Spekulation auf Fadenebene

Edgar Kalkowski (48475)

Sommersemester 2009

Zusammenfassung

In der heutigen Zeit werden Multicores zunehmen für jedermann verfügbar. Damit einhergehend stellt sich die Frage, wie man das dadurch vorhandene Potenzial an Parallelität auch für alltägliche Anwendungen ausnutzen kann. Eine händische Anpassung der großen Menge an vorhandener Software scheint aussichtslos, weshalb man Verfahren benötigt, Code automatisch zu parallelisieren. Traditionelle Ansätze – wie das Polytopmodell – analysieren die Datenabhängigkeiten eines Programms zur Übersetzungszeit und transformieren das Programm in eine andere Darstellung, die besser parallel ausführbar ist. Um die zahlreichen Einschränkungen dieser statischen Verfahren bezüglich des Quellprogramms zu überwinden, wurde zu einer dynamischen Analyse von Datenabhängigkeiten zur Laufzeit übergegangen. Dabei wird das Programm *spekulativ* parallel ausgeführt und bei Verletzung einer Datenabhängigkeit ein Rollback durchgeführt. Diese Arbeit stellt eine Software- und eine Hardware-Umsetzung dieser spekulativen Parallelisierungstechnik vor.

Inhaltsverzeichnis

1 Einleitung

Das sogenannte „Moorsche Gesetz“ besagt, dass sich die Leistung der Prozessoren unserer Computer alle zwei Jahre verdoppelt. Aufgestellt wurde diese Behauptung in den siebziger Jahren von Gordon Moore und bis heute hat sie sich in etwa bewahrheitet. Jedoch ist mittlerweile klar, dass man mit der Verkleinerung und immer schnelleren Taktung einzelner Prozessoren an einer Grenze angelangt ist. Aufgrund von Quanteneffekten ist eine weitere Verkleinerung nicht möglich und aufgrund dessen, dass übliche Computer mit Strom arbeiten, der sich durch den Chip ausbreiten muss und dafür Zeit braucht, kann auch die Taktrate nicht beliebig weiter erhöht werden.

In der Konsequenz geht man dazu über, nicht mehr die Leistung eines einzelnen Prozessors zu verbessern, sondern Computer mit mehreren Rechenkernen auszustatten. Handelsüblich sind heutzutage bereits Multicores mit zwei oder vier Kernen und es ist zu erwarten, dass bereits in naher Zukunft mehrere Zig oder Hundert Kerne möglich sein werden.

Um die so geschaffene Rechenpower auch ausnutzen zu können, ist es erforderlich, dass die auf einem Multicore ausgeführte Software die vorhandene Parallelität auch ausnutzen kann. Das Gebiet der Parallelprogrammierung gewinnt also zunehmend an Bedeutung. Traditionell beschäftigte man sich dort mit der automatischen Parallelisierung von mathematischen Programmen, die an Universitäten oder Forschungszentren auf speziellen Parallelrechnern liefen. Die heutige Herausforderung besteht darin, auch Möglichkeiten aufzuzeigen, wie bereits bestehende sequentielle und nicht unbedingt mathematische Programme möglichst automatisch parallelisiert werden können.

Da solche Programme oft einen komplexen Kontrollfluss und komplexe Speicherzugriffsmuster haben, deren Analyse zur Übersetzungseit nicht möglich ist, können klassische Parallelisierungstechniken – wie das Polytopmodell – nicht eingesetzt werden. Stattdessen wird *spekuliert*, dass keine oder nur wenige Datenabhängigkeiten im Programm vorliegen und zur Laufzeit analysiert, ob Abhängigkeiten verletzt wurden. Falls das der Fall ist, wird ein Rollback-Mechanismus angestoßen, der das Programm wieder in einen konsistenten Zustand überführt.

Zu Beginn dieser Arbeit wird kurz das klassische Polytopmodell vorgestellt und verschiedene Erweiterungen diskutiert, die einige Einschränkungen des klassischen Modells bezüglich des sequentiellen Quellprogramms aufheben. Danach wird diskutiert, wie man Programme mit spekulativen Techniken parallelisieren kann. In diesem Zusammenhang wird eine Software- und eine Hardware-Implementierung vorgestellt, die jeweils Spekulation auf Fadenebene realisieren. Schließlich werden die gefundenen Ergebnisse kurz zusammengefasst und es wird ein Ausblick auf zukünftige Entwicklungen gegeben.

2 Forschungsstand

In diesem Kapitel wird zunächst das Polytopmodell in seiner ursprünglichen Form vorgestellt. Danach werden zwei Erweiterungen diskutiert, die es ermöglichen, auch `while`-Schleifen zu parallelisieren.

2.1 Klassisches Polytopmodell

Bereits seit den siebziger Jahren wurden systematische, modellbasierte Verfahren entwickelt, um Programme und insbesondere Schleifen zu parallelisieren. 1993 stellte Lengauer ein modellbasiertes Framework zur Parallelisierung von `for`-Schleifen vor, das auf dem mathematischen *Polytopmodell* basierte (vgl. [Len93]).

Eine verschachtelte `for`-Schleife wird dabei zunächst auf ein mehrdimensionales Polytop abgebildet. Eine Koordinaten-Dimension im Raum des Polytops entspricht dabei genau der Indexmenge einer der geschachtelten `for`-Schleifen. Die Begrenzungen des Polytops werden durch die Schleifengrenzen festgelegt. Jeder Punkt mit ganzzahligen Koordinaten innerhalb des Polytops entspricht dann genau einer Schleifeniteration mit den zugehörigen Indizes.

In Abbildung 1 ist oben links ein Beispiel für eine zweifach verschachtelte `for`-Schleife gegeben. Unten links sieht man die Transformation dieser Schleife in ein zweidimensionales Polytop für $n = 3$.

Nachdem eine Polytopdarstellung der Schleife gefunden wurde, werden die Abhängigkeiten zwischen den verschiedenen Schleifeniterationen analysiert und als gerichtete Kanten in das Polytop eingetragen. Im Beispiel von Abbildung 1 wird in der Iteration (i, j) das Array `A` an den Stellen $(i-1, j)$ und $(i, j-1)$ gelesen. Die Schleifendurchläufe mit diesen Koordinaten müssen also bereits beendet sein, bevor der Durchlauf (i, j) beginnen kann. Demzufolge sind im Polytopmodell (unten links in Abbildung 1) jeweils die gerichteten Kanten $(i-1, j) \to (i, j)$ und $(i, j-1) \to (i, j)$ eingetragen.

