Großer Beleg

Parallelisierung von Valgrind

Frank Tetzel

20. Januar 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. Hermann Härtig
Betreuende Mitarbeiter: Dipl.-Inf. Björn Döbel
Dipl.-Inf. Michael Roitzsch

Inhaltsverzeichnis

Aufgabenstellung

Valgrind ist ein flexibles Framework zum Erstellen von Tools für dynamische Binary-Instrumentierung. Das von Valgrind implementierte Shadow-Memory-Konzept ermöglicht die feingranulare Analyse von Speicherzugriffen. Um Shadow Memory möglichst einfach zu implementieren, verzichtet Valgrind auf die parallele Ausführung von Threads und serialisiert deren Ausführung. Dies führt jedoch zu einer Erhöhung des Ausführungs-Overheads bei Anwendungen mit mehreren echt parallelen Threads.

Bisherige Ansätze Valgrind zu parallelisieren sind mit einer deutlichen Reduktion der Funktionalität (keine Shadow Values) verbunden. Im Rahmen dieser Arbeit soll untersucht werden, ob Alternativen existieren, die ohne solche Einschränkungen auskommen. Hierzu ist zuerst der Valgrind-Kern zu analysieren und unter Nutzung geeigneter Mechanismen zu parallelisieren. Mögliche Optionen, wie bspw. Software-Transactional Memory oder feingranularen Locking, sind auf ihre Anwendbarkeit zu untersuchen.

Im weiteren ist zu prüfen, wie sich die implementierten Änderungen auf die verfügbaren Valgrind-Werkzeuge auswirken. Die Parallelisierung dieser Werkzeuge selbst ist nicht Bestandteil der Aufgabe. Hier sind jedoch Lösungsmöglichkeiten aufzuzeigen und zu bewerten, welche in einer zukünftigen Folgearbeit umgesetzt werden können.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 20. Januar 2012

Frank Tetzel

Motivation

Das Aufgabenfeld für Computerprogramme wächst stetig. Immer mehr Aufgaben werden computergestützt bearbeitet. Dabei ist es kaum vermeidbar, dass die Komplexität der Software immer weiter steigt. Mit der Komplexität schleichen sich auch immer mehr Programmfehler in die Software. Menschen machen bekanntlich Fehler. Besonders systemnahe Programmiersprachen wie C und C++ schützen den Programmierer aus Performancegründen wenig vor seinen eigenen Fehlern.

Debugwerkzeuge sind im Alltag eines Programmierers nicht mehr wegzudenken. Bei der unliebsamen Suche nach Programmfehlern ist es wichtig, möglichst schnell die Ursache des Problems zu lokalisieren. Analysetools, die Software vor oder während der Ausführung auf Schwachstellen hin untersuchen, können dabei eine erhebliche Hilfe sein. Valgrind [24] ist ein Framework zum Erstellen von dynamischen Analysetools, die während der Ausführung den Code instrumentieren und verschiedenste Daten über den Programmablauf sammeln. Das recht beliebte Standardwerkzeug von Valgrind, Memcheck, findet dabei Fehler in der Speicherverwaltung wie Speicherlecks und Pufferüberläufe.

Memcheck nutzt dabei Shadow Memory um Metadaten wie Initialisierungszustand und Adressierbarkeit des Speichers mitzuführen. Um Parallelisierungsproblemen bei Shadow Memory aus dem Weg zu gehen, führt Valgrind Programme mit parallelen Threads nur seriell aus. Das bedeutet, dass auf einem Mehrkernsystem nur ein Kern belastet wird und viel Rechenkapazität ungenutzt bleibt. Ziel dieser Arbeit ist es einen ersten Schritt in Richtung paralleler Ausführung unter Valgrind zu unternehmen. Dabei wird an vielen Stellen von Vorarbeiten wie pValgrind [25] profitiert.

Gliederung

Kapitel 1 stellt grundlegende Verfahren und Werkzeuge vor, die im weiteren Verlauf der Arbeit wichtig sind. Im nächsten Kapitel 2.5 werden mehrere Synchronisationsverfahren auf deren Eignung für die Nutzung innerhalb von Valgrind untersucht. Dabei spielt die einfache Implementierbarkeit eine wichtige Rolle. Die verschiedenen Parallelisierungsprobleme in Valgrind und deren Lösungen werden in Kapitel 3 besprochen. Das anschließende Kapitel 4 vergleicht die Performance zwischen den parallelisierten Valgrind und den unveränderten, seriellen Valgrind. In Kapitel 5 wird die Parallelisierung von Shadow Memory diskutiert und ein möglicher Lösungsansatz vorgestellt. Das letzte Kapitel 6 schließt mit einer Zusammenfassung ab.

1. Einführung

Valgrind ist ein Framework um Werkzeuge zu erstellen, welche andere Programme instrumentieren. In Kapitel 1.1 wird das Konzept Instrumentierung und verschiedene Arten der Instrumentierung beschrieben. Das anschließende Kapitel 1.2 stellt das Framework Valgrind selbst vor. Im weiteren Verlauf der Arbeit wird die Parallelisierung von Valgrind diskutiert. Im Kapitel 1.3 wird pValgrind, eine verwandte Arbeit zur Parallelisierung von Valgrind, besprochen.

1.1. Instrumentierung

Bei der Analyse von Programmen wird zwischen statischer und dynamischer Analyse unterschieden. Die statische Analyse untersucht den Quelltext des zu untersuchenden Programms. Beispiele für die statische Analyse von C/C++-Quelltext sind die Programme Cppcheck [2] und Clang Static Analyzer [1]. Die dynamische Analyse untersucht das Programm während der Ausführung und sammelt je nach Werkzeug verschiedene Daten. Sie analysiert somit das wirkliche Verhalten bei der Ausführung und auch nur die dabei verwendeten Ausführungspfade. Das zu untersuchende Programm wird Gastprogramm genannt. Die dynamische Analyse wird häufig mittels Instrumentierung realisiert.

Bei der Instrumentierung wird Analysecode in das Gastprogramm integriert um während der Ausführung Informationen zu sammeln. Die Integration kann während der Kompilation stattfinden und neuen Quelltext einfügen. Ein Beispiel dafür ist gprof [4], welches die Ausführungszeit von Funktionen misst und direkt im Compiler GCC [3] enthalten ist. Der spezielle Parameter '-pg' in GCC fügt Instruktionen zur Messung ein. Um die Analyse durchzuführen, muss der Quelltext des Gastprogramms verfügbar sein und entsprechend neu kompiliert werden. Danach wird das modifizierte Programm ausgeführt und sammelt die entsprechenden Daten für die Analyse.

Eine andere Methode für die Integration des Analysecodes ist die binäre Instrumentierung. Der Analysecode wird direkt in das kompilierte Programm integriert. Somit wird der Quelltext nicht benötigt und die Analyse von proprietären Programmen, die häufig nur im Binärformat vorliegen, ist ebenfalls möglich. Es wird allgemein zwischen zwei Methoden unterschieden wie Analysecode in ein kompiliertes Programm eingebracht werden kann:

- Copy-And-Annotate [24] übernimmt einen Großteil des Binärcodes des Gastprogramms unverändert (Copy). Jede Instruktion wird dabei mit Kommentaren versehen (Annotate), welche die Effekte der Instruktion beschreiben. Mit diesen Annotationen arbeitet dann das Werkzeug um den einzufügenden Code zu erstellen. Die neuen Instruktionen werden dann zwischen die kopierten eingefügt.

- Disassemble-And-Resynthesize [24] überführt den kompletten Maschinencode des Gastprogramms in eine Zwischenrepräsentation (intermediate representation, IR) (Disassemble). Das Analysewerkzeug arbeitet ausschließlich auf der Zwischenrepräsentation und fügt den Analysecode in Form von IR-Instruktionen ein. Anschließend wird der komplette instrumentierte Code aus der Zwischenrepräsentation zurück in Maschinencode überführt (Resynthesize).

Beide Methoden haben ihre Vor- und Nachteile. Disassemble-And-Resynthesize besitzt einen hohen Implementationsaufwand für den JIT-Compiler, der die Übersetzung in und aus der IR sicherstellt. Die komplette Umwandlung in IR hat allerdings den entscheidenden Vorteil, dass darauf operierende Analysewerkzeuge beliebige Änderungen daran vornehmen können. Des Weiteren kann bei der Rückübersetzung von IR in Maschinencode der originale Gastcode zusammen mit dem Analysecode optimiert werden. Copy-And-Annotate ist performanter, da es einen Großteil des Maschinencodes unverändert übernimmt und lediglich annotiert. Die Sprache für die Annotationen ist allerdings nicht minder komplex als eine komplette Zwischenrepräsentation. Sie muss ebenfalls alle Effekte der Instruktion des Maschinencodes abbilden, damit das Werkzeug dementsprechenden angepassten Analysecode erzeugen kann, um möglichst das Verhalten des Gastprogramms nicht zu beeinflussen.

Für komplexe Analysewerkzeuge bietet sich die Nutzung von Disassemble-And-Resynthesize an, da damit auch beliebige Änderungen des Gastprogramms möglich sind. Bei einfacheren Werkzeugen lohnt sich der Einsatz von Copy-And-Annotate um den Performancevorteil auszunutzen.

1.2. Valgrind

Valgrind [24] ist ein Framework zur binären Instrumentierung von Programmen. Neben dem Framework zum Entwickeln eigener Werkzeuge werden auch einige fertige Werkzeuge mitgeliefert. Memcheck analysiert das Programm auf Fehler in der Speicherverwaltung. Cachegrind ist ein Cache-Profiler, Massif ein Heap-Profiler. Helgrind untersucht das Programm auf Data Races. Im Unterschied zu vielen anderen Debugwerkzeugen muss das Programm nicht erneut übersetzt werden, sondern kann im normalen Auslieferungszustand untersucht werden, da Valgrind Disassemble-And-Resynthesize umsetzt. Die Architekturen x86, AMD64, PPC32 und PPC64 und die Betriebssysteme Linux und Mac OS X werden offiziell unterstützt. Der Quellcode steht unter der GNU General Public License.

Valgrind lässt sich in 3 Bestandteile aufteilen: Den Valgrind-Kern (coregrind), die Übersetzungs- und Instrumentierungskomponente LibVEX und das Valgrind-Werkzeug. Abbildung 1.1 illustriert den Ablauf der Ausführung eines Gastprogramms innerhalb von Valgrind. Der Kern ist für das Starten des Gastprogramms und die Behandlung von Systemaufrufen zuständig. Beides ist betriebssystemspezifisch. LibVEX übersetzt den Maschinencode des Gastprogramms in eine Zwischenrepräsentation, welche von der Hardwarearchitektur unabhängig ist, und übersetzt auch wieder aus dieser Zwischenrepräsentation zurück in den Maschinencode. Ein Valgrind-Werkzeug operiert auf dieser Zwischenrepräsentation und verändert sie in geeigneter Weise. Dieses Verfahren hat

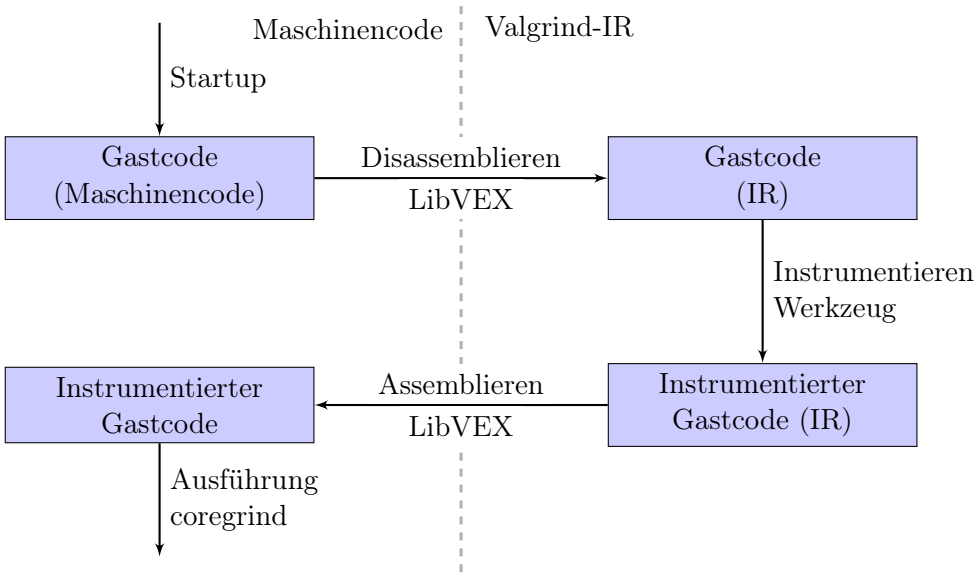

Abbildung 1.1.: Ablauf der Übersetzung und Instrumentierung in Valgrind

Vorteile für die Werkzeugentwickler. Zum einen ist ein Valgrind-Werkzeug dadurch unabhängig von der Hardwarearchitektur, zum anderen wird bei der Rückübersetzung in den Maschinencode sowohl der originale Gastcode als auch der vom Werkzeug veränderte Code zusammen optimiert. Ein schwerwiegender Nachteil ist der deutlich erhöhte Overhead bei der Ausführung mit Valgrind. Nulgrind, ein Werkzeug in Valgrind, welches keinerlei Instrumentierung vornimmt und nur zum Testen von der Hin- und Rückübersetzung gedacht ist, verlangsamt das Gastprogramm im Mittel um das 4,3-Fache [24]. Die Ausführung mit Memcheck ist um das 22,2-Fache langsamer.

