Software Plattform Embedded Systems 2020

GEFÖRDERT VOM

- Deliverable 5.3.B: Konzept für minimalinvasive Debugging-Tools -

Version: 1.1

Projektbezeichnung	SPES 2020	
Verantwortlich	Hartmut Lackner	
QS-Verantwortlich	Siemens CT	
Erstellt am	18.11.10	
Zuletzt geändert	27.01.2011 15:30	
Freigabestatus		Vertraulich für Partner:
		Projektöffentlich
	X	Öffentlich
Bearbeitungszustand		in Bearbeitung
		vorgelegt
	X	fertig gestellt

Weitere Produktinformationen

Erzeugung	Hartmut Lackner
Mitwirkend	

Änderungsverzeichnis

Änderung			Geänderte Kapitel	Beschreibung der Änderung	Autor	Zustand
Nr.	Datum	Version				
1	18.11.10	1.0	Alle	Initiale Produkterstellung	Hartmut Lackner	Freigabe QS
2	27.01.11	1.1	Alle	Einarbeitung der Korrekturen aus dem Review	Hartmut Lackner	Finalisiert

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der anhaltende Trend von Single- zu Multi-Core Prozessoren führt zu einer Zunahme von echter Parallelität in Programmen. Die nebenläufige Ausführung eines Programmes verkürzt dessen Abarbeitungszeit mit steigendem Anzahl parallel ausführbarer Einheiten. Der derzeitige Trend von sequentiell zu parallel programmierter Software stellt einen Großteil der Software-Entwickler vor neue Herausforderungen. Fehler, die bisher nur in Programmen auftraten, die über mehrere Prozessoren oder ein Netzwerk verteilt waren, finden jetzt ihren Weg in konventionelle Desktop-Anwendungen und werden somit allgegenwärtig.

Das folgende Beispiel unterstreicht das Gefahrenpotential dieser Fehlerklasse: Am Dienstag, dem 14. August 2003, legte ein Stromausfall in Kanada im Laufe des Nachmittags fast die komplette Stromversorgung des Nord-Ostens der USA lahm. Trotz einer Überwachungssoftware konnte sich der Stromausfall ungehindert im amerikanischen Stromnetz ausbreiten.

Das „General Electric XA/21 Energy Management System“ war seit 1990 durchgehend in Gebrauch und lief mehr als drei Millionen Stunden fehlerfrei. Nur in dieser einzigen Situation arbeitete die Software nicht korrekt: Sie hatte einen Concurrency Bug. Acht Wochen vergingen, bevor die Techniker von General Electrics die vier Millionen Zeilen Quelltext untersucht hatten und durch Zufall auf den Defekt stießen. Die von diesem Fehler verursachten Kosten beliefen sich auf insgesamt sechs Milliarden US-Dollar [1].

Wie das Beispiel zeigt, ist das Beherrschen dieser Fehlerklasse die Voraussetzung für den Einsatz von nebenläufiger Software in sicherheitskritischen Bereichen. Da traditionelle Ansätze des Debuggings teilweise nicht mehr anwendbar sind, stellt sich die Frage nach neuen Möglichkeiten und Herangehensweisen. Die zeitliche Interaktion, auch als Interleaving bezeichnet, muss beim Debbuging nun zusätzlich in Betracht gezogen werden. Dies minimalinvasive Ansätze, die das Zeitverhalten der zu untersuchenden Software so gering wie möglich beeinflussen.

Deshalb untersuchen wir in diesem Bericht die aktuellsten Verfahren zum Debugging ne-

benläufiger Software. Dabei zeigen wir Stärken und Schwächen dieser Verfahren auf und bewerten ihre Praxistauglichkeit. Auf Grundlage der Ergebnisse dieser Analyse definieren wir neue Konzepte zum Debugging nebenläufiger Software.

Das Dokument ist wie folgt strukturiert: Als erstes führen wir die zum Verständnis der Arbeit notwendigen Grundlagen ein. Hierfür werden aktuelle Debugging-Techniken vorgestellt und die nebenläufige Ausführung von Programmen von der Ebene der Hardware bis zur Software erläutert. Im dritten Kapitel stellen wir die Klasse der Concurrency Bugs vor und diskutieren aktuelle Lösungsansätze. Daran anschließend entwickeln wir unser eigenes Konzept zum Detektieren von Concurrency Bugs auf Basis von Zugriffsmustern. Abschließend fassen wir unsere Ergebnisse in Kapitel 4 zusammen.

Kapitel 2

Grundlagen

Das folgende Kapitel legt den Grundstein für die darauf folgenden Kapitel. Wichtige Fragen, wie z.B. warum die Zukunft der Computerbranche in der Hand der Software-Entwickler liegt, werden erläutert. Nach einem Überblick darüber, weshalb ein Erhöhen der Taktraten physikalisch nicht mehr ohne großen Aufwand möglich ist, folgt eine Einführung in das Debugging von Software. Abschließend wird das Thema Nebenläufigkeit behandelt und warum dieses Thema für mindestens die nächste Generation der Software-Entwickler die zentrale Herausforderung sein wird.

2.1 Debugging

Ein Computerbug, oder kurz Bug, beschreibt in der Computerwelt einen Defekt im System. Die Bandbreite erstreckt sich von einer falschen Fensterbeschriftung, über eine falsche Berechnung bis hin zum Absturz und einem damit möglichen Datenverlust. Der Prozess, den für das Fehlverhalten verantwortlichen Bug zu finden und zu beheben, wird als Debugging bezeichnet.

Im Allgemeinen wird die Namensgebung auf das Auffinden einer Motte 1957 von Grace M. Hopper in den damals noch aus Röhren und Relais bestehenden Computer MARK II zurückgeführt. In [2] wird aber gezeigt, dass schon 1890 das Wort „Bug“ in der Elektrotechnik bekannt war. Vor Grace M. Hopper und der Motte, im Jahr 1938, wird das Wort sogar in Webster's International Dictionary als „a defect in apparatus or its operation“ erstmals definiert.

2.1.1 Debugging innerhalb der Software-Entwicklung

Das Finden und Beheben von Fehlern beginnt nicht erst mit dem Schreiben und dem ersten Ausführen von Software. Untersuchungen [3] haben gezeigt, dass schon bei der

Anforderungsanalyse bis zu 20% der später gefundenen Defekte eingeführt werden. Weitere 38% der Defekte haben ihren Ursprung in der Designphase der Software, in der meist noch keine Zeile Code entstanden ist. Die restlichen 42% entstehen dann in der Implementierungsphase.

Seit der Softwarekrise und den damit verbundenen NATO Konferenzen 1968 und 1969 [4, 5] wurden viele Prozesse und Vorgehensmodelle entwickelt, die die hohe Komplexität und damit verbundene Fehleranfälligkeit von Software handhabbar machen sollten. Inzwischen wird ein Großteil der Defekte durch Codereviews, Modultests, Integrations- und Abnahmetests gefunden, aber eben nicht alle. Um es in den Worten von Edsger W. Dijkstra auszudrücken: „Program testing can be used to show the presence of bugs, but never to show their absence!“ [6]. Es ist also weiterhin wichtig Werkzeuge zu entwickeln, die die bestehende Software untersuchen und dem Entwicklerteam dabei zu helfen die Anzahl der Fehler so gering wie möglich zu halten.

2.1.2 Was ist ein Fehler?

Bis ein System versagt, bzw. die gewünschten Anforderungen nicht mehr erfüllt, durchläuft es mehrere Zustände. Zum einen muss das System an einer oder mehreren Stellen defekt sein. Dieser *Defekt* wird irgendwann *aktiviert* und führt zu einem *fehlerhaften Systemzustand*. Wenn dieser Fehler nicht vom System erkannt und behoben wird, *propagiert* der Fehler und führt letztendlich zum *Ausfall* des Systems. Das Ausfallen eines Teilsystems kann wiederum zu Defekten in anderen Teilsystemen führen, welche wiederum zu Ausfällen propagieren können. In Abbildung 2.1 ist diese Kette dargestellt. Nach Avizienes et al. [7] wird ein Defekt, Fehler und Ausfall wie folgt definiert:

- *Defekt, engl. Fault*: A fault is the adjudged or hypothesized cause of an error.
- *Fehler, engl. Error*: An error is part of a system state that may cause a subsequent failure.
- *Ausfall, engl. Failure*: A system failure is an event that occures when the delivered service deviates from correct service.

Eine Softwarekomponente ist defekt, wenn diese z.B. das Maximum zweier Werte ermitteln soll, aber stattdessen das Minimum zurückgibt. Sie erfüllt die Anforderung also nicht. Wird die defekte Komponente von einer Sortierfunktion verwendet, dann ist das System während der Ausführung der Sortierfunktion in einem fehlerhaften Zustand. Der Defekt wird durch das Verwenden aktiviert. Wenn diese Softwarekomponente getestet wird und das Ergebnis nicht mit einer aufsteigend geordneten Liste übereinstimmt, der Test also fehlschlägt, kommt es zu einem Ausfall. Der Fehler ist damit zu einem Ausfall propagiert.

Abbildung 2.1: Defekt, Fehler und Ausfall nach Avizienis et al.

2.1.3 Klassifikation von Defekten

Defekte in Software können nach deren Verhalten kategorisiert werden. Sie können einfach reproduzierbar sein, scheinbar willkürlich auftreten oder erst in Erscheinung treten, wenn die Software vom Entwickler oder Testentwickler auf Fehler hin untersucht wird. Ausgehend von der Klassifikation von Gray [8] werden Defekte in permanente und unregelmäßig auftretende Defekte eingeteilt. Er taufte die beiden Klassen Bohr- und Heisenbugs in Anlehnung an das statische Atommodell von Bohr und der Heisenbergschen Unschärferelation. In dieser Arbeit wird die Klasse der Heisenbugs noch weiter in Heisen- und Mandelbugs unterteilt.

Bohrbugs sind permanente Defekte, die meist in den frühen Phasen der Software-Entwicklung auftreten. Sie sind einfach zu testen, da das Aktivieren der Defekte sich deterministisch verhält. Ein Beispiel dafür ist die Verwendung eines $<$ Operators anstelle eines $>$ Operators in einer Sortierfunktion. Unabhängig von dem Ausführungszeitpunkt wird diese Funktion deterministisch falsch sortieren.

Mandelbugs sind Defekte, die scheinbar chaotisch in Erscheinung treten. Der Abstand zwischen dem Zeitpunkt des Aktivierens und dem Ausfall des Systems ist so groß, dass ein Erkennen von kausalen Zusammenhängen nicht mehr möglich ist.

Heisenbugs sind vorübergehende bzw. unregelmäßig auftretende Defekte, die erst durch das aktive Untersuchen aktiviert oder deaktiviert werden. Genau genommen sind auch diese Defekte permanent (der übersetzte Quelltext wird während der Ausführung meist nicht mehr verändert), doch verhält sich das Aktivieren der Defekte nicht-deterministisch, was ein Reproduzieren in einer Testumgebung fast unmöglich macht. So verändert sich z.B. das Zeitverhalten, wenn für Diagnosezwecke ein Ereignisprotokoll mitgeschrieben wird. Diese Veränderung kann zu Seiteneffekten führen, die im Normalbetrieb nicht erscheinen. Im schlimmsten Fall wird der Defekt überhaupt nicht mehr aktiviert. In der Literatur ist dieses Phänomen auch unter dem Namen „Probe Effect“ bekannt.

Eine Studie [8] über die Defekte in Software in hochverfügbaren Computersystemen hat ergeben, dass 70% der später vom Kunden gefundenen Defekte zur Klasse der Mandel- und Heisenbugs zählen. Nicht nur, dass nach intensivem Testen der Software die Software noch Defekte enthielt. Schwerwiegender ist der Umstand, dass die Defekte von den Nutzern der Software während des Gebrauchs entdeckt wurden, nachdem die Entwicklung abgeschlossen war. Dass Software nicht immer korrekt arbeitet, wird offensichtlich als Normalität angesehen.

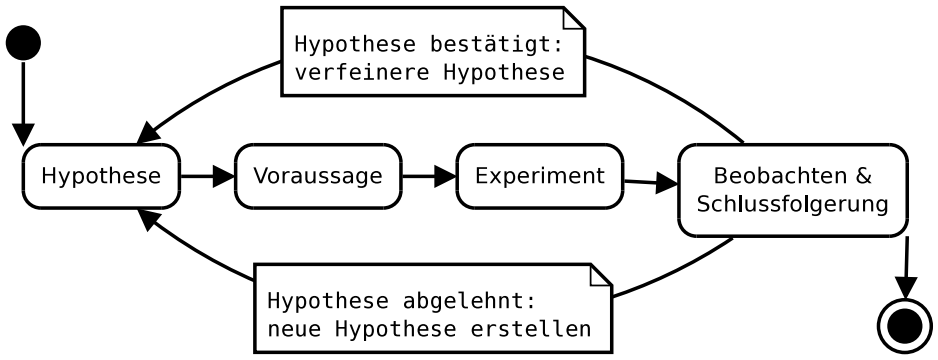

Abbildung 2.2: wissenschaftliche Methode

2.1.4 Der ideale Debugging Prozess

Ein Patentrezept um Defekte in Software lokalisieren zu können, existiert nicht. Eine Vorgehensweise, Phänomene in der Natur zu untersucht, neues Wissen zu erlangen oder bestehendes Wissen zu erweitern, existiert dagegen schon. Dieses Vorgehen wird die „wissenschaftliche Methode“ genannt. Eine wissenschafliche Methode ist ein generelles Vorgehen um einen bestimmten Sachverhalt zu erklären. Sie besteht nach [9] aus einer Menge von Daten, die durch Beobachtungen oder Experimente gewonnen wurden, und dem Formulieren und Testen von Hypothesen. Dieses Vorgehen kann auch zum Debuggen von Software verwendet werden.

In unserem Fall ist die Hypothese eine mögliche Erklärung dafür, warum ein Defekt aktiviert wurde und das System dadurch ausgefallen ist. In Abbildung 2.2 ist die „wissenschafliche Methode“ dargestellt.

Zu allererst muss eine Hypothese erstellt werden, die die Ursache, also die Aktivierung des Defekts, des Systemausfalls beschreibt. Dies kann in den ersten Iterationen noch relativ grob sein, zum Beispiel „Wenn wir Sortierfunktion $Sort()$ benutzen, fällt das System aus“ Aus der Hypothese wird eine Voraussage getroffen, welche im nächsten Schritt durch ein Experiment beobachtet wird. Wird diese Voraussage durch ein Experiment bestätigt, wird die Hypothese als wahr angesehen und in der nächsten Iteration verfeinert. Dies könnte zum Beispiel die neue Hypothese „Die Sortierfunktion $Sort()$ sortiert nur leere oder einelementige Listen korrekt“ sein. Falls die Voraussage nicht bestätigt wird, muss die Hypothese verworfen werden und eine neue aufgestellt werden. Nach einigen Iterationen sollte die Hypothese nun ausreichend verfeinert worden sein, um die Ursache des Ausfalls erklären zu können. Der Defekt kann nun behoben werden.

