Leistungsbewertung einer Linux-basierten Cluster Checkpointing-Implementierung

Bachelorarbeit

von

Eugen Feller

aus

Düsseldorf

vorgelegt an der

Abteilung Betriebssysteme
Prof. Dr. Michael Schöttner
Heinrich-Heine-Universität Düsseldorf

06. März 2008

Gutachter:
Prof. Dr. Michael Schöttner
Prof. Dr. Stefan Conrad

Danksagung

An dieser Stelle danke ich meinen Gutachtern Prof. Dr. Michael Schöttner und Prof. Dr. Stefan Conrad für die Begutachtung dieser Arbeit.

Außerdem bedanke ich mich bei meinem Betreuer Dipl.-Inf. John Mehnert-Spahn. Mit seinen Tipps, Ideen und Anregungen half er mir die Arbeit sowohl inhaltlich, als auch strukturell zu optimieren.

Meiner Famile danke ich für ihre Unterstützung während meines ganzen Studiums. Ohne sie wäre das ganze nicht möglich geworden.

Als letztes danke ich allen, die Interesse an dieser Arbeit zeigten und mich mit ihren Ratschlägen unterstützten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AVL	Adelson-Velskii-Landis
BER	Backward Error Recovery
COW	Copy-On-Write
FER	Forward Error Recovery
KDFS	Kerrighed Distributed File System
MAW	Malloc and Write
MGS	Modified Gram-Schmidt
PFN	Page Frame Number
PTE	Page Table Entry
PWD	Piecewise Deterministic
RBT	Red-Black-Tree
SMP	Symmetrisches Multiprozessorsystem
SSI	Single System Image
VMA	Virtual Memory Area

Kapitel 1

Einleitung

1.1 Motivation

Die Komplexität der IT-Systeme steigt ununterbrochen. Statistisch gesehen steigt mit der Komplexität der Systeme auch die Fehleranfälligkeit. Aus diesem Grund müssen Verfahren entwickelt werden, die diese Systeme möglichst tolerant gegenüber Fehlern machen. In einem Fehlerfall kann mit Hilfe dieser Verfahren die Funktionalität weiter gewährleistet werden. Fehlertoleranz wird bei allen Systemen benötigt, bei denen ein Ausfall schwerwiegende Folgen oder große Verluste verursachen kann. Ein fehlertolerantes System muss trotz Fehlern und internen Ausfällen seine Funktionen weiter erfüllen. Ein solches System kann Fehler enthalten, darf sie jedoch nicht sichtbar machen. Solche Fehler oder Ausfälle dürfen somit keine externe Wirkung zeigen.
Die Fähigkeit Fehler zur Laufzeit zu entdecken und entsprechend zu behandeln trägt grundlegend zur Sicherheit eines Systems bei. Solche Fehler können sowohl Entwicklungsfehler als auch Hardwareausfälle oder unerwartete Situationen sein. Der Entwickler sollte somit alle möglichen Fehler vorhersehen und für jeden Fehler eine Behandlung implementieren. Dies ist meist nicht möglich. Vorhersehbare Ausfälle und Fehler dürfen keine negativen Auswirkungen mit sich führen. Für Situationen, die keiner vorhersehen kann, wäre allein schon die Erkennung ideal. Dies ist äußerst schwierig, wenn man bedenkt, dass eine solche Situation nicht einmal der Entwickler vorher kannte. Ein verlässliches System muss jedoch sowohl erwartete als auch unerwartete Situationen meistern. Ein fehlertolerantes System muss seine Funktionalität trotz des Eintretens von Fehlern möglichst ohne sichtbare Konsequenzen fortsetzen. Dafür bedarf es unter anderem einer,

sofortigen automatischen Rekonfiguration [Mon99], wobei die fehlerhaften Komponenten ausgelassen werden müssen. Bei der Integration einer ausgefallen oder ausgeschalteten Komponente muss ihr Zustand an den des Systems angepasst werden. Dafür gibt es die Möglichkeit der Backward Error Recovery (BER) und der Fordward Error Recovery (FER). Bei der BER wird der letzte fehlerfreie Zustand wiederhergestellt. Dafür muss das System einen aktuellen, konsistenten und stabilen Zustand (Checkpoint) in definierten Zeitintervallen auf ein sicheres Medium (z.B. ein RAID) speichern. Dadurch wird eine BER zum aktuellsten, fehlerfreien Zustand ermöglicht. Der Vorgang des Abspeicherns (Checkpointing) kann dabei entweder sequentiell oder inkrementell durchgeführt werden. Bei dem sequentiellen Checkpointing wird stets eine komplette Abbildung der Anwendung gesichert. Es genügt in der Regel jedoch nur die Änderungen seit dem letzten Checkpoint zu sichern, wie dies beim inkrementellen Checkpointing gemacht wird. In diesem Falle verbrauchen die Sicherungen weniger Speicherplatz. Im Falle eines Ausfalls geht das System dann zurück zum letzten konsistenten Checkpoint und führt ab dieser Stelle den Prozess fort. Bei der FER wird versucht durch fehlerkorrigierende Hamming-Codes die Fehler vorauszuahnen. Diese Methode wird in dieser Arbeit nicht weiter betrachtet.

1.2 Ziel der Arbeit

In dieser Bachelorarbeit soll der bestehende Kerrighed Checkpointer um die Möglichkeit erweitert werden, inkrementelle Sicherungen erstellen zu können. Beim inkrementellen Sichern der seit dem letzten Checkpoint unveränderten Daten, wird ein möglicher I/O-Overhead verhindert und somit eine schnellere Sicherung gewährleistet. Da sich die physikalischen Seiten eines Prozesses beim inkrementellen Checkpointing verteilt auf der Festplatte befinden, bedarf dies der Konzeption einer Verwaltungsstruktur, die eine Lokalisierung dieser Seiten ermöglicht. Anschließend soll ein Mechanismus zum Abfangen der Schreibzugriffe und Einsammeln der veränderten Seiten auf Kernel-Ebene entwickelt werden. Dieser Mechanismus, der auf einer Verwaltungsstruktur aufbaut, wird im Rahmen dieser Arbeit in den bestehenden Kerrighed Checkpointer integriert. Anschließend wird die Performance des inkrementellen Checkpointings der Performance des sequentiellen Checkpointings anhand von Messungen gegenübergestellt.

1.3 Struktur der Arbeit

Die Arbeit ist in vier inhaltliche Abschnitte unterteilt: Grundlagen, Implementierung, Leistungsbewertung und Zusammenfassung.

In Kapitel 2 werden die theoretischen Grundlagen erläutert, wobei zuerst das Projekt Kerrighed vorgestellt wird und anschließend auf die Grundlagen des Checkpointings im Allgemeinen und innerhalb dieses Projekts eingegangen wird.

Kapitel 3 widmet sich dann der Implementierung des inkrementellen Checkpointings. Als erstes wird eine Übersicht über das Kapitel gegeben. Im folgenden Verlauf wird detailliert die Implementierung vorgestellt.

Die Verbesserungen, die sich durch die in Kapitel 3 vorgestellte Implementierung ergeben, werden in Kapitel 4 gemessen und anschließend analysiert.

In Kapitel 5 folgt schließlich eine Zusammenfassung der erreichten Ergebnisse und ein Ausblick.

Zuletzt befinden sich im Anhang A die Messergebnisse der in dieser Arbeit vorgestellten Programme in tabellarischer Form.

Kapitel 2

Grundlagen

2.1 SSI Cluster (Kerrighed)

Der Begriff Kerrighed [LGVM04] ist aus einem Forschungsprojekt entstanden, das ursprünglich mit einem Entwurf im Jahre 1999 gestartet ist. Kerrighed wird derzeit von Kerlabs entwickelt und ist unter der Adresse `http://www.kerrighed.org` erreichbar. Das Ziel dieses Projektes besteht darin, den Cluster als ein einziges, symmetrisches Multiprozessorsystem (SMP) darzustellen. Kerrighed setzt sich zusammen aus einem Satz von den im Linux-Kernel verteilten Diensten, die für das globale Management der Ressourcen des Clusters verantwortlich sind.
Kerrighed bietet einen konfigurierbaren, globalen Prozess-Scheduler. Bei der Nutzung des Kerrighed Scheduler Konfigurationsprogramms können Richtlinien leicht geschrieben und in den laufenden Betrieb des Clusters eingebaut werden. Kerrighed verfügt außerdem über eine Richtlinie, die es erlaubt, die Auslastung des Prozessors (CPU) dynamisch zu regeln, indem ein präventives, vom Empfänger ausgelöstes Programm zur Prozessmigration genutzt wird. Wenn ein Knoten ausgelastet sein sollte, entdeckt das System ein Ungleichgewicht und verlagert einen Prozess von dem ausgelasteten auf einen weniger ausgelasteten Knoten. Der Migrationsmechanismus von Kerrighed basiert auf mehreren Mechanismen, wie z.B. Process Ghosting, Containers, Migrable Stream und Kerrighed Distributed File System (KDFS). Prozess Ghosting [VLM+05] wird eingesetzt, um die Informationen über den Prozesszustand zu extrahieren und die entsprechenden Daten auf einem Datenträger zu speichern. Dieser Datenträger kann eine Festplatte, ein Netzwerk oder auch der Arbeitsspeicher sein. Der Container-Mechanismus [MGL04]

wird eingesetzt, um die Daten zwischen den Knoten auszutauschen. Dabei wird die Datenkonsistenz gewährleistet.

2.2 Checkpointing Grundlagen

2.2.1 Sequentielles und Inkrementelles Checkpointing

Sequentielles Checkpointing bietet die Möglichkeit das Checkpointing auf der Systemebene durchzuführen. Dabei werden alle zu einem Prozess gehörenden Daten als Checkpoint auf einem, in der Regel nicht flüchtigen, Speicher gesichert. Dies betrifft sowohl die verwendeten Speicherbereiche der physikalischen Seiten als auch die Dateideskriptoren und weitere Kernstrukturen des Prozesses. Die Menge der gesicherten Daten kann durch inkrementelle Sicherung des Prozesszustandes reduziert werden. Bei diesem Verfahren werden nur die seit dem letzten Checkpoint veränderten, physikalischen Seiten gespeichert.

2.2.2 Checkpoint-basierte Rollback-Recovery

2.2.2.1 Übersicht

Bei der checkpoint-basierten Rollback-Recovery wird zwischen koordiniertem und unabhängigem Checkpointing unterschieden. Die checkpoint-basierte Rollback-Recovery verlässt sich im Gegensatz zur log-basierten Rollback-Recovery ausschließlich auf die gesetzten Checkpoints. Bei den Systemen, die Checkpointing verwenden, werden ein oder mehrere Prozesse im Fehlerfall wiederhergestellt. Die anschließende Ausführung des wiederhergestellten Prozesses entspricht dabei der Ausführung vor dem Fehler.

2.2.2.2 Unabhängiges Checkpointing

Das unabhängige Checkpointing [Klü07] erlaubt es den Prozessen selbständig Checkpoints zu setzen. Dadurch entfällt die Koordination der Prozesse. Unabhängig von der Anzahl der gesetzten Checkpoints bleibt jedoch die Gefahr des Dominoeffektes welcher in Abschnitt 2.2.2.3 vorgestellt wird bestehen. Im Fehlerfall werden über einen Dependency-Request, der zu jedem Prozess gesendet wird, zuerst alle Abhängigkeiten zwischen den inkonsistenten und allen anderen Prozessen überprüft. Dabei sichern zuerst alle Prozesse ihre aktuellen Zustände und warten auf die vom Rollback-Initiator berechnete Recovery-Line. Die Abhängigkeiten zwischen den Prozessen werden dabei vom Rollback-Initiator berücksichtigt. Dieser gibt anschließend den letzten konsistenten Systemzustand (Recovery-Line) zurück. Diese Recovery-Line wird in Form einer Rollback-Request Nachricht den beteiligten Prozessen mitgeteilt. Danach kann der jeweilige Prozess seine Bearbeitung an einem früheren Checkpoint fortsetzen. Während des Recovery-Vorganges dürfen keine Fehler auftreten. Weitere Nachteile bestehen in der hohen Anzahl der gesetzten Checkpoints. Das vermehrte Aufkommen der Checkpoints kann durch die in Abschnitt 2.2.2.5 vorgestellte Garbage Collection vermieden werden.

2.2.2.3 Der Domino-Effekt

Der Domino-Effekt [Klü07] tritt dann auf, wenn inkonsistente Checkpoints es nicht erlauben, das System nach einem Rollback-Vorgang in einen konsistenten Zustand zu versetzen. Dies ist z.B. dann der Fall, wenn ein Prozess im Zuge eines Rollbacks das Senden einer Nachricht rückgängig macht, dieser Vorgang jedoch zu einem inkonsistenten Zustand des Gesamtsystems führt und der Empfängerprozess mit einem Rollback dem Initiator folgen müsste. Dabei kann diese Abfolge von Abhängigkeiten der Nachrichten und der lokalen Zustände der Prozesse auch bei weiteren Prozessen in der verteilten Anwendung auftreten und im schlimmsten Fall das Gesamtsystem wieder in den Anfangszustand der Anwendung zurücksetzen, da dieser den einzigen erreichbaren konsistenten Zustand darstellt. Dieses Phänomen ist die Folge des unabhängigen Setzens der Checkpoints aller beteiligten Prozesse.