Wurden alle Datenabhängigkeiten analysiert, wird zunächst ein *Schedule* berechnet, also eine Reihenfolge in der Zeit, in der die einzelnen Iterationen ausgeführt werden können, ohne dass die gefundenen Abhängigkeiten verletzt werden. Diese Funktion ordnet jedem ganzzahligen Punkt des Polytops einen Zeitindex zu, an dem die zugehörige Iteration frühestens ausgeführt werden darf.

Sei im Beispiel von Abbildung 1 $I = \{0, \ldots, n\}$ die Indexmenge der äußeren Schleife und für jedes $i \in I$ sei $J = \{0, \ldots, i+2\}$ die Indexmenge der jeweiligen inneren Schleife. Dann ist eine geeignete Schedule-Funktion z. B. $t : I \times J \to \mathbb{N}$ mit $t(i, j) = i + j$.

Findet man eine Schedule-Funktion, die nicht injektiv ist, so erhält man Zeitpunkte, an denen man mehrere Iterationen der geschachtelten Schleife gleichzeitig ausführen kann, ohne dass die gefundenen Datenabhängigkeiten verletzt werden. Mittels

```
for (int i = 0; i <= n; i++) {
  for (int j = 0; j <= i+2; j++) {
    A[i][j] = A[i-1][j] + A[i][j-1];
  }
}
```

```
for (int t = 0; t <= 2*n + 2; t++) {
  parallel for (int p = max(0, t-n); t <=
      min(t, t/2 + 1); t++) {
    A[t-p][p] = A[t-p-1][p] + A[t-p][p-1];
  }
}
```

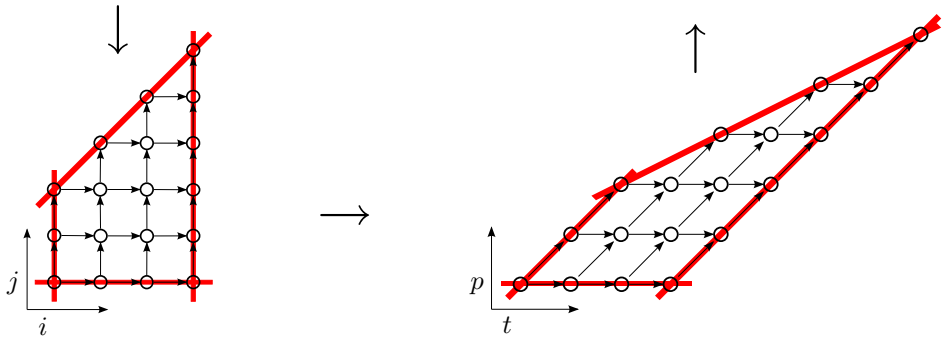

Abbildung 1: Transformation einer geschachtelten `for`-Schleife mit dem Polytopmodell

einer weiteren Funktion, der sog. *Allokation*, wird jede dieser Iterationen dann einem Prozessor zugewiesen, der sie ausführt. Eine geeignete Allokations-Funktion für das Beispiel aus Abbildung 1 ist z. B. $p : I \times J \to \mathbb{N}$ mit $p(i, j) = j$.

Transformiert man den Raum des Polytops mithilfe der Schedule- und der Allokations-Funktion, so erhält man ein „verzerrtes“ Polytop, das im unteren rechten Teil von Abbildung 1 dargestellt ist. Dieses transformierte Polytop wird nun wieder in eine geschachtelte `for`-Schleife übersetzt. Dazu müssen mithilfe der Schedule- und Allokations-Funktion die Schleifengrenzen und die Arrayindizes geeignet angepasst werden. Das Ergebnis dieser Transformation ist im oberen rechten Teil der Abbildung 1 zu sehen. Man erkennt, dass es in einigen Dimensionen des Polytops keine Datenabhängigkeiten mehr in Richtung der entsprechenden Achsen gibt, weshalb die entsprechenden `for`-Schleifen (im Beispiel die innere Schleife mit Index p) nun parallel ausgeführt werden können.

Wie obiges Beispiel zeigt, ist die Parallelisierung von `for`-Schleifen mithilfe des Polytopmodells relativ einfach durchführbar. Jedoch hat das ursprüngliche Modell zahlreiche Einschränkungen in Bezug auf das Quellprogramm:

1. Die verschachtelte Schleife muss *perfekt geschachtelt* sein, d. h. der Rumpf jeder Schleife darf entweder wieder genau aus einer perfekt geschachtelten Schleife bestehen oder aus einer Folge von Zuweisungen ohne Schleife (im Fall der innersten Schleife).
2. Die Schleifengrenzen müssen affine Ausdrücke von Indizes der äußeren Schleifen oder von Konstanten sein.
3. Die einzige erlaubte Datenstruktur ist das Array (Skalare sind hier nulldimensionale Arrays). Array-Indizes müssen affine Ausdrücke von Schleifen-Indizes oder Konstanten sein.
4. Die Schleifenabhängigkeiten müssen in allen Durchläufen uniform sein.

5. Es können nur `for`-Schleifen parallelisiert werden.

Durch Erweiterungen des ursprünglichen Verfahrens konnten einige dieser Einschränkungen mittlerweile aufgeweicht werden. So stellte z. B. Feautrier 1992 eine Möglichkeit vor, wie auch nicht perfekt geschachtelte Schleifen verarbeitet werden können (vgl. z. B. [Fea91]). Die dritte obige Bedingung konnte ebenfalls verallgemeinert werden, sodass heutzutage auch affin lineare Schleifenabhängigkeiten zulässig sind (vgl. z. B. [Gri96]).

2.2 Erweiterung auf `while`-Schleifen

Eine vergleichsweise starke Einschränkung des ursprünglichen Polytopmodells ist, dass nur `for`-Schleifen parallelisiert werden können. Es wurden verschiedene Erweiterungen vorgestellt, die das Modell auf `while`-Schleifen erweitern. Das Problem dabei ist, dass bei einer `while`-Schleife zur Übersetzungszeit und möglicherweise auch zur Laufzeit nicht bekannt ist, wie viele Iterationen ausgeführt werden. Im Folgenden werden kurz zwei Ansätze vorgestellt, die diesem Problem auf unterschiedliche Weise begegnen.

2.2.1 Konservativer Ansatz

Das Polytopmodell basiert auf den Schleifenindizes der `for`-Schleifen. Da `while`-Schleifen keinen Index besitzen, ist die erste Maßname zur Erweiterung des Polytopmodells, alle `while`-Schleifen mit einem künstlichen Index zu versehen. Ist das geschehen, kann man eine geschachtelte Schleife, die nun auch `while`-Schleifen beinhalten kann, analog zum Polytopmodell als ein Polyeder darstellen. Ein Polyeder ist ist Polytop, das in den Dimensionen, die `while`-Schleifen entsprechen, nicht beschränkt ist, da hier nicht klar ist, wie viele Iterationen tatsächlich ausgeführt werden.