Ein wichtiges Merkmal des Frameworks ist die Unterstützung von Shadow Memory (Schattenspeicher) [23]. Jeder Speicherbereich, der von einem zu untersuchenden Programm genutzt wird, kann vom Valgrind-Werkzeug einen Shadow Memory zugeordnet bekommen. Dieser Shadow Memory kann vom Werkzeug genutzt werden um Informationen über den Programmspeicher mitzuführen. Das Anlegen und Aktualisieren von Shadow Memory muss das Werkzeug vornehmen. Das Framework stellt lediglich unterstützende Funktionen bereit. Zum Beispiel verfolgt das Standardwerkzeug von Valgrind, Memcheck, den Initialisierungszustand vom ganzen Speicher des Gastprogramms mit Bit-Genauigkeit. Dadurch kann Memcheck Pufferüberläufe selbst von einem Bit erkennen. Valgrind unterstützt auch Schattenregister, womit das Verfahren auch auf CPU-Register anwendbar ist.

Der Shadow Memory muss während der Ausführung des Gastprogramms immer aktuell gehalten werden, was bei parallelen Gastprogrammen Synchronisationsprobleme aufwirft. Die Ausführung einer Instruktion muss sowohl den Speicher aktualisieren, auf dem sie operiert, als auch vom Valgrind-Werkzeug vorgegebene Aktionen auf dem Shadow Memory ausführen. Wenn genau zwischen diesen beiden Aktionen ein Umschalten auf einen anderen Thread durch den Scheduler des Betriebssystems durchgeführt wird,

kann es zu fehlerhaften Verhalten in der Folge kommen, da der Shadow Memory falsche Informationen über den Speicher beinhaltet. In Valgrind wurde dieses Problem pragmatisch gelöst, indem die Threads eines Gastprogramms seriell ausgeführt werden. Mit Hilfe eines sogenannten Biglocks wurde sichergestellt, dass jeweils nur ein Thread ausgeführt wird und niemals mehrere Threads gleichzeitig laufen. Die Serialisierung der Threads hat den gravierenden Nachteil, dass nur ein CPU-Kern ausgelastet wird, selbst wenn das Gastprogramm mehrere nutzen könnte. Bei heutigen Mehrkernsystemen bleibt somit viel Rechenkapazität ungenutzt.

1.3. pValgrind

Ein erster Versuch Gastprogramme unter Valgrind parallel auszuführen, war die wissenschaftliche Arbeit pValgrind [25]. Die Implementierung unterstützt nur die Architektur AMD64. Der Quellcode ist ebenfalls unter der GNU General Public License verfügbar.

Der Valgrind-Kern und einige ausgewählte Werkzeuge wurden mit Hilfe von Locking und dem Umstrukturieren einiger Datenstrukturen threadsicher gemacht. Die Zwischenrepräsentation wurde um atomare Instruktionen erweitert. Damit behalten atomare Instruktionen im Gastprogramm, zum Beispiel zur Synchronisation von mehreren Threads, ihre Eigenschaften nach der Hin- und Rückübersetzung durch LibVEX bei. Neuere Versionen von Valgrind unterstützen atomare Instruktionen inzwischen ebenfalls.

pValgrind kann Gastprogramme mit angepassten Werkzeugen parallel ausführen und dadurch den Overhead deutlich reduzieren. Shadow Memory wurde allerdings nicht umgesetzt, wodurch zum Beispiel das Standardwerkzeug von Valgrind, Memcheck, auf pValgrind nicht verfügbar ist.

2. Synchronisationsverfahren

Bei der Parallelisierung von Programmen kann es zu verschiedenen Anomalien in der Ausführung kommen. Kapitel 2.1 geht genauer auf die Anomalien ein und beschreibt sie anhand eines Beispiels. Synchronisationsverfahren regeln den gemeinsamen Zugriff auf geteilte Ressourcen, um Anomalien bei der parallelen Ausführung zu vermeiden. In den nächsten drei Kapiteln werden die Synchronisationsverfahren Locking, Transactional Memory und Read-Copy-Update vorgestellt. Es werden die verschiedenen Vor- und Nachteile diskutiert und anschließend an zwei Testprogrammen evaluiert.

2.1. Anomalien bei der Parallelisierung

Bei der Parallelisierung wird ein Programm in mehrere Ausführungspfade (Threads) aufgeteilt, welche dann unabhängig voneinander zur gleichen Zeit abgearbeitet werden. Allerdings ist die Ausführung nur echt parallel, wenn die CPU mehrere Kerne zur Verfügung stellt. Die Verteilung der Threads auf die CPU-Kerne übernimmt der Scheduler des Betriebssystems.

Wenn Threads auf den gleichen Speicherbereichen operieren, können bei der parallelen Ausführung verschiedene Anomalien auftreten, die bei seriellen Ausführung nicht entstehen. Ein Data Race liegt vor, wenn ein falsches Ergebnis des Programms oder allgemein ein fehlerhaftes Programmverhalten durch eine nicht beachtete Abarbeitungsreihenfolge der Threads verursacht wird. Eine gemeinsam genutzte Speicherzelle kann nicht parallel beschrieben werden, sondern immer nur nacheinander. Eine veränderte Schreibreihenfolge auf gemeinsamen Speicher kann zu unerwarteten Fehlern führen.

```
1  int global_value = 1;

3  void read_and_update(){
4      int local_value = global_value;


7      //some work
8      local_value *= 2;

10     global_value = local_value;
11 }
```

Listing 2.1: Thread A

```
1  int global_value = 1;

3  void read_and_update(){

6      int local_value = global_value;



11     //some work
12     local_value *= 2;

14     global_value = local_value;
15 }
```

Listing 2.2: Thread B

Listing 2.1 und 2.2 zeigen ein kleines Beispiel mit einem Data Race. Thread A und Thread B führen die gleiche Funktion `read_and_update` aus, allerdings zeitlich leicht ver-

setzt. Thread A liest zuerst die globale und somit für alle Threads verfügbare Variable global_value und speichert den Wert in einer lokalen, für andere Threads nicht sichtbare, Variable local_value ab. Als nächstes führt Thread B genau die gleiche Aktion aus und speichert den Wert ebenfalls lokal in einer Variable ab. Danach bearbeitet Thread A den Rest der Funktion: Die lokale Variable wird verdoppelt und der neue Wert zurück auf die globale Variable geschrieben. Im Anschluss führt auch der Thread B noch die restlichen Instruktionen der Funktionen aus. Beim Zurückschreiben auf die globale, gemeinsame Variable global_value wird dabei der zurückgeschriebene Wert von Thread A überschrieben und die Verdopplung des Werts durch Thread A geht verloren. Diese Art von Data Race nennt man Write/Write Race oder auch Lost Update. Beide Threads konkurrieren um den Schreibzugriff auf die gemeinsame Variable. Es entsteht ein Wettlauf darum, wer als letztes den Wert zurückschreibt und somit die anderen Werte überschreibt.

Die Reihenfolge der Schreib- und Lesezugriffe auf gemeinsamen Speicher kann das Ergebnis des Programms verfälschen. Im oben beschriebenen Beispiel muss sichergestellt werden, dass zwischen dem Lesen des globalen Wertes und dem Zurückschreiben die globale Variable nicht verändert wird. Derartige Bereiche im Programmcode nennt man kritische Abschnitte. Um den Zugriff auf gemeinsamen Speicher zu regeln und kritische Abschnitte abzusichern, wurden verschiedene Synchronisationsverfahren entwickelt, welche im folgenden Kapitel vorgestellt werden.

2.2. Locking

Das bekannteste und am meisten genutzte Verfahren ist Locking [14]. Um einen kritischen Abschnitt zu schützen, wird eine Sperrvariable, Mutex (**mut**ual **ex**clusion) genannt, beim Eintreten gesetzt und beim Austritt freigegeben. Wenn der Mutex gesetzt ist, blockieren alle weiteren Threads, die den kritischen Abschnitt betreten wollen, bis der Mutex wieder freigegeben wird. Damit wird sichergestellt, dass sich maximal ein Thread im kritischen Abschnitt befindet. Listing 2.3 zeigt die Verwendung von Locking mit Funktionen aus der pthread-Bibliothek.

Ein Problem beim Einsatz von Locking sind Verklemmungen. Diese entstehen, wenn für einen kritischen Abschnitt mehr als ein Mutex benötigt wird, weil zum Beispiel zwei eigentlich separate Datenstrukturen für diese spezielle Funktion verändert werden müssen. Es kann immer nur ein Mutex mit einer atomaren Instruktion gesetzt werden. Folglich muss der zweite Mutex B und alle weiteren innerhalb des kritischen Abschnitts vom ersten Mutex A gesetzt werden. Wenn nun ein zweiter Thread Mutex B schon hält und Mutex A als nächsten bekommen möchte, kommt es zu einer Verklemmung, da beide Threads auf den Mutex des jeweils anderen warten. Keiner der beiden gibt seinen Mutex frei, weil beide nicht mit der Bearbeitung des kritischen Abschnitts beginnen können und nur beim Austritt Mutexe freigegeben werden.

Eine einfache Methode um Verklemmungen zu verhindern, ist die Sortierung aller Mutexe und das Setzen nur in gleicher Reihenfolge. Damit würden zwei Threads, die Mutex A und B setzen wollen, dies in der selben Reihenfolge tun und sich nicht gegenseitig behindern. Die Sortierung muss allerdings global im gesamten Programm bekannt sein und beachtet werden. Das hat zur Folge, dass Programmteile nur sehr schlecht modular

```
1  pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int globalcounter=0;
4
5  void increase(){
6      pthread_mutex_lock(&pmutex);
7
8      ++globalcounter;
9
10     pthread_mutex_unlock(&pmutex);
11 }
```

Listing 2.3: Locking mittels der pthread-Bibliothek

in anderen Projekten wiederverwendet werden können.

2.3. Transactional Memory

Transactional Memory [19] (TM) ist ein Konzept zur Ausführung von nebenläufigen Programmen mit gemeinsamen Speicherbereich. Es basiert auf Transaktionen, welche aus dem Bereich Datenbanken bekannt sind und eine Folge von Anweisungen zusammenfassen. Transaktionen erfüllen folgende Eigenschaften: Sie sind atomar und isoliert voneinander. Das Ergebnis einer Transaktion wirkt sich ganz oder gar nicht auf den gemeinsamen Speicher aus. Nebenläufige Transaktionen sehen somit nie Zwischenergebnisse von anderen Transaktionen.

Es gibt verschiedene Arten der Implementierung von Transactional Memory. Software Transactional Memory (STM) ist eine reine Software-Implementierung. Hardware Transactional Memory (HTM) ist eine Implementierung mit Hardwareunterstützung, meist aufbauend auf dem Cache-Kohärenzprotokoll der CPU. Erste Prototypen und Spezifikationen sind ASF [7, 10] und der inzwischen eingestellte Prozessor Rock [13]. Beide Varianten haben Vor- und Nachteile. HTMs besitzen einen wesentlich geringeren Overhead, da sie das Cache-Kohärenzprotokoll nur geringfügig erweitern. Ein schwerwiegender Nachteil bei den meisten HTMs ist die feste maximale Größe einer Transaktion. Größere Transaktionen können von der HTM nicht verarbeitet werden und brechen die Ausführung ab. Bei STMs gibt es Implementierungen, welche unbeschränkt große Transaktionen zulassen und verarbeiten können.[1] Der Overhead ist allerdings erheblich größer als bei HTMs. Um die Vorteile beider auszunutzen gibt es hybride Ansätze (HyTM) [11], welche im normalen Betrieb wie eine HTM operieren, soweit Hardwareunterstützung auf dem System vorhanden ist. Bei der Verarbeitung von großen Transaktionen greift der hybride Ansatz auf eine STM zurück. Damit wird der Geschwindigkeitsvorteil von HTMs bei kleinen Transaktionen ausgenutzt und auch große Transaktionen können verarbeitet werden.

Ein markanter Unterschied zu anderen Synchronisationsverfahren ist die optimistische Ausführung nebenläufiger Transaktionen. Beim Locking wird beim Eintritt in den kritischen Abschnitt versucht ein Mutex exklusiv zu bekommen und es wird erst beim

[1] Einige vorgeschlagene HTMs können ebenfalls unbeschränkt große Transaktionen verarbeiten [8].

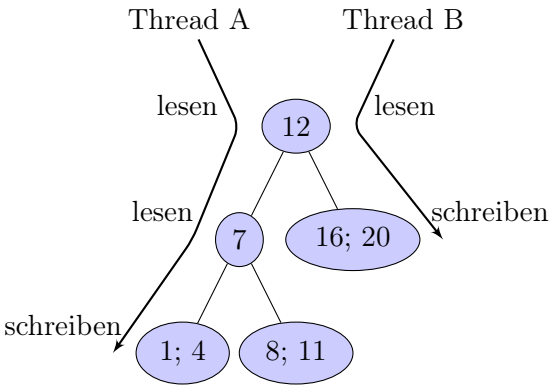

Abbildung 2.1.: Thread A und B schreiben parallel in einen Baum. Es kommt zu keinem Konflikt, da beide Threads auf unterschiedlichen Knoten schreiben. Der Wurzelknoten wird von beiden Threads nur gelesen, aber nicht verändert.

Austritt wieder freigeben. Jeder weitere Thread blockiert beim Eintritt, da er das Mutex nicht bekommen kann, solange ein Thread im kritischen Abschnitt ist. Damit wird sichergestellt, dass sich immer nur ein Thread im kritischen Abschnitt befindet. Dies wird auch pessimistische Ausführung genannt, da man davon ausgeht, dass jeder Thread den anderen beeinflusst.