2.2.2.4 Koordiniertes Checkpointing

Bei dieser Methode können die Prozesse keine selbstständigen Checkpoints setzen. Das Checkpointing wird stattdessen global koordiniert. Diese Methode vermeidet dadurch den unerwünschten Dominoeffekt. Außerdem wird hierbei die Berechnung der in Abschnitt 2.2.2.5 vorgestellten Garbage Collection [Klü07] vermieden, da die Prozesse immer von ihrem letzten Checkpoint aus gestartet werden. Dies führt zu einem geringeren Speicherplatzverbrauch und vereinfacht die Berechnung der Recovery-Line. Aufgrund der beim koordienerten Checkpointing notwendigen Koordination der Prozesse, wird der Aufwand zum Erstellen der Checkpoints erhöht.

2.2.2.5 Garbage-Collection

Sowohl beim sequentiellen als auch beim inkrementellen Checkpointing werden im Verlauf der Anwendung Checkpoints erzeugt. Einige dieser Checkpoints können sich mit der Zeit für die konsistente Wiederherstellung der Anwendung als nutzlos erweisen, weil sie veraltet sind. Desweiteren stellt die Checkpointgröße mit der Zeit ein Speicherplatzproblem dar. Deswegen müssen unnötig gesetzte Checkpoints mittels der Garbage-Collection entfernt werden. Dabei berechnet ein Garbage Collection Algorithmus zuerst die Recovery-Line und löscht anschließend alle Checkpoints, die sich vor dieser Recovery-Line befinden. Diese werden also auch nach einem Fehler nicht mehr benutzt.

2.2.3 Log-basierte Rollback-Recovery

Bei der log-basierten Rollback-Recovery [Klü07] werden Checkpoints und Protokolle benutzt, um den Prozessen die Wiederherstellung eines konsistenten Zustandes nach einem Fehler zu ermöglichen. Dies ist insbesondere wichtig, wenn Wartezeiten vermieden werden sollen. Das ist z.B. dann der Fall, wenn ein Prozess mit der Außenwelt interagiert. Außerdem ist dieses Verfahren nicht für den Dominoeffekt anfällig und erlaubt es den Prozessen zusätzlich selbstständige Checkpoints zu erstellen. Log-basiertes Rollback-Recovery funktioniert nach dem stückweisen, deterministischen Ausführungsmodell (PWD). Es werden somit Prozessabschnitte in eine Sequenz von deterministi-

schen Zustandsintervallen aufgeteilt. Eine empfangene Nachricht oder auch ein internes Ereignis trennen die einzelnen Abschnitte und stellen ein nichtdeterministisches Ereignis dar. Diese Ereignisse werden zusätzlich zu den Checkpoints im Speicher gesichert. Im Fehlerfall wird der Prozess in seinen früheren, fehlerfreien Zustand zurückgesetzt. Bei diesem Verfahren wird im Gegensatz zu Checkpoint Systemen, versucht die gleiche Ausführung wie sie auch vor dem Fehler stattfand, wiederherzustellen.

2.3 Checkpointing in Kerrighed

2.3.1 Allgemein

Das Forschungsprojekt Kerrighed beinhaltet von Haus aus einen koordinierten sequentiellen Single-Prozess-Checkpointer, der an der HHU entwickelt wurde. Dieser Checkpointer bietet derzeit eine Checkpoint/Restart Möglichkeit der Prozesse mit einem Thread an. Außerdem können Prozesse mit einer Vater-Sohn Beziehung gesichert und anschließend wiederhergestellt werden. Wie bereits in Abschnitt 2.2.1 erwähnt, müssen für einen erfolgreichen konsistenten Checkpoint einige Resourcen gesichert werden. Die Sicherung folgender Resourcen [MS07] wird derzeit vom Kerrighed Checkpointer unterstüzt:

Prozess memory	Der Prozess-Adressraum.
PID	Jede Anwendung wird mit einer eindeutigen PID identifiziert.
SYSV IPC shared memory segments	Es ist möglich SYSV IPC shared memory zu sichern und wiederherzustellen.
Anstehende Signale	Signale [Wol04], die noch nicht ausgeliefert wurden, können gesichert und wiederhergestellt werden.
Signal-Handler	Die callback Funktionen, die im Falle eines Signals ausgeführt werden sollen, können mittels der sighand_struct gesichert und wiederhergestellt werden.

Tabelle 2.1: Zustandsinformationen

Sollte ein Prozess offene Dateien haben, werden lediglich die Deskriptoren gesichert, nicht der Inhalt. Die Sicherung der entsprechenden Inhalte soll mit dem, in Abschnitt

2.1 erwähnten KDFS in Zukunft unterstützt werden. Die derzeitige Implementierung unterstützt bereits jetzt die Sicherung von SYSV IPC Shared-Memory-Segmenten. Mit der zukünftigen Einführung der *Kerrighed Dynamic Stream Facility* sollen auch Sicherungen von Pipes und Sockets ermöglicht werden.

2.3.2 Ghost-Mechanismus

Der Ghost-Mechanismus [VLM+05] ist ein zentraler Mechanismus, der in Kerrighed zum Übertragen der Daten zwischen den Knoten in einem Netzwerk eingesetzt wird. Der primäre Einsatz dieses Mechanismus findet bei der Prozessmigration statt. Dabei wird ein Abbild des zu migrierenden Prozesses in einen Network-Ghost kopiert und über das Netzwerk verschickt. Auf der anderen Seite wird ein neuer Network-Ghost eingerichtet und mit den Daten des sendenden Network-Ghosts initialisiert. Dieser Ghost wird zur anschließenden Wiederherstellung des Prozesses gebraucht.
Der Ghost-Mechanismus wird in Kerrighed als Basis für Checkpointing und Restart eingesetzt. Dabei wird anstelle der Übertragung des Images über das Netzwerk ein File-Ghost eingerichtet, mit dem Inhalt des Images und einigen weiteren relevanten Informationen initialisiert und mittels der vom Ghost-Mechanismus zur Verfügung gestellten Funktion *ghost_write()* auf die Festplatte geschrieben. Dieses Image wird im Falle eines Restarts aus dem entsprechend dafür erstellten File-Ghost gelesen.
Wie bereits erwähnt, stellt der Ghost-Mechanismus einige Funktionen für den Programmierer bereit. Diese umfassen neben den gewöhnlichen Schreib-/Lese-Operationen auch Funktionen zum Erstellen, Schließen und Freigeben eines Ghostes. Diese Funktionen sind bei Kerrighed in der Datei *modules/ghost/ghost.h* definiert:

```
1 ghost_t *create_ghost ( ghost_type_t type, int access );
2 int free_ghost ( ghost_t *ghost ) ;
3 static inline int ghost_write ( ghost_t *ghost, const void
     *buff, size_t length );
4 static inline int ghost_read ( ghost_t *ghost, void *buff,
     size_t length );
5 static inline int ghost_close (ghost_t * ghost);
```

Es ist dabei anzumerken, dass die Schreib/Lese - Operationen des Ghost Moduls keine Möglichkeit der Angabe eines Offsets anbieten. Somit ist per Default nur ein sequentieller Zugriff auf einen Ghost möglich.

2.3.3 Strukturen des Kerrighed Checkpointers

Der in Kerrighed vorhandene Checkpointer sichert mit einem *checkpoint PID* Aufruf eine komplette Anwendung. Die dabei gesicherten Inhalte werden in einem neu erstellten Ordner */var/chkptn/PID/* festgehalten. Nach einem erfolgreichen Checkpoint werden in dem entsprechenden Ordner folgende Dateien mittels des Ghost-Mechanismus angelegt und mit einer Versionsnummer *(Vx)* versehen:

description_Vx.txt (ASCII Datei)	Beinhaltet neben der Angabe des Datums und Versionsnummer des Checkpoints eine optionale Beschreibung.
global_Vx.bin (Binäre Datei)	Enthält die Application ID, Versionsnummer und die Knotenmaske.
node_0_Vx.bin (Binäre Datei)	Beschreibung der in einen Checkpoint Prozess involvierten, lokalen Aufgaben.
task_pid_Vx.bin (Binäre Datei)	Beinhaltet die zur PID zugehörigen Kernelstrukturen wie z.B. Register, Stack, Signal Masken.
task_mm_pid_Vx.bin (Binäre Datei)	Enthält die virtuellen Adressen, Page Protection Flags und alle Seiten des Prozessadressraumes.

Tabelle 2.2: Dateien des Kerrighed Checkpointers

In dieser Arbeit soll der bestehende sequentielle Checkpointer erweitert werden, um inkrementelle Sicherungen zu erstellen. Die Datei *task_mm_pid_Vx.bin* ist mit ihren virtuellen Adressen und den entsprechenden physikalischen Seiten dabei besonders wichtig. Diese Seiten werden oft verändert und müssen bei Bedarf gesichert werden. Der Inhalt der restlichen Dateien bleibt nahezu konstant und kann vernachlässigt werden. Bei jedem Checkpoint werden in der Datei *task_mm_pid_Vx.bin* alle virtuellen Adressen, Page Protection Flags und die physikalischen Seiten eines Prozesses gespeichert. Dabei besteht jeder Eintrag der Datei aus drei Elementen. Diese sind die virtuelle Adresse, Page Protection Flags und die physikalische Seite selbst. Zum Schluss enthält die Datei eine 0 zum Kennzeichnen des Endes der Datei und die Anzahl der exportierten Seiten.

Dieser Zusammenhang soll anhand der folgenden Beispieltabelle verdeutlicht werden:

Virtuelle Adresse	Page Protection Flags	Seite
0x100	Page Protection Flags	Seite 0
...	...	...
0xXYZ	Page Protection Flags	Seite *N*
0	Anzahl der Seiten ->	(*N+1*)

Tabelle 2.3: Aufbau der task_mm_pid_Vx.bin Datei

Kapitel 3

Implementierung des inkrementellen Checkpointings in Kerrighed

3.1 Übersicht

An dieser Stelle soll die Implementierung des inkrementellen Checkpointings in Kerrighed vorgestellt werden. Dazu wird zuerst die Repräsentation eines Linux Prozesses auf der Kernel-Ebene beschrieben. Dies ist für das Verständnis des weiteren Verlaufs des Kapitels wichtig, da die Seiten eines Prozesses sowohl bei der sequentiellen als auch bei der inkrementellen Sicherung eingesammelt werden müssen. Dann wird die in dieser Arbeit konzipierte Struktur zur Verwaltung der veränderten, physikalischen Seiten beschrieben. Diese Verwaltungsstruktur ist beim inkrementellen Checkpointing für die Lokalisierung der physikalischen Seiteninhalte notwendig. Es folgt die Beschreibung des Mechanismus zum Abfangen der Schreibzugriffe und zum Einsammeln der Seiten auf der Kernel-Ebene. Zum Schluss wird der Restart eines Prozesses mit Hilfe der Verwaltungsstruktur vorgestellt.

3.2 Repräsentation eines Prozesses auf der Kernel-Ebene

Im Zuge der Prozessverwaltung [Rus99] auf der Kernel-Ebene, wird jeder Prozess durch einen Prozessdeskriptor vom Typ *struct task_struct* repräsentiert. Diese Prozessdeskriptoren werden vom System in dem Taskvektor festgehalten. Dies bedeutet, dass die maximale Anzahl der Prozesse in einem System durch die Größe des Taskvektors beschränkt ist. Dieser beinhaltet per Default 512 Einträge. Sobald ein Prozess erstellt wird, erzeugt das System einen neue Prozessdeskriptor und fügt diesen dem Taskvektor hinzu. Da der Prozessdeskriptor viele Einträge beinhaltet, die in dem Kontext dieser Arbeit nicht relevant sind, beschränkt sich diese Arbeit auf das Speicherdeskriptor Feld (*struct mm_struct*), welches die virtuelle Adressstruktur eines Prozesses darstellt. Die meisten Prozesse besitzen virtuellen Speicher (außer Kernel-Threads und Daemons). Der Kernel muss sich somit um die Verwaltung dieses Speichers kümmern. Dies wird dadurch erreicht, dass in jedem Prozessdeskriptor ein Zeiger auf einen Speicherdeskriptor mitgeführt wird. Der Speicherdeskriptor wiederum beinhaltet einen Zeiger (mmap) auf die erste Virtual Memory Area (VMA) [Gor04] vom Typ *struct vm_area_struct*. Da der virtuelle Adressraum eines Prozesses in dem Kernel neben einer doppelt verketteten Liste von VMAs durch einen Adelson-Velskii-Landis Baum (AVL) [CLR01] repräsentiert wird, beinhaltet der Eintrag (mmap_avl) einen Zeiger auf die VMA-Wurzel des Baumes. Jede VMA besitzt ein *vm_start* und ein *vm_end* Adressfeld [RC02]. Diese zwei virtuellen Adressen geben den Start- und Endbereich einer VMA an. Neben einigen weiteren Feldern wie dem *vm_flags* Feld, gibt es einen *vm_next* Eintrag. Dieser Eintrag beinhaltet einen Zeiger auf die nächste VMA.