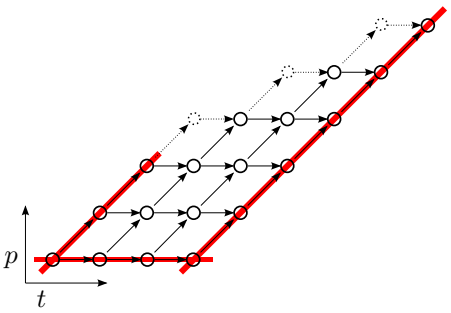

Abbildung 2: Transformiertes Polyedermodell einer geschachtelten Schleife, die eine `while`-Schleife enthält

Analog zum Polytopmodell kann das so gewonnene Polyeder nun transformiert werden, sodass in einigen Dimensionen keine Datenabhängigkeiten in Richtung der Achsen mehr existieren und die Schleifen, die diese Dimensionen repräsentieren, somit parallel ausgeführt werden können. In Abbildung 2 ist ein Beispiel für ein solches transformiertes Polyeder zu sehen.

Problematisch ist nun, dass nicht bekannt ist, wann die `while`-Schleife terminiert, die Iterationen der `while`-Schleife jedoch auf unterschiedlichen Prozessoren ausgeführt werden. Im konservativen Ansatz (vgl. z. B. [LG95]) wird dieses Dilemma gelöst, indem jeder Prozessor so lange wartet, bis die vorherige Iteration durchgeführt wurde und somit klar ist, ob der aktuelle Schleifendurchlauf noch ausgeführt werden muss oder die Schleife bereits terminiert ist. Im Gegensatz zu dem im nächsten Absatz vorgestellten spekulativen Ansatz bietet diese Vorgehensweise den Vorteil, dass

sie praktisch keinen Speicheroverhead verursacht. Allerdings wird die Ausführung des parallelen Programms durch das Warten auf vorherige Iterationen der `while`-Schleife verzögert.

2.2.2 Spekulativer Ansatz

Ein anderer Ansatz, der z. B. von Collard untersucht wurde (vgl. [Col94]), löst das Problem etwas anders. Zunächst wird die geschachtelte Schleife ebenfalls in ein Polyeder übersetzt und dieses wie oben transformiert, um eine Darstellung der Schleifen zu gewinnen, in der manche Schleifen parallel ausgeführt werden können.

Allerdings wird hier nicht jeweils gewartet, bis bei einer `while`-Schleife die vorherige Iteration komplett ist, um feststellen zu können, ob der nachfolgende Durchlauf überhaupt noch ausgeführt werden muss. Es werden im Gegenteil alle unter Beachtung der Datenabhängigkeiten möglichen Iterationen spekulativ ausgeführt. Sobald die Termination der Schleife festgestellt wurde, werden alle noch laufenden spekulativen Iterationen abgebrochen und deren Ergebnisse verworfen.

Um am Ende einer `while`-Schleife feststellen zu können, welches Ergebnis das der letzten Iteration (und damit das insgesamt gültige) ist, werden die Ergebnisse aller Durchläufe zwischengespeichert. Das führt im Vergleich zum konservativen Ansatz zu einem sehr hohen Speicheroverhead, auf der anderen Seite wird die Parallelität besser ausgenutzt, da keine Wartezeiten entstehen.

2.3 Bewertung

Die bisher vorgestellten Verfahren zur automatischen Parallelisierung sind bereits gut erforscht und liefern gute Ergebnisse. Trotz aller Erweiterungen bleiben aber einige Einschränkungen bestehen, sodass diese Verfahren zur Parallelisierung von „alltäglichen“ Programmen mit komplexem Kontrollfluss und nicht zur Laufzeit analysierbaren Datenabhängigkeiten nicht ausreichen.

Im nächsten Kapitel wird ähnlich zum spekulativen Ansatz zur Integration von `while`-Schleifen ins Polytopmodell ein Verfahren vorgestellt, wie auch diese Einschränkungen noch überwunden werden können.

3 Spekulative Parallelität

Im letzten Kapitel wurde kurz das Polytopmodell vorgestellt und erläutert, wie man damit sequentielle Programme, deren Datenabhängigkeiten sich zur Übersetzungszeit analysieren lassen, parallelisieren kann. Trotz aller Erweiterungen bleibt die Anwendung des Polytopmodells jedoch vor allem auf mathematisch bzw. numerische Anwendungen beschränkt, da bei komplexen Datenstrukturen die Datenabhängigkeiten zwischen verschiedenen Iterationen einer Schleife nicht zur Übersetzungszeit festgestellt werden können.

In diesem Kapitel wird ein Verfahren erläutert, das auf der spekulativen Ausführung von Programmteilen in mehreren Threads oder Fäden beruht und auch die Parallelisierung von Programmen mit komplexen Speicherzugriffen und komplexem Kontrollfluss ermöglicht. Dabei wird zunächst das Verfahren allgemein erläutert und danach eine Software- und eine Hardware-Implementierung vorgestellt.

3.1 Grundlagen

In diesem Abschnitt wird zunächst geklärt, welche Schritte allgemein notwenig sind, um ein sequentielles Programm spekulativ zu parallelisieren. Der erste Schritt besteht dabei darin, den Quelltext zu analysieren und Regionen zu identifizieren, die man parallelisieren möchte. Dazu eigenen sich z. B. Profiler, mit denen man feststellen kann, welche Programmteile die meiste Ausführungszeit benötigen.

Hat man solche potenziellen *spekulativen Regionen* gefunden, so werden diese in einzelne *Epochen* aufgeteilt. Jede Epoche wird dabei später von einem eigenen Thread ausgeführt. Epochen können einzelne Iterationen einer Schleife oder Rekursion sein, oft bietet es sich jedoch an, Schleifen um einige Durchläufe abzurollen. Der Grund dafür ist, dass das Auführen jeder Epoche mit einigem Overhead verbunden ist. So muss entweder für jede Epoche ein neuer Thread erzeugt werden, der sie ausführt, oder die Epochen müssen auf wiederverwendete Threads aufgeteilt werden. Es ist also wichtig, einen geeigneten Wert für die Größe der Epochen zu finden.

