technische universität
dortmund

Bachelorarbeit

Erweiterung des ereignisbasierten Betriebssystems CyPhOS um eine dynamische cache-gewahre Speicherverwaltung

Christian Hakert
28. Juli 2017

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
M.Sc. Hendrik Borghorst

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

Zusammenfassung

Moderne cyber-physikalische Anwendungen stellen hohe Anforderungen an ihre Ausführungsplattform. Neben harten Echtzeitanforderungen setzen solche Anwendungen auch häufig eine dynamische Ressourcenverwaltung voraus. Programme sollen z. B. während ihrer Ausführung um Funktionalitäten oder sogar um weitere Programminstanzen erweitert werden können. Ein weiteres Ziel ist es, diesen Anforderungen nicht nur mit spezieller, sondern auch mit möglichst günstiger general-purpose Hardware gerecht zu werden. Das Betriebssystem CyPhOS bietet Anwendungen eine Umgebung, um diese mit harten Echtzeitanforderungen auf moderner general-purpose Hardware auszuführen. Diese Arbeit erweitert CyPhOS um eine dynamische Speicherverwaltung, sodass auch die Anforderungen an die dynamische Ressourcennutzung während der Ausführung der Programme im Bezug auf Speicher erfüllt werden können.

Inhaltsverzeichnis

1. Einleitung **1**

2. Einleitung zum Betriebssystem CyPhOS **3**
2.1. Komponenten 3
2.2. Events und Trigger 3

3. Das Speichermodell von CyPhOS **5**

4. Grundlegende Konzepte der Speicherverwaltung **7**
4.1. Speicherabhängigkeiten und Synchronisierung 7
4.2. Speicherklassen 8
4.2.1. Immer im Cache (fully cached) 8
4.2.2. Dynamisch-gecachet (dynamic cached) 9
4.2.3. Nicht-gecachet (not cached) 9
4.3. Micro-Heaps und Component-Heaps 10
4.4. Freispeicherverwaltung 11
4.5. Schnittstelle 12

5. Implementierung **15**
5.1. Anlegen der Speicherpools 15
5.2. Micro-Heaps und Component-Heaps 15
5.3. Freispeicherverwaltung 18
5.3.1. Belegung von Speicher 18
5.3.2. Freigabe von Speicher 19
5.4. Cachen von Inhalten 20
5.5. Dynamisch gecachete Speicher (Access-Flag) 21
5.5.1. Aktivieren des Access-Flag 21
5.5.2. Nutzung des Access-Flag 22
5.5.3. Behandlung des Access-Flag-fault 22
5.6. Nicht gecacheter Speicher 23
5.7. `new`-Operator 23
5.8. Speicherabhängigkeiten 24
5.8.1. Kohärenz im Translation-Look-Aside-Buffer 25

6. Anwendungsbeispiel: CAN-Bus **27**
6.1. Der CAN-Treiber 27

6.2. Anwendungskomponente: Gangwahlschalter 27

7. Evaluation **29**
7.1. Konstruierte Testumgebung 29
7.2. Auswertung der Messungen 30
7.3. Zugriffszeiten in den verschiedenen Speicherklassen 31
7.4. Zugriffszeiten unter parallel stattfindenden Zugriffen 34
7.4.1. Keine Verdrängungen im Cache 34
7.4.2. Verdrängungen im Cache 38
7.5. Vergleich mit reinem Hardware-Cache-Management 42
7.5.1. Kein paralleler Betrieb (vgl. Kapitel 7.3) 42
7.5.2. Paralleler Betrieb (vgl. Kapitel 7.4) 46
7.6. Verhalten der Vorladezeiten der Komponenten 48
7.7. Fazit der Evaluation 51

8. Zusammenfassung und Ausblick **53**
8.1. Konzept der Speicherverwaltung 53
8.2. Leistungsfähigkeit der Speicherverwaltung 53
8.3. Stand der Forschung 54
8.4. Ausblick 54
8.4.1. Cache-Gewahre Speicherallokation 54
8.4.2. Alignment an Cache-Grenzen 55
8.4.3. Dynamische Speicherverwaltung für Systemfunktionen . . 55

Literaturverzeichnis **57**

Abbildungsverzeichnis **59**

A. Statistische Kennzahlen der Evaluationsergebnisse (Kapitel 7) **I**
A.1. Zugriffszeiten ohne Störungen I
A.2. Zugriffszeiten unter Störungen I
A.2.1. Keine Verdrängungen I
A.2.2. Verdrängungen II
A.3. Hardware-Cache-Verwaltung II

B. Zugriffszeitmessungen auf einem Wandboard mit i.MX6 CPU **V**
B.1. Speicherzugriffe ohne Störung V
B.2. Zugriffszeiten unter parallel stattfindenden Zugriffen VII
B.2.1. Keine Verdrängungen im Cache VIII
B.2.2. Verdrängungen im Cache XI
B.3. Vergleich mit reinem Hardware-Cache-Management XIV
B.3.1. Kein paralleler Betrieb (vgl. Kapitel B.1) XIV
B.3.2. Paralleler Betrieb (vgl. Kapitel B.2) XVII

B.4. Verhalten der Vorladezeiten der Komponenten XIX
B.5. Fazit der Evaluation XXI

C. Statistische Kennzahlen der Evaluationsergebnisse auf einem Wandboard (Anhang B) XXIII
C.1. Zugriffszeiten ohne Störungen XXIII
C.2. Zugriffszeiten unter Störungen XXIII
C.2.1. Keine Verdrängungen XXIII
C.2.2. Verdrängungen XXIV
C.3. Hardware-Cache-Verwaltung XXIV

1. Einleitung

CyPhOS ist ein Betriebssystem, das ein Echtzeitverhalten auf moderner Hardware ermöglichen soll [7]. Um dies zu erreichen, benötigt das Betriebssystem eine Möglichkeit, Inhalte gezielt in Cache-Ways laden zu können und diese vor der Verdrängung zu schützen. Der CoreLink Level 2 Cache-Controller L2C-310 bietet die Möglichkeit einzelne Cache-Ways des L2-Caches zu sperren [2]. Dieser Controller kommt in Cortex-A9 Prozessoren zum Einsatz [4]. Wird ein Cache-Way gesperrt, so können Inhalte nicht mehr aus diesem verdrängt werden. Es können auch keinen neuen Inhalte in den gesperrten Cache-Way abgelegt werden. CyPhOS implementiert auf dieser Basis eine Software-Cache-Verwaltung, mit der Inhalte gezielt in einen Cache-Way geladen werden können und dieser anschließend gesperrt werden kann. Somit kann sichergestellt werden, dass ein bestimmter Inhalt garantiert im L2-Cache liegt.
Das Betriebssystem CyPhOS ist größtenteils in C++ geschrieben, bietet aber keine Möglichkeiten, zur Laufzeit Speicher zu allozieren oder Objekte zu erstellen. Daher wird CyPhOS in dieser Arbeit um eine dynamische Speicherverwaltung erweitert. Damit soll es Anwendungen ermöglicht werden, zur Laufzeit Speicher für individuelle Verwendungszwecke zu allozieren und auch wieder freigeben zu können. Dadurch soll den Anwendungen ein breiteres Spektrum an Möglichkeiten geboten werden, wie sie z. B. im Kontext der cyber-physikalischen Systeme benötigt werden. Weiterhin soll die Speicherverwaltung das dynamische Erstellen und Vernichten von Objekten ermöglichen. Dadurch können Programme ihre Funktionalitäten erweitern oder das Betriebssystem kann Programme generieren, die anschließend zur Ausführung kommen.
Damit auch für die Speicherverwaltung und vor allem für die allozierten Speicherbereiche dieselben Echtzeitanforderungen erfüllt werden können, wie CyPhOS sie für die Anwendungen bereits erfüllt, muss die Speicherverwaltung ebenfalls eine aktive Cache-Verwaltung durchführen. Für allozierte Speicher muss dieselbe Cache-Gewahrheit sichergestellt werden können, wie für Anwendungen selber. Dies erfordert eine Interaktion zwischen der Speicherverwaltung und den Betriebssystemfunktionen, weshalb die Speicherverwaltung in das Betriebssystem selber integriert wird.

2. Einleitung zum Betriebssystem CyPhOS

CyPhOS (Cyber-physical operating system) ist ein Echtzeitbetriebssystem für moderne Mehrkern-Hardware. Dabei wird die Annahme zu Grunde gelegt, dass die parallele Nutzung geteilter Ressourcen durch mehrere Prozessorkerne der Hauptgrund für unvorhersagbare Laufzeiten ist. Während der Ausführung eines Programms auf einem Mehrkern-Prozessor ist der gemeinsame Bus zum Hauptspeicher u. a. eine gemeinsam genutzte Ressource [7]. CyPhOS stellt sicher, dass während der Ausführung des Betriebssystems keine Zugriffe in den Hauptspeicher erfolgen und somit die gemeinsame Nutzung dieser Ressource vermieden wird.
Um Hauptspeicherzugriffe zu vermeiden führt CyPhOS eine aktive Cache-Verwaltung durch. Diese sorgt dafür, dass jeglicher Programmcode und jegliche Programmdaten, die zur Ausführung kommen, vorher in den L2-Cache geladen wurden. Um dies zu erreichen, stellt CyPhOS eine Abstraktion für Anwendungsprogramme zur Verfügung.

2.1. Komponenten

Jedes Anwendungsprogramm, welches für CyPhOS entwickelt wird, ist eine Komponente. Die Komponente besteht lediglich aus dem Daten- und Text-Segment des kompilierten Programmcode. Wird die Komponente ausgeführt, so wird ihr ein Stack aus einem zentralen Pool zugewiesen. Eine Komponente kann verschiedene Einstiegspunkte haben, an denen sie zur Ausführung kommt. Beim Kompilieren und Linken des Betriebssystems werden die Komponenten auf eine bestimmte Art angeordnet. Dadurch ist es möglich, eine einzelne Komponente in einen Cache-Way zu laden und sicherzustellen, dass ihre Ausführung keine Hauptspeicherzugriffe benötigt.

2.2. Events und Trigger

Wenn eine Komponente ausgeführt werden soll, muss CyPhOS diese zuerst in den Cache laden, falls dies nicht schon zu einem früheren Zeitpunkt geschehen ist und die Komponente nicht wieder aus dem Cache verdrängt wurde. Dies hat zur

Folge, dass das Betriebssystem für jede Ausführung einer Komponente benachrichtigt werden muss. Um dies zu gewährleisten führt CyPhOS eine Abstraktion zum Ausführen von Komponenten ein. Eine Funktion der Komponente kann nicht aus einer anderen heraus angesprungen werden. Anstatt dessen muss ein Event ausgelöst werden. Für ein Event ist ein Trigger hinterlegt, der in Folge des Events ausgeführt werden soll. Zu dem Trigger gehört unter anderem eine Funktion einer Komponente, die ausgeführt werden soll. Trigger stellen die Einstiegspunkte in eine Komponente dar.

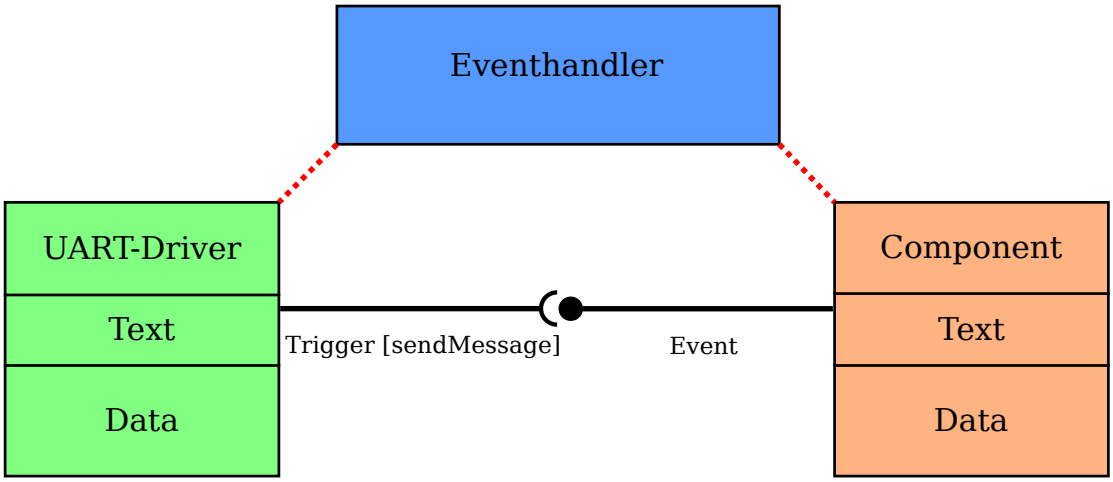

Abbildung 2.1.: Interaktion zwischen zwei Komponenten

Der UART Treiber stellt einen Trigger zur Verfügung, um eine Nachricht abzusenden. Die Komponente muss ein Event auslösen, um diese Funktion nutzen zu können. Logisch interagiert die Komponente direkt mit dem Treiber, tatsächlich erfolgt die Auslösung des Trigger aber durch den Eventhandler.

Um ein Event auszulösen, muss eine Komponente den Eventhandler benachrichtigen. Dieser kann daraufhin überprüfen, ob die Zielkomponente bereits im Cache liegt, diese eventuell laden und den zugehörigen Trigger zur Ausführung bringen.

3. Das Speichermodell von CyPhOS

Ein zentrales Merkmal von CyPhOS ist die aktive Cache-Verwaltung. Damit Komponenten gezielt in den L2-Cache geladen werden können, werden diese beim Linken des Betriebssystems auf eine bestimmte Art angeordnet.

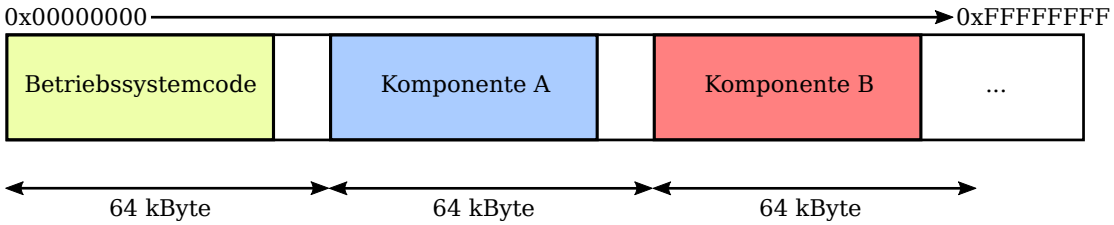

Abbildung 3.1.: Hauptspeicherabbild von CyPhOS mit zwei Komponenten

Wie in Abbildung 3.1 zu erkennen ist, werden alle Betriebssystemfunktionen an den Anfang des Hauptspeichers platziert. Dazu gehören u. a. das Eventhandling und die Cache-Verwaltung. Anschließend folgt der kompilierte Code aller Komponenten. Dieser besteht jeweils aus dem Text und Daten-Segment. Jede Komponenten wird an eine 64 kB Grenze ausgerichtet. Jeder Cache-Way des Prozessors hat eine Größe von 64 kB, somit ist gewährleistet, dass jede Komponente in einen separaten Cache-Way geladen werden kann und aufgrund der Cache-Assoziativität keine Überschneidungen auftreten.
Während des Startvorgangs des Betriebssystems CyPhOS wird der L2-Cache initialisiert und alle Cache-Ways werden gesperrt. Der Cache-Controller kann nur ungesperrte Cache-Ways verwenden, um Inhalte abzulegen. Aus gesperrten Cache-Ways werden Inhalte niemals verdrängt. Wenn eine Komponente in den Cache geladen werden soll, wird ein Cache-Way entsperrt und Preload-Instruktionen für den gesamten Speicher der Komponente aufgerufen. Der Cache-Controller legt dann die Komponente gezwungenermaßen in den entsperrten Cache-Way. Anschließend wird der Cache-Way wieder gesperrt und das Laden der Komponente ist abgeschlossen.
Während ein Cache-Way entsperrt ist, muss sichergestellt werden, dass kein anderer Kern Zugriffe auf nicht gecacheten, cachebaren Speicher ausführt. Dadurch würde der Cache-Controller eben auch die zugegriffenen Inhalte in dem entsperrten

Cache-Way platzieren. Zum einen wird so ein paralleler Zugriff dadurch verhindert, dass immer nur eine Komponente zeitgleich vorgeladen wird. Zum anderen erfolgen nirgendwo im Betriebssystem Zugriffe auf Inhalte die nicht im Cache liegen.
Während des Startvorgangs des Betriebssystems wird der Systemcode in einen Cache-Way geladen und dieser anschließend für jegliche weitere Verwendung gesperrt. Wird ein Event ausgelöst, welches einen Trigger startet, der eine nicht geladene Komponente zur Ausführung bringt, so wird diese Komponente vor der Ausführung des Trigger in einen freien Cache-Way geladen. Dieses Verhalten wird in Abbildung 3.2 veranschaulicht.

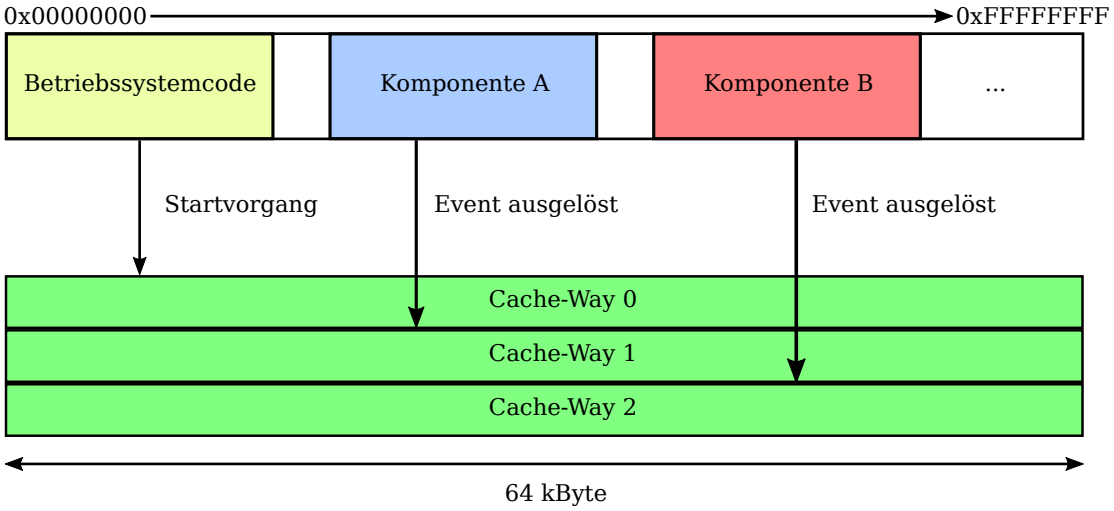

Abbildung 3.2.: Laden der Komponenten in den L2-Cache

4. Grundlegende Konzepte der Speicherverwaltung

Die Speicherverwaltung, die für CyPhOS entwickelt werden soll, muss die Cache-Gewahrheit des Systems sicherstellen. Dies bedeutet, dass sichergestellt werden muss, dass Zugriffe in reservierten Speicher keine Hauptspeicherzugriffe verursachen, sondern lediglich auf den L2-Cache zugreifen. Weiterhin soll die Speicherverwaltung dynamisch betrieben werden. Komponenten sollen zur Laufzeit Speicher reservieren und freigeben können. Sowohl logisch, als auch physikalisch werden reservierte Speicherbereiche der reservierenden Komponente zugeordnet. Somit können diese gemeinsam mit der Komponente vorgeladen werden. Dem Betriebssystem stehen nur eine stark begrenzte Anzahl von Cache-Ways zur Verfügung. Durch die Zuordnung der Speicherbereiche zu den Komponenten wird erreicht, dass der L2-Cache nur von den aktuell ausgeführten Komponenten und ihrem reservierten Speicher belegt wird.

4.1. Speicherabhängigkeiten und Synchronisierung

Die Speicherverwaltung soll ermöglichen, dass reservierter Speicher von verschiedenen Komponenten genutzt werden kann. Ein Anwendungsbeispiel dazu ist in Abbildung 4.1 dargestellt.
Hierbei muss die Cache-Gewahrheit ebenfalls sichergestellt werden. Um dies zu gewährleisten, kann eine Anwendungskomponente für jeden beliebigen Trigger Speicherabhängigkeiten angeben. Diese Abhängigkeiten werden von dem Eventhandler vor der Ausführung des Trigger überprüft und die entsprechenden Speicherbereiche in den Cache geladen. Zusätzlich findet im Eventhandler noch eine Synchronisierung statt. Es wird sichergestellt, dass niemals mehrere Komponenten gleichzeitig ausgeführt werden, die den selben Speicherbereich benötigen. Speicherabhängigkeiten können zur Laufzeit von jeder Komponente für jeden Trigger und für jeden reservierten Speicherabschnitt hinzugefügt und entfernt werden.

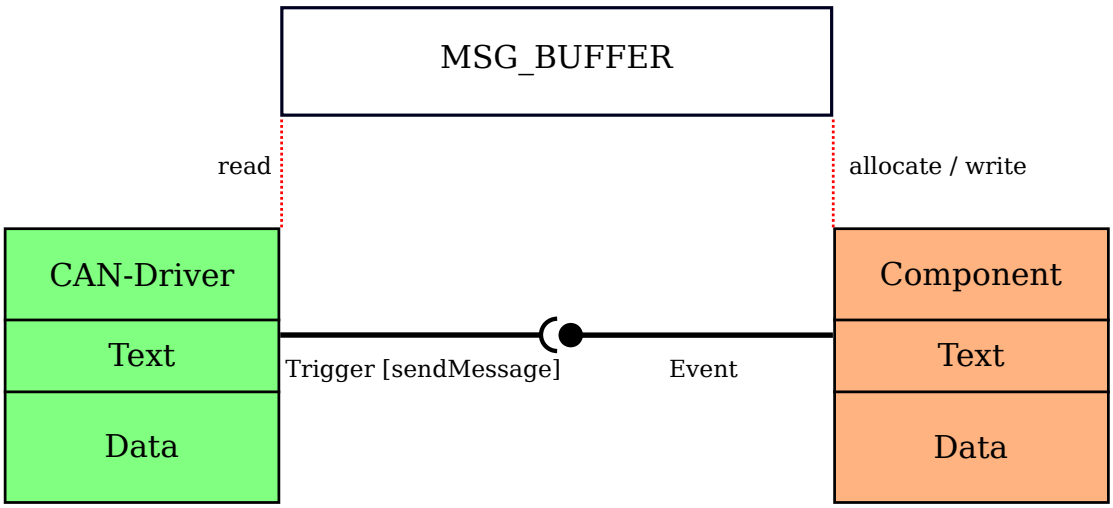

Abbildung 4.1.: Gemeinsam genutzter Speicher

Das Beispiel zeigt eine Komponente, die eine Nachricht über den CAN-Bus versendet. Der Treiber für den CAN-Bus stellt den Trigger “sendMessage” zur Verfügung. Die Komponente bereitet die zu versendende Nachricht in einem reservierten Speicherbereich vor. Der CAN-Treiber liest diesen Speicherbereich während der Ausführung des Trigger und versendet die Nachricht. Die Komponente kann eine Abhängigkeit für den Trigger “sendMessage” des CAN-Treibers hinzufügen, sodass die vorbereitete Nachricht garantiert im Cache liegt, wenn dieser Trigger zur Ausführung kommt. Das Schema lässt sich auch mit beliebig vielen Komponenten anwenden.

4.2. Speicherklassen

Die Anforderung an die Cache-Gewahrheit der Speicherverwaltung bringt starke Einschränkungen mit sich. Um den Anwendungen eine flexiblere und uneingeschränktere Speicherverwaltung anzubieten, werden drei verschiedene Speicherklassen implementiert, die unterschiedliche Grade der Cache-Gewahrheit erreichen.

4.2.1. Immer im Cache (fully cached)

Speicherbereiche, die in dieser Klasse reserviert werden, werden immer zusammen mit der Komponente in den L2-Cache geladen. Der Eventhandler überprüft vor der Ausführung eines Trigger die eingetragenen Speicherabhängigkeiten. Anschließend wird die Komponente selbst, aber auch die Speicherbereiche, die für diesen Trigger eingetragen sind, in einen freien Cache-Way geladen. In dieser Speicherklasse wird

sichergestellt, dass die reservierten Speicherbereiche bei der Ausführung der Komponente bereits im L2-Cache liegen. Der Nachteil ergibt sich aus der begrenzten Anzahl der Cache-Ways und der beschränkten Cache-Way Größe. Somit können für einen Trigger nur wenige und kleine Speicherbereiche aus dieser Klasse genutzt werden. Die Zugriffszeiten für Speicher aus dieser Klasse wiederum sind maximal gering und vorhersagbar. Die reservierten Speicherbereiche aus dieser Klasse verhalten sich identisch wie der Programmcode und die Programmdaten der Komponente selbst.

4.2.2. Dynamisch-gecachet (dynamic cached)

Die zweite Speicherklasse soll die Einschränkungen der ersten Speicherklasse aufweichen, ohne größere Nachteile mit sich zu bringen. Eine Anwendung kann in dieser Speicherklasse beliebig viel Speicher reservieren. Kommt ein Trigger zur Ausführung, so wird vorerst keiner der reservierten Bereiche geladen. Dies geschieht erst beim ersten Zugriff in einen reservierten Speicherbereich. Daraufhin wird der entsprechende Block in einen freien Cache-Way geladen. Alle weiteren reservierten Speicherbereiche dieser Klasse bleiben unangetastet. Für den ersten Zugriff auf einen Speicherbereich ergeben sich dadurch höhere Zugriffszeiten, alle weiteren Zugriffe sind dann allerdings ebenso schnell und vorhersagbar wie Zugriffe auf Speicher der ersten Klasse. Ein lokales Speicherverhalten der ausgeführten Anwendungen kommt den Zugriffszeiten zusätzlich zu Gute, denn ein Speicherbereich, der in den Cache geladen wurde, wird erst verdrängt, wenn der Cache-Way für andere Komponenten oder Speicherbereiche benötigt wird.
Für die Anwendungsentwicklung ergibt sich bei der Verwendung dieser Speicherklasse der Nachteil, dass eine statische Analyse des Programmverhaltens nicht mehr möglich ist. Insbesondere kann keine statische Überprüfung stattfinden, ob die zugegriffenen Inhalte in den Cache geladen werden können.

4.2.3. Nicht-gecachet (not cached)

Zuletzt wird Anwendungen noch die Möglichkeit geboten, Speicher zu reservieren, der niemals in den Cache geladen wird. Die Zugriffszeiten für diese Speicherklasse sind relativ hoch und unvorhersagbar. Es können allerdings nahezu beliebig große Speicherbereiche reserviert und genutzt werden, die nur durch die konfigurierbare Größe des Pool für nicht-gecachete Inhalte begrenzt sind. Die Verwendung dieser Speicherklasse ist besonders im Kontext der Mixed-Criticality Systeme sinnvoll. Dabei kann es z. B. von Interesse sein Messdaten in großem Rahmen abzuspeichern und Optimierungen mithilfe dieser Daten zu berechnen. Diese Aufgabe kann ein separater Prozessorkern übernehmen. Die Messdaten können in einem nichtgecacheten Speicherbereich abgelegt werden und die Optimierung würde somit die

zeitlichen Eigenschaften der anderen parallel laufenden Programme nicht beeinflussen.

4.3. Micro-Heaps und Component-Heaps

Damit die Leistungsfähigkeit der Speicherverwaltung erhöht wird, muss der reservierte Speicher möglichst fragmentierungsfrei platziert werden, damit beim Laden möglichst wenige Cache-Ways benötigt werden. In Abbildung 3.1 ist zu erkennen, dass einiger Speicherplatz bisher ungenutzt verbleibt. Aufgrund der Ausrichtung der Komponenten an 64 kB Grenzen im Speicher, liegt am Ende einer Komponente immer ein ungenutzter Speicherbereich, der allerdings immer zusammen mit der Komponente in den Cache geladen wird. Da Komponenten häufig relativ klein sind, eignet sich dieser freie Bereich um Speicheranfragen der Komponente dorthin abzulegen. Die Speicherverwaltung füllt für jede Komponente diesen Bereich mit einem Micro-Heap auf. Für alle weiteren Speicheranfragen werden am Ende des kompilierten Betriebssystems zwei Speicherpools angelegt.