Der große Vorteil dieses Vorgehens ist die Nachvollziehbarkeit der durchgeführten Schritte. Werden die einzelnen Schritte zusätzlich dokumentiert, so kann der Debuggingprozess auch unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden. Ein wahlloses Suchen nach der Ursache wird durch dieses Vorgehen systematisiert.

2.1.5 Herausforderungen beim Debugging

Die wissenschaftliche Methode des Debuggings geht stark von der Beobachtbarkeit des Systems aus. Wie schon erwähnt, arbeiten heutige Prozessoren mit Frequenzen im Giga-Hertz Bereich. Dies entspricht 10^9 Zustandsänderungen des Prozessors in einer Sekunde. Um den aktuellen Zustand des Prozessors jede Sekunde bestimmen zu können, muss diese Datenmenge ersteinmal gespeichert werden. Für den Systemzustand eines Computers müssten zusätzliche Daten wie der Cacheinhalt und der RAM gespeichert werden. Die größte Herausforderung beim Debugging ist demnach die potentiell riesige Menge an Daten, die es zu untersuchen gilt [10].

Damit Menschen diese Menge an Daten überhaupt versetehen können, werden die Zustände der Transistoren zu Registerwerten und diese wiederum zu Variablen und Datenstrukturen abstrahiert. Ein Software-Entwickler, der für die Kommunikation mit der Datenbank zuständig ist, wird wahrscheinlich nicht die Registerbelegung eines Prozessors im Detail untersuchen. Ein Gefühl für die richtige *Daten Abstraktion* ist für ein erfolgreiches Debugging unumgänglich.

Ein heutiges Software-Produkt besteht aus einer Menge von Softwarekomponenten. Der modulare Aufbau des Software-Produkts ermöglicht eine *strukturelle Abstraktion* des Systems während der Entwicklung, aber auch während des Debuggings. Statt alle Komponenten zu untersuchen, werden hier einzelne Komponenten genauer untersucht. Für einen Hardwareentwickler könnten dies die ALU, der L1-Cache oder die Register sein. Für einen Software-Entwickler könnten es zum Beispiel einzelne Klassen oder ausgewählte Schichten einer Software sein.

Beide Abstraktionsebenen müssen noch mit der *temporalen Abstraktion* verbunden werden. Diese gibt an, in welcher Zeitgranularität das System beobachtet werden soll und bestimmt im Wesentlichen die Menge der zu untersuchenden Daten. Für einen Software-Entwickler könnte zum Beispiel bei jedem Funktionsaufruf der aktuelle Zustand in Form von interessanten Variablen ausgegeben werden.

Neben der Wahl des richtigen Abstraktionslevels, spielt der schon erwähnte „Probe Effect“ eine weitere Rolle. Er macht das Debugging noch komplizierter. Werner Heisenberg [11] schrieb dazu:

> [...], dass jeglicher Vorgang aufgrund der ontologischen Struktur des verwendeten Beobachtungsapparates den Gegenstand, und damit das Resultat der Beobachtung, verändert. Es gibt keine neutrale Beobachtung, die den Beobachtungsgegenstand unverändert hinterlässt.

Beobachtungen von Software geschehen oft durch das Protokollieren wichtiger Ereignisse. Mit einer kleinen Zeitverzögerung werden Zustandsänderungen und andere wichtige Ereignisse an den Bildschirm gesendet oder in eine Datei geschrieben. Das Ausgeben eines Protokolls, wie durch Ausgabe auf den Bildschirm oder durch das Schreiben in eine Protokolldatei, verändert aber auf der Prozessorebene das Zeitverhalten der Software,

da diese Anweisungen nun zusätzlich geladen und ausgeführt werden müssen. Was jetzt beobachtet wird, ist nicht mehr das zu untersuchende Programm, sondern vielmehr das Programm inklusive der Beobachtung selbst. Deshalb ist ein minimal invasives Vorgehen bei der Beobachtung ebenso wichtig, wie die angemessene Wahl an Daten-, struktureller und temporaler Abstraktion.

2.1.6 Debugging Methoden

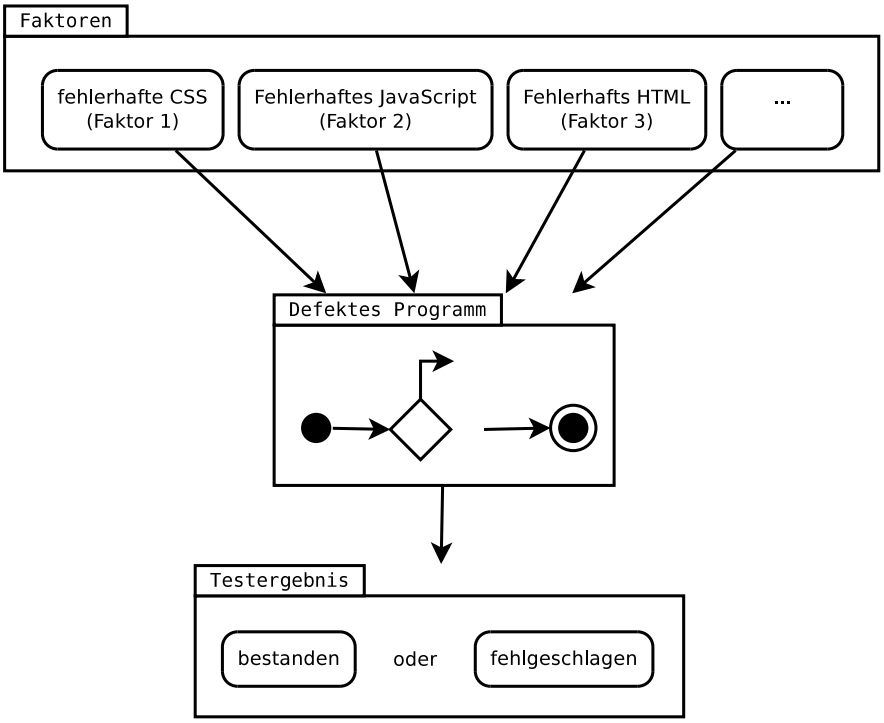

Abbildung 2.3: Delta Debugging

Ist die Aktivierung des Defekts reproduzierbar, so kann die wissenschafliche Methode in Abbildung 2.2 automatisiert werden. Diese Methode heißt *Delta Debugging* und wurde 2001 von Andreas Zeller vorgestellt [12].

Im Grunde ist es ein Divide-and-Conquer Ansatz, in dem die Menge der möglichen Fehlerquellen systematisch verkleinert wird. Man stelle sich ein defektes Programm vor. Die Aktivierung des Defektes hängt von mehreren Faktoren ab. Diese Menge der Faktoren wird schrittweise verkleinert, bis am Ende die Ursache für die Aktivierung des Defekts eindeutig identifiziert werden kann.

In Abbildung 2.3 ist dieses Vorgehen exemplarisch dargestellt. Das defekte Programm ist in diesem Beispiel ein Browser, der das Anzeigen von HTML Seiten ermöglicht. Bei bestimmten Seiten stürzt das Programm ab. Für die Aktivierung des Defekts können nun

eine Auswahl von Faktoren zuständig sein. Der Delta Debugging Algorithmus mischt die Menge der Faktoren bis die kleinste defektaktivierende Menge gefunden wird. Teilabschnitte von HTML oder JavaScript können hierbei auch einzelne Faktoren darstellen.

Die Methode setzt neben der Wiederholbarkeit des Experiments auch einen funktionierenden Test voraus. Anhand des Tests kann der Delta Debugging Algorithmus seine Entscheidung über die Auswahl der Faktoren für die nächste Iteration treffen.

Um einen beobachteten Ausfall wiederholen zu können, wurde die Methode des *Deterministic Replay* entwickelt. Fällt ein System in einer Testumgebung aus, wird danach das System erneut gestartet. Diesmal werden aber alle Parameter mitprotokolliert, die für ein Reproduzieren des gleichen Systemverhaltens in einem späteren Debuggingdurchgang wichtig sind. Gerade bei der Ausführung von Java-Programmen, die in einer virtuellen Maschine ausgeführt werden, also nicht nativ auf dem Prozessor, ist die Menge dieser Parameter überschaubar [13]. Die Ausführung des Systems wird so lange wiederholt, bis der temporäre Defekt wieder aktiviert wird. Unter Verwendung des Protokolls, kann nun die Aktivierung des temporären Defekts deterministisch beliebig oft wiederholt werden.

2.1.7 Debugger

Um Debugging Methoden, wie im Vorhergehenden vorgestellt anzuwenden, werden konkrete Verfahren benötigt. Im Folgenden werden die für diese Arbeit wichtigsten Verfahren analysiert. Neben der Beschreibung der Verfahren wird insbesondere ihre Auswirkung auf das Laufzeitverhalten der zu untersuchenden Anwendung beurteilt. Für die spätere Auswahl eines Verfahrens zum Debugging nebenläufiger Programme ist diese Beurteilung von hoher Wichtigkeit. Die Verfahren werden deshalb nach dem Schweregrad der Programmlaufzeitveränderung absteigend geordnet vorgestellt.

Stepper

Verfahren die den Programmablauf zeitlich am stärksten beeinflussen sind Stepper. Dazu zählen klassische Debugger, wie der GNU Debugger „gdb“, den gesamten Programmablauf anhalten können und dann die schrittweise Ausführung der Befehle im Quellcode ermöglichen. Damit das zu untersuchende Programm mit einem Debugger untersucht werden kann, müssen während des Kompilierprozesses Debugging Informationen generiert und dem Debugger zur Verfügung gestellt werden. Diese Debugging Informationen beschreiben die Datentypen und Namen jeder verwendeten Variable sowie der Funktionen und erlaubt es dem Debugger eine Relation zwischen Zeilennummer im Quelltext und Speicher-Adresse im Binärcode herzustellen [14].

Um ein Programm (auch als „Debuggee“ bezeichnet) während der Ausführung durch einen Debugger an einer bestimmten Stelle anhalten zu lassen, muss der Entwickler einen Breakpoint setzen. Nach Anhalten des Programms durch den Debugger, kann

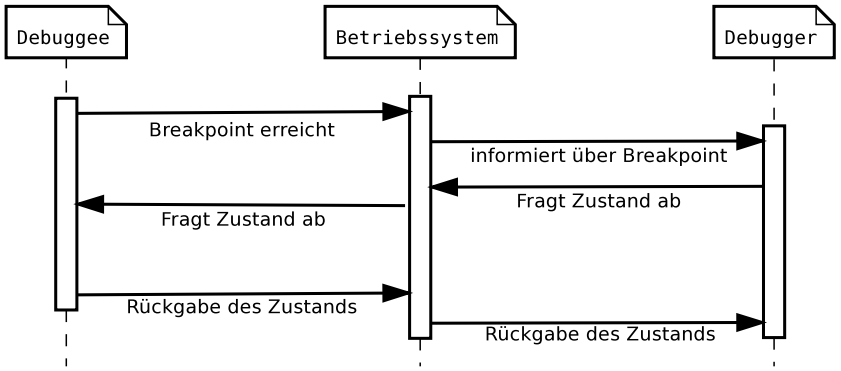

Abbildung 2.4: Das Zusammenspiel von Debuggee, Betriebssystem und Debugger

der Entwickler Variablenwerte, Registerinhalte, den Inhalt des Programmstacks oder den Speicherinhalt genauer untersuchen. Die Ausführung des Programms kann danach schrittweise (engl. Single-Stepping) oder normal weiter ausgeführt werden. Das Betriebsystem muss das Steuern eines Debuggees explizit unterstützen. Unter Unix wird dies z.B. durch den Systemaufruf *ptrace* realisiert. In Abbildung 2.4 ist das Zusammenspiel von Debuggee, Debugger und Betriebssystem dargestellt.

Die Realisierung des Debuggers kann sowohl in Soft- als auch Hardware geschehen. Während gdb eine Softwarelösung ist, sind JTAG-Komponente [15] eine Hardwarelösung auf Prozessor- und Schaltkreisebene. Hardwarelösungen werden auf eingebettete Systeme angewandt, die in ihrer Arbeitsumgebung untersucht werden sollen. Sie werden dann eingesetzt, wenn die Mittel der Entwicklungsumgebung erschöpft sind.

Stepper sind eine maximal invasive Lösung, das sie das Laufzeitverhalten der zu untersuchenden Anwendung so stark beeinflussen, wie kein anderes hier untersuchtes Verfahren. Sie bieten dafür den Vorteil, jede Variable und jede Funktionen zu einem beliebigen Ausführungspunkt genaustens zu untersuchen.

Code-Instrumentierung

Ein weniger invasives Verfahren als das Stepping ist die Code-Instrumentierung. Bei einer Instrumentierung des Codes wird der Quellcode oder der Zielcode um Zusatzinfomrationen angereichert. Diese Zusatzinformationen dienen dazu, das Verhalten und den Zustand des beobachteten Programms näher untersuchen zu können, ohne dabei dessen Ausführung zu behindern. Da die Interpretation der Zusatzinformationen CPU-Zeit benötigt, verursacht das Verfahren Nebeneffekte auf dem ausführenden System, die sich auf das Laufzeitverhalten auswirken können.

Für die Ausführung instrumentierten Codes gibt es zwei Herangehensweisen. Zum einen kann das zu untersuchende Programm in einer virtuellen Maschine ausgeführt werden,

die unter der Kontrolle des Beobachters steht. Einen solchen Ansatz verfolgt das Instrumentierungswerkzeug *Valgrind*. Hierbei wird während der Laufzeit jeder Befehl des zu instrumentierenden Programms in einen Befehl der Valgrind-Maschine, den UCode übersetzt. Dieser neue Befehl wird darauf hin von der virtuellen Maschine ausgeführt. Dieser Zwischenschritt versetzt den Beobachter in die Lage eigene Werkzeuge zwischenzuschalten, die auf die übersetzten Befehle reagieren. Der Orginal Programmcode des zu untersuchenden Programms wird nicht auf der Hardware ausgeführt sondern der übersetzte UCode. Da die Übersetzung des Programmes zu UCode für jede Answeisung stattfinden muss, ist eine Laufzeitverlängerung nicht zu umgehen.

Einen anderen Ansatz verfolgt das von Intel unterstützte Instrumentierungswerkzeug *PIN*. PIN arbeitet ähnlich einem Just-In-Time Compiler, wobei hier die Eingabe der Binärcode des zu instrumentierenden Programms ist. Die erste Instruktion des Programms wird kurz vor der Ausführung abgefangen und von PIN bei Bedarf um eigene Anweisungen ergänzt. Die so erweiterte Sequenz von Anweisungen wird nach der Instrumentierung ausgeführt. Diese Instrumentierung kann von eigenen „PINTools“ gesteuert werden. Der Unterschied zu Valgrind besteht darin, dass der von PIN instrumentierte Code auch wirklich auf der Hardware ausgeführt wird. Wird wenig instrumentiert, ist die Laufzeitverlängerung ebenfalls gering.

