roll back: {:?}", x),
16     }
17   }
18 }
```

Abbildung 5.16: Vereinfachter Auszug aus der Implementierung des Zurückrollens in Smash

Abbildung 5.16 ist ein vereinfachter Auszug aus dem Code zum Zurückrollen in Smash. Ich habe Smash in dieser Arbeit der Einfachheit halber nur linear Zustandsübergänge ablaufen lassen, weshalb mit einer Schleife in Zeile 2 die rückgängig zu machenden Übergänge durchlaufen werden. In Zeile 4 und davor werden die Zustandsstruktur und die Laufzeit der Komponente

geholt. Wie in Abschnitt 5.2.2 erklärt, wird hier ein Zurückrollen eingeleitet und die Komponente prüft, ob der Übergang gültig ist.

In Zeile 8 wird an die Koordinationsroutine der Laufzeitumgebung eine weiterer Übergang gereicht, hier allerdings einer zum Zurückrollen. `self.recv` ist die Tokio-Pipe, die beim Starten der Laufzeit erzeugt wurde, um von Komponenten Nachrichten zu empfangen.

Wenn es einen Fehler bei der Aktivierung gibt, wird dieser geloggt. Sobald alle durchlaufenen Schritte ein Pendant beim Zurückrollen hatten, wird Smash mit einem Fehlercode gestoppt.

### 5.2.5 Interaktion

Es gibt, wie in Abbildung 5.8 erläutert, verschiedene Arten von Änderungen: neben „normalen“Änderungen, die bei der Aktivierung ohne weitere Interaktion durchgeführt werden, gibt es manche Änderungen, die potentiell gefährlich sind und eine Bestätigung brauchen. Dabei wird unterschieden zwischen Änderungen, die rein optional sind – *NeedsConfirmationSkip* – und Änderungen, die notwendig sind – *NeedsConfirmationAbort*.

Optionale Änderungen, die eine Bestätigung brauchen, werden, falls diese nicht erteilt wird, schlicht ignoriert. Bei benötigten Änderungen sorgt dies für eine Verweigerung der Aktivierung. Die Bestätigung erfolgt interaktiv und wird vom Controller für jede erhaltene Änderung erfragt, die einen der zwei o.g. Typen hat.

Während dies für Bordmittel wie `nixos-rebuild` möglich ist, gibt es viele andere Werkzeuge, die keine Interaktion während der Aktivierung zulassen, z.B. `colmena` [107]. Für diesen Fall gibt es aktuell die Option `--always-yes`, welche es ermöglicht, alle Änderungsanfragen mit Ja zu beantworten. Derzeit wird nach dieser Regel verfahren, wenn `switch-to-configuration` auf `stderr` kein eigenes TTY hat. Damit ist zumindest sichergestellt, dass existierende Werkzeuge nicht mit Smash brechen. Komplexere Regeln, nach denen die Bestätigungen automatisiert werden, gibt es derzeit nicht.

### 5.2.6 Wiederbelebung von Zuständen

Es gibt bestimmte Aspekte für Zustände, die über das Entfernen der Konfiguration hinaus von Smash gespeichert werden müssen, vgl. Abschnitt 4.1 zu *Reaktivierung*.

Grundsätzlich ist diese Funktionalität immer dann relevant, wenn ein bestimmter Zustand nur teilweise von Smash verwaltet wird. Damit soll sichergestellt werden, dass nur von Smash verwaltete Elemente gelöscht werden. Wenn diese aus der Konfiguration entfernt werden. Ein Anwendungsfall wäre,

dass Datenbanken und deren Inhaber nur gelöscht werden, wenn diese über eine NixOS-Konfiguration mit Smash verwaltet werden. Manuell angelegte Daten werden hierbei nicht gelöscht.

```
#[derive(Debug)]
pub enum Error<T> {
  NotADir,
  DoesntExist,
  IoError(T),
  Serde(SerdeError),
}
pub trait LongtermMemory<V> {
  fn upsert<T: serde::Serialize>(
    &self, identifier: &str, data: &T,
  ) -> impl Future<Output = Result<(), Error<V>>>;
  fn read<T: serde::de::DeserializeOwned>(
    &self, identifier: &str,
  ) -> impl Future<Output = Result<T, Error<V>>>;
  fn delete(&self, identifier: &str) -> impl Future<
    Output = Result<(), Error<V>>
  >;
  fn exists(&self, identifier: &str) -> impl Future<
    Output = Result<bool, Error<V>>
  >;
}
```

Abbildung 5.17: Trait für Speicherung zur Reaktivierung

Abbildung 5.17 zeigt das Trait, das ich in Smash benutze für das autarke Speichern von Informationen über Zustände. Die Daten werden mit Serde [102] serialisiert bzw. deserialisiert. Aktuell existiert nur eine Implementierung, die diese Informationen im Dateisystem ablegt.

Da Komponenten – von den Annahmen von Varlink abgesehen – keine Beschränkungen in ihrer Struktur haben, ist aktuell diese Schnittstelle ein Implementierungsdetail.

## 5.3 vorhandene Komponenten

Um das Laufzeitverhalten von Smash zu untersuchen, sind Komponenten nötig. Ich habe zum einen die unten beschriebenen Komponenten vom Typ

*Upgrade* implementiert, welche die Grundlage für die Aktivierung von NixOS-Systemen bilden.

Zusätzlich gibt es noch einige besondere Komponenten, z.B. dediziert fürs Testing – vgl. Abschnitt 5.4 – oder Komponenten, die explizit für die Evaluation von Artefakten gedacht sind – vgl. Abschnitt 6.

Zusätzlich gibt es die in Abschnitt 5.1.2 beschriebene Komponente, die das Verhalten von `switch-to-configuration` eingebettet hat.

Eine weitere Komponente – genannt `zfs` – ist vom Typ *Upgrade*. Diese erzeugt von einer im Payload konfigurierbaren Menge an ZFS [5]-Datasets Snapshots und rollt auf diese im Übergang *Checkpoint* → *Rollback* zurück.

Auf diese Weise werden Änderungen, die z.B. durch ein Update einer Software in der fehlgeschlagenen Aktivierung durchgeführt worden sind, rückgängig gemacht. Allgemein ist dadurch sichergestellt, dass der exakte Zustand von vor dem fehlgeschlagenen Upgrade wiederhergestellt ist.

## 5.4 Tests

Das Zusammenspiel bei Aktivierung und Zurückrollen von Komponenten, sowie den Diensten selbst ist komplex. Um Regressionen bei größeren Änderungen zu vermeiden, wird eine Strategie benötigt, um automatisiert möglichst viele Szenarien zu testen.

Dies wird aufbauend auf dem NixOS Test-Framework [105] realisiert, welches es ermöglicht, innerhalb eines Bauvorgangs ein Netz an virtuellen Maschinen mit NixOS-Konfigurationen zu starten und mit einem Skript Tests durchzuführen. Wie von van der Burg et al. festgestellt, ist dieser Ansatz zum einen schnell genug, um die Tests kontinuierlich für Änderungen laufen zu lassen und gleichzeitig erleichtern diese die Vorbereitung großer Testumgebungen [105]. Tatsächlich werden auf Basis dieses Frameworks bis heute Tests in NixOS implementiert.

Für Smash wurde u.a. ein Test implementiert, welcher neben einer NixOS-Konfiguration für verschiedene Fälle sog. Spezialisierungen [81] existieren, welche die Konfigurationen, welche innerhalb des Tests aktiviert werden sollen, repräsentieren. Auf diese Weise können verschiedenen Szenarien bei diesem Aktivierungsvorgang getestet werden.

Spezialisierungen sind Module, welche eine bestehende Konfiguration erweitern, wie hier in Zeile 10 gezeigt. Diese werden im im Storepfad des NixOS-Systems im Unterverzeichnis `specialisation/<name>` abgelegt. Ab Zeile 13 ist eine Hilfsfunktion für die Tests implementiert, welche die Testmaschine auf eine Spezialisierung wechselt.

```
1  pkgs.testers.runNixOSTest {
2    name = "smash-simple";
3    nodes.machines = {
4      imports = [ ./smash-module.nix ];
5      system.state = {
6        enable = true; loglevel = "trace"; };
7        upgraders.snapshot.zfs.datasets = [ "tank/test" ];
8      };
9      boot.supportedFileSystems = [ "zfs" ];
10     specialisation.name.configuration = { /* ... */ };
11   };
12   testScript = { nodes, ... }:
13   let machine = nodes.machine.system.build.toplevel; in ''
14     def switch_specialisation(name, succeed=True):
15         run = getattr(
16           machine, 'succeed' if succeed else 'fail'
17         )
18         run(
19             f"${machine}/specialisation/{name}"
20             + "/bin/switch-to-configuration switch"
21         )
22   '';
23 }
```

Abbildung 5.18: Basiskonfiguration für die Integrationstests

Einzelne Testfälle werden ebenfalls im `testScript` implementiert. Abbildung 5.19 zeigt einen Testfall, welcher lediglich die korrekte Funktionalität von `switch-to-configuration` verifiziert.

Es wird also geprüft, dass der SSH Daemon korrekt gestartet ist, dass eine Datei mit Namen `foo` in `/etc` abgelegt wurde und dass ein Snapshot des Dateisystems durchgeführt wurde. Der Einfachheit halber habe ich die Vorbedingungen – z.B. dass SSHd vorher nicht lief und dass korrekt geloggt wird – ausgelassen.

Der Test aus Abbildung 5.20 prüft das Verhalten beim Zurückrollen: dafür habe ich eine Komponente vom Typ *Service* geschrieben, welche im Übergang *Inactive* → *Active* die Dateien löscht, die im Payload mitgegeben wurden – Zeile 9 und im Anschluss im Zuge des Übergangs einen Fehler auslöst, sodass das Zurückrollen von *Smash* eingeleitet wird.

In Abbildung 5.18 wurde in Zeile 7 definiert, welche ZFS [5] Datasets

```
{
  specialisation = {
    simple.configuration = {
      services.openssh = {
        enable = true;
      };
      environment.etc = {
        foo.text = "hello";
      };
    };
  };
}
```

```
with subtest("simple"):
  switch_specialisation(
    "simple"
  )
  machine.succeed(
    "test -e /etc/foo"
  )
  machine.require_unit_state(
    "sshd.service", "active"
  )
  machine.wait_for_open_port(22)
  assert 1 == len("\n".split(
    machine.succeed(
      "zfs list -t snapshot -H"
    )
  ))
```

Abbildung 5.19: Testfall, dass Smash NixOS-Konfigurationen korrekt aktiviert

einen Snapshot bekommen sollen, auf den im Fehlerfall zurückgerollt wird. Der Einfachheit halber habe ich mich hierbei darauf beschränkt, ein Dateisystem unter `/mnt` zu sichern. Die Behauptungen links stellen sicher, dass die gelöschten Dateien nach der fehlgeschlagenen Aktivierung wieder existieren und den erwarteten Inhalt haben. Nachdem im vorherigen Testfall – vgl. Abbildung 5.19 – SSHd aktiviert wurde und auf diese Konfiguration zurückgerollt wird, wird außerdem sichergestellt, dass dieser wieder aktiv ist. Ein weiterer Test stellt sicher, dass mögliche Abkürzungen wie in Abschnitt 4.2.4 beschrieben, auch genommen werden.

Insgesamt ist dieser Aufbau sehr hilfreich, um in der Entwicklung Regressionen zu entdecken, die alternativ nur mit aufwändigen manuellen Tests erkennbar wären. Insbesondere erwarte ich, dass derartige Tests auch dabei helfen zu verifizieren, dass Komponenten auch mit z.B. neueren Versionen eines Systemdienstes arbeiten. Ein letzter Vorteil ist, dass damit diverse Fehlerfälle testbar sind, welche manuell nur schwierig zu prüfen sind.