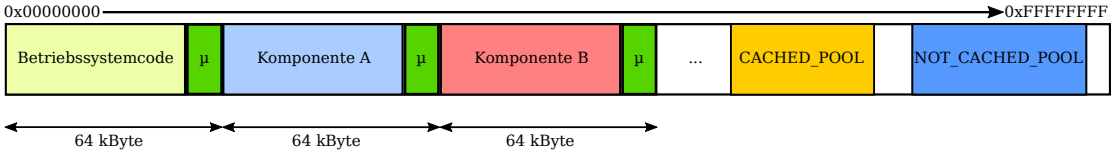

Abbildung 4.2.: Micro-Heaps und Speicherpools

Alle als μ dargestellten Speicherbereiche, stehen für einen Micro-Heap.

Speicheranfragen für nicht-gecacheten Speicher werden immer in den Pool für nicht-gecachete Inhalte gelegt. Dieser wird durch die MMU davor geschützt in den Cache geladen zu werden. Dadurch wird verhindert, dass, während ein Cache-Way entsperrt ist, parallele Zugriffe auf einem anderen Kern in diesen Cache-Way abgelegt werden.
Anfragen für immer-gecacheten Speicher werden zuerst in den Micro-Heap der Komponente gelegt. Falls dieser nicht mehr ausreichend Platz bietet, wird im Pool für gecachete Inhalte ein Component-Heap für die reservierende Komponente angelegt. Dieser hat eine exakte Größe von 64 kB, um in einen Cache-Way geladen werden zu können. Falls auch dieser voll ist, wird ein weiterer Component-Heap angelegt. Die Component-Heaps werden untereinander verkettet, um von einer Komponente jederzeit auf ihre Component-Heaps zugreifen zu können. Dies wird in Abbildung 4.3 veranschaulicht.
Prinzipiell könnten der Pool für gecachete Inhalte und der Pool für nicht-gecachete

Inhalte auch vereinigt werden, da sich der MMU Schutz vor dem Cachen auf einer Granularität von 64 kB einrichten lässt. Davon wird hier allerdings Abstand genommen, da in dem Fall häufig Wartungsoperationen, wie Cache- und TLB-Invalidations anfallen würden. Diese können im Mehrkernbetrieb zu weiteren Problemen führen, die gesondert abgefangen bzw. behandelt werden müssten.

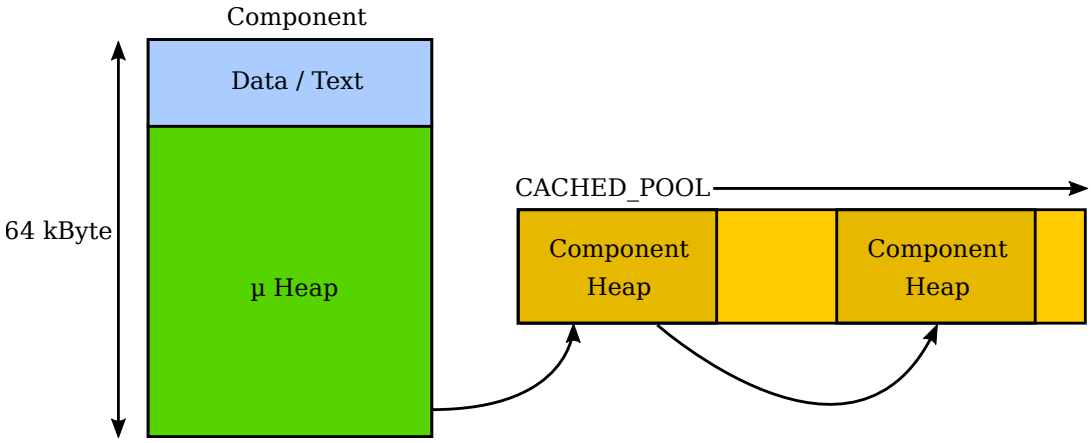

Abbildung 4.3.: Verkettung der Micro-Heaps und Component-Heaps

Jeder Komponente werden ihre eigenen Component-Heaps zugeordnet. In einem Component-Heap liegen daher auch nie Speicherbereiche von verschiedenen Komponenten. Damit wird die physikalische Bindung von reserviertem Speicher an die Komponente sichergestellt. Wenn der Eventhandler reservierten Speicher für eine Komponente lädt, so werden die Component-Heaps genauso wie Komponenten selber in einen freien Cache-Way geladen.

4.4. Freispeicherverwaltung

Die Freispeicherverwaltung funktioniert sowohl in den Micro-Heaps, als auch in den Component-Heaps, als auch in dem Pool für nicht-gecachete Inhalte analog. Die Verwaltungsinformationen werden in den reservierten Speicher selbst geschrieben. Dadurch ist es nicht notwendig, die Verwaltungsinformationen zentral zu speichern und diesen zentralen Speicher in den Cache zu laden. Zudem reicht bei dieser Art der Verwaltung die Adresse des reservierten Speichers aus, um alle Verwaltungsinformationen abzurufen.

4.5. Schnittstelle

Die Speicherverwaltung selbst ist keine Anwendungskomponente. Dadurch muss nicht jedes Mal, wenn Speicher reserviert wird eine zusätzliche Komponente in den Cache geladen werden. Des weiteren ist der kompilierte Quellcode der Speicherverwaltung verhältnismäßig klein und würde in einer eigenen Komponente Speicherplatz verschwenden. Aus diesen Gründen wird der Code der Speicherverwaltung den Betriebssystemfunktionen zugeordnet und beim Linken in eben diesen 64 kB Block gelegt.
Anwendungen können vier Funktionen der Speicherverwaltung aufrufen.

`allocate`

```
enum MemoryType {FULLY_CACHED, NOT_CACHED, DYNAMIC_CACHED}
uintptr_t allocate(OSC *osc, dword_t size, MemoryType type)
```

Mit dieser Funktionen können Anwendungskomponenten Speicherbereiche reservieren. Mithilfe des `type` Parameter kann bestimmt werden, welcher Speicherklasse der reservierte Speicher zugeordnet werden soll. Der `osc` Parameter ist eine Referenz auf die reservierende Komponente, damit der Speicher physikalisch und logisch dieser Komponente zugeordnet werden kann. Zudem kann über den `osc` Parameter ein Zugriff auf den Micro-Heap und die verketteten Component-Heaps erfolgen.

`free`

```
void free(uintptr_t memory)
```

Diese Funktion gibt einen reservierten Speicherbereich frei. Dabei werden in dem entsprechenden Speicherblock Regionen gesucht, die vor oder hinter der freizugebenden liegen und eventuell verschmolzen, damit die entstehende Lücke auch für größere Speicheranfragen genutzt werden kann. Wird durch die Freigabe ein Component-Heap geleert, so wird dieser aus der Verkettung entfernt und der Platz im Pool für gecachete Inhalte wieder freigegeben.

Speicherabhängigkeiten in Triggern

```
void addTriggerDependency(OSC *osc,
EventHandling::Trigger *trigger, uintptr_t memory)
void removeTriggerDependency(OSC *osc,
EventHandling::Trigger *trigger, uintptr_t memory)
```

Diese beiden Funktionen können genutzt werden um Speicherabhängigkeiten für beliebige Trigger einzutragen und wieder zu entfernen.

Mithilfe der `allocate` und `free` Funktion werden auch der `new` und `delete` Operator von C++ implementiert. Dies ermöglicht es den Anwendungskomponenten, dann Objekte zur Laufzeit anzulegen und zu verwalten. Die Speicherabhängigkeiten kann die Komponente ebenfalls für erzeugte Objekte angeben.

5. Implementierung

Die Quellcodes für die Speicherverwaltung sind größtenteils separat in einigen Klassen angelegt. Da die Speicherverwaltung eine Interaktion mit CyPhOS erfordert, befinden sich einige Teile des Quellcodes auch in den Quellcodedateien von CyPhOS. Die bestehenden Linkerskripte wurden so erweitert, dass der kompilierte Code der Speicherverwaltung in die Betriebssystemkomponente integriert wird.

5.1. Anlegen der Speicherpools

Die beiden Speicherpools für gecacheten (CACHED_POOL) und nicht-gecacheten Speicher (NOT_CACHED_POOL) müssen hinter dem Ende des kompilierten Betriebssystems angelegt werden. Die Platzierung des Codes in den Speicher erfolgt in drei Linkerskripten, die alle unter `arch/armv7/cortexa9` liegen. Das Skript `ldscript-end.ld` platziert die letzten Inhalte des Betriebssystems am Ende des Speichers. Am Ende dieses Skripts wird noch ein Symbol abgelegt, dessen Adresse im Quellcode bestimmt werden kann. Ab der Adresse dieses Symbols wird der Hauptspeicher von CyPhOS nicht mehr verwendet und die Speicherverwaltung kann die beiden Speicherpools dort anlegen. Das Berechnen der Startadressen der Pools erfolgt aus dem Symbol und der definierten Größenangabe der Pools im Konstruktor der Klasse `MemoryAllocator` [1].
In den Pool für gecacheten Speicher werden immer Component-Heaps der Größe 64 kB abgelegt. Zur Verwaltung dieses Pools wird eine Bitmaske angelegt, die für jeden 64 kB Abschnitt des Pool für gecachete Inhalte angibt, ob dieser durch einen Component-Heap belegt ist, oder ob er noch frei ist.
Hinter den beiden Speicherpools wird eine Seitentabelle angelegt, die für die Implementierung des dynamisch gecacheten Speichers (5.5) benötigt wird.

5.2. Micro-Heaps und Component-Heaps

Die Verwaltung der Micro-Heaps und Component-Heaps wird durch die Klassen `MicroHeap` [2] und `ComponentHeap` [3] bereitgestellt. Die Instanzen dieser Klassen

[1] `dynamicmemory/MemoryAllocator.{h,cc}`
[2] `dynamicmemory/MicroHeap.{h,cc}`
[3] `dynamicmemory/ComponentHeap.{h,cc}`

speichern maßgeblich die Startadresse und Endadresse eines freien Speicherbereichs ab. Die Klasse `OSC`, welche das Grundgerüst für eine Komponente liefert, hat einen Micro-Heap als Attribut. In den Makros zur Erstellung einer Komponente unter `component/OSC.h` wird dieses Attribut mit den spezifischen Speicheradressen für die Komponente initialisiert. Durch das Linkerskript wird am Ende des kompilierten Codes einer jeden Komponente ein Symbol platziert. Die Adresse dieses Symbols entspricht der Startadresse des Micro-Heaps. Die Endadresse des Micro-Heaps kann aus der Startadresse der Komponente und der Cache-Way Größe berechnet werden. Die Speicherverwaltung kann über die Methode `MicroHeap *OSC::getMicroHeap()` auf die Instanz des Micro-Heaps zugreifen und die Startadresse auslesen, ab der reservierter Speicher platziert werden kann.
Die Klasse `ComponentHeap` funktioniert ähnlich wie die Klasse `MicroHeap`. Instanzen dieser Klasse werden von der Speicherverwaltung im Pool für gecachete Inhalte angelegt. Dazu kommt der `placement-new` Operator zum Einsatz, der ein Objekt an einer vorgegeben Speicheradresse zur Laufzeit erstellt (5.7). Neben der Startadresse des freien Speichers, stellt `ComponentHeap` noch eine Methode zur Verfügung, die die Startadresse des gesamten Component-Heap, also mit der gespeicherten Instanz der Klasse, angibt. Diese Informationen werden beim Laden eines Component-Heap in den Cache benötigt.
Eine Instanz eines Micro-Heap speichert die Referenz `nextComponentHeap` ab, mit der die Speicherverwaltung auf die an eine Komponente geketteten Component-Heaps zugreifen kann. Jede Instanz eines Component-Heaps speichert die Referenz `next`, mit der auf weitere Component-Heaps zugegriffen werden kann.
Eine Übersicht der Funktionen, Adressen und Referenzen ist in Abbildung 5.1 dargestellt.

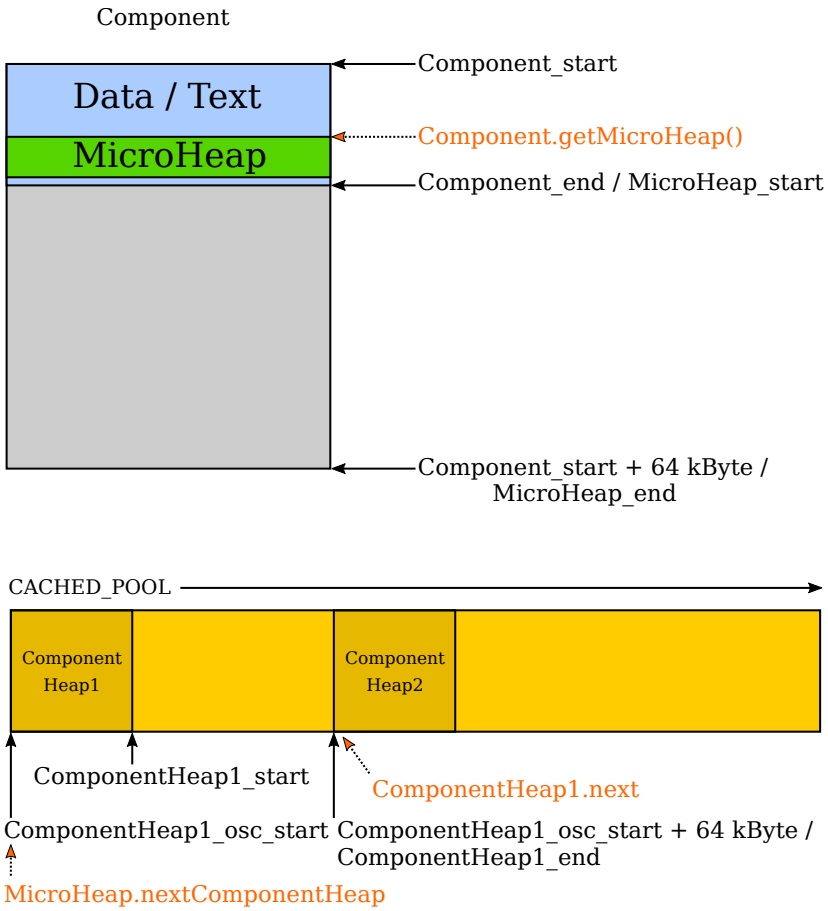

Abbildung 5.1.: Adressübersicht der Heaps

Das Beispiel zeigt eine Komponente, die zwei Component-Heaps hat. Referenzen sind in orange dargestellt, Adressen in schwarz.

5.3. Freispeicherverwaltung

5.3.1. Belegung von Speicher

Die Freispeicherverwaltung funktioniert für alle verschiedenen Speichersektionen gleichermaßen. Dazu gehören alle Micro-Heaps, alle Component-Heaps und der gesamte Pool für nicht gecachete Inhalte. Die Verwaltungsinformationen werden vor den reservierten Speicher geschrieben. Dadurch wird kein extra Speicher für die Verwaltung benötigt, die Verwaltungsinformationen werden automatisch mit gecachet und die Verwaltung bleibt dynamisch. Für die Verwaltungsinformationen steht das `struct memory_allocator_unit` [4] bereit. Jede Instanz dieser Datenstruktur enthält folgende Informationen:

1. Eine Variable, die angibt, ob der Speicherbereich frei oder benutzt ist
2. Eine Referenz auf den Heap, in dem sich dieser Speicherbereich befindet
3. Eine Größenangabe, wie groß der freie Speicherbereich ist
4. Eine Größenangabe des davor liegenden Speicherbereichs
5. Eine Spinlock-Instanz, über die eine Synchronisierung der Speicherzugriffe erfolgt

Jeder Micro-Heap, jeder neu erstellte Component-Heap und der Pool für nichtgecachete Inhalte werden mit einem großen Speicherblock initialisiert. Dieser wird auf nicht benutzt gesetzt. Die Größe des Bereichs errechnet sich aus den Start- und Endangaben des Heaps. Sobald Speicher reserviert werden soll, iteriert die Speicherverwaltung über die Speichersektionen. Mithilfe der Größenangabe in der Verwaltungseinheit kann direkt zu der nächsten Verwaltungseinheit gesprungen werden, die den nächsten Block verwaltet. Sobald ein Block gefunden wird, der ausreichend groß ist und der nicht verwendet wird, wird dieser Block als verwendet markiert. Anschließend wird überprüft, ob die verbleibende Größe des freien Speichers ausreichend Platz für eine weitere Verwaltungseinheit und die kleinste reservierbare Einheit Speicher bietet. Ist dies der Fall, so wird am Ende des Blocks ein weiterer Block erstellt, in den Speicheranfragen abgelegt werden können.
Je nach angeforderter Speicherklasse, werden verschiedenen Heaps betrachtet:

- Immer im Cache: Zuerst wird nach einem freien Platz im Micro-Heap der Komponente gesucht. Ist dort kein freier Platz, werden alle verketteten Component-Heaps durchsucht. Ist dort auch kein freier Platz, wird ein neuer Component-Heap erstellt und durchsucht. Kann kein neuer Component-Heap erstellt werden, wird die Allokation abgebrochen.

[4] `dynamicmemory/MemoryPool.h`

- Dynamisch gecachet: Der Micro-Heap wird ignoriert, da diese Speicherbereiche nicht durch die MMU überwacht werden. Das Vorgehen erfolgt ansonsten analog zur ersten Speicherklasse.
- Nie gecachet: Der gesamte Pool für nicht-gecachete Inhalte wird wie ein Heap verwaltet. Es wird ein freier Block gesucht. Kann keiner gefunden werden, so wird die Allokation abgebrochen.

Die Verwaltung der Speicherblocks in einem beliebigen Heap wird durch die Funktion `searchInHeap` [5] implementiert.

5.3.2. Freigabe von Speicher

Wird ein benutzter Speicherbereich freigegeben, so überprüft die Speicherverwaltung zuerst, in welchen Heap sich der Speicherblock befindet. Da die Verwaltungsinformationen immer vor den reservierten Speicher geschrieben werden, errechnet sich die Speicherverwaltung die Adresse der zugehörigen Verwaltungseinheit und setzt diese auf nicht benutzt. Mithilfe der Größenangabe des vorherigen Blocks, kann die Speicherverwaltung auch auf die Verwaltungseinheit des vorherigen Blocks zugreifen. Nach dem Freigeben wird sowohl nach vorne, als auch nach hinten iteriert, und eine Folge von nicht belegten Speicherblocks zu einem größeren Block zusammengefasst. Dadurch können später auch größere Speicheranfragen in diesem Block abgelegt werden und die Fragmentierung wird verringert. Diese Funktionalität übernimmt die Funktion `freeMAU` [6].
Sobald der Speicherblock freigegeben wurde, findet noch eine Überprüfung statt, ob sich der Speicherbereich in einem Component-Heap befunden hat und ob dieser Component-Heap durch das Freigeben leer geworden ist. Ist dies der Fall, so wird der Platz des Component-Heaps im Pool für gecachete Inhalte wieder freigegeben und der Component-Heap wird aus der Verkettung der anderen Component-Heaps entfernt.

[5] `dynamicmemory/MemoryAllocatorSystem.cc`
[6] `dynamicmemory/MemoryAllocatorSystem.cc`

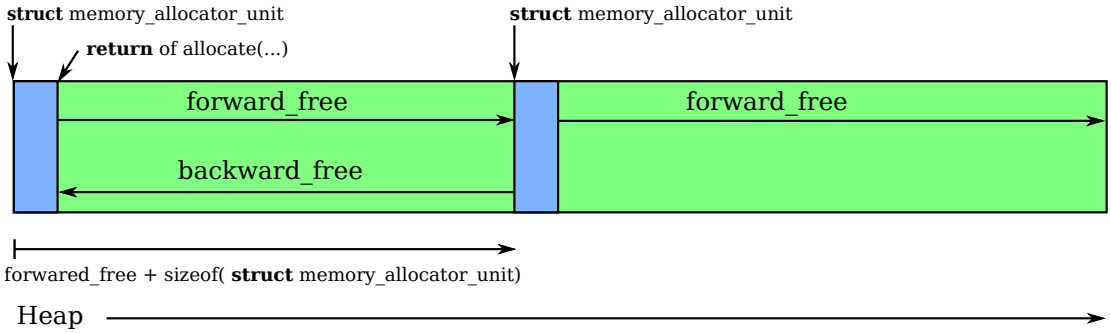

Abbildung 5.2.: Verwaltungseinheiten in einem Heap

Blau dargestellt sind die Instanzen der Verwaltungseinheit. Grün dargestellt ist der für die Anwendungen verfügbare Speicher.

5.4. Cachen von Inhalten

Das aktive Cachen von Speicherbereichen erfolgt immer dann, wenn ein Trigger zur Ausführung kommen soll, für den Speicherabhängigkeiten eingetragen sind, oder wenn ein Zugriff auf einen durch die MMU überwachten Bereich erfolgt. In beiden Fällen wird immer ein gesamter Component-Heap in den Cache geladen. Speicherabhängigkeiten die für Speicherbereiche in einem Micro-Heap eingetragen sind können ignoriert werden, da der Micro-Heap bereits zusammen mit der zugehörigen Komponente geladen wurde.
Das Laden eines Component-Heap erfolgt analog zu dem Laden einer Komponente. Zuerst wird ein freier Cache-Way gesucht. Kann keiner gefunden werden, so wird gemäß der im Betriebssystem vorgegebenen Verdrängungsstrategie eine andere Komponente oder ein anderer Component-Heap verdrängt. Dieser Cache-Way wird entsperrt, sodass der Hardware Cache-Controller Zugriffe in diesen Cache-Way ablegen kann. Anschließend werden für alle Speicherworte des Component-Heaps Preload-Instruktionen aufgerufen, sodass der Hardware-Cache-Controller den Component-Heap gezwungenermaßen in den freien Cache-Way legt. Sind alle Preload-Instruktionen abgeschlossen, so wird der Cache-Way wieder gesperrt. Diese Funktionalität implementiert die Funktion `preloadOSCHeap` [7].

[7] `arch/armv7/driver/ARMv7CacheControl_critical.cc`

5.5. Dynamisch gecachete Speicher (Access-Flag)

Wird "dynamisch gecachet" bei der `allocate` Funktion angefragt, so ist das Programmverhalten nahezu identisch, als wenn immer gecacheter Speicher angefragt wird. Der einzige Unterschied besteht darin, dass der Micro-Heap der Komponente nicht in Betracht gezogen wird, um die Speicheranfrage abzulegen. Dies ist dadurch bedingt, dass nur der Pool für gecachete Inhalte durch die MMU überwacht wird und nur dort Speicherzugriffe festgestellt werden können. Wird eine Speicherabhängigkeit für einen dynamisch gecacheten Speicherbereich angegeben, so wird der entsprechende Component-Heap ebenso vor der Ausführung des Trigger vorgeladen, wie bei immer gecachetem Speicher. Ebenso werden auch Component-Heaps auf Zugriffe überwacht, in denen immer gecacheter Speicher liegt und für diese keine Speicherabhängigkeiten eingetragen sind. Das dynamische Laden in den Cache kann allerdings nicht für alle Speicherbereiche aus der Klasse "immer im Cache" sichergestellt werden, da immer gecachete Speicherbereiche auch in den Micro-Heaps liegen können.
Zur Überwachung des Pool für gecachete Inhalte kommt das Access-Flag zum Einsatz [1]. Das Access-Flag ist ein Bit im Seitentabelleneintrag einer Speichersektion. Wird ein Eintrag mit diesem Bit auf 0 gesetzt in den *Translation-Look-Aside-Buffer (TLB)* geladen, so wird ein Access-Flag-fault ausgelöst, welches durch eine Unterbrechungsbehandlung behandelt werden kann. Das Access-Flag kann nicht standardmäßig genutzt werden. Die Voraussetzungen sind eine aktivierte MMU mit einer Seitentabelle und das explizite Aktivieren des Access-Flag durch das Systemkontrollregister. CyPhOS verwendet standardmäßig eine Seitentabelle mit einer Seitengröße von 1 MB. Dies ist zur Überwachung einzelner Component-Heaps mit einer Größe von 64 kB ungeeignet. Daher muss eine Second-level Seitentabelle eingerichtet werden, die 64 kB große Seiten verwaltet und für jede Seite ein Access-Flag bereitstellt.

5.5.1. Aktivieren des Access-Flag

CyPhOS stellt bereits Funktionen zur Verfügung, die die Seitentabelle aufsetzen und die MMU aktivieren. Zuerst muss die bestehende Seitentabelle so verändert werden, dass alle Access-Flags auf 1 stehen, damit nicht direkt nach der Aktivierung des Access-Flag Access-Flag-faults ausgelöst werden. Dies wird in der Funktion `resetAllAccessFlags` [8] erledigt. In der bereits vorhandenen Funktion `enableMMU` wird das Systemkontrollregister geschrieben, um das Access-Flag zu aktivieren. CyPhOS ruft diese Funktion bereits auf allen Kernen auf.

[8] `arch/armv7/driver/ARMMMU.cc`

Zuletzt muss noch die Second-level Seitentabelle eingerichtet werden, die eine Granularität von 64 kB ermöglicht. Die Adressen des Pool für gecachete Inhalte werden so ausgelegt, dass sie an 1 MB Grenzen ausgerichtet sind. In der first-level Seitentabelle muss für jede 1 MB Seite ein Referenzeintrag auf die second-level Seitentabelle eingerichtet werden. Die Ausrichtung des Pool für gecachete Inhalte an 1 MB Grenzen ermöglicht es also eine second-level Seitentabelle nur für den Pool für gecachete Inhalte einzurichten. Sowohl das Aufsetzen der second-level Tabelle als auch das Setzen der Referenzeinträge in der first-level Seitentabelle wird in der Funktion `init` [9] implementiert. Die Access-Flags der second-level Seitentabelle werden initial alle auf 0 gesetzt, da keine unkontrollierten Zugriffe in den Pool für gecachete Inhalte erfolgen.
Die Steuerung dieser Abläufe hintereinander erfolgt in der von CyPhOS zur Verfügung gestellten Funktion `init_primary_cpu_c` [10].

5.5.2. Nutzung des Access-Flag

Im initialen Zustand nach dem Start von CyPhOS sind alle Access-Flags des Pool für gecachete Inhalte auf 0 gesetzt. Demnach würden bei jedem Zugriff in diesen Pool Access-Flag-faults ausgelöst. Wird ein Component-Heap nun in den Cache geladen, so wird das Access-Flag auf 1 gesetzt. Wird dieser Pool gemäß der Verdrängungsstrategie wieder aus dem Cache verdrängt, so wird das Access-Flag wieder auf 0 gesetzt. Beides erfolgt in den Funktionen `preloadOSCHeap` [11] und `getLRUWay` [12]. Während der Erstellung eines Component-Heap wird das Access-Flag ebenfalls auf 1 gesetzt, da Zugriffe in den Heap nötig sind, aber keine Erforderlichkeit besteht den Heap in der nächsten Zeit in den Cache zu laden. Dadurch erfolgen zwar Hauptspeicherzugriffe, diese haben an dieser Stelle allerdings keine negative Auswirkung, da das Zeitverhalten beim Allozieren keine Zeitgarantie zur Verfügung stellt. Auch Interferenzen mit dem Preloading können nicht auftreten, da ein Spinlock sicherstellt, dass während einer Speicherallokation kein Preload stattfindet.

5.5.3. Behandlung des Access-Flag-fault

Wird ein Access-Flag-fault ausgelöst, so wird eine Data-abort-exception ausgelöst, für die CyPhOS eine rudimentäre Behandlungsroutine anbietet. Diese Routine wird so erweitert, dass sie die Funktion `handleDataAbortException` [13] aufruft.

[9] `dynamicmemory/MemoryAllocatorSystem.cc`
[10] `arch/armv7/cortexa9/boot/initOS.cc`
[11] `arch/armv7/driver/ARMv7CacheControl_critical.cc`
[12] `arch/armv7/driver/ARMv7CacheControl_critical.cc`
[13] `dynamicmemory/MemoryAllocatorSystem.cc`

Das data-fault-address-register (DFAR) beinhaltet die Adresse, die zu dem Data-abort geführt hat. Dem data-fault-status-register (DFSR) kann entnommen werden, ob die data-abort-exception durch ein Access-Flag-fault ausgelöst wurde. Die Behandlungsfunktion liest diese beiden Register aus und überprüft ob es sich um ein Access-Flag-fault handelt und ob die Fehler-verursachende Adresse aus dem Pool für gecachete Inhalte stammt. Ist dies der Fall, wird der Component-Heap, aus dem diese Adresse stammt, mit der Funktion `preloadOSCHeap` in einen freien Cache-Way geladen. Dadurch wird das Access-Flag dieses Heaps auf 1 gesetzt. Nach dem zurückkehren in die Behandlungsroutine von CyPhOS wird der Programmzähler auf seinen gesicherten Wert gesetzt und das Programm an der Stelle fortgesetzt, an der die Unterbrechung aufgetreten ist.