Aktives Logging

Eine der einfachsten und sehr häufig verwendeten Debugging Methoden ist das „printf-Debugging“. Hierbei wird an potentiell interessanten Stellen eine Ausgabe an die Konsole gesendet, die den aktuellen Zustand des Systems ausschreibt. Die Wahl der Stellen ist in den ersten Ausführungen noch relativ beliebig. Intuitiv wird von einem globalen Systemzustand hin zu einzelnen Werten von Variablen abstrahiert und strukturelle Einschränkungen getroffen. Diese Methode funktioniert recht gut für kleine überschaubare Programme in denen die Frage „Welcher Ausführungspfad wird von meinem Programm genommen?“ oder „Welchen Wert hat Variable x?“ im Mittelpunkt steht.

Printf Debugging, sowie der oben beschriebene ideale Debugging Prozess, setzt die Wiederholbarkeit des Experimentes voraus. Durch die spekulative Ausführung von Instruktionen und das Verwenden von Caches für schnellere Zugriffe auf Daten in einem Prozessor, ist die Ausführung von Software auf der untersten Ebene nicht mehr deterministisch wiederholbar. Vor allem Defekte, die nur bei einer bestimmten Reihenfolge von ausgeführten Instruktionen aktiviert werden, treten mit einer sehr geringen Wahrscheinlichkeit auf. Wenn jetzt noch wahllos neue Instruktionen in Form von *printfs* hinzugenommen oder entfernt werden, wird ein exaktes Reproduzieren des Ausfalls noch schwieriger.

Passives Logging

Im Gegensatz zum aktiven Logging wird der Quellcode des zu untersuchenden Programms beim passiven Logging nicht verändert. Stattdessen werden relevante Daten an Schnittstellen des Systems direkt mitgeschnitten, ohne dass das abhörende System dabei aktiv mit dem zu untersuchenden System kommuniziert. Diese Schnittstellen können Kommunikationsschnittstellen sein oder aber auch spezielle Debugschnittstellen, die das Beobachten der Hardware ermöglicht.

Das Verfahren ermöglicht keine vollständige Analyse des Verhaltens oder des Zustands am zu untersuchenden System. Häufig erfordert die Analyse der mitgeschnittenen Daten zudem noch eine Filterung, die den Datensatz auf die für die Untersuchung relevanten Daten reduziert. Somit ist die Beeinflussung des zu untersuchenden Systems minimal, der Rückschluss auf interne Systemzustände jedoch aufwändig. Sind Debug-Schnittstellen vorhanden, erleichtern diese den Zugriff auf interne Systemzustände signifikant. Hierfür eignen sich z.B. auch die bereits oben erwähnten JTAG-Komponenten, wenn auf die Anwendung des Steppings oder anderer invasiver Features verzichtet wird.

2.2 Concurrency - nebenläufige Ausführung

Der Versuch bessere Performance durch Parallelität zu erreichen, zieht sich durch alle Schichten des Computers. Von der Hardware, über die Prozesse in einem Betriebssystem, bis hin zur Parallelität innerhalb eines Programms.

2.2.1 Parallelität in Hardware

Auf der untersten Ebene wird Parallelität benutzt, um die vorhandenen Ressourcen wie Register und Caches so gut wie möglich auszulasten. Dies ist unter dem Namen „Bit Level Parallelism“ (BLP) bekannt. Während der Anfangszeit der Prozessoren lag die Wortgröße bei 4 Bit. Damit eine Recheneinheit z.B. zwei Zahlen addieren kann, muss der Additionsbefehl dekodiert, beide Zahlen geladen und im letzten Schritt muss die Addition durchgeführt werden. Außerdem muss das Ergebnis gespeichert werden. Können die Zahlen aber nicht mit 4 Bit dargestellt werden,müssen die fehlenden höheren Bits zusätzlich nachgeladen werden. Durch die Vergrößerung von 4 Bit, über 8 Bit zu 32 Bit konnte dieser Vorgang parallelisiert werden. Vorher geteilte Wörter werden nun in einem Wort gespeichert und geladen.

In den Achtzigern bis zur Mitte der Neunziger Jahre wurde BLP von „Instruction Level Parallelism“ (ILP) abgelöst. Um die einzelnen Teile des Prozessors so gut wie möglich auszulasten bzw. „idle-time“ zu vermeiden, begannen Entwickler einzelne Befehle parallel ausführen zu lassen. Dies bringt jedoch nur einen Performancegewinn, wenn die

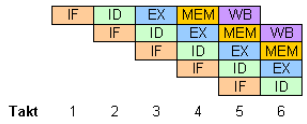

Abbildung 2.5: typische Pipeline einer RISC Architektur

einzelnen Befehle (engl. „instructions“) unabhängig voneinander sind. Pipelining ist ein Beispiel für die Umsetzung von ILP.

Beispielhaft ist eine Pipeline in Abbildung 2.5 dargestellt. Ausgehend von einer RISC Architektur benötigt der Prozessor fünf Taktzyklen um einen Befehl auszuführen. Als erstes muss der Befehl geladen (Instruction Fetch = IF), dann dekodiert (Instruction Decode = ID) und ausgeführt werden (Execute = EX). Unter Umständen muss auf den Speicher gewartet werden (Memory Access = MEM) und zum Schluss wird das Ergebnis in einem Register gespeichert (Write Back = WB).

Ein Prozessor ohne eine Pipeline würde diese einzelnen Ausführungsschritte seriell, also nacheinander ausführen und nach fünf Takten das Ergebnis liefern. Die Ausführung des nächsten Befehls würde ebenfalls fünf Takte dauern. Angenommen die beiden eingehenden Befehle sind voneinander unabhängig, dann könnte man nach dem ersten IF Schritt gleich beginnen den zweiten Befehl zu bearbeiten. Die Verwendung von Pipelining erlaubt es zu jedem Takt ein Ergebnis (hier WB) zu speichern.

Entwickler stoßen aber auch hier an eine Grenze, namentlich die „ILP-Wall“, nämlich genügend Parallelität aus einem Fluss von Instruktionen zu extrahieren, um alle Komponenten des Prozessors mit Arbeit zu versorgen.

2.2.2 Prozesse

Die Software-Entwicklung in den sechziger Jahren unterscheidet sich gravierend von den heutigen Entwicklungsprozessen. Entwickler gaben ihren Code einem Operator im Rechenzentrum in Form von Lochkarten und warteten auf das Ergebnis des Programmlaufs. Nachdem die Lochkarten vom Operator eingelesen wurden, wurde versucht das Programm zu kompilieren. Schon ein vergessenes Komma führte zu einem Abbruch und der Operator begann einen anderen Stapel von Lochkarten einzulesen. Es ist klar, dass das Warten auf ein Ergebnis zu einem erheblichen Zeitverlust führte. Auch nachdem die Lochkarten und der Operator verschwunden waren, konnte nur eine Person zu einem Zeitpunkt an dem Computer arbeiten. Um diesen Umstand für ihre Studenten zu verbessern, haben im Jahre 1961 Carbato et al. am Massachusetts Institute of Technology (MIT) in den USA das „Compatible Time-Sharing System“ (kurz CTSS) entwickelt. In einer ersten Demonstration des Systems konnten drei Benutzer einen Computer gleichzeitig nutzen. Dies ermöglichte eine neue interaktive Benutzung des Computers, die bis heute geblieben ist. Von nun an konnten mehrere Studenten parallel Programme ent-

wickeln und bekamen sofort eine Fehlermeldung, dass das Programm z.B. nicht kompiliert werden konnte. Die Entwicklungszeit konnte so dramatisch gesenkt werden. Das CTSS ist der Vorgänger von dem nie fertiggestellten Betriebssystem Multics aus dessen Entwicklung das heutige Unix bzw. Linux hervorgegangen ist.

Time-Sharing erlaubt das Ausführen mehrerer Prozesse und ist eine Erweiterung des Multiprogrammings. Die gemeinsam genutzte CPU wird für jeden Prozess für eine gewisse Zeitdauer reserviert. Die Dauer ist so gering, dass es für einen menschlichen Benutzer scheint, als ob alle Prozesse parallel ausgeführt würden. Dies stimmt in einem System mit einer einzelnen CPU natürlich nicht. Vielmehr werden die Prozesse nebenläufig ausgeführt. Welcher Prozess von der CPU gerade ausgeführt wird, entscheidet der Scheduler des Betriebssystems. In einem System mit mehreren CPUs können Prozesse dagegen tatsächlich parallel ausgeführt werden.

Ein Prozess wird im Betriebssystem intern als ein „Process Control Block“ (PCB) dargestellt. In dieser Datenstruktur wird der aktuelle Zustand des Programms in Form des Programmzählers, des Inhalts der CPU-Register, der temporären Daten wie dem Stack und den Werten der globalen Variablen gespeichert. Zudem werden Zeiger auf aktuell benutzte Ressourcen, zum Beispiel Dateihandle in dem PCB gespeichert. Ein Prozess beschreibt damit vollständig die Ausführungseinheit eines Programms. Durch diese Abstraktion kann ein Programm mehrmals ausgeführt werden und das Betriebssystem ist in der Lage den PCB eines Prozesses zu pausieren, den PCB zwischenzuspeichern und einen anderen Prozess vom Prozessor ausführen zu lassen. Wann ein Prozesswechsel (engl. context switch) stattfindet, hängt vom Scheduler des Betriebssystems ab.

2.2.3 Threads

Heutige Software ist intern ebenfalls hoch parallelisiert. Ein Schreibprogramm unterstützt zum Beispiel den Benutzer, indem falsch geschriebene Wörter hervorgehoben werden. Damit die Rechtschreibprüfung das Schreibprogramm bei der Ausführung nicht blockiert, wird die Rechtschreibprüfung in einem seperaten Thread (engl. für Faden) ausgeführt. Während der Hauptthread auf die Eingaben vom Benutzer wartet, kann nun der andere Thread den schon geschriebenen Text überprüfen. Threads können als Teilprozesse innerhalb eines Prozesses gesehen werden und können ebenso pausiert und wieder weiterausgeführt werden. Da Threads innerhalb eines Prozesses laufen, können Ressourcen, wie geöffnete Dateien und Speicherbereiche von mehreren Threads genutzt werden, was die Kommunikation innerhalb eines Programms bzw. Prozesses vereinfacht und beschleunigt.

Folgende Vorteile ergeben sich durch die Verwendung von Threads [16]:

- *Ansprechbarkeit*: Führt ein Programm eine lange Berechnung durch, blockiert normalerweise diese Operation das Programm. Wird die Operation in einem Thread ausgeführt, bleibt das Hauptprogramm für den Anwender ansprechbar.

- *Ökonomie*: Es ist für ein Betriebssystem weniger resourcenintensiv einen neuen Thread zu erstellen, als einen neuen Prozess zu generieren. Auch ein Wechsel zwischen Threads ist weniger aufwendig als ein Prozesswechsel. Zusätzlich werden bei einem Prozesswechsel meist die Pipeline und Caches neu gefüllt, was anfangs die Zugriffszeit auf den Speicher vergrößert. Dies entfällt bei der Nutzung von Threads.
- *Ausnutzung paralleler Technologien*: Durch die Entwicklung hin zu Mehr-Prozessor-Systemen können Threads echt parallel ausgeführt werden, was die Abarbeitungsgeschwindigkeit stark steigern kann.

Ein Prozess ist ein vom Betriebssystem bereitgestelltes Konstrukt. Die Verwaltung von Threads innerhalb eines Prozesses kann ebenfalls vom Betriebssystem auf der Kernelebene übernommen werden. Diese Threads heißen dann „Kernelthreads“. Des Weiteren kann die Thread-Funktionalität von einer Bibliothek im Userspace bereitgestellt werden. In diesem Fall werden die Threads auch „Userthreads“ genannt. Normalerweise werden Anwendungen von Benutzern im *Userspace* und Systemprozesse vom Betriebssystem im *Kernelmode* ausgeführt. Zum Schutz vor böswilligen Nutzern dürfen priviligierte Aktionen, wie z.B. ein Prozesswechsel nur im Kernelmode ausgeführt werden. Damit Userthreads von einer parallelen Ausführung auf einer Multi-Core Architektur profitieren können, muss es ein Mapping zwischen Kernel- und Userthreads geben.

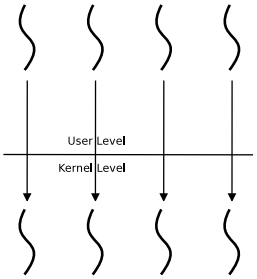

Abbildung 2.6: one-to-one Mapping

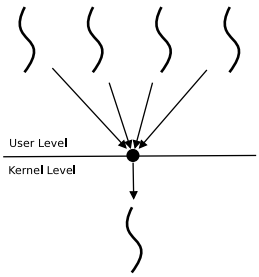

Abbildung 2.7: many-to-one Mapping

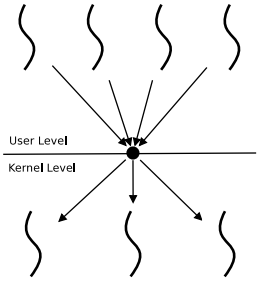

Abbildung 2.8: many-to-many Mapping

Bei einem Mapping von mehreren Userthreads zu einem Kernelthread können die einzelnen Userthreads nicht parallel auf verschiedenen Prozessoren bzw. Kernen ausgeführt werden. Wenn ein Userthread blockiert, wird auch der zugehörige Kernelthread blockiert und das gesammte Programm bleibt stehen. Dies ist in Abbildung 2.7 dargestellt. Das andere Extrem ist eine Eins zu Eins Zuordnung von User- zu Kernelthreads, welches in Abbildung 2.6 dargestellt ist. Dies ermöglicht ein paralleles Abarbeiten der Threads. Doch das Erzeugen und Verwalten von Kernelthreads kann die Performance eines Betriebssystems vermindern. Aus diesem Grund geben Betriebssysteme, die ein solches Mapping verwenden, eine Höchstgrenze von Kernelthreads an. Ein Kompromiss dieser beiden Mappings wird in einem Many-to-Many Mapping verwendet. Hier werden mehrere Userthreads auf weniger oder gleich viele Kernelthreads gemappt. Der Entwickler kann

so viele Userthreads kreieren wie er will und das Betriebssystem kann entscheiden auf wieviele Kernelthreads gemappt werden soll. Dies ist in Abbildung 2.8 dargestellt.

Für die Entwicklung von gethreadeter Software existieren eine Menge von Bibliotheken, die den Software-Entwickler unterstützen. Manche Programmiersprachen besitzen nativ Konstrukte für Threads und Synchronisation wie z.B. Java. Andere weitverbreitete Sprachen wie C benötigen eine Bibliothek wie PThreads. Die POSIX Thread Bibliothek PThreads stellt die Threadingfunktionalität in Form von Funktionen bereit. Der Programmierer muss explizit parallel programmieren, was das Portieren von einem seriellen Programm hin zu einem parallelen Programm erschwert. Auch ist dieses händische parallelisieren sehr fehleranfällig. Einen anderen Ansatz verfolgt die Thread-Bibliothek von OpenMP. Diese Bibliothek verwaltet die Threads für den Programmierer und ermöglicht einen gewichteten Fokus auf das zu parallelisierende Problem. Der Software-Entwickler gibt dem Compiler Hinweise in Form von Compiler-Direktiven (Pragmas), zum Beispiel welche Schleifen parallel ablaufen sollen. Unter Verwendung der OpenMP Bibliothek erzeugt der Compiler dann einen parallelisierten Binärcode.