```nix
# specialisation.fail
{ pkgs, ... }: {
  system.state.components={
    service.demo = {
      implementation =
        pkgs.smash
        + "/bin/failing";
      payload = {
        paths_to_delete = [
          "/mnt/foo"
          "/mnt/bar"
        ];
      };
    };
  };
}
```

```python
ma = machine
with subtest("rollback"):
  switch_specialisation(
    "fail", False
  )
  assert (
    "hello world" == ma.succeed(
      "cat /mnt/foo"
    ).strip()
  )
  assert (
    "bar" == ma.succeed(
      "cat /mnt/bar"
    ).strip()
  )
  machine.require_unit_state(
    "sshd.service", "active"
  )
```

Abbildung 5.20: Testfall, dass Smash die Konfiguraiton und das Dateisystem zurückrollt.

# Kapitel 6

# Evaluation

In diesem Abschnitt beschäftige ich mich mit der Evaluation von Smash: es ist noch ein sehr junges Projekt und entsprechend experimentell und nicht für den produktiven Gebrauch geeignet. Hier zeige ich, dass Smash dennoch ein Weg nach vorne ist, indem ich zwei Einreichungen mit – im Vergleich zu den Einreichungen aus Kapitel 3 – stärkeren Anforderungen an Zustandsverwaltung evaluiere.

Zum einen soll gezeigt werden, dass leicht möglich ist, die gesamte Umgebung mit Code abzubilden und schnell in Betrieb zu nehmen. Sobald die Evaluationsumgebung mit Nix- und Smashkonfiguration abgebildet ist, soll es ausreichen, mit einem Deploymentbefehl die Umgebung in Betrieb zu nehmen. Dabei bilde ich mit den Bordmitteln von NixOS den Aspekt der Konfigurationsverwaltung ab und die Zustandsverwaltung mit Smash. Es soll gezeigt werden, dass leicht zwischen verschiedenen Umgebungen gewechselt werden kann, die Zustandsverwaltung als Teil des Deployments erledigt wird und diese konvergent nach Traugott et al. [103] ist.

Andererseits ist es auch wichtig, dass Smash kein Rückschritt ist: es soll möglichst einfach in bestehende Deployments integrierbar sein und die bisherigen Aufgaben zur Aktivierung – bisher von `switch-to-configuration` [87] durchgeführt – übernehmen können.

## 6.1 Meces

In „Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems“ schlagen Gu et al. ein verbessertes Verfahren zur Umskalierung von Datenpipelines vor [59].

Unter Pipelines werden hier Verarbeitungsabläufe nach dem *MapReduce*-Ansatz [51] bezeichnet. Dabei werden Aufgaben in sog. *Map*-Schritte un-

terteilt, wo mit möglichst viel Parallelität einzelne Datensätze verarbeitet werden und darauffolgend in den *Reduce*-Schritten aggregiert werden. Diesem Modell folgend gibt es Dienste, mit welchen mit hoher Parallelität und hohem Durchsatz auf diese Weise Daten verarbeitet werden.

In der Praxis ist es häufig so, dass einzelne Schritte – in der Einreichung Operatoren genannt – einen Zustand haben: ein triviales Beispiel ist, dass jeder *Reduce*-Operator bereits den vorherigen Zustand gespeichert haben muss. Bei der Evaluation solcher Umgebungen wird häufig mit dem Wörteranzahl-Beispiel gearbeitet, wo eine Menge an Dokumenten an *Map*-Operatoren gegeben wird, die das Dokument in einzelne Wörter auftrennen und die *Reduce*-Operatoren zählen dann alle aus der Pipeline empfangenen Wörter zusammen, wodurch der Zählerstand ein Zustand des Operators ist [51, 59].

Dies führt allerdings zu folgendem Problem: da die Operatoren auf beliebig viele Maschinen zu Skalierungszwecken verteilt werden können, muss es auch möglich sein, diese umzuskalieren. Das ist relevant, wenn z.B. eine Maschine für eine Systemwartung außer Betrieb genommen werden soll oder wenn zum Hochskalieren weitere Maschinen in das Cluster mit aufgenommen werden, um so den Durchsatz zu steigern. Das Umverteilen von Operatoren bedeutet hier auch, dass die jeweiligen Zustände mit umverteilt werden müssen [59].

Bisherige Ansätze erhöhen signifikant die Latenz: im einfachsten Fall wird das gesamte Cluster zum Neuskalieren gestoppt. Ein etwas effizienterer Ansatz ist es, nur die Operatoren auszuschalten, die von der Neuskalierung betroffen sind. Das kann dennoch die Latenz signifikant steigern, wenn davon z.B. ein zentraler *Reduce*-Operator betroffen ist und somit effektiv der Rest der Pipeline zum erliegen kommt [59].

In dieser Einreichung wird Meces vorgeschlagen, ein Verfahren, das Neuskalierungen effizienter umsetzen kann. Ein Teil des Problems ist, dass die jeweiligen Instanzen die Zustände eines Operators von sich aus an die neu zuständigen Instanzen senden. Mit Meces ist es auch möglich, dass die neu zuständigen Operatoren bei Bedarf Teile des Zustandes bei den alten Operatoren anfragen können, um so die Wartezeiten für Elemente in der Pipeline zu minimieren und somit sicherzustellen, dass die am meisten benutzten Teile des Zustandes eines Operators als erstes umverteilt werden [59].

Um sicherzustellen, dass keine Daten mehr zu den vorherigen Operatoren übertragen werden, wird eine Migrationsphase eingeleitet: dabei wird eine Steuerungsnachricht gesendet. Immer wenn diese einen Operator erreicht, wird überprüft, ob dieser das Routing seiner Nachrichten verändern muss. Außerdem leitet der Operator – falls nötig – eine Zustandsmigration ein. Sobald dieser damit fertig ist, wird das der Pipeline signalisiert. Sobald alle Operatoren diese Phase durchlaufen haben, ist die Migration beendet.

Die Evaluation dieser soll zeigen, dass die Latenzspitzen bisheriger Umverteilungsverfahren bei Meces nicht mehr existieren. Dazu wird eine Wörterzähl-Pipeline in einer modifizierten Version von Apache Flink [30], die das Meces-Verfahren implementiert, durchgeführt und die Latenz der Durchlaufzeit einzelner Nachrichten gemessen und verglichen.

### 6.1.1 Ausgangszustand der Einreichung

Die Reproduktion von dieser Einreichung war herausfordernd: der Aufbau der Evaluation war ein Verzeichnis mit verschiedenen Shell-Skripten. Dabei gab es ein Skript pro Versuch, wobei diese allerdings bis auf eine Variable Duplikate waren. Die eigentliche Interaktion mit Flink war allerdings in einem Skript, welches von allen Versuchsskripten aufgerufen wurde und dort mit If/Else-Strukturen wiederum die verschiedenen Fälle differenziert hat [58]. Zusätzlich vermute ich, dass während der Arbeit an der Einreichung von von einem Wörterzählverfahren auf das Zusammenzählen von zufällig generierten Zahlen gewechselt wurde: letzteres wird in den Beschreibungen so erläutert [59], allerdings sind noch einige ungenutzte Parameter in den Skripten übrig, die für ersteres benötigt worden sind [58]. Dies konnte ich erst nach Analyse des Codes bestätigen.

Diese Mischung aus Duplikaten und einem Shellskript, das dann trotzdem für alle Versuche mit Flink interagiert machte es sehr schwer für mich, die tatsächlich benutzten Parameter für die Versuche zu bestimmen. Zusätzlich waren einige der Variablen – u.a. `PARTIALPAUSE` – nicht definiert und obwohl in der Einreichung angegeben war, dass jede Pipeline 600s vor einer Neuskalierung läuft [59], waren es hier nur 120s. Wobei zu erwähnen ist, dass ich mit 600s Laufzeit ähnliche Ergebnisse wie in Abschnitt 6.1.4 erhalten habe.

Meine Interpretation ist, dass diese Einreichung ausschließlich für die drei o.g. Experimente als reproduzierbar gelistet wird, wobei dort laut Dokumentation eine fertig vorbereitete Umgebung zur Verfügung gestellt wurde [58].

Der Umstand, dass Flink keine guten Fehlermeldungen gibt, hat die Sache weiter erschwert: so wurden bei Fehlkonfigurationen bei einer Neuskalierung zwar direkt Exceptions geworfen, allerdings hatte das nur zur Folge, dass der Client wartete, bis ein Timeout erreicht war. Die Fehlermeldungen waren meist auch sehr knapp gehalten, in manchen Fällen wurden generische Java-Exceptions, wie `IllegalArgumentException` ohne Fehlermeldung geworfen.

### 6.1.2 Zustandsverwaltung mit Smash

Die zu zählenden Wörter sind Zufallszahlen, die in eine Nachrichtenwarteschlange, Apache Kafka [29] eingetragen werden. Die Flink-Pipeline konsu-

miert diese Nachrichten mit beliebiger Parallelität und zählt die Menge der Wörter zusammen.

Damit die Umgebung lauffähig ist, wird somit ein Kafka mit vorkonfiguriertem Topic benötigt, wofür sich der Einsatz einer Smash-Komponente anbietet. Die Implementierung der kongruenten Verwaltung von Topics ist bereits in Abbildung 5.15 enthalten. Zur Kommunikation mit Kafka wird *rdkafka* [49] benutzt. Mit diesem System ist sichergestellt, dass genau ein Topic mit der erwarteten Konfiguration existiert, im Vergleich zum Artefakt, wo bei Starten der Umgebung bedingungslos versucht wird, dasselbe Topic anzulegen [58], was allerdings fehlschlägt, wenn dieses bereits existiert.

### 6.1.3 Paketierte Komponenten & Deploymentintegration

Für die Evaluation werden zwei Java-Programme gebraucht: ein Dienst, der in Kafka Wörter für die Pipeline schreibt und die um Meces erweiterte Flink-Variante. Ersteres ist ein triviales Java-Paket, letzteres benötigte diverse Anpassungen.

Abbildung 6.1 enthält eine gekürzte Form des Builds. In Zeile 6 liegt ein Patch vor, der die Maven-Konfiguration massiv vereinfacht: viele Bestandteile – z.B. Bindings für Python – funktionieren in dieser Buildumgebung aufgrund weiterer fehlender Abhängigkeiten nicht und deshalb habe ich diese aus dem Build entfernt. In Zeile 8 befindet sich die Prüfsumme der Fixed-Output-Derivation innerhalb welcher Maven seine Abhängigkeiten herunterlädt, weshalb Netzwerkzugriff notwendig ist. Darunter werden noch Tests und Linter deaktiviert, um den Build weiter zu vereinfachen.

Das Maven ist nicht direkt das aus `nixpkgs` [80], sondern eine ältere Version, die zwingend von diesem Paket benötigt wird. Es benutzt Java 8 intern, was die Anforderung ist – und mit neueren Versionen schlägt die Kompilierung fehl – allerdings lief Flink bei mir nur mit Java 11 stabil.

Es werden Daten für das Meces-Verfahren, ein Neuskalierungsverfahren, das auf die Reihenfolge der zu migrierenden Zustände keine Rücksicht nimmt, sowie dem Verfahren, die gesamte Pipeline zu serialisieren und mit der neuen Konfiguration neuzustarten, getestet. Die verschiedenen Konfigurationen werden über mehrere Spezialisierungen [81] abgebildet, vgl. Abbildung 6.2.

Eine dieser Konfigurationen musste zusätzlich modifiziert werden, um für eine der Spezialisierungen einen Port zu starten, unter welchem eine REST API von Flink läuft, um die Umskalierung überhaupt zum Laufen zu kriegen.

Die Skripte im Artefakt selbst sind sehr stark von der Umgebung abhängig, d.h. von dem Ablageort der einzelnen Dienste und dem Betriebssy-