Dieser Zusammenhang soll anhand der Abbildung 3.1 verdeutlicht werden:

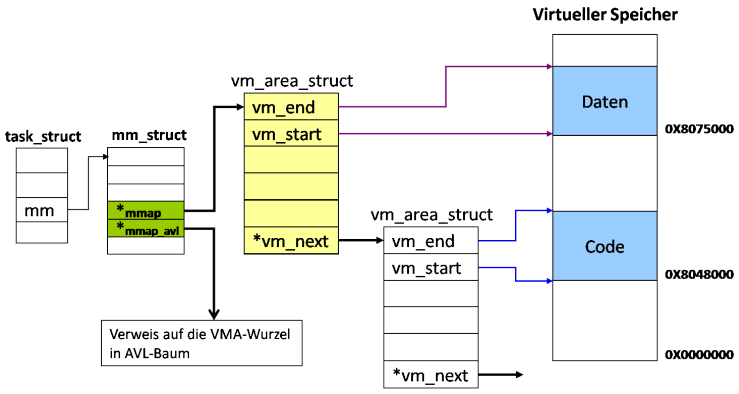

Abbildung 3.1: Linux Prozess-Struktur

Jede virtuelle Adresse wird im Kernel durch eine dreistufige Page Table [Int08] auf eine physikalische Adresse abgebildet. AMD64-Architektur verwendet dagegen eine vierstufige Abbildung. In beiden Fällen werden virtuelle Adressen in mehrere Felder unterteilt. Jedes Feld gibt einen Offset für seine jeweilige Page Table an. Um eine virtuelle Adresse in eine physikalische Adresse zu übersetzen nimmt der Prozessor den Inhalt jedes Feldes, konvertiert den Inhalt in einen Offset innerhalb der Page Table und liest daraus die Page Frame Number (PFN) [BC05] der nächsten Page Table aus. Dieser Vorgang wird dreimal wiederholt bis die endgültige PFN der physikalischen Seite gefunden wurde. Anschließend wird das letzte Feld (Byte Offset) verwendet, um die Daten innerhalb der physikalischen Seite ausfindig zu machen. Dieser Vorgang soll anhand der Abbildung 3.2 verdeutlicht werden.

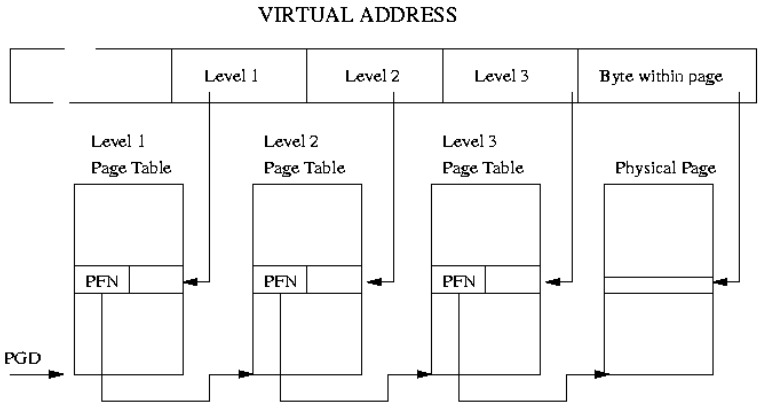

Abbildung 3.2: Linux 3-stufige Page Table

Die Einträge einer Page Table heißen Page Table Entries (PTEs). Diese beinhalten neben einigen Zugriffsschutzbits, die bereits erwähnte PFN der nächsten Page Table. Das Dirty-Bit (D) wird im Falle eines Schreibzugriffs auf eine Seite vom Prozessor gesetzt. Damit wird der Speicherverwaltung signalisiert, dass die entsprechende Seite ohne zusätzliche Maßnahmen nicht aus dem Speicher entfern werden darf. Das Accessed-Bit (A) wird im Falle eines beliebigen Zugriffs auf die Seite vom Prozessor gesetzt. Mit dem Present-Bit (P) wird angegeben ob die Seite derzeit im Speicher eingelagert wird. Mit dem Read/Write-Bit (R/W) wird festgelegt, ob ein Schreibzugriff auf die entsprechende Seite möglich ist. Das User/Supervisor-Bit (U/S) ermöglicht die Realisierung des Speicherschutzes. Seiten, bei denen dieses Bit gesetzt ist, können nur von Programmen aus dem Ring 0-2 genutzt werden. Außerdem existieren einige frei zur Verfügung stehende Bits. Die Abbildung 3.3 zeigt eine PTE:

Abbildung 3.3: Linux Page Table Entry

3.3 Entwurf einer Verwaltungsstruktur für modifizierte Seiten

Wie in Kapitel 2 bereits erwähnt, erstellt der bestehende Kerrighed Checkpointer für jede Sicherung mehrere Dateien. In der *task_mm_pid_Vx.bin* Datei befinden sich neben den virtuellen Adressen und den Page Protection Flags, die Inhalte der physikalischen Seiten. Für jeden *checkpoint* Aufruf erstellt der Kerrighed Checkpointer eine solche Datei, sichert alle Seiten der Anwendung und zählt die Versionsnummer aller zu einem Checkpoint gehörenden Dateien hoch, um zwischen den Sicherungen unterscheiden zu können. Dieser Checkpointer soll, wie in Kapitel 1 erwähnt, um die Möglichkeit erweitert werden inkrementelle Sicherungen zu erstellen. Dafür müssen nicht nur die Änderungen der Inhalte der physikalischen Seiten seit der letzten Sicherung erkannt werden. Es muss auch eine Möglichkeit geschaffen werden diese Änderungen zu verwalten, um einen Restart der Anwendung zu ermöglichen.
Dazu wurde eine Verwaltungsstruktur entwickelt, die die Abbildung zwischen den virtuellen Adressen und der jeweils aktuellen Version der Dateien, in der die physikali-

sche Seite gespeichert ist, enthält. Dadurch kann zu jeder virtuellen Adresse die aktuelle Version der physikalischen Seite eindeutig ermittelt werden. Dafür enthält die Verwaltungsstruktur zu jeder virtuellen Adresse einen Offset und eine Versionsnummer. Mit der Versionsnummer wird bestimmt, in welcher Datei (*task_mm_pid_Vx.bin*) sich der physikalische Inhalt einer Seite zu einer virtuellen Adresse befindet. Der Offset bezeichnet die Inhaltsposition einer physikalischen Seite innerhalb der Datei. Dadurch kann eindeutig bestimmt werden in welcher Datei sich der Inhalt jeder gesicherten, physikalischen Seite befindet und an welcher Stelle dieser Inhalt innerhalb der Datei liegt.
Aufgrund der vielen möglichen Szenarien, die zur Laufzeit jeder Anwendung auftreten können, wurde eine weitere Struktur eingeführt. Diese Struktur *misc_t* beinhaltet einige wichtige Informationen, die es ermöglichen auf einige Szenarien zu reagieren. Sollte sich z.B. die Anzahl der VMAs aufgrund von hinzugekommenen Shared Memory-Segmenten oder zusätzlicher Speicherallokation zur Laufzeit ändern, so muss die VMA entweder in die Verwaltungsstruktur eingefügt oder ein kompletter Checkpoint erstellt werden. Das gleiche gilt auch für zusätzliche Adressen, die zu einer bereits bestehenden VMA (z.B. durch Speicherallokation) hinzukommen können. In dieser Arbeit wird im Falle einer Änderung der Anzahl von VMAs oder der Adressen eine komplette Sicherung erstellt. Die *misc_t* Struktur wird zusätzlich in der Verwaltungsstruktur mitgeführt. Der folgende Ausschnitt zeigt den Aufbau der *misc_t* Struktur:

```
1 typedef struct {
2         long vma_count;
3         long address_count;
4         long app_id;
5 }misc_t;
```

Der Eintrag *vma_count* gibt die Anzahl der VMAs [BC05] des Prozesses an. In dem *address_count* Feld ist die Anzahl der virtuellen Adressen gespeichert. Die *app_id* gibt die Application ID des Prozesses an. Dies ist sowohl für Prozesse mit einer Vater-Sohn Beziehung als auch für unabhängige Prozesse wichtig und wird beim Restart solcher Anwendungen verwendet. Die endgültige Verwaltungsstruktur, wie sie auf der Festplatte abgelegt wird, soll anhand der nachfolgenden zwei Tabellen verdeutlicht werden.

Vma_Count	Address_Count	App_ID
4	20	0
Offset	Virtuelle Adresse	Versionsnummer
0	0x100	1
1	0x200	1
2	0x300	1
3	0x400	1
4	0x500	1
...	...	1
19	0xXYZ	1

Tabelle 3.1: Verwaltungsstruktur, initialer, inkrementeller Checkpoint

Die erste Tabelle 3.1 beschreibt dabei den initialen, inkrementellen Checkpoint. In dieser ersten Sicherung werden alle Seiten der Anwendung gesichert. Somit enthält die erste Verwaltungsstruktur alle gültigen, virtuellen Adressen mit dem entsprechenden Verweis auf die Versionsnummer der Datei (*task_mm_pid_v1.bin*) in der sich die zugehörigen physikalischen Seiteninhalte befinden. Eine virtuelle Adresse ist dabei erst dann gültig, wenn sie eine physikalische Seite abbildet. Der Offset wird bei jedem Checkpoint Aufruf mit einer 0 initialisiert und mit jeder gültigen, virtuellen Adresse hochgezählt. Durch den Offset und die Versionsnummer wird die Abbildung der virtuellen Adressen auf deren physikalischen Seiteninhalte in den entsprechenden Dateien gewährleistet. Zusätzlich hält sie in der *misc_t* Struktur die Informationen über die Anzahl der VMAs, aller gültigen, virtuellen Adressen und die Application ID bereit. Die Verwaltungsstruktur wird bei jeder Sicherung angelegt und auf der Festplatte unter einem Namen (*control_pid_v1.bin*) gespeichert. Die Versionsnummer ist 1, da es sich hierbei um die erste Sicherung handelt.

Vma_Coount	Address_Count	App_ID
4	20	0
Offset	Virtuelle Adresse	Versionsnummer
0	0x100	1
1	0x200	1
2	0x300	1
0	0x400	2
1	0x500	2
...	...	1
19	0xXYZ	1

Tabelle 3.2: Verwaltungsstruktur, inkrementeller Checkpoint

Anhand der zweiten Tabelle 3.2 soll der Inhalt der nächsten, inkrementellen Verwaltungsstruktur dargestellt werden. Dabei wird immer die vorherige Verwaltungsstruktur (in diesem Fall die *control_pid_v1.bin*) geladen, an den entsprechenden Stellen (Offset, Versionsnummer) aktualisiert und unter einer neuen Versionsnummer (hier v2) auf der Festplatte gespeichert. Dadurch können die physikalischen Seiten einer Anwendung zu einem beliebigen Sicherungszeitpunkt lokalisiert werden. Das Lokalisieren eines Seitendatensatzes ist ausschließlich beim Restart einer Anwendung wichtig, da sich die physikalischen Seiten der Anwendung in mehreren unterschiedlichen Dateien befinden können. Bei dieser inkrementellen Sicherung wurde die Anzahl der VMAs und die Anzahl der gültigen, virtuellen Adressen nicht verändert. Der einzige Unterschied zu der Tabelle 3.1 besteht in der Änderung des physikalischen Seiteninhalts an den beiden virtuellen Adressen *0x400* und *0x500*. Diese Änderungen sind unmittelbar mit der Aktualisierung des Offsets und der Versionsnummer verbunden. Die virtuelle Adresse *0x400* enthält somit den Offset 0, da sich der Inhalt, der zu dieser Adresse gehörenden physikalische Seite an der ersten Stelle innerhalb der Datei (*task_mm_pid_v2.bin*) befindet. Die Versionsnummer ist dabei 2, da es sich bei dieser Sicherung um die zweite inkrementelle Sicherung handelt. Mit Hilfe der Application ID kann ermittelt werden, ob es sich bei der oben genannten Sicherung um die Sicherung desselben Prozesses handelt wie zuvor.

Die in dieser Arbeit konzipierte Verwaltungsstruktur wurde unter Verwendung eines Red-Black-Trees (RBTs) [CLR01] realisiert, bei dem der Schlüssel jedes Knotens eine virtuelle Adresse darstellt. Ein RBT ist ein binärer Suchbaum, welcher einen sehr schnellen Zugriff auf die in ihm gespeicherten Elemente garantiert. Dabei werden die schnellen Zugriffszeiten durch spezielle Eigenschaften erreicht, welche zusammen ga-

rantieren, dass der Baum immer annähernd balanciert bleibt. Dadurch wird die Höhe des RBTs mit n Werten nie größer als $\log_2 n$. Somit können die wichtigen Operationen: *suchen*, *einfügen*, *löschen* in $O(\log n)$ ausgeführt werden. Jeder Knoten in einem RBT beinhaltet außerdem eine Zusatzinformation, die seine Farbe trägt. Die Abbildung 3.4 zeigt einen möglichen RBT mit acht Schlüsseln:

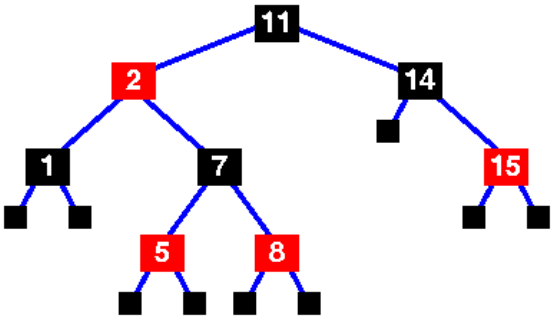

Abbildung 3.4: Red-Black-Tree

Auf nähere theoretische Details zu dem RBT wird in dieser Arbeit nicht weiter eingegangen.
Im Kernel existiert bereits eine Implementierung des RBTs. Sie befindet sich in der Datei <*linux/rbtree.h*> und wird bei der Speicherverwaltung eingesetzt. Knoten, die sich im RBT befinden sollen, müssen die Struktur *rb_node* beinhalten. Da es keine Funktionen zum Suchen und Einfügen der Elemente gibt, wurden diese unter Benutzung der vorgegeben Operationen (*rb_first, rb_last, rb_next, rb_prev*) implementiert:

```
1 page_entry_t *search_rb_tree(struct rb_root *root, unsigned
      long addr);
2 void insert_rb_tree(struct rb_root *root, page_entry_t *
    new_element);
```