Bei der optimistischen Ausführung geht man davon aus, dass in großen Datenstrukturen lesende und schreibende Threads nur recht selten an den gleichen Stellen operieren und man somit die Threads auch parallel ausführen kann, ohne dass Daten verloren gehen. Abbildung 2.1 veranschaulicht diesen Sachverhalt anhand zweier schreibender Threads, welche parallel auf einem Baum zugreifen. Da nicht ausgeschlossen werden kann, dass zwei Threads auf den gleichen Daten operieren, müssen Konflikte erkannt und behandelt werden. Dafür werden alle Schreib- und Leseoperationen protokolliert und im Konfliktfall die Auswirkungen der Transaktion rückgängig gemacht. Dieser Vorgang wird Rollback genannt. Nach dem Rollback sieht es so aus als wäre die Transaktion nie gelaufen und die Transaktion wird neu aufgesetzt. In einer ungünstigen Konstellation von einer langen lesenden und vielen kurzen schreibenden Transaktionen muss der Leser gegebenenfalls mehrmals zurückgerollt und neugestartet werden [21].

Solange die Annahme von wenigen Konflikten zutrifft, hat der optimistische Ansatz deutliche Geschwindigkeitsvorteile gegenüber dem pessimistischen, da die volle Nebenläufigkeit ausgenutzt werden kann. Gibt es viele Konflikte, müssen häufig kostenintensive Rollbacks durchgeführt werden, was deutlich langsamer als der pessimistische Ansatz sein kann.

Eine geringe Konfliktrate kann auch beim feingranularen Locking ausgenutzt werden. Es wird nicht die komplette große Datenstruktur mit einem Mutex geschützt, sondern kleinere Teilstrukturen mit mehreren Mutexen. Der Programmieraufwand ist allerdings deutlich größer, da auf Verklemmungen geachtet werden muss. TM macht es dem Programmierer in diesem Fall wesentlich einfacher, da man wegen der optimistischen Aus-

```
1  #define RO                    1
2  #define RW                    0
3  #define START(id, ro)         { stm_tx_attr_t _a = {id, ro}; \
4                                   sigjmp_buf *_e = stm_start(&_a); \
5                                   if (_e != NULL) sigsetjmp(*_e, 0)
6  #define LOAD(addr)            stm_load((stm_word_t *)addr)
7  #define STORE(addr, value)    stm_store((stm_word_t *)addr, (stm_word_t)value)
8  #define COMMIT                stm_commit(); }
9
10 int globalcounter=0;
11
12 void increase(){
13     int localcounter;
14     START(0, RW);
15         localcounter = (int)LOAD(&globalcounter);
16         ++localcounter;
17         STORE(&globalcounter, localcounter);
18     COMMIT;
19 }
```

Listing 2.4: TinySTM

führung auf die Unterteilung in Teilstrukturen verzichten kann und sich Transaktionen auch nicht verklemmen können.

Transaktionen sind auch verschachtelbar und somit modular wiederverwendbar. Eine Funktion, die eine Transaktion beinhaltet, kann problemlos aus einer anderen Transaktion in einem anderen Kontext aufgerufen werden. Die Funktion benötigt dazu keine Anpassung. Beim Locking könnten in diesem Fall Verklemmungen entstehen, da innerhalb eines kritischen Abschnitts ein weiteres Mutex gelockt wird.

Anwendung von TM

Es gibt mehrere freie Bibliotheken, welche meist STM implementieren, zum Beispiel TinySTM [16]. Sie stellen Funktionen zum Kennzeichnen von Transaktionsanfang und -ende bereit. Alle Schreib- und Leseoperationen auf dem gemeinsamen Speicher müssen ebenfalls über Bibliotheksfunktionen durchgeführt werden, da diese Aktionen für den Rollback protokolliert werden. Listing 2.4 zeigt eine kleines Codebeispiel mit TinySTM. Um den Code der Transaktion möglichst leserlich zu halten, wurden die spezifischen TinySTM-Funktionen hinter Makros versteckt.

Um den Aufwand für den Programmierer möglichst gering zu halten, gibt es auch Spracherweiterungen für verschiedene Programmiersprachen, die einen sogenannten Atomic Block [17] als Sprachkonstrukt einführen. Dadurch muss lediglich Transaktionsanfang und -ende gekennzeichnet werden. Um das Protokollieren der Lese- und Schreiboperationen kümmert sich der Compiler. Von Intel stammt eine C/C++-Spracherweiterung [6], welche einen hybriden Ansatz ermöglicht, wenn Hardwareunterstützung verfügbar ist. Es existieren erste Prototypen für den Intel Compiler und einem Entwicklungszweig von GCC [5]. Beide Implementierungen sind noch nicht vollständig und noch nicht für den produktiven Einsatz geeignet. Listing 2.5 zeigt die Benutzung der Spracherweiterung. Es wurde die gleiche Funktion wie bei Listing 2.4 umgesetzt und zeigt schon an diesem sehr einfachen Beispiel eine deutliche Vereinfachung.

```
1 int globalcounter=0;
2
3 void increase(){
4     __transaction{
5         ++globalcounter;
6     }
7 }
```

Listing 2.5: C/C++-Spracherweiterung

Um sicherzustellen, dass die Transaktion sauber zurückrollen kann, überprüft der Compiler die Codeabschnitte und Funktionen, die innerhalb der Transaktion gerufen werden, auf Transaktionssicherheit. Ein Codeabschnitt ist sicher, wenn die Auswirkungen der Ausführung beim Rollback zurückgenommen werden können. Ein Systemaufruf in den Betriebssystemkern ist nicht sicher, da sich der Kern Daten merken könnte, welche nicht zurücknehmbar sind. Ein anschauliches Beispiel ist das Versenden von Netzwerkpaketen. Die gesendeten Daten sind nicht widerrufbar.

Um auch derartige Funktionen innerhalb von Transaktionen zu ermöglichen, bieten sich Irrevocable Transactions [26] an. Das TM-System garantiert, dass Irrevocable Transactions erfolgreich abschließen. Somit wird niemals ein Rollback ausgeführt und zum Beispiel ein Systemaufruf innerhalb der Transaktion nur einmal pro Durchlauf gerufen. Die Überführung einer Transaktion in eine Irrevocable Transaction kann mitten in der Transaktion geschehen. Sie wird entweder durch ein Schlüsselwort ausgelöst, welches der Programmierer explizit eingefügt hat, oder automatisch durch das TM-System direkt vor einem unsicheren Funktionsaufrufs. Schlägt die Überführung fehl, weil zum Beispiel schon ein Konflikt mit einer anderen Irrevocable Transaction besteht, kann die Transaktion noch zurückgesetzt werden, da noch keine unsichere Funktion ausgeführt wurde. In den meisten Implementationen darf nur eine Irrevocable Transaction zur selben Zeit laufen, da ein Konflikt zwischen zwei Irrevocable Transactions nicht auflösbar ist. Um eine gute parallele Ausführung zu gewährleisten, sollten möglichst wenige Irrevocable Transactions verwendet werden.

Debugwerkzeuge und Profiler, um in Transaktionen nach Fehlern und Flaschenhälsen zu suchen, stecken noch in den Anfängen. tm_db [18] ist eine freie Bibliothek, welche den Einbau von Debugschnittstellen in STMs und Debuggern erleichtern und vereinheitlichen soll. Erste Profiler [27] können helfen die Bereiche von Datenstrukturen zu erkennen, welche für die meisten Konflikte der Transaktionen verantwortlich sind.

2.4. Read-Copy-Update

Ein weiteres Synchronisationsverfahren ist Read-Copy-Update (RCU) [22]. Es eignet sich am besten für Datenstrukturen, welche größtenteils parallel gelesen werden, da es mehrere Leser parallel zu einem Schreiber zulässt. Im Gegensatz zum read/write-Lock können während ein Thread schreibend auf die Datenstruktur zugreift, parallel dazu auch weiterhin Threads, die ausschließlich lesen, ausgeführt werden. Um dies sicherzustellen verwaltet RCU zwei Versionen der beschriebenen Daten. Wenn der schreibende

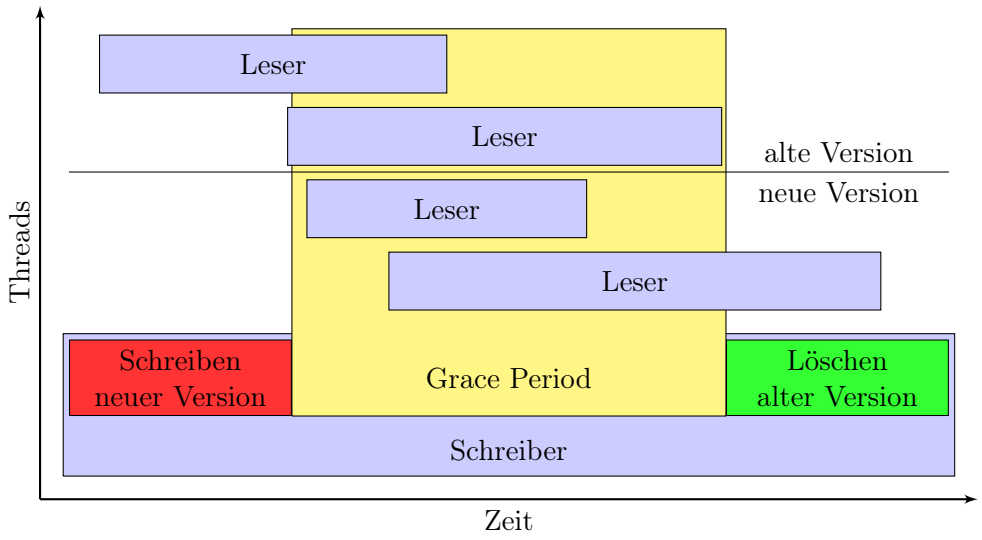

Abbildung 2.2.: Ablauf eines Schreibzugriffs in RCU mit parallelen Lesezugriffen

Thread seine Operationen beendet hat, wartet er einen gewissen Zeitraum (Grace Period), wonach sichergestellt ist, dass kein Leser mehr die alte Version der Datenstruktur verwendet. Abbildung 2.2 illustriert die Grace Period. Wie die Länge der Grace Period ermittelt wird, ist von der Implementation abhängig und von Anwendungsfall zu Anwendungsfall unterschiedlich. Nach der Grace Period kann die alte Version gelöscht werden und der Schreiber den kritischen Abschnitt verlassen. Neue Leser, welche während der Grace Period auf der Datenstruktur lesen, verwenden die neue Version und werden somit von der Löschung der alten Version nicht beeinflusst.

Das Verfahren wird in einigen Betriebssystemkernen verwendet, zum Beispiel Linux, da es dort viele Datenstrukturen gibt, die größtenteils gelesen werden. Ein ähnliches Verfahren ist Multi-Version Concurrency Control (MVCC) [20], welches bei Datenbanksystemen und Transactional Memory Verwendung findet. Ähnlich wie bei RCU legt ein Schreiber eine neue Version eines Datums an, anstelle die alte Version zu überschreiben. In Gegensatz zu RCU sind bei MVCC auch mehr als zwei Versionen (alte und neue) möglich. Das Transaktionssystem der Datenbank stellt sicher, dass beim Lesen immer die aktuellste Version eines Datums verwendet wird.

2.5. Evaluierung von Synchronisationsverfahren

Um ein geeignetes Synchronisationsverfahren für Valgrind zu finden, wurden verschiedene Verfahren an zwei einfachen Testprogrammen ausprobiert. Neben dem Test der Performance stand auch der nötige Aufwand bei der Integration und Verwendung in Valgrind im Mittelpunkt. Neu benötigte Bibliotheken müssen statisch gelinkt werden und sollten nach Möglichkeit keine Abhängigkeit zu der Standardbibliothek libc aufweisen, da Valgrind nicht dagegen gelinkt werden kann. Des Weiteren sollten die Codeanpassung möglichst gering ausfallen.

```
1  volatile int spinlock=0;
2  void spinlock_lock(volatile int *spinlock){
3      asm volatile(
4      "    movl $1, %%eax\n"
5      "1:\n"
6      "    xchg %%eax, (%0)\n"
7      "    test %%eax, %%eax\n"
8      "    jnz 1b\n"    // jump backwards to 1: local label
9      :                 // output, nothing, dereferenced and written above
10     : "r"(spinlock)   // input
11     : "%eax"          // used registers (clobbered)
12     );
13 }
14
15 void spinlock_unlock(volatile int *spinlock){
16     asm volatile(
17     "    movl $0, %%eax\n"
18     "    xchg %%eax, (%0)\n"
19     :
20     : "r"(spinlock)
21     : "%eax"
22     );
23 }
```

Listing 2.6: Spinlock-Implementierung

Das erste Testprogramm LinkedList ist eine einfach verkettete Liste, in welche Zufallszahlen sortiert eingefügt werden. Parallel zum Einfügen wird auch noch lesend auf die Datenstruktur zugegriffen. Es wird geprüft, ob eine zufällig gewählte Zahl in der Liste enthalten ist. Als zweite Datenstruktur zum Testen wurde ein B-Baum gewählt. In diesen werden ebenfalls parallel Zufallszahlen eingefügt und geprüft, ob zufällig gewählte Werte enthalten sind.

In beiden Testprogrammen wurde grobgranulares Locking verwendet, d.h. der komplette Zugriff auf die Datenstruktur wird in einem Lock bzw. einer Transaktion ausgeführt. Die Datenstruktur wurde nicht in separat zu behandelnde Teilstrukturen aufgeteilt. Damit soll der Programmieraufwand für den Entwickler vergleichbar klein gehalten werden und nicht auf Besonderheiten der einzelnen Synchronisationsverfahren eingegangen werden.