Der nächste Schritt hin zu einem spekulativen parallelen Programm ist die Überlegung, wie man zur Laufzeit Datenabhängigkeiten zwischen den Epochen erkennen kann. Die wichtigste Voraussetzung für das hier vorgestellte Verfahren ist dabei, dass die Schreibvorgänge der einzelnen Threads in den globalen Speicher seriell bleiben. Das bedeutet, dass alle Theads ihre Schreibvorgänge zunächst lokal puffern und lediglich der Thread mit dem ältesten Iterationsindex darf seine Ergebnisse in den globalen Speicher schreiben.

Durch diese Vorgehensweise erreicht man, dass WAW („write after write“) Abhängigkeiten gar nicht erst verletzt werden können, da alle Schreiboperationen im globalen Speicher in derselben Reihenfolge wie im sequentiellen Programm stattfinden. Aus demselben Grund werden auch Verletzungen von WAR („write after read“)

Datenabhängigkeiten vermieden.

Die einzigen Datenabhängigkeiten, die verletzt werden können, sind also vom Typ RAW („read after write“). Eine solche Verletzung tritt dann auf, wenn eine Epoche einen veralteten Wert gelesen hat, der von einer früheren Epoche überschrieben wurde, was aufgrund der gepufferten Schreibvorgänge erst zu einem späteren Zeitpunkt für die anderen Epochen sichtbar wird. Vorraussetzung dafür ist, dass für jede Speicherzelle gemerkt wird, welche Epoche den enthaltenen Wert zuletzt gelesen hat.

Wird die Verletzung einer Datenabhängigkeit erkannt, gibt es verschiedene Möglichkeiten, damit umzugehen. Eine Möglichkeit wäre, die Epoche, die einen veralteten Wert gelesen hat, auf den Status vor dem Laden des falschen Werts zurückzusetzen und von dort erneut rechnen zu lassen. Dies setzt jedoch voraus, dass z. B. alle Schreibvorgänge im lokalen Puffer der Epoche, die nach dem Laden passiert sind, zurückgesetzt werden müssen. Oft ist es jedoch einfacher, die gesamte Epoche neu zu starten. Man beachte, dass es zu keinem Deadlock kommen kann, da stets zumindest die Epoche mit dem aktuell ältesten Iterations-Index einen Fortschritt macht und ihre Ergebnisse auch in den globalen Speicher schreiben darf.

3.1.1 Beispiel

Analysiert man das Programm `compress` mit einem Profiler, so zeigt sich, dass es den Großteil seiner Rechenzeit (über 99 %, vgl. [SM98]) in einer einzigen `while`-Schleife verbringt, die Zeichen aus der Eingabedatei einliest und komprimiert. Dazu wird eine Hashmap verwendet, deren Einträge in der Schleife sowohl gelesen als auch modifiziert werden. Da die zu komprimierende Zeichenkette einen Einfluss auf die Hashmap hat, kann man zur Übersetzungszeit unmöglich Datenabhängigkeiten zwischen konsekutiven Iterationen bestimmen. Nimmt man an, dass eine gute Hashfunktion Abhängigkeiten zwischen aufeinanderfolgenden Schleifendurchläufen verhindern kann, so bietet es sich an, Iterationen der Schleife aus `compress` spekulativ in verschiedenen Threads auszuführen und zu hoffen, dass tatsächlich in den meisten Fällen keine Abhängigkeiten existieren.

In Abbildung 3 ist eine mögliche Situation dargestellt, wie sie in diesem Fall auftreten kann. Vier Prozessoren führen vier aufeinanderfolgende Schleifen-Iterationen aus. Zwischen den ersten drei Epochen besteht keinerlei Abhängigkeit, weshalb alle drei fehlerfrei durchlaufen und ihre Ergebnisse in den globalen Speicher schreiben. Sobald jedoch die erste Epoche ihre Werte schreibt, stellt sie fest, dass Epoche 4 den überschriebenen Wert `hash[10]` gelesen hat. Das bedeutet, dass hier eine RAW-Verletzung vorliegt und Epoche 4 neu gestartet werden muss.

3.1.2 Weiterleiten von Werten zwischen Epochen

Eine wichtige Technik im Zusammenhang des Abbrechens und Neustartens von Epochen ist die Weiterleitung von Werten zwischen einzelnen Epochen. Wenn sich zur Laufzeit herausstellt, dass häufig Datenabhängigkeiten zwischen aufeinanderfolgenden Epochen bestehen, die von unterschiedlichen Threads ausgeführt werden, dann macht es ggf. Sinn, einen Thread zu suspendieren, bis der andere den benötigten

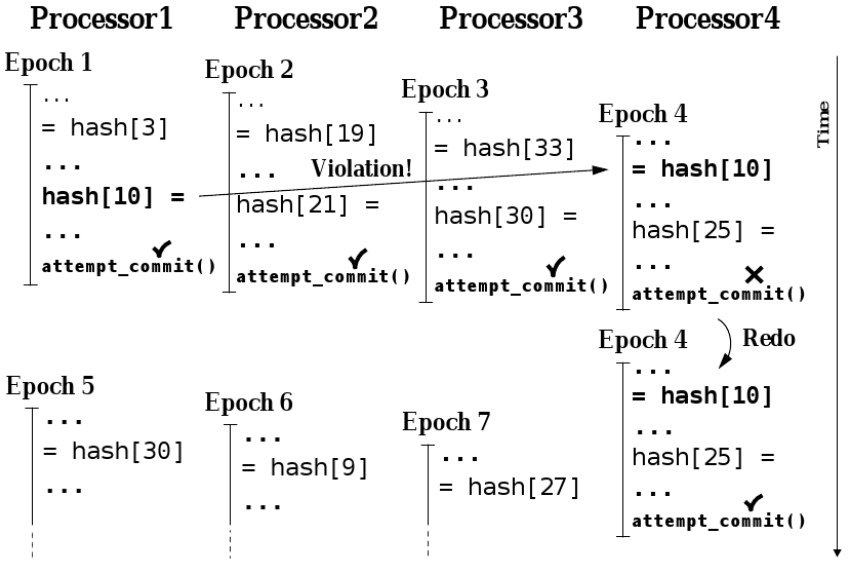

Abbildung 3: Spekulative Ausführung von `compress`

Wert berechnet hat. So spart man sich den Overhead für das Abbrechen und Neustarten des Threads.

Dieser Mechanismus wird nicht von allen Implementierungen unterstützt und soll hier nicht näher besprochen werden. Details sind z. B. in [SM98] zu finden.