5.6. Nicht gecacheter Speicher

Alle Aufrufe der `allocate` Funktion, die nicht gecacheten Speicher anfordern werden in den Pool für nicht-gecachete Inhalte abgelegt. Dieser wird genauso verwaltet wie ein Heap. In der `init` Funktion, die während des Startvorgangs aufgerufen wird, wird ein Schutz für den Pool für nicht-gecachete Inhalte eingerichtet. In der first-level Seitentabelle wird in jedem Tabelleneintrag, der auf den Pool für nicht-gecachete Inhalte verweist, ein Bit gesetzt welches das Cachen dieser Sektion verbietet. So können im Mehrkern-Betrieb Anwendungen weiterhin auf nicht gecacheten Speicher zugreifen, ohne das Preloading auf einem anderen Kern zu stören. Für nicht gecacheten Speicher erfolgt keine Zugriffskontrolle und kein aktives Vorladen.

5.7. `new`-Operator

CyPhOS ist zum größten Teil in C++ geschrieben. Daher ist es für Anwendungskomponenten möglich Klassen zu definieren und Objekte dieser Klassen zu erstellen. Ohne die dynamische Speicherverwaltung müssen zur Übersetzungszeit alle Objekte feststehen. Durch den `new`-Operator ist es möglich Objekte auch dynamisch zur Laufzeit zu erstellen. Dabei ist zwischen dem `new`-Operator und dem `new`-Ausdruck zu unterscheiden [5]. Wird das `new` Schlüsselwort in einer Programmzeile verwendet, so interpretiert der Compiler dies als den `new`-Ausdruck. Der `new`-Ausdruck benötigt den `new`-Operator, um den für das Objekt benötigten Speicherplatz zu reservieren. Durch den `new`-Ausdruck wird in dem reservierten Speicher anschließend das Objekt angelegt. Spezifisch implementiert werden muss nur der `new`-Operator, der den Speicher reserviert. Dies erfolgt in der Datei `common/new.{h, cc}`. Die `new`-Operatoren rufen lediglich die `allocate` Funktion der Speicherverwaltung auf, geben die entsprechenden Argumente weiter und

geben einen Pointer auf den reservierten Speicher zurück. Zusätzlich wird noch ein `placement-new` Operator definiert, dem als Parameter bereits eine Adresse angegeben werden kann, an die das Objekt gelegt wird. Des Weiteren gibt es einen `new`-Operator, dem als Parameter zusätzlich ein Pointer auf eine Adressvariable übergeben werden kann. In diese Variable schreibt der Operator die reservierte Speicheradresse. Dies ist wichtig, da der `new`-Ausdruck, der in einem Anwendungsprogramm steht, nicht zwangsweise einen Pointer auf den reservierten Speicherbereich liefert. C++ platziert die Objekte teilweise mit einem Offset in dem reservierten Speicher und gibt einen Pointer auf das Objekt zurück. Zum Angeben von Speicherabhängigkeiten oder zum Freigeben von Speicher wird aber der Beginn der reservierten Sektion benötigt.
Der `delete`-Operator ist analog mit der free Funktion implementiert.
Zur direkten und einfachen Nutzung der Speicherverwaltung und des `new`-Operator, wird eine doppelt verkettete Liste implementiert, die den `new`-Operator zum Erstellen von neuen Elementen verwendet. Die Implementierung dieser Liste erfolgt in der Datei `common/LinkedList.{h, cc}`.

5.8. Speicherabhängigkeiten

Das unter Kapitel 4 dargestellte Konzept der Speicherklassen wird im Bezug auf die Speicherabhängigkeiten in der Implementierung erweitert. Wird eine Speicherabhängigkeit angegeben, so wird der entsprechende Speicher vor der Ausführung des Trigger auf jeden Fall in den Cache geladen. Es ist demnach möglich auch Speicher, der aus der Klasse “dynamisch gecachet” reserviert wurde, nicht erst bei einem Zugriff in den Cache zu laden. Wird für einen Speicherbereich aus der Klasse “immer im Cache” keine Abhängigkeit angegeben, so wird dieser Speicher auch nicht vor der Ausführung des Trigger in den Cache geladen.
Speicherabhängigkeiten können für immer gecachete und dynamisch gecachete Speicherbereiche angegeben werden. Die `addTriggerDependency` [14] Funktion betrachtet hierbei nur Speicheradressen, die aus dem Pool für gecachete Inhalte stammen. Soll eine Speicherabhängigkeit definiert werden, die aus einem Micro-Heap einer anderen Komponente stammt, so muss diese Komponente sowieso als Abhängigkeit des Trigger der ersten Komponente angegeben werden und wird somit bei der Ausführung dieses Trigger in den Cache geladen. CyPhOS verlangt bei der Initialisierung eines Trigger bereits ein Array von Komponenten, die in den Cache geladen werden müssen wenn der Trigger zur Ausführung kommt. Abhängigkeiten für Speicher im Micro-Heap können auf diese Art definiert werden.
Jeder Trigger hat eine Liste als Attribut, in der die Speicherabhängigkeiten gespeichert werden, die über die `addTriggerDependency` Funktion angegeben werden.

[14] `dynamicmemory/MemoryAllocator.cc`

Der dynamisch allozierte Speicher für diese Listen wird immer aus der Kategorie "immer im Cache" in der Betriebssystemkomponente reserviert. In der Funktion `preloadOSC` [15] wird über die Liste des Trigger iteriert, der zur Ausführung kommen soll. Alle Elemente führen zu einem Aufruf der `preloadOSCHeap` Funktion, sodass alle abhängig markierten Speicher garantiert im Cache liegen, wenn der Trigger ausgeführt wird. Am Ende des Aufrufs der `addTriggerDependency` Funktion, wird die `preloadOSC` Funktion zusätzlich manuell aufgerufen, denn es besteht die Möglichkeit, dass eine Speicherabhängigkeit definiert wurde, die auf den Trigger zutrifft, in dem diese Definition stattfindet. Bevor die nächste Instruktion dieses Trigger stattfinden darf, muss diese Abhängigkeit also erfüllt werden.
Neben dem Preloading in den Cache, sollen die Speicherabhängigkeiten noch eine Synchronisierungsfunktion erfüllen. Das heißt, es dürfen nicht zwei Trigger gleichzeitig ausgeführt werden, die eine gemeinsame Speicherabhängigkeit haben. Dies wird durch einen Spinlock in jedem reservierten Speicherabschnitt realisiert. Der Scheduler ruft die Funktion `tryTaskLock` [16] auf. Nur wenn diese Funktion true zurückgibt, kommt der entsprechende Trigger zur Ausführung. Diese Funktion wird so erweitert, dass sie versucht alle Spinlocks der Speicherabhängigkeiten für diesen Trigger zu belegen. Ist dies nicht erfolgreich, werden die belegten Spinlocks wieder gelöst und der Vorgang abgebrochen. Da diese Prozedur für jeden Trigger ausgeführt wird, ist die Synchronisierung sichergestellt. Um die Funktionalität zu gewährleisten werden alle Locks nach der Ausführung des Trigger wieder freigegeben. Dies geschieht in der Funktion `handlerFinished` [17].
Für Speicherbereiche für die keine Speicherabhängigkeiten angegeben werden, wird systemseitig keine Synchronisierung bereitgestellt.

5.8.1. Kohärenz im Translation-Look-Aside-Buffer

Jeder CPU-Kern hat einen eigenen TLB. Seitentabelleneinträge mit einem Access-Flag von 0 werden niemals in den TLB geladen. Wird im Betriebssystem nun ein Component-Heap verdrängt und sein Access-Flag wieder auf 0 gesetzt, so ist es möglich, dass in einigen TLBs noch der fehlerhafte Eintrag steht und auch nicht aktualisiert wird. Um dem Vorzubeugen, wird der TLB nach dem Verdrängen auf allen Kernen invalidiert. Dazu kommt das Maintenance-Operation-Broadcasting zum Einsatz, welches die Cortex-A9 Spezifikation implementiert [3]. Daraus ergibt sich ein weiteres Problem. Finden auf einem Kern aktuell Speicherzugriffe statt, und wird dieser Speicher nun durch ein Programm auf einem anderen Kern verdrängt, so wird das Access-Flag für den Speicher, auf den aktuell zugegriffen wird, zurückgesetzt und der TLB geleert. Geschieht dies während einer Zugriffsoperation, so wird nicht ein erneutes Access-Flag-fault ausgelöst, sondern die Ausführung

[15] `arch/armv7/driver/ARMv7CacheControl_critical`
[16] `eventhandling/EventHandler.cc`
[17] `eventhandling/EventHandler.cc`

bricht mit einer prefetch-abort-exception ab. Dies kann nur verhindert werden, wenn sichergestellt ist, dass keine Inhalte verdrängt werden, auf die aktuell zugegriffen wird. Die Verwaltung der Cache-Ways verdrängt keine Speicherinhalte, die gelockt sind. Speicherinhalte werden aber nur gelockt, wenn eine Komponente zur Ausführung kommt, die diese Speicherinhalte als Abhängigkeit eingetragen hat. Der Anwendungsentwickler muss demnach sicherstellen, dass, wenn Komponenten zur Ausführung kommen, die Speicher verwenden, für die sie keine Abhängigkeit eingetragen haben, keine Verdrängungen stattfinden können. Da Verdrängungen hauptsächlich beim Kontextwechsel stattfinden, muss die Ausführungsreihenfolge eventuell synchronisiert werden.
Für die relevanten Systemfunktionen, die auch Speicher verdrängen würden (z. B. erfordert das Locken von Speicherabhängigkeiten vor der Ausführung einer Komponente Zugriffe in den Speicher selber, da dort die Locks platziert sind) wird das Access Flag auf dem aktuellen Kern deaktiviert und anschließend wieder aktiviert. Damit können die Systemfunktionen auf Speicher zugreifen, ohne dabei anderen Speicher verdrängen zu müssen.

6. Anwendungsbeispiel: CAN-Bus

Das folgende Kapitel beschreibt beispielhaft die Nutzung der Speicherverwaltung für eine Anwendungskomponente. In Abbildung 4.1 ist eine mögliche Anwendung der Speicherverwaltung bereits symbolisch dargestellt. Diese Anwendung ist exemplarisch in CyPhOS implementiert, um ein Beispiel für die Verwendung der Speicherverwaltung bereit zu stellen.

6.1. Der CAN-Treiber

Der CAN-Treiber ist bereits in CyPhOS in der Klasse `MCP2515SPI` [1] implementiert. Dieser bietet verschiedene Trigger an, um Nachrichten auf den CAN-Bus zu senden und um Diese zu empfangen. Zur Nutzung mit der Speicherverwaltung wird der Treiber um den Trigger `trigger_sendMessageFromMemory` erweitert. Diese erwartet als Argument die Speicheradresse einer vordefinierten Datenstruktur, in der alle Informationen über die zu sendende Nachricht abgelegt sind. Eine Instanz von dieser Datenstruktur kann von einer Komponente angelegt werden und anschließend die Adresse dem Trigger übergeben werden.

6.2. Anwendungskomponente: Gangwahlschalter

Die Klasse `GearSelector` [2] implementiert die Steuerung eines Gangwahlschalter über den CAN-Bus. Dazu verwendet diese den CAN-Treiber und den Trigger `trigger_sendMessageFromMemory`. Im Start Trigger der Klasse werden Speicherbereiche aus der Klasse "dynamisch-gecachet" für die verschiedene Nachrichten reserviert. Es werden Abhängigkeiten sowohl für den Trigger `trigger_sendMessageFromMemory`, als auch für den Trigger `receiveGear` angegeben. Dies sorgt dafür, dass der Speicher für die zu sendenden CAN-Nachrichten im Cache liegt, wenn die Klasse `GearSelector` diese mit Nachrichten beschreibt und wenn der CAN-Treiber aus dem Speicher liest, um die Nachricht zu versenden. Da der reservierte Speicher aus der Klasse "dynamisch gecachet" stammt, können die

[1] `arch/armv7/imx6/driver/MCP2515SPI.{h,cc}`
[2] `autolabcomponents/GearSelector.{h,cc}`

Inhalte nur aus Component-Heaps stammen. Die Anwendung braucht demnach keine Komponentenabhängigkeit für den Trigger zu definieren, die die gesamte `GearSelector` Komponente in den Cache laden würde, wenn der Trigger zur Ausführung kommt. Würde der Speicher aus der Klasse "immer im Cache" stammen, so müsste die Komponente `GearSelector` in das Abhängigkeitsarray des Trigger `trigger_sendMessageFromMemory` eingetragen werden. Dadurch würde bei jeder Ausführung des Trigger `trigger_sendMessageFromMemory` die Komponente `GearSelector` mit in den Cache geladen und somit auch die Speicherinhalte, die in ihrem Micro-Heap stehen.
Das Versenden einer Nachricht erfordert in dieser Implementierung mehrere Codezeilen, die sich für verschiedene Nachrichten kaum unterscheiden. Zum vereinfachten Versenden der Nachrichten steht daher das Makro `SEND_CAN_MESSAGE_WITH_MEMORY` [3] zur Verfügung.

[3] `arch/armv7/imx6/driver/MCP2515SPI.h`

7. Evaluation

Um die Leistungsfähigkeit der Speicherverwaltung zu evaluieren, werden folgende Punkte genauer untersucht.

1. Unterschiede in den Zugriffszeiten der verschiedenen Speicherklassen.
2. Veränderungen der Zugriffszeiten bei parallel stattfindenden Speicherzugriffen.
3. Vergleich der Zugriffszeiten mit reinem Hardware-Cache-Management
4. Verhalten der Vorladezeiten der Komponenten bei parallel stattfindenden Speicherzugriffen

7.1. Konstruierte Testumgebung

Zum Evaluieren der Speicherverwaltung werden Komponenten implementiert, die Speicher reservieren und auf Diesen zugreifen und dabei die Zugriffszeiten messen. Gesteuert werden diese Tests aus der Klasse `MemoryEvaluator` [1]. Ein einfaches Zugriffsmuster wird in der Klasse `MemoryEvalMultipool` [2] implementiert. Dabei werden 16 Pools einer bestimmten Größe reserviert und gemäß einer vorgegebenen, nicht sequentiellen Reihenfolge Inhalte von einem in den anderen Pool kopiert. Dadurch werden sequentielle Speicherzugriffe größtenteils vermieden und es können realistische Messwerte erzielt werden. Die Ergebnisse der Messung werden in einem reservierten Speicher aus der Klasse "nie im Cache" abgespeichert und nach dem Test über die serielle Schnittstelle übertragen. Durch das Ablegen der Messdaten im Pool für nicht-gecachete Inhalte wird die Cache-Verwaltung nicht zusätzlich belastet und die Messdaten nicht verfälscht.
Zur Untersuchung der Zugriffszeiten unter parallel stattfindenden Speicherzugriffen wird die Klasse `MemoryEvalDisturber` [3] verwendet. Diese liest aus einem reservierten Speicherpool Daten aus und schreibt diese nach einer Rechnung wieder in den Pool. Der Pool kann aus verschiedenen Speicherklassen stammen.
Kommt es bei den Tests zu Verdrängungen, so sind die durch die Implementierung

[1] `testcomponents/MemoryEvaluator.{h,cc}`
[2] `testcomponents/MemoryEvalMultipool.{h,cc}`
[3] `testcomponents/MemoryEvalDisturber.{h,cc}`

der Speicherverwaltung gegebenen Vorgaben zu beachten. Die Speicherinhalte, auf denen die Zeit gemessen wird, sind als Triggerabhängigkeit für den messenden Trigger eingetragen. Demnach hat das Vorladen des Speichers keinen Einfluss auf die Messung, da die Inhalte vor der Ausführung des Triggers und somit vor Beginn der Messung vorgeladen werden, und der Speicher kann während der Messung zudem nicht verdrängt werden. Wird ein Test durchgeführt, bei dem das System über seine Grenzen hinaus belastet wird, so werden für die störenden Speicherbereiche keine Abhängigkeiten eingetragen, da sie sonst nicht im Lauf eines Trigger aus dem Cache verdrängt werden können, um Platz für weitere Speicherbereiche zu schaffen. Dies führt aber auch dazu, dass die Störung erst gestartet werden kann, wenn die zu testende Komponente bereits läuft, da ansonsten durch das Preloading der Komponente ein Speicherbereich verdrängt werden könnte, auf den die Störung gerade zugreift.
Alle hier dargestellten Messungen sind auf einem Odroid U3 Board mit einem Samsung Exynos 4412 Prozessor durchgeführt.

7.2. Auswertung der Messungen

Da es sich bei CyPhOS um ein Echtzeitbetriebssystem handelt ist die Vorhersagbarkeit von Zugriffszeiten von besonderem Interesse. Daher wird in den folgenden Messungen insbesondere der Mittelwert und die Varianz untersucht. Für eine Prognose der Zugriffszeiten würde der Mittelwert als Erwartungswert herangezogen. Die durchschnittliche prozentuale Abweichung von diesem Erwartungswert wird im folgenden als Maß für die Vorhersagbarkeit aus dem Erwartungswert verglichen. Die prozentuale Abweichung ist durch den Variationskoeffizienten ($\frac{\text{Standardabweichung}}{\text{Mittelwert}}$) gegeben.
Für die gemessenen Werte werden einige statistische Kennzahlen (Mittelwert (μ), Varianz (σ^2), Standardabweichung (σ) und der Variationskoeffizient (σ/μ)) tabellarisch im Anhang A dargestellt. Zur Veranschaulichung der Daten kommen zusätzlich Boxplots und Histogramme zum Einsatz.
Bei den Messungen für gecacheten Speicher werden meistens zwei oder mehrere Häufungspunkte zu erkennen sein, um die die Zugriffszeiten schwanken. Diese lassen sich zum Teil durch Level-1 und Level-2 Hits erklären. Bei der Berechnung des Variationskoeffizienten ergibt sich dadurch eine höhere Schwankung als eigentlich berücksichtigt werden muss. Im Sinne der Vorhersagbarkeit von Zugriffszeiten sind Ausreißer nach unten prinzipiell unkritisch. Daher wird bei den entsprechenden Messungen der Variationskoeffizient auch immer für den größten Häufungspunkt der Messungen angegeben um eine Vergleichbarkeit mit den anderen Messungen zu ermöglichen.

7.3. Zugriffszeiten in den verschiedenen Speicherklassen

An dieser Stelle werden die Zugriffe ohne Störung untersucht. Dies entspricht einer Anwendung, die alleine auf dem System ausgeführt wird. Der Zugriff wird dabei auf 16 Pools der Größe 16 kB (16384 Byte) getestet. Die Messung wird für jede Speicherklasse 100000 mal durchgeführt. Die Ergebnisse werden in Taktzyklen für das Kopieren eines Bytes von einem in den anderen Pool umgerechnet.

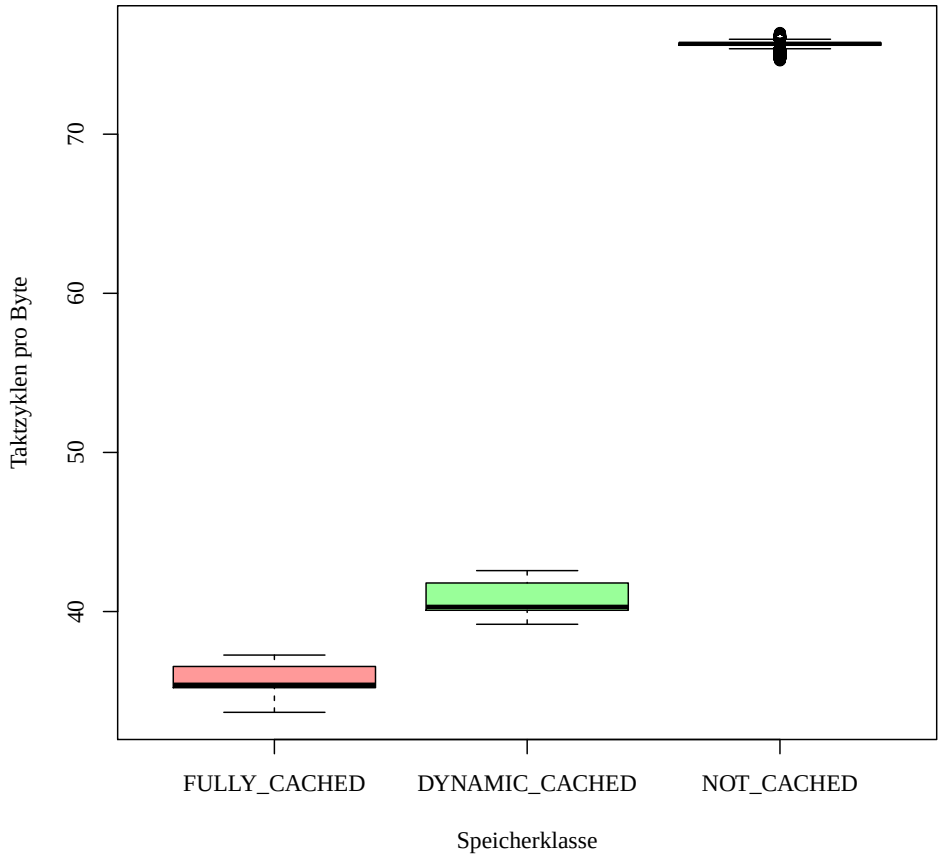

Abbildung 7.1.: Boxplots aller Zugriffszeiten

Deutlich zu erkennen ist der Unterschied zwischen gecachetem und nicht-gecachetem Speicher. Speicher, der in dem Pool für nicht-gecachete Inhalte abgelegt wird, wird weder in den L1-Cache, noch in den L2-Cache geladen.
Der Unterschied der Zugriffszeiten zwischen immer-gecachetem und dynamisch-gecachetem Speicher entsteht durch die unterschiedliche Platzierung der Inhalte. Bei der Allokation von immer-gecachetem Speicher werden einige Inhalte in den Micro-Heap der Komponente gelegt, andere Inhalte liegen in Component-Heaps im Pool für gecachete Inhalte . Bei der Allokation von dynamisch-gecacheten Inhalten liegen hingegen alle Inhalte in Component-Heaps im Pool für gecachete Inhalte. Die Speicherinhalte liegen aufgrund dessen anders strukturiert im Speicher. Verschiedene Effekte können beim Laden in den Cache dafür sorgen, dass eine unterschiedliche Ausrichtung der Inhalte zu unterschiedlichen Zugriffszeiten führt.

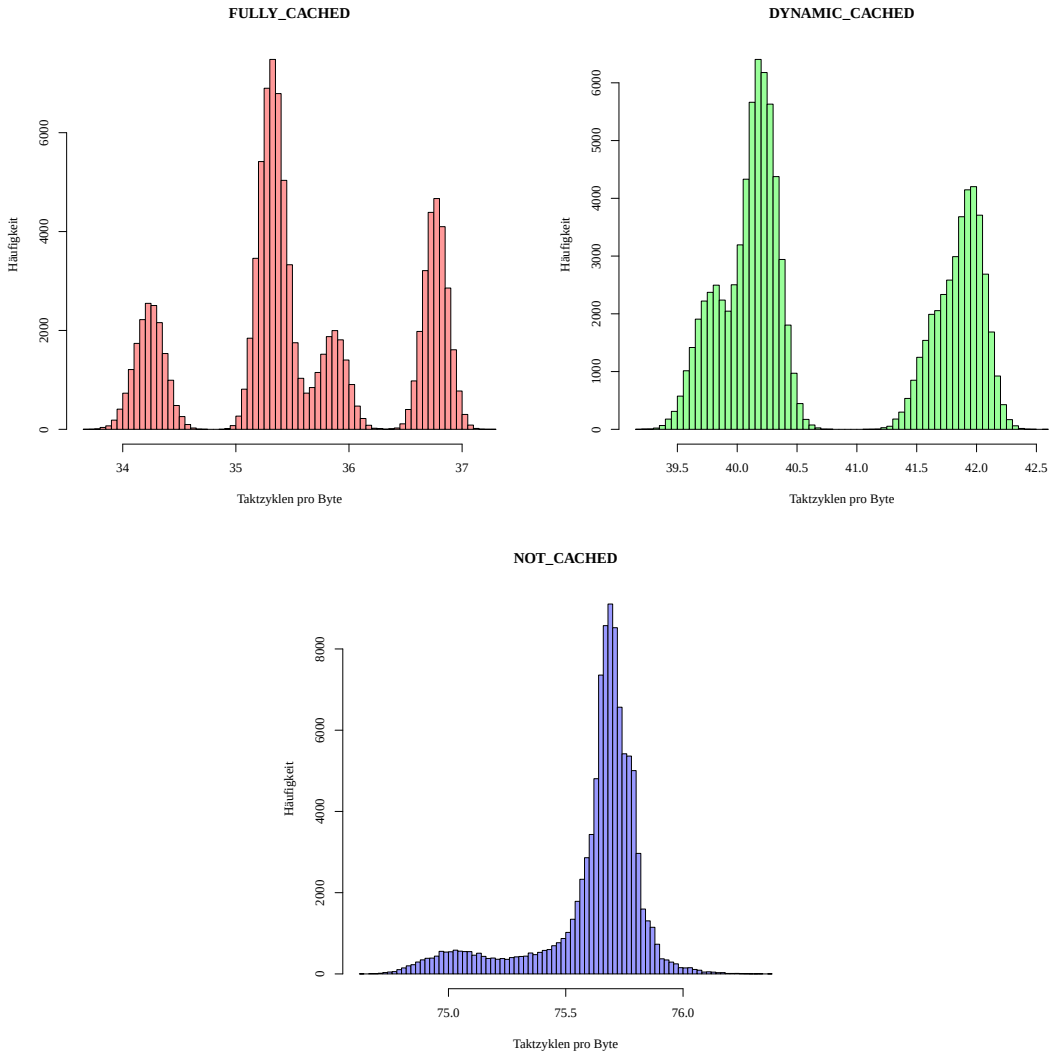

Abbildung 7.2.: Histogramme aller Zugriffszeiten

Aus den Messungen lässt sich erkennen, dass nicht-gecacheter Speicher sowohl im Mittelwert, als auch in der Schwankung deutlich schlechter abschneidet als gecacheter Speicher. Dynamisch gecacheter Speicher weist etwas höhere Zugriffszeiten und eine etwas höhere Schwankung auf, als immer-gecacheter Speicher. Die durchschnittlich Abweichung übersteigt 0.5% allerdings in beiden Klassen nicht.

7.4. Zugriffszeiten unter parallel stattfindenden Zugriffen

7.4.1. Keine Verdrängungen im Cache

In den folgenden Messreihen wird untersucht, wie groß der Einfluss von parallel stattfindenden Speicherzugriffen auf die Zugriffszeiten ist. Die resultierenden Werte sind vergleichbar mit dem Mehrkern Betrieb von mehreren Anwendungen. Die Messungen werden insbesondere mit Kapitel 7.3 verglichen. Somit kann festgestellt werden, wie robust sich die Zugriffszeiten unter Störung verhalten. Die Zugriffszeiten werden exakt wie in Kapitel 7.3 gemessen. Während jeder Zeitmessung kommt ein weiterer Trigger zur Ausführung, der in vier 32 kB (32768 Byte) großen Pools Daten kopiert. Sowohl die Messung, als auch die Unterbrechung kommen zeitgleich auf verschiedenen Kernen zur Ausführung. Es wird jeweils eine Messung durchgeführt, in der der parallel Zugriffe auf gecacheten und auf nicht-gecacheten Speicher erfolgen. In diesem Test können alle Inhalte, auf die Zugriffe erfolgen, dauerhaft im Cache liegen.
Aus den gemessenen Daten lässt sich erkennen, dass auf gecachete Speicherinhalte ein paralleler Zugriff auf weitere gecachete Speicherinhalte einen größeren Einfluss hat, als ein paralleler Zugriff auf nicht-gecachete Speicherinhalte. Dies lässt sich durch die höhere Last an dem Cache-Controller erklären.
Die Einflüsse auf den gecacheten Speicher durch einen parallelen Zugriff halten sich absolut in geringem Maße. Der Erwartungswert der Zugriffszeiten steigt höchstens um wenige Taktzyklen pro Byte und auch die gemessene Schwankung bleibt bei allen Störungen deutlich unter 1% des Erwartungswert. Dies zeigt, dass sich Zugriffszeiten auf gecachete Speicherinhalte auch im parallelen Betrieb stabil verhalten. Auch bei Störungen schneidet der immer-gecachete Speicher etwas besser ab, als der dynamisch-gecachete Speicher. Der Erwartungswert und auch die erwartete Schwankung steigen stärker an, als bei immer-gecachetem Speicher. Nicht-gecachete Speicherinhalte hingegen reagieren empfindlich auf weitere Zugriffe auf nicht-gecachete Inhalte. Dieses Verhalten ist durch die höhere Belastung des Speicherbus zu erklären. Sowohl der Erwartungswert, als auch die erwartete Schwankung steigen deutlich an.