2.2.4 Cache-Kohärenz und Speicherkonsistenz

Für Multiprozessorsysteme hat die nebenläufige Ausführung von Threads Auswirkungen auf die Speicherverwaltung. Statt nur einem einzigen Cache, müssen mit der Einführung mehrere Kerne in einem Prozzesor, auch mehrere Caches mit dem Hauptspeicher synchronisiert werden. Dies erfordert zusätzlichen Aufwand für die Kommunikation zwischen dem Hauptspeicher und den angeschlossenen Caches. Die Speicherverwaltung hat den Anspruch immer den zeitlich letzten Wert einer Speicherzelle zu liefern. Nur wenn sie dieses Kriterium erfüllt, wird die Speicherverwaltung als kohärent bezeichnet. Zur Erhaltung der Kohärenz existieren moderne Cache-Koheränz-Protokolle, wie etwa das MESI- oder das Snoopy-Protokoll [17].

Kohärenz-Verfahren wirken sich aber nur direkt auf Lese-/Schreibanweisungen. Gängige Prozessoren und Compiler optimieren aber auch den Quellcode durch Umordnung der Anweisungen. So können trotz sichergestellter Cache-Kohärenz Inkonsistenzen auftreten, da das Cache-Koherenz-Protokoll nicht garantiert wann die Auswirkung einer Speicheroperation sichtbar wird. Für die Lösung dieses Problems existieren verschiedene Speicherkonsistenzmodelle. Das Modell der sequentiellen Konsistenz und das der schwachen Konsistenz, sind die beiden wichtigsten Speicherkonsistenzmodelle. Darüber hinaus finden noch das Prozessorkonsistenzmodell und Partial-Store-Ordering Modelle Anwendung.

Das sequentielle Speicherkonsistenzmodell stellt die größten Einschränkungen an die Reihenfolge der durchgeführten Speicherzugriffe. Ein Multiprozessorsystem ist als sequentiell konsistent zu bezeichnen, wenn das Ergebnis einer beliebigen Berechnung dasselbe ist, als wenn die Operationen aller Prozessoren auf einem Einprozessorsystem in einer sequentiellen Ordnung ausgeführt würden [18]. D.h. im sequentiellen Speicherkon-

sistenzmodell werden alle Speicheroperationen als atomare Operationen in der Reihenfolge des des Quellprogramms ausgeführt und zentral sequentialisiert.

Diese Bedingungen des sequentiellen Speicherkonsistenzmodells führt jedoch für nebenläufige Programme zu Effizienzeinbußen. Deshalb wurde das Modell der abgeschwächten Speicherkonsistenz entwickelt, welches die Effizienz steigert, aber gleichzeitig die Semantik des Programms verändern kann. Statt die Konsistenz der Speicherzugriffe zu jeder Zeit zu gewährleisten, wie vom sequentiellen Speicherkonsistenzmodell garantiert, wird dies nur zu bestimmten, vom Programmierer definierten, Synchronisationspunkten zugesichert.

Das abgeschwächte Speicherkonsistenzmodell toleriert somit Speicherinkonsistenzen in Maßen, sodass Optimierung zur Effizienzsteigerung wieder ausgeschöpft werden können. Ist ein Programm frei von Data Races (3.1.1), so ist das Modell der abgeschwächten Speicherkonsistenz sogar äquivalent zu dem der sequentiellen Speicherkonsistenz.

2.2.5 Gewinn und Grenzen

$$speedup = \frac{1}{(1-f) + \frac{f}{n}} \tag{2.1}$$

Angenommen ein Prozessor soll zwei Zahlen A und B addieren und danach die beiden Zahlen multiplizieren. Ein einzelner Prozessor würde erst $A + B$ rechnen und dann im zweiten Schritt $A * B$. Wenn zwei Rechenkerne zur Verfügung stehen, kann ein Kern die Addition und der andere Kern parallel dazu die Multiplikation ausführen. Die Ergebnisse der beiden Rechnungen wären nach einem Rechenschritt vorhanden. Die parallele Berechnung wird nun in der Hälfte der ursprünglich benötigten Zeit ausgeführt.

Leider ist nicht jedes Problem vollständig parallelisierbar. Es gibt meistens noch einen seriellen Anteil, der nicht durch zusätzlich verfügbare Ressourcen schneller abgearbeitet werden kann. Um welchen Faktor und mit welcher Anzahl von Recheneinheiten ein Problem schneller berechnet werden kann, beschreibt das Gesetz von Gene M. Amdahl [19], welches in Formel 2.1 angegeben ist. Der Zusammenhang von Recheneinheiten n, parallelem Anteil f, seriellem Anteil $1 - f$ und Geschwindigkeitszuwachs $speedup$ wurde 1967 von Ihm auf einer Konferenz vorgestellt. Aus der Abbildung 2.9 ist ersichtlich, dass sich der Einsatz von mehreren Recheneinheiten nur lohnt, wenn das Problem zu einem großen Anteil parallelisierbar ist. Es wird angenommen, dass die hohe Parallelität keinen Mehraufwand an Kommunikation erzeugt. Die einzelnen Kurven zeigen den Anstieg vom Speedup wenn 10%, 20% bis 95% des Problems parallelisierbar sind.

Um den seriellen Anteil so schnell wie möglich abarbeiten zu können, besitzt der neuste Intel Multi-Core Prozessor Codename „Nehalem“ einen Turboboost Modus, der dafür sorgt, dass ein einzelner Core für kurze Zeit höher getaktet wird. Der Stromverbrauch des gesamten Prozessors bleibt gleich, da die anderen Cores für diese Zeit niedriger getaktet werden. Die Instruktionen können durch die höhere Taktrate des einzelnen Kerns

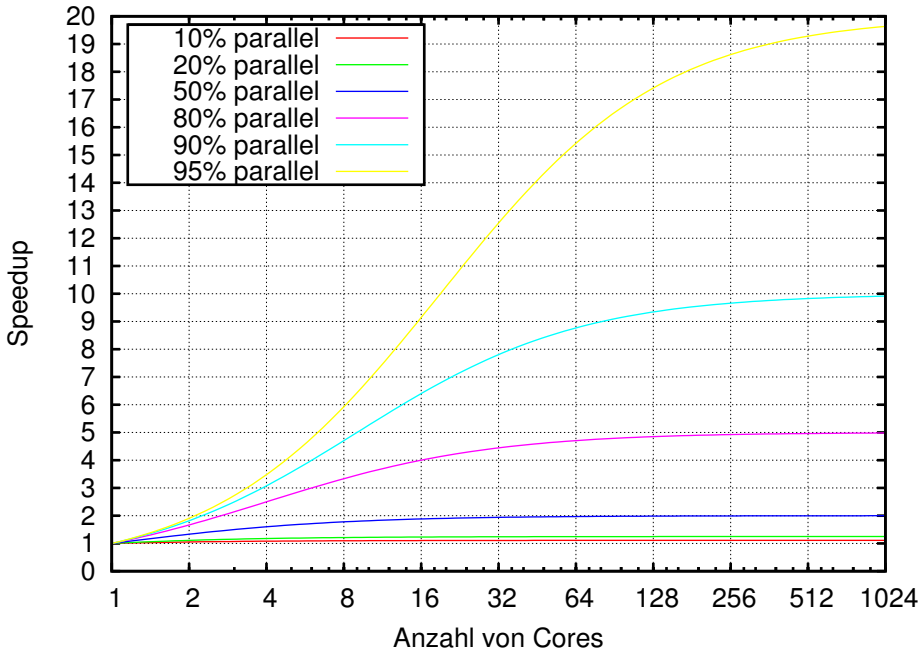

Abbildung 2.9: Speedup gegenüber Anzahl von Cores

schneller abgearbeitet werden. Vielleicht ist dieser Ansatz ein erster Schritt in eine asymmetrische Prozessortechnik mit speziellen Recheneinheiten für die parallele und serielle Ausführung von Instruktionen.

2.3 Zusammenfassung

In diesem Kapitel wurden die Grundlagen für eine detaillierte Diskussion von Defekten gelegt, die insbesondere in einer parallelen Ausführung auftreten. Es wurde erläutert, warum der Trend hin zu mehr Kernen auf einem Prozessor nicht mehr aufzuhalten ist, bzw. ein Erhöhen der Taktraten jenseits von 4 GHz physikalisch schwer zu beherrschen ist.

Von der untersten Hardwareebene bis hin zu Parallelität innerhalb von Prozessen, die wiederum parallel ausgeführt werden, ist Parallelität nun auf allen Ebenen eines Computers vorhanden. Ein weiteres Erhöhen der Performance kann nur noch durch Software geschehen, die die vorhandene Parallelität voll ausnutzt.

Dass Software in einer seriellen Welt nicht fehlerfrei entwickelt werden kann wurde in diesem Kapitel ebenfalls festgestellt. Neben einer Klassifizierung der Defekte in Bohr-, Mandel- und Heisenbugs wurde die wissenschaftliche Methode des Debuggings vorgestellt. Gerade Heisenbugs sind mit klassischen Debuggingsmethoden in einer parallelen

Welt nicht mehr zu lokalisieren und zu entfernen.

Kapitel 3

Lösungsansätze für Concurrency Bugs

In diesem Kapitel wird eine Unterklasse der Heisenbugs eingeführt. Nach einer genaueren Erklärung ausgewählter Defekte, werden existierende Lösungsansätze vorgestellt. Eine kurze Zusammenfassung einer Studie über die Relevanz dieser Defekte folgt. Für einen ausgewählten Defekt wird zudem ein Entwurf für dessen Detektion präsentiert.

3.1 Was sind Concurrency Bugs?

Das englische Wort für Nebenläufigkeit lautet „concurrency“. Demnach sind *Concurrency Bugs* Defekte, die durch eine nebenläufige Ausführung aktiviert werden. Diese Defekte sind Teil der Klasse der Heisenbugs. Sie verhalten sich nicht-deterministisch und treten relativ selten auf. Zu ihnen gehören Data Races, Atomicity Violations und Deadlocks, die wir im Folgenden kurz vorstellen.

3.1.1 Data Race

Definition 1. *Ein Data Race existiert, wenn zwei nicht synchronisierte Zugriffe auf eine geteilte Ressource nebenläufig stattfinden, wobei mindestens einer der Zugriffe schreibend ist.*

In Abbildung 3.1 ist eine solche Situation dargestellt. Diese ist auch unter dem Namen „Lost Update Problem“ bekannt, da hier die Veränderung eines Threads durch einen anderen überschrieben wird und damit ein Wert verloren geht. Es wird die Variable X nebenläufig von Thread $Thread_A$ und Thread $Thread_B$ verändert. Beide Threads lesen den aktuellen Wert von X und erhöhen diesen um den Wert 1. Da die Zugriffe nicht

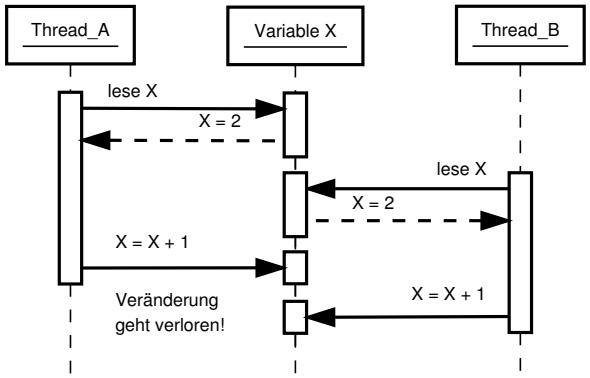

Abbildung 3.1: Ein Data Race

synchronisiert sind, wird die Erhöhung von $Thread_A$ im nächsten Schritt von $Thread_B$ überschrieben. Die Erhöhung des Wertes von $Thread_A$ geht damit verloren.

3.1.2 Atomicity Violation

Definition 2. *Eine Atomicity Violation existiert, wenn für das korrekte Arbeiten eine Menge von Anweisungen atomar ausgeführt werden muss, diese aber durch eine andere Anweisung gestört wird.*

In Abbildung 3.2 ist eine solche Situation dargestellt. $Thread_A$ stellt eine Anfrage an eine Datenbank. Zu allererst wird eine Verbindung zur Datenbank hergestellt ($db\_create\_handle()$). Nachdem die Verbindung zur Datenbank hergestellt wurde, wird eine Anfrage an die Datenbank gestellt ($db\_query(handle, query)$). Zum Schluss wird das Resultat über die Verbindung angefordert ($db\_result(handle)$). Kurz bevor $Thread_A$ aber das Resultat einfordert, beendet $Thread_B$ die Verbindung ($db\_disconnect(handle)$. Der danach folgende Zugriff von $Thread_A$ führt jetzt zum Abbruch, da die Verbindung nicht mehr existiert. Die Anweisungen von $Thread_A$ hätten atomar, also ununterbrochen in einem Block ausgeführt werden müssen.

Das schwierige an Atomicity Violations ist die ungenaue Definition, wodurch ein automatisches Lokalisieren solcher Verstöße nicht möglich ist. Welche Anweisungen zusammengehören, kann nur der Entwickler wissen. Man stelle sich nur eine Datenstruktur vor, die einer verketteten Liste entspricht. Für einen Performancegewinn wurde ein globaler Zähler eingeführt, der die aktuelle Anzahl der Elemente angibt. Bei dem Löschen oder Einfügen von Elementen in die Liste muss auch der Zähler aktualisiert werden. Der Programmierer muss wissen, dass es diesen Zähler gibt und welche semantische Bedeutung dieser besitzt. Falls dies nicht geschieht befindet sich das System in einem inkonsistenten Zustand.

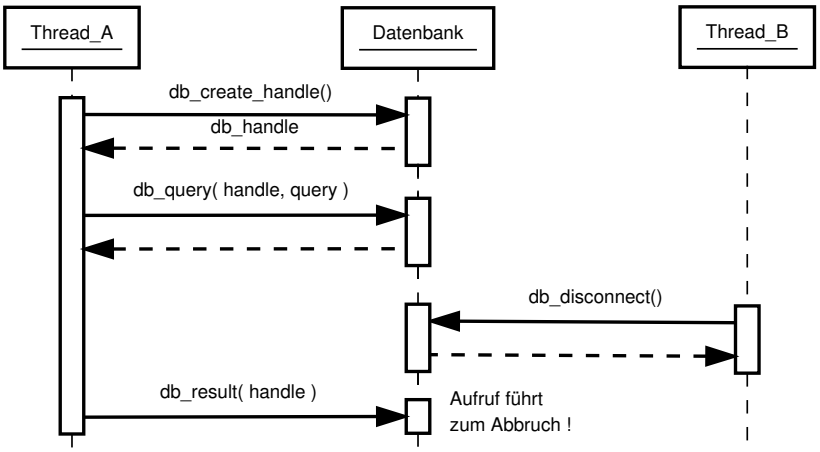

Abbildung 3.2: Eine Atomicity Violation

3.1.3 Deadlock

Definition 3. *Eine Menge von Threads befindet sich in einem Deadlock-Zustand, wenn jeder Thread auf die Freigabe einer Ressource wartet, die nur von einem anderen Thread aus der Menge freigegeben werden kann.*

Damit Defekte wie Data Races und Atomicity Violations nicht aktiviert werden, werden Synchronisationsanweisungen verwendet. Die richtige Verwendung dieser Anweisungen führt zu einem geregelten Zugriff auf geteilte Ressourcen. Sie erlauben einen wechselseitigen Ausschluss, damit nur ein Teilnehmer oder eine ausgewählte Gruppe Zugriff auf die Ressource hat. Die Ressource kann auch eine Menge von Anweisungen sein. Dann wird auch von der „critical section“ (engl. für Kritischer Bereich) gesprochen.