```
1  self: super: { meces = {
2    flink = self.callPackage ({ lib, fetchFromGitHub, }:
3      self.meces.maven.buildMavenPackage {
4        pname = "meces";
5        src = fetchFromGitHub { /* src params */ };
6        patches = [ ./flink-maven.patch ];
7        sourceRoot = "source/src";
8        mvnHash = /* ... */;
9        mvnParameters = lib.escapeShellArgs [
10         "-Dmaven.test.skip=true"
11         "-Dcheckstyle.skip"
12       ];
13       installPhase = ''
14         mkdir -p $out
15         cp -a flink-dist/target/.../. $out/
16         for i in RescaleWordCount NexmarkQ{1..8}; do
17           cp flink-examples/.../target/...-"$i".jar $out/lib/
18         done
19         cp flink-connectors/... $out/lib/
20       '';
21     }
22   ) { };
23 }; }
```

Abbildung 6.1: Nix-Paketdefinition von Flink

tem [58].

Aus diesem Grund habe ich hier – genauso wie bereits in Kapitel 3 – ein eigenes Pythonskript geschrieben, welches die Testvorgänge automatisiert, wie in Abbildung 6.3 verkürzt dargestellt.

Das Flink-Paket enthält einige Beispiel-Jobs. Im Artefakt wurde ein weiterer hinzugefügt, um anhand des Wörterzählbeispiels Neuskalierungen zu untersuchen [58]. Diese sind global mit dem Flink-Paket aus Abbildung 6.1 installiert, weshalb auch diese – vgl. Zeile 2 – global verfügbar sind.

In Zeile 7 wird ein Kindprozess gestartet, welcher 20000 Wörter pro Sekunde mit einer Länge von sieben Zeichen in das Kafka-Topic schreibt, genau wie im Artefakt auch [58]. Dann wird der Flink-Job in Zeile 8 gestartet und für 120 Sekunden laufen gelassen, um die Latenz im Normalbetrieb zu bestimmen.

Danach wird eine Umskalierung in Zeile 10 vorgenommen, von vorher drei

```
1  { lib, ... }:
2  with lib; {
3    specialisation = genAttrs [ "meces" "order" ] (strategy: {
4      configuration = {
5        environment.etc = {
6          "flink-conf.yaml".source =
7            lib.mkForce ./files/${strategy}/flink-conf.yaml;
8          "meces.conf.prop".source =
9            lib.mkForce ./files/${strategy}/meces.conf.prop;
10       };
11     };
12   });
13 }
```

Abbildung 6.2: Spezialisierungen für verschiedene Meces-Konfigurationen

Zähloperatoren auf jetzt fünf. In Zeile 12 die Ergebnisse gesammelt und ein Graph daraus erzeugt. Das Skript wurde aus dem Artefakt [58] übernommen.

Das Deployment habe ich in derselben Struktur wie in Kapitel 3 implementiert, allerdings mit dem Unterschied, dass Smash in die Aktivierung von NixOS übernommen werden musste.

In Abbildung 6.4 setze ich Smash in den Zeilen 1 bis 9 auf. Ich habe mich bewusst für lokale Pfade entschieden, um schnellere Feedbackzyklen zu ermöglichen. Für produktive Benutzung gibt es in Nix allerdings diverse Werkzeuge zum importieren Nix-basierter Abhängigkeiten – z.B. Flakes [13] – welche trivialerweise stattdessen benutzt werden können.

Das Ersetzen von klassicher Aktivierung mit Smash beschränkt sich hierbei auf wenige Zeilen Code, womit Smash faktisch ein direkter Ersatz dafür ist.

### 6.1.4 Ergebnisse

Die Evaluation wurde von den Autoren auf einer vorbereiteten AWS EC2 Instanz durchgeführt, die auch dem Evaluationskomitee zur Verfügung gestellt wurde [58]. Diese verfügt über 64 GB RAM und 32 dedizierte CPU Kerne aus der dritten Xeon-Generation. Entsprechend ist die dort verfügbare Hardware stärker also die von mir benutzte Hardware, welches dieselbe wie in Abschnitt 3 ist.

```
def run_evaluations_for(name):
    jar = f"/run/current-system/.../" + \
        "flink-examples-streaming_2.11-1.12.0-{name}.jar"
    for i in ["restart", "order", "meces"]:
        run(["redis-cli","flushall"], check=True)
        switch_specialisation(i)
        with kafka_producer(20_000, 7):
            with run_job(jar, mk_args(...)) as job_id:
                sleep(120)
                rescale(i)
                sleep(60)
                collect()
run_evaluations_for("rescalewordcount")
```

Abbildung 6.3: Auszug aus dem Testskript für die Evaluation von Meces

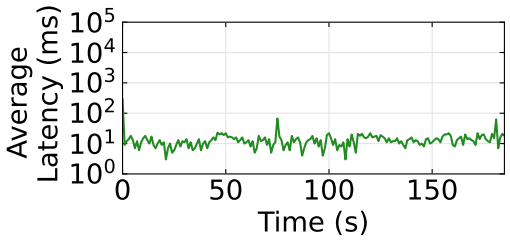


Abbildung 6.5: Latenz bei Neuskalierung mit Meces

Bei allen Messungen wurde nach 120 Sekunden eine Umskalierung eingeleitet mit einem Wechsel von drei auf fünf Zähloperatoren. Bei den bisherigen Verfahren zur Umskalierung – ohne Laden von wichtigen Zuständen bei Bedarf bzw. einem Neustart der Pipeline – zeigt sich in den Abbildungen 6.6 & 6.7 bei 120s deutlich eine Latenzspitze.

Bei der Benutzung von Meces zur Umskalierung zeigt sich keine ablesbare Latenz, die von der Umskalierung verursacht wurde. Damit ist die Kernaussage dieses Artefakts reproduziert [58].

Allerdings ist zu bedenken, dass die in der Einreichung beschriebene Lastspitze von 600ms [59], die zwar um das hundertfachte unter der Spitze eines Neustarts liegt aber trotzdem messbar sein sollte, nicht reproduzierbar ist.

Das Artefakt beschränkt sich auf diese Abbildung. Es enthält zwar den Code, um in Flink weitere Pipelines laufen zu lassen, allerdings werden diese von den Automatisierungsskripten nicht angesteuert und sind in der Dokumentation auch unerwähnt.

Weil unklar war, in welchem Format dort Informationen geloggt werden und wie daraus welche Abbildungen entstehen, habe ich diesen Schritt ausgelassen. Zusammengefasst lässt sich sagen: ich kann die Kernaussage zwar

```
{ imports = [
    /home/ma27/Projects/BA/statemgmt/nix/module.nix
  ];
  nixpkgs.overlays = [
    (self: super: {
      smash = self.callPackage
        /home/ma27/Projects/BA/statemgmt/package.nix { };
    })
  ];
  system.state = {
    enable = true;
    components.service.kafka = {
      implementation = "${pkgs.smash}/bin/kafka";
      payload = {
        bootstrap_server = "127.0.0.1:9092";
        topics.word_topic_par_24 =
          { replication_factor = 1; partitions = 1; };
      };
    };
  };
}
```

Abbildung 6.4: Auszug aus der NixOS-Konfiguration für Meces

reproduzieren, aber unter dem Vorbehalt, dass andere Charakteristiken aus der Einreichung nicht nachstellbar sind.

### 6.1.5 Fazit

Trotz der in Abschnitt 6.1.1 erläuterten Herausforderungen kann nachvollzogen werden, dass mit dem Mecesverfahren Umskalierungen deutlich weniger Latenz mit sich bringen. Bei der Umskalierung mit Meces war deutlich der Liveverfolgung des Logs anzusehen, dass statt drei jetzt fünf Nachrichten aus Kafka in einem Intervall verarbeitet werden.

Eine weitere Erkenntnis ist, dass sich Smash in das von mir vorher vorbereitete Deployment aus Apache Kafka [29] und dem modifizierten Flink aus dieser Einreichung gut integrieren ließ. Ich habe zuerst die einzelnen Bausteine ohne Smash-Integration vorbereitet und getestet und im Anschluss Smash mit eingebunden. Auf diese Weise zeigte sich auch, dass Smash gut in Bestandsumgebungen einbaubar ist.

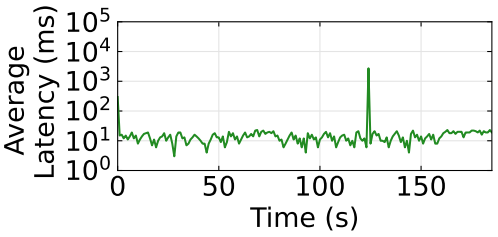


Abbildung 6.6: Latenz bei Neustart der Pipeline

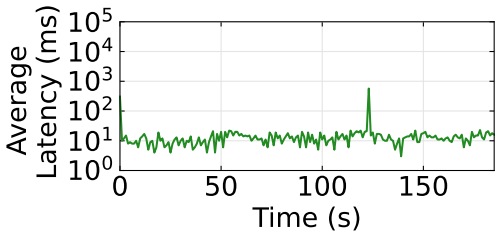


Abbildung 6.7: Latenz bei Neuskalierung der Pipeline ohne Berücksichtigung der Reihenfolge der Zustände

Die Zustandsverwaltung von Kafka-Topics hat den einen Teil, der nicht mit NixOS Bordmitteln abbildbar war, übernommen. Damit war es mir möglich, mit einem einzelnen Deployment eine Evaluationsumgebung für diese Einreichung aufzusetzen.

## 6.2 Moderne Hardwareisolationsmechanismen

In „Limitations and Opportunities of Modern Hardware Isolation Mechanisms“ untersuchen Chen et al. die Kosten von modernen Hardwareisolationsmechanismen, wie z.B. Pointerauthentifizierung [45].

Dazu zählen u.a. Intel MPK: dies benutzt vier bislang ungenutzte Bits aus der virtuellen Seitentabelle [45]. Diesen werden dann – analog zum UNIX-Rechteschema – Lese-, Schreib- und Ausführungsrechte zugewiesen. Mit dem Systemaufruf `pkey_mprotect` können dann einzelne Seiten im Speicher mit diesem Schlüssel geschützt werden, z.B. kann der Schreibzugriff eingeschränkt werden [3, 45].

Die tatsächlichen Berechtigungen werden über das lokale `pkru`-Register beschrieben, was den Vorteil hat, dass die gesamte Steuerung im Userspace geschehen kann [3, 45]. Anwendungen können dabei nur Lesezugriff haben und sich bei Bedarf Schreibzugriff geben [10]. Das erschwert es bereits, mögliche Lücken durch Speicherüberlauf auszunutzen. Vahldiek-Oberwagner et al. geben als weiteren möglichen Anwendungsfall die Ausführung einer Komponente an, die z.B. aufgrund von Fehlern bzw. Sicherheitslücken Speicherkorruption verursachen würde [104]. Eine Möglichkeit, wie verhindert werden kann, dass dann das `pkru`-Register in einer solchen nicht vertrauenswürdigen Umgebung wieder umgeschrieben wird ist das vorherige Umschreiben des Codes: dabei werden z.B. `wrpkru`-Instruktionen zum Beschreiben des Registers analysiert, ob Speicherbereiche betroffen sind, die nicht zu der aufgerufenen Komponente selbst gehören [104]. Auf diese Weise kann effektiv nicht ver-

trauenswürdiger Code im selben Adressraum ausgeführt werden. Chen et al. nennen als Alternative zum Umschreiben den Einsatz von Hardwarebreakpoints, um unerwünschte `wrpkru`-Instruktionen zu vermeiden [45].

Ein weiterer Vorteil ist, dass das Register threadspezifisch ist und nicht für den gesammten Prozess gilt, wie es mit dem `mprotect`-Systemaufruf der Fall wäre [3].

Zusätzlich werden vergleichbare Mechanismen auf anderen Architekturen beleuchtet, u.a. ARM MTE, welches eine Abbildung von 16 verschiedenen Tags auf Adressen beinhaltet, um auf diese Weise eine Zugriffskontrolle durchzuführen [45].

Neben Schutz des Speichers gibt es auch noch Mechanismen zum Schutz des Kontrollflusses, z.B. Intel CET [45], welches auf Hardwareebene Angriffe verhindern soll, bei denen die Rücksprungadresse ausgetauscht wird und damit der Kontrollfluss verändert wird. So wird über den *Shadowstack*, wo – hardwaregeschützt – die Rücksprungadresse ein weiteres Mal gespeichert wird. Im Falle von keiner Übereinstimmung gibt es auf Hardwareebene eine Exception und die Ausführung wird angehalten [45]. Bei Sprüngen wird auf sog. indirekte Zweigverfolgung gesetzt, d.h. vor jedem Sprung wird eine Instruktion, `endbr64`, generiert. Wenn diese vor einem Sprunt fehlt, wird ebenfalls die Anwendung angehalten [45].

Zur Evaluation der Mehrkosten dieser Verfahren testen Bi et al. diese auf verschiedenen Architekturen, wobei ich mich hier auf 64 Bit x86 beschränke. Dabei wurde ein LLVM [24] Pass implementiert, welcher u.a. den Schutz des Kontrollflusses implementiert. Zusätzlich wird ein Schema implementiert, das eine Anwendung in verschiedene Subsysteme unterteilt und nur einem Subsystem Zugriff über MPK-Tags gibt. Bei einem Wechsel des Kontrollflusses werden die Berechtigungen entsprechend angepasst [45].

Mit dieser LLVM wird die SPEC CPU 2006 [61] Benchmarking Suite – neben anderen, von mir hier nicht reproduzierten Benchmarks – ausgeführt, um so die Mehrkosten zu bestimmen.

### 6.2.1 Zustandsverwaltung mit Smash

Entscheidend für die Evaluation dieser Einreichung war, dass bestimmte Hardwareeinstellungen korrekt gesetzt sind. Chen et al. messen die Mehrkosten von Isolationsmechanismen anhand von verschiedenen Benchmarks [45], u.a. SPEC CPU 2006 [61]. Dabei würde die Varianz durch Hyperthreading die Ergebnisse verfälschen, weshalb dieses für die Durchführung ausgeschaltet sein muss.

Da insbesondere das Reaktivieren von Hyperthreading im Nachhinein bei laufendem Betrieb das System destabilisieren kann, sehe ich das als Anwen-

dungsfall für die *Behauptungen* – vgl. Abschnitt 4.1 – an.