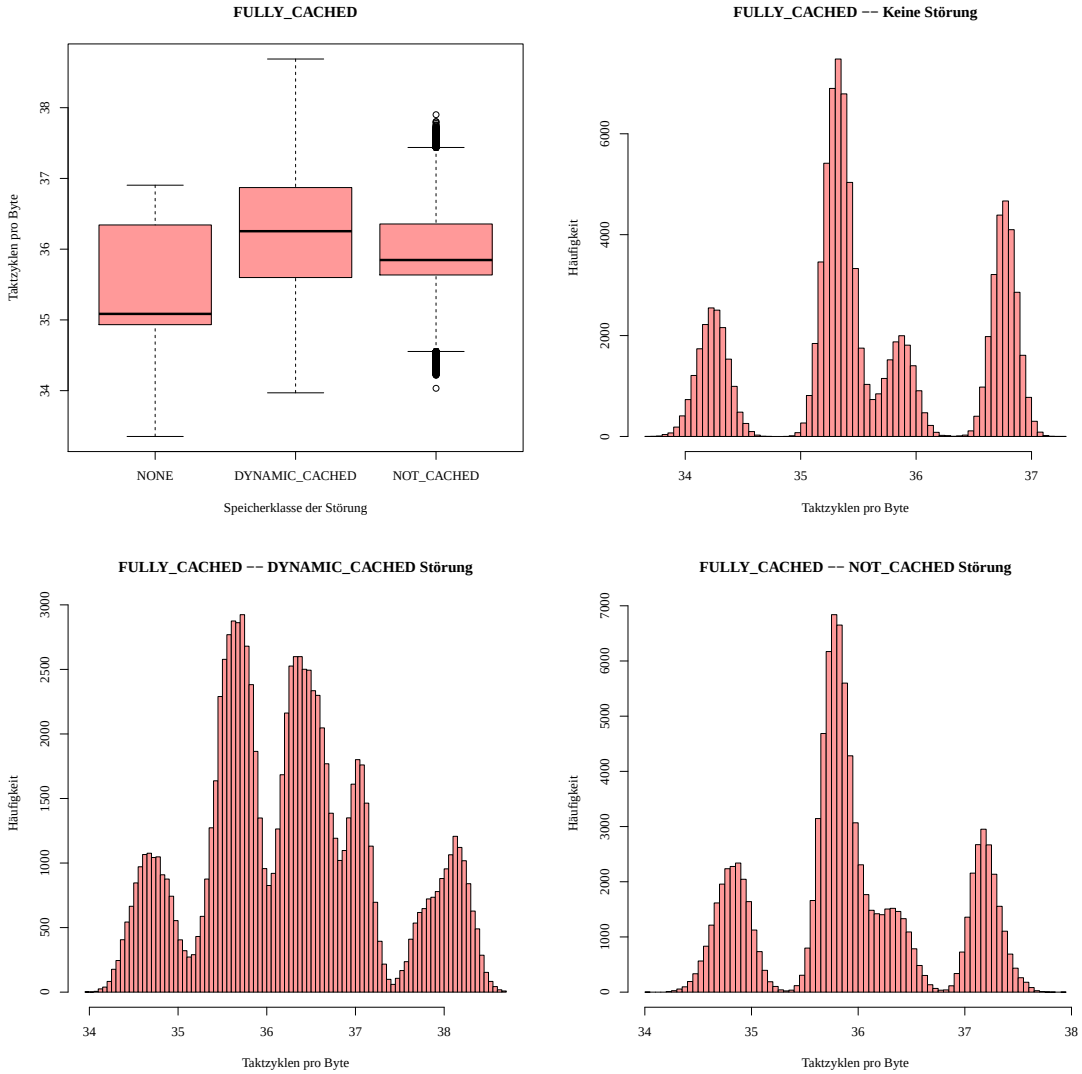

Abbildung 7.3.: FULLY_CACHED Speicher unter Störungen

Auffällig ist, dass die Verteilung der Zugriffszeiten unter den Störungen ähnlich ist, wie ohne Störungen. Sowohl in den Histogrammen, als auch in dem Boxplot ist zu erkennen, dass die parallel stattfindenden Zugriffe einen eher geringen Einfluss auf die Zugriffszeiten vornehmen.

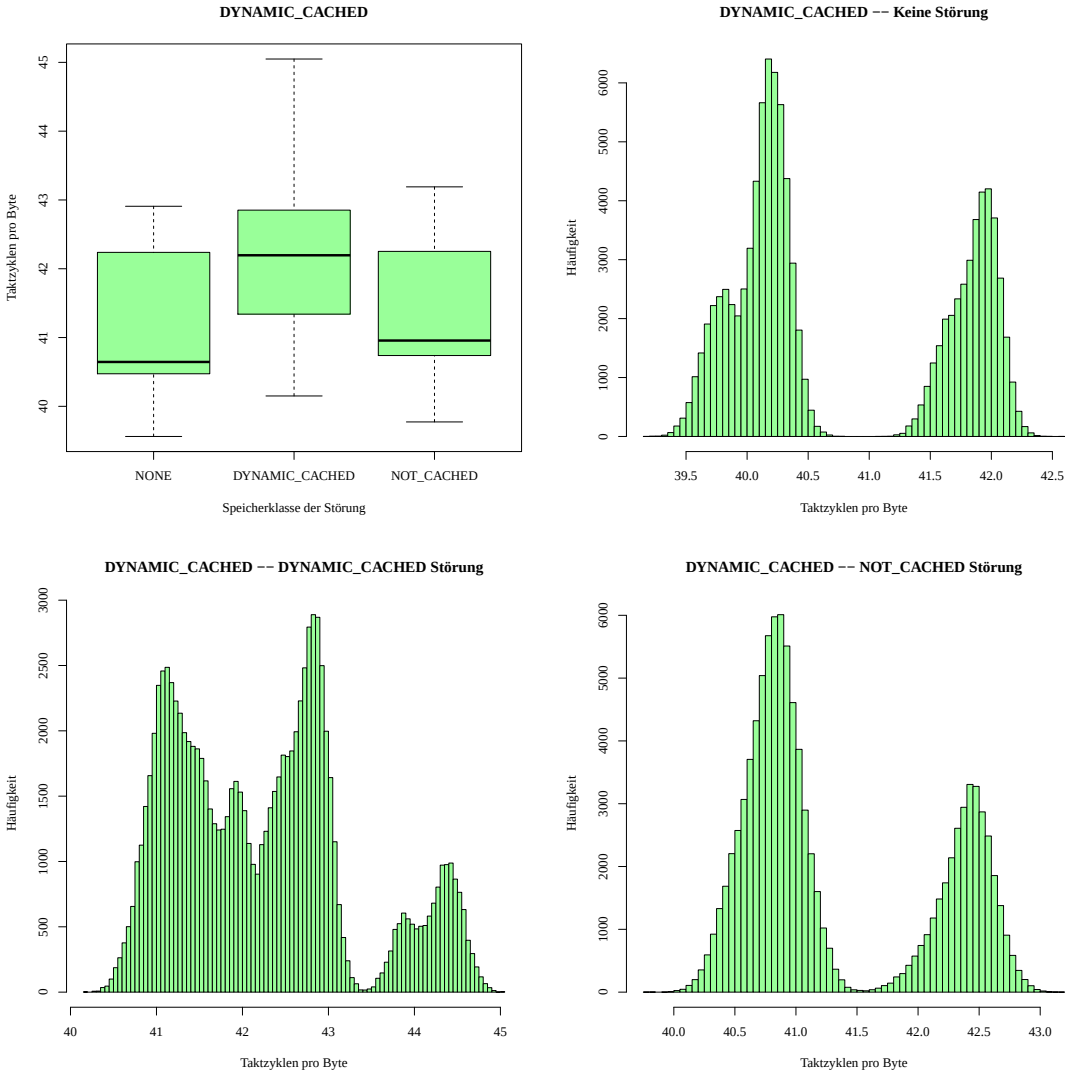

Abbildung 7.4.: DYNAMIC_CACHED Speicher unter Störungen

Der dynamisch-gecachete Speicher verhält sich unter parallel stattfindenden Speicherzugriffen sehr ähnlich, wie immer-gecacheter Speicher (Abbildung 7.3). Die Verteilung der Zugriffszeiten verändert sich allerdings unter parallel stattfindenden Zugriffen auf gecacheten Speicher stärker.

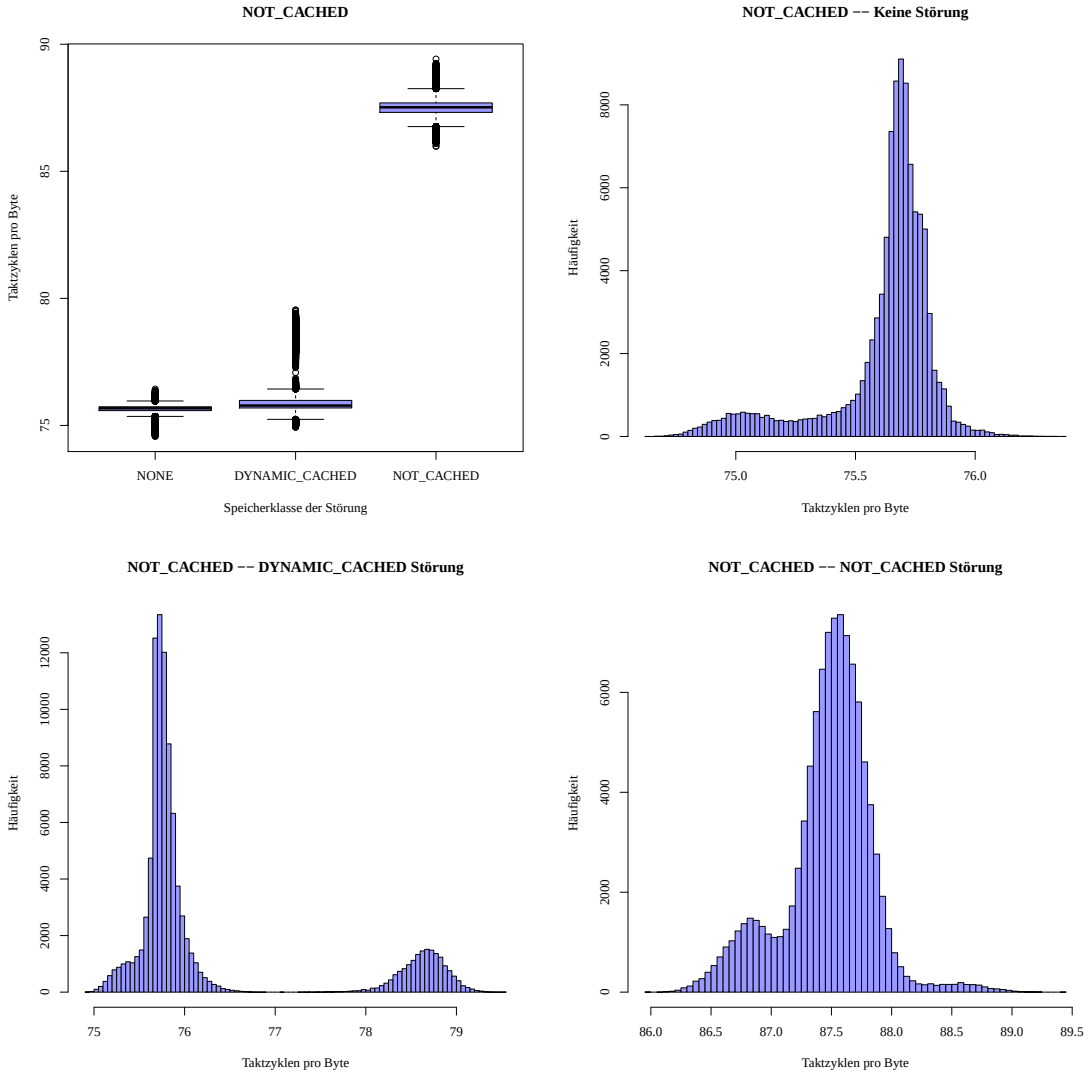

Abbildung 7.5.: NOT_CACHED Speicher unter Störungen

In dem Boxplot lässt sich besonders gut erkennen, dass nicht-gecacheter Speicher besonders empfindlich auf parallele Zugriffe auf nicht-gecacheten Speicher reagiert. Die Zugriffszeiten werden deutlich größer und schwanken auch stärker. Dies unterstützt die Hypothese, dass ein Echtzeitverhalten im Mehrkern Betrieb erreicht werden kann, wenn parallele Zugriffe auf den Hauptspeicher durch den gemeinsamen Bus vermieden werden können.

7.4.2. Verdrängungen im Cache

In den nachfolgenden Messungen werden die selben Zugriffszeiten wie in Kapitel 7.4.1 gemessen. Die parallel stattfindenden Speicherzugriffe werden allerdings verändert, so dass nicht auf vier, sondern auf elf bzw. zehn 32 kB große Pools parallel zugegriffen wird. Dies führt dazu, dass während jeder Messung die Speicherinhalte aus dem Cache verdrängt werden, da die Cache-Ways gemäß der LRU-Strategie verteilt werden. Dieser Test simuliert eine extreme Nutzung des Betriebssystems. Wie bereits in Kapitel 7.1 ausgeführt, sind für die Speicherbereiche des Tests Triggerabhängigkeiten eingetragen, sodass diese nicht verdrängt werden können, während die Zugriffszeit gemessen wird. Es werden nur Cache-Ways verdrängt, in denen Inhalte stehen, die von der störenden Komponente stammen oder nicht mehr benötigt werden.
Aus den Messungen geht hervor, dass das Betriebssystem auch unter übermäßiger Belastung noch geringe Zugriffszeiten gewährleisten kann. Dieses Ergebnis ergibt sich allerdings nur, da die zu testenden Inhalte priorisiert wurden, da sie durch die Speicherabhängigkeiten vor einer Verdrängung geschützt wurden. Die zusätzliche Belastung durch das ständige Verdrängen und Nachladen auf einem CPU-Kern wirkt sich auf die zu testende Komponente nur sehr gering aus. Der Mittelwert der Zugriffszeiten steigt höchstens um wenige Taktzyklen und auch die Schwankung steigt um weniger als 1% des Mittelwerts.

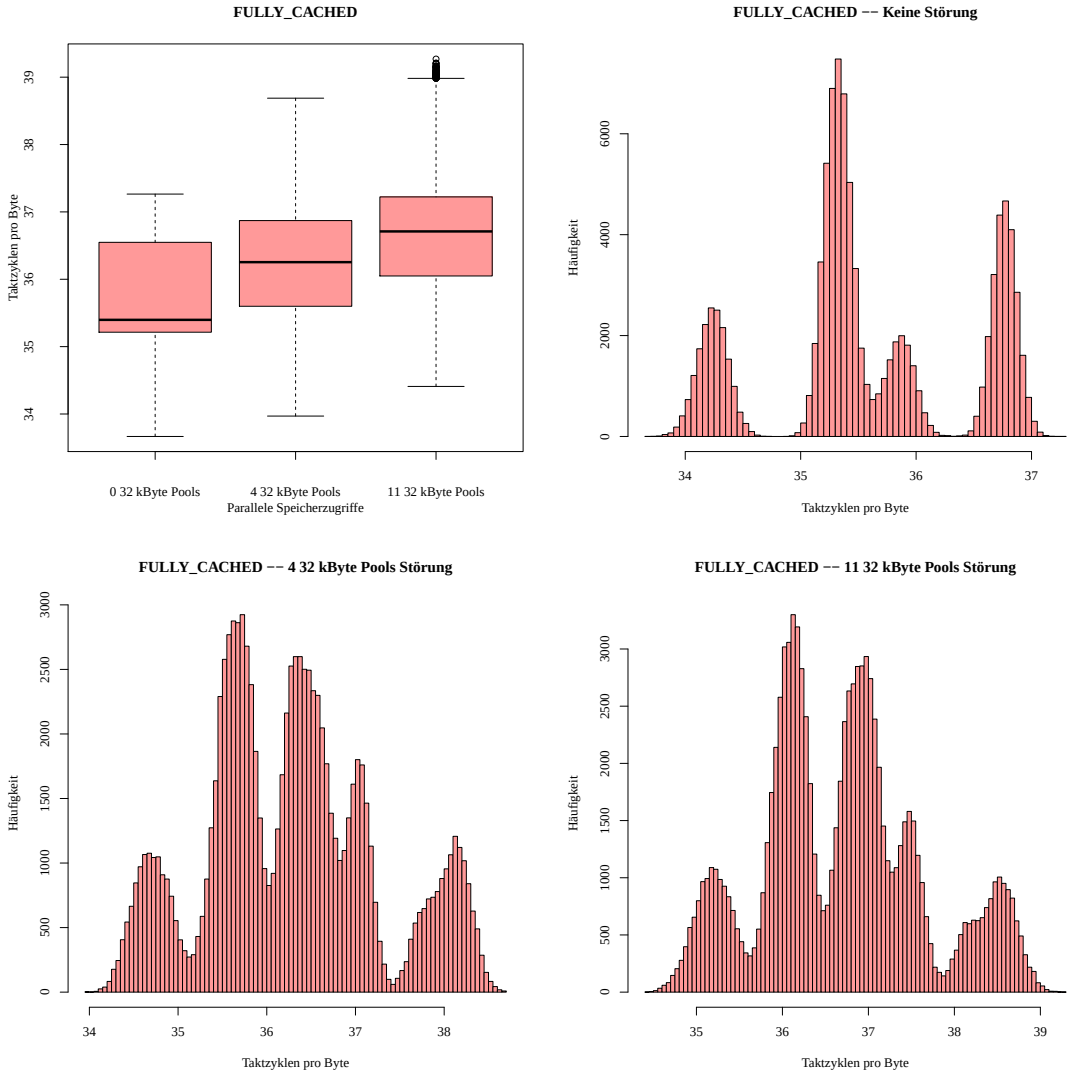

Abbildung 7.6.: FULLY_CACHED Speicher im parallelen Betrieb

In dem Boxplot lässt sich gut erkennen, dass sich die Zugriffszeiten auch unter steigender paralleler Last stabil verhalten. Das Maximum steigt dabei auch nur um ca. 2 Taktzyklen an. Auch die Histogramme zeigen, dass sich die Verteilung der Zugriffszeiten nicht stark verändert. Diese Beobachtungen sprechen dafür, dass mit dieser Implementierung der Speicherverwaltung die Effekte des parallelen Betriebs größtenteils abgeschirmt werden können, so wie CyPhOS es bereits für die Ausführung von Anwendungen tut.

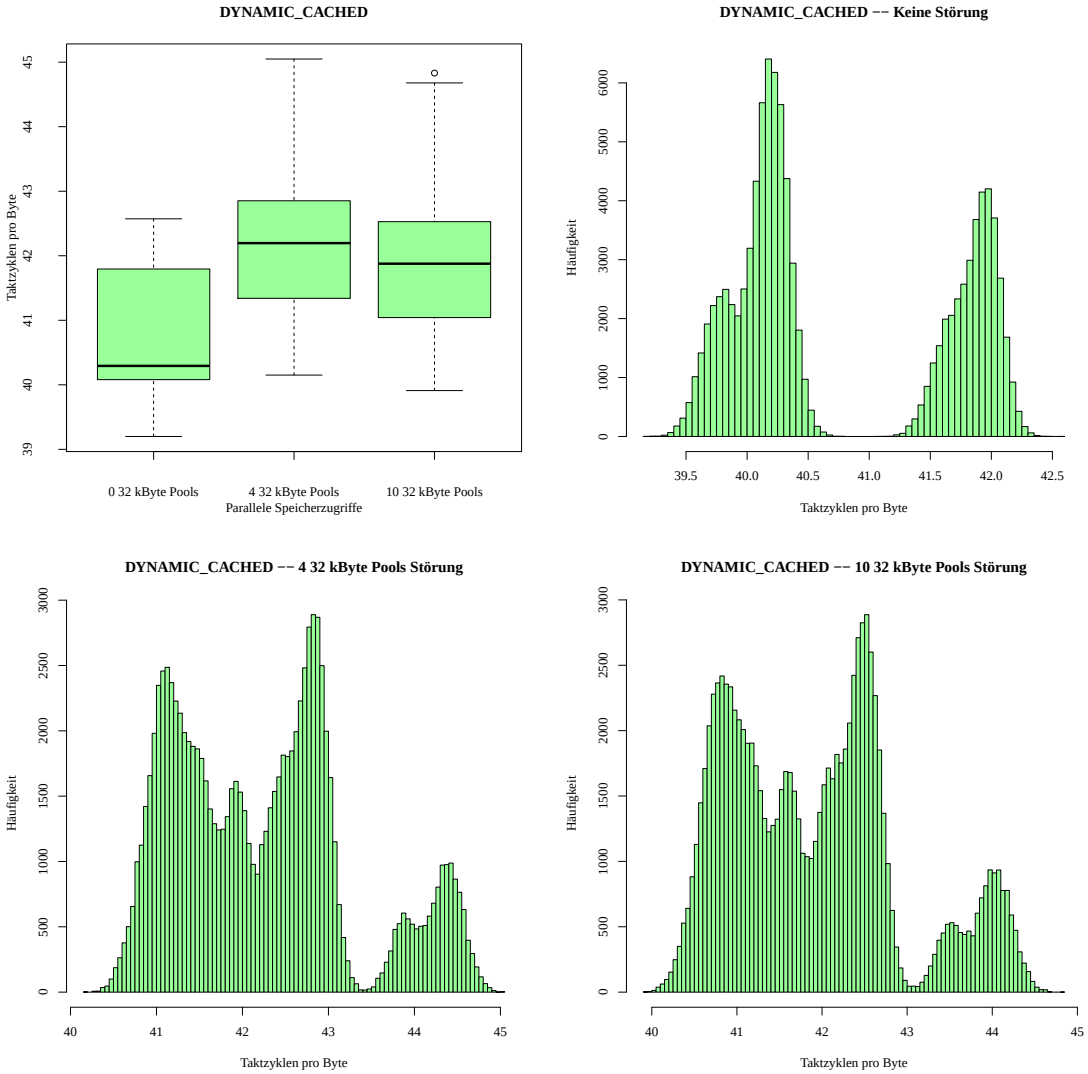

Abbildung 7.7.: DYNAMIC_CACHED Speicher im parallelen Betrieb

Die Zugriffszeiten auf Speicher aus der Klasse “dynamisch-gecachet” verhalten sich sehr ähnlich den Zugriffszeiten für die Klasse “immer im Cache” (Abbildung 7.6). Die Zugriffszeiten unterscheiden sich auch hier nur um wenige Taktzyklen. Auffällig erscheint in dem Boxplot, dass die Zugriffszeiten unter stattfindender Verdrängung scheinbar geringer werden. Bei genauerer Analyse der Histogramme stellt sich die Verteilung der Zugriffszeiten allerdings nahezu identisch heraus, wie die unter geringer paralleler Last. Das Bild im Boxplot kann durch die etwas andere Verteilung entstehen.

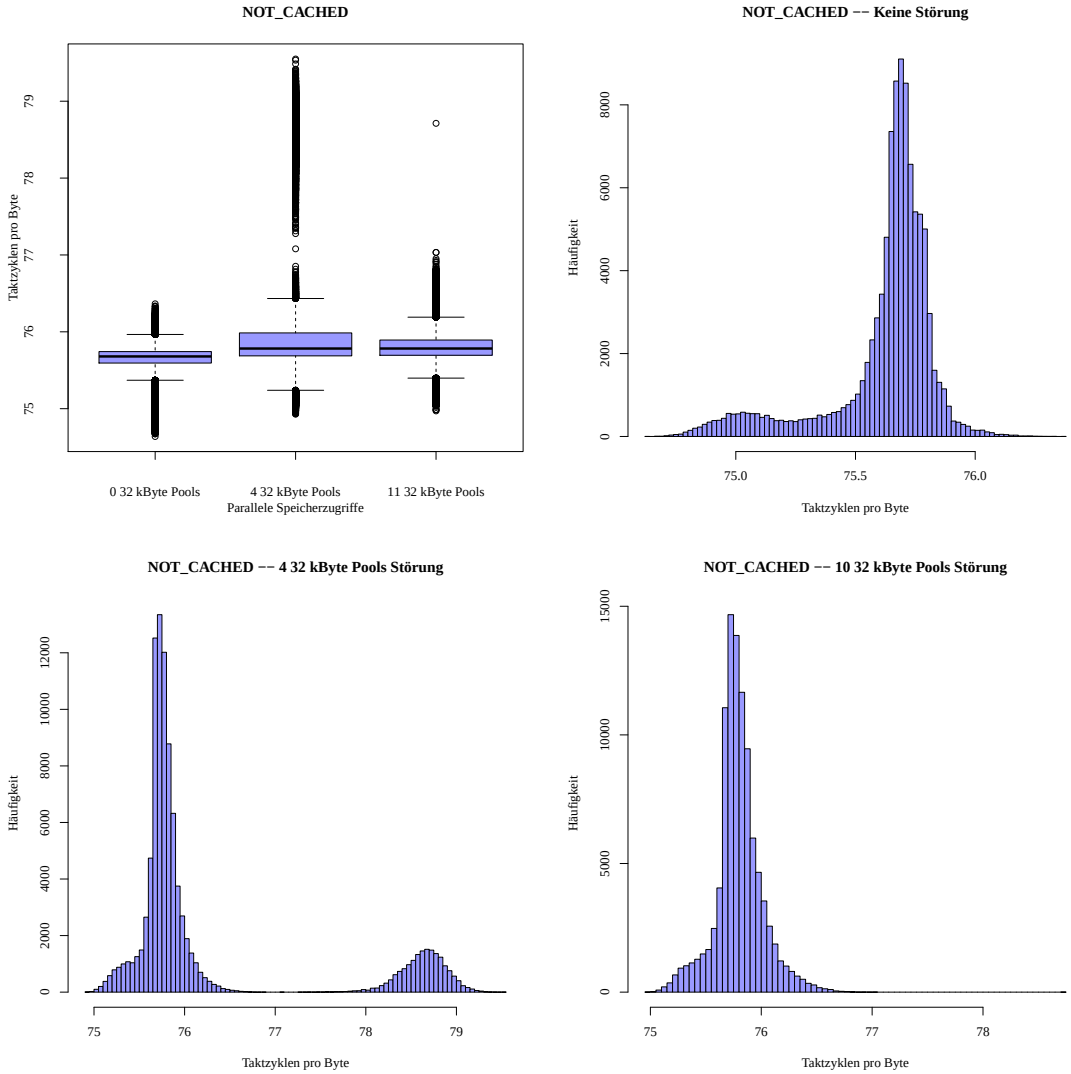

Abbildung 7.8.: NOT_CACHED Speicher im parallelen Betrieb

Der Einfluss von parallel stattfindenden Speicherzugriffen auf gecacheten Speicher zeigt sich erwartungsgemäß gering auf die Zugriffszeiten auf nicht-gecacheten Speicher. Finden Verdrängungen statt, so erfolgen durch das Vorladen zwar Zugriffe in den Hauptspeicher, diese sind allerdings sequentiell und verhältnismäßig selten.

7.5. Vergleich mit reinem Hardware-Cache-Management

In den nachfolgenden Tests werden einige der Messungen aus Kapitel 7.3 und Kapitel 7.4 wiederholt. Diesmal wird die Software-Cache-Kontrolle von CyPhOS deaktiviert. Somit übernimmt der Hardware-Cache-Controller des Exynos Prozessors die Verwaltung der L2-Cache-Ways. Anhand dieser Ergebnisse wird beurteilt, inwiefern die Software-Cache-Kontrolle besser funktioniert als ein nicht aktiv kontrollierter Cache.