Es existieren mehrere Möglichkeiten um die Synchronisation von Threads zu steuern. Zum Einen existieren reine Softwarelösungen, die der Programmierer selbst implementieren muss und zum Anderen bereitgestellte Konstrukte, welche auf besondere Befehlsunterstützung des Prozessors zugreifen.

Der *Peterson Algorithmus* erlaubt das wechselseitig ausschliessende Betreten eines kritischen Bereiches von zwei Threads. Der Algorithmus ist in Listing 3.1 und 3.2 dargestellt. Es ist wichtig anzumerken, dass der Algorithmus für den jeweiligen Thread verschieden ist. Es ist eine reine Softwarelösung und geht davon aus, dass zur gleichen Zeit nur ein Schreib- bzw. Lesebefehl auf eine Speicherzelle stattfinden kann, was nicht für alle Speichercontroller gelten muss.

Die beiden Threads $Thread_i$ und $Thread_j$ kommunizieren über eine Variable $turn$, welche angibt, welcher Thread den kritischen Bereich betreten darf und über ein Array $flag[2]$, welche die Threads speichert, die bereit sind den kritischen Bereich zu betreten. $Thread_i$ will nun den Bereich betreten und setzt als erstes $flag[0]$ auf 1 und und

```
flag[0]   = 0;
flag[1]   = 0;
turn;

flag[0] = 1;
turn = 1;
while (flag[1] == 1 && turn == 1)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[0] = 0;
```

Listing 3.1: Peterson Algorithmus für $Thread_i$

```
flag[0]   = 0;
flag[1]   = 0;
turn;

flag[1] = 1;
turn = 0;
while (flag[0] == 1 && turn == 0)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[1] = 0;
```

Listing 3.2: Peterson Algorithmus für $Thread_j$

$turn$ auf 1. Damit wird dem anderen Thread signalisiert, dass dieser den Bereich betreten darf. Wenn in der Zwischenzeit $Thread_j$ den kritischen Bereich betreten hat, wartet $Thread_i$ solange bis dieser den Bereich verlassen und $flag[1]$ auf 0 gesetzt hat. Wenn beide Threads zur gleichen Zeit den kritischen Bereich betreten wollen, entscheidet die Logik des Speichercontrollers, wessen Wert in der Variable $turn$ gespeichert wird. Diese Lösung enthält ein Data Race, dessen Aktivierung bewusst gewollt ist.

Eine weitaus bekanntere Methode für das Synchronisieren von Threads ist das Verwenden von Locks. Bei korrekter Verwendung erlaubt ein Lock (engl. für Schloss) das Betreten eines kritischen Bereiches nur dem Besitzer eines Schlüssels. Wenn ein Thread den kritschen Bereich verlassen hat, wird der Schlüssel wieder freigegeben und ein anderer kann sich den Schlüssel nehmen und versuchen den Bereich zu betreten.

Für dieses Vorgehen werden Semaphore verwendet. Ein Semaphore ist eine Integer Variable, auf welche nur indirekt zugegriffen werden darf. Wenn der Wert der Semaphore zwischen Null und Eins liegt, wird diese auch Mutex genannt, da die Semaphore für den wechselseitigen Ausschluss verwendet werden kann. Wenn der Wert der Semaphore zwischen 0 und n liegen darf, so wird von einer „counting semaphore“ gesprochen. In diesem Fall dürfen dann maximal n Teilnehmer den kritischen Bereich betreten. Eintritt in den kritischen Bereich erlangt man mit dem Methodenaufruf $wait()$. Nach dem Verlassen wird dies den anderen Teilnehmern durch den Aufruf von $signal()$ mitgeteilt.

- *wait(S)* Bei einem Aufruf dieser Methode wird so lange gewartet, bis der Wert von S kleiner gleich $n - 1$ ist. Danach wird S um Eins verringert. Dies stellt sicher, dass maximal n Teilnehmer den kritischen Bereich betreten können.
- *signal(S)* Dies gibt einen Schlüssel für den kritischen Bereich wieder frei. Dazu wird S um den Wert 1 erhöht.

Das Betriebssystem muss sicherstellen, dass nicht mehrere Threads parallel $wait(S)$

oder $signal(S)$ aufrufen. Die Methoden müssen also atomar ausgeführt werden. Dazu verwendet das Betriebssystem vom Prozessor bereitgestellte Anweisungen, wie z.B. $testAndSet(a)$ oder $swap(a, b)$, mit deren Hilfe das Problem des kritischen Bereiches ähnlich wie der Peterson Algorithmus in Hardware gelöst werden kann.

Zum Einen ist das richtige Verwenden von Synchronisationsmechanismen nicht trivial und zum anderen kann das Verwenden selbst zu neuen Defekten führen. Neben einem Livelock, in dem das System arbeitet aber keinen globalen Fortschritt stattfindet, spielt der Deadlock (engl. für tödliche Verklemmung) eine große Rolle.

Anhand von fünf im Kreis sitzenden Philosophen, deren Leben aus Denken und Essen besteht, hat E. W. Dijkstra in [20] als Erster das Problem des Deadlocks erklärt. Das Problem der fünf speisenden Philosophen besteht darin, dass zum Essen jeweils zwei Gabeln benötigt werden, aber nur fünf Gabeln auf dem Tisch liegen. Eine naive Implementierung eines Philosophen würde zum Beispiel die linke und dann die rechte Gabel greifen und nach einer Weile wieder freigeben. Wenn alle Philosophen nun parallel die linke Gabeln ergreifen, dann würde keine Gabel mehr auf dem Tisch liegen und jeder Philosoph würde ewig auf das Freiwerden der rechten Gabel warten. Wir hätten also einen Deadlock.

```
1  void incorrect_increment( )
2  {
3    pthread_mutex_lock( &mutex );
4    global_cnt = global_cnt + 1;
5    // pthread_mutex_unlock( &mutex );
6  }
```

Listing 3.3: Potentieller Livelock

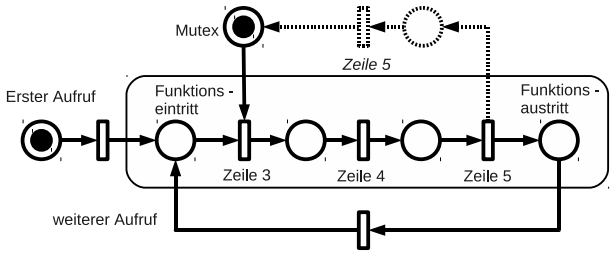

Abbildung 3.3: Potentieller Livelock

Das Listing 3.3 enthält den Quelltext der fehlerhaften Funktion $incorrect\_increment()$, welches eine potentielle Livelock Situation enthält. Die Funktion soll eine geteilte Ressource $global\_cnt$ sicher um den Wert Eins erhöhen. Dazu wird eine Synchronisationsprimitive Mutex $mutex$ verwendet, welches in Zeile 3 erlangt wird. Nach dem Erhöhen wird das Mutex aber nicht wieder freigegeben, da diese Aktion in Zeile 5 auskommentiert ist. Erneutes Aufrufen der Funktion $incorrect\_increment()$ führt zu einem Liveock, da jetzt die Anweisung in Zeile 3 auf das Freiwerden des Mutexes wartet. Die Ausführung, welche zu einem Livelock führt, ist in Abbildung 3.3 in Form eines Petri Netzes dargestellt. Der gestrichelte Pfad stellt die auskommentierte Zeile 5 dar. Es ist ersichtlich, dass zum Schalten der Transition Zeile 3 auf der Stelle des Mutexes eine Marke liegen muss. Nach einmaligem Ausführen, wird diese Marke aber nicht mehr zurückgelegt, wodurch die Transition nicht mehr schalten kann. Die Funktion wartet nun bis zum Programmabbruch, auf das Freiwerden des Mutexes und verrichtet keine produktive Arbeit.

3.1.4 Concurrency Bugs in Software

Ob Concurrency Bugs in heutiger Software überhaupt eine Rolle spielen und unter welchen Bedingungen Concurrency Bugs in Software aktiviert werden, wurde 2008 von Shan Lu et al. in einer Studie näher untersucht [21].

Von 105 untersuchten Defekten, führten 34 zu einem Systemabsturz. Weitere 37 haben zu einem Stillstand des Systems geführt. Dies bestätigt, dass Concurrency Bugs die Zuverlässigkeit von Software erheblich beeinträchtigen können und für den Einsatz in sicherheitskritischen Systemen eine große Bedrohung bedeuten.

Die Untersuchung hat ergeben, dass meist (in 101 von 105 Fällen) nicht mehr als zwei Threads an der Aktivierung des Defektes beteiligt waren. Der angegebene Grund dafür ist, dass Threads untereinander nicht viel kommunizieren. Meist kreiert ein Masterthread mehrere Arbeiterthreads, welche nach Beendigung ihr Ergebnis dem Masterthread mitteilen. Dies ist ein Argument gegen das Ausführen von Software unter schwerer Last (auch unter dem Namen „Stresstesten“ bekannt) um Concurrency Bugs zu aktivieren, da durch den erhöhten Arbeitsaufwand die Kommunikation der Threads nicht verändert wird.

Eine weitere Erkenntnis der Untersuchung ist, dass bei 66% der Nicht-Deadlock Defekte nur eine Variable involviert war. Um einen Großteil von Data Races zu erkennen, reicht also das Beobachten einer Variablen.

Eine weitere wichtige Erkenntnis der Studie ist, dass 90% der untersuchten Defekte sich deterministisch reproduzieren bzw. aktivieren lassen, wenn eine bestimmtes Interleaving vom Prozessor ausgeführt wird. Weitere 97% der Deadlock Defekte lassen sich reproduzieren, wenn eine zeitliche Ordnung von maximal vier Synchronisationsanweisungen eingehalten wird.

3.2 Existierende Verfahren

3.2.1 Kategorisierung existierender Ansätze

Neben neuen Methoden der Software-Entwicklung existieren auch eine Menge von Softwarewerkzeugen, die den Entwickler bei dem Testen und Debuggen von Software unterstützen. Diese Werkzeuge untersuchen entweder den Quelltext statisch oder benötigen wenigstens einen Ausführungslauf der Software, welcher dann dynamisch zur Laufzeit untersucht wird, oder postmortem, nachdem das Programm terminiert hat.

Der Vorteil von statischen Analyseverfahren ist, dass das Programm nicht bei jedem Testlauf neukompliliert und gestartet werden muss. Die Analyse kann im Hintergrund stattfinden. Leider benötigen viele dieser Werkzeuge Hinweise vom Entwickler in Form von

Annotationen. Den Quelltext einer größeren, bestehenden Software mit solchen Annotation zu versehen ist mit großem Aufwand verbunden, was für viele Unternehmen gegen einen Einsatz solcher Analysewerkzeuge spricht [22].

Dynamische Analysewerkzeuge untersuchen das Programm während der Laufzeit. Dies geschieht meist mit Hilfe eines Traces, der ausgewählte Ereignisse protokolliert, welche dann von dem Werkzeug analysiert werden. Die größten Nachteile hierbei sind zum einen die große Menge an Daten, die ein Werkzeug potentiell untersuchen können muss, und die Frage, wie der Programmtrace überhaupt erstellt wird.

3.2.2 Data Races erkennen - Lockset-Algorithmus

Dass Defekte wie Data Races in multi-threaded Software vorhanden sind, ist schon seit mehr als 40 Jahren bekannt. Schon 1974 wurde von Hoare das Konzept des Monitors vorgestellt um Data Races zu vermeiden [23].

Ein Monitor ist eine Gruppe von geteilten Ressourcen und zugehörige Zugriffsmethoden. Wenn ein Thread eine der Ressourcen verändern oder lesen will, geschieht dies durch die Methoden. Intern wird für den Zugriff nur ein Lock bzw. Mutex verwendet, was bei dem Eintritt in die Methode akquiriert und beim Austritt wieder freigegen wird. Diese Vorgehen garantiert eine Serialisierung der Zugriffe auf die geteilten Ressourcen und es kann kein Data Race stattfinden.

Eine Möglichkeit Data Races in einem Programmlauf zu detektieren ist die Analyse der zeitlichen Reihenfolge von Zugriffen verschiedener Threads auf geteilte Ressourcen. Dazu wird Lamports *happend-before* Relation verwendet [24]. Zugriffe innerhalb eines Threads werden in der Reihenfolge geordnet, in der diese ausgeführt werden. Zugriffe verschiedener Threads werden in der Reihenfolge geordnet, welche die Synchronisationsanweisungen erlaubt. Wenn $Thread_A$ ein Mutex akquiriert und $Thread_B$ danach versucht ebenfalls das Mutex zu akquirieren, dann wird gesagt, dass der Zugriff auf das Mutex von $Thread_A$ *happend-before* dem Zugriff von $Thread_B$. Um eine solche Aussage treffen zu können, muss aber die Semantik der Synchronisationsanweisung mit in Betracht gezogen werden. Vielleicht erlaubt die Anweisung auch den Zugriff von mehr als einem Thread parallel, dann wäre die obige Aussage falsch. Wenn für eine Menge von Zugriffen von verschiedenen Threads auf eine geteilte Ressource keine *happened-before* Relation aufgestellt werden kann, ist ein Data Race möglich. Neben dem Aufwand die Zugriffe und Aktionen aller Threads zu protokollieren, hat dieses Verfahren einen weiteren Nachteil.

Unter Umständen werden potentielle Data Races wie in Abbildung 3.4 nicht erkannt. Ein anderer Schedule der Threads könnte dazu führen, dass $Thread_B$ vor $Thread_A$ auf die gemeinsame Ressource Y zugreift. Es hängt also von der Ausführung des Programms ab, ob ein happened-before Ansatz ein Data Race erkennt oder nicht.

Der von Savage et al. vorgeschlagener *Lockset Algorithmus* ist in der Lage solche Data

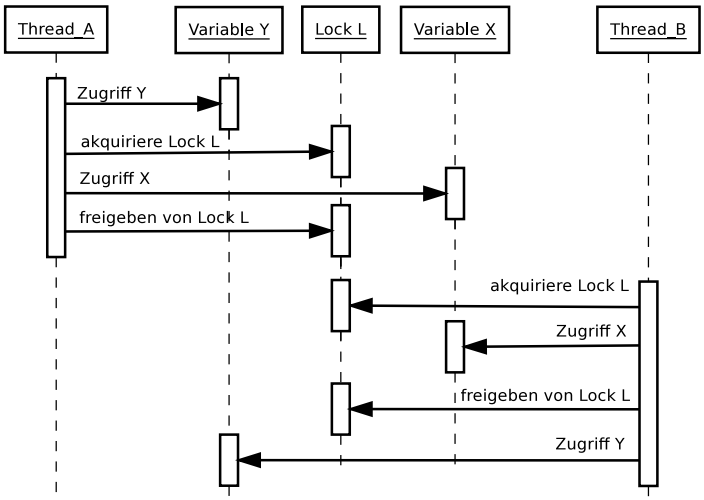

Abbildung 3.4: Data Race auf Variable Y wird mit einem happened-before Ansatz nicht detektiert

Races trotzdem zu erkennen. Hier wird nicht von der zeitlichen Ordnung ausgegangen, sondern überprüft, ob Zugriffe auf geteilte Ressourcen vom Programmierer explizit synchronisiert werden. In dem Programm *Eraser* wurde dieser Ansatz umgesetzt.