Es wurden verschiedene Locking- und TM-Systeme eingesetzt: Ein Spinlock, ein Mutex mit Hilfe von Futexes und ein Mutex mittels der Bibliothek pthread, ein Read/Write-Lock der pthread-Bibliothek, RCU mit Hilfe der Bibliothek userspace-rcu und Transaktionen mittels der STM-Bibliothek TinySTM und den C/C++-Spracherweiterung im Intel Compiler und GCC.

Eine sehr einfache Spinlock-Implementierung wurde in Inlineassembler umgesetzt. Sie blockiert nie, sondern fragt ständig nach, ob das Lock jetzt frei ist. Die Einfachheit der Implementation ist gut geeignet für eine Integration in Valgrind. Listing 2.6 zeigt die verwendeten Funktionen. pValgrind benutzt eine sehr ähnliche Spinlock-Implementierung als Ersatz für das Biglock.

Ein weiterer Lockingmechanismus ist das vom Linux-Kern bereitgestellte futex [15]. Es verwendet lediglich einen Systemaufruf von Linux und einige atomare Instruktionen. Da

futex keine weiteren Abhängigkeiten besitzt, ist es recht einfach in Valgrind integrierbar. Es ist allerdings nur auf Linux verfügbar und somit nicht auf andere Betriebssysteme portierbar.

Als Vergleich zum futex wurde ebenfalls noch das normale Mutex der pthread-Bibliothek, im Folgenden als pmutex bezeichnet, verwendet. Pthread ist die Standardbibliothek für Threads unter Linux und wird auch zum Anlegen der Threads in den Testprogrammen verwendet. Valgrind wird statisch gelinkt und besitzt nur minimale Unterstützung für libc-Funktionen. Die Nutzung von neuen Bibliotheksfunktionen ist somit nur eingeschränkt möglich.

Ein weiteres Lock aus der pthread-Bibliothek ist das Read-Write Lock (rwlock). Es ermöglicht mehreren Lesern parallel auf der Datenstruktur zu operieren. Wenn ein Schreiber den kritischen Abschnitt betreten möchte, blockiert er so lange bis alle momentan laufenden Leser ihr Lock freigegeben haben. Neue Leser blockieren ebenfalls schon beim Eintritt. Danach operiert der Schreiber exklusiv auf der Datenstruktur, kein anderer Schreiber oder Leser läuft parallel. Erst nach Austritt des Schreibers können wieder mehrere Leser parallel ausgeführt werden.

Das RCU-Verfahren, in Kapitel 2.4 erläutert, wurde mit Hilfe der Bibliothek userspace-rcu (urcu) [12] in der Version 0.5.4 angewandt. Die Integration in Valgrind ist ähnlich wie mit pthread problematisch.

Transactional Memory wurde in Form von TinySTM (tinystm) [16], einer STM-Bibliothek, in Version 1.0, den 4. Prototyp des Intel C Compilers mit TM-Unterstützung (intelstm) [6] und einem Entwicklungszweig des GCC mit TM-Unterstützung (gcctm) [5] in Revision 171386 verwendet. Die Anwendung dieser TM-Systeme wurde in Kapitel 2.3 diskutiert.

Das Testsystem, welches die Benchmarks ausführte, besaß zwei Intel Xeon X5650 CPUs mit jeweils 6 echten Kernen und Hyperthreading. Somit wurden dem Betriebssystem 24 Kerne angezeigt. Alle Testprogramme wurden mit GCC 4.6 auf 2. Optimierungsstufe gebaut und gegen glibc 2.13 gelinkt. Ausnahmen waren die TM-Compiler gcctm und intelstm, welche ihre Testprogramme selbst kompilierten.

Abbildung 2.3 zeigt vergleichend die Ergebnisse von LinkedList für TM-Systeme und Lockingmechanismen. Die TM-Systeme tinystm und intelstm sind deutlich langsamer als der Lockingmechanismus futex und das RCU-Verfahren. Die Ursache dafür ist eine sehr hohe Konfliktrate. Alle Leser und Schreiber fangen vom Kopf der Liste an und traversieren dann immer zum nächsten Element. Damit werden die ersten Elemente von fast allen Lesern und Schreibern gelesen. Fügt nun ein Schreiber relativ weit vorn ein Element ein und beendet erfolgreich, müssen alle Leser und Schreiber, die darüber hinaus gelesen haben, abbrechen und zurückrollen. Ein Rollback ist sehr kostenintensiv. Die Kurve von tinystm zeigt dies sehr deutlich. Je mehr Threads parallel ausgeführt werden, desto höher ist die Konfliktrate und damit auch die Ausführungszeit.

Die weiteren Ergebnisse werden in Abbildung 2.4 und 2.5 dargestellt, getrennt in TM-Systeme und Lockingmechanismen. RCU und Spinlock schneiden bei den Lockingmechanismen am besten ab. Ursachen dafür ist der sehr kurze kritische Abschnitt. Dadurch lohnt es sich per Busy Waiting auf das Spinlock zu warten. RCU lässt immer beliebig viele Leser parallel zu einem Schreiber zu. Dadurch können sehr zügig alle Leser auf noch relative kurzen Listen abgearbeitet werden und danach nur noch die Schreiber fertig wer-

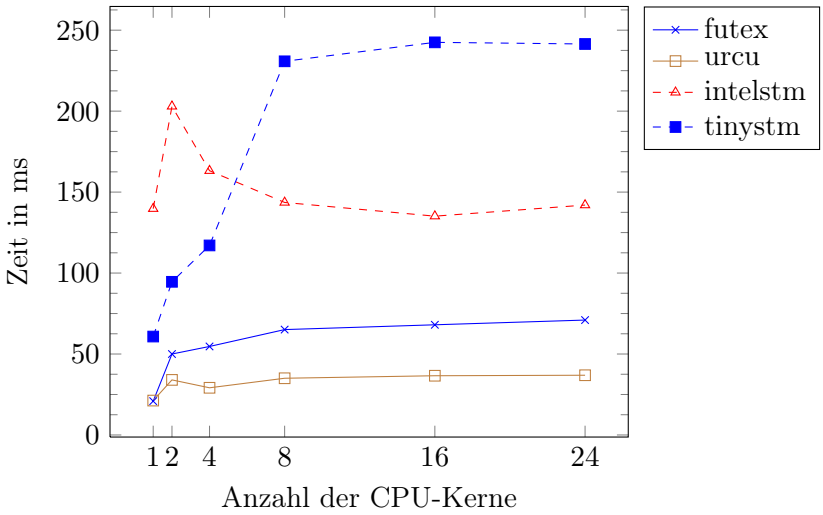

Abbildung 2.3.: Benchmark von LinkedList

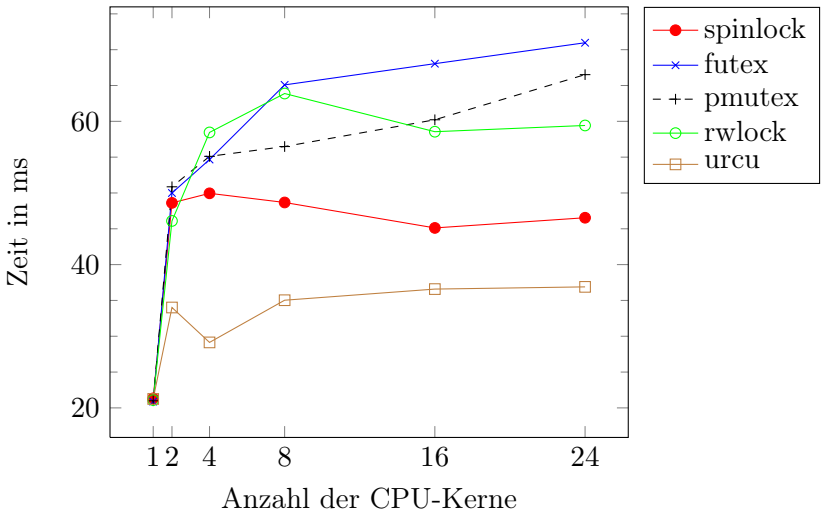

Abbildung 2.4.: Benchmark der Lockingmechanismen mit dem Testprogramm LinkedList

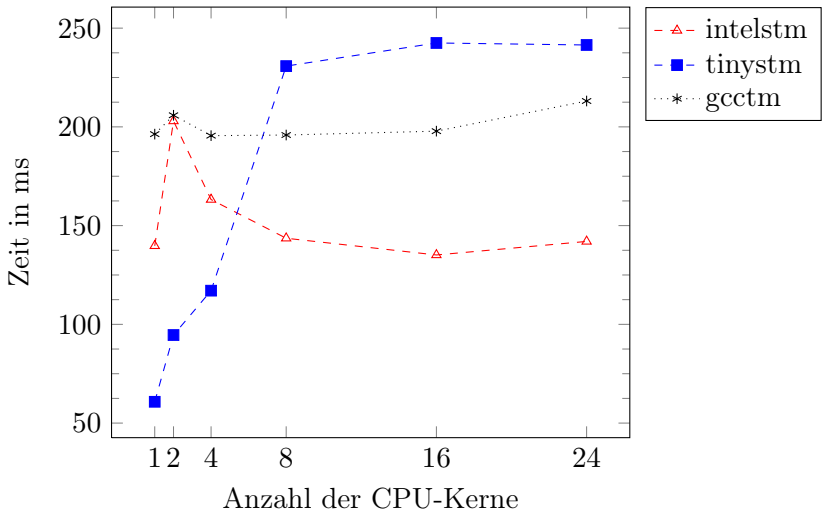

Abbildung 2.5.: Benchmark der TM-Systeme mit dem Testprogramm LinkedList

den. Beim Read-Write Lock können zwar auch mehrere Leser gleichzeitig laufen, doch dazu kommt es nur recht selten. Sobald ein Schreiber auf das rwlock wartet, blockieren auch alle weiteren Leser. Da dieses Testprogramm fast ausschließlich aus den Zugriff auf die Listenstruktur besteht, warten fast immer Schreiber auf das rwlock und es kommt nur selten zum parallelen Ablauf der Leser.

B-Baum

Die Datenstruktur des B-Baums ist wesentlich besser geeignet für Transactional Memory. Die Konfliktrate ist deutlich geringer, da sich ein B-Baum sehr schnell verzweigt und die Daten in unterschiedlichen Teilbäumen abgelegt sind. Diese Teilstrukturen können unabhängig voneinander verändert werden ohne das es dabei zu Konflikten kommt. Die optimistische Ausführung der TM-Systeme nutzt diese Parallelität aus. Abbildung 2.6 zeigt, dass die TM-Systeme tinystm und intelstm mit zunehmender Anzahl an CPUs schneller werden. Durch die Verwendung von grobgranularen Locking können die Lockingmechanismen die Datenparallelität nicht ausnutzen und es kommt zu einem massiven Wettstreit um die Sperrvariable, der mit zunehmender Anzahl an CPUs immer größeren Overhead erzeugt. Feingranulares Locking könnte die Datenparallelität nutzbar machen, ist allerdings mit sehr hohen Programmieraufwand verbunden. Es wurde hier mit Absicht nicht betrachtet um den Programmieraufwand für Transactional Memory und Locking vergleichbar zu gestalten.

Abbildung 2.7 vergleicht die restlichen Lockingmechanismen. gcctm, welches den selben Code wie intelstm verwendet, hatte Ausführungsprobleme und verfing sich nach einer gewissen Zeit in einer Endlosschleife auf einer CPU. Die Entwicklung von der TM-Unterstützung im GCC-Compiler ist noch nicht abgeschlossen.

Die beiden sehr einfachen Beispiele zeigen, dass jedes Synchronisationsverfahren seine Vor- und Nachteile hat. Es muss im Einzelfall entschieden werden, welches die beste

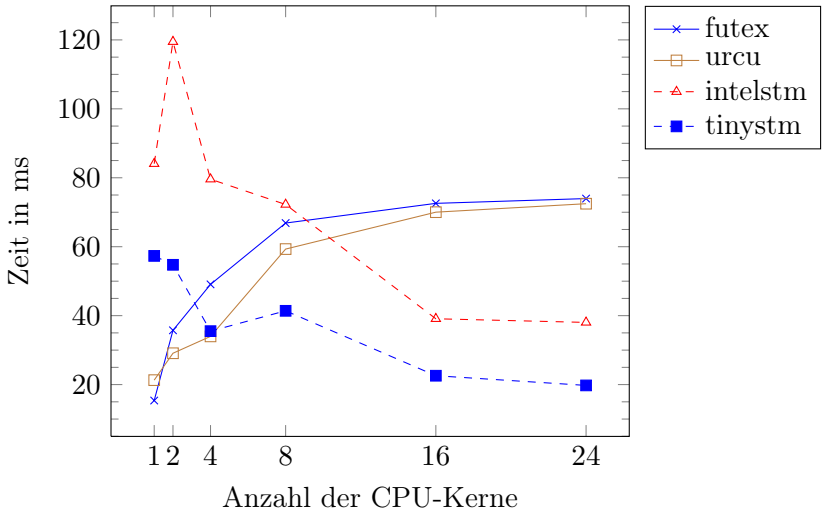

Abbildung 2.6.: Benchmark der Lockingmechanismen auf den B-Baums

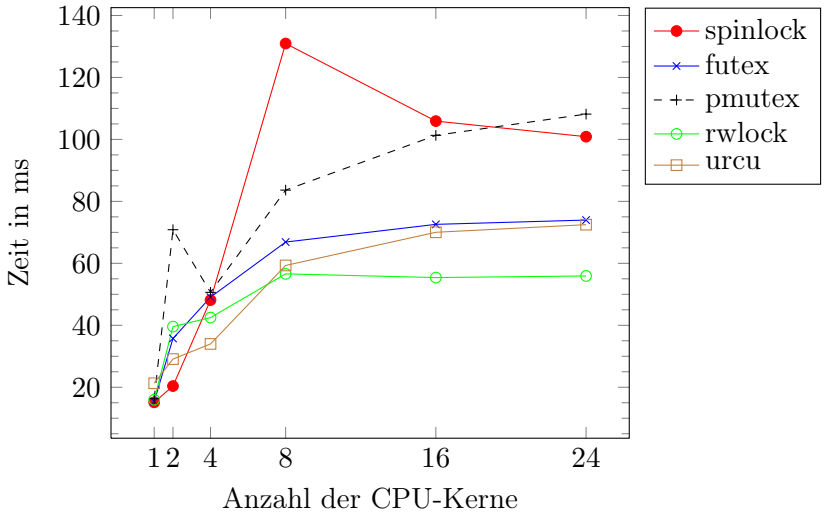

Abbildung 2.7.: Benchmark des B-Baums

Lösung ist. Transactional Memory ist gut geeignet für große, schreiblastige, nicht partitionierbare Datenstrukturen [21], da es für diesen Anwendungsfall kein anderes Synchronisationsverfahren anbietet. Im Allgemeinen sollte die Konfliktrate möglichst gering sein. Für Szenarien mit hoher Konfliktrate bietet sich Locking an.