3.2 Software-Umsetzung

Ein Beispiel für eine Software-Implementierung der vorgestellten Verfahren zum spekulativen Parallelisieren von sequentiellen Programmen ist die in C++ geschriebene Bibliothek *PolyLibTLS*, die z. B. in [OM08] vorgestellt wird. Die Idee hinter der Bibliothek ist, dass alle Lade- und Schreiboperationen innerhalb einer spekulativen Region durch die Funktionsaufrufe `specLD()` und `specST()` ersetzt werden. Der Code in diesen Funktionen regelt den Speicherzugriff und erkennt Datenabhängigkeites-Verletzungen.

Dazu hat jeder Thread lokal eine Map, in die beim Aufruf von `specST()` der gespeicherte Wert mitsamt Adresse eingefügt wird. Beim Aufruf von `specLD()` wird in einer global sichtbaren Datenstruktur nachgeschaut, ob die letzte Epoche, welche die entsprechende Speicherzelle gelesen hat einen älteren Index hatte als die aktuell lesende. Falls ja wird der Wert auf den Index der aktuell lesenden Epoche aktualisiert. Zusätzlich schaut `specLD()` in der lokalen Map nach, ob die angeforderte Speicherzelle nicht bereits lokal überschrieben wurde. In diesem Fall wird der lokal gepufferte Wert zurückgegeben.

Um den Speicheroverhead für das Merken der Epochen-Indizes, die eine Speicherzelle zuletzt gelesen haben, zu minimieren, verwendet *PolyLibTLS* eine Hashmap, in der ein Eintrag jeweils für mehrere Speicherzellen zuständig ist. Dies kann zu fälschlicherweise erkannten Datenabhängigkeiten und somit zu überflüssigen Rollbacks führen. Man hofft jedoch, dass aufgrund der Tatsache, dass in einem Programm oft aufeinanderfolgende Speicherzellen gelesen werden, die durch die Hashfunktion auf

verschiedene Einträge abgebildet werden, der Overhead durch zusätzliche Rollbacks geringer ist als der für das Speichern der Epochen-Indizes für alle Speicherzellen.

Wie in Abschnitt 3.1 beschrieben, darf jeweils der Thread mit dem ältesten Epochen-Index seine Ergebnisse in den globalen Speicher schreiben. Dazu geht er hier alle Einträge in seiner lokalen Map durch und fragt im globalen Verzeichnis der Lesezugriffe an, ob eine der von ihm geschriebenen Speicherzellen von einer Epoche mit späterem Index gelesen wurde. Ist das der Fall, schreibt der Thread sein Ergebnis zwar auch in den globalen Speicher, veranlasst jedoch zusätzlich den Abbruch der Epoche, die den falschen, veralteten Wert gelesen hat, und aller darauffolgenden Epochen.

3.2.1 Speichermodelle

Die Bibliothek unterstützt verschiedene Speichermodelle für verschiedene Abschnitte des Speichers. Ein Modell implementiert genau das bisher dargestellte Verhalten, dass Schreibzugriffe zunächst gepuffert werden und nur die aktuell älteste Epoche in den globalen Speicher schreiben darf. Lesezugriffe auf den globalen Speicher sind stets erlaubt, jedoch wird protokolliert welche Epoche zuletzt auf welche Speicherzelle zugegriffen hat.

Ein anderes Modell gewährt nur lesenden Zugriff auf den Speicher und führt bei einer versuchten Schreiboperation direkt zum Abbrechen der entsprechenden Epoche. Durch die Verwendung dieses Speichermodells für Speicherbereiche, von denen wahrscheinlich ist, dass aus ihnen nur gelesen wird, kann Overhead vermieden werden.

Ein drittes Speichermodell erlaubt es sogar, parallel in den globalen Speicher zu schreiben und die serielle Commit-Phase aufzugeben. Dieses Modell ist im Detail in [OM07] beschrieben.

Das Design der Bibliothek *PolyLibTLS* zielt auf eine einfache Erweiterbarkeit gerade in Hinsicht auf neue Speichermodelle ab. So ist es vergleichsweise einfach möglich, ein eigenes Modell zu implementieren, das genau auf die eigenen Bedürfnisse zugeschnitten ist.

3.2.2 Beispiel

In Listing 1 ist eine `for`-Schleife angegeben, die in Listing 2 auszugsweise in *PolyLibTLS*-fähigen Code übersetzt wurde. Die Original-Schleife benutzt zwei Referenzen `A` und `B`, wobei auf `A` nur lesend und auf `B` nur schreibend zugegriffen wird. Da zur Übersetzungszeit nicht klar ist, ob möglicherweise `B` z. B. auf den Anfang von `A` zeigt, kann diese Schleife nur spekulativ parallelisiert werden.

Da vermutlich außerdem `A` und `B` auf disjunkte Speicherbereiche zeigen und somit auf den Speicher von `A` nur lesend zugegriffen wird, kann hierfür das Speichermodell „read-only“ verwendet werden (vgl. Zeile 1 in Listing 2). Da in den Speicher von `B` auf jeden Fall geschrieben wird, muss hier das übliche Speichermodell mit serieller Commit-Phase verwendet werden (vgl. Zeile 2). In Zeile 3 werden die beiden Speichermodelle in einem vereinigten Modell zusammengefasst. Dieses Modell kümmert sich daraum, dass das korrekte Speichermodell benutzt wird, falls sich die

Listing 1: sequentielle Version

```
1 for (int i = 0; i < N; i++) {
2   double sum = 0.0;
3   for (int j = 0; j < M; j++)
4     sum += A[i][j];
5   sum = 1.0 / (1.0 + exp(-sum));
6   B[i] += sum;
7 }
```

Listing 2: *PolyLibTLS*-Version

```
1 SpRO spA(A, M * N);
2 SpSC spB(B, N);
3 USpM<SpRO, &spA, SpSC, &spB> spU;
4 ...
5 double sum = 0.0;
6 for (int j = 0; j < M; j++)
7   sum += spU.specLD<SpRO, &spA>(&A[i][j]);
8 sum = 1.0 / (1.0 + exp(-sum));
9 spU.specST<SpSC, &spB>(&B[i], spU.specLD<SpSC, &spB>(&B[i
    ]) + sum);
```

Speicherbereiche von `A` und `B` überlappen.

Die Zeilen 7 und 9 aus Listing 2 sind die ins spekulative Modell übersetzten Zuweisungen aus den Zeilen 4 und 6 von Listing 1. Da die Variable `sum` lokal ist und somit keine Interferenz mit anderen Threads passieren kann, brauchen die Zugriffe auf `sum` auch nicht spekulativ sein.