Mit der Funktion *search_rb_tree()* ist es möglich, einen Knoten mit einer bestimmten virtuellen Adresse, innerhalb des Baumes zu suchen. Als Rückgabewert liefert die Funktion einen Zeiger auf eine *page_entry_t* Struktur. Diese Struktur repräsentiert einen Knoten des Baumes. Mit der Funktion *insert_rb_tree()* ist es möglich, einen neuen Knoten in den Baum einzufügen. Dafür wird der Funktion *insert_rb_tree()* der Wurzelknoten und der neue Knoten übergeben.
Wie bereits erwähnt, stellt jeder Schlüssel eines Knotens eine virtuelle Adresse dar. Neben dieser Information beinhaltet jeder Knoten unter anderem einen Offset, Versions-

nummer und eine Struktur *rb_node*, die den Knoten im Baum repräsentiert. Der nachfolgende Ausschnitt zeigt die *page_entry_t* Struktur:

```
1 typedef struct {
2         loff_t offset;
3         unsigned long addr;
4         int version;
5         struct rb_node node;
6 }page_entry_t;
```

Beim initialen Checkpoint wird der RBT mit Hilfe folgender Funktion aufgebaut. Diese Funktion erstellt einen Knoten und fügt diesen in den Baum ein.

```
1 void create_page_entry_and_add_to_tree(loff_t *offset,
     unsigned long addr, int version,struct rb_root *
     rbtree_root);
```

Sollte es sich bei dem Checkpoint um einen inkrementellen Checkpoint handeln, so muss der RBT lediglich aktualisiert werden. Das heißt, es muss ermittelt werden ob der übergebenen, virtuellen Adresse ein Knoten zugeordnet ist. Sollte es einen Knoten geben wird ein neuer Knoten mit den vorgegeben Informationen (virtuelle Adresse, Offset, Versionsnummer) erzeugt und der alte Knoten damit ersetzt. Dazu gibt es eine weitere Funktion, die einen neuen Knoten erzeugt, den Baum durchsucht und den alten Knoten ersetzt. Sie hat dieselben Parameter wie die vorherige Funktion zum Erstellen und zum Einfügen eines Knotens in den Baum.

```
1 void create_page_entry_and_update_tree(loff_t *offset,
     unsigned long addr, int version,struct rb_root *
     rbtree_root);
```

Nachdem der RBT erfolgreich aufgebaut bzw. aktualisiert wurde, müssen dessen Inhalte auf der Festplatte in einer Verwaltungsstruktur gespeichert werden, um einen späteren Restart der Anwendung zu ermöglichen. Dafür wird mit Hilfe des Ghost-Mechanismus für jeden inkrementellen Checkpoint ein neuer File-Ghost erstellt, dessen Versionsnummer bei jeder neuen Sicherung hochgezählt wird. Dieser File-Ghost bekommt, wie auch die Datei, in der die Inhalte der physikalischen Seiten gespeichert sind, einen Namen *(control_pid_Vx.bin)*. Anschließend wird zuerst die *misc_t* Struktur und dann alle Knoten des Baumes in den File-Ghost geschrieben.

```
1 ghost_write(control_ghost,&misc,sizeof(misc_t));
2 void write_tree_to_control_ghost(struct rb_root *
    rbtree_root,ghost_t *control_ghost);
```

Wie die modifizierten physikalischen Seiten erkannt werden und wie die hier beschriebene Verwaltungsstruktur entsprechend aktualisiert wird, ist ein Bestandteil der nachfolgenden Abschnitte.

3.4 Abfangen von Schreibzugriffen und Einsammeln der Seiten auf der Kernel-Ebene

Da beim inkrementellen Checkpointing nur die Seiten gesichert werden, die seit der letzten Sicherung verändert wurden, müssen diese Änderungen zuerst erkannt werden. Zum Ermitteln der Änderungen wird bei jedem inkrementellen Checkpoint für jedes gültige PTE das Write-Bit zurückgesetzt. Wie eine PTE aufgebaut ist, wurde bereits in dem Abschnitt 3.2 erläutert. Ein Zurücksetzen dieses Bits bewirkt bei einem anschließenden Schreibzugriff auf diese Seite das Auslösen einer Page Fault Exception. Behandelt wird sie durch den Page Fault Exception Handler des Kernels. Dieser prüft nach, ob es sich bei dieser Seite um eine Copy-On-Write (COW) [BC05] Seite handelt. Die Information darüber, wie viele Prozesse sich eine Seite teilen, wird in der *count* Variable des Page Descriptors [Tan02] festgehalten. Im Falle, dass eine Seite vom Prozess freigeben wird, wird diese Variable dekrementiert bis der Wert -1 erreicht ist. Erst dann gilt die Seite nicht mehr als belegt. Sollte diese Variable den Wert 0 haben (sie wird also nur von einem Prozess verwendet), so wird das Write-Bit gesetzt, um eine Page Fault Exeption zu vermeiden. Sollte es sich bei dieser Seite um eine COW-Seite handeln, wird eine Kopie der Seite erzeugt und im Speicher eingelagert. Anschließend wird das Write-Bit für diese Seite gesetzt. Die andere Seite (das Original) bleibt dabei schreibgeschützt. Dies ist insbesondere für die Prozesse mit einer Vater-Sohn Beziehung wichtig. Bei einem anschließenden inkrementellen Checkpoint kann das Write-Bit jedes PTE abgefragt werden. Dies geschieht mit dem Makro *pte_write()*. Ein gesetztes Write-Bit deutet auf eine Seitenmodifizierung hin. In diesem Falle kann die Seite gesichert werden.
Der oben gennante Vorgang setzt voraus, dass für die aktuelle Sicherung eine neue Verwaltungsstruktur (*control_pid_Vx.bin*) angelegt und falls vorhanden, die vorherige

Verwaltungsstruktur (*control_pid_V(x-1).bin*) geladen wurde. Das Laden der vorherigen Verwaltungsstruktur ist, wie bereits in Abschnitt 3.3 erwähnt, bei jeder inkrementelle Sicherung notwendig, da diese Verwaltungsstruktur zum Auffinden der veränderten Seiten verwendet wird. Bei einem inkrementellen Checkpoint-Aufruf werden die Inhaltsänderungen der physikalischen Seiten seit dem letzten inkrementellen Checkpoint erfasst und die geladene Verwaltungsstruktur aktualisiert in einer neuen Datei auf der Festplatte abgespeichert.
Dieser Vorgang ist für den erfolgreichen Restart der Anwendung notwendig. Da sich die Anzahl der VMAs und der entsprechenden virtuellen Adressen ändern könnte, müssen vor dem Einsammeln der physikalischen Seiten eines Prozesses diese beiden Werte ermittelt werden. Dafür wird zuerst eine Struktur vom Typ *misc_t* angelegt und mit den Informationen über die Anzahl der VMAs und der gültigen, virtuellen Adressen gefüllt. Anschließend wird die neu angelegte Struktur in die für die aktuelle Sicherung mittels des Ghost-Mechanismus angelegte Verwaltungsstruktur *(control_pid_Vx.bin)* geschrieben. Der nachfolgende Quellcodeabschnitt soll dies vedeutlichen:

```
1 misc_t get_vma_and_address_count_from_vm(struct
      vm_area_struct *vma);
2 ghost_write(control_ghost,&misc,sizeof(misc_t));
```

Die beiden Werte für die Anzahl der VMAs und der Adressen werden mit den Werten aus der vorherigen Verwaltungsstruktur verglichen. Sollte einer der beiden Werte verändert worden sein, wird ein kompletter Checkpoint erstellt. Die Gründe für das Erstellen eines kompletten Checkpoints wurden bereits in Abschnitt 3.3 erläutert. Falls sich die Anzahl der VMAs oder der gültigen, virtuellen Adressen nicht ändert, wird aus der zuvor geladenen Verwaltungsstruktur *(control_pid_V(x-1).bin)* ein RBT erstellt und mit der inkrementellen Sicherung fortgefahren.
Beim initialen, inkrementellen Checkpoint wird für jede gültige virtuelle Adresse der zugehörige PTE aus der Liste der VMAs ermittelt. Das *pte_present()* Makro prüft für jede Seite, ob sie sich derzeit im physikalischen Speicher befindet. Sollte dies der Fall sein, kann sie mittels des Ghost-Mechanismus auf die Festplatte gesichert werden. Dabei wird, falls es sich um die initiale, inkrementelle Sicherung handelt, ein RBT Knoten erstellt mit den Informationen (virtuelle Adresse, Offset, Versionsnummer) gefüllt und in den zuvor erstellten RBT eingefügt. Zusätzlich wird in jedem Knoten, wie in Abschnitt 3.3 erwähnt, die virtuelle Adresse und die Versionsnummer des Checkpoints festgehalten.

Bei der nächsten inkrementellen Sicherung wird zusätzlich, mittels des *pte_write()* Makros, das Write-Bit der Seite abgefragt. Befindet sich die Seite derzeit im Speicher und ist ihr Write-Bit gesetzt, so wurde diese Seite modifiziert und muss gesichert werden. In diesem Fall muss der aus der vorherigen Verwaltungsstruktur generierte RBT an der entsprechenden Stelle aktualisiert werden. Diese Aktualisierung wird durch die bereits in Abschnitt 3.3 erwähnte Funktion *create_page_entry_and_update_tree()* erreicht.
Schließlich wird sowohl bei der initialen als auch bei der nächsten inkrementellen Sicherung die PFN aus dem bereits geholten PTE extrahiert und anhand dieser Nummer die zugehörige, physikalische Seite ermittelt. Das *kmap()* Makro gibt anschliessend die physikalische Adresse der ermittelten Seite zurück. Zusätzlich zu dem Inhalt der 4KB Seite werden schließlich mittels des Ghost-Mechanismus die zugehörige virtuelle Adresse und die zugehörigen Page Protection Flags in den entsprechenden File-Ghost *(task_mm_pid_Vx.bin)* gesichert. Anschließend wird das Write-Bit des PTEs mit Hilfe des Kernel Makros *set_pte()* zurückgesetzt. Ein Zurücksetzten des *Write*-Bits, wird wie in diesem Abschnitt bereits erläutert, zum anschließenden Erkennen der modifizierten Seiten eingesetzt.

```
1          ghost_write (ghost, &addr, sizeof (unsigned long));
2          /* Export page protection */
3          ghost_write (ghost, &prot, sizeof(pgprot_t));
4          /* Export physical page content */
5          ghost_write (ghost, (void*)page_addr, PAGE_SIZE);
6          /* reset write bit */
7          set_pte(pte,pte_wrprotect(*pte));
```

Anschließend wird der neue bzw. aktualisierte RBT in die Verwaltungsstruktur *(control_pid_Vx.bin)* geschrieben und der zuvor allozierte Speicher freigegeben.

```
1 /* write page tree entries to control ghost and close it */
2 write_tree_to_control_ghost(&rbtree_root,control_ghost);
3 /* free memory for each page entry */
4 free_tree_memory(&rbtree_root);
```

3.5 Restart eines Prozesses mit Hilfe der Verwaltungsstruktur

Für eine erfolgreiche Wiederherstellung eines Prozesses müssen nicht nur die VMAs richtig aufgebaut werden, sondern auch andere Prozessstrukturen wie Stack, Heap, Register u.v.m. Da der Aufwand für das Abspeichern dieser Strukturen sowohl beim sequentiellen als auch beim inkrementellen Checkpointing konstant ist, werden diese Strukturen vom Kerrighed Checkpointer, wie in Kapitel 2 erläutert, in separaten Dateien gesichert und im Falle eines Restarts zuerst wieder aufgebaut. Somit sind vor der Wiederherstellung der eigentlichen Seiteninhalte die restlichen Strukturen, wie der Prozessdeskriptor (*struct task_struct*), bereits aufgebaut. Das gleiche gilt für die doppelt verkettete Liste der VMAs. Anschließend folgt dann die Wiederherstellung der eigentlichen physikalischen Seiten, die von den virtuellen Adressen der VMAs abgebildet werden. An dieser Stelle setzt sowohl der sequentielle, als auch der inkrementelle Restart an. Im Falle eines sequentiellen Restarts muss die bereits wiederhergestellte Liste der VMAs durchgelaufen werden und für alle gültigen, virtuellen Adressen einer VMA der entsprechende physikalische Seiteninhalt aus der zugehörigen Datei *(task_mm_pid_Vx.bin)* geholt werden. Dazu gehört, neben der virtuellen Adresse und den Page Protection Flags, der eigentliche Inhalt der physikalischen Seite. Dieses Vorgehen wurde entsprechend für den inkrementellen Restart angepasst. Vor der eigentlichen Wiederherstellung der physikalischen Seiteninhalte wird zuerst die Verwaltungsstruktur aus der entsprechenden Datei (*control_pid_Vx.bin*) geladen. Dies ist wichtig, da sich im Falle eines inkrementellen Restarts die Inhalte der physikalischen Seiten in mehreren unterschiedlichen Dateien (*task_mm_pid_Vx.bin*) befinden und lokalisiert werden müssen. Für die Lokalisierung dieser Inhalte ist die Verwaltungsstruktur zuständig. Als erstes muss, aufgrund der sequentiellen Abläufe (*ghost_read)* des Ghost-Mechanismus, zuerst die Application ID aus der Verwaltungsstruktur gelesen werden. Diese befindet sich in der Struktur *misc_t*. Anschließend wird der RBT aufgebaut, indem die Verwaltungsstruktur komplett durchlaufen und jeder Knoten einzeln in einen leeren Baum eingefügt wird. Der nachfolgende verkürzte Quellcodeabschnitt soll dies verdeutlichen:

```
1 /* init rbtree root */
2 struct rb_root rbtree_root = RB_ROOT;
3 /* get app_id from control ghost */
4 app_id=get_app_id_from_ghost(control_ghost);
```

```
5 generate_tree_from_control_ghost(&rbtree_root,
      control_ghost);
```