3. Parallelisierung von Valgrind

Kapitel 3.1 beschreibt die zentralen Komponenten, die bei der Ausführung von Gastcode in Valgrind eine Rolle spielen. Dabei wird das unveränderte, serielle Valgrind betrachtet. Die einzelnen Veränderungen, die im Zuge dieser Arbeit entstanden sind, um parallele Ausführung von Gastcode zu ermöglichen, werden in den anschließenden zwei Kapiteln an den betreffenden Komponenten erläutert.

3.1. Ausführung von Gastcode im seriellen Valgrind

Valgrind serialisiert die Ausführung von Threads. Dazu wird ein sogenanntes Biglock eingesetzt. Nur der Thread, der dieses Biglock bekommen hat, wird ausgeführt, alle anderen blockieren. Um sicherzustellen, dass ein Thread das Biglock auch wieder freigibt, werden die ausgeführten Codeblöcke (Basic Block, BB) mitgezählt. Am Anfang der Ausführung eines Threads weist der Scheduler ihm ein Kontingent an Codeblöcken zu. Dieses gibt die maximale Anzahl an Codeblöcken an, welche noch ausgeführt werden dürfen. Ist das Kontingent erschöpft, muss der Thread das Biglock freigeben. Damit wird sichergestellt, dass ein Thread nicht alle anderen Threads aushungern kann. Valgrind verwendet die Anzahl von ausgeführten Codeblöcken anstelle von Zeitscheiben, welche sonst bei Schedulern üblich sind.

Das Biglock wurde mittels einer Pipe implementiert. Bei der Initialisierung wird die Pipe erzeugt und mit einem Zeichen beschrieben. Die Funktion Lock[1] versucht von der Pipe das Zeichen mit Hilfe des Systemrufs read zu lesen. Ist das Biglock noch frei, das Zeichen somit in der Pipe, kehrt der Systemruf erfolgreich zurück und liefert das Zeichen aus der Pipe, welche danach leer ist. Ist das Biglock schon belegt, die Pipe also leer, blockiert der Systemruf solange im Betriebssystemkern bis wieder ein Zeichen in der Pipe zum Lesen bereit ist. Die Unlock-Funktion schreibt ein Zeichen zurück in die Pipe und gibt somit das Biglock wieder frei.

Hat ein Thread das Biglock bekommen, kann er mit der Ausführung von Gastcode beginnen. Abbildung 3.1 stellt den Ablauf der Ausführung innerhalb von Valgrind schematisch dar. Als erstes weist der Scheduler dem ausführenden Thread einen BB-Zähler zu (Basic Block) und ruft anschließend den Dispatcher auf. Der BB-Zähler sorgt dafür, dass das Kontingent an Codeblöcken nicht überschritten wird. Er ist die Anzahl an Codeblöcken, die der Thread noch ausführen kann, bevor der Thread wieder in den in den Scheduler zurückkehrt und das Biglock abgeben muss.

Der Dispatcher ist ein in Assembler geschriebener Codeabschnitt, welcher die Ausführung des Gastprogramms steuert. Er prüft zuerst, ob er laut BB-Zähler noch weitere

[1] Das Interface für das Biglock ist einem Semaphor nachempfunden (sema_down, sema_up, ...), obwohl das Biglock ein Mutex ist. Hier werden der Einfachheit halber Funktionsnamen, die üblich bei Mutexen sind, verwendet.

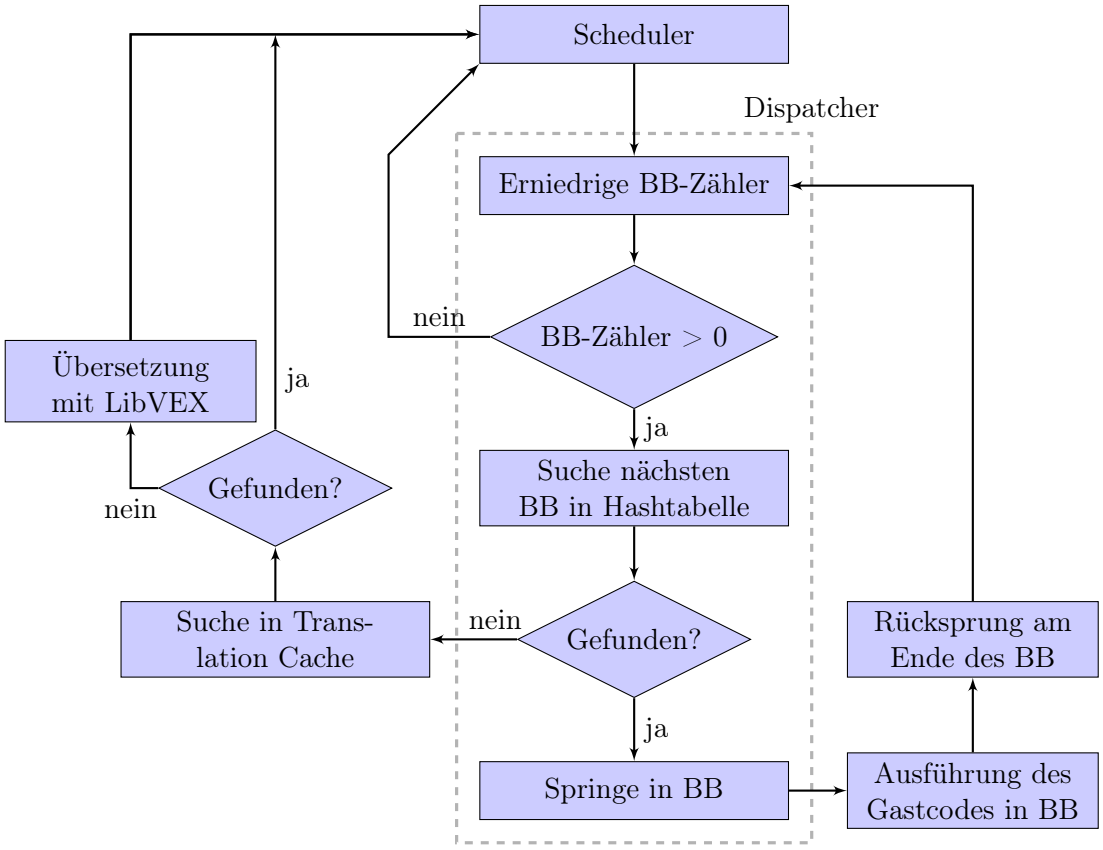

Abbildung 3.1.: Ablauf der Ausführung von instrumentierten Code in Valgrind

Codeblöcke ausführen darf. Wenn dem so ist, sucht er sich aus den Translation Cache den als nächstes auszuführenden Codeblock heraus und springt per Sprungbefehl in den instrumentierten Maschinencode hinein. Damit wird dieser Codeblock nativ ausgeführt. Am Ende des Codeblocks wurden spezielle Instruktionen von LibVEX bei der Übersetzung eingefügt um wieder zurück in den Dispatcher zu springen und die Ausführung des Gastes zu unterbrechen. Der Dispatcher sucht dann den nächsten Basic Block aus dem Translation Cache und springt diesen an.

Der Translation Cache ist eine zentrale Datenstruktur innerhalb von Valgrind, welche die Ergebnisse der Übersetzung durch LibVEX speichert und sie für eine spätere Wiederverwendung bereithält. Es wird der instrumentierte Maschinencode gespeichert, damit mehrmaliges Übersetzen vermieden wird. Diese Datenstruktur ist global und nicht threadsicher. Sie wird zur Laufzeit befüllt, da LibVEX erst übersetzt, wenn der Codeabschnitt verwendet werden soll.

Um die Suche im Translation Cache zu beschleunigen, verwendet der Dispatcher eine Hashtabelle, welche wie der Translation Cache ebenfalls global ist. Sie ordnet einem Codeblock im Gastprogramm den instrumentierten Maschinencode im Translation Cache zu. Falls die Suche in der Hashtabelle fehlschlägt, kehrt der Dispatcher in die ihn aufrufende C-Funktion zurück, welche dann den kompletten Translation Cache durchsucht. Wenn auch diese Suche fehlschlägt, wird eine Übersetzung durch LibVEX gestartet. Die Hashtabelle fungiert somit als eine Art Cache für häufig verwendete Codeblöcke. Sie enthält allerdings nur Zeiger in den Translation Cache und keine Kopie des instrumentierten Codes.

Um parallele Ausführung von Gastcode zu ermöglichen, wurden problematische Datenstrukturen wie das Biglock und der Translation Cache verändert. Die nächsten beiden Kapitel befassen sich mit den einzelnen Änderungen an den beiden Strukturen.

3.2. Veränderungen am Biglock

Die Serialisierung durch das Biglock wurde aufgelöst. Beim Betreten des Dispatchers, welcher dann in den Gastcode springt und ihn ausführt, wird das Biglock freigegeben. Damit können mehrere Threads gleichzeitig Gastcode ausführen. Erst beim Verlassen des Dispatchers wird das Biglock wieder angefordert. Das Biglock wurde nicht komplett aufgelöst, da es noch viele interne Datenstrukturen gibt, die nicht threadsicher sind, insbesondere der Address Space Manager. Er verwaltet, welcher Speicherbereich vom Werkzeug, dem Gastprogramm und den zentralen Komponenten von Valgrind selbst verwendet werden, und stellt sicher, dass sich diese unterschiedlichen Bereiche nicht überschneiden.

Des Weiteren wurde versucht die Biglock-Implementierung auf ein futex umzustellen in der Hoffnung einen leichten Performancegewinn zu erzeugen. Das Interface wurde dabei nicht verändert. Es konnte allerdings keine nennenswerte Änderung bei der Ausführung festgestellt werden, weswegen die alte Implementierung wiederhergestellt wurde, um die Änderungen am Quelltext von Valgrind möglichst klein zu halten.

Die Biglock-Implementierung wird nun außerdem auch noch an anderen Stellen als allgemeine Implementation für Mutexe verwendet. Bei der Parallelisierung mussten ei-

nige Datenstrukturen mit einen Mutex abgesichert werden wie die Liste zu löschender Einträge im Translation Cache. Das verwendete Verfahren wird im nächsten Abschnitt erläutert.

3.3. Veränderungen am Translation Cache

Da der Translation Cache und die Hashtabelle global sind, können Probleme bei der parallelen Ausführung von Gastcode entstehen. Ein Thread A löscht einen Eintrag im Translation Cache während ein anderer Thread B genau auf diesem Codeabschnitt operiert. Holt Thread B danach die als nächstes auszuführende Instruktion aus den Speicher, kann dort inzwischen eine völlig andere Instruktion eines anderen Basic Blocks stehen. Es kommt zu Fehlverhalten des Gastprogramms. Des Weiteren ist das parallele Einbringen von neuen Übersetzungen problematisch.

Als erste Änderung wurde die globale Hashtabelle aufgelöst und jedem Thread eine eigene, lokale Hashtabelle zugeordnet. Die Ausführungspfade der Threads können völlig unterschiedlich aussehen, weshalb sie auch unterschiedliche Codeblöcke abarbeiten. Ein lokaler Cache wie die Hashtabelle verhindert Kollisionen mit anderen Threads und beschleunigt die Ausführung, da weniger Cache-Misses[2] auftreten.

Durch diese Änderung können mehrere Threads parallel im Gastcode und Dispatcher laufen. Jeder Thread besitzt eine lokale Hashtabelle und greift nur lesend auf den Gastcode, der weiterhin im globalen Translation Cache gespeichert ist, zu. Änderungen am Translation Cache wie das Hinzufügen und Löschen von Einträgen wurden durch ein Mutex abgesichert. Das Löschen bedarf dabei noch einer weiteren besonderen Behandlung, da nur Einträge freigegeben werden dürfen auf denen kein Thread mehr läuft.

Die naive Methode mittels Reference Counting festzustellen, wann ein Eintrag verwendet wird, scheitert am zu hohen Aufwand für das Mitführen der Zähler. Der Dispatcher ist ein performancekritischer Abschnitt. Ein Basic Block besteht meist aus 5 bis 30 Instruktionen. Die innere Schleife des Dispatchers benötigt 14 Instruktionen. Für Reference Counting müssten weitere Instruktionen hinzugefügt werden. Insbesondere für das Ein- und Austreten aus einem Basic Block müsste ein Zähler zum Beispiel mittels atomaren Instruktionen inkrementiert und dekrementiert werden. Atomare Instruktionen sind allerdings im Vergleich zu anderen Instruktionen recht teuer und da diese Befehle sehr häufig durchlaufen werden, wird die Ausführung gebremst. Eine prototypische Implementation war selbst bei paralleler Ausführung in 4 Threads langsamer als das unveränderte, serielle Valgrind.