3.2.3 Speedup

Der Speedup, der durch die Parallelisierung eines Programms mithilfe der Bibliothek *PolyLibTLS* erreicht werden kann, wurde in [OM08] zunächst für einige synthetische Programme untersucht, die jeweils aus einer einzelnen Schleife bestanden. Diese Schleife war in einem Falls leer, in einem anderen Fall enthielt sie nur eine Leseanweisung bzw. eine Schreibanweisung. Der Speedup für solche synthetischen Programme ist jedoch für die Praxis nicht besonders interessant, da es ja gerade darum geht, nicht nur spezielle (mathematische) Programme zu parallelisieren, sondern alltägliche Programme.

In einem zweiten Versuch wurde dann auch versucht, bekannte Benchmark-Programme mit der Bibliothek zu parallelisieren. Die Ergebnisse sind in Abbildung 4 dargestellt. Die erste Spalte der Tabelle enthält den Namen des jeweiligen Benchmark-Programms. In der zweiten Spalte steht der Speedup, der erreicht werden konnte, wenn man das gegebene Programm per Hand parallelisierte. Die dritte und vierte Spalte enthalten die Speedups unter Verwendung der Bibliothek, wobei in der dritten Spalte lediglich das klassische Speichermodell zum Einsatz kam, während in der vierten Spalte zusätzlich das Modell, das nur Lesezugriffe erlaubt, verwendet wurde.

Man sieht sofort, dass der Speedup, der unter Verwendung der Bibliothek erreicht wurde, nicht an den eines handparallelisierten Programms heranreicht. Jedoch kann

Seq/Spec	Hand Par	SpLSC	+SpRO
IDEA Cipher	3.91	2.25	3.22
IDEA DeKey	3.89	1.85	3.09
SparMatMult	2.11	1.25	2.00
NeuralNetFW	2.04	1.19	1.90
NeuralNetBW	1.52	0.75	1.26
FFTtransf	1.99	0.83	0.83

Abbildung 4: erreichbarer Speedup mit *PolyLibTLS*

bei den meisten Programmen gerade bei der Kombination verschiedener Speichermodelle ein deutlicher Speedup erzielt werden, sodass die Parallelisierung mittels *PolyLibTLS* durchaus attraktiv erscheint.

Das schlechte Ergebnis der Fourier Transformation (letzte Zeile) erklären die Entwickler dadurch, dass in diesem Programm gleich viele Schreib- und Lesezugriffe sowie ein gezwungenermaßen sequentieller Abschnitt vorkommen. Sie betonen jedoch, dass man auch hier mit anderen Speichermodellen einen Speedup von 1,83 erreichen kann.

3.3 Hardware-Umsetzung

Jeder Multicore besitzt in irgendeiner Form ein Cache-Kohärenz-Protokoll, da irgendwie geregelt werden muss, wie die verschiedenen Kerne auf den Speicher zugreifen. Eine verbreitete Architektur, die hier als Referenz verwendet werden soll, besitzt einen privaten Level-1-Cache pro Prozessorkern und einen gemeinsamen Level-2-Cache (vgl. Abbildung 5). Dabei ist es für die hier vorgestellte Anwedung wichtig, ob sich z. B. zwischen Level-2-Cache und Hauptspeicher noch eine Stufe der Cache-Hierarchie befindet oder zwischen Level-1-Cache und Prozessorkern noch ein Cache-Level dazwischengeschaltet ist. Wichtig ist lediglich, dass jeder Kern einen privaten Teil an Speicher hat und dass es einen gemeinsamen Speicher gibt.

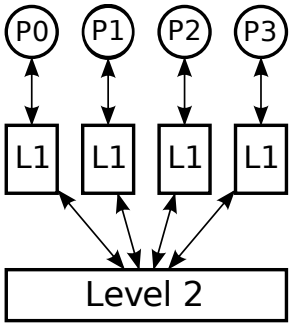

Abbildung 5: abstrakte Architektur

Die Hardware-Implementierung, oder besser hardwareunterstützte Implementierung, von Spekulation auf Fadenebene benutzt den privaten Speicher jedes Rechenkerns zur Pufferung der Schreibergebnisse des aktuell auf dem Kern ausgeführten Threads. Damit einhergehend wird das Cache-Kohärenz-Protokoll so erweitert, dass es dazu verwendet werden kann, verletzte Datenabhängigkeiten zwischen den Threads auf den verschiedenen Kernen zu erkennen und einen Rollback durchzuführen.

Dazu wird auch hier wieder das Schreiben in den gemeinsamen Speicher serialisiert, sodass nur der Kern mit dem aktuell niedrigsten Iterations-Index seine Ergebnisse vom Level-1- in den Level-2-Cache schreiben darf. Erreicht werden kann dies z. B. durch Herumreichen eines Token zwischen den Threads. Der aktuell älteste Thread hält das Token so lange, bis alle seine Ergebnisse in den globalen Speicher geschrieben

wurden und gibt es dann an seinen Nachfolger weiter.

Diese Art der Umsetzung von Spekulation auf Faden- oder Threadebene wurde z. B. in [SM98] und [SCM97] untersucht.

3.3.1 Erweiterung des Cache-Kohärenz-Protokolls

Ein typisches Cache-Kohärenz-Protokoll speichert zu jeder Cache-Line beispielsweise die folgenden Flags:

`D` ***(„dirty“)*** : Die Cache-Line wurde überschrieben, jedoch noch nicht in den tiefer liegenden Hierarchiestufen aktualisiert.

`S` ***(„shared“)*** : Eine Kopie der Cache-Line befindet sich auch in anderen Caches.

`E` ***(„exclusive“)*** : Der Prozessor hat exklusiven Zugriff auf diese Cache-Line.

`I` ***(„invalid“)*** : Die Cache-Line enthält im Moment keine gültigen Daten.

Die Funktionsweise des Protokolls sei anhand eines Beispiels demonstriert:

- Prozessor 1 lädt einen Wert aus dem Speicher. Die entsprechende Cache-Line wird in den Level-1-Cache des Prozessors geladen. Unter der Annahme, dass noch kein anderer Prozessor einen Wert aus derselben Cache-Line geladen hat, ist sie mit dem Flag `E` versehen.
- Prozessor 2 lädt denselben Wert aus dem Speicher. Die entsprechende Cache-line wird auch in den Level-1-Cache von Prozessor 2 geladen und in beiden Level-1-Caches wird die Cache-Line nun mit dem Flag `S` versehen.
- Möchte einer der beiden Prozessoren den Wert in der geladenen Cache-Line verändern, muss er zunächst exklusiven Zugriff auf die Cache-Line erhalten. Dazu wird die Cache-Line aus allen anderen Level-1-Caches entfernt (dort mit dem Flag `I` versehen) und im Cache des schreibenden Prozessors schließlich wieder mit dem Flag `E` versehen. Sobald der Wert geschrieben wurde, ändert sich das Flag zu `D`. Falls nun ein anderer Prozessor auf den Wert zugreifen möchte, wird dieser zunächst in die tiefer liegende Speicherhierarchie zurückgeschrieben und dann der aktualisierte Wert in den entsprechenden Level-1-Cache geladen.