Bei dem Verwenden von Synchronisierungsanweisungen existiert kein direkter Zusammenhang zwischen den Anweisungen und den zu schützenden Ressourcen. Diese Zusammenhänge werden von Eraser aus der Ablaufgeschichte des Programms abgeleitet. In Listing 3.4 ist der einfache Lockset-Algorithmus dargestellt. Eraser verwendet eine erweiterte Form, da der einfache Algorithmus ein sehr hohe False-Positive Rate aufweist. Der verbesserte Algorithmus ignoriert Zugriffe, die geteilte Ressourcen initialisieren. Zudem wird das ausschließliche Lesen von verschiedenen Threads nicht als Data Race interpretiert.

```
/* vor dem Programmstart */
let locks_held(t) be the set of locks held by thread t
for each v, initialize C(v) to the set of all locks

/* während der Laufzeit */
On each access to v by thread t
    set C(v) := C(v) ∩ locks_held(t)
    if C(v) == {} then issue warning.
```

Listing 3.4: einfacher Lockset-Algorithmus

Anhand der Situation in Abbildung 3.4 wird der Lockset-Algorithmus zur Erklärung beispielhaft angewendet.

- Zu allererst werden die Mengen $locks\_held(Thread_A)$ und $locks\_held(Thread_B)$ auf $\{\}$ gesetzt.

- Im nächsten Schritt werden alle geteilten Ressourcen mit allen existierenden Locks initialisiert. Da nur das Lock l existiert werden $C(x)$ und $C(y)$ auf $\{l\}$ gesetzt.
- Nach der Initialisierung wird nun das Programm gestartet und der Programmlauf untersucht. $Thread_A$ greift auf Variable y zu. $C(y) \cap locks\_held(Thread_A) = \{\}$. Die Locking-Disziplin wurde verletzt, da der Zugriff nicht durch ein Lock gesichert wurde. Demnach wird eine Warnung ausgegeben.
- Im nächsten Schritt akquiriert $Thread_A$ das Lock l und greift danach auf die Variable y zu. $C(x) \cap locks\_held(Thread_A) = \{l\}$. Dieser Zugriff ist durch ein Lock gesichert, also wird keine Warnung ausgegeben.
- Mit den Zugriffen von $Thread_B$ verhält es sich analog.

Ein weiterer Vorteil des Lockset-Algorithmus ist, dass eine Warnung ausgegeben wird, wenn eine geteilte Variable von verschiedenen Locks gesichert wird. Dies ist in unserem Beispiel nicht dargestellt.

3.2.3 Erkennen von Atomicity Violations - CTrigger

Atomicity Violation sind in heutiger Software weit verbreitet [21], da Programmierer noch immer sequentiell denken. Es wird oft von Programmierern implizit angenommen, dass Coderegionen atomar ausgeführt werden. Auch wenn komplexere Synchronisationsanweisungen wie *Transaktional Memory* für die Serialisierung dieser Anweisungen verwendet werden, sind Atomicity Violations immer noch möglich, da Programmierer die Gruppe der Anweisungen teilen und unterschiedlichen Transaktionen zuweisen können.

Damit Defekte wie Atomicity Violations aktiviert und detektiert werden können, muss das zu testende Programm mit Eingaben so stimuliert werden, dass der Programmfluss die Defekte Coderegion erreicht. Techniken für das Bestimmen von *bug-triggering-inputs* für sequentielle Programme sind in der Testszene schon länger bekannt [25]. Damit in einem multi-threaded Programm aber ein Defekt aktiviert wird, muss der Prozessor zusätzlich ein aktivierendes Thread-Interleaving ausführen. Erst wenn diese beiden Bedingungen erfüllt sind, kann im nächsten Schritt der Ausfall detektiert werden.

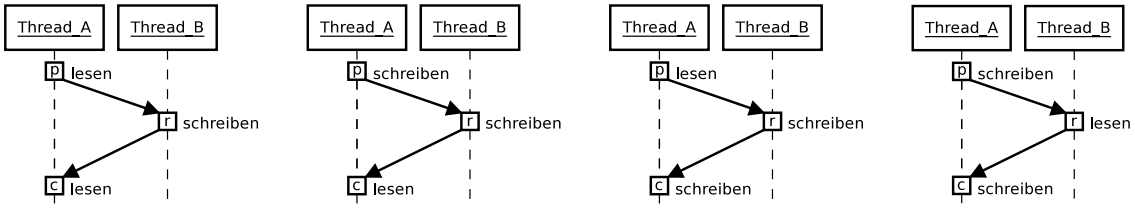

Abbildung 3.5: Vier von acht Interleavings, welche nicht serialisierbar sind.

Definition 4. *Serialisierbarkeit ist definiert als Eigenschaft einer Menge von Anweisungen, deren Wirkung auf Daten bei einer parallelen Ausführung gleich der einer sequentiellen Ausführung ist.*

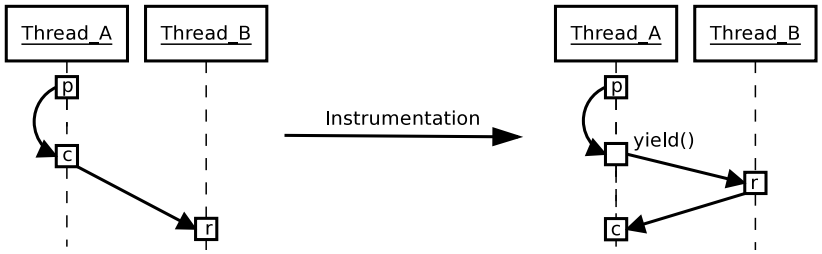

Abbildung 3.6: Veränderung des Zeitverhaltens eines Interleavings durch einen expliziten Kontextwechsel.

Atomarität kann auch als Serialisierbarkeit verstanden werden. Nicht-serialisierbare Interleavings sind nach der Definition also Abläufe von Answeisungen, die nebenläufig ausgeführt eine andere Auswirkung auf Daten haben als eine sequentiellen Ausführung. In [26] werden acht elementare Interleavingmuster aufgezählt, wovon vier nicht-serialisierbar sind. In Abbildung 3.5 ist die Ausführung von den drei beteiligten Anweisungen P, R und C dargestellt. Jede der Andweisungen kann entweder ein Schreib- oder Lesezugriff auf eine geteilte Ressource sein. Der aktuell ausgeführten Anweisung C (C steht hierbei für *current*) geht die Anweisung P voraus (P steht für *preceding*). Die Ausführung dieser beiden Anweisungen wird durch die R Anweisung (R steht für *remote*) unterbrochen.

„CTrigger“ [27] ist ein Framework, das gezielt Thread-Interleavings ausführt, welche mit einer geringen Wahrscheinlichkeit in einem gewöhnlichen Testlauf ausgeführt werden. Diese Interleavings entsprechen den oben dargestellten vier elementaren Interleavingmustern. Das Testframework arbeitet in folgenden zwei Phasen:

- In der ersten Phase werden Interleavings mit hoher Wahrscheinlichkeit ausgeführt und Anweisungen im Programmcode identifiziert, die durch zeitliche Variation zu nicht-serialisierbaren Interleavings mutieren könnten. Diese Menge von potentiellen Testkandidaten wird reduziert, da die Ausführung von einigen Interleavings durch Synchronisationsanweisungen nicht möglich ist. Würde das Testframework versuchen ein solches Interleaving zu erzwingen, würden wir einen Fehler detektieren, der kein Fehler ist. Danach werden die Kandidaten der Ausführungswahrscheinlichkeit nach sortiert, sodass nicht-serialisierbare Interleavings mit geringer Wahrscheinlichkeit hoch eingestuft werden. Es wird davon ausgegangen, dass die gering eingestuften Interleavings schon in anderen Testläufen während des Testens ausgeführt wurden.
- In der zweiten Phase instrumentiert das Framework das zu testende Programm, sodass die hoch eingestuften Interleavings ausgeführt werden. Dazu wird das Zeitverhalten der Ausführungen der einzelnen Threads durch das Einfügen von *sleep()* und *yield()*, also expliziten Wartezeiten und Kontextwechseln verändert. Dies ist in Abbildung 3.6 dargestellt. Ein klassischer Programmtest, wie er auch bei dem Testen von sequentiellen Programmen verwendet wird, entscheidet nach der Instrumentation ob die Ausführung korrekt oder fehlerhaft ist.

Der hier vorgestellte Ansatz stellt eine Alternative zum klassischen Stresstesten [28] dar. Das zu testende Programm wird gezielt auf Binärebene instrumentiert, sodass häufig nicht-serialisierbare Interleavings ausgeführt werden. Laut eigenen Angaben konnte das Testframework „CTrigger“ innerhalb von vier Minuten Defekte aktivieren, die auch nach mehreren Tagen des klassischen Stresstestens nicht aktiviert wurden.

3.2.4 Potentielle Deadlocksituationen erkennen - Helgrind

Um Data Races und Atomicity Violations zu verhindern werden Synchronisationsanweisungen verwendet. Wie eingangs erwähnt, birgt aber auch dies Gefahren, wie z.B. Live- oder Deadlocks.

Unter Linux ist die Verwendung von „Helgrind“ für das Aufspüren von Concurrency Bugs in Software weit verbreitet. Helgrind setzt auf das Instrumentierungsframework „Valgrind“ [29] auf. Im nächsten Kapitel wird näher auf die Funktionsweise von Valgrind eingegangen.

Helgrind ist ein dynamisches Analysewerkzeug, das während der Laufzeit des zu untersuchenden Programms die korrekte Verwendung der Posix Threadbibliothek überwacht. Wenn zum Beispiel ein Lock akquiriert wurde, aber nicht mehr freigegeben wird, wird der Programmierer darüber informiert. Neben der eben genannten Überwachung wird auch die Reihenfolge des Akquirierens und Freigebens von Locks protokolliert. Die Zugriffsreihenfolge wird in einem Lock-Order-Graphen dokumentiert.

Bei jeder Akquisition eines Locks fügt Helgrind eine Kante in den Lock-Order-Graphen ein. Wird dabei ein Kreis im Graph festgestellt, warnt Helgrind vor einem Deadlock. Es ist wichtig anzumerken, dass keine Kanten aus dem Graphen entfernt werden. Dies erlaubt Helgrind in „die Vergangenheit zurück zu blicken“ und so den Entwickler vor potentielle Deadlocks zu warnen.

```
void thread_a( )
{
  pthread_mutex_lock( &l1 );
  pthread_mutex_lock( &l2 );

  ressource_r = ressource_r + 1;

  pthread_mutex_unlock( &l2 );
  pthread_mutex_unlock( &l1 );
}
```

Listing 3.5: Ausführung von $Thread\_A$

```
void thread_b( )
{
  pthread_mutex_lock( &l2 );
  pthread_mutex_lock( &l1 );

  ressource_r = ressource_r + 1;

  pthread_mutex_unlock( &l1 );
  pthread_mutex_unlock( &l2 );
}
```

Listing 3.6: Ausführung von $Thread\_B$

Angenommen $Thread_A$ greift auf die Ressource R zu. Diese ist durch die Locks L_1 und L_2 gesichert wie in Listing 3.5 dargestellt ist. Bei der Akquisition der Locks wird jeweils

eine Kante in den Graphen eingefügt. Dies ist in Abbildung 3.7 dargestellt.

Nach der Nutzung der Ressource R gibt $Thread_A$ die beiden Locks wieder frei. Zu einem späteren Zeitpunkt will $Thread_B$ auf die Ressource R zugreifen. Auch $Thread_B$ akquiriert die Locks L_1 und L_2, aber in verkehrter Reihenfolge. Siehe dazu Listing 3.6. Bei der Akquisition von Lock L_2 durch $Thread_B$ warnt Helgrind nun den Programmierer vor einem *potentiellen* Deadlock, da ein Kreis im Lock-Order-Graphen festgestellt wurde. Dies ist in Abbildung 3.8 dargestellt.

Es ist ein potentielles Deadlock, da sich das zu untersuchende Programm momentan nicht in einem Deadlockzustand befindet. Will aber $Thread_A$ wieder auf die Ressource R zugreifen will, wird $Thread_A$ auf das Freiwerden von Lock L_2 und $Thread_B$ auf das Freiwerden von $Lock_1$ warten, was der weiter oben gegebenen Definition eines Deadlocks entspricht.

Das Verwenden des Lock-Order-Graphen von Helgrind ähnelt sehr dem Ansatz von „Eraser“, wobei hier nicht die Locking-Disziplin, sondern die Locking-Order-Disziplin überwacht wird.

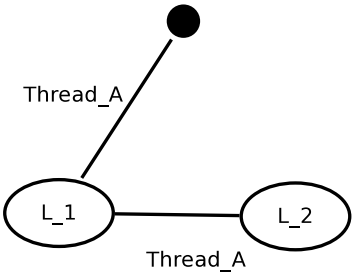

Abbildung 3.7: $Thread_A$ akquiriert erst Lock L_1 und danach L_2

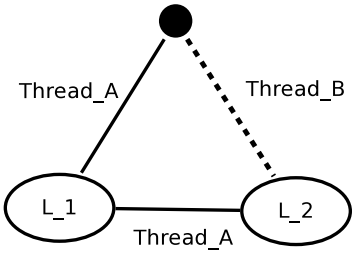

Abbildung 3.8: $Thread_B$ akquiriert Lock L_2 und erzeugt einen potentiellen Deadlock

3.3 Erkennen von Lost-Updates

In diesem Abschnitt wird ein Lost-Update Detektor entworfen und seine Anwendbarkeit abgeschätzt. Ein Lost-Update ist eine besondere Form eines Data Races:
Definition 5. *Ein Lost-Update hat stattgefunden, wenn mindestens zwei Threads den Wert einer Variable nebenläufig erhöhen, also den alten Wert lesen und dann erhöht zurückschreiben, und einer der Threads das Update des anderen überschreibt.*

3.3.1 Entwurf eines Lost-Update Detektors

Ausgehend von der oben genannten Definition eines Data Races, werden in dem hier beschriebenen Detektor während der Laufzeit die Schreib- und Lesezugriffe auf geteilte

Speicherbereiche beobachtet.

Wenn eines der Zugriffsmuster in Abbildung 3.9 den Zugriffen des Programms entspricht, wird die Situation als potentielles Lost-Update angesehen und ein Breakpoint gesetzt. Nachdem das zu untersuchende Programm angehalten worden ist, kann der Zustand des Programms vom Entwickler näher untersucht werden. Über das Interface des Debuggers kann das Programm weiter ausgeführt werden. Die Funktion des Breakpointsetzens kann auch deaktiviert werden. In diesem Fall wird der Programmfluss nicht gestoppt, sondern nur ein Vermerk in der Logdatei gespeichert.