7.5.1. Kein paralleler Betrieb (vgl. Kapitel 7.3)

Für die nachfolgenden Messreihen werden die Zugriffszeiten für die verschiedenen Speicherklassen ohne eine parallele Störung gemessen. Die Größe der Speicherpools entspricht exakt den Testbedingungen aus Kapitel 7.3.
In allen Testreihen fällt auf, dass die Hardware-Verwaltung einen deutlich geringeren Mittelwert aufweist. Wird die Software-Verwaltung verwendet, so wird auch eine virtuelle Speicheradressierung verwendet. Dies erhöht die Zugriffszeiten. Die prozentuale Schwankung ist bei dem Test mit dynamische gecachetem Speicher unter Hardware-Verwaltung etwas geringer, übersteigt aber in allen Tests nie 0.7% (für den größten Häufungswert). Der massive Unterschied bei den nicht-gecacheten Speicherinhalten rührt daher, dass bei einer reinen Hardware-Cache-Verwaltung Inhalte nicht explizit vom Caching ausgeschlossen werden. Die Speicherinhalte werden also trotzdem in den Cache gelegt.

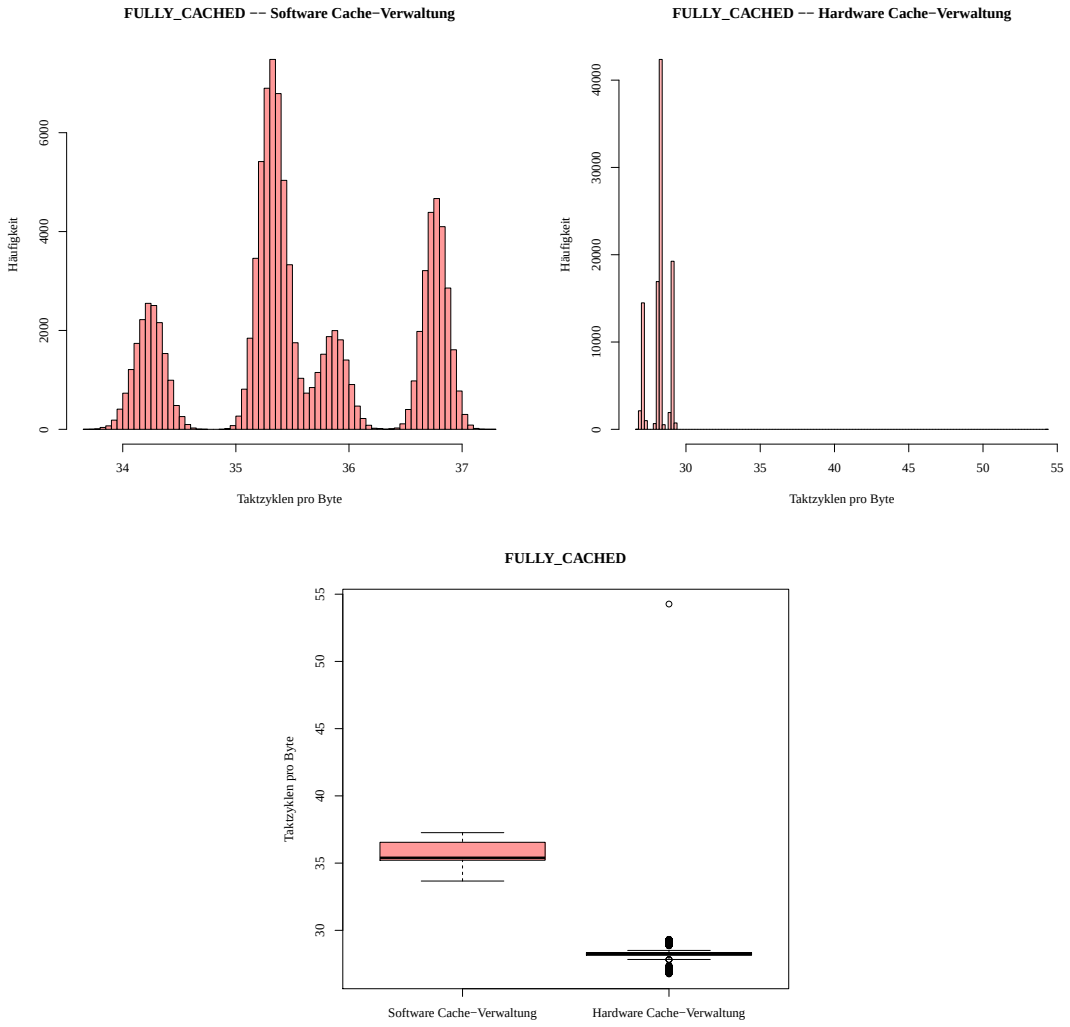

Abbildung 7.9.: FULLY_CACHED Speicher Hardware / Software verwaltet

Aus dem Histogramm lässt sich erkennen, dass der Maximalwert deutlich weiter vom Mittelwert entfernt liegt, wenn die Hardware-Cache-Verwaltung zum Einsatz kommt ($\mu \approx 35$ und $max \approx 37$ bei Software-Verwaltung, $\mu \approx 28$ und $max \approx 54$ bei Hardware-Verwaltung). Dies liegt daran, dass bei reiner Hardware-Cache-Verwaltung Inhalte verdrängt werden können, während die Komponente noch läuft. Dies führt bei dem nächsten Zugriff dann zu einem Ausreißer, da der Inhalt erst wieder in den Cache geladen werden muss. Dieses Verhalten wird bei der Software-Cache-Verwaltung aktiv verhindert.

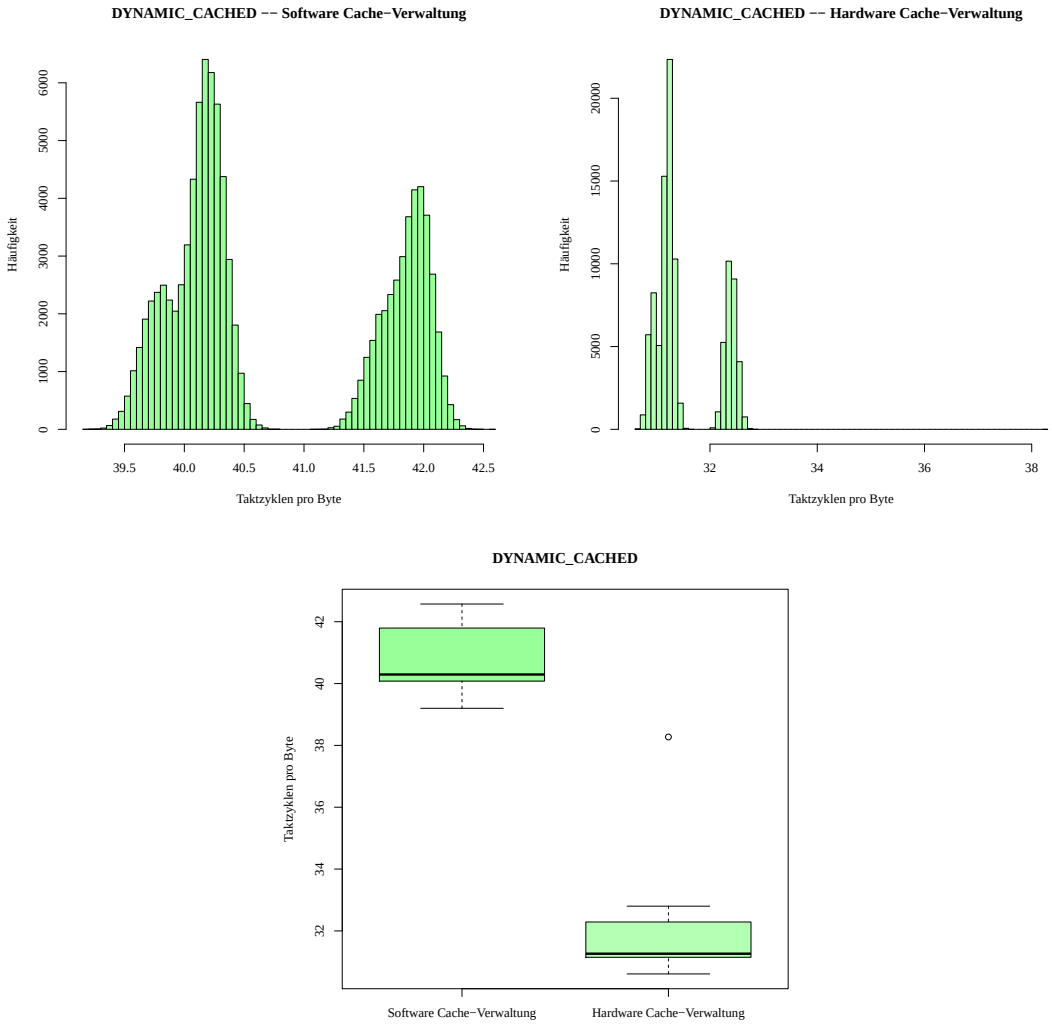

Abbildung 7.10.: DYNAMIC_CACHED Speicher Hardware / Software verwaltet

Ähnlich wie bei immer-gecachetem Speicher (Abbildung 7.9) ist der Maximalwert bei der Hardware-Cache-Verwaltung im Vergleich zum Mittelwert deutlich größer als bei der Software-Cache-Verwaltung ($\mu \approx 40$ und $max \approx 42$ bei Software-Cache-Verwaltung, $\mu \approx 31$ und $max \approx 38$ bei Hardware-Cache-Verwaltung). Der hohe Maximalwert kommt auch hier durch Ausreißer zustande, bei denen Inhalte während der Ausführung eines Trigger aus dem Cache verdrängt wurden und dann bei einem Zugriff erst wieder vorgeladen werden müssen. Die Software-Verwaltung schließt ein Verdrängen der Inhalte aus, während der Trigger in Betrieb ist.

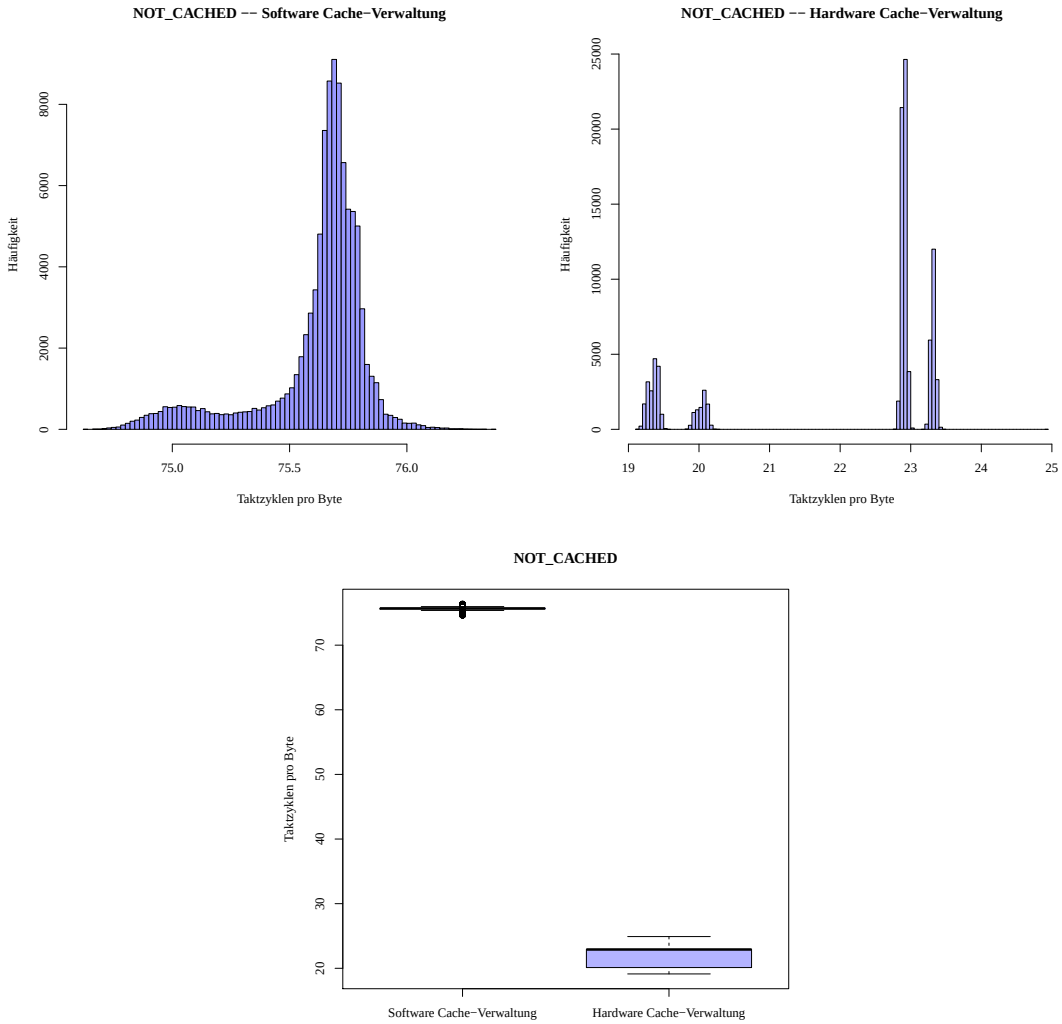

Abbildung 7.11.: NOT_CACHED Speicher Hardware / Software verwaltet

Der große Unterschied zwischen Software-Verwaltung und Hardware-Verwaltung liegt darin begründet, dass Speicherinhalte bei der Software-Verwaltung explizit vom Laden in den Cache ausgeschlossen werden. Bei der Hardware-Verwaltung hingegen werden diese Speicherinhalte wie alle anderen behandelt.

7.5.2. Paralleler Betrieb (vgl. Kapitel 7.4)

In dieser Testreihe wird abermals untersucht, wie sich die Zugriffszeiten im parallelen Betrieb verändern. Dabei wird die Software-Cache-Verwaltung mit der Hardware-Cache-Verwaltung verglichen. Zum Ermitteln der Zugriffszeiten kommt die selber Testumgebung wie in Kapitel 7.4 zum Einsatz. Eine Störung durch Zugriffe in nicht-gecacheten Speicher kann an dieser Stelle nicht verglichen werden, da dieser Test mit der Hardware-Cache-Verwaltung nicht durchgeführt werden kann.

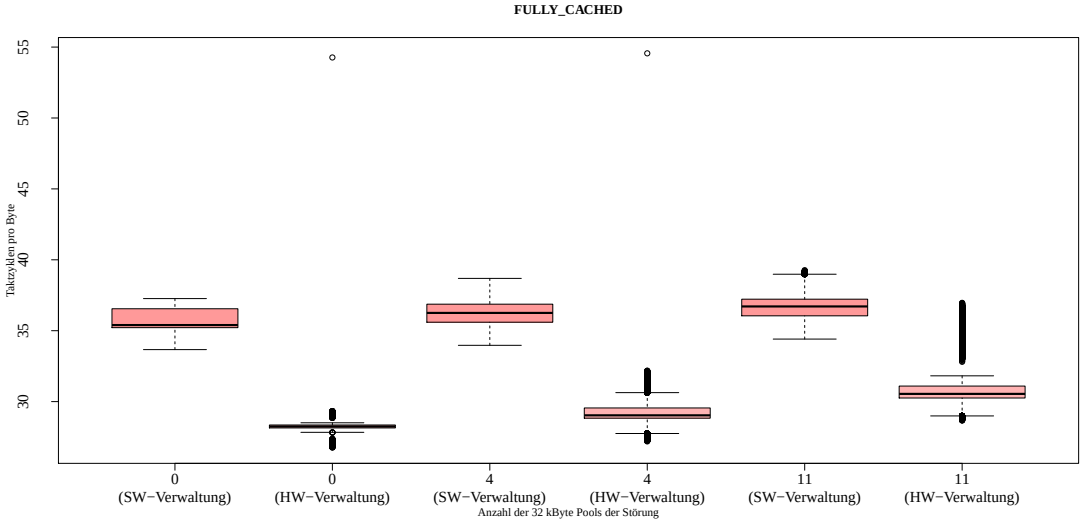

Abbildung 7.12.: FULLY_CACHED Speicher unter Störungen (Hardware und Software verwaltet)

Auffällig ist, dass bei allen Testreihen mir Hardware-Cache-Verwaltung die Zugriffszeiten im Mittel niedriger liegen als bei Verwendung der Software-Cache-Verwaltung, die Streuung allerdings größer ist. In allen Tests treten bei der Hardware-Cache-Verwaltung mehr Ausreißer auf, die auch weiter vom Mittelwert weg streuen. Im extrem parallelen Betrieb (11 parallel zugegriffene Pools) sind in der Messreihe der Hardware-Cache-Verwaltung extrem viele Ausreißer zu beobachten, in der Messreihe der Software-Cache-Verwaltung sind hingegen nur sehr wenige Ausreißer zu finden. Auch die Differenz der Mittelwerte bzw. Mediane zwischen Hardware- und Software-Verwaltung ist im extrem parallelen Betrieb erheblich geringer. Die Zugriffszeiten bleiben bei der Software-Cache-Verwaltung demnach stabiler, wenn die parallele Auslastung des Systems steigt, als es die Zugriffszeiten bei der Hardware-Cache-Verwaltung tun.

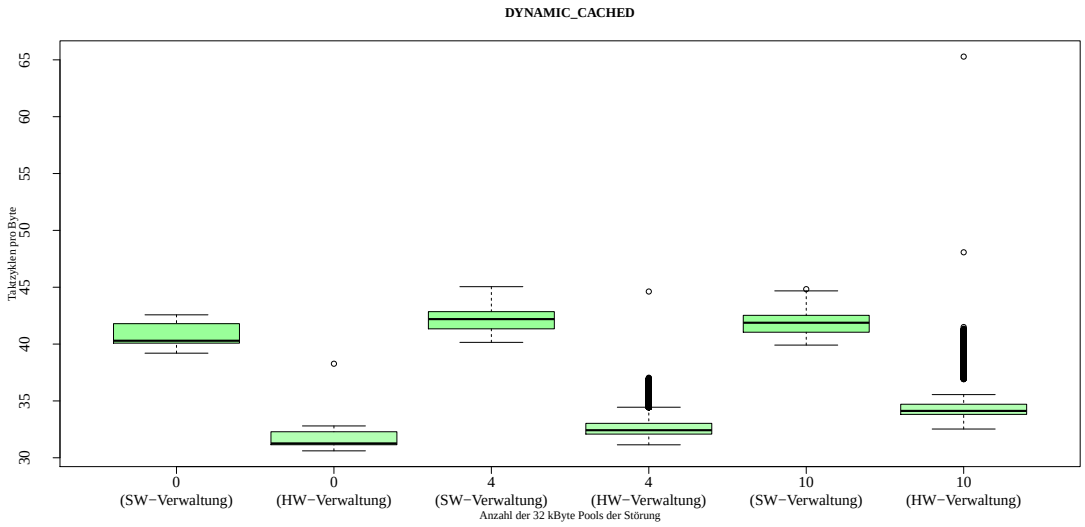

Abbildung 7.13.: DYNAMIC_CACHED Speicher unter Störungen (Hardware und Software verwaltet)

Die Messungen unterscheiden sich prinzipiell nicht von den Messungen für immergecacheten Speicher (Abbildung 7.12). Die Hardware-Cache-Verwaltung behandelt ohnehin alle Speicherklassen gleich und auch bei der Software-Cache-Verwaltung gibt es nur geringfügige Unterschiede in der Verwaltung, die die Zugriffszeiten beeinflussen könnten.

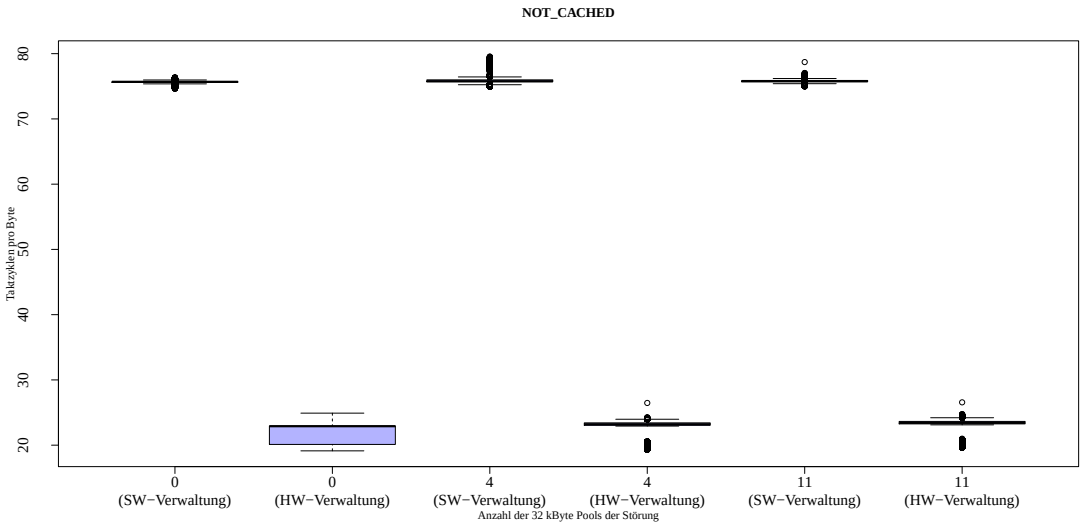

Abbildung 7.14.: NOT_CACHED Speicher unter Störungen (Hardware und Software verwaltet)

In diesen Messungen besteht zwischen Hardware und Software-Verwaltung ein großer Unterschied, da die Hardware-Cache-Verwaltung die Inhalte gleichermaßen in den Cache lädt, wie alle anderen Inhalte auch. Die Auswirkungen im parallelen Betrieb lassen sich somit nicht vergleichen.

7.6. Verhalten der Vorladezeiten der Komponenten

In der letzten Testreihe wird untersucht, welchen Einfluss die Nutzung der Speicherverwaltung auf das Vorladen der Komponenten in den Cache hat. Die Komponente `MemoryEvalOSCPreload` [4] ist eine speziell angelegte Testkomponente, für die die Vorladezeit gemessen wird, wenn der Trigger `do_the_work` zur Ausführung kommt. Dieser Test wird 100000 mal wiederholt.
Um den Einfluss des parallelen Betriebs zu untersuchen, kommt abermals die Komponente `MemoryEvalDisturber` [5] zum Einsatz. Während der Messung der Vorladezeit wird auf einem anderen CPU-Kern auf Speicherpools zugegriffen, die aus allen 3 Speicherklassen stammen können. Für den Test wird zur parallelen Störung auf 6 Pools der Größe 32 kB zugegriffen.
Damit auch bei jedem Test die Komponente tatsächlich komplett in den Cache geladen wird und nicht schon Bereits Inhalte der Komponente im Cache liegen, wird

[4] `testcomponents/MemoryEvalOSCPreload.{h,cc}`
[5] `testcomponents/MemoryEvalDisturber.{h,cc}`

nach jeder Messung der Trigger `cleanup` der Komponente `MemoryEvalOSCClean` [6] ausgeführt. Dieser alloziert Speicher über die Speicherverwaltung aus der Klasse dynamisch-gecachet. Diese reservierten Speicher werden bei der Reservierung sofort in den Cache geladen, dadurch wird die zu testende Komponente aus dem Cache verdrängt.

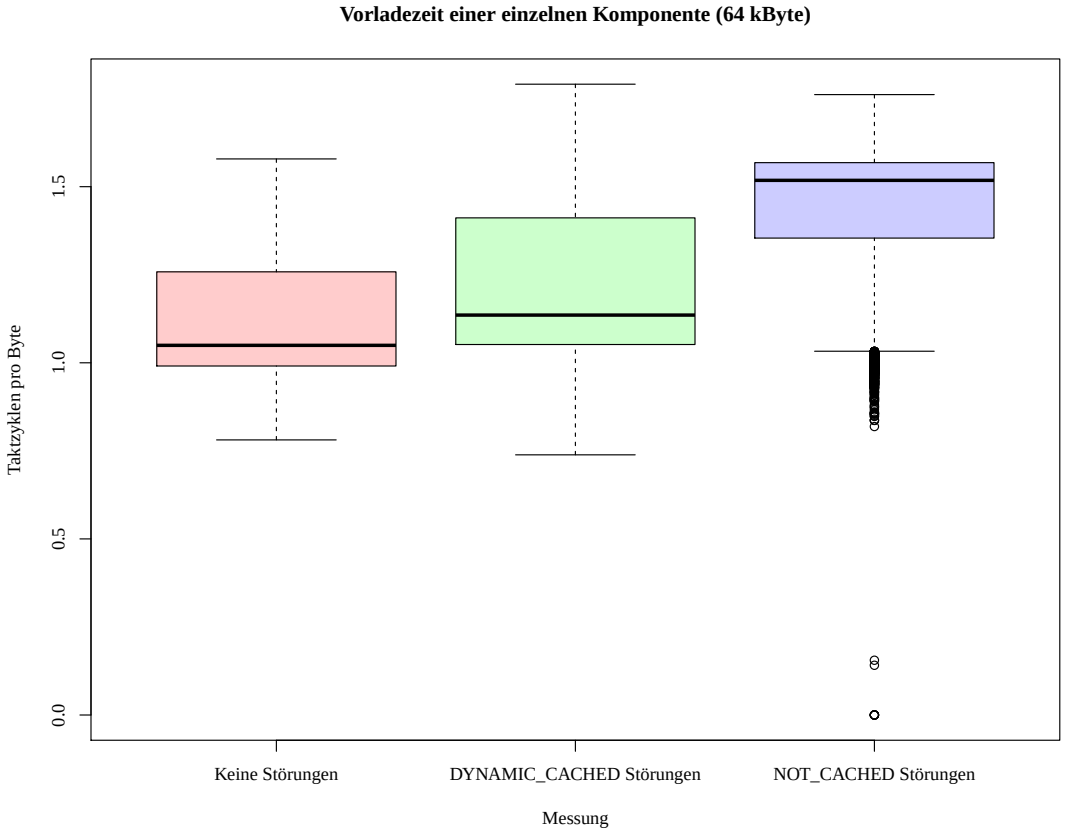

Abbildung 7.15.: Boxplots der Vorladezeiten unter Störungen

Gut zu erkennen ist, dass die Vorladezeiten in allen Tests um ca. 1 Taktzyklus pro Byte schwanken. Die stärkste Auswirkung hat ein paralleler Zugriff auf nicht-gecacheten Speicher, da sich dann dieser Zugriff und das Vorladen den Speicherbus teilen müssen. Auch Zugriffe auf gecacheten Speicher haben eine Auswirkung, denn der Cache-Controller wird dabei einer höheren Last ausgesetzt. Insgesamt halten sich die Auswirkungen von parallelem Betrieb in engen Grenzen.