```
1  let enabled = std::fs::read_to_string(
2    "/sys/devices/system/cpu/smt/active"
3  ).map_err(|e| proto_err!(/* reporting */))
4    .and_then(|val| {
5      let num = val.strip_suffix("\n").unwrap().parse::<u8>()
6        .map_err(|e| proto_err!(/* reporting */))?;
7      Ok(num == 1)
8    })?;
9  if enabled != self.desired_state {
10   proto_bail!(
11     format!(
12       "Expected hyperthreading to be {}, but it is {}",
13       if enabled { "enabled" } else { "disabled" },
14       if self.desired_state { "enabled" } else { "disabled" }
15     ),
16     Affected::StateTransition
17   )
18 }
```

Abbildung 6.8: Smash-Check für (in)aktives Hyperthreading

Abbildung 6.8 enthält den Code, den ich in der Smash-Komponente vom Typ *Check* dafür implementiert habe. Aus der NixOS-Konfiguration kann die Prüfung, ob Hyperthreading aktiv ist, mit der Deklaration der Konfigurationsoption `hardware.hyperthreading.enable = true;` durchgeführt werden. Mit `false` wird sichergestellt, dass die Konfiguration nur aktiviert wird, wenn Hyperthreading inaktiv ist.

Standardmäßig ist der Wert dieser Option `null`: in diesem Fall führt die Komponente keine Prüfung durch und jeder Wert wird akzeptiert.

Die Makros in u.a. Zeile 10 sind ähnlich wie die `bail!`/`err!` Makros, allerdings generieren die für Varlink – vgl. Abschnitt 5.2.1 – Fehlermeldungen.

Des Weiteren soll für diese Evaluation die minimale und maximale CPU-Frequenz auf einen konstanten Wert gesetzt werden und weitere Funktionalität, wie Turboboost deaktiviert werden. Dies habe ich mit einer *Service*-Komponente gelöst, die – wie in Abbildung 6.9 implementiert – im Übergang *Inactive* → *Upgrade* pro CPU-Kern für jeden Parameter wie folgt vorgeht:

- Wenn im Payload ein Wert angegeben ist, der nicht mit dem aktuellen Wert übereinstimmt, ändere diesen – ab Zeile 2.

- Falls dies zutrifft und kein Wert von Smash zur Reaktivierung gespeichert wurde, sichere den Ursprungswert.

- Wenn ein Ursprungswert gesichert wurde und kein Wert im Payload angegeben ist, setzen wieder den Ursprungswert.

Damit ist gegeben, dass wenn der Konfigurationswert für z.B. die maximale Frequenz nicht mehr in der NixOS-Konfiguration gesetzt ist, auf den Ursprungswert zurückgerollt wird.

```
if let Some(desired) = self.cpu_config.max_freq {
  if *max != desired {
    let change = Operation {
      path: path.clone(), value: desired,
      backup_value: if !backup.1.contains_key(cpu_name) {
        Some(*max)
      } else { None },
      affected: Element::MaxFreq { cpu: cpu_name.clone(), },
    };
    plan.push(change);
  }
} else if let Some(previous_value)
  = backup.1.get(cpu_name).map(|(_, max, _)| max)
{
  if *previous_value != *max {
    let change = Operation {
      path: path.clone(), value: *previous_value,
      backup_value: None,
      affected: Element::MaxFreq { cpu: cpu_name.clone(), },
    };
    plan.push(change);
  }
}
```

Abbildung 6.9: Smash-Komponente für die Angleichung der maximalen CPU-Frequenz

In dieser Komponente bin ich auch nach dem in Abschnitt 5.2.3 beschriebenem Muster vorgegangen, zur Zeit der Änderungsanalyse einen Plan zu erzeugen, welcher dann innerhalb der Zustandsübergänge durchgeführt wird.

Die Struktur von `Operation` ist hier allerdings eine leicht andere: diese Komponente setzt nicht nur die maximale, sondern pro Kern auch die minimale Frequenz und die C-States auf *C0*, was effektiv die Möglichkeiten zur Energieeinsparung der CPU deaktiviert.

Die Struktur, die u.a. in Zeile 13 entpackt wird, ist die, die für die Reaktivierung – vgl. Abschnitt 5.2.6 – gespeichert werden, um für andere Evaluationen wieder auf die Standardwerte der Maschine wechseln zu können. Der zweite Wert des Tupels ist eine Zuordnung von CPU Name – `cpu0` bis `cpuN` – zu einem Tripel mit minimaler & maximaler Frequenz und einem Flag, ob *C0* aktiv ist.

Anhand der `Operation` wird also zum einen der Pfad im *Sysfs* bestimmt, an den ein Wert – `value` – geschrieben wird und dessen vorheriger Wert – `backup_value` – gesichert wird.

Ich habe zur Evaluation dieser Einreichung SPEC [61], eine Benchmarking-Umgebung, betreiben müssen. Diese vermischt Daten und Code in einem Arbeitsverzeichnis. Noch dazu ist das eine kompliziert aufgebaute Perl-Codebasis von 2006, bei der das Zusammenspiel schwierig zu überblicken ist. Entsprechend bin ich zu dem Ergebnis gekommen, dass es nicht zielführend ist, dafür ein reines Paket zu erzeugen, sondern das ein Problem ist, das auch mit Zustandsverwaltung gelöst werden muss.

Daraus ist die Komponente `backed-up-env` entstanden mit folgendem Verhalten:

- Es gibt eine Basisumgebung, die idealerweise ein Storepfad ist.
- Es wird ein ZFS Dataset angelegt bzw. konfiguriert und in einen Präfix gemounted.
- Es werden Prüfsummen der Inhalte des Storepfades und des gemounteten Datasets erzeugt.
- Wenn diese übereinstimmen, tue nichts.
- Sonst erzeuge einen Snapshot des Datasets und setze das Verzeichnis auf den Inhalt des Storepfades.

Auf diese Weise kann eine Aktivierung genutzt werden, um auf einen sauberen Stand nach zurückzuwechseln. Gleichzeitig sind die Daten nicht verloren und können in `<mountpoint>/.zfs/snapshots` auslesen und weiterbenutzt werden.

Nachdem dieser Ansatz sehr disruptiv ist, wird das nicht ungefragt durchgeführt, sondern ist eine Änderung vom Typ *NeedsConfirmationSkip* nach Abschnitt 5.2.5.

### 6.2.2 Paketierte Komponenten

Der große Vorteil an dieser Einreichung war, dass einige Teile des Artefakts bereits rudimentär Nix unterstützt haben: so gab es sog. Entwicklungsumgebungen, bei denen eine Shell geöffnet wird mit allen Abhängigkeiten des Bauvorgangs, sowie Umgebungsvariablen für CMake-Einstellungen [44]. Auf diese Weise konnte ich einige Aspekte, z.B. die Builds für die LLVM Pakete, direkt übernehmen. Zu bemerken ist allerdings, dass es zwar die Paketierung erleichtert hat, aber sich auf derartige Entwicklungsumgebungen beschränkt hat und keine fertigen Pakete gab. Entsprechend war es mit dieser Einreichung nicht möglich, Vorteile, wie kontinuierliche Testbarkeit zu sich zu Nutze zu machen.

Zusätzlich scheint es so, als gäbe es dennoch leichte Unterschiede zwischen dem Bauen in einer Entwicklungsumgebung und dem Bauvorgang in einer Sandbox: so fehlten bei den Bauvorgängen u.a. diverse Headerdateien und am Ende hatte ein Programm Dateien aus der Nix Sandbox mit dem Präfix `/build` statt eines Storepfades in einem `RUNPATH` referenziert, was ich mit

```
{
  /* ... */

  preFixup = ''
    patchelf $out/bin/llvm-omp-device-info \
      --shrink-rpath --allowed-rpath-prefixes \
        '$ORIGIN:${placeholder "out"}:${gcc-prefix}'
  '';
}
```

beheben musste, wobei `placeholder "out"` eine Referenz auf den Outputpfad selbst ist und `gcc-prefix` eine von mir erzeugte Umgebung mit `libc` und GCC [99], welche alle benötigten Header enthielt.

Zusätzlich gab es einen Syntaxfehler in einer der LLVM Branches [18]. Wenn man diesen entfernt – für mich sah das stark nach einem Kommentar aus – schlug der Bauvorgang von LLVM selbst allerdings mit einem internen Compilerfehler fehl, weshalb ich dann entschieden habe, die dafür relevanten Tests auszulassen.

Auch wenn SPEC CPU 2006 [61] über Smash ausgerollt wird – vgl. Abschnitt 6.2.1, braucht das Paket etwas Aufbereitung mit Nix: die Idee ist hier, dass am Ende eine imperativ benutzbare Benchmarkingsuite unter `/var/lib/speccpu2006` liegt. Diese benötigt allerdings einige Patches, damit die funktioniert.