Nachdem der RBT komplett aufgebaut wurde, kann mit der eigentlichen Wiederherstellung der physikalischen Seiten begonnen werden. Dafür wird die doppelt verkettete Liste der VMAs durchlaufen und zu jeder gültigen Adresse einer VMA der entsprechende physikalische Seiteninhalt aus der Datei *task_mm_pid_Vx.bin* geholt. Das Auslesen der Daten geschieht jedoch nicht mehr wie beim sequentiellen Restart direkt aus der Datei *task_mm_pid_Vx.bin*, wo sich die physikalischen Seiten befinden, sondern es wird zuerst geprüft, ob die entsprechende virtuelle Adresse in dem Baum vorhanden ist. Dazu wird die virtuelle Adresse mit den Schlüsseln der Knoten des Baumes verglichen. Sollte der Knoten mit der entsprechenden virtuellen Adresse in dem Baum sein, wird zuerst anhand der Versionsnummer und des Offsets des Knotens, ermittelt in welcher Datei und an welcher Stelle in der Datei, sich der physikalische Seiteninhalt zu der jeweiligen virtuellen Adresse befindet. Anschließend wird der physikalische Seiteninhalt aus der lokalisierten Datei ausgelesen. Dieser Prozess wird für alle VMAs wiederholt, um den kompletten Prozessadressraum wiederherzustellen. Der Vorgang soll anhand des folgenden, stark verkürzten Quellcodeabschnittes verdeutlicht werden:

```
1 for (address = vma->vm_start; address < vma->vm_end;
      address += PAGE_SIZE) {
2         /* get pgrot from address */
3         if(!get_pgprot_from_virtual_address(rbtree_root,&
              prot,address,tsk,app_id))
4                 continue;
5         /* get page from address */
6         if(!get_page_from_virtual_address(rbtree_root,
              page_addr,address,tsk,app_id)
7                 continue;
8 }
```

Der zuvor beschriebene Ablauf wurde durch folgende Funktionsaufrufe realisiert. Mit Hilfe der Funktion *get_pgprot_from_virtual_address()* werden die Page Protection Flags zu einer gültigen, virtuellen Adresse aus der entsprechenden Datei (*task_mm_pid_Vx.bin*) gelesen. Sollte die virtuelle Adresse sich nicht im Baum befinden und somit keine Page Protection Flags haben, so wird sie übersprungen.
Die zweite Funktion *get_page_from_virtual_address()* ist identisch. Sie holt zu einer

gültigen, virtuellen Adresse den eigentlichen Inhalt der physikalischen 4KB Seite. Der Vorgang des Wiederherstellens eines Prozesses gilt als abgeschlossen, sobald der komplette Inhalt aller physikalischen Seiten erfolgreich wiederhergestellt wurde.
Zuletzt wird der zuvor allozierte Speicherbereich für die Knoten des Baumes mit der Funktion *free_tree_memory()* wieder freigegeben und der File-Ghost, der die Verwaltungsstruktur enthält, geschlossen.

Kapitel 4

Leistungsbewertung und Analyse

4.1 Ausgangssituation

In diesem Kapitel wird die Performance des inkrementellen Checkpointings der Performance des sequentiellen Checkpointings gegenübergestellt. Dazu wurden zwei Opteron Knoten mit folgender Konfiguration verwendet:

Prozessor (CPU):	AMD Opteron 244 1,8GHz
Arbeitsspeicher (RAM):	2GB
Auslagerungsspeicher (Swap):	2GB
Netzwerkkarte (Nic):	Gigabit
Kerrighed Version: LinuxSSI, Revision:	3318

Tabelle 4.1: Ausgangssituation

Um das Checkpointing-Verhalten zu einem bestimmten Zeitpunkt zu messen, wurde ein Bash-Skript erstellt, das alle 10 ms einen sequentiellen bzw. inkrementellen Checkpoint Aufruf an die jeweilige Anwendung absetzt. Die Messungen für die Wiederherstellung einer Anwendung wurden sowohl nach dem Reboot, als auch ohne Reboot des Systems durchgeführt. Um das Verhalten von sequentiellem und inkrementellem Checkpointing möglichst gut zu zeigen, wurden zwei Anwendungen verwendet, die im folgenden Verlauf dieses Kapitels beschrieben werden.

4.2 Anwendungsklassen

Es gibt unterschiedliche Klassen von Anwendungen. Viele Anwendungen weisen unterschiedliche Verhaltensweisen beim Beschreiben des Speichers auf. Die rechenintensiven Anwendungen reservieren Speicher und beschreiben dessen Inhalt permanent neu. Bei solchen Anwendungen muss der Inhalt im Falle des inkrementellen Checkpointings permanent neu gesichert werden. Dadurch ergibt sich eine Laufzeit, die wesentlich höher ist als beim sequentiellen Checkpointing, weil das Erkennen der Veränderungen viel mehr Zeit in Anspruch nimmt als einen vollständigen Checkpoint zu erstellen. Anwendungen, die viel Speicher allozieren, diesen Speicher komplett beschreiben und anschließend den Inhalt nicht oft ändern, sind dagegen sehr gut für inkrementelles Checkpointing geeignet, weil hier die Anzahl der veränderten Seiten und somit die Anzahl der benötigten Vergleiche gering bleibt. Anwendungen, die den Inhalt in unterschiedlichen Abständen nahezu komplett neu beschreiben, sind nur teilweise geeignet, weil hier die Zeit für das Vergleichen der Daten zum ungünstigen Zeitpunkt sehr hoch sein kann und somit das Checkpointing um einiges verlangsamt wird. Sollten jedoch im weiteren Verlauf der Sicherungen nur wenige Änderungen erfolgen, macht das inkrementelle Checkpointing wiederum mehr Sinn, weil hier, wie vorher schon erwähnt, die Anzahl der Vergleiche gering wäre und somit auch das Checkpointing schneller gehen würde. Außerdem ist das inkrementelle Checkpointing insbesondere für die Klasse der datenintensive Anwendungen, wie z.B. Datenbanken sehr relevant, weil hier eine permanente Sicherung des kompletten Datenbestandes sehr teuer ist.

4.3 Anwendungen

4.3.1 Malloc and Write (MAW)

Bei dieser Anwendung handelt es sich um ein selbst geschriebenes Programm. Es alloziert n-MB Speicherplatz und beschreibt diesen Speicher mit Zufallswerten im Bereich 1-255. Anschließend werden in einer Endlosschleife per Zufall einige Werte innerhalb des Speicherbereichs neu beschrieben. Die Größe des zu reservierenden Speicherbereichs wurde auf 10, 20 und 30 MB festgelegt.

4.3.2 Modified Gram-Schmidt (MGS)

Der modifizierte Gram-Schmidt Algorithmus (MGS) [Sch05] faktorisiert eine Matrix A in Faktoren Q und R, wobei die Matrix Q eine orthonormale Matrix mit $Q^T Q = I$ (I ist Einheitsmatrix) und die Matrix R eine obere Dreiecksmatrix ist. Der MGS-Algorithmus ist ein grundlegender Baustein im wissenschaftlichen Rechnen. In der Numerik kann er z.B. zur Lösung von Gleichungssystemen mit iterativen Lösungsmethoden eingesetzt werden. In dieser Arbeit wird das Checkpointing eines solchen Faktorisierungsvorgangs mit unterschiedlichen Größen der Matrix A untersucht. Die Größe der Matrix A wurde mit 500x500, 1000x1000, 1500x1500 und 2000x2000 gewählt, weil bei dieser Auswahl der Speicherbedarf nicht die Grenzen des Systems überschreitet und sich die Skalierbarkeit der Anwendung gut beobachten lässt.

4.4 Messungen von sequentiellem und inkrementellem Checkpointing

4.4.1 Messungen zu der Anwendung MAW

4.4.1.1 Checkpoint

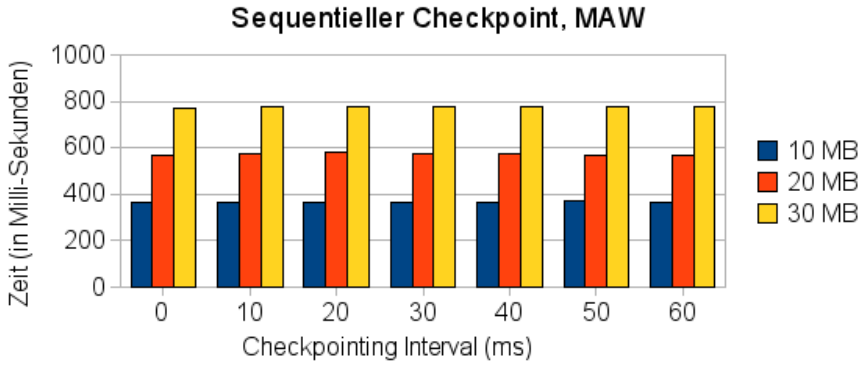

Abbildung 4.1: MallocAndWrite, Sequentieller Checkpoint

Die Abbildung 4.1 zeigt die sequentiellen Checkpointing Zeiten der Anwendung *MAW*. Auf der x-Achse ist das Sicherungsintervall eingetragen. Auf der y-Achse befindet sich

die Zeit für das Checkpointing der jeweiligen Datengrößen. Die Farben geben die Größe der zu sichernden Daten an. Beim sequentiellen Checkpointing werden alle Seiten der Anwendung gesichert. Somit unterscheidet sich die Dauer das Checkpointing zu allen Zeitpunkten nicht.

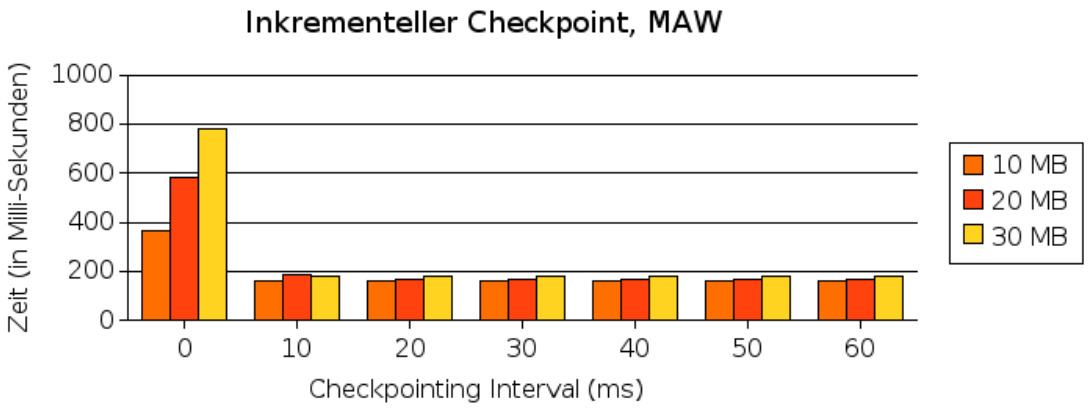

Abbildung 4.2: MallocAndWrite, Inkrementeller Checkpoint

Die Abbildung 4.1 stellt die inkrementellen Checkpointing Zeiten der Anwendung *MAW* dar. Diese Anwendung zeigt einen Fall, bei dem eine inkrementelle Sicherung sich viel schneller durchführen lässt als die Erstellung einer kompletten, sequentiellen Sicherung. Dies liegt daran, dass bei dieser Anwendung nach dem initialen Checkpoint nur wenige Seiten verändert werden. Dieses Verhalten führt dazu, dass nur wenige Seiten gesichert werden müssen und ermöglicht somit eine schnellere Sicherung der Anwendung. Bei der Betrachtung der ersten Sicherung von 10 Megabyte ist dieses Verhalten gut zu sehen. Hier dauert die initiale Sicherung zum Zeitpunkt 0 genauso lange wie die sequentielle Sicherung. Dies liegt daran, dass bei der ersten Sicherung sowohl beim sequentiellen als auch beim inkrementellen Checkpoint alle Seiten eingesammelt werden müssen. Zusätzlich zum Einsammeln der Seiten kommt es bei der inkrementellen Sicherung, aufgrund der benötigten Verwaltungsstruktur, zu weiteren Verzögerungen. Somit dauert die erste inkrementelle Sicherung in der Regel länger als die erste sequentielle Sicherung. Bereits die zweite Sicherung zum Zeitpunkt 10 zeigt jedoch die Vorteile des inkrementellen Checkpointing. Beim sequentiellen Checkpointing werden hierbei stets alle Seiten gesichert. Somit dauert die Sicherung genauso lange wie zum Zeitpunkt 0. Die inkrementelle Sicherung verläuft jedoch doppelt so schnell, weil nur die veränderten Seiten gesichert werden müssen. Bei den restlichen Sicherungsintervallen 20-50 dauert die inkrementelle Sicherung in etwa genauso lange wie zum Zeitpunkt 10, weil sich in der Zwischenzeit

immer die gleiche Menge an Seiten ändert. Die Aktualisierung der Verwaltungsstruktur verursacht aufgrund der guten Laufzeit des in Kapitel 3 vorgestellten RBTs keinen großen zusätzlichen Rechenaufwand.