Eine andere Methode lässt den Dispatcher unverändert und funktioniert ähnlich wie RCU [22]. Es wurde in pValgrind [25] vorgestellt. Die Löschung eines Eintrags wird solange verzögert, bis sicher kein Thread mehr diesen Eintrag benutzt. Dazu wird der zu löschende Eintrag im Translation Cache als löschend markiert und aus den Hashtabellen der Threads entfernt. Ein Thread, der nun versucht diesen Codeblock auszuführen, würde ihn nicht im Translation Cache finden und eine neue Übersetzung anstoßen. Danach wird der Eintrag in eine Liste zu löschender Einträge zusammen mit einer Bitliste eingefügt. Die Bitliste gibt an, welche Threads zu diesem Zeitpunkt ausführungsbereit waren

[2] der gesuchte Eintrag wurde im Cache nicht gefunden

und somit potentiell gerade diesen Eintrag benutzen. Wenn ein Thread den Dispatcher verlässt, schaltet er in jeder Bitliste sein Bit aus. Falls eine Bitliste komplett Null ist, wird der entsprechende Eintrag endgültig gelöscht.

Ein Spezialfall, den es zu beachten gilt, ist das Recyceln von Sektoren im Translation Cache. Der Translation Cache ist in eine feste Anzahl von Sektoren fester Größe aufgeteilt, die nach und nach befüllt und erst beim ersten Benutzen alloziert werden. Wenn ein Sektor voll ist, wird auf den nächsten Sektor zurückgegriffen. Neue Übersetzungen werden anschließend dort hinein gespeichert. Ist kein unbenutzter Sektor verfügbar, wird der älteste befüllte Sektor recycelt. Beim Recyceln werden alle Einträge im Sektor gelöscht, auch solche die möglicherweise noch von anderen Threads benutzt werden. Es wird lediglich der allozierte Speicher wiederverwendet. Der Inhalt wird verworfen. Im Unterschied zur Löschung einzelner Einträge muss hier der aufrufende Thread, der das Recyceln veranlasst, solange blockieren bis kein Thread mehr auf den zu löschenden Einträgen arbeitet. Dies ist notwendig, weil der aufrufende Thread einen neuen Eintrag einbringen will und dafür Platz schaffen muss. Er kann somit nicht weiterlaufen.

All diese Änderung bewirken, dass der Kernbereich, der Dispatcher, und somit auch der Gastcode, parallel ausgeführt werden kann. Wird der Dispatcher allerdings verlassen um zum Beispiel eine neue Übersetzung in den Translation Cache einzufügen, serialisiert das Biglock weiterhin den Zugriff auf solche globale Datenstrukturen wie den Translation Cache.

4. Leistungsbewertung

Um das Ergebnis der Arbeit zu bewerten, wurden mehrere Performancetests durchgeführt. Dabei wurde die gleiche Benchmarksuite NAS Parallel Benchmarks (NPB) [9] verwendet wie bei der Leistungsbewertung von pValgrind [25]. NPB wurde in Version 3.3.1 in der OpenMP-Variante verwendet und mit gfortran in Version 4.6.2 mit den voreingestellten Flags gebaut.

Jeder Benchmark von NPB in der Klasse W wurde dreimal durchgeführt und die Ergebnisse arithmetisch gemittelt. Die Klasse des Benchmarks gibt die Parameter vor, mit denen der Benchmark sowohl kompiliert als auch ausgeführt wird. Der Benchmark lu musste ausgelassen werden, da er bei der Ausführung mit dem unveränderten, seriellen Valgrind kein Ende fand und selbst nach Stunden nicht durchgelaufen war. Das parallelisierte Valgrind lief hingegen problemlos durch. Eine genaue Ursachenforschung, welche Bestandteile des Benchmarks dieses Verhalten hervorrufen, wurde nicht durchgeführt.

Das selbe Testsystem wie in Kapitel 2.5 kam zum Einsatz. Hyperthreading wurde allerdings deaktiviert um starke Schwankungen bei den Ergebnissen zu vermeiden, welche mit Hyperthreading auftraten.

Die Ausführungszeit der Benchmarks unter dem parallelisierten und dem seriellen Valgrind werden in den folgenden Graphen in Form des Speedups miteinander verglichen. Als serielles Valgrind kam das unveränderte Valgrind mit der Revisionsnummer 11510 zum Einsatz. Diese Revision war auch der Zeitpunkt der Abspaltung um die hier vorgestellte parallelisierte Fassung von Valgrind zu erstellen. Beide Versionen wurden mit den gleichen GCC 4.6.2 und gleichen Flags gebaut. Bei der Messung wurde das Werkzeug Nulgrind eingesetzt und somit keine Instrumentierung durchgeführt.

Die Abbildungen 4.1 und 4.2 zeigen den erreichten Speedup des parallelen Valgrind gegenüber dem seriellen Valgrind. Dabei verhalten sich die einzelnen Benchmarks von NPB sehr unterschiedlich. Abbildung 4.1 zeigt dabei noch recht gutartige Ergebnisse. Die Benchmarks bt und dc skalieren mit der wachsenden Anzahl an Threads und der Speedup wird immer größer. Dies zeigt, dass die Parallelisierung von Valgrind erfolgreich die Ressourcen von Mehrkernsystemen ausnutzen kann.

Der Benchmark ft verhält sich bis 8 Threads ähnlich, fällt dann allerdings bei 12 Threads stark ab. Die Ursache dafür ist eine Verlangsamung der Ausführung im parallelen Valgrind mit 12 Threads wie Abbildung A.5 zeigt. Dieses Verhalten tritt auch bei den Benchmarks cg, mg, sp und ua auf. Alle diese Benchmarks lasten das Testsystems vollständig aus, indem so viele Threads wie verfügbare CPU-Kerne verwendet werden.

Ein weiteres markantes Verhalten ist der sehr große Speedup bei 2 Threads, besonders bei den Benchmarks in Abbildung 4.2 zu beobachten, und das starke Abfallen danach bei 4 Threads. Die Ursache dafür ist ein merkwürdiges Verhalten des seriellen Valgrinds bei diesen speziellen Benchmarks. Abbildungen A.2, A.9 und A.7 zeigen, dass das serielle Valgrind bei 2 Threads deutlich langsamer ist als mit einem Thread oder mehr als 2

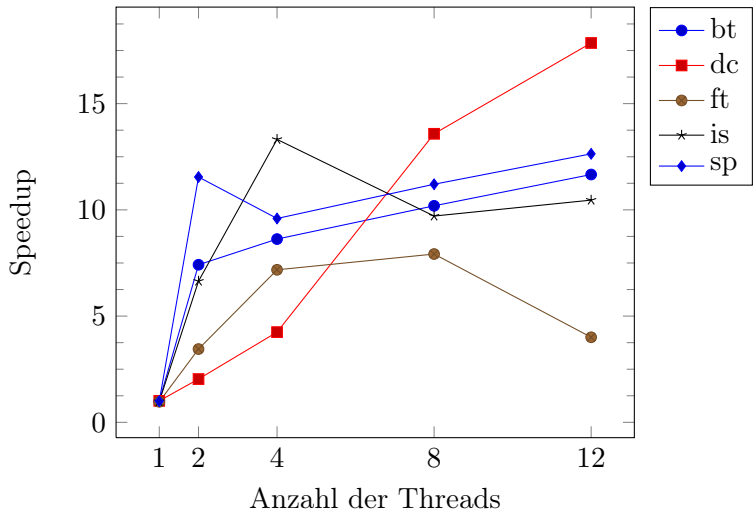

Abbildung 4.1.: Benchmarks von Valgrind

Threads. Dieses Verhalten ist recht erstaunlich, da das serielle Valgrind die Threadausführung serialisiert. Es kann immer nur ein Thread des Programms ausgeführt werden. Dadurch sollte eine Erhöhung der Threadanzahl, wie von 2 auf 4 Threads, keine Performanceverbesserung nach sich ziehen.

Eine mögliche Ursache dafür könnte sein, dass die OpenMP-Implementation adaptiv je nach Anzahl der Threads das verwendete Mutex wählt und bei 2 Threads auf ein Spinlock zurückgreift. Die von der GNU OpenMP-Implementierung libgomp verwendet Thread-Bibliothek pthread unterstützt neben reinen Spinlocks auch Mutexe, die für eine kurze Zeit Spinning ausführen und erst danach blockieren. Da aber nur ein Thread laufen kann, ist dieser Anteil von Spinning völlig kontraproduktiv und verbraucht nur Rechenzeit. Das parallele Valgrind verhält sich erwartungsgemäß und wird mit steigender Anzahl von Threads immer schneller.

Ein Nachweis der obigen These konnte allerdings nicht erbracht werden. Eine manuell instrumentierte Fassung von der pthread-Bibliothek konnte nicht nachweisen, dass Spinning verwendet wurde. Eine genaue Untersuchung des Zusammenspiels von der OpenMP-Implementierung von GCC libgomp und der pthread-Bibliothek wurde nicht durchgeführt.

Der Benchmark ep zeigt, dass die Parallelisierung von Valgrind noch verbesserungsbedürftig ist. Das serielle Valgrind ist immer deutlich schneller als das parallele Valgrind wie Abbildung A.4 verdeutlicht. Dies ist noch ein offenes Problem.

Die Messungen zeigen, dass durch die Parallelisierung von Valgrind ein teils deutlicher Performancegewinn erzielt wurde.

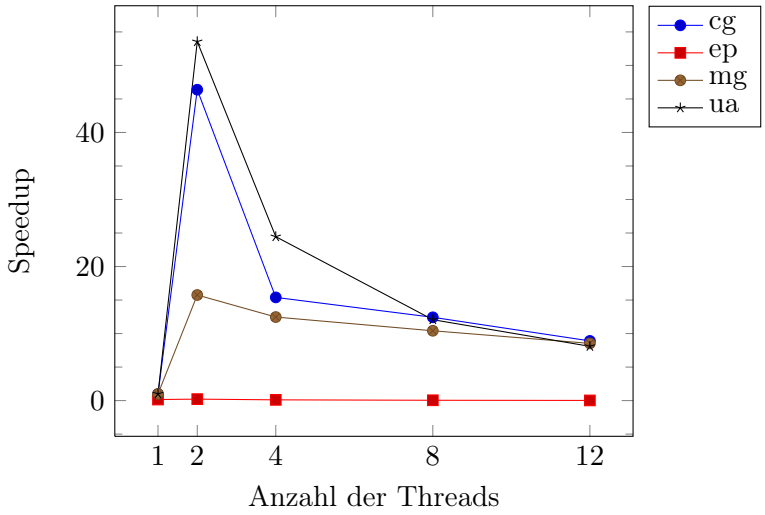

Abbildung 4.2.: Benchmark von Valgrind

5. Parallelisierung von Shadow Memory

In diesem Kapitel wird die Implementation von Shadow Memory in Valgrind und seinen Werkzeugen vorgestellt und ein Verfahren zur Parallelisierung diskutiert. Die Implementation dieser Verfahren ist nicht Teil der Arbeit.

Der Aufbau und die Nutzung des Shadow Memorys ist vom jeweiligen Werkzeug abhängig. Welche Daten sollen gesammelt und wann müssen diese aktualisiert werden? Mit wie vielen Bits wird ein Byte Speicher im Shadow Memory repräsentiert? Diese Dinge sind von Werkzeug zu Werkzeug unterschiedlich. Deswegen überlässt Valgrind die Implementation des Shadow Memorys den Werkzeugen. Valgrind stellt lediglich die dafür notwendigen Funktionen bereit. Werkzeuge können zum Beispiel Bibliotheksfunktionen mit eigenen überschreiben. Memcheck nutzt das, um die Funktionen malloc/new/new[] zu überschreiben. Die neue Funktion alloziert neben den normalen Daten auch den dazugehörigen Shadow Memory. Bei der Instrumentierung des Gastcodes werden dann verschiedene Instruktionen eingebaut um den Shadow Memory aktuell zu halten.

Eine einfache und performante Implementierung von Shadow Memory ist half-and-half [23]. Der gesamte Speicher des Systems wird in zwei feste Bereiche unterteilt. Der erste Bereich ist für die normalen Daten des Gastprogramms, der zweite Bereich ist Shadow Memory. Da der Speicher fest aufgeteilt ist, reicht eine einfache Adressverschiebung aus um von einer Speicherzelle zu deren Shadow Value zu gelangen. Im einfachsten Fall wird jedes Byte im Speicher von einem Byte im Shadow Memory repräsentiert. Optimierte Werkzeuge benötigen allerdings nur wenige Bits pro Byte und schrumpfen dabei den festen Bereich des Shadow Memorys. Sie verwenden dann eine skalierte Adressverschiebung.

Der Vorteil bei diesem Verfahren ist der sehr geringe Aufwand, zu einem Byte im Speicher den passenden Shadow Value zu finden. Der Overhead ist damit sehr gering. Ein schwerwiegender Nachteil ist, dass einige Programme damit nicht funktionieren. Da der Shadow Memory fest und zusammenhängend im Adressraum liegt, kann das Gastprogramm niemals Daten in diesem Bereich ablegen. Gastprogramme, die dies benötigen, können somit nicht ausgeführt werden.