```
1  stdenv.mkDerivation (finalAttrs: {
2    pname = "speccpu";
3    version = "2006";
4    src = fetchurl { /* ... */ };
5    dontFixup = true;
6    patches = [ /* ... */ ];
7    postPatch = ''
8      ln -sf ${lib.getExe gnumake} bin/specmake
9      cp bin/lib/x86_64-linux/Safe.pm bin/lib/Safe.pm
10     rm -r bin/lib/x86_64-linux bin/lib/List bin/lib/Scalar
11     cp ${replaceVars ./static/linux_x86_mpk.cfg {
12       inherit glibc;
13     }} config/linux_x86_mpk.cfg
14     # usw.
15     patchelf --force-rpath \
16       --set-interpreter ${glibc}/lib/ld-linux-x86-64.so.2 \
17       --set-rpath ${glibc}/lib bin/specinvoke
18   '';
19   installPhase = "mkdir $out; cp -r . $out/";
20 })
```

Abbildung 6.10: Paket für die SPEC CPU 2006 Umgebung

Abbildung 6.10 enthält die Paketdefinition für SPEC CPU 2006. In Zeile 5 wird die `fixupPhase` deaktiviert, da diese versucht, u.a. Debugsymbole von Binärdateien zu entfernen. Nicht nur ist das hier irrelevant, es verlangsamt den Bauvorgang auch deutlich.

Zeile 6 enthält einige Patches, die die Syntax des Perlcodes anpassen: ich benutze zwar die Perlumgebung aus dem Artefakt [44], allerdings habe ich dennoch an einigen Stellen Syntaxfehler von Perl bekommen. Anscheinend haben sich im Laufe der Zeit einige Regeln zum Setzen von Klammern geändert, die sich aber trivial lösen ließen. Zusätzlich habe ich den *Safe*-Modus von Perl, der nur eine eingeschränkte Interpreterumgebung zur Verfügung stellt, nicht in Betrieb nehmen können und stattdessen deaktiviert. Nachdem die Benchmarks auf einer Maschine laufen, die ausschließlich dafür da ist, habe ich entschieden, diese Stellen komplett zu entfernen.

Ab Zeile 11 kopiere ich die Konfigurationsdateien für SPEC aus dem Artefakt [44] in das korrekte Verzeichnis, sodass diese erkannt werden. Ich habe dort des Weiteren den Pfad zur LLVM/Clang-Installation durch das Schema `/etc/llvm-NAME` abgeändert, sodass ich, wie in Abbildung 6.11 implemen-

tiert, die LLVM-Installationen verfügbar machen konnte.