4.4.1.2 Restart

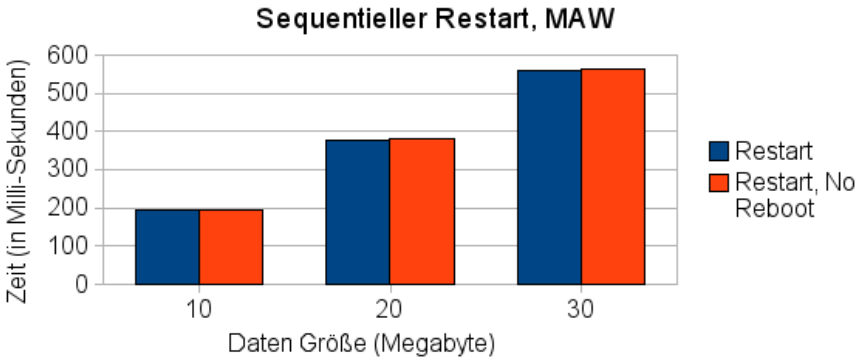

Abbildung 4.3: MallocAndWrite, Sequentieller Restart

Die Abbildung 4.3 zeigt die sequentiellen Restart Zeiten für die Anwendung *MAW*. Wie in der Abbildung zu sehen, befindet sich auf der x-Achse die Datengröße der Sicherung. Die y-Achse beinhaltet die benötigte Zeit für den Restart der Anwendung. Der sequentielle Restart dieser Anwendung dauert sowohl mit, als auch ohne einen Reboot in etwa gleich lang. Dies hängt damit zusammen, dass diese Anwendung keine Strukturen benutzt, die im Falle eines Restarts komplett neu aufgebaut werden müssten. Diese Strukturen könnten z.B. Shared Memory Segmente sein. Bei den Anwendungen, die solche Strukturen benutzen würde der Restart ohne einen vorherigen Reboot schneller von statten gehen.

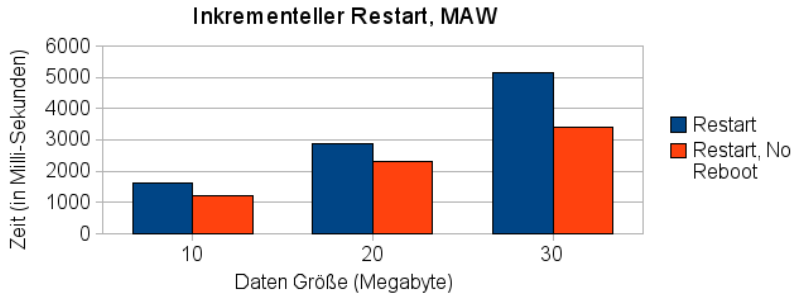

Abbildung 4.4: MallocAndWrite, Inkrementeller Restart

Einige Nachteile des inkrementellen Checkpointings können anhand der Restartwerte auf der Abbildung 4.4 beobachtet werden. Diese sind schlechter als beim sequentiellen Restart. Das hängt damit zusammen, dass beim sequentiellen Restart die physikalischen Seiteninhalte direkt aus der entsprechenden Datei *(task_mm_pid_Vx.bin)* sequentiell gelesen werden. Beim inkrementellen Restart jedoch, muss zuerst der RBT aus der Information innerhalb der Verwaltungsstruktur aufgebaut werden. Anschließend müssen die virtuellen Adressen innerhalb des Baumes ermittelt werden, um den physikalischen Seiteninhalt zu lokalisieren. Die physikalischen Seiteninhalte müssen unter Umständen aus vielen unterschiedlichen Dateien gelesen werden. Diese Lesevorgänge verursachen aufgrund der vielen Kopfbewegungen der Festplatte weitere Verzögerungen. Diese Vorgänge beanspruchen die meiste Zeit beim Wiederherstellen der Anwendung. Einige Optimierungsmöglichkeiten, um die Laufzeit zu verbessern, werden in Kapitel 5 vorgestellt.

4.4.1.3 Checkpointgrößen

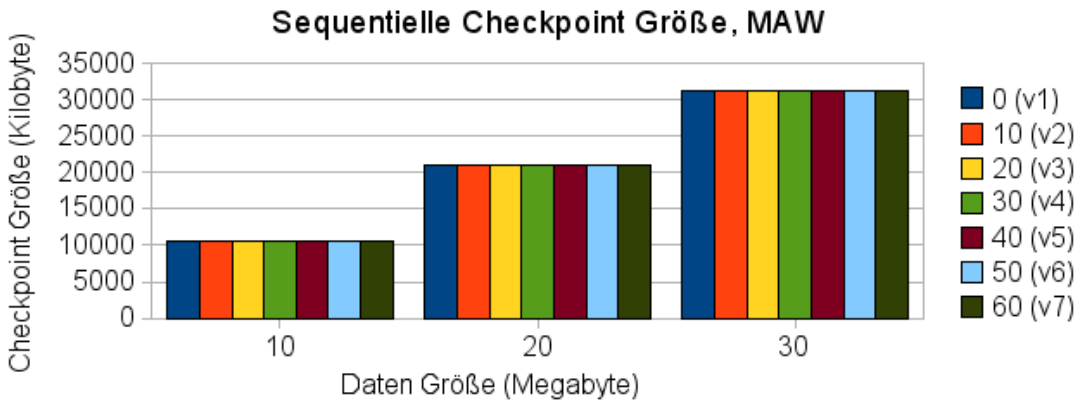

Abbildung 4.5: MallocAndWrite, Sequentielle Checkpointgröße

Die Abbildung 4.5 zeigt die sequentiellen Checkpointgrößen. Wie in der Abbildung zu sehen, befindet sich auf der x-Achse die Datengröße der Sicherung. Die y-Achse beinhaltet die Checkpointgröße. Mit den Farben wird der Zeitpunkt der Sicherung angegeben. Da beim sequentiellen Checkpointing alle Seiten gesichert werden, ist die Checkpointgröße zu allen Zeitpunkten gleich.

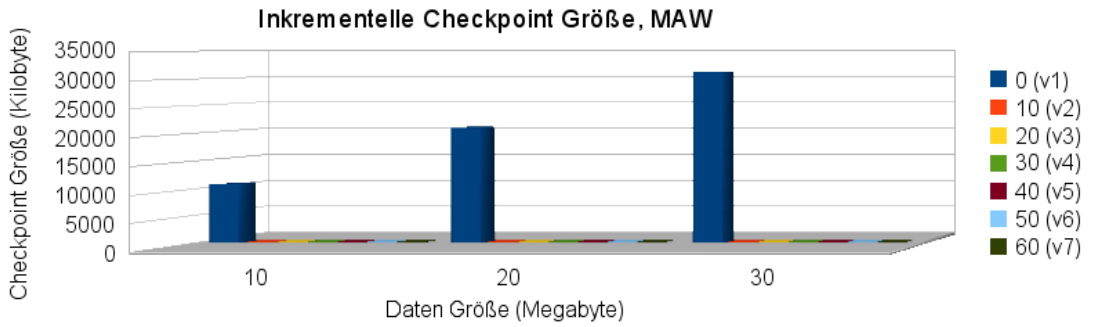

Abbildung 4.6: MallocAndWrite, Inkrementelle Checkpointgröße

In der Abbildung 4.6 werden die inkrementellen Checkpointgrößen veranschaulicht. Da bei dieser Anwendung zum Zeitpunkt 10-60 im Falle einer inkrementellen Sicherung nur die wenigen veränderten Seiten gesichert werden, resultiert dies in kleineren Checkpointgrößen.

4.4.1.4 Kontrollgrößen

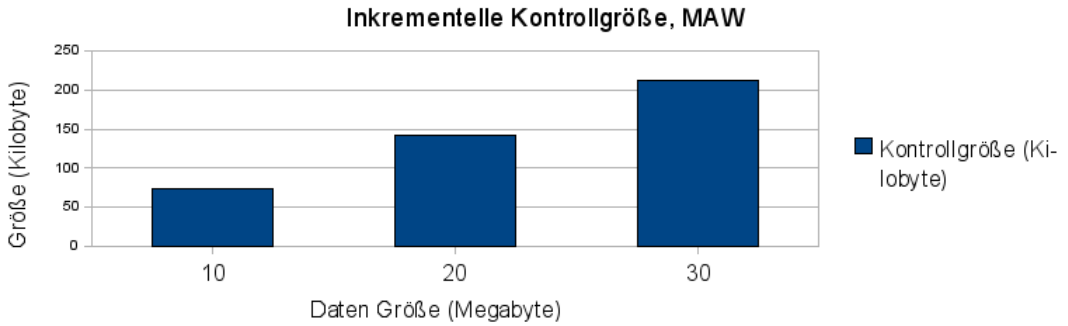

Abbildung 4.7: MallocAndWrite, Kontrollgrößen

Die Abbildung 4.7 zeigt die Größen der angelegten Verwaltungsstrukturen. Diese Größen unterscheiden sich im Bezug auf die Anzahl der zu sichernden Seiten. Somit ist die Verwaltungsstruktur des 10MB Checkpoints kleiner als die des 20MB Checkpoints. Dies hängt mit der Anzahl der Knoten des RBTs zusammen. Diese ist bei einem 20MB Checkpoint in etwa doppelt so groß wie die des 10MB Checkpoints.

4.4.2 Messungen zu der Anwendung MGS

4.4.2.1 Checkpoint

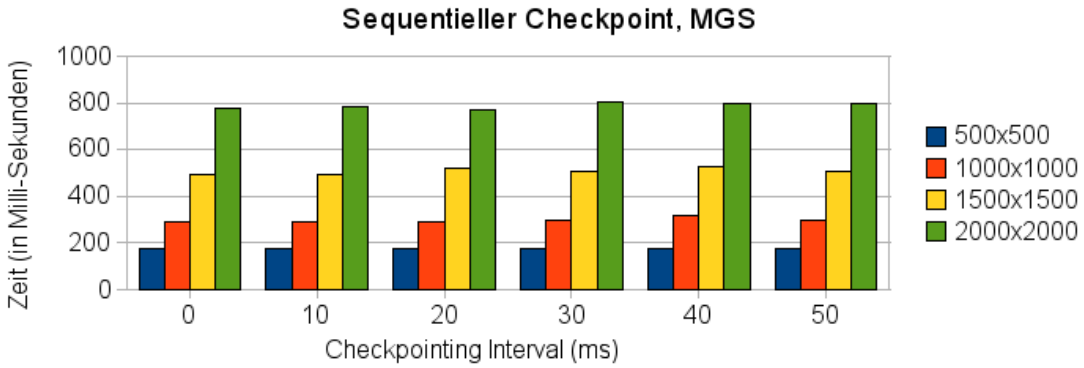

Abbildung 4.8: Modified Gram-Schmidt, Sequentieller Checkpoint

Die Abbildung 4.8 zeigt das sequentielle Checkpointing Verhalten der Anwendung *MGS*. Auf der x-Achse befindet sich das Sicherungsintervall. Die y-Achse gibt die benötigte Zeit für die Sicherung an. Unterschiedliche Farben bezeichnen die Matrixgrößen. Wie schon bei der Anwendung *MAW* zu sehen war, wird auch bei dieser Anwendung zu allen Zeitpunkten die selbe Anzahl der Seiten gesichert. Somit dauert das Checkpointing der Daten zu allen Zeitpunkten gleich lang.

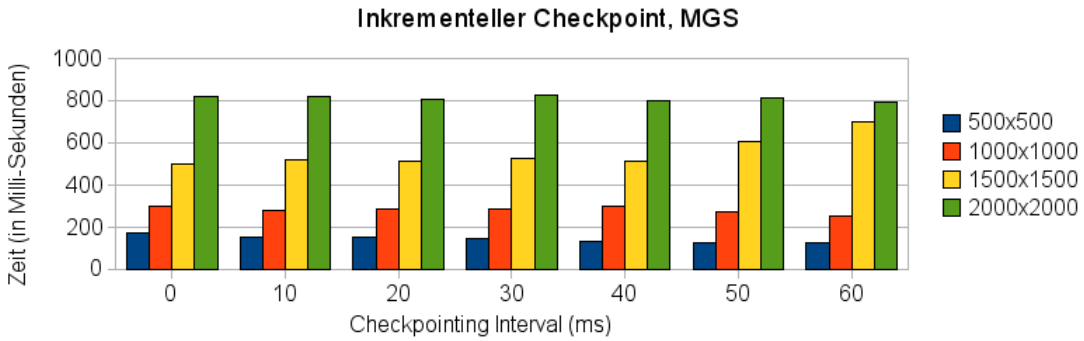

Abbildung 4.9: Modified Gram-Schmidt, Inkrementeller Checkpoint

Die Abbildung 4.9 zeigt das inkrementelle Checkpointing Verhalten der Anwendung *MGS*. Diese Anwendung zählt zu der Klasse der rechenintensiven Anwendungen. Ferner werden die Seiteninhalte zu jedem Zeitpunkt permanent neu beschrieben. Solche Anwendungen haben beim inkrementellen Checkpointing teilweise sogar eine schlechtere Laufzeit als beim sequentiellen Checkpointing. Dies hängt damit zusammen, dass bei einer

größeren Menge der veränderten Seiten der RBT sehr oft durchsucht werden muss, um die Inhalte zu aktualisieren. Das Durchsuchen des Baumes führt trotz einer guten Laufzeit von O(log n) zu weiteren Verzögerungen. Zusätzlich zum Durchsuchen des Baumes beim inkrementellen Checkpointing kommt noch der I/O-Overhead durch die Sicherung der Seiten auf die Festplatte hinzu. Im Falle des sequentiellen Checkpointings entfällt der Teil des RBTs komplett und resultiert in einer schnelleren Checkpointing Laufzeit. Dieses Verhalten wird in der Abbildung 4.8 verdeutlicht. Durch die nahezu komplette Sicherung der Seiten, dauert das inkrementelle Checkpointing in der Regel länger, als das sequentielle Checkpointing. In den Fällen, bei denen das sequentielle Checkpointing dennoch schneller geht, ist die Anzahl der zu sichernden Seiten so klein, dass der Rechenaufwand für das Suchen im Baum zu vernachlässigen ist.