Um möglichst mit vielen Programmen kompatibel zu sein, teilt Memcheck den Speicher nicht in feste Bereiche, sondern legt während der Laufzeit Blöcke von Shadow Memory an. Sie liegen möglichst weit entfernt von Daten des Gastprogramms. Die Blöcke sind in einer zweistufigen Liste organisiert. Die erste Stufe, primary map (PM), ist eine einfache Liste fester Größe, deren Einträge, secondary map (SM), einen festen Adressbereich des Shadow Memorys abbilden. Memcheck bildet ein Byte Speicher im Allgemeinen mit 2 Bit Shadow Memory ab. Da Memcheck die Initialisierung mit Bitgenauigkeit verfolgt, werden im Spezialfall, dass das Byte nur zum Teil initialisiert wurde, 10 Bit benötigt [23].

Das Hauptproblem bei der Parallelisierung ist die zusätzliche Aktualisierung des Sha-

dow Memorys. Nachdem eine Operation auf den Gastdaten durchgeführt wurde, muss ebenfalls eine entsprechende Operation auf dem Shadow Memory ausgeführt werden. Eine Instruktion wird somit in mindestens zwei Instruktionen überführt. Bei komplexeren Werkzeugen können auch mehrere Instruktionen benötigt werden um den Shadow Memory zu aktualisieren.

Die Aufteilung einer Instruktion in mehrere Instruktionen kann schon während der normalen Übersetzung von Maschinencode in IR und zurück in Maschinencode passieren. Die IR in Valgrind ist RISC-basiert und von der Maschinensprache unabhängig. Es gibt keine direkte eins-zu-eins Übersetzung der Instruktionen. Bei der Rückübersetzung entsteht somit oft anderer Maschinencode, selbst wenn das Werkzeug keinerlei Instrumentierung vorgenommen hat. Im Allgemeinen ist das kein Problem. Nur bei atomaren Instruktionen muss die Atomarität gewahrt bleiben. Um dies sicherzustellen, unterstützt die IR in Valgrind auch atomare Instruktionen.

Listing 5.1 und 5.2 veranschaulichen diesen Sachverhalt an einem kleinem Beispiel. Ein atomares Inkrementieren eines globalen Zähler ist kaum wieder zu erkennen. Die entscheidenden Instruktionen im rückübersetzten Code sind auf Zeile 3 und 8. Zeile 3 inkrementiert den Wert des Zählers lokal und Zeile 8 schreibt den inkrementierten Wert atomar zurück in den Zähler. Die verwendete Instruktion cmpxchgl überprüft dabei, ob der Zähler inzwischen manipuliert wurde. Falls der Zähler verändert wurde, springt Zeile 11 mit Umweg über den Dispatcher zurück auf Zeile 1 und das ganze wird erneut probiert. Die Atomarität bleibt somit bewahrt.

```
1 lock incl counter
```

Listing 5.1: atomares Inkrementieren eines globalen Counters

```
1  movl $0x8049088,%esi
2  movl (%esi),%edi
3  leal 0x1(%edi),%esi  # inc
4  movl $0x8049088,%edx
5  movl %edi,%ecx
6  movl %edi,%eax
7  movl %esi,%ebx
8  lock cmpxchgl {%eax->%ebx},(%edx)
9  cmovnz %eax,%ecx
10 cmpl %edi,%ecx
11 if (%eflags.nz) { movl $0x8048074,%
       eax ; movl $dispatcher_addr,%edx
         ; jmp *%edx }
12 pushl 0x2C(%ebp)
13 movl 0x28(%ebp),%ecx
14 movl 0x24(%ebp),%edx
15 movl 0x20(%ebp),%eax
16 call[3] 0x380C35C0   #
       x86g_calculate_eflags_c
17 addl $0x4,%esp
18 movl %eax,%edi
19 movl %edi,0x2C(%ebp)
20 movl $0x12,0x20(%ebp)
21 movl %esi,0x24(%ebp)
22 movl $0x0,0x28(%ebp)
```

Listing 5.2: rückübersetzter Assembler

Bei der Verwendung von Shadow Memory reicht das allerdings nicht aus. Falls ein Gastprogramm ein Datum mittels atomarer Instruktion manipuliert, ist die folgende Aktualisierung des Shadow Memorys immer noch ein Problem. Beide Instruktionen zu-

sammen sind nicht atomar. Es kann zu unerwünschten Zwischenergebnissen im Shadow Memory kommen und somit zu falschen Aussagen über den Zustand des Speichers vom Gastprogramm. Dies ist einer der Hauptgründe für die serielle Ausführung von Threads in Valgrind.

Eine mögliche Lösung des Problems ist die Nutzung von Transactional Memory. Die ursprüngliche Operation des Gastprogramms und die hinzugefügten Instruktionen für den Shadow Memory werden in einer Transaktion zusammengefasst. Durch die Transaktion werden die separaten Operationen wieder wie eine einzelne atomare Operation betrachtet. Aus Performancegründen bietet sich die Nutzung von HTMs wie ASF [7, 10] an.

ASF ist eine Befehlssatzerweiterung von x86 und ermöglicht die einfache Implementierung von Transaktionen auf Assemblerebene. Wie alle HTM-Systeme besitzt auch ASF Obergrenzen für die Länge von Transaktionen und der Verschachtelungstiefe. Alle ASF-konformen Implementierungen müssen mindestens 4 geschützte Variablen unterstützen und eine maximale Verschachtelungstiefe von 256. Geschützte Variablen lassen sich spekulativ verändert und werden im Falle eines Abbruchs der Transaktion zurückgerollt.

Um ASF einsetzen zu können, müssten die neuen Instruktionen in die IR von Valgrind übernommen und die Werkzeuge in Valgrind, welche Shadow Memory verwenden, entsprechend angepasst werden. Problematisch dabei dürfte die geringe Größe von Transaktionen sein. Komplizierte Indexstrukturen für den Shadow Memory sind damit nur schlecht umsetzbar. Der Rückgriff auf eine STM-Implementierung oder serieller Ausführung ist bei besonderen Instruktionen und Werkzeugen wahrscheinlich nötig.

Ein möglicher Optimierungsschritt ist, nur atomare Instruktionen des Gastprogramme in Transaktionen zu überführen. Damit bleibt die Instruktion und die Aktualisierung des Shadow Memorys ebenfalls atomar. Alle anderen Instruktionen können normal ausgeführt werden. Wenn es Data Races auf diesen Variablen gibt, ist das Gastprogramm sowieso nicht threadsicher. Das Fehlverhalten im Falle von Data Races auch im Shadow Memory abzubilden, bringt keine zusätzlichen Informationen für das Werkzeug.

Es gibt zur Zeit noch keine Hardware, die ASF unterstützt. Um eine Implementation zu testen, muss auf Emulatoren zurückgegriffen werden [10].

Das aufgezeigte Verfahren zur Parallelisierung von Shadow Memory wurde nich hinsichtlich seiner Machbarkeit untersucht. Für die Durchführung eines Nachweis, ob ASF für Parallelisierung verwendbar ist, bietet sich eine Folgearbeit an.

6. Zusammenfassung

Es wurde in dieser Arbeit gezeigt, dass der Kern von Valgrind parallelisierbar ist. Für die auftretenden Parallelisierungsprobleme wurden Lösungen aufgezeigt und implementiert. Die Messungen in Kapitel 4 zeigen, dass die Ausnutzung von mehreren Kernen bei der Analyse von nebenläufigen Anwendungen einen deutlichen Performancegewinn mit sich bringt. Die Probleme bei der Parallelisierung von Shadow Memory wurden in Kapitel 5 diskutiert und ein möglicher Lösungsansatz mit Hilfe von Transactional Memory und ASF aufgezeigt. Die Weiterentwicklung und die Umsetzung dieses Ansatzes könnten in einer Folgearbeit bearbeitet werden. Ein weiteres offenes Thema ist die Parallelisierung der Werkzeuge von Valgrind.

Des Weiteren bleibt noch zu klären, ob das Biglock nicht vollständig aufgelöst werden kann und ob das für die Performance Vorteile bringt. Außerdem ist noch offen, warum der Benchmark ep im parallelisierten Valgrind so schlechte Ergebnisse abgeliefert hat.

Anhang A.