[6] `testcomponents/MemoryEvalOSCClean.{h,cc}`

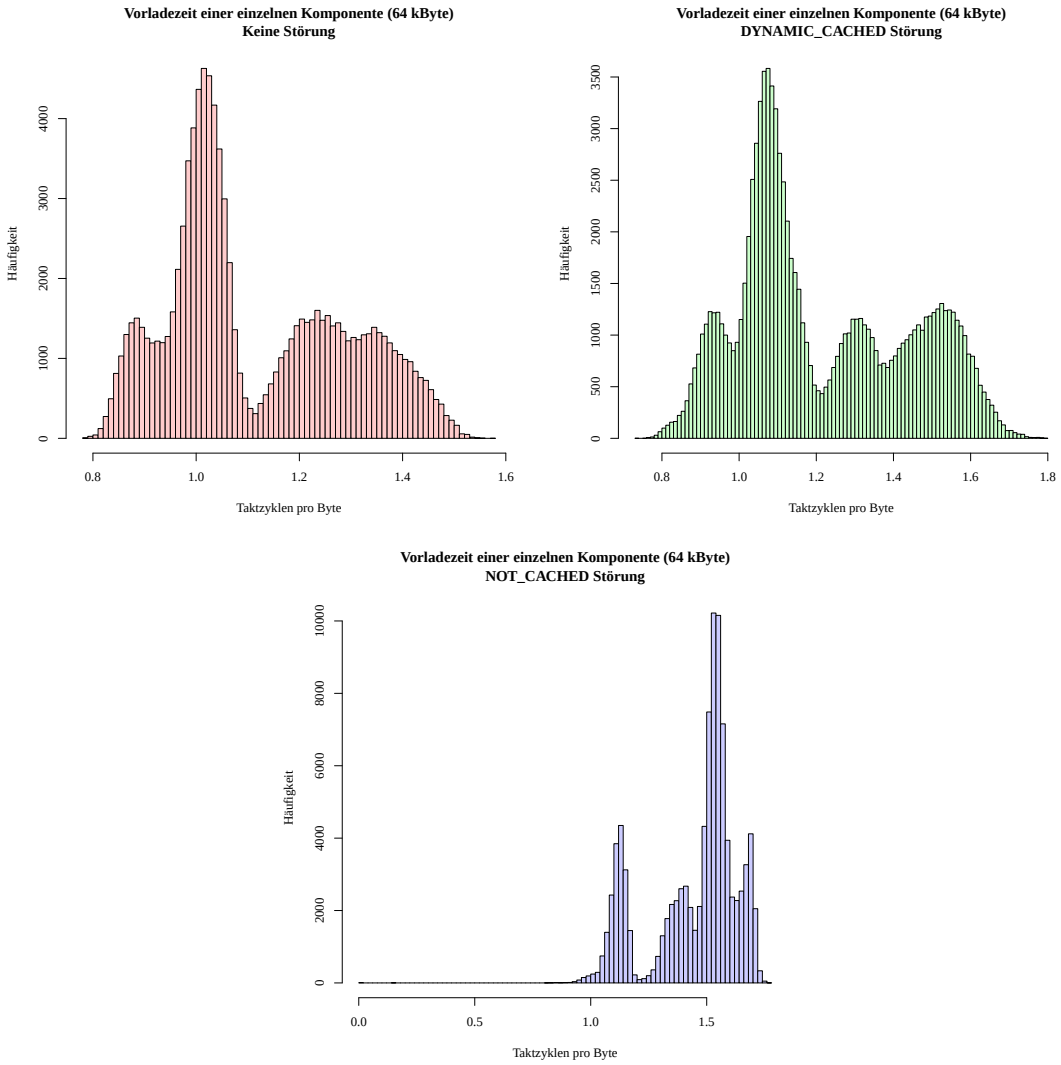

Abbildung 7.16.: Histogramme der Vorladezeiten

Auch in den Histogrammen lässt sich derselbe Effekt beobachten wie in dem Boxplot (Abbildung 7.15). Parallele Zugriffe auf gecacheten Speicher ändern die Verteilung der Zugriffszeiten nur sehr geringfügig. Parallele Zugriffe auf nicht-gecacheten Speicher hingegen verändern auch die Verteilung der Zugriffszeiten, sodass sich ein sehr großer Häufungswert bei einer hohen Zugriffszeit ausbildet.

7.7. Fazit der Evaluation

Die verschiedenen Messungen der Evaluation zeigen, dass es bezüglich der Zugriffszeiten Unterschiede zwischen den verschiedenen Speicherklassen gibt. Den Anwendungen werden demnach tatsächlich Speicherklassen mit verschiedenen Verwendungszwecken zur Verfügung gestellt.
Die Abweichung vom erwarteten Mittelwert der Zugriffszeiten ist für alle Speicherklassen gering (unter 1%). Bei den gecacheten Speicherklassen können in den Messungen nur sehr wenige bis keine Ausreißer festgestellt werden. Den Anwendungen kann aufgrund dieser Messungen ein verlässliches Zeitverhalten beim Zugriff auf Daten ermöglicht werden.
Wird das Betriebssystem mit mehreren Programmen gleichzeitig betrieben, die auch alle Speicherzugriffe auslösen, so ist eine Veränderung der Zugriffszeiten unvermeidlich. Die Zugriffszeiten für gecacheten Speicher weisen dennoch auch im parallelen Betrieb nahezu keine Ausreißer auf und auch der Mittelwert und die erwartete Schwankung steigt nicht übermäßig an.
Bei einem parallelen Betrieb, bei dem häufig auf nicht-gecachete Inhalte zugegriffen wird, reißen auch die Zugriffszeiten der parallelen Zugriffe auf gecacheten Speicher häufiger aus. Das Maximum der Zugriffszeiten steigt dabei aber nicht bedeutend an.
Wird das System über seine Leistungsfähigkeit hinaus belastet (z. B. indem mehr gecacheter Speicher permanent benötigt wird, als Cache-Ways verfügbar sind), so kann durch die Struktur der Speicherverwaltung eine Priorisierung für kritische Inhalte erzielt werden. Auch im Betrieb über die Leistungsfähigkeit hinaus verändern sich die Zugriffszeiten für so priorisierten Speicher nur sehr gering. Die Schwankung und auch der Mittelwert können gering gehalten werden.
Im Vergleich mit der Hardware-Cache-Verwaltung des Cortex-A9 Prozessors schneidet die Software-Cache-Verwaltung auf den ersten Blick schlechter ab, da die Mittelwerte mit der Hardware-Cache-Verwaltung deutlich geringer liegen. Es zeigt sich allerdings auch, dass beim Betrieb mit der Hardware-Cache-Verwaltung häufiger Ausreißer und höhere Maximalwerte auftreten, was mit der Software-Cache-Verwaltung vermieden werden kann.
Die Vorladezeiten der Komponenten werden vom parallelen Betrieb der Speicherverwaltung kaum beeinflusst, wobei auch dabei eine starke Nutzung von nicht-gecachetem Speicher den stärksten Einfluss aufweist.
Verglichen mit den selben Messungen auf einem anderen Prozessor (Anhang B) wird deutlich, dass sich die Zugriffszeiten auf anderen Hardware deutlich unterschiedlich verhalten können.

8. Zusammenfassung und Ausblick

8.1. Konzept der Speicherverwaltung

Ein maßgebliches Ziel dieser Arbeit ist es, ein Konzept für eine Speicherverwaltung zur Verfügung zu stellen, die die Anforderungen eines Cache-Gewahren Betriebssystems wie CyPhOS erfüllt. Diese Speicherverwaltung ermöglicht es, Anwendungen mit der Auswahl zwischen drei Speicherklassen den Grad der Cache-Gewahrheit des reservierten Speichers zu bestimmen. Das Konzept der Speicherabhängigkeiten für Trigger erfordert zwar ein Verständnis für die Arbeitsweise von CyPhOS, bietet aber auch eine hohe Flexibilität in der Nutzung der reservierten Speicherbereiche. Die Idee der Cache-Gewahrheit einer Anwendung während der Ausführung wird durch dieses Konzept vollständig unterstützt und sogar noch erweitert.
Auch für die Verwendung des `new` Operators ist ein Verständnis der Speicherverwaltung erforderlich, dies ermöglicht aber alle Vorteile der Speicherverwaltung auch für dynamisch erstellte Instanzen zu nutzen.

8.2. Leistungsfähigkeit der Speicherverwaltung

In Kapitel 7 wird ausführlich die Leistungsfähigkeit der Speicherverwaltung untersucht. Dabei wird deutlich, dass auf jeden Fall einige Echtzeitanforderungen erfüllt werden können. Auch im parallelen Betrieb bleiben Zugriffszeiten für reservierte Speicherbereiche noch bis zu einem gewissen Grad vorhersagbar. Durch die in Anhang B gegebenen Informationen wird allerdings auch deutlich, dass sich die zeitlichen Eigenschaften des Systems auf unterschiedlicher Hardware stark unterschiedlich verhalten können. Bei der Portierung von zeitkritischen Anwendungen auf eine andere Hardwareplattform kann das Zeitverhalten nicht verlässlich beibehalten werden.
Die Evaluation untersucht hier nur die direkten Auswirkungen im Bezug auf die Cache-Verwaltung des Betriebssystems. Im Multicore-Betrieb treten noch weitere Effekte auf, deren Einfluss auf die Zugriffszeiten untersucht werden kann. Das Verhalten der L1-Caches kann die Zugriffszeiten beispielsweise beeinflussen.

8.3. Stand der Forschung

Forschungsarbeiten im Kontext von Cache-Gewahrheit beschäftigen sich in der Regel nicht mit Konzepten zu einer kompletten Cache-Gewahren Speicherverwaltung in einem Echtzeitbetriebssystem, da es vergleichbare Betriebssysteme wie CyPhOS nicht gibt. Das in dieser Arbeit entwickelte Konzept kann demnach nicht mit bereits vorgestellten Konzepten verglichen werden. Dennoch gibt es einige Forschungsergebnisse, die in der Speicherverwaltung in CyPhOS sinnvolle Anwendung finden könnten.
[6] beschäftigt sich mit dem effektiven Vorladen von Datenstrukturen. Dazu kommen verschiedene Mechanismen zum Einsatz, die unter anderem anhand von Abhängigkeiten Speicherbereiche spekulativ vorladen, um im Endeffekt geringe Zugriffszeiten zu ermöglichen. Eine Analyse der reservierten Speicher und spekulatives Vorladen aufgrund der Ergebnisse kann auch für die Speicherverwaltung in CyPhOS implementiert werden. Das Betriebssystem stellt bereits einen Vorlademechanismus zur Verfügung, der im Betriebssystem arbeitet. Im Schnitt könnte so für dynamisch-gecachete Speicher die Zugriffszeit noch verringert werden. Für Speicherinhalte, die ohnehin immer im Cache liegen, ergibt sich durch eine Zugriffsvorhersage kein Vorteil. CyPhOS selber könnte für das Vorladen von Komponenten einen Vorteil aus der Analyse ziehen.
[10], [8] und [9] stellen Konzepte vor, mit denen sich das Cache-Verhalten von Datenstrukturen verbessern lässt. Dies lässt sich auch zum Teil in CyPhOS anwenden. Das Cache-Verhalten ist schon sehr gut, da die Datenstrukturen ohnehin komplett im Cache liegen. CyPhOS verwaltet allerdings nur den L2-Cache aktiv, ein optimiertes Verhalten im L1-Cache könnte die Zugriffszeiten noch weiter verbessern. Zudem kann die Speicherverwaltung Ergebnisse dieser Forschung verwenden um Speichernfragen intelligenter zu platzieren, sodass auch ohne zusätzliche Optimierungen in der Anwendungskomponente die Zugriffszeiten optimiert werden können. Das Betriebssystem kann so auch mithilfe der Speicherverwaltung optimierte Datenstrukturen zur Verfügung stellen, die die Anwendungskomponenten nutzen können.

8.4. Ausblick

8.4.1. Cache-Gewahre Speicherallokation

Die hier vorgestellte Version der Speicherverwaltung stellt für den Allokationsprozess keinerlei Zeitgarantien zur Verfügung. Zudem erfolgen bei der Speicherallokation Zugriffe in den Hauptspeicher, wodurch hohe Laufzeiten entstehen. Ein möglicher Ansatz wäre das Cache-Management von CyPhOS so anzupassen, dass ein beliebiger Speicherbereich in den Cache vorgeladen werden kann, ohne dass

dieser mit bestimmten Datenstrukturen initialisiert ist. Dadurch könnte die Speicherverwaltung die Initialisierung eines neuen Speicherbereichs auf bereits gecachetem Speicher vornehmen. Auch die Verwaltungsstrukturen der Speicherverwaltung könnten so abgeändert werden, dass die Laufzeit einer Allokation vorhersagbar wird. Für n vorher stattfindende Speicherallokationen ist die Laufzeit $\mathcal{O}(n)$, da zuerst über alle reservierten Speicherbereiche iteriert wird, um einen freien Slot zu finden. Die Verwaltungsstruktur müsste so geändert werden, dass sich eine Laufzeit von $\mathcal{O}(1)$ ergibt.

8.4.2. Alignment an Cache-Grenzen

In dieser Version der Speicherverwaltung werden Speicheranfragen in den verschiedenen Heaps nacheinander angeordnet. Die Platzierung der Komponenten und Component-Heaps erfolgt technisch bedingt zwar an Cache-Way Grenzen, die Verwaltung innerhalb dieser Heaps orientiert sich aber nicht an der Struktur des Cache. Speichernfragen könnten so platziert werden, dass möglichst viele ganze Cache-Lines für den Bereich verwendet werden. Zudem könnte der Verwendungszweck der reservierten Speicherbereiche beachtet werden und so eine Cache-Optimierte Ausrichtung erfolgen. Einige Forschungsarbeiten beschäftigen sich bereits mit dieser Thematik und stellen Lösungsansätze bereit (8.3).

8.4.3. Dynamische Speicherverwaltung für Systemfunktionen

Die Speicherverwaltung ist größtenteils als eigenständiges Modul angelegt, welches nicht genutzt werden muss. CyPhOS kann diese Speicherverwaltung verwenden um erweiterte Systemfunktionen zur Verfügung zu stellen. Dadurch könnten automatisch Komponenten zur Laufzeit erstellt werden, die selber dynamisch erstellte Trigger anbieten um Funktionen auszuführen.

Literaturverzeichnis

[1] *ARM Architecture Reference Manual*, armv7a and armv7-r edition. URL: `http://liris.cnrs.fr/~mmrissa/lib/exe/fetch.php?media=armv7-a-r-manual.pdf`.

[2] *CoreLink Level 2 Cache Controller L2C-310 Technical Reference Manual*, r3p3 edition. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246h/DDI0246H_l2c310_r3p3_trm.pdf`.

[3] *Cortex-A9 MPCore Technical Reference Manual*, r3p0 edition. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407g/DDI0407G_cortex_a9_mpcore_r3p0_trm.pdf`.

[4] *Cortex-A9 Technical Reference Manual*, r4p1 edition. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf`.

[5] *Working Draft, Standard for Programming Language C ++*, n4140 edition, 11 2014.

[6] Andreas Moshovos und Gurindar S. Sohi Amir Roth. Dependence based prefetching for linked data structures. In *ASPLOS VIII*, 1998.

[7] Hendrik Borghorst and Olaf Spinczyk. Increasing the predictability of modern COTS hardware through cache-aware OS-design. In *Proceedings of the 11th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '15)*, July 2015.

[8] Andre Seznec Dan N. Truong, Francois Bodin. Improving cache behavior of dynamically allocated data structures. Technical report, IRISA - INRIA Campus de Beaulieu 35042 Rennes, CEDEX France.

[9] James R. Larus Trishul M. Chilimbi, Mark D. Hill. Making pointer-based data structures cache conscious. *Computer*, 2000.

[10] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 2001.

Abbildungsverzeichnis

2.1. Interaktion zwischen zwei Komponenten 4

3.1. Hauptspeicherabbild von CyPhOS mit zwei Komponenten 5
3.2. Laden der Komponenten in den L2-Cache 6

4.1. Gemeinsam genutzter Speicher 8
4.2. Micro-Heaps und Speicherpools 10
4.3. Verkettung der Micro-Heaps und Component-Heaps 11

5.1. Adressübersicht der Heaps 17
5.2. Verwaltungseinheiten in einem Heap 20

7.1. Boxplots aller Zugriffszeiten 32
7.2. Histogramme aller Zugriffszeiten 33
7.3. FULLY_CACHED Speicher unter Störungen 35
7.4. DYNAMIC_CACHED Speicher unter Störungen 36
7.5. NOT_CACHED Speicher unter Störungen 37
7.6. FULLY_CACHED Speicher im parallelen Betrieb 39
7.7. DYNAMIC_CACHED Speicher im parallelen Betrieb 40
7.8. NOT_CACHED Speicher im parallelen Betrieb 41
7.9. FULLY_CACHED Speicher Hardware / Software verwaltet . . . 43
7.10. DYNAMIC_CACHED Speicher Hardware / Software verwaltet . 44
7.11. NOT_CACHED Speicher Hardware / Software verwaltet 45
7.12. FULLY_CACHED Speicher unter Störungen (Hardware und Software verwaltet) 46
7.13. DYNAMIC_CACHED Speicher unter Störungen (Hardware und Software verwaltet) 47
7.14. NOT_CACHED Speicher unter Störungen (Hardware und Software verwaltet) 48
7.15. Boxplots der Vorladezeiten unter Störungen 49
7.16. Histogramme der Vorladezeiten 50

B.1. Boxplots aller Zugriffszeiten VI
B.2. Histogramme aller Zugriffszeiten VII
B.3. FULLY_CACHED Speicher unter Störungen VIII
B.4. DYNAMIC_CACHED Speicher unter Störungen IX

B.5. NOT_CACHED Speicher unter Störungen X
B.6. FULLY_CACHED Speicher im parallelen Betrieb XI
B.7. DYNAMIC_CACHED Speicher im parallelen Betrieb XII
B.8. NOT_CACHED Speicher im parallelen Betrieb XIII
B.9. FULLY_CACHED Speicher Hardware / Software verwaltet . . . XIV
B.10. DYNAMIC_CACHED Speicher Hardware / Software verwaltet . XV
B.11. NOT_CACHED Speicher Hardware / Software verwaltet XVI
B.12. FULLY_CACHED Speicher unter Störungen (Hardware und Software verwaltet) XVII
B.13. DYNAMIC_CACHED Speicher unter Störungen (Hardware und Software verwaltet) XVII
B.14. NOT_CACHED Speicher unter Störungen (Hardware und Software verwaltet) XVIII
B.15. Boxplots der Vorladezeiten unter Störungen XIX
B.16. Histogramme der Vorladezeiten XX

Anhang

A. Statistische Kennzahlen der Evaluationsergebnisse (Kapitel 7)

A.1. Zugriffszeiten ohne Störungen

Messreihe	μ	σ^2	σ	σ/μ
FULLY_CACHED	35.57405	0.7243201	0.85107	0.0239239
FULLY_CACHED Größte Häufung (> 36.2)	36.76683	0.0120384	0.1097197	0.002984202
DYNAMIC_CACHED	40.76291	0.793944	0.8910353	0.02185897
DYNAMIC_CACHED Größte Häufung (> 41.0)	41.85473	0.03895577	0.1973722	0.004715648
NOT_CACHED	75.61834	0.05160237	0.2271616	0.003004054

A.2. Zugriffszeiten unter Störungen

A.2.1. Keine Verdrängungen

FULLY_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
DYNAMIC_CACHED Störung	36.26706	0.971065	0.9854263	0.02717139
DYNAMIC_CACHED Störung Größte Häufung (> 37.3)	38.01115	0.06664567	0.2581582	0.006791645
NOT_CACHED Störung	35.94628	0.613687	0.7833817	0.02179313
NOT_CACHED Störung Größte Häufung (> 36.9)	37.20027	0.01976302	0.140581	0.003779033

DYNAMIC_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
DYNAMIC_CACHED Störung	42.24875	1.09308	1.045505	0.02474641
DYNAMIC_CACHED Störung Größte Häufung (> 43.4)	44.22403	0.08847006	0.2974392	0.006725737
NOT_CACHED Störung	41.32582	0.6128287	0.7828338	0.01894297
NOT_CACHED Störung Größte Häufung (> 41.5)	42.39026	0.05120943	0.226295	0.005338373

NOT_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
DYNAMIC_CACHED Störung	76.25565	1.249793	1.117942	0.01466044
NOT_CACHED Störung	87.47345	0.1412045	0.3757719	0.00429584

A.2.2. Verdrängungen

FULLY_CACHED:

Parallele Störung	μ	σ^2	σ	σ/μ
4 Pools	36.26706	0.971065	0.9854263	0.02717139
11 Pools	36.69438	0.9068536	0.9522886	0.02595189

DYNAMIC_CACHED:

Parallele Störung	μ	σ^2	σ	σ/μ
4 Pools	42.24875	1.09308	1.045505	0.02474641
10 Pools	41.93246	1.05504	1.027151	0.02449538

NOT_CACHED:

Parallele Störung	μ	σ^2	σ	σ/μ
4 Pools	76.25565	1.249793	1.117942	0.01466044
11 Pools	75.79598	0.04501218	0.2121607	0.002799103

A.3. Hardware-Cache-Verwaltung

FULLY_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
Software Verwaltet	35.57405	0.7243201	0.85107	0.0239239
Software Verwaltet Größte Häufung (> 36.2)	36.76683	0.0120384	0.1097197	0.002984202
Hardware Verwaltet	28.2181	0.4015983	0.6337178	0.02245785
Hardware Verwaltet Größte Häufung (> 28.8)	29.08459	0.0327815	0.1810566	0.006225175

DYNAMIC_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
Software Verwaltet	40.76291	0.793944	0.8910353	0.02185897
Software Verwaltet Größte Häufung (> 41.0)	41.85473	0.03895577	0.1973722	0.004715648
Hardware Verwaltet	31.53373	0.3431684	0.5858058	0.01857712
Hardware Verwaltet Größte Häufung (> 31.9)	32.38957	0.01286437	0.1134212	0.003501781

NOT_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
Software Verwaltet	75.61834	0.05160237	0.2271616	0.003004054
Hardware Verwaltet	22.11832	2.362482	1.537037	0.06949156

B. Zugriffszeitmessungen auf einem Wandboard mit i.MX6 CPU

Die nachfolgenden Messungen sind unter exakt den gleichen Bedingungen wie in Kapitel 7 durchgeführt. Als Hardware Plattform für CyPhOS kommt hier allerdings ein Wandboard mit i.MX6 CPU (ebenfalls ARMv7 Cortex-A9) zum Einsatz. Die statistischen Kennzahlen der Messdaten sind wie in Kapitel 7 ausgewertet in Anhang C dargestellt.

B.1. Speicherzugriffe ohne Störung

Die Messungen erfolgen durch die Komponente `MemoryEvalMultipool` [1]. Zugegriffen wird auf 16 16 kB große Pools.

[1] `testcomponents/MemoryEvalMultipool.{h,cc}`

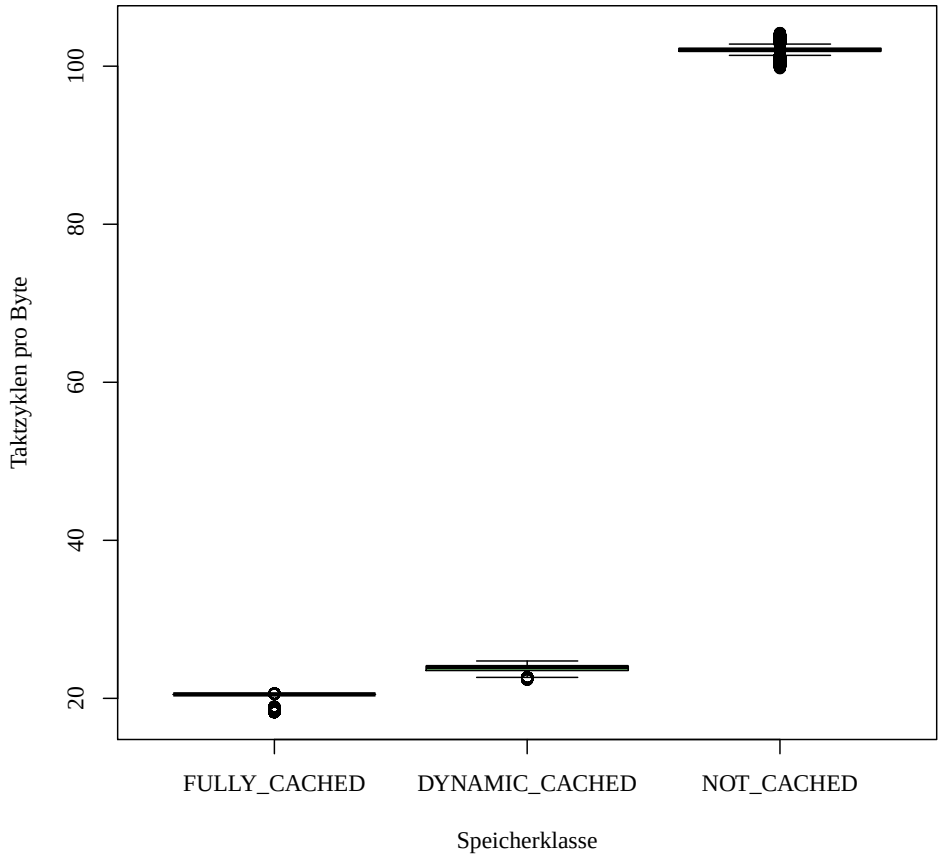

Abbildung B.1.: Boxplots aller Zugriffszeiten

Auffällig ist die deutliche geringere Zugriffszeit für gecachete Speicherinhalte ($\mu \approx 35$ für FULLY_CACHED und $\mu \approx 40$ für DYNAMIC_CACHED auf dem Exynos Prozessor). Zudem weichen die Zugriffszeiten für die zwei verschiedenen gecacheten Speicherklassen nur sehr gering voneinander ab. Die Zugriffszeiten auf nicht gecacheten Speicher sind jedoch deutlich größer ($\mu \approx 75$ auf dem Exynos Prozessor). Diese Ergebnisse lassen auf eine andere Architektur der Hauptspeicheranbindung und des Cache-Controller schließen.
Die prozentuale Schwankung der Werte um den Mittelwert schneidet aufgrund des geringeren Mittelwerts für immer gecachete Speicher zwar etwas schlechter ab, die absolute Schwankung ist aber für beide Prozessoren in etwa gleich ($\sigma \approx 0.85$ auf dem Exynos Prozessor und $\sigma \approx 0.85$ auf dem i.MX6 Prozessor). Dynamisch gecachete Speicher schneiden auf dem i.MX6 Prozessor deutlich besser ab, da auf dem Exynos Prozessor der dynamisch gecachete Speicher im Gegensatz zum immer gecacheten Speicher sowohl im Mittelwert als auch in der Schwankung deutlich höhere Werte aufweist.

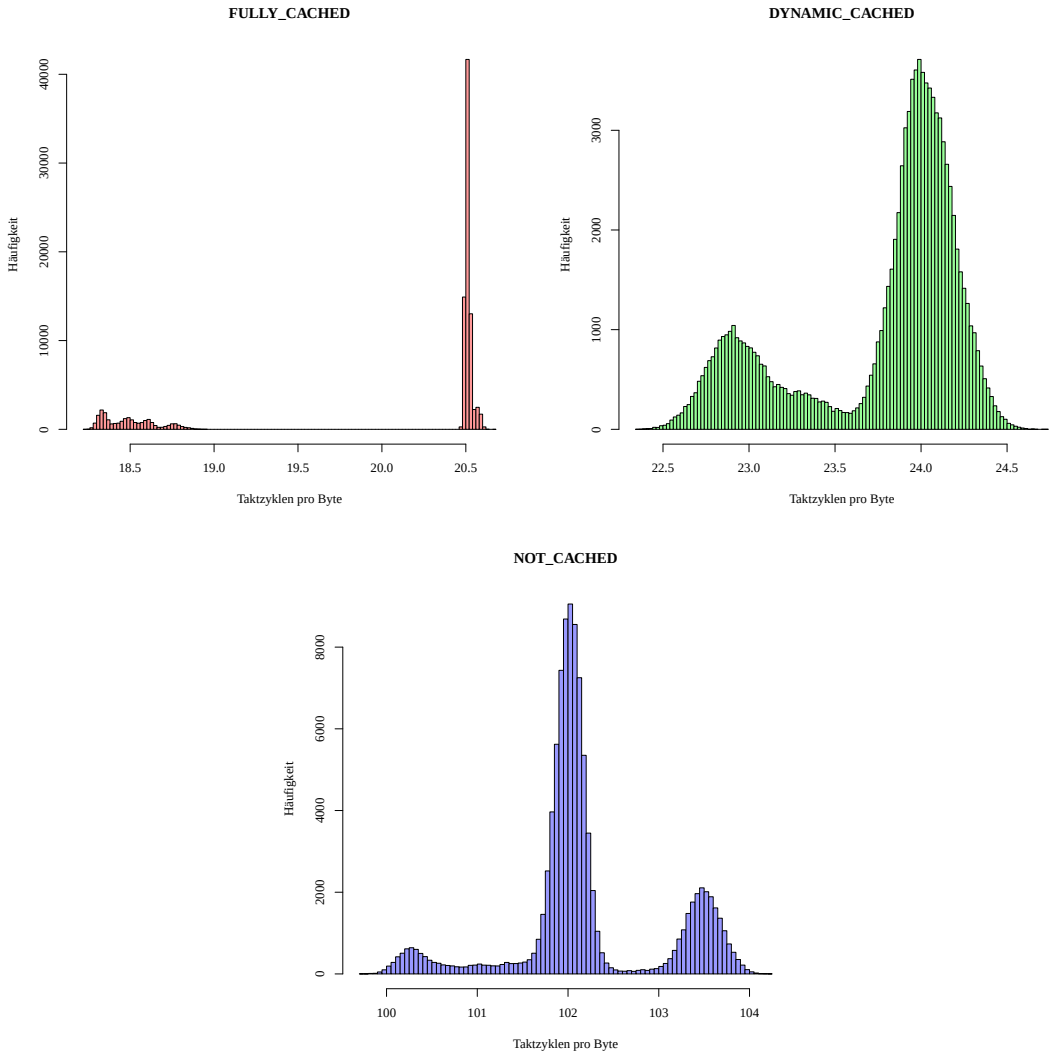

Abbildung B.2.: Histogramme aller Zugriffszeiten

B.2. Zugriffszeiten unter parallel stattfindenden Zugriffen

Die Messung wird auf den selben Daten wie in Kapitel B.1 durchgeführt. Die Komponente `MemoryEvalDisturber` [2] stört die Messung mit einem parallelen Zugriff auf vier Pools der Größe 32 kB.