4.4.2.2 Restart

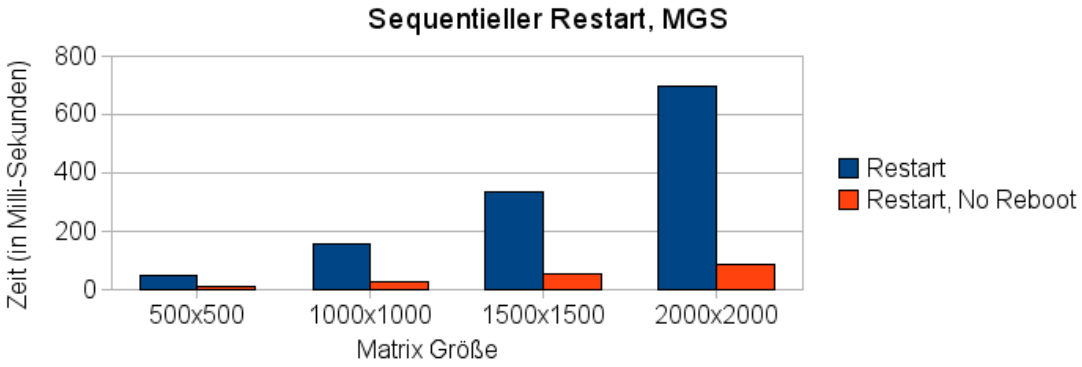

Abbildung 4.10: Modified Gram-Schmidt, Sequentieller Restart

Die Abbildung 4.10 zeigt das sequentielle Restartverhalten der Anwendung *MGS*. Die Anwendung *MGS* benutzt für ihre Berechnungen einige Kerrighed spezifische Strukturen. Diese müssen im Falle eines Restarts nach einem Reboot wieder aufgebaut werden. Dadurch dauert der sequentielle Restart dieser Anwendung nach einem Reboot länger.

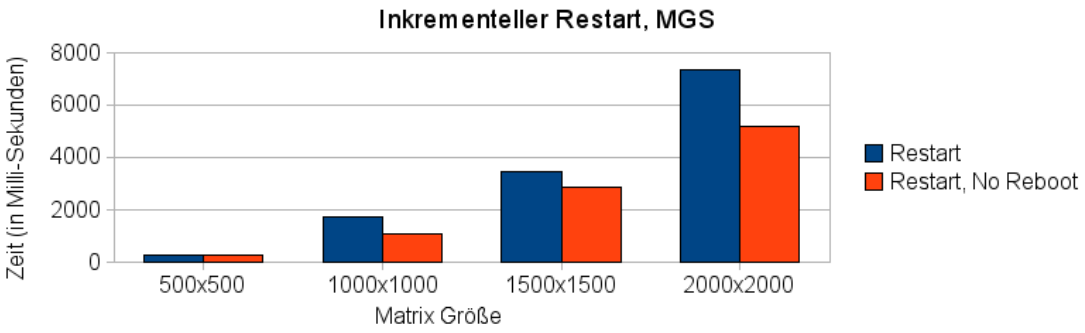

Abbildung 4.11: Modified Gram-Schmidt, Inkrementeller Restart

In der Abbildung 4.11 werden die inkrementellen Restartzeiten dargestellt. Hier zeigt sich ein ähnliches Bild wie schon bei der inkrementellen Restart Messung zu der Anwendung *MAW*. Der inkrementelle Restart dauert aufgrund des zusätzlichen Rechenaufwandes, der durch die Abbildung mit der Verwaltungsstruktur entsteht, länger als der sequentielle Restart. Zusätzlich kommt noch wie beim Restart der Anwendung *MGS* der Zeitverlust durch die Kopfbewegungen der Festplatte hinzu. Einige mögliche Optimierungen, um die Laufzeit zu verbessern, werden wie bereits erwähnt in Kapitel 5 vorgestellt.

4.4.2.3 Checkpointgrößen

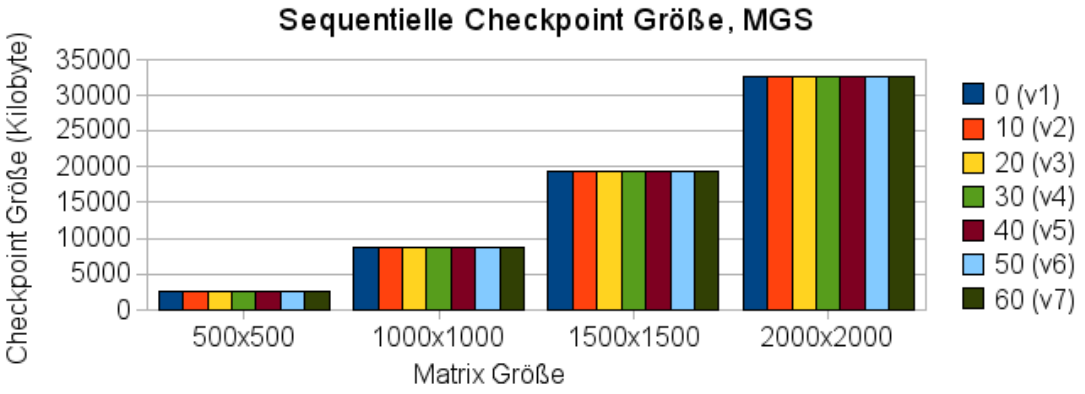

Abbildung 4.12: Modified Gram-Schmidt, Sequentielle Checkpointgröße

In der Abbildung 4.12 sind die resultierenden, sequentiellen Checkpointgrößen aufgetragen. Wie in der Abbildung zu sehen, befindet sich auf der x-Achste die Matrixgröße und

auf der y-Achse die Checkpointgröße. Die Farben geben den Zeitpunkt der Sicherung an. Da beim sequentiellen Checkpointing alle Seiten gesichert werden, ist die Checkpointgröße auch bei dieser Anwendung zu allen Zeitpunkten gleich.

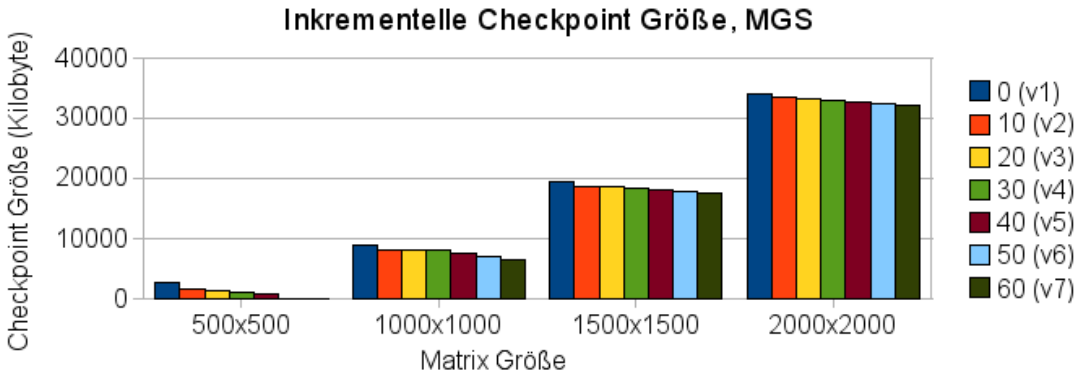

Abbildung 4.13: Modified Gram-Schmidt, Inkrementelle Checkpointgröße

Die Abbildung 4.13 zeigt die inkrementellen Checkpointgrößen. Wie bereits beim *Checkpointing* erwähnt, werden bei dieser Anwendung fast alle Seiten überschrieben. Dadurch unterscheidet sich die Checkpointgröße zu allen Zeitpunkten für die sequentielle und für die inkrementelle Sicherung nur geringfügig, weil nahezu alle Seiten gesichert werden.

4.4.2.4 Kontrollgrößen

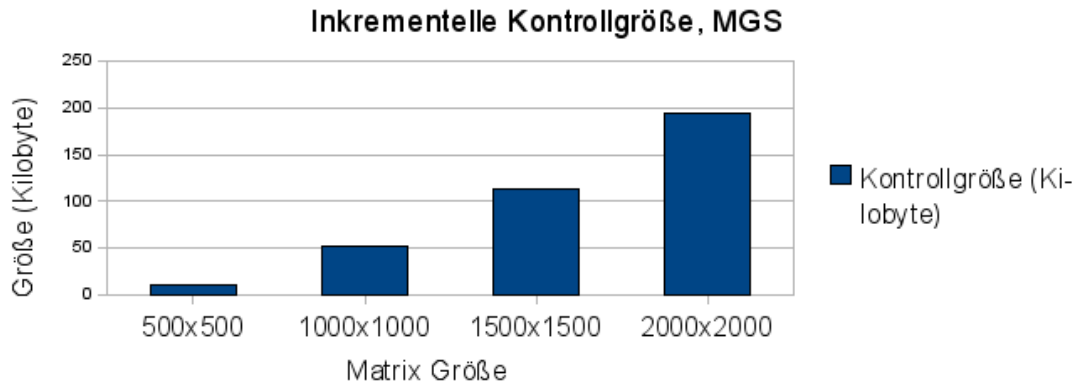

Abbildung 4.14: Modified Gram-Schmidt, Inkrementelle Kontrollgrößen

In der Abbildung 4.14 werden die Größen der Verwaltungsstrukturen abgebildet. Wie auch bei der *MAW* Anwendung, hängt die Größe der Verwaltungsstruktur von der Größe

der Matrix und somit von dem resultierenden RBT ab. Der Baum hat bei einer Matrixgröße von *1000x1000* doppelt so viele Knoten wie der Baum für die *500x500* Matrix. Somit ist auch die Verwaltungsstruktur doppelt so groß.

4.5 Schlussfolgerung

In diesem Kapitel wurde die Performance des inkrementellen Checkpointings der Performace des sequentiellen Checkpointings gegenübergestellt. Es lässt sich zusammenfassend sagen, dass es stark von der Anwendungsklasse abhängt, ob sequentielles oder inkrementelles Checkpointing besser geeignet ist. Die in diesem Kapitel vorgestellten Anwendungen *MAW* und *MGS* haben einige Vor- und Nachteile der beiden Checkpointing Strategien aufgezeigt. Die Anwendungen bei denen die Inhalte der physikalischen Seiten selten verändert werden, eignen sich sehr gut für das inkrementelle Checkpointing. Aufgrund des möglichen, hohen Aufwands für die Aktualisierung der Verwaltungsstruktur, sind die Anwendungen bei denen die Inhalte der physikalischen Seiten häufig verändert werden, für das inkrementelle Checkpointing weniger geeignet. In beiden Fällen dauert der Restartvorgang immer etwas länger als beim sequentiellen Checkpointing. Aufgrund der Vielzahl der durchgeführten Messungen, werden im Anhang die Messergebnisse der in dieser Arbeit vorgestellten Programme in tabellarischer Form dargestellt.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

In dieser Bachelorarbeit wurde der bestehende Kerrighed Checkpointer um die Möglichkeit der inkrementellen Sicherungen erweitert. Das inkrementelle Sichern der Daten ist in allen Bereichen notwendig, wo ein kompletter Checkpoint sehr umfangreich wäre. Dies ist insbesondere bei datenintensiven Anwendungen, wie z.B. Datenbanken sehr relevant, weil hier eine permanente Sicherung des kompletten Datenbestandes sehr teuer ist. Da sich beim inkrementellen Checkpointing die Inhalte der physikalischen Seiten verteilt auf der Festplatte, in mehreren Dateien befinden können, wurde eine Verwaltungsstruktur zur Lokalisierung dieser Inhalte entwickelt. Diese Verwaltungsstruktur wurde unter Verwendung eines Red-Black-Trees (RBTs) realisiert. Mit Hilfe der in der Verwaltungsstruktur gespeicherten Informationen wird ein Restart der Anwendung ermöglicht. Für die Erkennung der veränderten Seiten wurde ein Mechanismus zum Abfangen der Schreibzugriffe und Einsammeln der Seiten auf der Kernel-Ebene entworfen und in den bestehenden Checkpointer erfolgreich integriert. Dieser Mechanismus erkennt die veränderten Seiten und aktualisiert die Verwaltungsstruktur. Danach wird die Verwaltungsstruktur mittels des Ghost-Mechanismus auf der Festplatte gesichert. Der Restart-Vorgang musste ebenfalls erweitert werden und erkennt jetzt automatisch, ob es sich bei der wiederherzustellenden Sicherung um eine inkrementelle oder um eine sequentielle Sicherung handelt. Es ist nun möglich mit der Angabe eines zusätzlichen Flags *checkpoint -i* von der Kommandozeile aus inkrementelle Sicherungen zu erstellen.
Beide Checkpointing-Strategien wurden anschließend bezüglich deren Performance un-

tersucht und analysiert.
Zusammenfassend läst sich sagen, dass es stark von der Anwendungsklasse abhängt, ob sequentielles oder inkrementelles Checkpointing besser geeignet ist. Die durchgeführten Messungen zeigten, dass die Anwendungen bei denen die Inhalte der physikalischen Seiten selten verändern werden, sehr gut für das inkrementelle Checkpointing geeignet sind. Anwendungen bei denen die Inhalte der physikalischen Seiten häufig überschrieben werden, sind für das inkrementelle Checkpointing weniger geeignet. Bei solchen Anwendungen verläuft das inkrementelle Checkpointing u.U. sogar langsamer. Häufige Änderungen der Seiten erfordern mehrfaches durchlaufen des RBTs, um dessen Inhalte zu aktualisieren. Zusätzlich zu der Aktualisierung der Verwaltungsstruktur kommt noch der I/O-Overhead durch das Sichern der physikalischen Seiteninhalte auf die Festplatte hinzu. Unabhängig von der Anwendungsklasse, dauert der anschließende inkrementelle Restart immer etwas länger. Dies hängt damit zusammen, dass im Falle eines inkrementellen Restarts zuerst der RBT aus den Informationen innerhalb der Verwaltungsstruktur aufgebaut werden muss. Danach müssen die virtuellen Adressen innerhalb des Baumes ermittelt werden, um den physikalischen Seiteninhalt zu lokalisieren. Anschliessend werden die physikalischen Seiteninhalte, unter Umständen, aus mehreren unterschiedlichen Sicherungsdateien gelesen. Dafür sind viele Kopfbewegungen der Festplatte notwendig. Dennoch ist das implementierte, inkrementelle Verfahren nützlich und kann weiter optimiert werden.