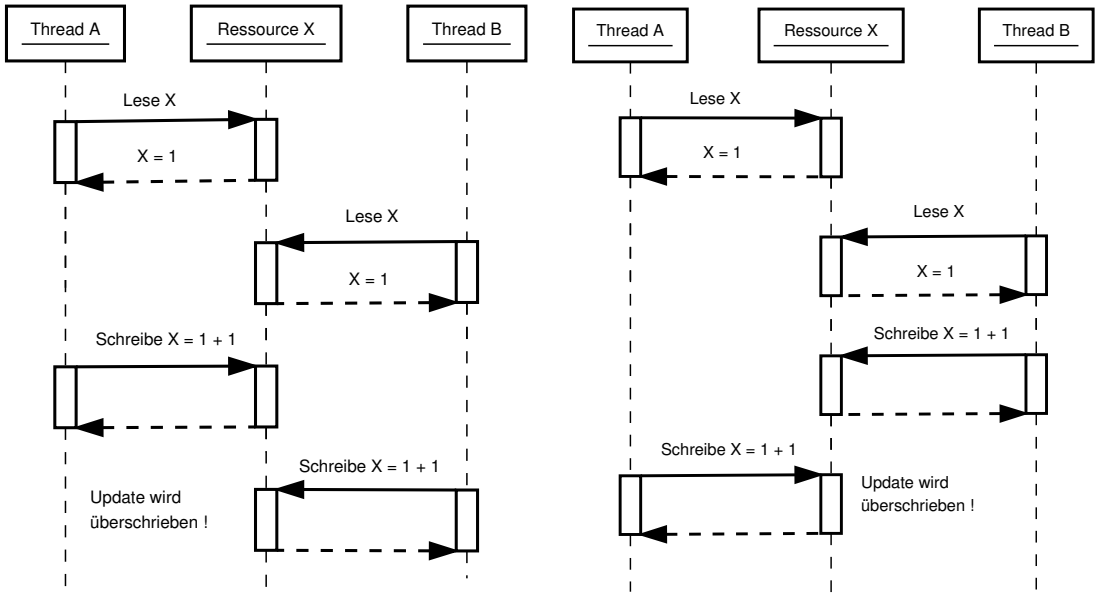

Abbildung 3.9: Schreib- und Lese Muster, die als Lost-Update gelten

3.3.2 Erkennen von Schreib- und Lesemustern

Das Erkennen der Schreib- und Lesemuster lässt sich als Zustandsmaschine beschreiben. Bei jedem Zugriff auf eine geteilte Speicherressource wird die Zustandsmaschine aktualisiert. Die Zustandsmaschine speichert die aktuell erlaubten Zugriffe auf einen Speicherbereich. Wenn ein Thread einen nicht erlaubten Zugriff tätigt, wird dies als potentielles Lost-Update angesehen. Die Speicherstruktur ist eine Liste von Tupeln (t, x) mit t als Threadid und x als Speicheradresse.

Folgende Zustände kann ein Tupel besitzen:

- Der Zustand $(t, x) = r$ gibt an, dass der letzte Zugriff von Thread t auf die Speicheradresse x lesend war.
- Der Zustand $(t, x) = w$ gibt an, dass der letzte Zugriff von Thread t auf die Speicheradresse x schreibend war.

- Der Zustand $(t, x) = lost$ gibt an, dass der nächste schreibende Zugriff von Thread t auf die Speicheradresse x ein vorheriges Update überschreiben wird. Es wird also nicht gespeichert, was in der Vergangenheit geschehen ist, sondern was in Zukunft passieren kann.

In Abbildung 3.10 ist der Algorithmus zur Erkennung eines Lost-Updates dargestellt. Der Algorithmus wird bei jedem Schreib- und Lesezugriff eines Threads t auf eine Speicheradresse ausgeführt. Der Algorithmus aktualisiert dabei die Tupel (t, x) wie oben beschrieben.

Zur Erklärung der Zustandsmaschine wird im Folgenden das linke der zwei Zugriffsmuster in Abbildung 3.9 exemplarisch ausgeführt und das Lost-Update erkannt.

1. Der erste Zugriff von Thread $Thread_A$ auf die Speicheradresse $Ressource_X$ ist lesender Natur. Also wird das Tupel $(Thread_A, Ressource_X)$ auf r gesetzt.
2. Der erste Zugriff von Thread $Thread_B$ liest die Speicheradresse $Ressource_X$. Das Tupel $(Thread_B, Ressource_X)$ wird ebenfalls auf r gesetzt.
3. Nun erfolgt ein schreibender Zugriff auf die Speicheradresse $Ressource_X$ von $Thread_A$ und das Tupel $(Thread_A, Ressource_X)$ wird auf w gesetzt. Zusätzlich werden alle vorherigen lesenden Zugriffe auf die Speicheradresse $Ressource_X$ auf den Wert $lost$ gesetzt. Dies geschieht in unserem Fall mit dem Tupel $(Thread_B, Ressource_X)$.
4. Im letzten Schritt greift $Thread_B$ schreibend auf die Speicheradresse $Ressource_X$ zu. Der Wert des Tupels wurde im vorherigen Schritt auf $(Thread_B, Ressource_X) = lost$ gesetzt und weist im jetzigen Schritt auf ein Lost-Update hin. Eine Fehlermeldung wird ausgegeben. Um weitere Lost-Updates zu detektieren, wird nach der Ausschrift der Wert des Tupels auf w gesetzt.

3.3.3 Grenzen des entworfenen Detektors

Der Detektor erkennt nur Lost-Updates, die auch wirklich stattfinden. Es gibt jedoch auch Situationen, in denen das Zugriffsmuster einem Lost-Update entspricht, aber vom Entwickler gewollt bzw. bewusst in Kauf genommen wird.

Da das Verwenden von Synchronisationsmechanismen auch eine Geschwindigkeitseinbuße bedeutet, wird bei Zählern in hochperformanter Software oft darauf verzichtet. Ein Zähler führt aber genau die in Abbildung 3.9 dargestellten Zugriffe durch. In diesem Fall wird der Verlust eines oder zwei Werte in Kauf genommen, da der Zähler nur einen groben Richtwert darstellt.

Ob ein Data Race bzw. Lost-Update bewusst gewollt ist oder ein Bug im Programm darstellt, kann in letzter Konsequenz nur der Entwickler entscheiden. Um solche harmlosen Data Races ignorieren zu können, werden vom Entwickler zusätzliche Informationen benötigt, die meist in Form von Kommentaren im Programmcode hinterlassen werden. Der von mir entwickelte Detektor unterstützt diese zusätlichen Annotationen nicht. Da

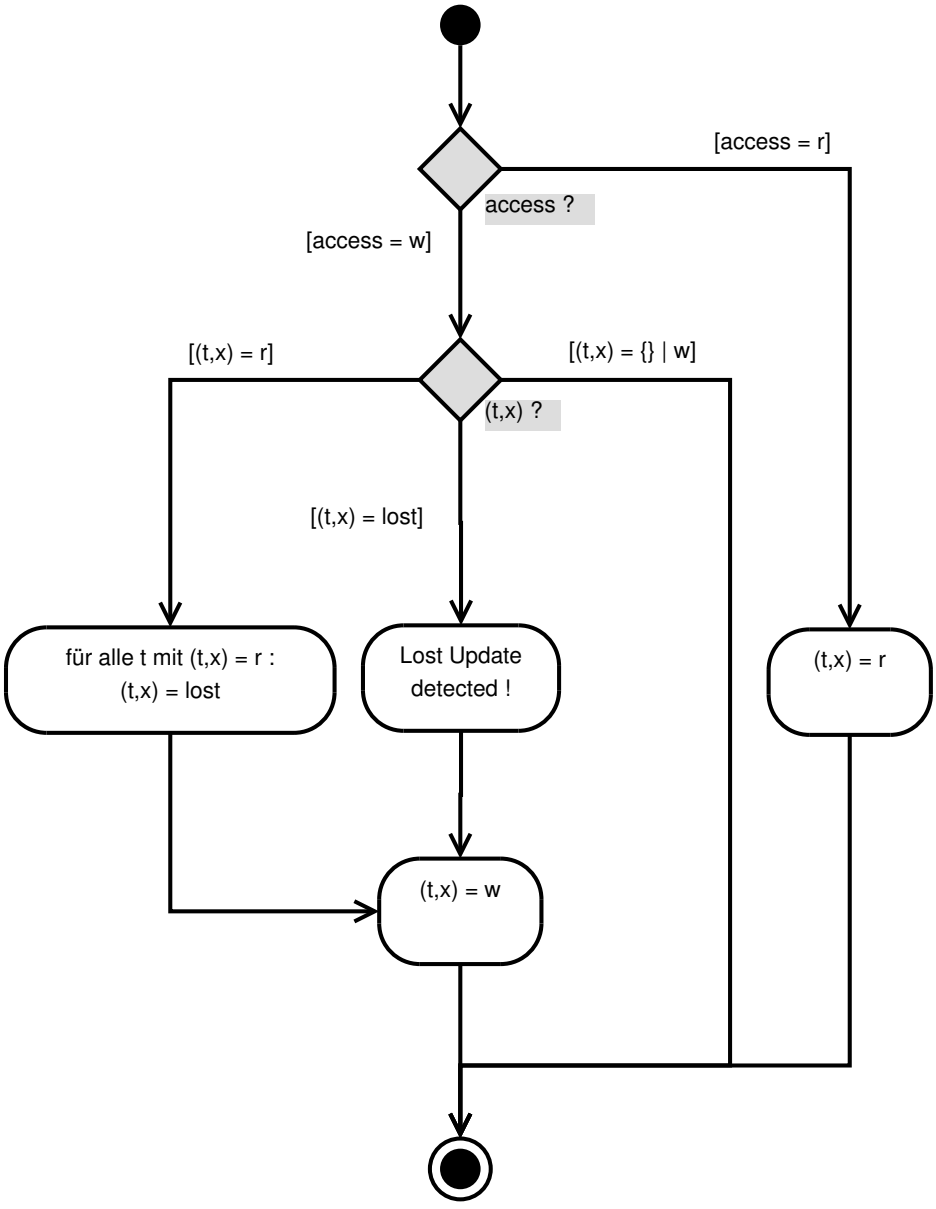

Abbildung 3.10: Die Aktualisierung der Zustände und das Erkennen eines Lost-Updates.

der Programmfluss im „debug-mode“ sowieso angehalten wird, kann der Entwickler entscheiden ob das gemeldete Data Race gefährlich ist oder ignoriert werden kann.

3.4 Laufzeitverhalten instrumentierter Anwendungen

Um die Ausführung eines Programmes näher untersuchen zu können, müssen interessante Ereignisse an eine beobachtende Instanz übermittelt werden. In unserem Fall geschieht dies durch Codeinstrumentierung. Für die Analyse ist eine solche Instrumentierung eine ”enableing technology” [30], welche die Grundlage vieler Analyseverfahren darstellt.

Zum einen kann das Programm in einer virtuellen Maschine ausgeführt werden, die unter der Kontrolle des Beobachters steht. Einen solchen Ansatz- verfolgt das Instrumentierungswerkzeug Valgrind. Hierbei wird während der Laufzeit jeder Befehl des zu instrumentierenden Programms in einen Befehl der Valgrind-Maschine, den UCode übersetzt. Dieser neue Befehl wird darauf hin von der virtuellen Maschine ausgeführt. Dieser Zwischenschritt versetzt den Beobachter in die Lage eigene Werkzeuge zwischenschalten, die auf die übersetzten Befehle reagieren. Der originale Programmcode des zu untersuchenden Programms wird nicht auf der Hardware ausgeführt sondern der übersetzte UCode. Da die Übersetzung des Programmes zu UCode für jede Answeisung stattfinden muss, ist eine Laufzeitverlängerung nicht zu umgehen. Bei eigenen Versuchen wurde eine Verlängerung der Laufzeit um mindestens den Faktor vier festgestellt.

Einen anderen Ansatz verfolgt das von Intel unterstützte Instrumentierungswerkzeug PIN. PIN arbeitet ähnlich einem Just-In-Time Compiler, wobei hier die Eingabe der Binärcode des zu instrumentierenden Programms ist. Die erste Instruktion des Programms wird kurz vor der Ausführung abgefangen und von PIN bei Bedarf um eigene Anweisungen ergänzt. Die so erweiterte Sequenz von Anweisungen wird nach der Instrumentierung ausgeführt. Diese Instrumentierung kann von eigenen PINTools gesteuert werden.

Der Unterschied zu Valgrind besteht darin, dass der von PIN instrumentierte Code auch wirklich auf der Hardware ausgeführt wird. Wird wenig instrumentiert, ist die Laufzeitverlängerung ebenfalls gering. Ein weiterer Vorteil für die Verwendung von von PIN gegenüber Valgrind als Instrumentierungsgrundlage besteht in der ”echten Ausführung” der Programminstruktionen. Es ist also möglich, dass Valgrind ein Data Race erkennt, dieses sich aber im Verhalten des Programms nicht wiederspiegelt. Dies ist insbesondere für Programme in der Testphase der Software-Entwicklung relevant, da hier andere Technologien wie z.b Unit Tests verwendet werden. Bei der Verwendung von einem auf Valgrind basierenden Detektor ist es möglich, dass der Detektor das Data Race detektiert, aber der dazu vorhandene Test nicht fehlschlägt. Die Meldung des Detektors könnte nun als False-Positive gesehen werden. Eine solche Situation kann mit einem Detektor auf Basis des PIN Frameworks nicht auftreten.

Um die Performance-Einbußen abzuschätzen, die bei der Realisierung des Lost-Update

Detektors auf Basis von PIN in Kauf genommen werden müssten wurde eine Laufzeituntersuchung durchgeführt. Die Eingriffe von PIN bzw. der Grad der Instrumentierung hängen maßgeblich von dem verwendeten PIN-Tool ab. In diesem Fall wird bei jedem Schreib- und Lesezugriff instrumentiert, was einen enorme Performance-Einbuße bedeutet.

```
1 #include <stdio.h>
2
3 int main()
4 {
5   printf( "Hello World\n" );
6   return 0;
7 }
```

Listing 3.7: Hello World in C

Das im folgenden Benchmark verwendete PINTool zählt Instruktionen, die in dem zu untersuchenden Programm ausgeführt werden. Dazu wird bei jeder auszuführenden Instruktion ein interner Counter erhöht, was einem ähnlichen Zugriffsverhalten des weiter oben vorgestellten Detektors ähnelt.