Um mit einem abgewandelten Protokoll spekulative Speicherzugriffe zu puffern und Verletzungen von Datenabhängigkeiten zu erkennen, werden drei zusätzliche Flags eingeführt:

`SpecLD` : Die Cache-Line wurde spekulativ geladen.

`SpecST` : Die Cache-Line wurde spekulativ überschrieben.

`V` ***(„violation“)*** : Die Cache-Line hat eine Abhängigkeit verletzt und die enthaltenen Werte sind somit nicht mehr gültig.

Kombiniert man diese neuen Flags mit den bereits bestehenden, so ergeben sich eine Menge Kombinationen, die jedoch teilweise nicht zulässig sind. So darf es z. B. nicht sein, dass eine Cache-Line gleichzeitig mit `D` und `SpecST` markiert ist, da eine als `D` markierte Cache-Line die einzige gültige Kopie der in ihr enthaltenen Werte speichert und diese zunächst in die tieferliegenden Speicherhierarchie zurückgeschrieben werden müssen, bevor sie spekulativ modifiziert werden dürfen.

Zeit	Befehl T1	Cache T1	Befehl T2	Cache T2
1	...		...	
2	...		`LOAD %2, 34`	34, {SpecLD}
3	`STORE 34, %3`	34, {SpecST, SpecLD}	...	34, {SpecLD}
4	`commit()`	34, {SpecST, SpecLD}	...	34, {SpecLD, V}
5	...		~~`commit()`~~	34, {SpecLD, V}

Abbildung 6: Beispiel zur Erkennung der Verletzung einer RAW-Abhängigkeit. In den Cache-Lines ist jeweils die Adresse des geladenen Wertes sowie eine Menge an Flags angegeben.

3.3.2 Beispiel zur Funktionsweise

Die Funktionsweise des vorgestellten Verfahrens wird nun anhand des Beispiels der Verletzung einer RAW-Abhängigkeit dargestellt (vgl. Abbildung 6). Um die Übersichtlichkeit zu wahren, werden dabei nicht alle Flags der Cache-Lines angegeben, sondern lediglich die für die Spekulation interessanten.

Angenommen es werden auf zwei Prozessorkernen zwei Epochen in zwei Threads parallel ausgeführt. Thread 2 lädt nun einen Wert und damit die entsprechende Cache-Line in seinen lokalen Level-1-Cache. Die Cache-Line wird als `SpecLD` markiert (vgl. Zeitschritt 2 in Abbildung 6). Im nächsten Zeitschritt lädt Thread 1 ebenfalls eine spekulative Kopie derselben Speicherzelle in seinen Level-1-Cache, allerdings um sie zu überschreiben. Demzufolge wird die Cache-Line im Cache von Thread 1 mit den Flags `SpecLD` und `SpecST` versehen.

Sobald Thread 1 seine Epoche abgearbeitet hat, werden die lokalen Ergebnisse im Cache von Thread 1 in den globalen Speicher übernommen (da Thread 1 momentan die älteste Epoche ausführt, ist das erlaubt). Dabei schickt Prozessorkern 1 für jede zu überschreibende Speicherzelle eine Nachricht an alle anderen Kerne. Falls in einem der Level-1-Caches der anderen Kerne eine Kopie derselben Speicherzelle vorliegt und als spekulativ geladen markiert ist (was im Beispiel der Fall ist), wurde eine RAW-Abhängigkeit verletzt und die entsprechende Cache-Line wird als `V` markiert (vgl. Zeitschritt 4 in Abbildung 6).

Sobald Thread 2 seine Commit-Phase erreicht (vgl. Zeitschritt 5 in Abbildung 6) und die Hardware feststellt, dass im Level-1-Cache als `V` markierte Cache-Lines vorliegen, wird eine Software-Routine aufgerufen, die für einen Rollback sucht. Im einfachsten Fall wird die betroffene Epoche neu gestartet.

3.3.3 Erweiterungsmöglichkeiten

Ähnlich wie bei der reinen Software-Umsetzung sind auch im Falls der hardwareunterstützten Implementierung einige Erweiterungen denkbar, welche die Effizienz des parallelisierten Programms weiter erhöhen.

Ein Ansatzpunkt ist das Neustarten von Epochen im Fall entdeckter Abhängigkeitsverletzungen. Eine Möglichkeit wäre herauszufinden, an welcher Stelle die Abhängigkeit verletzt wurde und die Epoche nur bis zu diesem Zeitpunkt zurückzuset-

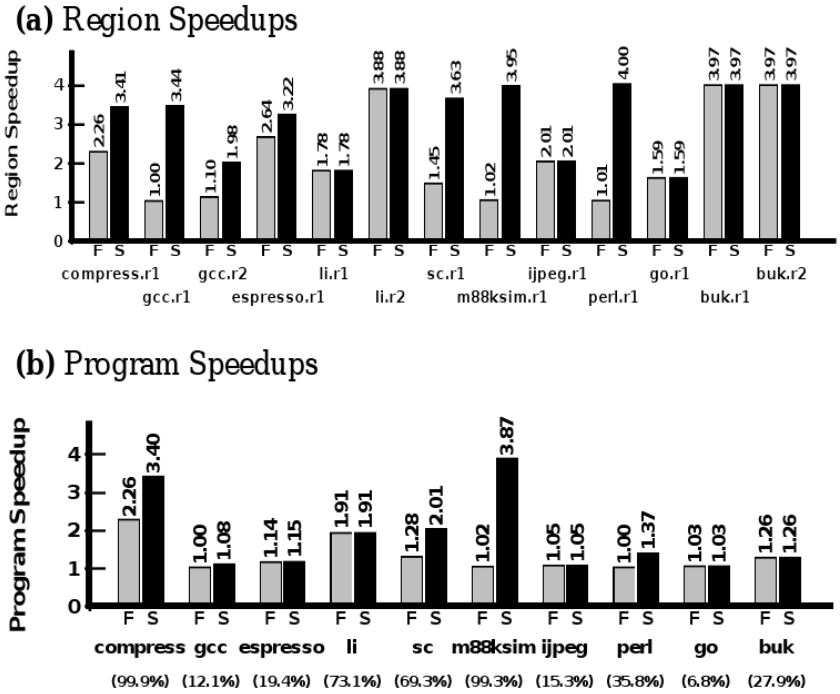

Abbildung 7: Erreichbarer Speedup mit einer hardware-basierten Implementierung. In Teil b ist in Prozent der Anteil an parallelisierten Regionen am gesamten Programmcode angegeben.

zen. Das Problem ist hier, dass mit dieser Vorgehensweise ein sehr hoher Aufwand verbunden ist, da z. B. alle Modifikationen am lokalen Speicher aufgezeichnet werden müssten, um hinterher rückgängig gemacht werden zu können. Etwas einfacher erscheint da die bereits in Abschnitt 3.1.2 angesprochene Weiterleitung einzelner Werde zwischen verschiedenen Epochen.