Benchmarks von NPB

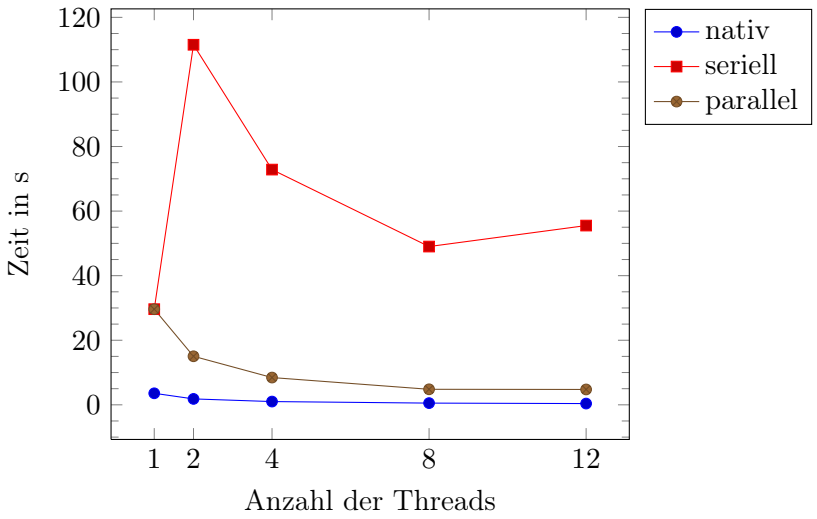

Abbildung A.1.: Benchmark bt

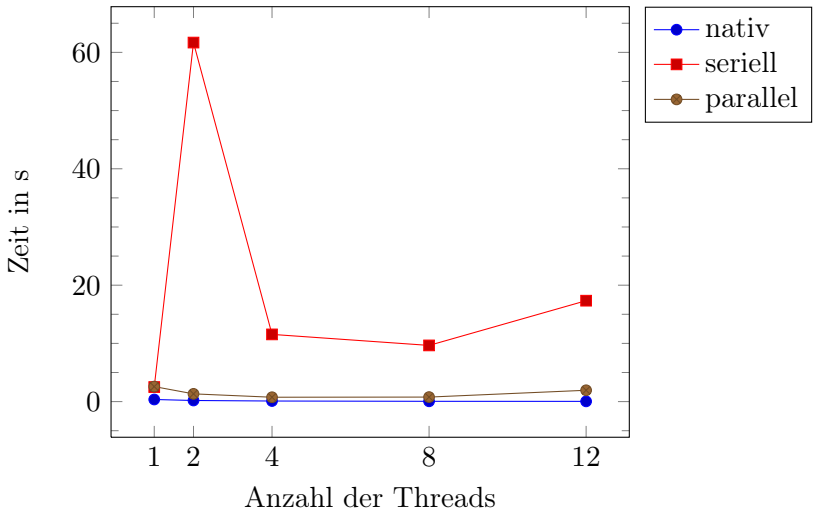

Abbildung A.2.: Benchmark cg

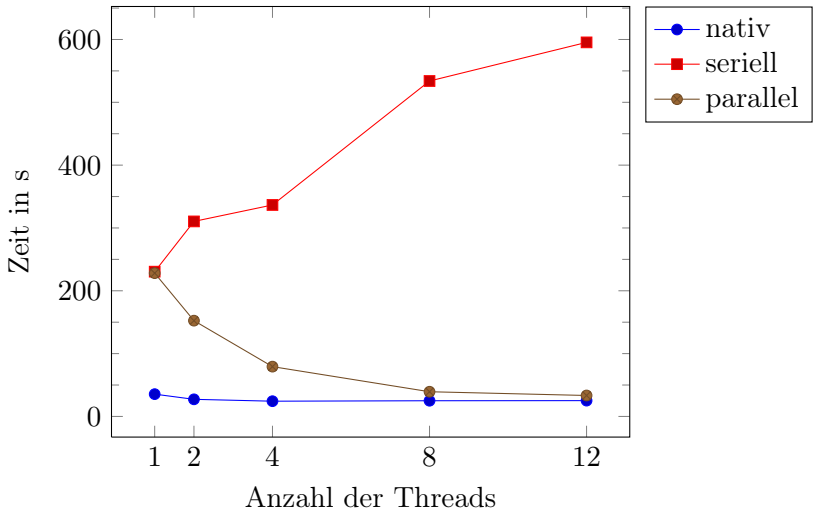

Abbildung A.3.: Benchmark dc

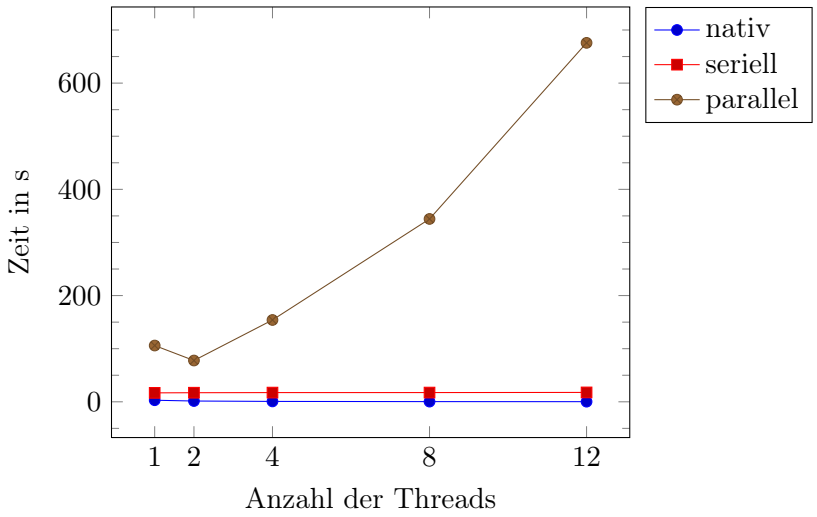

Abbildung A.4.: Benchmark ep

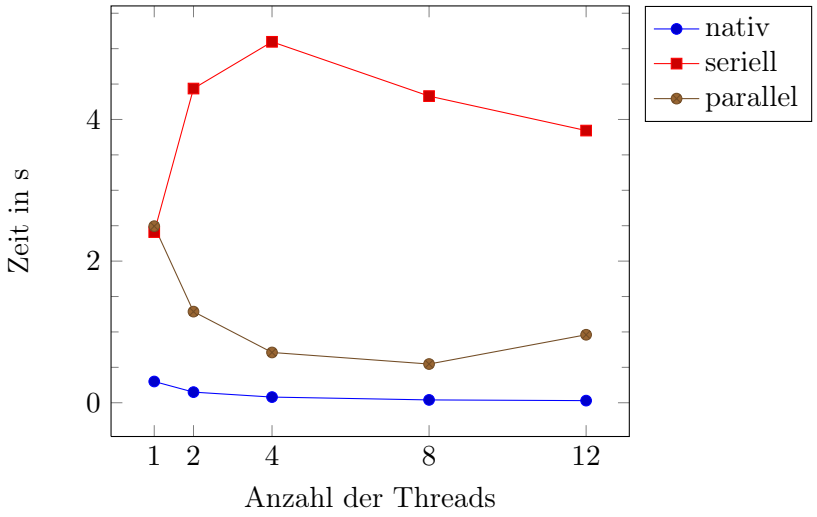

Abbildung A.5.: Benchmark ft

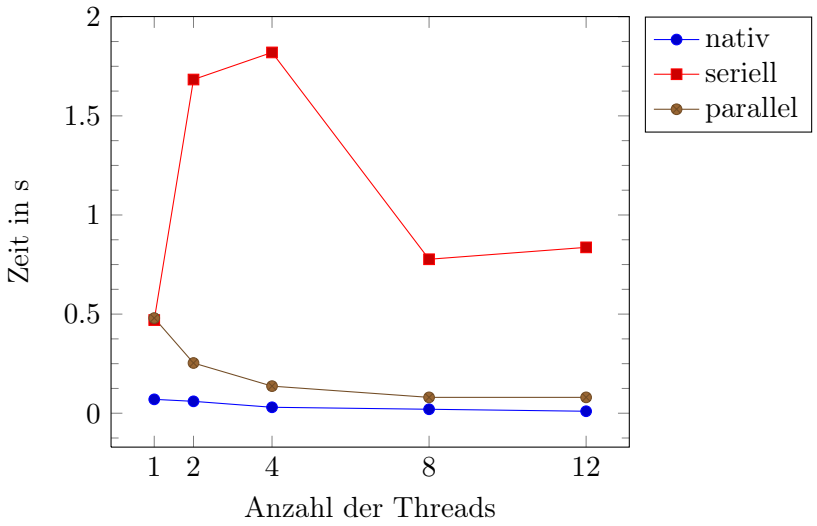

Abbildung A.6.: Benchmark is

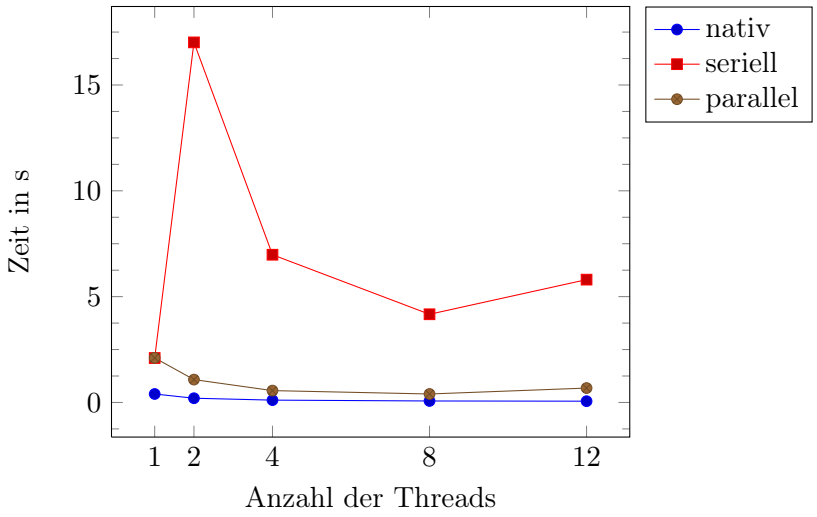

Abbildung A.7.: Benchmark mg

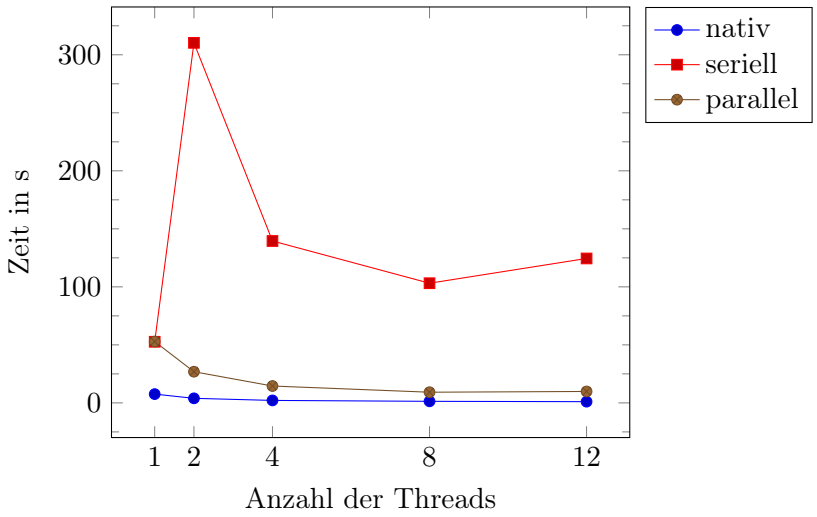

Abbildung A.8.: Benchmark sp

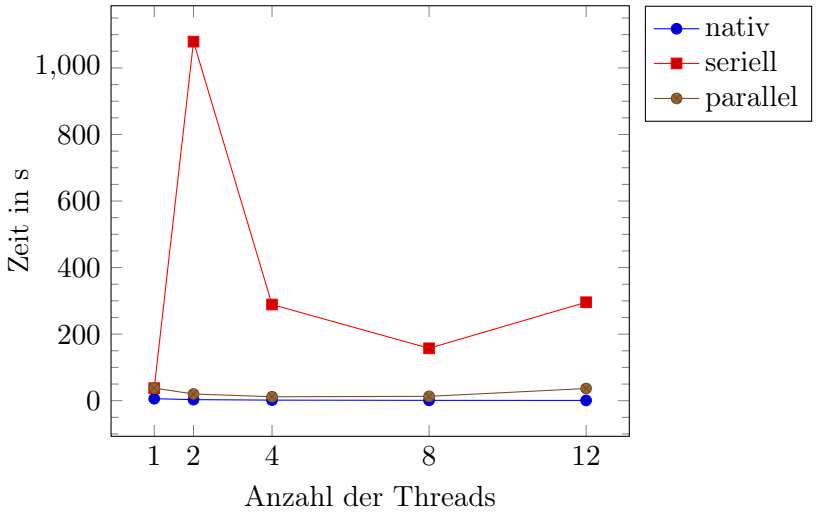

Abbildung A.9.: Benchmark ua

Literaturverzeichnis

[1] *Clang Static Analyzer*. http://clang-analyzer.llvm.org/

[2] *Cppcheck*. http://cppcheck.sourceforge.net/

[3] *GNU Compiler Collection (GCC)*. http://gcc.gnu.org/

[4] *GNU Profiler (gprof)*. http://sourceware.org/binutils/docs/gprof/

[5] *Transactional Memory in GCC*. http://gcc.gnu.org/wiki/TransactionalMemory

[6] Adl-Tabatabai, Ali-Reza ; Shpeisman, Tatiana: Draft Specification of Transactional Language Constructs for C++. (2009). http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/

[7] Advanced Micro Devices, Inc.: *Advanced Synchronization Facility - Proposed Architectural Specification*. 2.1, March 2009

[8] Ananian, C. S. ; Asanovic, Krste ; Kuszmaul, Bradley C. ; Leiserson, Charles E. ; Lie, Sean: Unbounded Transactional Memory. In: *IEEE Micro* 26 (2006), January, 59–69. http://dx.doi.org/2006-02-1702:00:03.800. – DOI 2006–02–17 02:00:03.800. – ISSN 0272–1732

[9] Bailey, D. H. ; Barszcz, E. ; Barton, J. T. ; Browning, D. S. ; Carter, R. L. ; Fatoohi, R. A. ; Frederickson, P. O. ; Lasinski, T. A. ; Simon, H. D. ; Venkatakrishnan, V. ; Weeratunga, S. K.: The NAS Parallel Benchmarks / The International Journal of Supercomputer Applications. 1991. – Forschungsbericht

[10] Christie, Dave ; Chung, Jae-Woong ; Diestelhorst, Stephan ; Hohmuth, Michael ; Pohlack, Martin ; Fetzer, Christof ; Nowack, Martin ; Riegel, Torvald ; Felber, Pascal ; Marlier, Patrick ; Rivière, Etienne: Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In: *Proceedings of the 5th European conference on Computer systems*. New York, NY, USA : ACM, 2010 (EuroSys '10). – ISBN 978–1–60558–577–2, 27–40

[11] Damron, Peter ; Fedorova, Alexandra ; Lev, Yossi ; Luchangco, Victor ; Moir, Mark ; Nussbaum, Daniel: Hybrid transactional memory. In: *SIGOPS Oper. Syst. Rev.* 40 (2006), October, 336–346. http://doi.acm.org/10.1145/1168917.1168900. – ISSN 0163–5980

[12] Desnoyers, M. ; McKenney, P. ; Stern, A. ; Dagenais, M. ; Walpole, J.: User-Level Implementations of Read-Copy Update. In: *Parallel and Distributed Systems, IEEE Transactions on* PP (2011), Nr. 99, S. 1. http://dx.doi.org/10.1109/TPDS.2011.159. – DOI 10.1109/TPDS.2011.159. – ISSN 1045–9219

[13] Dice, Dave ; Lev, Yossi ; Moir, Mark ; Nussbaum, Daniel: Early experience with a commercial hardware transactional memory implementation. In: *SIGPLAN Not.* 44 (2009), March, 157–168. http://doi.acm.org/10.1145/1508284.1508263. – ISSN 0362–1340

[14] Dijkstra, E. W.: Solution of a problem in concurrent programming control. In: *Commun. ACM* 8 (1965), September, 569–. http://dx.doi.org/http://doi.acm.org/10.1145/365559.365617. – DOI http://doi.acm.org/10.1145/365559.365617. – ISSN 0001–0782

[15] Drepper, Ulrich: Futexes are Tricky. (2005), December

[16] Felber, Pascal ; Fetzer, Christof ; Riegel, Torvald: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008

[17] Harris, Tim ; Fraser, Keir: Language support for lightweight transactions. In: *SIGPLAN Not.* 38 (2003), October, 388–402. http://doi.acm.org/10.1145/949343.949340. – ISSN 0362–1340

[18] Herlihy, Maurice ; Lev, Yossi: tm_db: A Generic Debugging Library for Transactional Programs. In: *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA : IEEE Computer Society, 2009. – ISBN 978–0–7695–3771–9, 136–145

[19] Herlihy, Maurice ; Moss, J. Eliot B.: Transactional memory: architectural support for lock-free data structures. In: *SIGARCH Comput. Archit. News* 21 (1993), May, 289–300. http://doi.acm.org/10.1145/173682.165164. – ISSN 0163–5964

[20] Majumdar, Dibyendu: A Quick Survey of MultiVersion Concurrency Algorithms. In: *Read* (2002), 1–8. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.4557&rep=rep1&type=pdf

[21] McKenney, Paul E. ; Michael, Maged M. ; Triplett, Josh ; Walpole, Jonathan: Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. In: *SIGOPS Oper. Syst. Rev.* 44 (2010), August, 93–101. http://doi.acm.org/10.1145/1842733.1842749. – ISSN 0163–5980

[22] McKenney, Paul E. ; Slingwine, John D.: Read-Copy Update: Using Execution History to Solve Concurrency Problems. In: *Parallel and Distributed Computing and Systems*. Las Vegas, NV, October 1998, S. 509–518. – Available: http://www.rdrop.com/users/paulmck/RCU/rclockpdcsproof.pdf [Viewed December 3, 2007]

[23] Nethercote, Nicholas ; Seward, Julian: How to shadow every byte of memory used by a program. In: *Proceedings of the 3rd international conference on Virtual execution environments*. New York, NY, USA : ACM, 2007 (VEE '07). – ISBN 978–1–59593–630–1, 65–74

[24] Nethercote, Nicholas ; Seward, Julian: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: *SIGPLAN Not.* 42 (2007), June, 89–100. `http://doi.acm.org/10.1145/1273442.1250746`. – ISSN 0362–1340

[25] Robson, Daniel ; Strazdins, Peter: Parallelisation of the Valgrind Dynamic Binary Instrumentation Framework. (2008)

[26] Welc, Adam ; Saha, Bratin ; Adl-Tabatabai, Ali-Reza: Irrevocable transactions and their applications. In: *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. New York, NY, USA : ACM, 2008 (SPAA '08). – ISBN 978–1–59593–973–9, 285–296

[27] Zyulkyarov, Ferad ; Stipic, Srdjan ; Harris, Tim ; Unsal, Osman S. ; Cristal, Adrián ; Hur, Ibrahim ; Valero, Mateo: Discovering and understanding performance bottlenecks in transactional applications. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, NY, USA : ACM, 2010 (PACT '10). – ISBN 978–1–4503–0178–7, 285–294

Abbildungsverzeichnis

Listings