In Tabelle 3.1 ist der Zeitzuwachs durch die Instrumentierung dargestellt. Es werden die Anzahl der Instruktionen, die Ausführungszeit ohne Instrumentierung und die Ausführungszeit mit Instrumentierung gegenüber gestellt. Folgende vier Programme wurden verwendet:

- Das Programm *HelloWorld* in Listing 3.7 ist ein einfaches Programm mit der geringsten Anzahl an Instruktionen von den hier vorgestellten Programmen. Es wird eine Bibliothek eingebunden und aus dieser eine Funktion aufgerufen, die den Text „Hello World“ ausgibt. Die Ausführungszeit dieses einfachen Programms steigt durch die Instrumentierung um den Faktor 401.
- Das Programm *dataRace_simple* ist ein Programm, in dem vier Threads konkurrierend eine globale Variable lesen und den Wert erhöhen. Die Ausführungszeit steigt hier um den Faktor 788.
- Das Programm *dataRace_complex* ist ähnlich dem Programm *dataRace_simple*, wobei hier noch zusätzlich Mutexe für das Synchronisieren der Schreibzugriffe verwendet werden. Die Ausführungszeit steigt hier um den Faktor 761.
- Das Programm */usr/bin/ls* ist ein Linuxbefehl. Es liest den Inhalt des aktuellen Ordners und schreibt die Dateinamen aus. Um die Messungen nicht zu beeinflussen, wurde das Programm in einem leeren Ordner ausgeführt. Die Ausführungszeit steigt hier um den Faktor 632.

Um den Einfluss von Cachingeffekten in der Hardware auf die Ausführung zu minimieren wurde jedes Programm mehrmals ausgeführt und der Durchschnitt der ermittelten Werte genommen. Zusammenfassend bleibt festzustellen, dass die Ausführungszeit im Durchschnitt um den Faktor 645 zunimmt. Es ist offensichtlich, dass das Instrumentieren

Name	Instruktionen	ohne Instrum.	mit Instrum.	Faktor
HelloWorld	108476	0.006023 Sek.	2.420782 Sek.	401
dataRace_simple	391817	0.006650 Sek.	5.246407 Sek.	788
dataRace_complex	407281	0.006941 Sek.	5.286534 Sek.	761
/usr/bin/ls	9203775	0.010387 Sek.	6.566337 Sek.	632

Tabelle 3.1: Zeitmessungen ohne und mit Instrumentation auf Binärebene

von jedem Schreib- und Lesezugriff sehr invasiv ist, wodurch die Ausführungszeit stark zunimmt.

3.5 Zusammenfassung

Durch die Einführung von Multi-Core Systemen wurde auch eine neue Klasse an Defekten eingeführt. Data Races, Atomicity Violations und Deadlocks gehören zu dieser neuen Klasse an Concurrency Bugs. Wie diese Defekte erkannt und entfernt werden können, wurde anhand von verschiedenen Lösungen erläutert. Für ein besonderes Data Race, dem Lost-Update wurde ein eigener Ansatz vorgestellt, der anhand von Zugriffsmustern potentielle Lost-Updates erkennt und beim Debuggen von Software helfen kann.

Die Umsetzung des Ansatzes ist zumindest mit einer naiven Code-Instrumentierung zunächst nicht realistisch. Dies liegt nicht am vorgeschlagen Muster zur Identifikation von Lost-Updates oder der generellen Machbarkeit, sondern an der Auswirkungen der Code-Instrumentierung. Die Programmlaufzeit würde durch die Instrumentierung derart verzerrt, dass eine echtzeitfähige Ausführung unmöglich ist.

Kapitel 4

Zusammenfassung

Die Revolution der Multi-Core Prozessoren ist nicht mehr aufzuhalten. Die Verwendung von mehreren Recheneinheiten, die parallel an einem gemeinsamem Problem arbeiten ermöglicht das Fortführen der Performancesteigerung. Diese Steigerung wurde in der Vergangenheit durch die Erhöhung der Taktfrequenz erreicht, was physikalisch nun nicht mehr möglich ist.

Damit Software auch die Performace liefert, die die darunterliegende Hardware bietet, muss diese parallelisiert werden, was ein sehr fehleranfälliger Prozess ist. Durch die Parallelität wird eine neue Klasse an Defekten eingeführt, die in der seriellen Ausführung keine Bedeutung besaß. Die Klasse der Concurrency Bugs wurde in dieser Arbeit erläutert.

Fehlerintolerante Software zu entwickeln war und ist eine zentrale Herausforderung in der Software-Entwicklung. Ein Großteil der Zeit wird in das Debugging von Software investiert. Die Klasse der Concurrency Bugs stellt für das klassische Debugging eine noch größere Herausforderung dar, da die klassischen Ansätze nicht mehr greifen.

Die hier vorgestellten Ansätze bieten eine Lösung für die vorgestellten Concurrency Bugs. Ein Data Race kann durch korrekt verwendete Synchronisation vermieden werden. Durch gezieltes Ausführen von potentiellen Interleavings können Atomicity Violations häufiger ausgeführt und damit effizienter detektiert werden. Ob Synchronisationsmechanismen korrekt verwendet wurden kann ebenfalls überwacht werden. Auch der in diesem Rahmen entstandene Detektor bietet für Lost-Updates einen zufriedenstellenden Lösungsansatz.

Das Zusammenspiel des Detektors mit einem Debugger ist im Vergleich zu anderen Werkzeugen neu. Die Möglichkeit das Programm anzuhalten und genauer zu untersuchen, wenn ein ungünstiges Interleaving zu einem Lost-Update geführt hat, ist mit einem klassischen Debugger nicht möglich. Dieser Ansatz bietet einem Entwickler eine gewohnte Debuggingumgebung und genügend Informationen das Lost-Update genauer untersuchen zu können.

Ein gravierender Nachteil des Detektors ist die Verwendung einer rein dynamischen Analyse. Es wird nur ein Lost-Update detektiert, wenn dieses auch wirklich stattgefunden hat. Hier könnte eine mögliche Erweiterung ein Zusammenspiel mit einer statischen Codeanalyse ein besserer Ansatz sein. Unter der Berücksichtigung von den aus der statischen Analyse gewonnenen Informationen wäre auch ein gezielteres Instrumentieren möglich, das die Invasivität der Lösung verringern könnte.

Concurrency Bugs sind in existierender Software weit verbreitet. Wie das einführende Beispiel gezeigt hat, sind diese für den Einsatz von Software in sicherheitskritischen Bereichen eine enorme Herausforderung. Ob die hier vorgestellten Defekte die Klasse der Concurrency Bugs ausreichend beschrieben haben, wird die Forschung der nächsten Jahre zeigen. Ein Beherrschen dieser Fehlerklasse wird unseren Erachtens die zentrale Rolle in der Software-Entwicklung spielen.

Anhang A

Beispielprogramme

Alle Programme wurden mit dem GCC Compiler Version 4.4.4 kompiliert. Folgende Compilerflags wurden durchgehend verwendet:

- $-Wall$ - Das Programm wird mit höchster Warnstufe kompiliert. Jede Meldung wird als Warnung bewertet, wodurch das Programm erst erfolgreich kompiliert, wenn keine Warnungen mehr existieren.
- $-O0$ - Dieses Flag (Großbuchstabe o) setzt die Optimierungsstufe des Compilers auf Null, was die Optimierung komplett ausstellt. Dies führt dazu, dass das Binärprogramm mehr dem Quelltext des Orginalprogramms entspricht.
- $-g$ - Das Programm wird mit Debugging Informationen kompiliert. Das erzeugte Binärprogramm ist etwas umfangreicher, da nun zusätzliche Symbolinformationen enthalten sind.
- $-lpthread$ - Die Posix Thread Bibliothek wird verwendet.

A.1 dataRace_free

In diesem Programm führen vier Threads nebenläufig die Funktion $increase_counter$ aus. Es wird ein Mutex für die korrekte Synchronisation verwendet. Das Programm beinhaltet kein Concurrency Bug.

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4
5 #define NUMBER_OF_THREADS 4
6 int global_counter = 0;
7 pthread_mutex_t mutex1;
8
```

```
 9  // data race free
10  void *increaseCounter( )
11  {
12    pthread_mutex_lock( &mutex1 );
13
14    int tmp = global_counter + 1;
15    global_counter = tmp;
16
17    pthread_mutex_unlock( &mutex1 );
18
19    // exit
20    pthread_exit( NULL );
21    }
22
23  int main( int argc, char *argv[] )
24  {
25    printf( "counter value = %d\n", global_counter );
26
27    // initialize mutex
28    pthread_mutex_init( &mutex1, NULL );
29
30    //create and run a couple of threads
31    pthread_t threads[ NUMBER_OF_THREADS ];
32    long t;
33    for( t = 0; t < NUMBER_OF_THREADS; t++ ) {
34      if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
35        printf( "ERROR: pthread_create(%ld)\n", t );
36        return -1;
37      }
38    }
39
40    //wait for all threads to finish
41    int i;
42    for ( i = 0; i < NUMBER_OF_THREADS; i++ ) {
43      if( pthread_join( threads[i], NULL ) ) {
44        printf( "ERROR: thread_join(%d)\n", i );
45        return -1;
46      }
47    }
48
49    // exit
50    printf( "counter value = %d\n", global_counter );
51    pthread_exit( NULL );
52  }
```

Listing A.1: dataRace_free.c

A.2 dataRace_simple

Diese Programm entspricht dem Programm in Listing A.1 mit dem Unterschied, dass hier die Synchronisation entfernt wurde. Dieses Programm enhält ein Data Race bzw. Lost-Update.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  #define NUMBER_OF_THREADS 4
5  int global_counter = 0;
6
7  // contains data race, no lock
8  void *increaseCounter( )
9  {
10   int tmp = global_counter + 1;
11
12   //extra load -> might reveal data race
13   int i; for( i=0; i<2048; i++ ) ;
14
15   global_counter = tmp;
16
17   // exit
18   pthread_exit( NULL );
19 }
20
21 int main( int argc, char *argv[] )
22 {
23   printf( "counter value = %d\n", global_counter );
24
25   //create and run a couple of threads
26   pthread_t threads[ NUMBER_OF_THREADS ];
27   long t;
28   for( t = 0; t < NUMBER_OF_THREADS; t++ ) {
29     if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
30       printf( "ERROR: pthread_create(%ld)\n", t );
31       return -1;
32     }
33   }
34
35   //wait for all threads to finish
36   int i;
37   for ( i = 0; i < NUMBER_OF_THREADS; i++ ) {
38     if( pthread_join( threads[i], NULL ) ) {
39       printf( "ERROR: thread_join(%d)\n", i );
40       return -1;
41     }
42   }
43
44   // exit
45   printf( "counter value = %d\n", global_counter );
46   pthread_exit( NULL );
```

```
47 }
```

Listing A.2: dataRace_simple.c

A.3 dataRace_complex

Dieses Programm ist die erweiterte Form des Programms in Listing A.2. Die Funktion erhöht zweimal den globalen Counter $global_counter$. Für die Synchronisation werden zwei unterschiedliche Mutexe verwendet. Dieses Programm ist fehlerhaft und enthält ein Data Race bzw. Lost-Update, da für eine Variable zwei verschiedene Locks verwendet werden.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  #define NUMBER_OF_THREADS 4
5  int global_counter = 0;
6  pthread_mutex_t mutex1, mutex2;
7
8  // contains data race, despite locks!
9  void *increaseCounter( )
10 {
11   pthread_mutex_lock( &mutex1 );
12   int tmp = global_counter + 1;
13   //extra load -> might reveal data race
14   int i; for( i=0; i<1024; i++ ) ;
15   global_counter = tmp;
16   pthread_mutex_unlock( &mutex1 );
17
18
19   pthread_mutex_lock( &mutex2 );
20   tmp = global_counter + 1;
21   //extra load -> might reveal data race
22   int j; for( j=0; j<1024; j++ ) ;
23   global_counter = tmp;
24   pthread_mutex_unlock( &mutex2 );
25
26   // exit
27   pthread_exit( NULL );
28 }
29
30
31 int main( int argc, char *argv[] )
32 {
33   printf( "counter value = %d\n", global_counter );
34
35   // initialize mutexes
36   pthread_mutex_init( &mutex1, NULL );
37   pthread_mutex_init( &mutex2, NULL );
38
39   //create and run a couple of threads
```

```
40    pthread_t threads[ NUMBER_OF_THREADS ];
41    long t;
42    for( t = 0; t < NUMBER_OF_THREADS; t++ ) {
43      if ( pthread_create( &threads[t], NULL, increaseCounter, &t ) ) {
44        printf( "ERROR: pthread_create(%ld)\n", t );
45        return -1;
46      }
47    }
48
49    //wait for all threads to finish
50    int i;
51    for ( i = 0; i < NUMBER_OF_THREADS; i++ ) {
52      if( pthread_join( threads[i], NULL ) ) {
53        printf( "ERROR: thread_join(%d)\n", i );
54        return -1;
55      }
56    }
57
58    // exit
59    printf( "counter value = %d\n", global_counter );
60    pthread_exit( NULL );
61  }
```

Listing A.3: dataRace_complex.c

Literaturverzeichnis

[1] U.-C. P. S. O. T. Force, “Final report on the august 14, 2003 blackout in the united states and canada: Causes and recommendations,” tech. rep., U.S.-Canada Power System Outage Task Force, April 2004.

[2] P. A. Kidwell, “Stalking the elusive computer bug,” *IEEE Annals of the History of Computing, Volume 20*, vol. 1058-6180, 1998.

[3] unknown, “Proceesings ieee metrics,” *3rd IEEE International Software Metrics Symposium (METRICS)*, 1996.

[4] *NATO Software Engineering Conference*, 1968.

[5] J. N. Buxton and B. Randell, eds., *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[6] E. W. Dijkstra, “Notes on Structured Programming.” circulated privately, apr 1970.

[7] A. A. et al., “Fundamental concepts of dependability,” 2001.

[8] J. Gray, “Why do computers stop and what can be done about it?,” in *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, 1986.

[9] Merriam-Webster, *Merriam-Webster’s Collegiate Dictionary, 11th Edition thumb-notched with Win/Mac CD-ROM and Online Subscription*. Merriam-Webster, 2003.

[10] B. Vermeulen and K. Goossens, *Debugging Multi-Core Systems on Chip*. CRC Press/Taylor & Francis Group, 2010.

[11] W. Heisenberg, “Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik,” *Zeitschrift für Physik*, vol. 43, pp. 172–198, March 1927.

[12] A. Zeller, “Automated debugging: Are we close,” *Computer*, vol. 34, pp. 26–31, 2001.

[13] J. deok Choi and H. Srinivasan, “Deterministic replay of java multithreaded applications,” in *In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 48–59, 1998.

[14] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*. San Francisco, CA, USA: No Starch Press, 2008.

[15] M. Schwabl-Schmidt, *Systemprogrammierung für AVR-Mikrocontroller*. Elektor, 2009.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Wiley, December 2004.

[17] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," *SIGARCH Comput. Archit. News*, vol. 11, pp. 124–131, June 1983.

[18] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, pp. 690–691, September 1979.

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, (New York, NY, USA), pp. 483–485, ACM, 1967.

[20] E. W. Dijkstra, "Hierarchical ordering of sequential processes." published as [**?**]WD:EWD310pub, n.d.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 329–339, ACM, 2008.

[22] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*, (New York, NY, USA), pp. 62–71, ACM, 2009.

[23] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, pp. 549–557, 1974.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[25] H. Balzert, *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg; Berlin: Spektrum Akademischer Verlag, 1 ed., 2000.

[26] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 37–48, ACM, 2006.

[27] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 25–36, ACM, 2009.

[28] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[29] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007.

[30] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, “Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles,” *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 267–279, 2007.