```
1  { pkgs, ... }: {
2    environment.etc = {
3      "dev-env".source = pkgs.mars-hw-isolation.perl-env.drvPath;
4
5      "llvm-mpk".source = pkgs.mars-hw-isolation.llvm-clang-mpk;
6      "llvm-c".source = pkgs.mars-hw-isolation.llvm-clang-c;
7      "llvm-nacl".source = pkgs.mars-hw-isolation.llvm-clang-nacl;
8      "llvm-gs".source = pkgs.mars-hw-isolation.llvm-clang-segue;
9    };
10 }
```

Abbildung 6.11: LLVM-Installationen für SPEC

In Zeile 3 kopiere ich die Derivation einer Entwicklungsumgebung. Mit dieser ist es möglich, über `nix-shell /etc/dev-env` eine Umgebung zu öffnen, in welcher u.a. die Perl-Installation aus dem Artefakt [44] verfügbar ist.

In Abbildung 6.10 waren zusätzlich noch folgende Anpassungen notwendig:

- *specinvoke* ist ein ELF-Programm, welches beim Starten eines Benchmarks aufgerufen wird. Nachdem ich keine Quelldateien dafür finden konnte und es nur eine `glibc` linkt, habe ich mit `patchelf` [52] den `rpath` hiervon editiert. `glibc` ist rückwärtskompatibel, daher ist das hier ohne Weiteres machbar.

- *specmake* ist ein kompiliertes GNU Make [100]. Dieses habe ich schlicht durch einen Symlink zu einem neueren Make aus `nixpkgs` [80] ersetzt.

Nach Ausrollen dieser Konfiguration können Benchmarks dann mit einem Aufruf von `bin/runspec --config=... --iterations 1 tests...` aus der Entwicklungsumgebung heraus ausgeführt werden.

### 6.2.3 Ergebnisse

Ich habe mich darauf beschränkt, SPEC CPU 2006 [61] zu benutzen und keine weiteren Benchmarks mit der Version von 2017 durchgeführt. Außerdem habe ich mit der bereits in Abschnitt 6.1.4 verwendeten Hardware gearbeitet, entsprechend beschränken sich die Ergebnisse auf die x86 Hardware.

| | nativ | MPK | GS |
|---|---|---|---|
| 401.bzip2 | 1120 s | 1120 s | 1130 s |
| 429.mcf | 420 s | 433 s | 445 s |
| 462.libquantum | 413 s | 415 s | 442 s |

Abbildung 6.12: Ergebnisse der Reproduktion

Wie auch in der Einreichung, habe ich verschiedene Benchmarks aus SPEC CPU 2006 mit folgenden Konfigurationen laufen lassen: MPK meint hier die in Abschnitt 6.2 beschriebene LLVM-Variante mit CET und Intel MPK. Wie auch in der Einreichung erläutert, zeigt sich hier, dass die MPK-Variante nur sehr minimale Mehrkosten mit sich bringt, bei `462.libquantum` liegt das auch noch im angegebenen Bereich von 0,4% [45].

Interessanterweise habe ich auch Läufe gehabt, wo die MPK-basierten Benchmarks wenige Sekunden schneller waren. Damit lässt sich dennoch sagen, dass die Geschwindigkeit äquivalent ist.

Leider enthält die Einreichung kein Skript zur Erzeugung der tatsächlichen Abbildungen, weshalb ich nur einen Auszug der Ergebnisse in Tabellenform darstelle. Ich vermute stark, dass das auch der Grund ist, weshalb diese Einreichung kein *reproduziert*-Label hat.

In der Einreichung gab es noch eine Variante mit einem auf NaCl [56], um zu zeigen, dass rein softwarebasiertes Sandboxing einen messbaren Nachteil gegenüber Mechanismen mit Hardwareunterstützung haben. Das konnte ich nicht reproduzieren, da die dafür notwendigen LLVM Patches mit einem internen Compilerfehler fehlgeschlagen sind.

Zusätzlich wurden die Benchmarks mit einer Reimplementierung von Segue [76] in LLVM erneut laufen gelassen, wobei als Mehrkosten ca. 4% Performanceinbußen angegeben worden sind [45]. Eine Vergrößerung in dieser Größenordnung war auch hier nachvollziehbar.

### 6.2.4 Fazit

Aus meiner Sicht zeigt diese Einreichung bereits, dass Nix – selbst in rudimentärer Ausführung – eine Hilfe ist: damit war bereits explizit definiert, welche Abhängigkeiten gebraucht werden. Insbesondere die Installation des älteren Perls, das – ich vermute gerade so – mit SPEC CPU 2006 kompatibel ist, war sehr wertvoll. Das gilt nicht nur für potentielle Prüfer, sondern auch für die Autoren: die Perl-Pakete, die hier benutzt wurden sind von ca. 2017 und sind nach wie vor im Binarycache unter `cache.nixos.org` verfügbar. Das macht es wesentlich angenehmer, mit älterer Software zu arbeiten, falls

ein solcher Bedarf bei Evaluationen besteht.

Gleichzeitig zeigt sich hier aber auch sehr deutlich, wie leicht sich Fehler – z.B. der Syntaxfehler in einer der LLVM Branches – einschleichen können und ich glaube, wenn man solche Umgebungen bzw. zumindest das Einrichten dieser automatisiert testen könnte, dann wäre es schwerer, dass ein solcher Fehler übersehen wird.

Das in Abschnitt 6.2.1 von mir entwickelte Vorgehen zur Verwaltung von SPEC CPU 2006 ist aus meiner Sicht eine interessante und neue Möglichkeit, um Umgebungen zu verwalten, bei denen Code und Daten nicht getrennt sind, wobei dieser Ansatz nur dann sinnvoll ist, wenn es um solche Evaluationsumgebungen geht. Es erlaubt, gleichzeitig mit Nix solche Umgebungen vorzubereiten – vgl. Abschnitt 6.2.2 – sowie trivial auf einen sauberen Zustand zurückzuwechseln, ohne dabei Ergebnisse verwerfen zu müssen.

Dieser Ansatz bietet sich allerdings nur auf einem Copy-on-Write Dateisystem wie ZFS an, wo nicht jeder Snapshot zu einer Doppelung aller Dateien führen würde. Leider ist die aktuelle Implementierung noch etwas unergonomisch: wenn man iterativ Konfigurationen deployen muss, wird jedes Mal der Zustand zurückgesetzt. Mögliche Lösungen wären entweder, mehr Deploymentwerkzeugen wie `colmena` [107] so umzubauen, dass Interaktion möglich ist und diese – wie Smash selbst – nach Bestätigung fragen können, ob eine solche Operation durchgeführt werden soll. Aktuell ist das inaktiv, wenn das Deploymentwerkzeug Smash kein TTY zur Verfügung stellt.

# Kapitel 7

# Diskussion

In diesem Abschnitt lasse ich die Ergebnisse dieser Arbeit Revue passieren: wie lassen sich die Erkenntnisse aus Smash und seiner Architektur in die Artefaktevaluation eingliedern? In wieweit lassen sich die gefunden Probleme damit lösen? Ist es möglich, Smash in das bestehende NixOS-Projekt zu integrieren?

## 7.1 Über den Einsatz von Smash und NixOS

Nach über acht Jahren Erfahrung mit Nix(OS) ist mein Eindruck, dass Nix u.a. deshalb so schwer zu erlenen ist, weil es die Komplexität einzelner Buildsysteme sichtbar macht. Mein Fazit aus den letzten Kapiteln ist, dass daran kein realistischer Weg vorbeiführt: dadurch, dass bestimmte Abhängigkeiten „versteckt“ werden, kommt es häufig vor, dass diese zwar bei der Entwicklung erfüllt sind, aber aufgrund einer veränderten Umgebung einige Jahre später zu Problemen führen, wie man in allen von mir durchgeführten Evaluationen sieht. Das hat zur Folge, dass teilweise ganze Maschinen zur Verfügung gestellt werden, was allerdings, wenn im Zuge der Evaluation auch die Implementierung im Detail mit untersucht werden soll, zu einem Problem wird. Beispielsweise weil keine Patches anwendbar sind und allgemein der Review von Binärdateien deutlich schwieriger ist.

Für mich ist die Möglichkeit, schnell zwischen einzelnen Umgebungen auf einer Maschine zu wechseln, weil diese Umgebungen in Code formuliert sind und deren Bauvorgänge automatisiert zu testen gute Gründe, um in Evaluationsumgebungen verstärkt auf Nix zu setzen.

Allerdings ist das keine leichte Veränderung: die erste große Hürde dafür ist die Verbreitung: Nix wurde in den letzten Jahren zwar wesentlich populärer und hat damit auch eine größere Community, allerdings ist der

Aufwand, sich das hierfür notwendige Wissen anzueignen und ggf. Probleme zu lösen, nicht zu unterschätzen. Wenn Interesse an solchen Lösungen besteht, wäre es meiner Meinung nach auch im Interesse der Evaluationskomitees, ggf. Unterstützung und Vorlagen für solche Projekte zur Verfügung zu stellen. Außerdem gibt es nach wie vor viele unterdokumentierte Bereiche oder Themen, die schwer zu finden sind.

Das gilt auch für Smash selbst: wenn es nur von einer kleinen Gruppe benutzt wird, halte ich die Verbreitung für unrealistisch, da Komponenten für eine Vielzahl von Programmen gewartet werden müsste. Auf lange Sicht braucht das also mehr Verbreitung, damit die Komponenten sich weiterentwickeln und es genug Personal zur Pflege bestehender Komponenten gibt.

Smash lässt sich gut in bestehende Deployments integrieren – vgl. Abschnitt 6.1.3 und wäre damit ohne großen Aufwand geeignet, um durch *Dogfooding* weiterentwickelt zu werden. Aktuell gibt es nur außerhalb von NixOS Lösungen, um z.B. automatisiert im Fehlerfall zurückzurollen [94], da Smash letztlich die Aktivierung bei gleichbleibender Schnittstelle austauscht, kann dies in beliebige Installationen eingeführt werden.

Ich habe gezeigt, dass Zustandsverwaltung, sowie das Zurückrollen damit grundsätzlich möglich ist. Wie praktikabel es für weitere Anwendungsfälle ist, muss im Nachgang weiter untersucht werden. Zusätzlich hat Smash einige scharfe Kanten, die es ungeeignet für den Produktivbetrieb machen, siehe auch Kapitel 8 dazu.

Ein weiterer Punkt ist, dass Smash aktuell zwar durch den Einsatz von Varlink [93] auch mit Komponenten auf anderen Maschinen kommunizieren könnte, allerdings gibt es keine Unterstützung dafür. Mehr Koordination wäre für Umgebungen interessant, die zwingend mehrere Maschinen benötigen, ist aber nicht Teil des Umfangs von Smash, wie es hier präsentiert wurde.

### 7.1.1 Empfehlungen an die Implementierung

Des Weiteren gab es bei Entwurf und Implementierung noch folgende Beobachtungen, welche potentiell relevant für alternative Implementierungen sind:

- Es erscheint sinnvoll, Komponenten in eigene Prozesse auszulagern und mit IPC zu arbeiten: zum einen ermöglicht das die freie Wahl der Programmiersprache, was mehr Raum für Innovation zulässt und ermöglicht, die Sprache zu wählen, die am besten für eine Komponente geeignet ist.

  Zum anderen gibt das die Möglichkeit, auf schwere Abstürze einzelner Komponenten zu reagieren: wenn z.B. die C-Bindings eines Dienstes

verwendet werden und einen Segmentierfehler verursachen, bricht das nicht die gesamte Aktivierung, sondern nur die einzelne Komponente und es ist immer noch möglich, Rollbacks durchzuführen.

- Es ist sinnvoll, große Komponenten zu bauen. Bislang erscheint eine Komponente pro Subsystem eine gute Wahl.

  Dies liegt u.a. daran, dass das Ausführen von Komponenten potentiell teuer (s.o.) ist und zu kleine Komponenten die Nachvollziehbarkeit des Modells erschweren können.

- Es sollte nur das in Komponenten abgebildet werden, was tatsächlich nötig ist: z.B. reicht es bei einem Dateisystem, das Anlegen und Konfigurieren dort zu erledigen. NixOS bietet bereits ein Modul, das sicherstellt, dass Dateisysteme zum richtigen Zeitpunkt eingehängt werden. Hier muss nur sichergestellt werden, dass die Reihenfolge der Komponenten richtig ist.

- Es braucht – wie in Smash auch – einen Dienst, der bei einem Neustart bestimmte Komponenten, z.B. zur Einstellung der maximalen und minimalen CPU Frequenz aus Abschnitt 6.2.1 erneut ausführt.

  Das Betriebssystem setzt bei einem Neustart diese Einstellungen wieder zurück. Im Fall von Smash kann eine Komponente zum Start laufen gelassen werden, indem in der NixOS-Konfiguration mit der Option `rerunOnBoot = true;` gesetzt ist. Allerdings geschieht das ohne Abhängigkeiten, die nicht zur Startzeit erneut laufen. Entsprechend ist das eine Option, die mit Vorsicht zu benutzen ist.

  In Smash erfolgt dies in einem Systemd-Dienst, welcher startet, bevor alle verbliebenen Dienste gestartet werden, um zu vermeiden, dass potentiell Funktionalität mit unvollständiger Zustandsverwaltung gestartet wird.

  Ich sehe darin dennoch einen Vorteil gegenüber einem Neustart zum Zurücksetzen: mit dieser Lösung kann in sehr kurzer Zeit zwischen verschiedenen Konfigurationen mit ggf. verschiedenen CPU-Einstellungen gewechselt werden und diese auch auf die Standardwerte zurückgesetzt werden. All das ist möglich ohne Neustarts, die je nach Hard- und Firmware sehr lange dauern können und entsprechend die Feedbackzyklen deutlich verlängern. Um Konsistenz über Neustarts hinaus zu gewährleisten, ist o.g. Ansatz implementiert worden.

  Orthogonal dazu gibt es noch den Anwendungsfall, die Aktivierung für einen Neustart zu unterbrechen, was ich in Kapitel 8 noch einmal

aufgreife.

### 7.1.2 Nachteile

Ein Szenario, das aktuell nicht abgedeckt ist, ist dass die Überprüfung von Komponenten vom Typ *Check* nicht mehr stattfindet, wenn das System neugestartet wird und vorher in den BIOS-Einstellungen Hyperthreading wieder eingeschaltet wird. Meiner Meinung nach sollten Systeme nicht aufgrund von äußeren Umständen plötzlich nicht mehr starten. Wie dieser Umstand genau kommuniziert wird, ist von der aktuellen Version von Smash noch nicht abgedeckt und hängt letztendlich von er Art ab, wo die Evaluationsumgebung läuft: bei einem graphischen System wäre eine Benachrichtigung möglicherweise ausreichend, bei einem Knoten in einer Flotte von Maschinen sollte dieser Umstand im Monitoring auftauchen.