[2] `testcomponents/MemoryEvalDisturber.{h,cc}`

B.2.1. Keine Verdrängungen im Cache

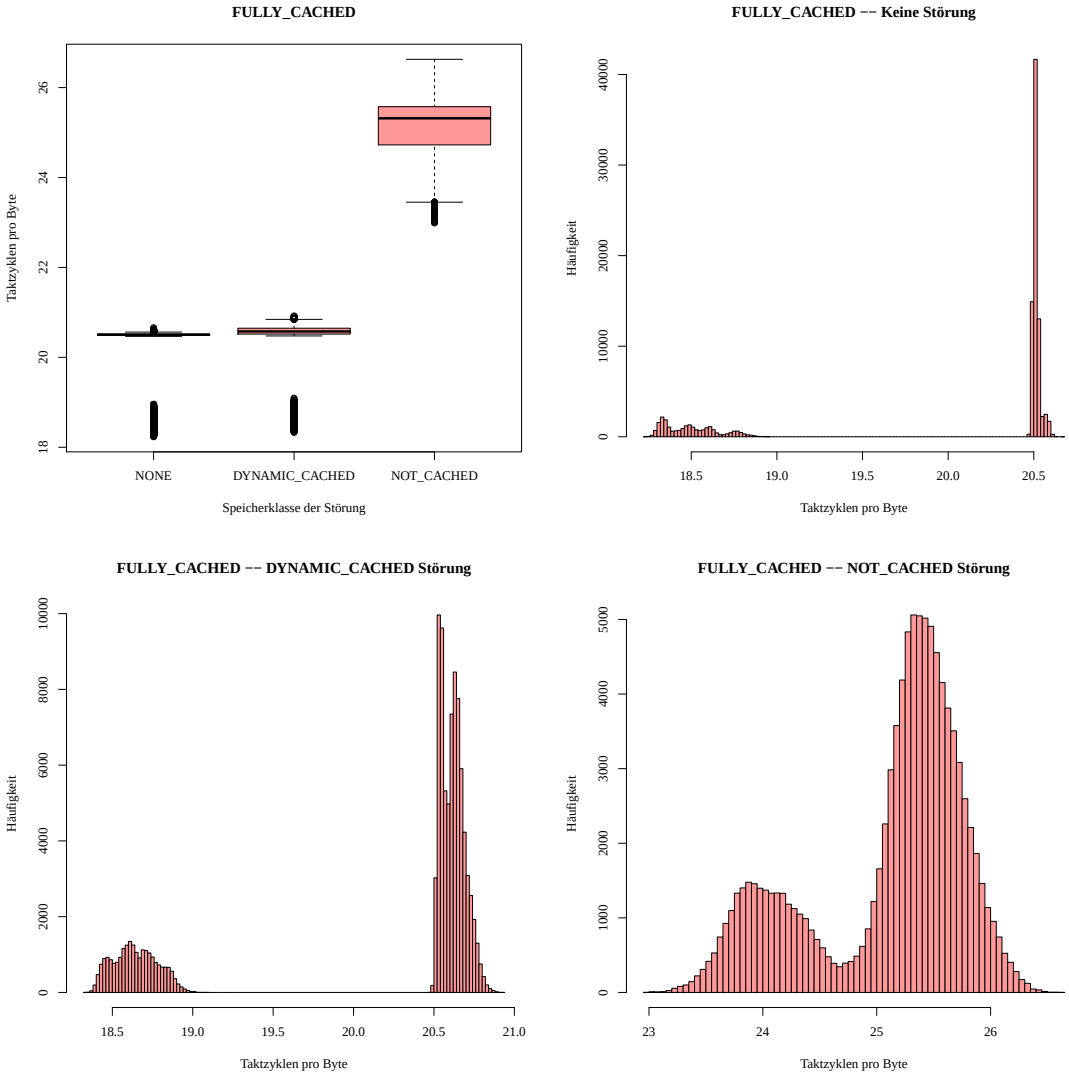

Abbildung B.3.: FULLY_CACHED Speicher unter Störungen

Das Verhalten unter parallelen Zugriffen unterscheidet sich für gecachete Speicher weitestgehend nicht von den Ergebnissen auf dem Exynos Prozessor. Der Maximalwert und die Schwankung steigen um einige wenige Zyklen pro Byte an. Die Auswirkung von einem parallelen Zugriff auf nicht gecacheten Speicher führt zu einer stärkeren Abweichung von der Messung ohne Störung als auf dem Exynos Prozessor. Zusammen mit den hohen gemessenen Zugriffszeiten für nicht gecacheten Speicher lässt dies auf eine schwächere Hauptspeicheranbindung auf dem i.MX6 Prozessor schließen.

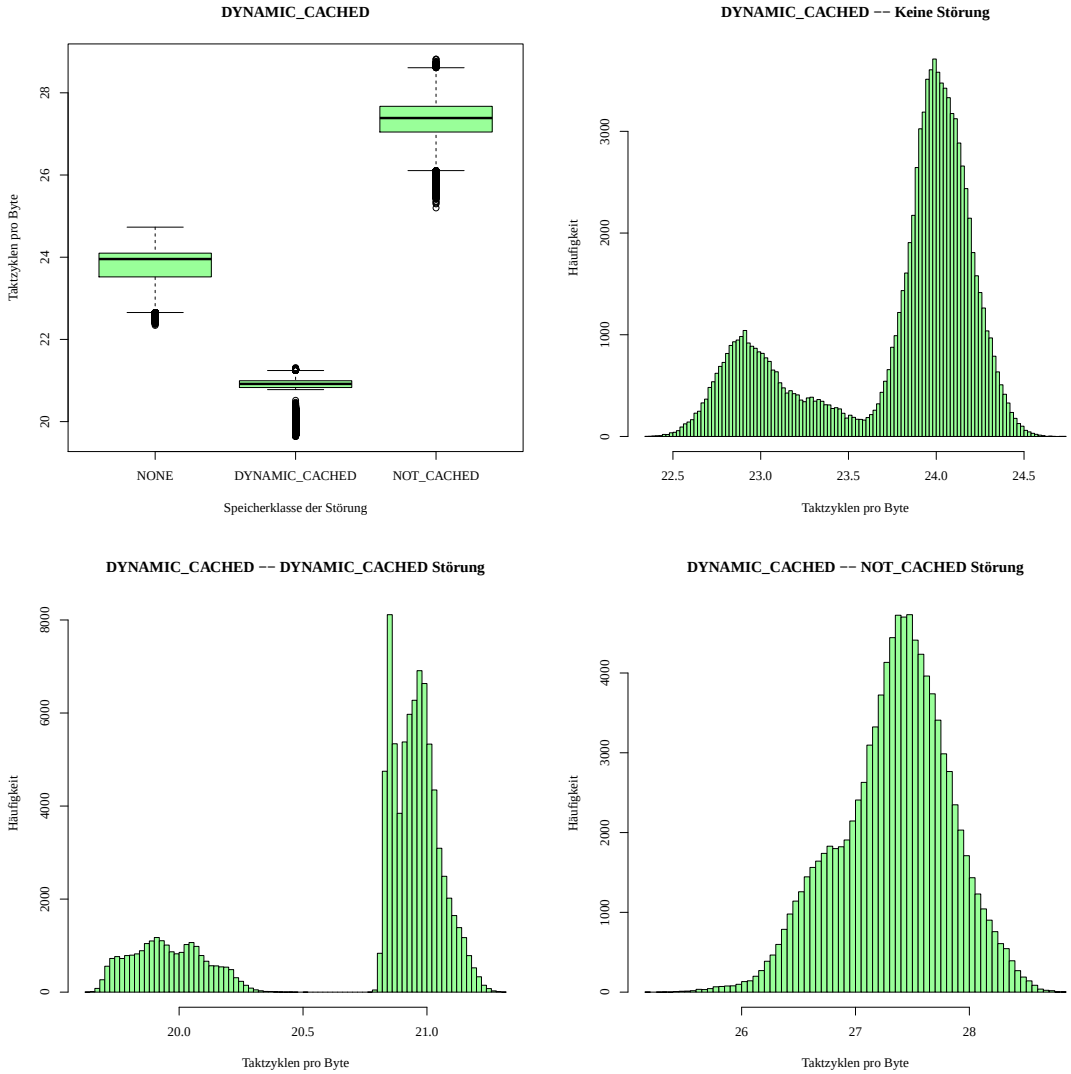

Abbildung B.4.: DYNAMIC_CACHED Speicher unter Störungen

Auffällig ist hier, dass bei einer Störung durch dynamisch gecacheten Speicher die Zugriffszeiten scheinbar gesenkt werden können. Dies kann möglicherweise aus der Vorladestrategie in die L1-Caches resultieren. Auch wie bei immer gecachetem Speicher (Abbildung B.3) hat ein paralleler Zugriff auf nicht gecacheten Speicher einen negativen Einfluss, der so auf dem Exynos Prozessor nicht zu beobachten ist.

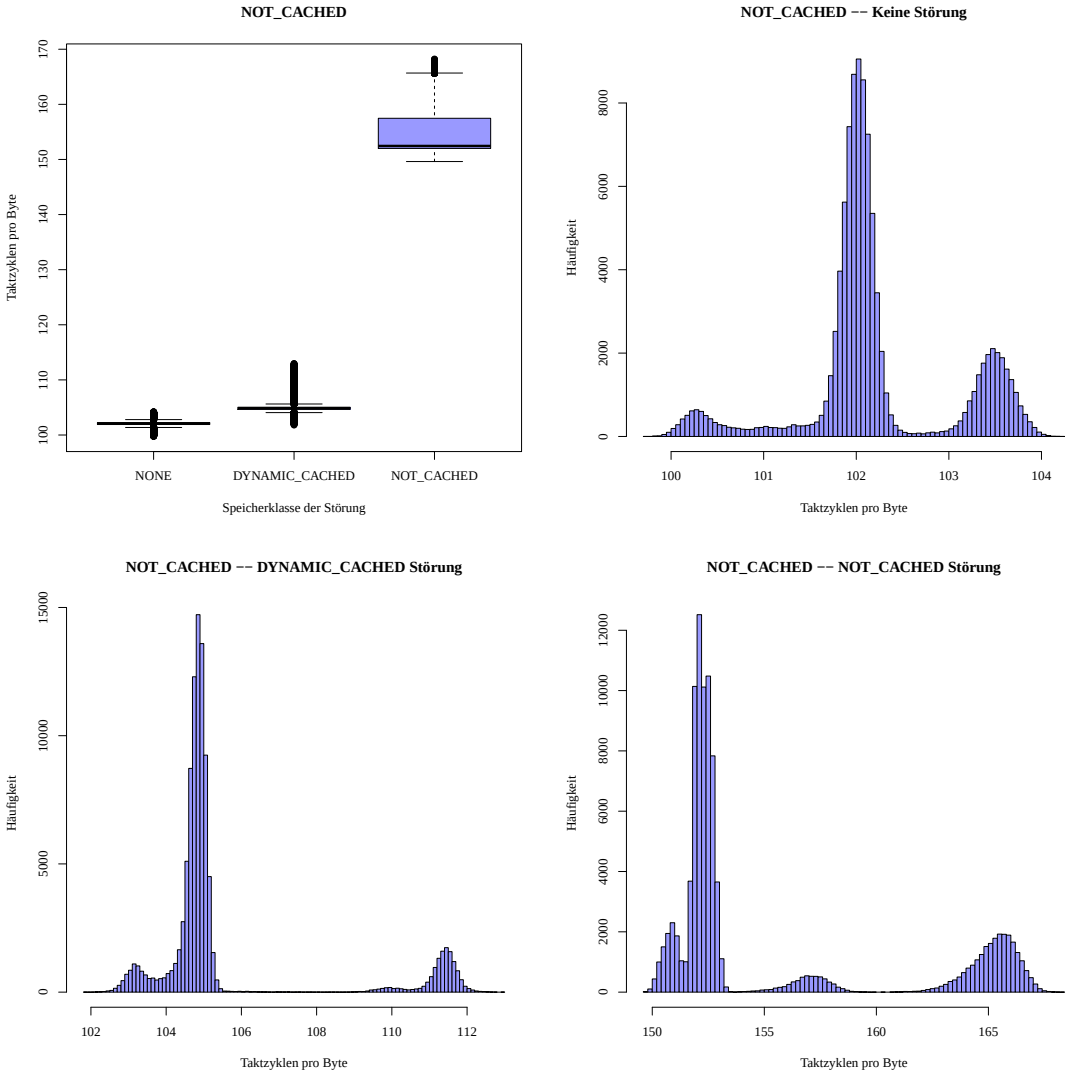

Abbildung B.5.: NOT_CACHED Speicher unter Störungen

Der Einfluss von parallelen Zugriffen auf gecachete Inhalte ist erwartungsgemäß sehr gering für nicht gecachete Speicherinhalte, dennoch stärker als bei der selben Messung auf dem Exynos Prozessor. Der Effekt einer weiteren Belastung der Hauptspeicheranbindung durch parallele Zugriffe auf nicht gecacheten Speicher zeigt sich hier deutlich größer als auf dem Exynos Prozessor. Daraus lässt sich abermals schließen, dass die Hauptspeicheranbindung weniger Leistungsfähig als die des Exynos Prozessors ist.

B.2.2. Verdrängungen im Cache

Die Messungen aus Kapitel B.2.1 werden wiederholt und die Größe der Störung auf elf bzw. zehn 32 kB Pools erweitert. Für die zu messenden Daten kommt gezwungenermaßen der selbe Priorisierungsmechanismus wie in Kapitel 7.4.2 zum Einsatz.

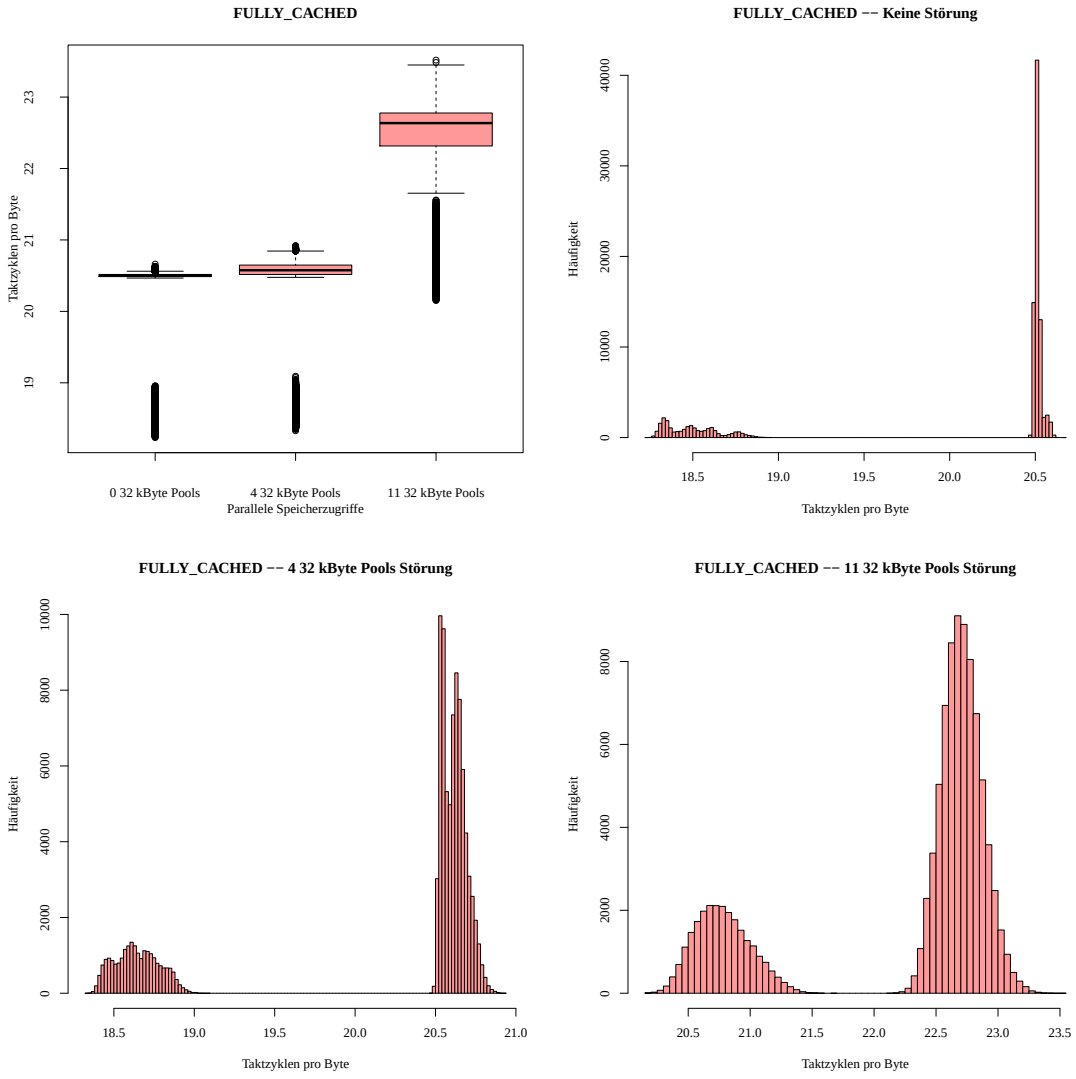

Abbildung B.6.: FULLY_CACHED Speicher im parallelen Betrieb

Im Vergleich zu den Messungen auf dem Exynos Prozessor fällt auf, dass der i.MX6 Prozessor stärker auf steigende parallele Störungen reagiert. Ähnlich wie auf dem Exynos Prozessor verändert sich die Verteilung der Zugriffszeiten nur geringfügig, es verändert sich hauptsächlich nur der Mittelwert.

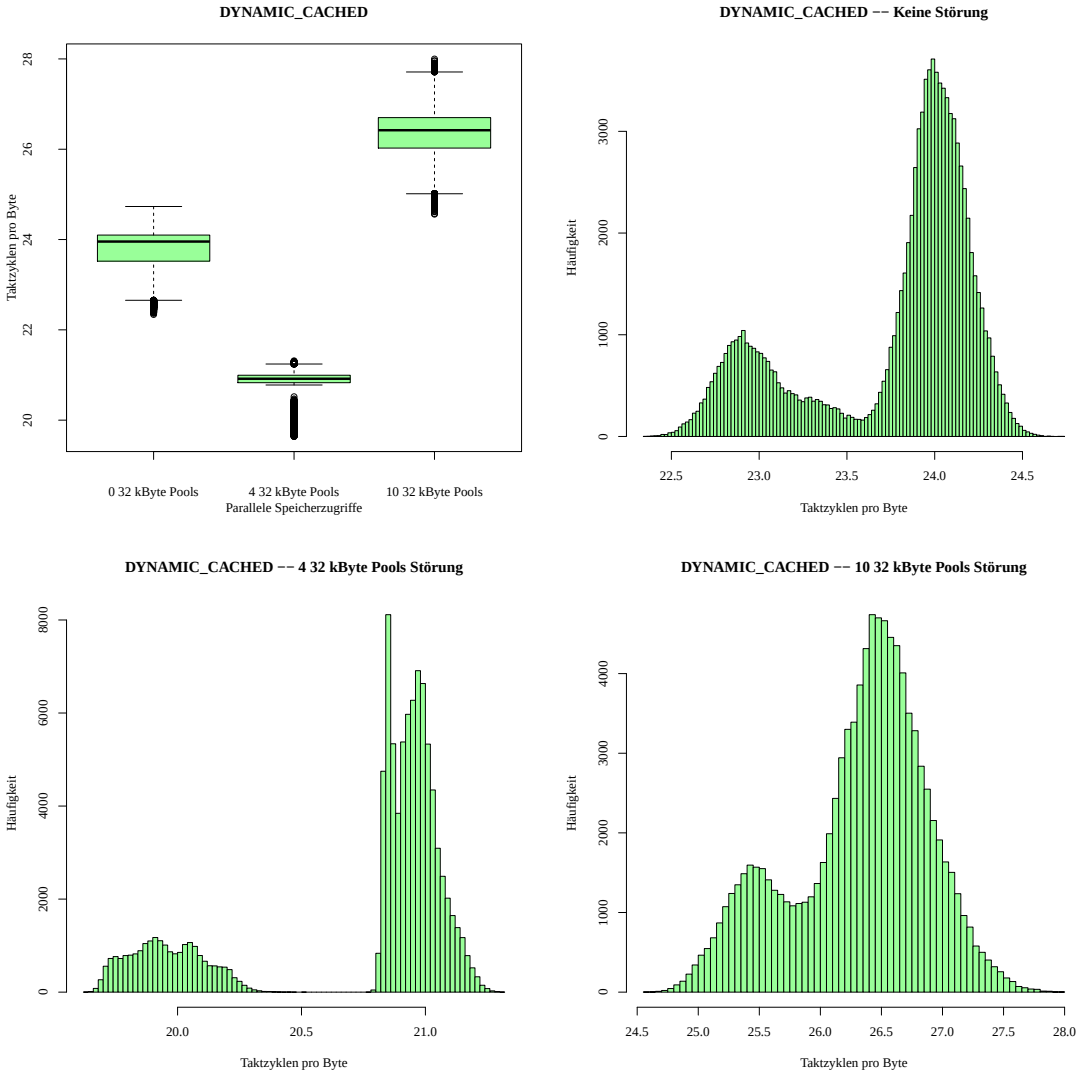

Abbildung B.7.: DYNAMIC_CACHED Speicher im parallelen Betrieb

Auch bei diesen Zugriffszeitmessungen fällt eine stärkere Reaktion auf steigende parallele Last aus (ausgenommen der Absenkung der Zugriffszeiten für 4 Heaps der Störung). Die Verteilungsfunktion änderst sich auch hier nur geringfügig, auch die Schwankung der Werte steigt nur geringfügig an.

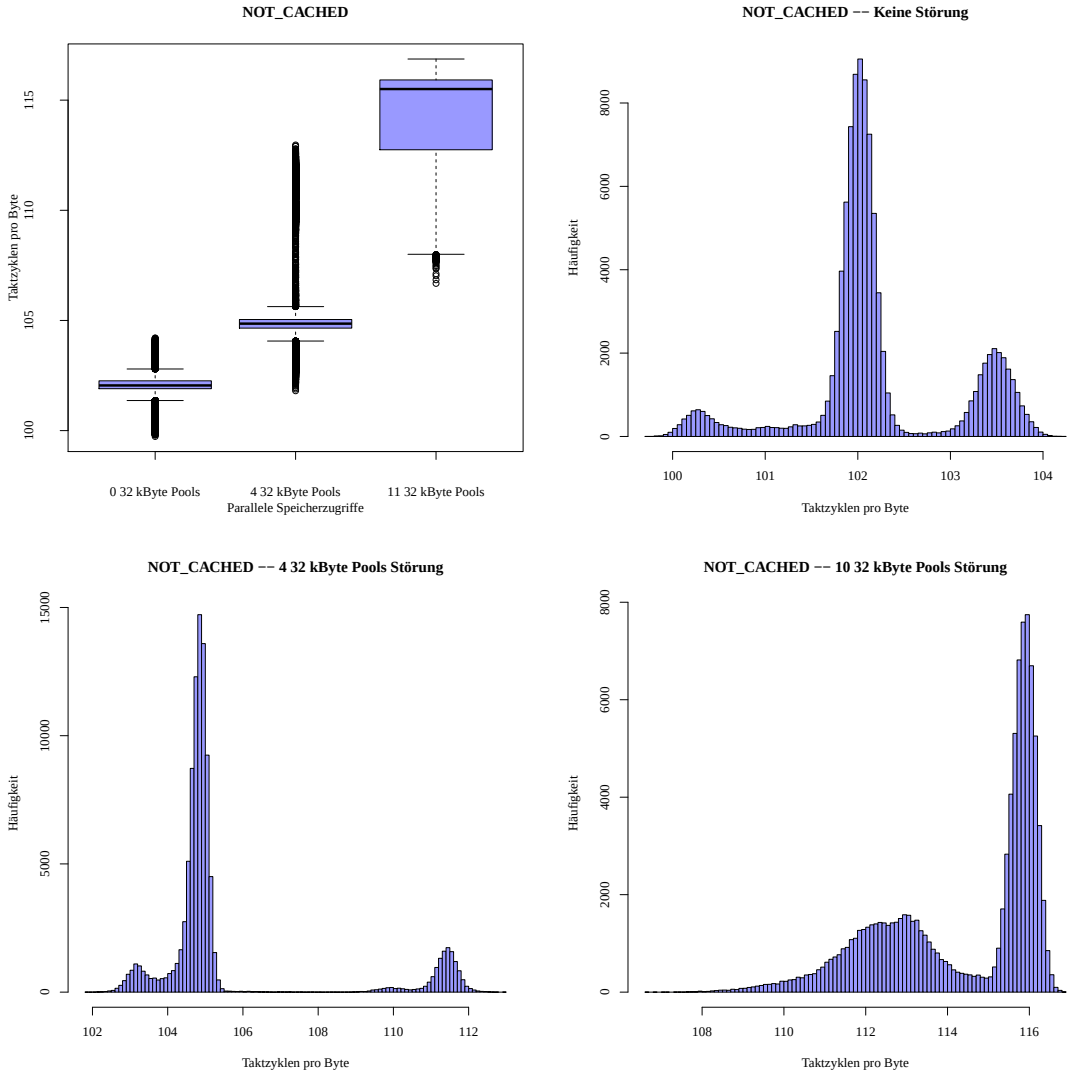

Abbildung B.8.: NOT_CACHED Speicher im parallelen Betrieb

Für nicht gecachete Speicher lässt sich die selbe Beobachtung machen, wie für immer und dynamisch gecachete Speicher. Der i.MX6 Prozessor reagiert stärker auf die parallele Störung als der Exynos Prozessor.