Eine andere Erweiterung, die oft implementiert wird und z. B. auch in [SCZM00] beschrieben ist, befasst sich mit dem Aufheben der Serialisierung der Schreibvorgänge in den globalen Speicher. Neben einer geringeren Abbruchrate der Epochen erreicht man dadurch auch eine bessere Skalierbarkeit. Beim serialisierten Schreiben in den globalen Speicher kann diese Operation bei vielen Prozessoren zum Flaschenhals werden, sodass alle Prozessoren die meiste Zeit darauf warten, dass ihre Epoche die älteste wird und sie ihre Ergebnisse endlich in den globalen Speicher schreiben dürfen.

3.3.4 Speedup

Abbildung 7 zeigt die Ergebnisse, die Steffan und Mowry in [SM98] mit ihrer hardwarebasierten Implementierung von Spekulation auf Fadenebene für einige Benchmarkprogramme erreicht haben. In Teil a ist dabei der Speedup der parallelisierten Regionen aufgetragen und Teil b zeigt jeweils den Speedup des gesamten Programms. Der Anteil der parallelisierten Regionen am Code des Programms steht ganz unten in Klammern.

Die mit einem F markierten helleren Balken zeigen jeweils den Speedup an, der

mit den in dieser Arbeit vorgestellten Methoden erreicht wurde. Bei den mit einem S markierten dunkleren Balken wurden zusätzlich die einzelnen Epochen so weit es ging vertauscht, um zusätzlich Datenabhängigkeiten zu vermeiden. Diese Optimierungen wurden dabei teilweise von Hand vorgenommen und die Autoren sagen selbst, dass es unwahrscheinlich ist, dass diese Optimierungen von einem Compiler automatisch durchgeführt werden können.

Trotzdem sind bereits die im Fall F erzielten Speedups bei den meisten Benchmarkprogrammen beachtlich, wenn man die geringe Überdeckung der Programme in Betracht zieht. Der geringe Anteil an parallelisiertem Code liegt laut den Autoren darin begründet, dass ihre Tools bisher nicht spekulative Regionen automatisch erkennen können, sondern diese von Hand gefunden werden mussten. Bei einem großen Programm wie z. B. dem gcc wurde somit nur bei einem geringen Teil des Programms überhaupt eine Parallelisierung versucht.

4 Zusammenfassung und Ausblick

In dieser Arbeit wurden verschiedene Verfahren vorgestellt, um sequentielle Programme möglichst automatisch zu parallelisieren. Zunächst wurde das klassische Polytopmodell vorgestellt und zwei Erweiterungen diskutiert, die es ermöglichen, mit dem Modell auch `while`-Schleifen zu erfassen.

Der hauptsächliche Fokus der Arbeit lag dann auf Techniken der spekulativen Parallelisierung von Programmen. Dieser Ansatz hat den Vorteil, dass einige Einschränkungen des Polytopmodells aufgehoben werden können, da Abhängigkeiten nicht mehr statisch zur Übersetzungszeit, sondern dynamisch zur Laufzeit analysiert werden. So ist es möglich, auch Programme mit komplexen Speicherzugriffsmustern und komplexem Kontrollfluss zu parallelisieren. Durch Analyse der Datenabhängigkeiten zur Laufzeit und eventuelle Rollbacks wird die Konsistenz des Programms gewahrt.

Nachdem die grundlegenden Anforderungen an Systeme spezifiziert wurden, die Spekulation auf Fadenebene implementieren, wurden zwei konkrete Implementierungsvorschläge vorgestellt und diskutiert. Die reine Software-Umsetzung bietet den großen Vorteil, dass keine Änderungen an bestehender Hardware erforderlich ist und auf diese Weise parallelisierte Programme somit auf den bereits verfügbaren Multicores laufen. Die hardware-unterstützte Realisieren nutzt dagegen die bestehenden Strukturen der Cache-Kohärenz-Protokolle und erweitert diese so, dass die Abhängigkeitsanalyse zur Laufzeit durch die Hardware vorgenommen werden kann.

Beide Ansätze konnten bislang das hochgesteckte Ziel der vollständig automatischen Parallelisierung nicht erreichen. Sie bieten dem Entwickler jedoch ein Framework an, das sich bereits um viele Dinge (wie Threadverwaltung, Scheduling etc.) kümmert. Je nach Anwendung kann es sich also lohnen, mit ein bisschen Mehraufwand ein paralleles Programm zu erhalten.

Experimentell konnten mit beiden Implementierungen gute Speedup-Werte für Benchmarkprogramme erzielt werden. Es bietet sich also an, bereits jetzt eine Softwarelösung zu nutzen, um bestehende sequentielle Programm zu parallelisieren und in der Zukunft eine Unterstützung für spekulative Techniken schon in der Hardware vorzusehen.

Literaturverzeichnis

[Col94] Jean-Francois Collard. Space-Time Transformation of while-loops using Speculative Execution. In *Proc. Scalable High Performance Computing Conf.*, pages 191–219, Knoxville, TN, May 1994. IEEE Computer Society Press.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[Gri96] Martin Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, University of Passau, 1996. also available as technical report MIP-9701.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[LG95] C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.

[OM07] Cosmin E. Oancea and Alan Mycroft. A lightweight model for software thread-level speculation (tls). In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 419, Washington, DC, USA, 2007. IEEE Computer Society.

[OM08] Cosmin E. Oancea and Alan Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 23–32, New York, NY, USA, 2008. ACM.

[PM01] Spiros Papadimitriou and Todd C. Mowry. *Exploring Thread-Level Speculation in Software: The Effects of Memory Access Tracking Granularity.* School of Computer Science, Carnegie Mellon University, 2001.

[SCM97] J.G. Staffan, C.B. Colohan, and T.C. Mowry. Architectural support for thread-level data speculation. Technical report, Tech Report CMU-CS-97-188, Carnegie Mellon University, School of Computer Science, 1997.

[SCZM00] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, 2000.

[SM98] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.