Außerdem ist die Implementierung von Komponenten derzeit noch mit sehr viel Codedoppelung verbunden. Das zeigt sich in Abschnitt 6.2.1 & Abbildung 5.15 beispielsweise: dort habe ich Code zur Annäherung an den gewünschten Zustand (Reconciliation) implementiert. Dabei ist das Muster immer sehr ähnlich: es wird geprüft, ob die Zustände sich unterscheiden und ein Ausführungsplan erzeugt. Wenn der Zustand nicht komplett kongruent behandelt werden soll – z.B. wenn nicht alle Kafka-Topics aus der Konfiguration kommen, sondern einige von anderen Anwendungen angelegt werden – muss noch ein Sicherungswert mit einbezogen werden, wie in Abbildung 6.2.1 und in Abschnitt 5.2.6 demonstriert. Dieses Vorgehen könnte wahrscheinlich in eine Bibliothek gekapselt werden. Ich habe das bewusst nicht in dieser Arbeit getan, weil ich erst einige solcher Implementierungen anfertigen wollte, um nicht am Ende mit falschen Abstraktionen dazustehen.

## 7.2 Zukünftige Chancen für Artefaktevaluation

Aktuell ist die Artefaktevaluation ein relativ unbedeutender Teil im Prozess der Veröffentlichung: das zeigt sich bereits daran, dass auf Listen im Internet, die von Evaluationskomitees gepflegt werden, nur ein Teil der jeweiligen Einreichungen überhaupt Artefakte hat und diese auch nicht alle reproduzierbar sind [22].

Zusätzlich scheinen die Anforderungen relativ gering zu sein: für USENIX ATC [28] bedeutet *funktional* nur, dass alle erforderlichen Schritte dokumentiert sein müssen, alle verwendeten Skripte existent und ersichtlich sein muss, dass diese zu einem verwertbaren Ergebnis führen [22]. Auch wenn für mich

*reproduzierbar* sich so ließt, als müssten *alle* Abbildungen nachstellbar sein, scheint dies faktisch nicht der Fall zu sein: für Meces [58] sind nur die Wörterzählexperimente dokumentiert, nicht aber die verbliebenen Abbildungen.

Meine Interpretation hierbei ist, dass das auch damit zusammenhängt, dass solche Umgebungen je nach Komplexität des darlegten Sachverhalts sehr schwierig nachzustellen sind. Die Beobachtung in dieser Arbeit ist aber, dass mit Nix solche Umgebungen wesentlich einfacher aufzusetzen sind, woraus sich für mich die Hoffnung ergibt, dass diesen Evaluationen mehr Relevanz beigemessen wird und die Artefakte leichter zu behandeln sind, wenn diese leichter durchzuführen sind und weniger Zeit & Energie investiert werden muss, um die Umgebung überhaupt lauffähig zu bekommen.

# Kapitel 8

# Weiterführende Arbeit

Ich habe in dieser Arbeit Smash implementiert in dem Versuch, eine Antwort auf das Problem der Zustandsverwaltung zu finden. Wie bereits in der Diskussion in Kapitel 7 erläutert, glaube ich, dass der Ansatz der Zustandsverwaltung nur dann eine Chance hat, wenn dieser Teil von NixOS selbst wird und nicht von Autoren oder Evaluationskomitees entwickelt und gewartet werden muss.

Das bedeutet auch, dass einige scharfe Kanten der Implementierung vorher beseitgt werden müssen und fehlende Funktionalität erarbeitet werden muss.

### Standardbibliothek für Komponenten

Wie bereits in Kapitel 7 angesprochen, geht die Implementierung von Komponenten derzeit mit sehr viel Duplikation einher. Eine Aufgabe, um Smash leichter benutzbar zu machen, ist es, herauszuarbeiten, welche Schritte wie am besten in eine Bibliothek integrierbar sind, um dies zu veringern.

Ein weiterer, im Expose zitierter Anwendungsfall ist die Behandlung von Daten bei Löschung, z.B. das Sichern einer Datenbank, sowie das sichern, dass dies tatsächlich möglich ist, z.B. dass genug Festplattenspeicher vorhanden ist. Dies war bei keiner der hier gewählten Evaluationen notwendig, ist aber für viele weitere mögliche Komponenten ein wichtiger Bestandteil der Funktionalität.

### Neustarts

Anwendungsfälle, bei denen die Aktivierung für einen Neustart unterbrochen wird – vgl. Abschnitt 4.2.5 – habe ich in dieser Arbeit nicht berücksichtigt. Herauszufinden, an welcher Stelle im Bootvorgang sich diese anbieten, wie

man diesen Zustand am sinnvollsten kommuniziert und ob es Möglichkeiten gibt, um solche Neustarts zu minimieren, wäre eine mögliche Folgeaufgabe.

Zusätzlich gibt es teils Änderungen, die im Anschluss einen Neustart erfordern: zum einen gibt es Komponenten, welche dies über eine zusätzliche *Upgrade*-Komponente abbilden könnten. Dieser Anwendungsfall lag hier nicht vor, daher gibt es in dieser Arbeit auch keine Implementierung dafür. Ein weiterer Anwendungsfall ist Konfiguration, die einen Neustart erfordert, beispielsweise das Setzen eines Kernelparameters. In Abschnitt 6.2.1 war dies z.B. nötig, um alle im Linux-Kernel implementierten, softwarebasierten Mitigationen von CPU-Sicherheitsschwachstellen zu deaktivieren (siehe dafür `mitigations=off`). Auch in NixOS wird nicht kommuniziert, dass dies einen Neustart erfordert. Nun ist das Ziel dieser Arbeit, es u.a. Evaluationskomitees zu erleichtern, Artefakte zu evaluieren und dabei wäre dieser Teil auch ein relevanter Aspekt, nachdem dieser Fall aktuell gar nicht behandelt wird.

### Integration mit Systemd

Ein Problem, in Smash derzeit keine allgemeine Lösung hat ist die Behandlung von Operationen, die einen aktiven Dienst erfordern: ein Beispiel wäre PostgreSQL, dessen Server zum Anlegen bzw. Konfigurieren von Datenbanken und -benutzern aktiv sein muss.

Allerdings geschieht der Schritt nach *Upgrade* von *Inactive*. Da die Komponente, die die Funktionalität von `switch-to-configuration` übernimmt, bereits vorher den Daemon von Systemd neu lädt, ist es möglich, temporär einen solchen Dienst für einen Zustandsübergang zu starten und derartige Operationen durchzuführen, allerdings wäre das vermutl. ein Fall für o.g. Standardbibliothek.

Zusätzlich muss sichergestellt sein, dass währenddessen kein anderer Dienst mit PostgreSQL zu diesem Zeitpunkt sprechen darf, was z.B. durch das Setzen entsprechender Firewallregeln in diesem Übergang implementiert werden kann.

### Fehlerbehandlung und Logging

Ich benutze *strukturiertes Logging*, d.h. jede Logzeile enthält eine `key=value`-Zuordnung, damit die Informationen auch für Maschinen verarbeitbar sind. Dies ist allerdings ad-hoc gelöst, wodurch die Benutzung solcher Werte sehr inkonsistent ist. Zusätzlich ist das häufig nicht der Fall für Fehler: um mit einer ersten Implementierung vorwärts zu kommen und trotzdem brauchbare Fehler zu erhalten, habe ich mich oft darauf beschränkt, das gesamte Objekt

mit `{:?}` ins Log zu schreiben. Das gesamte Subsystem könnte entsprechend etwas aufgeräumt werden.

Zusätzlich werden derzeit schwere Fehler nicht entsprechend behandelt: es ist zwar korrekt, dass eine Panik als nicht wiederherstellbarer Fehler angesehen werden kann und damit ein Implementierungsfehler, allerdings sollte es zumindest Anweisungen geben, wie ein Administrator das System wieder auf einen korrekten Stand bringt.

### Unterstützung für mehrere Instanzen

Wie bereits in Kapitel 7 erwähnt, braucht Smash Unterstützung für mehrere Instanzen, um Abhängigkeiten zwischen diesen korrekt abzubilden. Vermutl. könnte man dafür auch das Aeolus$^{-}$-Modell mit angepassten Komponenten benutzen, wie allerdings die genaue Struktur aussieht, ist nicht Teil dieser Arbeit und müsste in Zukunft noch erarbeitet werden.

# Kapitel 9

# Fazit

Mit NixOS und Smash war ich in der Lage, mit einem einzelnen Befehl zwischen verschiedenen Evaluationsumgebungen auf einer Maschine zu wechseln bzw. diese einzurichten. Die Bauvorgänge für die Software sind testbar mit kontinuierlicher Integration und mit einem Lockfile gepinnt, sodass die Artefakte mit allen heutigen Abhängigkeiten auch in Zukunft leicht nachbaubar sind. Hinzu kommen automatische Snapshots des Dateisystems bei der Aktivierung einer Konfiguration. Zustände werden entweder von Smash verwaltet oder es gibt Anweisungen bzgl. was fehlt und welche Schritte unternommen werden müssen.

Dem gegenüber steht ein wesentlich höherer Aufwand zur Paketierung und Implementierung von Smash-Komponenten. Ersteres hat zumindest teilweise damit zu tun, die verwendete Software nicht mehr auf dem neuesten Stand ist und entsprechend viele Anpassungen nötig waren. Wie in diesem Kapitel bereits erwähnt, bin ich außerdem der Meinung, dass Smash nur eine Chance auf weitere Benutzung für Evaluationsumgebungen hat, wenn es auch darüber hinaus verbreitet ist, d.h. idealerweise ein Teil von NixOS selbst ist.

Für Software, die aufgrund einer Mischung von Code und Daten schwierig zu paketieren ist – wie z.B. SPEC CPU 2006 [61] – habe ich in Abschnitt 6.2.1 einen Ansatz vorgeschlagen, der die Handhabung davon möglicherweise verbessert, dafür aber verstärkt in der Praxis ausgetestet werden muss.

Damit Software ohne viel Aufwand in `nixpkgs` [80] integriert werden kann und damit für Evaluationsumgebungen auf NixOS benutzt werden kann, ist es wichtig, dass diese keine Antipatterns bzgl. Paketierung enthält [63]. Es gibt sehr viel Software, welche aus verschiedenen Gründen schwer zu paketieren ist und – sofern man diese für ein Artefakt benötigt – die Verwendung von NixOS extrem erschwert. Das ist ein Thema, das alle Autoren von Software betrifft.

Insgesamt glaube ich, dass Smash ein kleiner Schritt in die richtige Rich-

tung ist, aber noch deutlich mehr Arbeit und Tests benötigt, um produktiv benutzt zu werden. Sofern die benutze Software sich gut paketieren lässt, ist der Einsatz von NixOS für Paket- und Konfigurationsverwaltung sehr hilfreich und ein Ansatz, der sich für Artefakte lohnt.

# Literaturverzeichnis


[1] ACME Challenge Types. https://letsencrypt.org/docs/challenge-types/.

[2] AWS Provider. https://registry.terraform.io/providers/hashicorp/aws/latest/docs.

[3] Charly castes. https://charlycst.github.io/posts/mpk/.

[4] Control groups v2. https://docs.kernel.org/admin-guide/cgroup-v2.html.

[5] Das Z-Dateisystem (ZFS). https://docs.freebsd.org/de/books/handbook/zfs/.

[6] Debian. https://www.debian.org/.

[7] ECMA-404 The JSON Data Interchange Standard. https://www.json.org/json-en.html.

[8] Graph data structure library for rust. https://github.com/petgraph/petgraph.

[9] ld.so(8). https://man7.org/linux/man-pages/man8/ld.so.8.html.

[10] Memory Protection Keys. https://docs.kernel.org/core-api/protection-keys.html#syscalls.

[11] mmapbench. https://github.com/SepehrDV2/mmap-anon-benchmarks/tree/extmem-eval.

[12] NetworkX: stratify a DAG into generations. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.dag.topological_generations.html.

[13] Nix flakes. https://nix.dev/concepts/flakes.html.

[14] pahole. https://git.kernel.org/pub/scm/devel/pahole/pahole.git.

[15] pahole v1.24: FAILED: load BTF from vmlinux: Invalid argument. https://lore.kernel.org/all/2b8a762d-d013-c1df-1be0-29df6126f8c6@fb.com/T/.

[16] QEMU, A generic and open source machine emulator and virtualizer. https://www.qemu.org/.

[17] Rust: A language empowering everyone to build reliable and efficient software. https://www.rust-lang.org/.

[18] Syntax error in llvm. https://github.com/mars-research/llvm-SFI/blob/e925e5eab9e91adb184995d734f7c4c9867d2497/llvm/lib/Target/X86/X86NaClPass.cpp#L256.

[19] systemd-run(1). https://www.freedesktop.org/software/systemd/man/latest/systemd-run.html.

[20] systemd.network(5). https://www.freedesktop.org/software/systemd/man/latest/systemd.network.html.

[21] systemd.resource-control(5). https://www.freedesktop.org/software/systemd/man/latest/systemd.resource-control.html.

[22] Systems Research Artifacts. https://sysartifacts.github.io/.

[23] tc(8). https://man7.org/linux/man-pages/man8/tc.8.html.

[24] The LLVM Compiler Infrastructure. https://llvm.org/.

[25] Uffdio_register ioctl. https://man7.org/linux/man-pages/man2/uffdio_register.2const.html.

[26] Userfaultfd. https://docs.kernel.org/admin-guide/mm/userfaultfd.html.

[27] zram: Compressed ram-based block devices. https://docs.kernel.org/admin-guide/blockdev/zram.html.

[28] Usenix annual technical conference, 2025. https://www.usenix.org/conference/atc25.

[29] Apache Foundation. Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. `https://kafka.apache.org/`.

[30] Apache Foundation. Stateful Computations over Data Streams. `https://flink.apache.org/`.

[31] Y. Babichenko. Finite state machine framework for Rust with readable specifications. `https://github.com/eugene-babichenko/rust-fsm`.

[32] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[33] P. Bi, M. Xiao, D. Yu, and G. Zhang. obbr artifacts. `https://github.com/bpq233/oBBR`.

[34] P. Bi, M. Xiao, D. Yu, and G. Zhang. oBBR: Optimize retransmissions of BBR flows on the internet. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 537–551, Boston, MA, July 2023. USENIX Association.

[35] M. Bosch. Fork of petgraph, with topological generations. `https://git.mbosch.me/bachelorarbeit/petgraph`.

[36] M. Bosch. nixos/postgresql: drop ensurePermissions option. `https://github.com/nixos/nixpkgs/commit/d363f526259bb22416d885e244c89061515d0b23`.

[37] M. Bosch. S.M.A.S.H. – State Management avoiding Shell. `https://git.mbosch.me/bachelorarbeit/statemgmt/`.

[38] M. Bosch. Subtree split of nixpkgs' switch-to-configuration-ng, restructured as library. `https://git.mbosch.me/bachelorarbeit/libstc/`.

[39] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, Oct. 2016.

[40] N. Cardwell, Y. Cheng, S. H. Yeganeh, I. Swett, V. Vasiliev, P. Jha, Y. Seung, M. Mathis, and V. Jacobson. BBR v2, A Model-based Congestion Control, 2019. `https://datatracker.ietf.org/meeting/104/materials/slides-104-iccrg-an-update-on-bbr-00`.

[41] Carl Lerche. Tokio is an asynchronous runtime for the Rust programming language. It provides the building blocks needed for writing network applications. https://tokio.rs/.

[42] M. Catan, R. Di Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the complexity of cloud application deployment. In *ESOCC 2013: Service-Oriented and Cloud Computing*, volume 8135 of *LNCS*, pages 1–3. Springer-Verlag, 2013.

[43] D. Chai, J. Zhang, L. Yang, and Y. Jin. Artifacts for Efficient Decentralized Federated Singular Vector Decomposition. https://github.com/Di-Chai/Excalibur.

[44] X. Chen and Z. Li. Artifact for Opportunities and Limitations of Modern Hardware Isolation Mechanisms, 2024. https://github.com/mars-research/atc24-artifact.

[45] X. Chen, Z. Li, T. Jain, V. Narayanan, and A. Burtsev. Limitations and opportunities of modern hardware isolation mechanisms. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 349–368, Santa Clara, CA, July 2024. USENIX Association.

[46] Chromium developers. Checking out and building Chromium on Linux. https://chromium.googlesource.com/chromium/src/+/main/docs/linux/build_instructions.md.

[47] Chromium developers. Playing with quic. https://www.chromium.org/quic/playing-with-quic/.

[48] N. Community. Postgres service module blocks WAL restoration, 2024. https://github.com/NixOS/nixpkgs/issues/346886.

[49] Confluent Inc. The Apache Kafka C/C++ library. https://github.com/confluentinc/librdkafka.

[50] K. Cook. libsubcmd: Fix use-after-free for realloc(..., 0). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/patch/?id=52a9dab6d892763b2a8334a568bd4e2c1a6fde66.

[51] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[52] E. Dolstra. A small utility to modify the dynamic linker and RPATH of ELF executables. https://github.com/nixos/patchelf.

[53] E. Dolstra. The purely functional software deployment model, 2006.

[54] freedesktop.org Project. D-Bus is a simple system for interprocess communication and coordination. `https://www.freedesktop.org/wiki/Software/dbus/`.

[55] Google. BoringSSL. `https://boringssl.googlesource.com/boringssl`.

[56] Google Inc. Google Native Client. `https://developer.chrome.com/docs/native-client`.

[57] D. Graur, O. Mraz, M. Li, S. Pourghannad, C. A. Thekkath, and A. Klimovic. Artifacts for Pecan: Cost-Efficient ML Data Preprocessing with Automatic Transformation Ordering and Hybrid Placement. `https://github.com/eth-easl/pecan-experiments`.

[58] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang. Meces artifacts. `https://github.com/ATC2022No63/Meces-Artifact/`.

[59] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang. Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 539–556, Carlsbad, CA, July 2022. USENIX Association.

[60] A. Hemel. NixOS: the nix based operating system, 2006.

[61] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[62] M. Hock, R. Bless, and M. Zitterbart. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP), Toronto, ON, 10–13 October 2017*, page 1–10. Institute of Electrical and Electronics Engineers (IEEE), 2017.

[63] K. Hoste. How To Make Package Managers Cry. `https://archive.fosdem.org/2018/schedule/event/how_to_make_package_managers_cry/`.

[64] B. Hubert. PowerDNS Server. `https://github.com/PowerDNS/pdns`.

[65] Intel Corporation. Intel® hyper-threading technology. `https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html`.

[66] Intel Corporation. The system call intercepting library. https://github.com/pmem/syscall_intercept.

[67] S. Jalalian, S. Patel, M. R. Hajidehi, M. Seltzer, and A. Fedorova. Extmem artifacts. https://github.com/SepehrDV2/ExtMem.

[68] S. Jalalian, S. Patel, M. R. Hajidehi, M. Seltzer, and A. Fedorova. ExtMem: Enabling Application-Aware virtual memory management for Data-Intensive applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 397–408, Santa Clara, CA, July 2024. USENIX Association.

[69] N. Jared Baur. nixos/switch-to-configuration: add new implementation. https://github.com/NixOS/nixpkgs/pull/308801.

[70] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? volume 19, 04 2010.

[71] T. Lane. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[72] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Lennart Poettering. System and Service Manager. https://systemd.io/.

[74] Z. Lin, L. Xiang, J. Rao, and H. Lu. Artifacts for P2CACHE: Exploring Tiered Memory for In-Kernel File Systems Caching. https://github.com/YesZhen/P2CACHE.

[75] J. Malka, S. Zacchiroli, and T. Zimmermann. Reproducibility of Build Environments through Space and Time. In *46th International Conference on Software Engineering (ICSE 2024) - New Ideas and Emerging Results (NIER) Track*, Lisbonne, Portugal, Apr. 2024.

[76] S. Narayan, T. Garfinkel, E. Johnson, Z. Yedidia, Y. Wang, A. Brown, A. Vahldiek-Oberwagner, M. LeMay, W. Huang, X. Wang, M. Sun,

D. Tullsen, and D. Stefan. Segue & colorguard: Optimizing sfi performance and scalability on modern architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 987–1002, New York, NY, USA, 2025. Association for Computing Machinery.

[77] NixOS Community. Download NixOS. `https://nixos.org/download/#download-nixos-accordion`.

[78] NixOS Community. Locale change in postgresql for synapse. `https://github.com/NixOS/nixpkgs/pull/80447#discussion_r381637434`.

[79] NixOS Community. Nix & NixOS | Declarative Builds and Deployments. `https://nixos.org/`.

[80] NixOS Community. Nix packages collection & nixos. `https://github.com/nixos/nixpkgs`.

[81] NixOS Community. NixOS Specialisations. `https://wiki.nixos.org/wiki/Specialisation`.

[82] NixOS Community. Overlays - Nixpkgs Reference Manual. `https://nixos.org/manual/nixpkgs/stable/#chap-overlays`.

[83] NixOS Community. Disko declarative disk management, 2023. `https://github.com/nix-community/disko`.

[84] NixOS Community. postgresql_15 requires granting permissions on schema public, ensureusers insufficient, 2023. `https://github.com/NixOS/nixpkgs/issues/216989`.

[85] NixOS Community. Nixpkgs Manual: Linux kernel, 2024. `https://nixos.org/manual/nixpkgs/stable/#sec-linux-kernel`.

[86] NixOS Community. postgres fails to start when template database with collation version mismatch is needed at startup, 2024. `https://github.com/NixOS/nixpkgs/issues/318777`.

[87] NixOS Community. switch-to-configuration.pl, 2025. `https://switch-to-configuration.pl/`.

[88] Numtide. install nixos everywhere via ssh. `https://github.com/nix-community/nixos-anywhere`.

[89] L. PAGE. Us6285999b1: Method for node ranking in a linked database, 1997.

[90] D. L. Paul Kranenburg. linux syscall tracer. `https://strace.io/`.

[91] S. Qian and A. Goel. Artifacts for Massively Parallel Multi-Versioned Transaction Processing. `https://github.com/ShujianQian/epic-artifact`.

[92] Red Hat Inc. Rust implementation of the Varlink protocol. `https://github.com/varlink/rust`.

[93] Red Hat Inc. Varlink is an interface description format and protocol. `https://varlink.org/`.

[94] Serokell. deploy-rs: A simple multi-profile Nix-flake deploy tool. `https://github.com/serokell/deploy-rs`.

[95] H. Shirwadkar, S. Kadekodi, and T. Tso. Artifacts for FastCommit: resource-efficient, performant and cost-effective file system journaling. `https://github.com/harshadjs/fast-commit-atc-2024`.

[96] I. V. Sysoev. nginx. `https://nginx.org/`.

[97] The BSD project. OpenSSH. `https://www.openssh.com/`.

[98] The GNU project. GNU Bash. `https://www.gnu.org/software/bash/`.

[99] The GNU project. GNU Compiler Collection. `https://gcc.gnu.org/`.

[100] The GNU project. GNU Make. `https://www.gnu.org/software/make/`.

[101] The Linux Foundation. The open source infrastructure as code tool. `https://opentofu.org/`.

[102] D. Tolnay. Serde is a framework for serializing and deserializing Rust data structures efficiently and generically. `https://serde.rs/`.

[103] S. Traugott and L. Brown. Why Order Matters: Turing Equivalence in Automated Systems Administration. In *Proceedings of the 16th USENIX Conference on System Administration*, LISA '02, page 99–120, USA, 2002. USENIX Association.

[104] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, Aug. 2019. USENIX Association.

[105] S. van der Burg and E. Dolstra. Automating system tests using declarative virtual machines. *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 181–190, 2010.

[106] S. Zacchiroli. Large-scale modeling, analysis, and preservation of free and open source software, 2017.

[107] Zhaofeng Li. A simple, stateless NixOS deployment tool. `https://github.com/zhaofengli/colmena`.

# Glossar

**Aeolus** Modellierung von konvergenten Deployments. 16, 47

**Aktivierung** Ausrollen einer neuen NixOS-Konfiguration. Kann Zustandsverwaltung enthalten auf Basis des Aeolus-Modells. 6, 8, 14, 46–48, 57, 71, 79, 83, 88, 95, 103, 107

**BBR** Bottleneck Bandwidth and Roundtrip Time (BBR), ein Verfahren zur TCP-Staukontrolle. 33

**Closure** Menge aus einem Nix-Storepfad und allen Laufzeitabhängigkeiten. 33, 55

**Derivation** Die Beschreibung eines Builds für den Nix Paketmanager. 7, 9–11, 36

**FFI** (Foreign Function Interface) Mechanismus, um u.a. Funktionen, die in einer anderen Sprache implementiert sind, aufzurufen. 63

**Komponente** Baustein im Aeolus-Modell & Smash, welche ein Subsystem repräsentiert und Zustandsübergänge von diesem abbildet. 16, 49, 53, 59, 70, 86, 93, 102

**Komponentenmodell** Drückt die Menge an Komponenten und deren Beziehung untereinander aus. 16, 49

**Kontextmanager** Konstrukt aus Python, das für einen Codeblock einen Kontext vorbereitet und nach Verlassen wieder entfernt. Wird beispielsweise für Locking benutzt. 27, 38

**Lockfile** Datei, welche Abhängigkeiten auf eine bestimmte Versionen bzw. Git Revisionen fixiert. 4, 111

**MPK** Memory Protection Keys. 91

**NixOS** Eine Linuxdistribution, welche deklarative Bauvorgänge und transaktionale Upgrades bietet. 7, 21, 55, 57, 107, 111

**NUMA** (Non-Uniform Memory Access) Dies ist eine Architektur, bei der CPUs ihren eigenen, lokalen Speicher haben, aber andere CPUs Zugriff darauf haben. 26

**Reconciliation** Der Vorgang des Angleichens des realen Zustandes eines Systems an den gewünschten Zustand. 6, 12, 104

**Routine** Nicht-blockierender Arbeitsvorgang in einer asynchronen Laufzeitumgebung. Manchmal auch *green thread* genannt. 64, 68

**Segmentierfehler** (*segfault*) Versuch auf Speicherbereiche zuzugreifen, auf die der aktuelle Prozess keinen Zugriff hat. 26, 32, 63, 103

**Smash** Im Zuge dieser Arbeit entstandene Referenzimplementierung von Zustandsverwaltung auf NixOS. 57, 59, 61, 62, 67, 78, 88, 93, 101, 107

**Snapshot** Unveränderliches Abbild des Dateisystems zu einem bestimmten Zeitpunkt. 6, 46, 49, 53, 100

**strukturiertes Logging** Bezeichnung für (teilweise) maschinenlesbares Format von Logzeilen. 69, 108

**toplevel** *toplevel* bzw. *system.build.toplevel* ist das Attribut einer NixOS-Konfiguration, das den Storepfad mit dem System selbst enthält. 57, 80

**Varlink** JSON-basiertes IPC-Protokoll. 15, 63

**Zurückrollen** Vorgang, bei dem eine ältere Version einer Konfiguration eines NixOS-Systems aktiviert wird. 46, 52, 76

**Zustand** Für eine Abgrenzung des Begriffs Zustand siehe Abschnitt 2.2. 77, 79, 100, 104