5.2 Ausblick

Das in dieser Arbeit erarbeitete Konzept des inkrementellen Checkpointings kann weiter verbessert werden. Sollte sich die Anzahl der VMAs oder der virtuellen Adressen ändern, wird stets ein kompletter Checkpoint erzeugt. Dies ist nicht zwingend notwendig und kann durch die Aktualisierung der Verwaltungsstruktur an den entsprechenden Stellen verhindert werden. Es erfordert jedoch, aufgrund der unter Umständen vielen Vergleiche, eine andere, effizientere Verwaltungsstruktur. Die in dieser Arbeit realisierte Verwaltungsstruktur mit Hilfe eines RBTs kann z.B. durch eine mehrstufige Seitentabelle, wie die, von der virtuellen Speicherverwaltung verwendet wird, ersetzt werden. Dadurch würden sich die Zeiten für das Suchen der virtuellen Adressen auf eine sehr gute Laufzeit von O(1) verbessern. Eine solche Struktur würde eine effiziente Aktuali-

sierung der Verwaltungsstruktur, im Falle einer Veränderung der Anzahl von VMAs oder der virtuellen Adressen, ermöglichen. Für den Spezialfall, bei dem eine VMA verschwindet und eine neue VMA mit dem gleichen virtuellen Addressraumbereich hinzukommt, ist auch bei dieser Verwaltungsstruktur die Einführung einer eindeutigen Nummer zur Identifikation der VMA notwendig.
Durch eine Optimierung des Restart-Mechanismus, kann der RBT bei dem inkrementellen Restart umgangen werden. Dies wird dadurch ermöglicht, dass die virtuellen Adressen innerhalb der Verwaltungsstruktur in der selben Reihenfolge liegen, wie die virtuellen Adressen der VMAs, die im Falle eines Restart durchlaufen werden. Dadurch wird sich die Laufzeit des Restarts verbessern, weil sowohl der Aufbau des Baumes, als auch das Suchen im Baum entfällt.
Ein anderer, sehr wichtiger Aspekt ist die bereits in Kapitel 2 erwähnte Garbage Collection. Jede Sicherung benötigt Speicher. Da die Anwendungen im Laufe der Zeit unter Umständen sehr viele Änderungen durchführen, müssen diese auf der Festplatte bei jedem Checkpointvorgang gesichert werden. Mit der Zeit kann sich jedoch ein Teil dieser Sicherung als veraltet und somit nutzlos erweisen. Diese nutzlosen Daten sollen mit Hilfe eines Garbage Collection Algorithmus beseitigt werden.

Anhang A

Messergebnisse von sequentiellem und inkrementellem Checkpointing

Folgender Kapitel stellt die Messergebnisse der in dieser Arbeit vorgestellten Programme in tabellarischer Form dar.

A.1 Malloc and Write (MAW)

A.1.1 Checkpoint

Die folgende Tabelle liefert die Checkpoint-Ergebnisse zu der Anwendung *MAW*:

Checkpointing Interval (ms)	0	10	20	30	40	50	60
10 MB, sequentiell	364,42	362,28	363,06	360,81	365,84	369,37	365,54
10 MB, inkrementell	364,84	159,94	158,73	158,74	158,91	158,72	158,81
20 MB, sequentiell	569,11	575,21	580,14	574,11	574,08	570,38	569,36
20 MB, inkrementell	584,6	185,74	167,76	167,84	167,95	167,47	168,4
30 MB,sequentiell	773,11	774,02	778,42	777,92	777,66	774,82	776,96
30 MB, inkrementell	782,57	177,99	177,25	177,12	177,1	177,12	177,18

Tabelle A.1: Checkpoint-Ergebnisse, MallocAndWrite

A.1.2 Restart

Die folgende Tabelle liefert die Restart-Ergebnisse zu der Anwendung *MAW*:

	10	20	30
Restart, sequentiell	193,88	378,77	558,99
Restart, inkrementell	1618,44	2880,55	5151,62
Restart, No Reboot, sequentiell	194,7	379,37	562,14
Restart, No Reboot, inkrementell	1211,87	2289,57	3417,14

Tabelle A.2: Restart-Ergebnisse, MallocAndWrite

A.1.3 Checkpointgrößen

Die folgende Tabelle liefert die resultierenden Checkpointgrößen der Anwendung *MAW*:

Daten Größe (Kilobyte)	10	20	30
0 (v1), sequentiell	10680,82	20936,81	31200,82
0 (v1), inkrementell	10680,82	20936,81	31200,82
10 (v2), sequentiell	10680,82	20936,81	31200,82
10 (v2), inkrementell	35,97	32,06	34,12
20 (v3), sequentiell	10680,82	20936,81	31200,82
20 (v3), inkrementell	35,97	36,07	36,07
30 (v4), sequentiell	10680,82	20936,81	31200,82
30 (v4), inkrementell	36,07	32,06	36,07
40 (v5), sequentiell	10680,82	20936,81	31200,82
40 (v5), inkrementell	36,07	36,07	32,06
50 (v6), sequentiell	10680,82	20936,81	31200,82
50 (v6), inkrementell	36,07	32,06	32,06
60 (v7), sequentiell	10680,82	20936,81	31200,82
60 (v7), inkrementell	36,07	36,07	36,07

Tabelle A.3: Checkpointgrößen, MallocAndWrite

A.1.4 Kontrollgrößen

Die folgende Tabelle liefert die Größe der Verwaltungsstrukturen der Anwendung *MAW*:

	10	20	30
Kontrollgrößen (Kilobyte)	72,86	142,86	212,86

Tabelle A.4: Kontrollgrößen, MallocAndWrite

A.2 Modified Gram-Schmidt (MGS)

A.2.1 Checkpoint

Die folgende Tabelle liefert die Checkpoint-Ergebnisse zu der Anwendung *MGS*:

Checkpointing Interval (ms)	0	10	20	30	40	50
500x500, sequentiell	174,8	173,67	173,4	173,52	173,35	173,21
500x500, inkrementell	176,72	156,04	153,96	149,55	134,1	127,17
1000x1000, sequentiell	291,55	291,15	292,71	295,66	315,66	295,65
1000x1000, inkrementell	298,74	283,41	286,97	287,64	303,65	271,63
1500x1500, sequentiell	491,27	495,61	519,61	503,62	523,63	507,58
1500x1500, inkrementell	502,68	519,63	515,64	523,72	511,63	607,68
2000x2000, sequentiell	776,3	782,21	767,6	803,61	795,61	799,61
2000x2000, inkrementell	819,51	818,13	807,65	823,64	799,62	811,65

Tabelle A.5: Checkpoint-Ergebnisse, Modified Gram-Schmidt

A.2.2 Restart

Die folgende Tabelle liefert die Restart-Ergebnisse zu der Anwendung *MGS*:

	500x500	1000x1000	1500x1500	2000x2000
Restart, sequentiell	48,4	156,17	336,35	695,33
Restart, inkrementell	259,34	1081,26	2839,04	5166,96
Restart, No Reboot, sequentiell	9,29	24,67	50,93	86,57
Restart, No Reboot, inkrementell	262,77	1700,83	3480,32	7381,98

Tabelle A.6: Restart-Ergebnisse, Modified Gram-Schmidt

A.2.3 Checkpointgrößen

Die folgende Tabelle liefert die resultierenden Checkpointgrößen der Anwendung *MGS*:

Checkpoint Größe (Kilobyte)	500x500	1000x1000	1500x1500	2000x2000
0 (v1), sequentiell	2560	8601,6	19456	32768
0 (v1), inkrementell	2519,04	8949,76	19445,76	34160,64
10 (v2), sequentiell	2560	8601,6	19456	32768
10 (v2), inkrementell	1474,56	8151,04	18759,68	33536
20 (v3), sequentiell	2560	8601,6	19456	32768
20 (v3), inkrementell	1382,4	8110,08	18728,96	33157,12
30 (v4), sequentiell	2560	8601,6	19456	32768
30 (v4), inkrementell	1126,4	8017,92	18452,48	32931,84
40 (v5), sequentiell	2560	8601,6	19456	32768
40 (v5), inkrementell	849,92	7546,88	18186,24	32716,8
50 (v6), sequentiell	2560	8601,6	19456	32768
50 (v6), inkrementell	20,48	7065,6	17889,28	32501,76
60 (v7), sequentiell	2560	8601,6	19456	32768
60 (v7), inkrementell	20,48	6533,12	17623,04	32276,48

Tabelle A.7: Checkpointgrößen, Modified Gram-Schmidt

A.2.4 Kontrollgrößen

Die folgende Tabelle liefert die Größe der Verwaltungsstrukturen der Anwendung *MGS*:

	500x500	1000x1000	1500x1500	2000x2000
Kontrollgrößen (Kilobyte)	10,24	51,2	112,64	194,56

Tabelle A.8: Kontrollgrößen, Modified Gram-Schmidt

Literaturverzeichnis

[BC05] BOVET, Daniel P.; CESATI, Marco: *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2005. – ISBN 978–0596000028

[CLR01] CORMEN, Thomas H.; LEISERSON, Charles; RIVEST, Ronald L.: *Introduction to Algorithms, Second Edition*. MIT Press, 2001. – ISBN 978–0262032933

[Gor04] GORMAN, Mel: *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004. – ISBN 978–0131453487

[Int08] INTEL: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide*. Intel Corporation, 2008 `http://download.intel.com/design/processor/manuals/253668.pdf`. – ISBN 253668–026US

[Klü07] KLÜGEL, Niklas: *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*. `http://www13.informatik.tu-muenchen.de/lehre/seminare/SS07/hauptsem/checkpointing.pdf`. Version: 2007

[LGVM04] LOTTIAUX, Renaud; GALLARD, Pascal; VALLEE, Geoffy; MORIN, Christine: *OpenMosix, OpenSSI and Kerrighed: A Comparative Study*. `http://www.irisa.fr/paris/Biblio/Papers/Lottiaux/LotBoiGalValMor05CCGrid.pdf`. Version: 2004

[MGL04] MORIN, Christine; GALLARD, Pascal; LOTTIAUX, Renaud: *Towards an efficient single system image cluster operating system*. `http://www.irisa.fr/paris/Biblio/Papers/Morin/MorGalLotVal03FGCS.pdf`. Version: 2004

[Mon99] MONTENEGRO, S.: *Prinzipien der Fehlertoleranz*. `http://sergio.montenegros.de/public/ft_einf.html`. Version: 1999

[MS07] MEHNERT-SPAHN, John: *Design and implementation of basic checkpoint/restart mechanisms in LinuxSSI*. http://www.xtreemos.eu/publications/project-deliverables/d2-2-3-final.pdf. Version: 2007

[RC02] RUBINI, Alessandro; CORBET, Jonathan: *Linux-Gerätetreiber, 2. Auflage*. O'Reilly, 2002 http://www.oreilly.de/german/freebooks/linuxdrive2ger/book1.html. – ISBN 978–3897211384

[Rus99] RUSLING, David A.: *The Linux Kernel*. http://tldp.org/LDP/tlk/. Version: 1999

[Sch05] SCHWIND, Michael: *Vergleich paralleler modifizierter Gram-Schmidt Algorithmus*. http://gustav.informatik.tu-chemnitz.de/kicc05/files/paper07.pdf. Version: 2005

[Tan02] TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. Prentice Hall, 2002. – ISBN 978–3827370198

[VLM+05] VALLEE, Geoffroy; LOTTIAUX, Renaud; MARGERY, David; MORIN, Christine; BERTHOU, Jean-Yves: *Ghost Process: a Sound Basis to Implement Process Duplication, Migration and Checkpoint/Restart in Linux Clusters*. http://www.irisa.fr/paris/Biblio/Papers/Vallee/ispdc05.ps. Version: 2005

[Wol04] WOLF, Jürgen: *Linux-UNIX-Programmierung*. Galileo Computing, 2004. – ISBN 3–89842–749–8

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 06. März 2008 Eugen Feller