B.3. Vergleich mit reinem Hardware-Cache-Management

B.3.1. Kein paralleler Betrieb (vgl. Kapitel B.1)

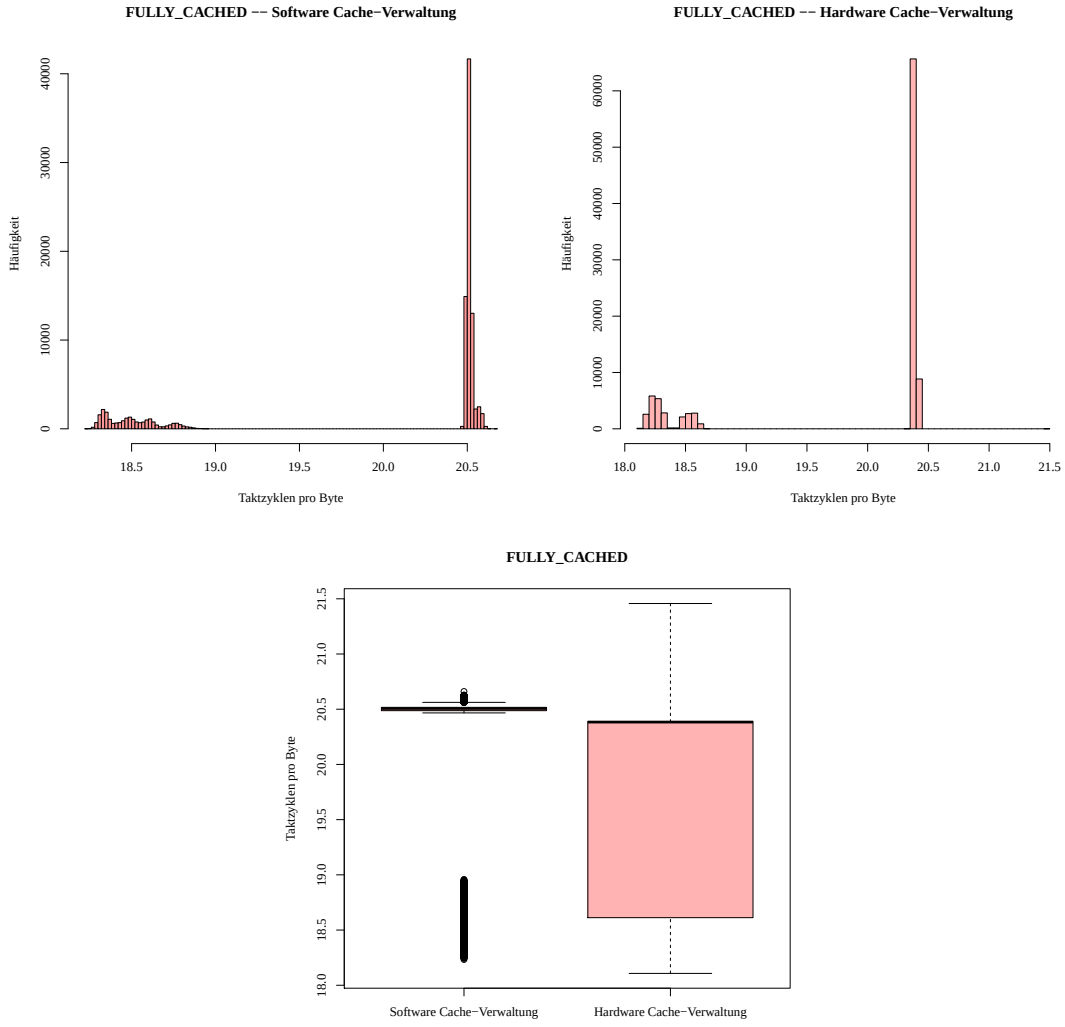

Abbildung B.9.: FULLY_CACHED Speicher Hardware / Software verwaltet

Die Software-Cache-Verwaltung schneidet im Gegensatz zum Exynos Prozessor besser ab als die Hardware-Cache-Verwaltung. Das Maximum der gemessenen Zugriffszeiten steigt unter der Hardware-Verwaltung sogar deutlich höher an als unter der Software-Verwaltung.

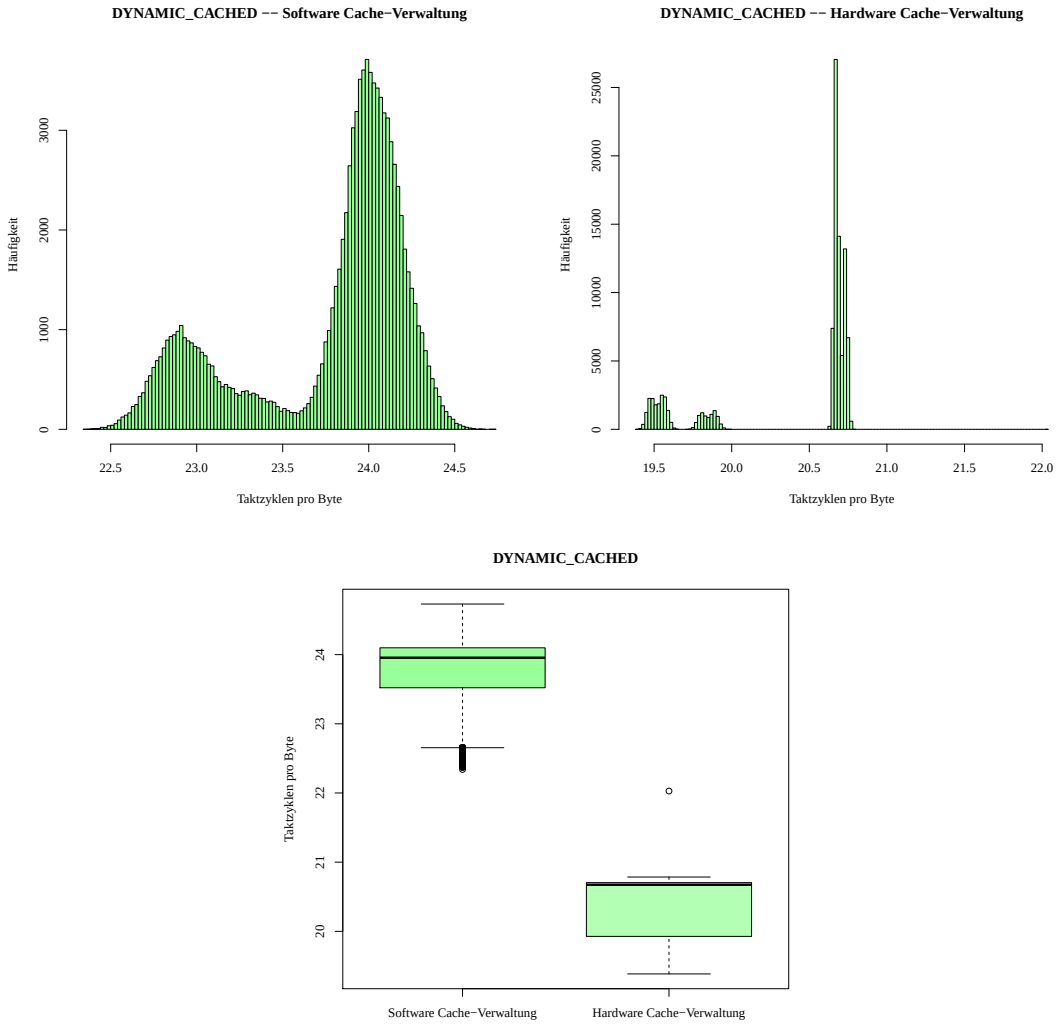

Abbildung B.10.: DYNAMIC_CACHED Speicher Hardware / Software verwaltet

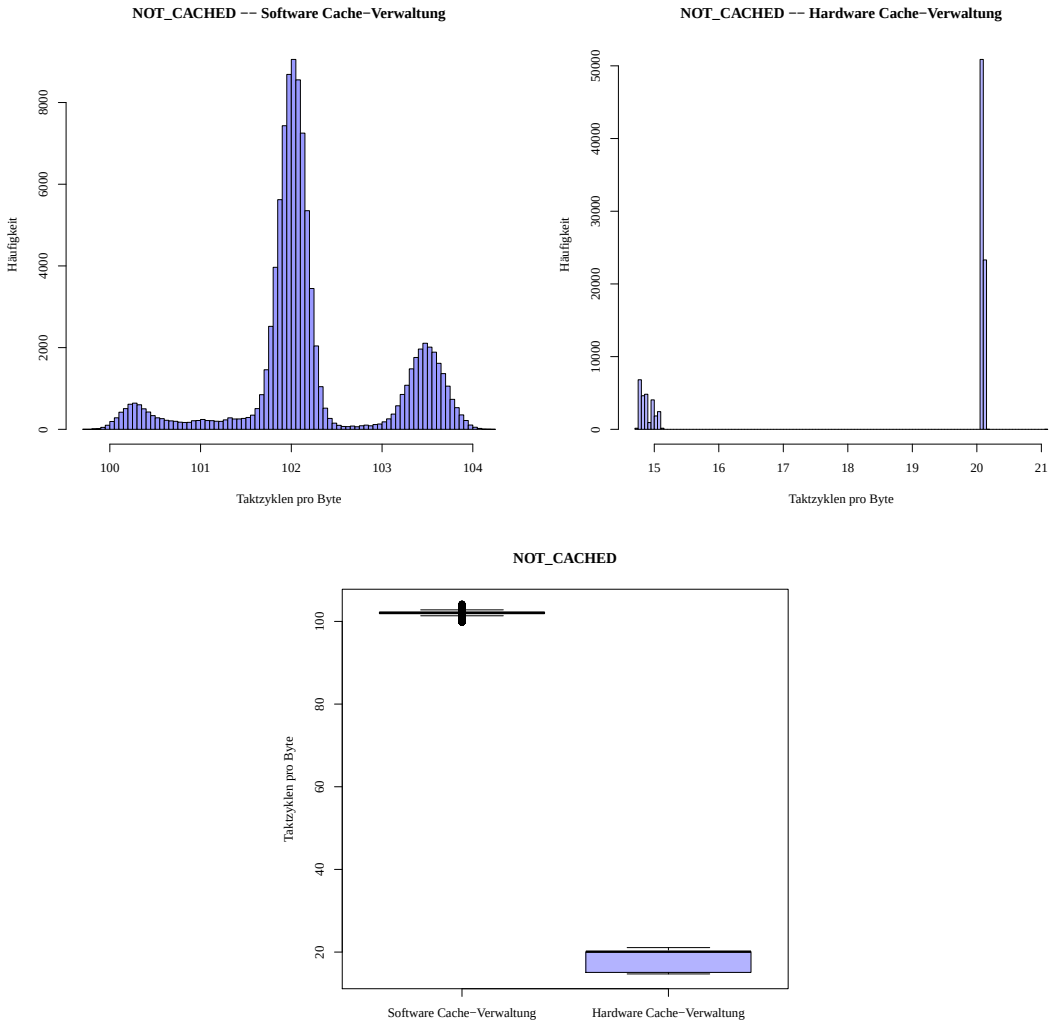

Abbildung B.11.: NOT_CACHED Speicher Hardware / Software verwaltet

Für nicht gecachete Speicher weist auch auf dem i.MX6 Prozessor die Hardware-Verwaltung deutlich geringere Zugriffszeiten auf, da diese Speicherinhalte unter der Hardware-Verwaltung nicht gesondert behandelt werden.

B.3.2. Paralleler Betrieb (vgl. Kapitel B.2)

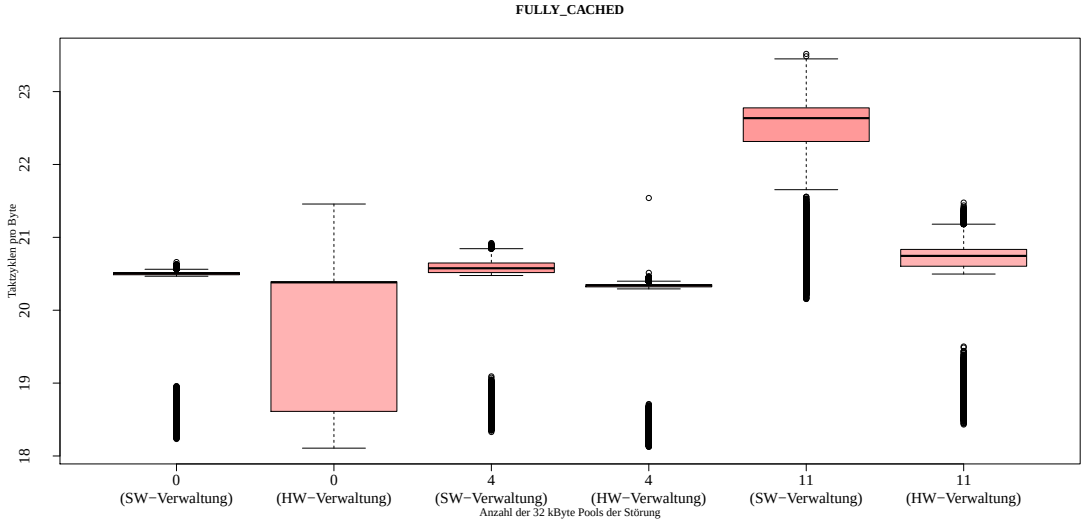

Abbildung B.12.: FULLY_CACHED Speicher unter Störungen (Hardware und Software verwaltet)

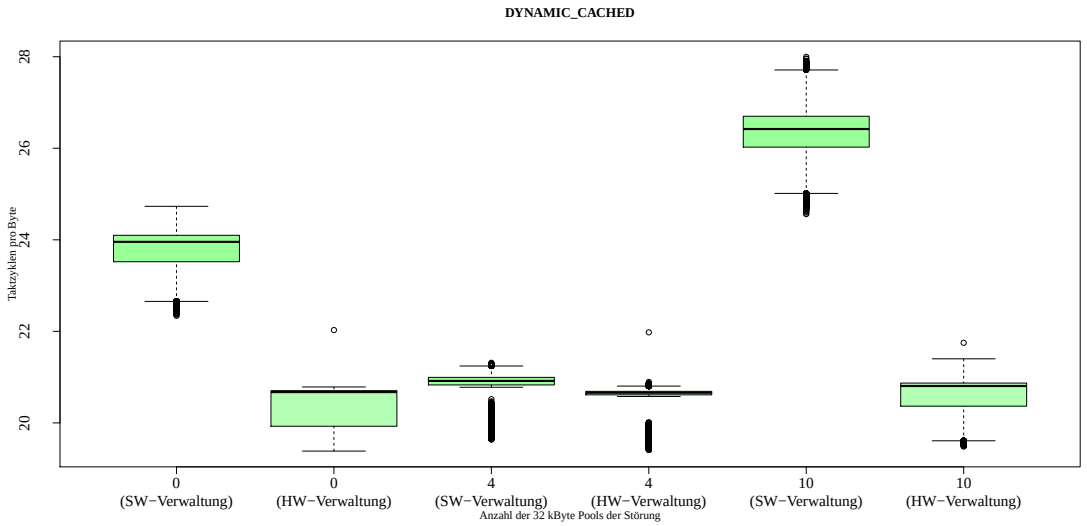

Abbildung B.13.: DYNAMIC_CACHED Speicher unter Störungen (Hardware und Software verwaltet)

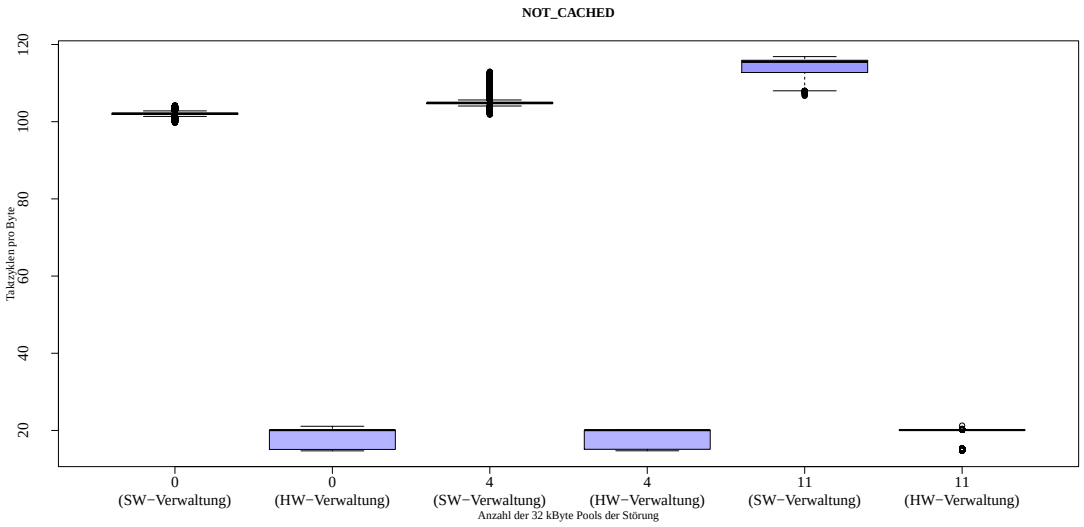

Abbildung B.14.: NOT_CACHED Speicher unter Störungen (Hardware und Software verwaltet)

Die Ergebnisse zeigen, dass die Verwendung der Software-Verwaltung in nahezu allen Fällen keine Nachteile gegenüber der Hardware-Verwaltung mit sich bringt. Bei genauerer Betrachtung der Messdaten schneidet die Software-Verwaltung in diesen Tests auch immer besser ab. Lediglich wenn das System über seine Leistungsfähigkeit hinaus belastet wird verbleiben die Zugriffszeiten bei der Hardware-Verwaltung deutlich geringer. Auffällig ist auch der Unterschied bei dynamisch gecachetem Speicher. Allerdings zeigen die Messdaten auch dort, dass die Hardware-Cache-Verwaltung mehr Ausreißer aufweist.

B.4. Verhalten der Vorladezeiten der Komponenten

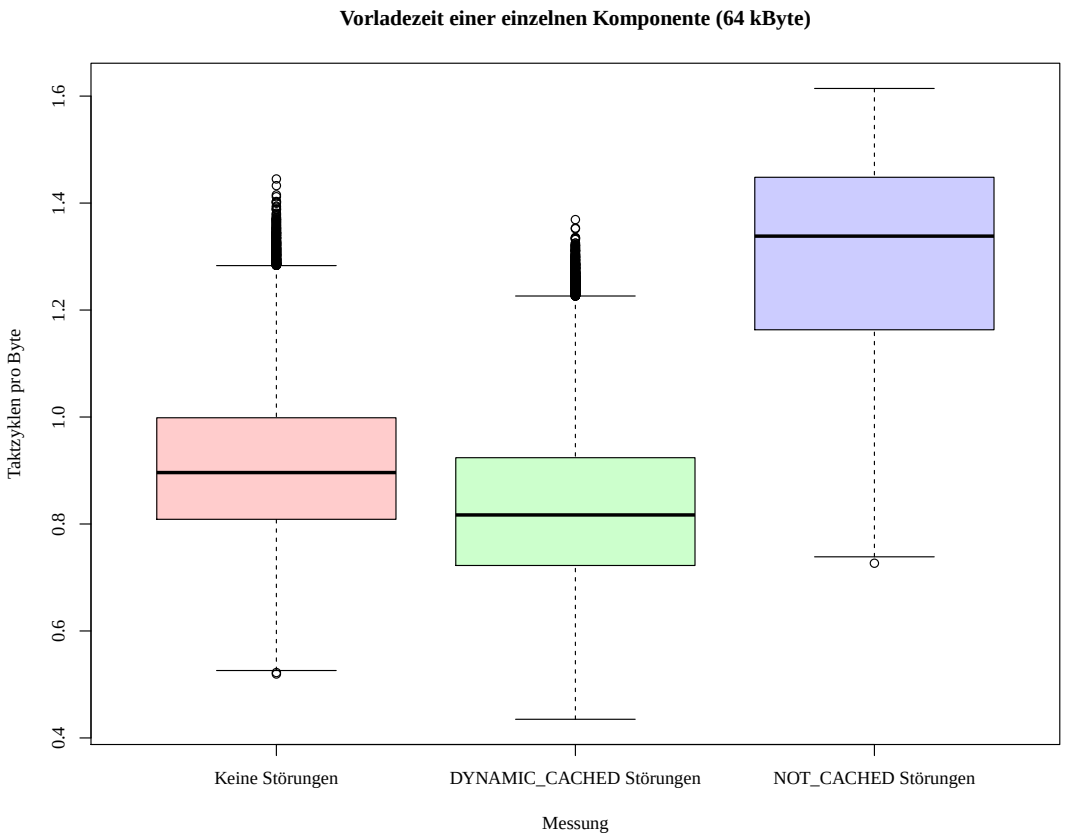

Abbildung B.15.: Boxplots der Vorladezeiten unter Störungen

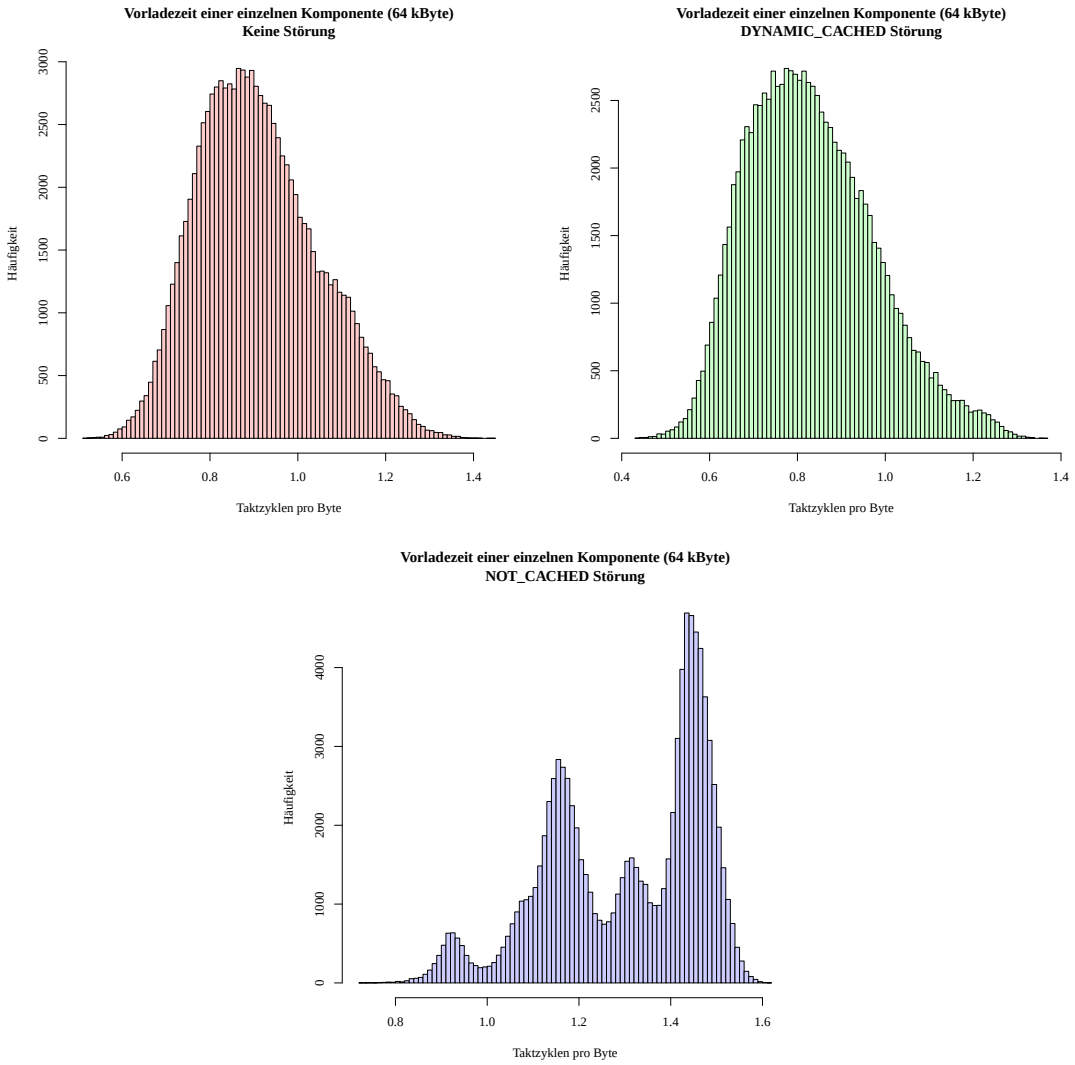

Abbildung B.16.: Histogramme der Vorladezeiten

Die Vorladezeiten der Komponenten weisen in diesen Messungen eine höhere Schwankung und deutlich mehr Ausreißer auf, als auf dem Exynos Prozessor. Die Auswirkung paralleler Störung haben einen ähnlich geringen Einfluss wie auch auf dem Exynos Prozessor. Die vielen Ausreißer und auch die stärkere Schwankung könnten in einer schwächeren Hauptspeicheranbindung begründet sein.

B.5. Fazit der Evaluation

Prinzipiell zeigen sich die Zugriffszeiten auf dem Wandboard deutlich geringer und vorhersagbarer als auf dem Odroid Board. Auch unter parallel stattfindenden Speicherzugriffen steigen die Mittelwerte und Schwankungen nicht sehr stark an, so dass Zeitgarantien gegeben werden könnten.
Wird das System über seine Leistungsfähigkeit hinaus belastet, so steigen die Zugriffszeiten merklich an, die Schwankung steigt aber nur sehr gering, so dass ein zeitkritischer Betrieb auch unter diesen Bedingungen noch möglich ist. Der Vergleich mit der Hardware-Cache-Verwaltung zeigt, dass in allen Fällen, bis auf die übermäßige Auslastung des Systems, die Software-Verwaltung sinnvoller ist. Die Zugriffszeiten verhalten sich mit der Software-Cache-Verwaltung geringer und stabiler.
Insgesamt lassen die Messungen eine schwache Hauptspeicheranbindung vermuten, weshalb massive Zugriffe auf nicht gecacheten Speicher problematisch sein können. Nicht nur die Zugriffszeit dieser Zugriffe ist hoch, sondern auch der Einfluss auf andere, gecachete Zugriffe ist groß. Dieser Schluss kann auch bei dem Exynos Prozessor gezogen werden, hier lässt sich der Effekt jedoch noch extremer beobachten.

C. Statistische Kennzahlen der Evaluationsergebnisse auf einem Wandboard (Anhang B)

C.1. Zugriffszeiten ohne Störungen

Messreihe	μ	σ^2	σ	σ/μ
FULLY_CACHED	20.04234	0.7352299	0.8574555	0.04278221
FULLY_CACHED Größte Häufung ($>$ 20.0)	20.51466	0.0004402831	0.02098292	0.001022826
DYNAMIC_CACHED	23.76976	0.2349881	0.4847557	0.0203938
DYNAMIC_CACHED Größte Häufung ($>$ 23.6)	24.03264	0.0270953	0.1646065	0.006849289
NOT_CACHED	102.1975	0.6428303	0.801767	0.007845267

C.2. Zugriffszeiten unter Störungen

C.2.1. Keine Verdrängungen

FULLY_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
DYNAMIC_CACHED Störung	20.16737	0.693594	0.8328229	0.04129557
DYNAMIC_CACHED Störung Größte Häufung ($>$ 20.0)	20.61726	0.005174809	0.07193614	0.003489123
NOT_CACHED Störung	25.10427	0.4692834	0.6850426	0.02728789
NOT_CACHED Störung Größte Häufung ($>$ 24.8)	25.46543	0.08420073	0.2901736	0.01139481

DYNAMIC_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
DYNAMIC_CACHED Störung	20.73333	0.1860447	0.431329	0.02080365
DYNAMIC_CACHED Störung Größte Häufung (> 20.5)	20.95865	0.008148451	0.09026877	0.004306993
NOT_CACHED Störung	27.34239	0.2386002	0.4884672	0.01786483

NOT_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
DYNAMIC_CACHED Störung	105.5483	5.308639	2.304048	0.02182932
NOT_CACHED Störung	155.4103	31.10299	5.577005	0.03588569

C.2.2. Verdrängungen

FULLY_CACHED:

Parallele Störung	μ	σ^2	σ	σ/μ
4 Pools	20.16737	0.693594	0.8328229	0.04129557
11 Pools	22.23248	0.7284629	0.8535004	0.03838979

DYNAMIC_CACHED:

Parallele Störung	μ	σ^2	σ	σ/μ
4 Pools	20.73333	0.1860447	0.431329	0.02080365
10 Pools	26.3246	0.3098195	0.5566143	0.02114426

NOT_CACHED:

Parallele Störung	μ	σ^2	σ	σ/μ
4 Pools	105.5483	5.308639	2.304048	0.02182932
11 Pools	114.3615	3.62937	1.90509	0.01665849

C.3. Hardware-Cache-Verwaltung

FULLY_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
Software Verwaltet	20.04234	0.7352299	0.8574555	0.04278221
Software Verwaltet Größte Häufung (> 20.0)	20.51466	0.0004402831	0.02098292	0.001022826
Hardware Verwaltet	19.86775	0.7945178	0.8913573	0.04486453
Hardware Verwaltet Größte Häufung (> 20.0)	20.38714	0.0001333897	0.01154944	0.0005665063

DYNAMIC_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
Software Verwaltet	23.76976	0.2349881	0.4847557	0.0203938
Software Verwaltet Größte Häufung (> 23.6)	24.03264	0.0270953	0.1646065	0.006849289
Hardware Verwaltet	20.42492	0.2195706	0.4685837	0.02294176
Hardware Verwaltet Größte Häufung (> 20.5)	20.42492	0.2195706	0.4685837	0.02294176

NOT_CACHED:

Messreihe	μ	σ^2	σ	σ/μ
Software Verwaltet	102.1975	0.6428303	0.801767	0.007845267
Hardware Verwaltet	18.75164	5.198336	2.279986	0.1215886

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift