FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Optimierung von Mehrgitteralgorithmen auf der IA-64 Rechnerarchitektur

Markus Stürmer

Diplomarbeit

Optimierung von Mehrgitteralgorithmen auf der IA-64 Rechnerarchitektur

Markus Stürmer

Diplomarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Ing. J. Treibig
Bearbeitungszeitraum:	1.11.2005 – 2.5.2006

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 2. Mai 2006 .....

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die numerische Simulation ist heute ein etabliertes Werkzeug für Wissenschaft und Wirtschaft, um teure, gefährliche oder schwer beobachtbare Abläufe nachzustellen. In vielen Bereichen setzt der immense Bedarf an Rechenleistung dieser Anwendung aus Zeit- und Kostengründen jedoch Grenzen.

Dem kann zum einen durch Entwicklung neuer Modelle und effizienterer Lösungsverfahren begegnet werden. Die andere Möglichkeit und Thema dieser Arbeit ist es jedoch, die verfügbaren Ressourcen eines Computersystems möglichst gut zu nutzen.

Physikalische Zusammenhänge lassen sich häufig über Differentialgleichungen modellieren, die im Allgemeinen jedoch nicht analytisch gelöst werden können. Ein gebräuchlicher Ansatz betrachtet den Zustand des zugrunde liegenden Modells nur an einer begrenzten Anzahl von Orten und reduziert das Problem auf die Lösung meist sehr großer Linearer Gleichungssysteme.

Mehrgitterverfahren stellen derzeit die effizientesten Verfahren zum Lösen solcher Gleichungssysteme dar und bieten sich deshalb für eine intensive Optimierung an.

Der primäre Ansatz ist dabei, den „Memory Wall“ zu umgehen, das zunehmende Missverhältnis zwischen der Verarbeitungsgeschwindigkeit moderner Prozessoren und der Bandbreite verfügbarer Hauptspeicher. Dabei wird versucht, einmal in die schnellen Caches gelangte Operanden möglichst effektiv zu Nutzen, um den Transfer in den Hauptspeicher zu reduzieren.

Viele der dazu angewandten Techniken konnten bei zweidimensionalen Problemen eine große Steigerung der Rechengeschwindigkeit erreichen [Wei01], insbesondere auf moderneren Prozessoren konnte dies aber nicht ebenso erfolgreich auf drei Dimensionen übertragen werden [Thü02][Kow04]. Dabei wurde jedoch von einer allgemeineren, abstrakten Vorstellung von Prozessor und Cachehierarchie ausgegangen und entsprechend in einer portablen Hochsprache programmiert.

Diese Arbeit konzentriert sich deshalb speziell auf eine Architektur und einen Mehrgitteralgorithmus, auf die die Optimierungen direkt zugeschnitten werden. So können die Möglichkeiten spezieller Techniken und von Optimierungsverfahren insgesamt genauer untersucht werden, ohne stets auf allgemeine Anwendbarkeit achten zu müssen.

Die IA-64-Architektur stellt für diese Anwendung eine interessante Plattform dar: Sie bietet ein hohes Potential an Rechengeschwindigkeit und große, gut angebundene Caches, besitzt jedoch eine einfachere Ausführungslogik als andere moderne Mikroprozessoren. Maschinensprache-Code in hoher Qualität ist deshalb schwieriger zu erzeugen, erlaubt aber auch eine genauere Kontrolle über die Ausführung und die Lokalität von Daten.

Die Wahl des Modellalgorithmus fiel auf ein einfaches Mehrgitterverfahren zur Lösung der Poisson-Gleichung. Das Problem ist speziell genug, um gut verstanden und stark optimiert zu werden.

Andererseits findet es sowohl in der Praxis als auch in der Theorie häufig Anwendung, so dass ausreichend Vorarbeiten und Vergleichsmöglichkeiten zur Verfügung stehen.

1.2 Aufbau

Im folgenden Kapitel sollen die nötigen *Grundlagen* zum Verständnis von Auswahl und Funktionsweise der angewandten Optimierungsverfahren vermittelt werden. Die erste Hälfte des Kapitels schildert deshalb das Mehrgitterverfahren und stellt seine Funktionsweise in Grundzügen dar. Es folgt eine Einführung in die IA-64-Architektur und eine Darstellung ihres aktuellen Vertreters, des Intel Itanium 2-Prozessor, sowie spezieller Konzepte und Instruktionen dieser Plattform.

Das dritte Kapitel beschreibt die durchgeführten *Optimierungen*. Es beginnt mit allgemeinen Techniken, die für viele numerische Algorithmen auf dieser Architektur genutzt werden können. Anschließend wird das Mehrgitterverfahren genauer analysiert, um darauf aufbauend die konkrete Optimierung des Algorithmus und seiner Komponenten nachzuvollziehen.

Das vierte Kapitel versucht anhand von Geschwindigkeitsmessungen den Erfolg der verschiedenen Optimierungstechniken zu *beurteilen* und schließt mit einer Einordnung der Leistungsfähigkeit der Plattform und des optimierten Mehrgitterlösers.

Kapitel 2

Grundlagen

2.1 Mehrgitterverfahren

Zur Lösung Linearer Gleichungssysteme, wie sie auch bei der Diskretisierung von Differentialgleichungen entstehen, bestehen grundsätzlich zwei Ansätze: Zum einen das direkte Lösen der Gleichung, zum anderen iterative Verfahren, die eine Anfangslösung durch wiederholte Berechnungen verbessern.

Direkte Löser leiden stärker unter der Ungenauigkeit der Computer-Algebra und lassen sich nur bedingt parallelisieren. Iterative Löser sind meist allgemeiner anwendbar, leichter zu parallelisieren und können sehr schnell Näherungslösungen berechnen.

Aber iterative Verfahren – darunter auch Jacobi- und Gauss-Seidel – haben meist die Eigenschaft von Glättern: Sie können hochfrequente Fehler sehr schnell aus der Ausgangslösung entfernen, benötigen allerdings sehr viele Iterationen zur Beseitigung „glatter“ Fehler. Dies kann zum einen experimentell gezeigt werden, indem eine bekannte Lösung gezielt mit bestimmten Störungen überlagert und als Anfangslösung genutzt wird, dies lässt sich aber ebenso rein mathematisch zeigen.

2.1.1 Einführung

Diese Schwäche iterativer Löser kann für viele Probleme durch Mehrgitterverfahren beseitigt werden, die eine Komplexität bis zu $\mathcal{O}(n)$ erreichen. Im folgenden werden die Grundlagen für den zu optimierenden Modellalgorithmus soweit umrissen, dass Grundidee und Funktionsweise verständlich werden. [BHM00]

2.1.1.1 Residualgleichung

Für das lineare Gleichungssystem

$$A \cdot u = f$$

sei v eine Näherungslösung des Vektors u.

Da u nicht bekannt ist, wird anstatt einer Maximal- oder euklidischen Norm des Fehlervektors

$$e = u - v$$

eine entsprechende Norm des Residuums

$$r = f - A \cdot v$$

verwandt.

Das Residuum ist dabei nur ein indirektes Maß für den Fehler – es beschreibt punktweise, wie weit die Näherungslösung die Gleichung erfüllt. Allerdings ist das Residuum genau dann ein Nullvektor, wenn keine Fehler mehr vorhanden ist.

Durch einfaches Einsetzen lässt sich die Residualgleichung

$$r = A \cdot u - A \cdot v = A \cdot e$$

herleiten.

2.1.1.2 Gitterfeinheit

Betrachtet man dieselbe Näherungslösung, und somit auch denselben Fehler, auf einem feineren und einem gröberen Gitter, so unterscheidet sich das Frequenzspektrum. Glatte Anteile schwingen auf dem gröberen Gitter stärker und sind somit leichter zu entfernen. Besonders hochfrequente Anteile werden jedoch zu glatten Anteilen, wenn ihre Frequenz im gröberen Gitter nicht mehr dargestellt werden kann – ein Vorgang, der aus der digitalen Messtechnik als Aliasing bekannt ist.

2.1.1.3 Korrektur-Schema

Die Kombination dieses Vorwissens führt direkt zum Modell-Mehrgitterverfahren:

Beginnend mit der Anfangslösung v für $A \cdot v = f$ werden die hochfrequenten Fehler durch wenige Iterationen eines iterativen Lösers entfernt, der deshalb meist als Glätter bezeichnet wird, und das zugehörige Residuum $r = f - A \cdot v$ berechnet. Anhand der Residual-Gleichung sollte nun eine gute Näherung des Fehlers e gefunden werden, mit der eine Verbesserung von v möglich ist.

Das Mehrgitterverfahren gewinnt seine Effizienz, indem die Berechnung eines genäherten Fehlers $A \cdot e = r$ auf einem gröberen Gitter durchgeführt wird. Dafür muss eine gröbere Fassung des Residuums r hergestellt werden (Restriktion), und der „grob“ berechnete Fehler zur Korrektur wieder verfeinert werden (Interpolation). Da die Lösung um einen interpolierten Fehler korrigiert wurde, werden häufig noch wenige Nachglättungsschritte durchgeführt.

Die Lösung der Residual-Gleichung auf dem gröberen Gitter kann dann rekursiv auf immer gröber werdenden Gittern durchgeführt werden, bis die Größe eine schnelle exakte Lösung zulässt.

Dieses Verfahren – zunehmende Vergröberung der Residua und anschließende Korrektur um die verfeinerte Näherung des Fehlers – wird als V-Zyklus (siehe Abb. 2.1) bezeichnet und kann so lange angewandt werden, bis eine ausreichend genaue Lösung berechnet wurde. Auch Modifikationen wie der W-Zyklus, der die Verfeinerungsphase vor Erreichen des feinsten Gitters nochmals durch eine Vergröberung ergänzt, sind gebräuchlich.

Nun stellt sich die Frage, wie Vergröberungen und Verfeinerungen von Gittern zu berechnen sind. Häufig beschränkt man sich hier auf Gitter mit $2^n - 1$ Unbekannten je Dimension, so dass die Verfeinerung durch einfache lineare Interpolation und die Restriktion durch eine gewichtete Summe der umliegenden Punkte realisiert werden kann.

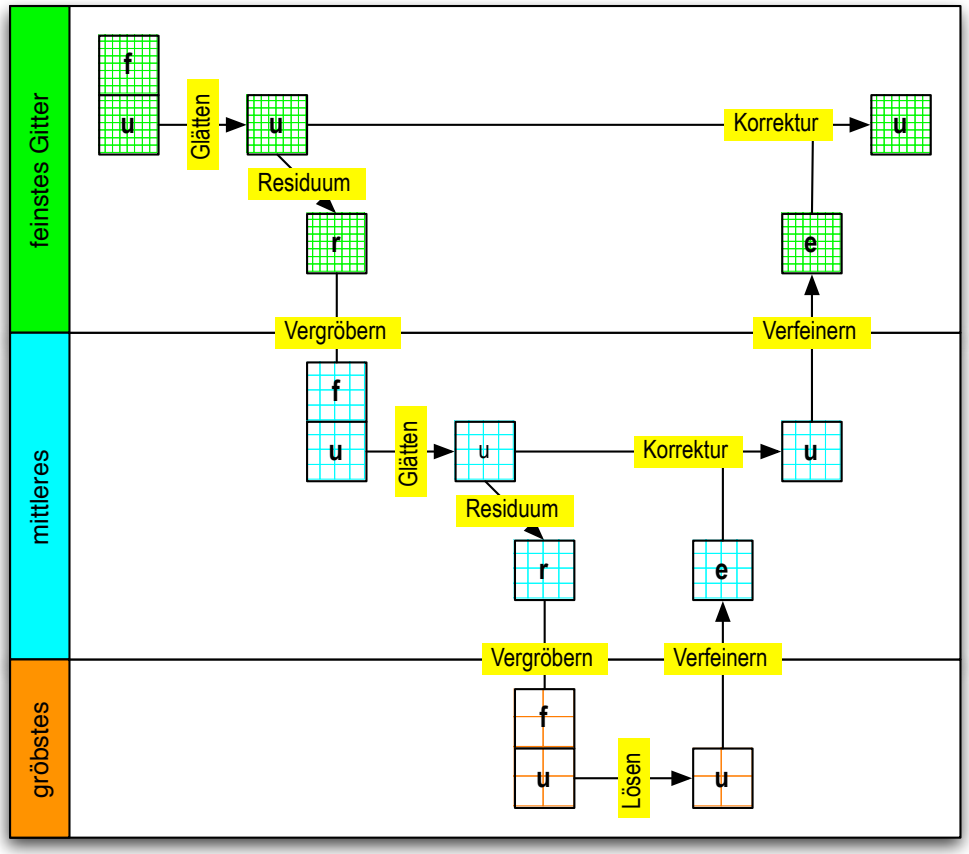
Letztendlich kann man kaum von einem einzigen Mehrgitterverfahren, sondern sollte eher von einer Mehrgitter-Methode sprechen. Dabei stehen nicht nur verschiedene Glätter, Restriktionen und Interpolationen zur Verfügung. Mit entsprechenden Anpassungen kann das Prinzip auch auf unstrukturierte Gitter oder Probleme ohne Gitterstruktur angewandt werden.

2.1.2 Modellalgorithmus

Als Modellalgorithmus dient ein einfaches Mehrgitterverfahren, mit dem eine Diskretisierung der dreidimensionalen Poisson-Gleichung

$$\Delta\phi = \frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2} + \frac{\delta^2}{\delta z^2} = f$$

Abbildung 2.1: Einfacher V-Zyklus



auf einem kubischen Gebiet Ω gelöst werden kann.

Es werden nur Dirichlet-Randbedingungen zugelassen, die Werte am Rand $\delta\Omega$ sind also vorgegeben.

Diese Partielle Differentialgleichung wird auf einem regelmäßigen Gitter der Größe $(2^n \cdot dh) \times (2^n \cdot dh) \times (2^n \cdot dh)$ mit Gitterabstand dh diskretisiert, und der Laplace-Operator Δ über Finite Differenzen zu

$$\begin{aligned}\Delta^\star = & \frac{\phi(x-dh,y,z) - 2\cdot\phi(x,y,z) + \phi(x+dh,y,z)}{h^2} \\ & + \frac{\phi(x,y-dh,z) - 2\cdot\phi(x,y,z) + \phi(x,y+dh,z)}{dh^2} \\ & + \frac{\phi(x,y,z-dh) - 2\cdot\phi(x,y,z) + \phi(x,y,z+dh)}{dh^2}\end{aligned}$$

angenähert.

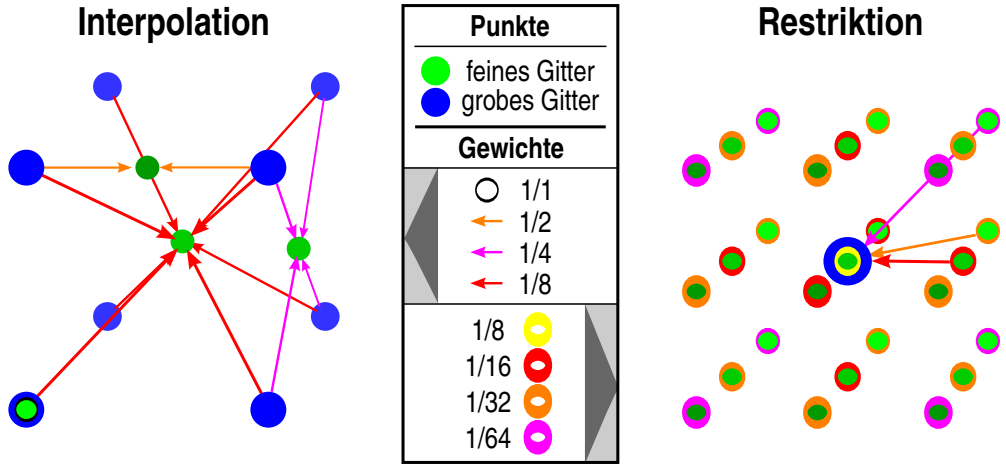
Als Glätter wird das Gauss-Seidel-Verfahren mit Rot-Schwarz-Anordnung der Unbekannten eingesetzt. Die Relaxation eines Wertes wird demnach über den 5-Punkt-Stempel

$$V_{x,y,z} = \frac{1}{6} \cdot (V_{x-1,y,z} + V_{x+1,y,z} + V_{x,y-1,z} + V_{x,y+1,z} + V_{x,y,z-1} + V_{x,y,z+1} - dh \cdot F_{x,y,z})$$

durchgeführt. Als „rot“ werden diejenigen Werte bezeichnet, für die $x + y + z$ ungerade ist. Eine vollständige Iteration besteht aus der Berechnung zunächst neuer roter und anschließend neuer schwarzer Werte, was im folgenden der Kürze halber als rote und schwarze Halbiteration bezeichnet wird. Alle Berechnungen einer Halbiteration besitzen keinerlei Datenabhängigkeiten voneinander und können deshalb in beliebiger Reihenfolge durchgeführt werden.

Die Verfeinerung wird durch trilineare Interpolation berechnet. Zur Restriktion wird die gewichtete Summe der benachbarten Gitterpunkte genutzt, wobei die Gewichte entgegengesetzt zur Interpolation gesetzt sind. (siehe Abb. 2.2)

Abbildung 2.2: Trilineare Interpolation und zugehörige Restriktion



Als Anfangslösung wird stets $U_{x,y,z} = 0$ genutzt und die Verfeinerung bis zu einem Gitter mit genau einer Unbekannten durchgeführt, das Gleichungssystem kann dann mit einer Relaxation exakt gelöst werden.

V-Zyklus in Pseudo-Code

```
globale anzahl Pre
globale anzahl Post

funktion Vzyklus(gitter V, gitter RS) {
    wenn (Anzahl_Unbekannte(V) == 1) {
        Glätte(V,RS,1) /* exakte Lösung */
    } sonst {
        neues feines gitter R
        neues feines gitter E
        neues grobes gitter Rgrob
        neues grobes gitter Egrob

        Glätte(V,RS,Pre)

        Setze_Anfangslösung(Esml)

        R = Residuum(V,RS)
        Rsml = Restriktion(R)

        Vzyklus(Egrob, Rgrob)

        E = Interpoliere(Egrob)
        V = V + E

        Glätte(V,RS,Post)
    }
}
```

2.2 Die IA-64-Architektur

Seit ihrer Erfindung wurden Mikroprozessoren immer leistungsfähiger und schneller; dies verdanken sie nicht nur der Entwicklung neuer Fertigungsverfahren, die innerhalb kurzer Zeit ja für alle Prozessoren zur Verfügung stehen.

Ebenso interessant sind die Strategien, Methoden und Konzepte, mit denen in den jeweiligen Prozessoren die höhere Komplexität in Geschwindigkeit umgesetzt wird. Die meisten Prozessoren führen mithilfe komplexer Logik Befehle zunehmend überlappt (Pipelining), auf Verdacht (spekulativ), in anderer Reihenfolge und parallel (Out of Order und superskalar) aus. Die Instruktionssätze beschreiben allerdings Aktionen, die jeweils nacheinander ausgeführt werden sollen. Ein Großteil der Prozessorlogik ist dann dafür nötig, die ursprüngliche Semantik sicherzustellen.

Die von Intel und Hewlett Packard entwickelte IA-64-Architektur und der zugehörige Befehlssatz realisieren dabei einen anderen Ansatz: Der Prozessorkern ist dabei weniger komplex und führt Instruktionen in ihrer ursprünglichen Reihenfolge aus. Dadurch ergibt sich die Möglichkeit, größere Registersätze, größere Caches und mehr Recheneinheiten zur Verfügung zu stellen. Der Befehlssatz bietet spezielle und zum Teil neuartige Instruktionen und Konzepte, die die Nachteile einer In-Order-Ausführung beseitigen sollen.

Während es bei anderen Architekturen Aufgabe des Prozessors ist, mögliche parallele Ausführung von Instruktionen (Instruction Level Parallelism) festzustellen und zu nutzen, muss diese vom Compiler oder Assembler-Programmierer explizit im IA-64-Befehlsstrom ausgewiesen werden. Zudem können Ausführungs- und Speicherlatenzen nur durch geschicktes Scheduling oder Nutzung spezieller Instruktionen während der Code-Erzeugung verborgen werden.

2.2.1 Explicitly Parallel Instruction Computing

Der IA-64-Befehlssatz realisiert das Explicitly Parallel Instruction Computing. Bei EPIC wird der Befehlsstrom durch Stopps in Instruktionsgruppen zerteilt, deren Instruktionen sowohl parallel als auch sequentiell abgearbeitet werden können.

Für die Instruktionen in einer Gruppe folgt daraus, dass ein geändertes Registerdatum nicht von nachfolgenden Befehlen verwendet (RAW – read after write) oder mehrfach geschrieben werden darf (WAW – write after write). Eine Instruktion darf allerdings ein Datum verändern, das von einer vorhergehenden Instruktion gelesen wurde (WAR - write after read). Es gibt wenige Sonderfälle, bei denen RAW- und WAW-Abhängigkeiten erlaubt sind, bei anderen können durch Seitenwirkungen nicht offensichtliche Abhängigkeiten auftreten. Für Operationen auf dem Hauptspeicher gelten diese Einschränkungen zunächst nicht, bei Schreiben und anschließendem Lesen desselben Datums in einer Instruktionsgruppe ist allerdings mit Zeiteinbußen zu rechnen.

Um die Dekodierung durch den Prozessor zu erleichtern, werden die Instruktionen zu Bundles aus je drei Instruktionen bestimmten Typs gruppiert. Die Itanium-Architektur definiert eine größere Anzahl von Templates, also Arten von Bundles. Das Template legt fest, ob und wo Stopps auftreten, und welche Slots das Bundle besitzt.

Jeder Slot kann nur eine bestimmte Kategorie von Instruktionen aufnehmen, die jeweils einer Art von Ausführungseinheit entspricht. Diese sind Speicher- (M), Gleitpunkt- (F), komplexe Integer- (I) und Sprunginstruktionen (B). Die sogenannten A-Instruktionen sind einfache Integer-Operationen, die sowohl in M- als auch I-Slots kodiert werden können. Außerdem gibt es spezielle Operationen (weite Sprünge, große Direktoperanden), die zwei Slots belegen (LX) und mit speziellen Templates kodiert werden müssen.

Es existieren folgende Slot-Kombinationen: MII, MMI, MFI, MMF, MIB, MBB, BBB, MMB, MFB und das spezielle MLX. Am Ende jeden Bundles kann zudem ein Stopp kodiert werden. Das MII-Bundle kann zusätzlich ein Stopp zwischen den beiden I-, das MMI-Bundle zwischen den beiden M-Slots enthalten.

Ist der Compiler nicht in der Lage, unter diesen Einschränkungen jeden Slot sinnvoll zu füllen, müssen explizite *No Operation* (nop) oder äquivalente Befehle (z. B. Addition mit 0) verwandt werden. Der IA-64-Befehlssatz definiert dabei für jede Instruktionskategorie eine eigene nop-Instruktion, die tatsächlich eine Recheneinheit belegt.

2.2.2 Registersatz

Im Anwendungsmodus stellt der IA-64-Befehlssatz insgesamt zehn Registerarten zur Verfügung:

General Purpose Registers: Diese 128 Allzweckregister sind jeweils 64 Bit breit. Ein jeweils weiteres zugeordnetes Bit wird für spezielle Befehle genutzt und gibt an, ob das Register einen gültigen Wert enthält (NaT – Not a Thing, siehe 2.2.4.4), es kann aber nicht für Berechnungen genutzt werden. Während die Register r0 bis r31 statisch sind, werden die Register r32 bis r127 als Registerstapel verwaltet. Für Unterprogramme wird transparent eine gewünschte Anzahl von exklusiven Registern zur Verfügung gestellt, durch die Überlappung von Registerfenstern können Parameter effizient übergeben werden. r0 ist fest auf den Wert 0 gesetzt.

Floating-Point Registers: Die 128 Gleitpunktregister f0 bis f127 sind jeweils 82 Bit breit (Vorzeichen, 64 Bit Signifikand, 17 Bit Exponent). Register f0 enthält stets den Wert $+0,0$ und f1 $+1,0$.

Predicate Registers: Die jeweils 1 Bit großen Register p0 bis p63 nehmen die Wahrheitswerte von Vergleichen und Tests auf und ermöglichen die Ausführung von bedingtem Code (siehe 2.2.4.1). p0 ist konstant „wahr“.

Branch Registers: Diese dem Adressraum entsprechend 64 Bit breiten Register b0 bis b7 werden für indirekte Sprünge genutzt, bei Unterfunktionsaufrufen wird die Rücksprungadresse z. B. automatisch in b0 hinterlegt.

Instruction Pointer: Der Instruktionszähler liefert jeweils die Adresse des aktuell ausgeführten Bundles.

Current Frame Marker: Dieses Register enthält Informationen über den Zustand des Registerstapels und die Registerrotation (siehe 2.2.4.2). Meist wird darauf nur indirekt zugegriffen.

Application Registers: Die insgesamt 128 Anwendungsregister werden für verschiedene Aufgaben benötigt und sind teilweise noch für zukünftige Entwicklungen reserviert. Einige der Anwendungsregister sind nur im privilegierten Modus schreib- oder adressierbar. Die meisten von ihnen besitzen auch spezielle Bezeichnungen in der Assemblersprache.

Performance Monitor Data Registers: Diese Register können für Profiling und Benchmarking genutzt werden. Ihre Verwendung wird im privilegierten Modus konfiguriert, dort können sie auch zum Lesen im Anwendungsmodus freigegeben werden.

User Mask: Entspricht einem Teil der Processor Status Registers.

Processor Identification Registers: Durch das Auswerten dieser Register kann ein Programm Version und Cache-Konfiguration der CPU erhalten, auf der es gerade ausgeführt wird.

2.2.3 Vorstellung des Itanium 2

Nachdem die IA-64-Architektur in Grundzügen vorgestellt wurde, beschäftigt sich das folgende Unterkapitel hauptsächlich mit dem Itanium 2-Prozessor als aktuellen Vertreter. Trotz großer Unterschiede in der Taktfrequenz (900 MHz bis 1,66 GHz) und der Größe des L3 Caches (1,5 bis 9 MB), unterscheiden sich seine Varianten trotz überarbeiteter Rechenkerne in Bezug auf Instruktionsverarbeitung oder Recheneinheiten nicht.

2.2.3.1 Recheneinheiten

Alle Itanium 2-Prozessoren beschicken die Recheneinheiten durch insgesamt elf Issue Ports, die den verschiedenen Instruktionskategorien M, I, F und B (vgl. 2.2.1) zugeordnet sind.

Über die vier Ports M0 bis M3 werden u. a. alle Speicheroperationen durchgeführt, über M0 und M1 z. B. alle Lade- und über M2 und M3 alle Speicheroperationen. Gleitpunkt-Ladeoperationen können sogar über alle vier Ports gestartet werden.

Über I0 und I1 werden komplexere Integer-Operationen ausgeführt, z. B. Testen, Setzen oder Extrahieren von Bitfeldern oder verschiedene Multimedia-Funktionen (SIMD-Integer-Operationen, bei denen ganze Register als Array von 1 oder 2 Byte großen Zahlen behandelt werden). Eine Schiebeeinheit kann nur über I1 erreicht werden. Multiplikationen mit Zahlen über 16 Bit müssen auf der Gleitpunkteinheit durchgeführt werden, wofür spezielle Befehle zum Bewegen zwischen Allzweck- und Gleitpunktregister und für eine Fixpunkt-Multiplikation zur Verfügung stehen.

Hinter allen M- und I-Ports ist zudem je eine einfache ALU vorhanden, die u. a. für Additionen, die meisten Vergleiche, aber auch für Postinkrement bei Speicheroperationen genutzt wird. Dadurch können die einfachen A-Instruktionen sowohl als M- als auch als I-Instruktionen auftreten. Ein Itanium 2-Prozessor kann somit bis zu sechs einfache Integer-Berechnungen pro Takt ausführen.

Über die beiden F-Ports kann jeweils eine Gleitpunktoperation gestartet werden. Dabei stellt das „Fused Multiply Add“ $(x \cdot y + z)$ die primäre Operation dar, Addition und Multiplikation werden durch Zuhilfenahme der fest als $+0,0$ und $+1,0$ belegten Register f0 und f1 bewerkstelligt. Zudem sind auch die Varianten „Fused Multiply Subtract“ $(x \cdot y - z)$ und „Fused Negative Multiply Add“ $(-(x \cdot y) + z)$ vorhanden. Der Itanium 2 kann somit getrennt betrachtet bis zu vier Gleitpunktoperationen pro Takt ausführen.

Für Division und Wurzel sieht der Befehlssatz keine einfache Instruktion vor, unterstützt aber die Erzeugung von Näherungslösungen und deren Verfeinerung durch mathematische Verfahren. Eine Sonderrolle nehmen SIMD-Gleitpunkt-Operationen ein, die auf zwei Zahlen einfacher Genauigkeit in einem Register arbeiten: Sie werden in einem Slot kodiert, belegen aber beide Recheneinheiten.

B0, B1 und B2 werden außer für tatsächliche Sprünge auch für weitere, sprungbezogenen Instruktionen benötigt, also auch Zugriffen auf die Sprungregister b0 bis b7 und Instruktionen zur Verbesserung der Sprungvorhersage.

Zwischen dem Auftreten der Instruktionen innerhalb der Bundles im Dispersal Window und den verwendeten Issue Ports besteht zunächst eine feste Zuordnung. Die Itanium-Prozessoren verfügen hierbei nur über bedingte Fähigkeiten zur Umordnung, so dass in einem Takt möglicherweise weniger Instruktionen gestartet werden, als bei optimaler Verteilung möglich wären.

2.2.3.2 Speicherhierarchie

Der Itanium 2 bietet eine dreistufige Cachehierarchie, deren Caches im Prozessortakt arbeiten (siehe Tabelle 2.1). Der L1 Instruction Cache kann pro Takt zwei Bundles (32 Byte) an den Prozessor liefern, in der Gegenrichtung ist kein Datenaustausch vorgesehen. Konsistenz mit dem Hauptspeicher muss bei selbstmodifizierendem Code deshalb explizit hergestellt werden. Der L1 Data Cache kann pro Takt je zwei Anfragen für Lesen und Schreiben bearbeiten und erreicht dabei in jeder Richtung eine Bandbreite von maximal 16 Byte pro Takt, wobei Store Misses direkt an den L2-Cache weitergegeben werden und zu keiner Einlagerung führen.

Der L2-Cache ist 16-Byte-weise in 16 Cachebänke unterteilt und kann pro Takt vier Anfragen, jedoch nur eine Anfrage pro Bank ausführen. Die Gleitpunkteinheit besitzt eine eigene Datenleitung, wodurch entweder 32 Byte gelesen oder je 16 Byte gelesen und geschrieben werden können. Zwei benachbarte Gleitpunktzahlen in derselben Cachebank können im selben Takt nur durch sogenannte „parallele Ladeoperationen“ übertragen werden. Nominal verfügt der L2-Cache über eine Bandbreite von 32 Byte pro Takt, erreicht allerdings 48 Byte beim Zugriff durch zwei parallel Loads und zwei Integer-Instruktionen, die über den L1-Cache zugreifen.

Der L3 Cache kann pro Takt 32 Byte lesen oder schreiben und benötigt somit vier Takte zum Übertragen einer Cachezeile.

Weder Itanium noch Itanium 2 verfügen über komplexe Hardware-Prefetcher, Cachezeilen werden nur entsprechend des vorhergesagten Instruktionsstroms und durch Auswertung von Postinkrementen eingelagert. Um bei Datenströmen hohen Durchsatz erreichen zu können, müssen Werte rechtzeitig mit expliziten Prefetch-Instruktionen vorgeladen werden (siehe 2.2.4.3).

Sowohl Itanium als auch Itanium 2 schreiben Änderungen grundsätzlich in den Cache und wenden auch dann, wenn Write Misses auf allen Cacheebenen auftreten, eine Write-Allocate-Strategie an.

Tabelle 2.1: Caches des Itanium 2

	Größe	**Cachezeile**	**Assoziativität**	**Latenz**
L1 Instruction	16 kB	32 B	4-fach	1 Takt
L1 Data	16 kB	64 B	4-fach	1 Takt
L2 (unified)	256 kB	128 B	8-fach	min. 5*
L3 (unified)	1,5 - 9 MB	128 B	4- oder 2-fach pro MB	min. 12 oder 14*

* 1 zusätzlicher Takt Latenz bei Zugriffen durch die FPU

2.2.3.3 Ausführung

Itanium und Itanium 2 arbeiten den Instruktionsstrom auf dieselbe Art und Weise ab: Der Scheduler betrachtet stets zwei Bundles (*Dispersal Window*) und kann somit pro Takt maximal sechs Instruktionen starten. Sind Bundles komplett abgearbeitet, werden dem Scheduler ein bzw. zwei neue Bundles zugeführt (*Bundle Rotation*).

Der Scheduler muss mit der Ausführung weiterer Instruktionen stets warten (*Split Issue*), wenn

- er ein Stopp antrifft,
- wenn eine Sprunganweisung – auch spekulativ – durchgeführt wird,
- keine entsprechenden Ausführungseinheiten mehr vorhanden oder ihr Issue Port bereits belegt ist, oder
- eine Cachezeilen-Grenze zwischen den Bundles liegt (alle zwei Bundles).

Pro Takt können also maximal zwei Bundles und damit sechs Instruktionen, die alle zu einer Instruktionsgruppe gehören müssen, ausgeführt werden. Der Itanium 2 besitzt allerdings mehr Ausführungseinheiten als sein Vorgänger und kann somit häufiger zwei Bundles pro Takt ausführen (*Dual Issue*).

2.2.4 Besondere Konzepte

Während der Itanium 2 durch seine vergleichsweise einfache In-Order-Architektur ein hohes Leistungspotential besitzt, benötigt er spezielle Konzepte und Unterstützung im Instruktionssatz, um dieses auch bei realen Anwendungen einsetzen zu können. Insbesondere müssen Strategien gefunden werden, die Latenzen durch Speicherzugriff und beim Ausführen von Instruktionen zu verbergen.

Obwohl der Befehlsstrom stets in Bundles kodiert wird, ist der Assembler in der Lage, beliebige Sequenzen von Instruktionen durch Stopps in Instruktionsgruppen zu zerteilen und, wenn nötig durch das Einfügen von nops, in Bundles zu gruppieren. Da hierbei keinerlei Umordnung von Instruktionen stattfindet und der Assembler Stopps defensiv einfügen muss, ist hohe Ausführungsgeschwindigkeit oft nur durch explizites Bundling erreichbar. Die meisten Beispiele sind der Lesbarkeit halber jedoch ohne Bundling und Template-Informationen und enthalten nur die notwendigen Stopps (;;).

2.2.4.1 Prädikate

Bei EPIC können durch Rechenoperationen nicht ohne weiteres implizit Status-Flags gesetzt werden, da die Ergebnisse der (potentiell) parallelen Berechnungen in keiner festen Reihenfolge sichtbar würden. Stattdessen wird das Ergebnis expliziter Vergleiche und Tests in 1 Bit große Prädikatregister geschrieben. Abhängig von diesen Prädikaten können dann zum einen übliche Sprünge, aber auch nahezu alle anderen Operationen bedingt durchgeführt werden.

Die Übersetzung des folgenden C-Fragments kann auf dem Itanium so innerhalb von zwei Takten ausgeführt werden und belegt dabei nur ein Viertel der verfügbaren Slots, die für anderen Berechnungen genutzt werden können:

C-Fragment

```
if (a == 5) b = 6; else a -= 1;
```

IA-64-Entsprechung

```
cmp.eq p1,p2 = 5, r32      // p1 wird dem Vergleich entsprechend auf wahr oder
                           falsch gesetzt, p2 auf nicht-p1
;;                         // Stopp nötig, bevor p1 und p2 gelesen werden können
(p1) add r33 = 6, r0       // kleine Konstanten erzeugt man am schnellsten durch
                           Addition mit dem konstanten Register r0
(p2) add r32 = -1, r32
```

Das folgende Beispiel enthält drei Ausnahmen von Read-After-Write und Write-After-Write-Abhängigkeiten:

- Das Register p0 liefert als Quelle konstant *wahr* und kann stets als Ziel angegeben werden, wenn ein Vergleichsergebnis nicht interessiert.

- Mehrere Vergleiche dürfen in derselben Instruktionsgruppe dasselbe Register ändern, wenn sie dies in derselben Weise oder gar nicht tun. Mit *.or* erweiterte Vergleiche setzen das erste Zielprädikat auf *wahr* und das zweite auf *falsch*, wenn der Test zutrifft, und führen sonst keinerlei Änderung durch. Es ist also egal, in welcher Reihenfolge ihre Änderungen wirksam werden. Bei vorher entsprechend initialisierten Prädikaten können so viele Vergleiche in einem Takt stattfinden. Solche parallelen Varianten existieren jedoch nur von einem Teil der Vergleichsoperationen.
- Als einzige Instruktionsart dürfen Sprungbefehle in derselben Instruktionsgruppe nach Setzen ihres Prädikats auftreten.

C-Fragment

```
if ( (a == 0) || (b == 3) || (c > 0) ) return;
```

IA-64-Entsprechung

```
cmp.eq p0,p1 = r0, r0          // p1 wird mit falsch initialisiert
;;                             // Stopp nötig
cmp.eq.or p1,p0 = r0, r33
cmp.eq.or p1,p0 = 3, r33
cmp.gt.or p1,p0 = r34, r0
(p1) br.ret.dpnt b0            // Bedingter Rücksprung an in b0 hinterlegte Adresse
```

Der Rücksprungbefehl trägt hier die Ergänzung *„.dpnt“*. Dies ist ein Hinweis an den Prozessor, statistische Informationen über das Sprungverhalten an dieser Stelle zu sammeln und für eine Vorhersage zu nutzen (dynamisch). Sind keine Informationen über das Sprungvorhalten vorhanden, soll bis zur endgültigen Auswertung davon ausgegangen werden, dass der Sprung nicht durchgeführt wird (not taken). Während der Befehlssatz es dem Prozessor freistellt, Hinweise aller Art (Hints) auch zu ignorieren, werden diese von den Itanium-Prozessoren beachtet.

2.2.4.2 Registerrotation

Ein entscheidendes Problem bei einer In-Order-Architektur stellt die Ausführung von Schleifen dar, wie am folgenden einfachen Beispiel, der Multiplikation eines Vektor mit einem Skalar, ersichtlich wird:

C-Fragment

```
#define N 100
int i;
double a[N];
double c;
int n = N;
/* hier Initialisierung */
for (i = 0; i<n; ++i) a[i] = a[i] * c;
```

Direkte IA-64-Umsetzung

```
                               // r32 enthält den Zeiger a, f8 c und r33 n
   add r34 = r0, r0            // Zähler auf 0 setzten
   br.cond.sptk .test
.loop:
   ldfd f9 = [ r32 ]           // Lade double von Adresse in r32
   ;;
   fmpy f9 = f9, f8            // Pseudo-Instruktion für fma f9 = f9, f8, f0
   ;;
   stfd [ r32 ] = f9, 8        // Speichern mit Postinkrement
.test:
   cmp.lt p1,p0 = r34, r33     // i < n ?
   add r34 = 1, r34            // Schleifenzähler kann bereits erhöht werden, da in
                               Schleife nicht benutzt
   (p1) br.cond.sptk .loop
```

Eine leichte Verbesserung bietet am Itanium die Verwendung eines besonderen Schleifenbefehls, der Sprünge abhängig vom Zähler in einem speziellen Anwendungsregister durchführt. Dadurch kann zudem eine optimale Sprungvorhersage getroffen werden.

Nutzung einfacher Schleifenunterstützung

```
    cmp.ge p1,p0 = 0, r33
    add r33 = -1, r33
    (p1) br.cond.spnt .exit     // Schleife überspringen, wenn n < 1
    ;;
    mov ar.lc = r33             // Loop Counter = n-1
.loop:
    ldfd f9 = [ r32 ] ;;
    fmpy f9 = f9, f8 ;;
    stfd [ r32 ] = f9, 8
    br.cloop.sptk .loop         // Springt bei ar.lc > 0, erniedrigt ihn dabei um 1
.exit:
```

Die so erreichbare Geschwindigkeit ist allerdings dennoch ernüchternd: Zur Berechnung eines neuen Wertes sind für Laden des Operanden im günstigsten Fall sechs Takte, für die Multiplikation vier und für Speicherung und den Rücksprung ein weiterer Takt einzuplanen.

Moderne Out-of-Order-Architekturen lösen dieses Problem, indem sie die Ladeoperationen mehrerer Schleifendurchgänge spekulativ ausführen und die Berechnungen bis zum Eintreffen der Operanden verzögern. Da in jeder Schleifeniteration dieselben Register adressiert werden, müssen ihnen jeweils unterschiedliche aus einem größeren Register File zugeordnet werden (Register Renaming). Alternativ oder unterstützend können Schleifen auch vom Compiler ausgerollt werden.

Die Itanium-Architektur stellt Instruktionen und Konzepte zur Verfügung, um Schleifen ähnlich der Fließbandverarbeitung im Prozessor über eine Software Pipeline abarbeiten zu können. Für das beschriebene Beispiel sollte eine vollständig aktivierte Software Pipeline wiederholt folgende Operationen durchführen:

Lade ein neues `a[n]`
Multipliziere bereits geladenes `a[n-m]`
Schreibe fertig berechnetes `a[n-m-o]`

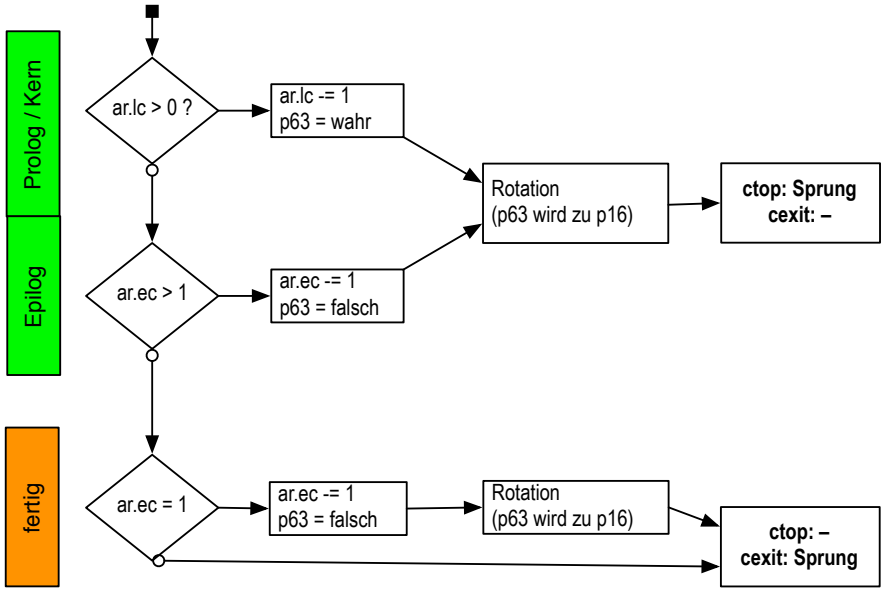
Neben der vollen Aktivierung (Kern) müssen die einzelnen Stufen der Pipeline auch kontrolliert be- (Prolog) und entladen (Epilog) werden. Werden nicht mehr Iterationen als Stufen durchgeführt, können Pro- und Epilog auch direkt ineinander übergehen oder überlappen.

Zur Realisierung von Software Pipelines bietet die Architektur ein explizites Register Renaming durch spezielle Sprungbefehle. Dabei wird die Bezeichnung eines Teils der Register geändert, so dass anschließend die Register n als Register $n+1$ und Register n_{max} als Register n_{min} angesprochen werden. Bei den Gleitpunktregistern sind dies f32 bis f127 und bei den Prädikatregistern p16 bis p63. Bei den Allzweckregistern können ab r32 in Schritten von acht zwischen keinem und 96 Registern für Rotation konfiguriert werden.

Während die Erzeugung einer Software Pipeline mit While-Schleifen weitere Konzepte benötigt (siehe hierzu auch 2.2.4.4), können Schleifen mit bekannter Iterationszahl leicht in Software Pipelines umgebaut werden. Hierfür werden neben dem oben erwähnten Anwendungsregister *ar.lc* noch der Epilogzähler *ar.ec* benötigt. *br.ctop* wird dabei am Ende der Schleife für den Rücksprung verwandt, *br.cexit* führt den Sprung unter entgegengesetzter Bedingung aus und wird v. a. beim Ausrollen von Schleifen genutzt (vgl. Abb. 2.3).

Durch Prädikate werden Instruktionen einer bestimmten Pipelinestufe zugeordnet, so dass Pro- und Epilog automatisch durchgeführt werden. Um die Pipeline beim ersten Durchlauf nicht vollständig inaktiv zu durchlaufen, wird die erste Pipelinestufe explizit aktiviert und der Schleifenzähler entsprechend um 1 niedriger initialisiert. Werden möglicherweise keine Iterationen durchgeführt, sollte dies vorher getestet und die Schleife übersprungen werden.

Abbildung 2.3: Funktionsweise von br.ctop und br.cexit



Pipelined Loop

```
// keine Allzweckregister für Rotation freigegeben
    add r2 = -1, r33              // n > 0 angenommen, vgl. oben
    mov pr.rot = 1 << 16          // die nicht-rotierenden Prädikate werden durch
                                  .rot maskiert und bleiben unverändert; p16 wird
                                  auf wahr, p17 bis p63 auf falsch gesetzt
    mov ar.ec = 11 ;;             // Anzahl Pipelinestufen
    add r33 = 0, r32              // Kopie von a zum verzögerten Schreiben
    mov ar.lc = r2 ;;             // Schleifeniterationen -1

.pipelinedloop:
    (p16) ldfd f32 = [ r32 ],8    // Pipelinestufe 1
    (p22) fmpy f38 = f38, f8      // Pipelinestufe 7
    (p26) stfd [ r33 ] = f42,8    // Pipelinestufe 11
    br.ctop.sptk .pipelinedloop
```

Da hier alle Instruktionen der Schleife innerhalb eines Taktes ausgeführt werden können, werden nach dem Laden entsprechend fünf und nach der Berechnung drei leere Stufen eingeplant, um die Latenzen (sechs Takte Laden aus L2-Cache, vier Takte für die Gleitpunktoperation) zu verbergen.

Eine optimierte Implementierung sollte aber vierfach ausgerollt werden. Zum einen bietet der Itanium 2 genügend Ressourcen zum Laden, Berechnen und Speichern von zwei Werten in einem Takt, zum anderen können bei nur einem Takt langen Schleifenkörpern die Sprünge nicht immer korrekt vorhergesagt werden (ein Takt Verzögerung bei jedem zweiten Durchlauf).

In einigen Fällen können auch explizite Pro- und Epiloge sinnvoll oder nötig sein. In 3.1 werden *Software Pipelined Loops* und deren Optimierung vertieft.

2.2.4.3 Prefetching und Speicherzugriff

Große und schnelle Caches sind ein Kernpunkt der Architektur. Über Prefetch-Instruktionen und zusätzliche Hinweise an praktisch allen Speicheroperationen wurde deshalb die Grundlage einer sehr genauen Kontrolle darüber gegeben, wann welche Daten auf welcher Speicherebene eingelagert werden.

So besteht zum Beispiel die Möglichkeit, Ganzzahl- und Gleitpunktdaten jeweils gezielt bis in den L1- oder L2-Cache, für besondere Anwendungen auch nur in den L3-Cache vorzuladen. Außerdem kann bei Prefetch-Instruktionen angegeben werden, ob man auch bei fehlenden TLB-Einträgen eine Auflösung der virtuellen Adresse wünscht.

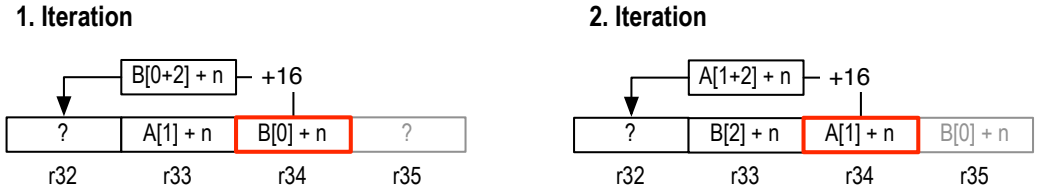
Auf allen Ebenen der Cachehierarchie sind auch Bereiche für „non temporal data“ vorgesehen, die nur kurzfristig benötigt wird. Die Itanium-Prozessoren realisieren dies, indem diese Daten zur baldigen Verdrängung freigegeben werden.

Eine übliche Strategie zum Vorladen mehrere Datenströme, die auch der Compiler beherrscht, verwendet dazu für Rotation konfigurierte Allzweckregister. Das folgenden Fragment könnte zur Vektoraddition `A[i] = A[i] + B[i]` gehören, wobei die Startadressen für A und B in r14 und r15 stehen. Pro Schleifeniteration bewegen sich die Datenströme jeweils um acht Byte. Das Prinzip wird noch einmal in Abbildung 2.4 verdeutlicht.

Prefetching mit rotierenden Registern

```
    add r34 = 900, r15          // Prefetche 900 Bytes vor den Datenströmen
    add r33 = 908, r14
[ ... ]
.loop:
[ ... ]
    lfetch.fault.nt1 [ r34 ]    // Vorladen der Daten bis in den L2-Cache, da sie
                                // für L1 non temporal sind
    add r32 = 16, r34
[ ... ]
    br.ctop.sptk .loop
```

Abbildung 2.4: Prefetching in Schleifen mit rotierenden Registern



Für Prefetch-Instruktionen und Integer-Ladeoperationen kann zudem angezeigt werden, dass die entsprechende Cachezeile bald verändert wird. Bei Mehrprozessorsystemen führt dies zu einer „Request for Ownership“. Die Cachezeile wird außerdem sofort als modifiziert markiert und muss bei der Verdrängung aus der Cachehierarchie auch ohne tatsächliche Änderung zurückgeschrieben werden. Diese Funktion sollte deshalb nur mit Bedacht angewandt werden.

Obwohl alle Itanium-Prozessoren Prefetch-Instruktionen und Zusätze an Ladebefehlen sehr zuverlässig beachten, werden diese vom Sprachstandard als Hints definiert, also ignorierbare Hinweise.

Auf den Itaniums findet implizites Prefetching auch bei allen Postinkrementen statt, wobei die Zeile eingelagert wird, die die nach dem Postinkrement spezifizierte Adresse enthält.

2.2.4.4 Explizite Spekulation

Spekulative Ausführung ist – von spekulativen Sprüngen abgesehen – üblicherweise nur für Out-of-Order-Architekturen bekannt, wo es sich als effektive Möglichkeit zum Verbergen von Latenzen bewährt hat.

Der IA-64-Befehlssatz bietet deshalb Instruktionen, mit denen auch auf der In-Order-Architekur explizite Spekulation möglich ist. Durch Hintergrundwissen können Compiler oder Programmierer im Idealfall sogar besser entscheiden, welche Instruktionen wahrscheinlich vorgezogen und somit sinnvoll spekulativ ausgeführt werden können.

Unterschieden wird dabei zwischen Data und Control Speculation: Das Konzept der Data Speculation findet insbesondere Anwendung, wenn Aliasing unwahrscheinlich ist, aber nicht ausgeschlossen werden kann. Control Speculation erlaubt das Verzögern von Ausnahmen, so dass bedingte Ladeoperationen vor Auswertung ihrer Bedingung ausgeführt werden können.

Data Speculation

Bei der folgenden Beispielfunktion darf ein C-Compiler (im Gegensatz zu Fortran) nach Sprachdefinition nicht annehmen, dass die als Parameter übergebenen Zeiger auf verschiedene Variablen verweisen:

Beispielsfunktion in C

```
void aliasing(double* a, double*) {
    *a = *a + 1.0;
    *b = *b + *b;
}
```

Ohne weitere Unterstützung durch die Hardware müssen die Rechenoperationen also defensiv nacheinander ausgeführt werden:

IA-64-Umsetzung ohne Spekulation

```
ldfd f8 = [ r32 ] ;;        // min. 6 Takte Latenz Laden
fadd f8 = f8, f1 ;;         // 4 Takte Latenz
stfd [ r32 ] = f8
ldfd f9 = [ r33 ] ;;        // min. 6 Takte Latenz Laden
fadd f9 = f9, f9 ;;         // 4 Takte Latenz
stfd [ r33 ] = f9           // 1 Takt
```

Durch zwei Verfahren können entweder nur der Ladevorgang oder auch zusätzliche Berechnungen auf Verdacht vorgezogen werden. Die Idee dabei ist es, durch einen speziellen Ladebefehl den betroffenen Speicherbereich überwachen zu lassen. Später lässt sich so überprüfen, ob der geladene Wert noch dem aktuellen Speicherzustand entspricht.

Wurden noch keine Berechnungen mit dem geladenen Wert durchgeführt, bietet sich ein spezieller Ladebefehl an, der einen Speicherzugriff nur bei zwischenzeitlicher Änderung durchführt (Load Check). Wurden bereits Berechnungen mit dem möglicherweise veralteten Wert durchgeführt, kann dies über eine Check-Operation überprüft und bei Bedarf zu einem Korrektur-Code gesprungen werden.

Der Prozessor macht hierfür spezielle Einträge in die Advanced Load Address Table, die jedoch nur eine begrenzte Anzahl aufnehmen kann . Ein fehlender Eintrag in der ALAT und somit fehlgeschlagener Check ist nach Sprachdefinition bei einer Änderung im Speicher deshalb zutreffend, aber nicht hinreichend dafür. Ein Hinweis an Load Checks und Advanced Checks kann das anschließende Löschen (*.clr*) oder Behalten des ALAT-Eintrags (*.nc*) für weitere Überprüfungen erbitten.

Vorziehen des Ladens durch Load Check

```
ldfd f8 = [ r32 ]
ldfd.a f9 = [ r33 ]            // Advanced Load
;;
fadd f8 = f8, f1 ;;
stfd [ r32 ] = f8
ldfd.c.clr f9 = [ r33 ] ;;     // Load Check: keine Aktion, wenn Wert noch aktuell
fadd f9 = f9, f9 ;;
stfd [ r33 ] = f9
```

Vorziehen der Berechnung mit Advanced Check

```
   ldfd f8 = [ r32 ]
   ldfd.a f9 = [ r33 ]          // Advanced Load
   ;;
   fadd f8 = f8, f1
   fadd f9 = f9, f9
   ;;
   stfd [ r32 ] = f8
   chk.a.clr f33, .recover      // Sprung zum Label .recover, wenn die Berechnung
                                // auf einem veralteten Wert basiert
   stfd [ r33 ] = f9
.back:
[...]

.recover:
   ldfd f9 = [ r33 ] ;;
   fadd f9 = f9, f9 ;;
   stfd [ r33 ] = f9
   br.cond.sptk .back           // Programm wieder fortsetzen
```

Control Speculation

Bei bedingter Ausführung muss mit dem Laden von Operanden üblicherweise so lange gewartet werden, bis die Bedingung ausgewertet ist. Im folgenden Beispiel kann *a* auch einen ungültigen Zeiger enthalten.

Bedingte Ausführung in C

```
/* a und b sind Zeiger vom Typ int* */
if (*b!=0) {
   *a += 3;
}
```

Durch Verwendung einer spekulativen Ladeoperation werden auftretende Ausnahmen verzögert. Bei den Allzweckregistern wird das zusätzliche Not-a-Thing-Bit (siehe 2.2.2) gesetzt und die Art der Ausnahme im Register kodiert, bei den Gleitkommaregistern wird ein reservierter Not-a-Thing-Wert verwendet.

Ist die Bedingung dann ausgewertet, kann mit einer Speculation-Check-Operation, analog zur Data Speculation, festgestellt werden, ob das Register einen gültigen Wert enthält, oder ob ein Sprung zu einer Sonderbehandlung der Ausnahme ausgeführt werden soll. Jeder Versuch, ein Register mit einer verzögerten Ausnahme mit anderen Instruktionen zu lesen, führt zu einer weiteren, speziellen Ausnahme, aufgrund der die meisten Betriebssysteme die Anwendung beenden.

Control Speculation bei IA-64

```
ld4.s r8 = [ r32 ]
ld4 r9 = [ r33 ]
;;
cmp.eq p1,p0 = r0, r9
;;
(p1) chk.s r8, .recover
(p1) add r8 = 3, r8
;;
(p1) st4 [ r32 ] = r8
```

Bei Software Pipelines für While-Schleifen werden häufig einige spekulative Stufen benötigt, wenn die Schleifenbedingung nur mit Verzögerung berechnet werden kann. Es existieren auch Ladeoperationen, die sowohl daten- als auch kontroll-spekulativ sind.

Kapitel 3

Optimierungstechniken

3.1 Schleifenoptimierungen

3.1.1 Ausrollen von Schleifen

Auch auf der IA-64-Architektur kann das Ausrollen von Schleifen eine wirkungsvolle Strategie zur Optimierung sein. Dies gilt insbesondere, wenn

- der Schleifenkörper innerhalb eines Taktes ausgeführt wird, und die Sprungvorhersage deshalb nicht mehr korrekt arbeiten kann,
- durch parallele Ladeoperationen die Bandbreite bei gleichzeitiger Nutzung weniger Instruktions-Slots erhöht werden kann,
- die ausgerollte Version schneller ausgeführt werden kann, weil ein besseres Bundling möglich ist, oder so häufiger zwei Bundles pro Takt ausgeführt werden können, oder
- die Anzahl von Prefetch-Instruktionen dadurch sinnvoll reduziert werden kann.

Software Pipelines können im IA-64-Befehlssatz so geschrieben werden, dass im Anschluss kein Aufroll-Code mehr nötig ist, um übrig gebliebene Einzeliterationen durchzuführen. Bei der einfachsten Variante werden mehrere Versionen der Schleife mit denselben Operationen auf jeweils verschiedene Registern genutzt, zwischen denen eine Rotation oder das vorzeitige Verlassen der Schleife durch *br.cexit* erreicht wird. Pro Durchlauf des Schleifenkörpers werden damit entsprechend mehr Rotationen durchgeführt, so dass bei n-fachem Ausrollen nur jede n-te Stufe genutzt werden kann.

Unrolled Pipelined Loop für skalare Vektormultiplikation

```
    add r2 = -1, r33              // n > 0 angenommen
    mov pr.rot = 1 << 16          // Stufe 1 aktivieren
    mov ar.ec = 11 ;;             // Anzahl Pipelinestufen
    add r33 = 0, r32              // Kopie v. Pointer a zum verzögerten Schreiben
    mov ar.lc = r2 ;;             // Schleifeniterationen -1

.pipelinedloop:
    (p16) ldfd f32 = [ r32 ],8
    (p22) fmpy f38 = f38, f8
    (p26) stfd [ r33 ] = f42,8
    br.cexit.spnt .exit ;;
    (p16) ldfd f62 = [ r32 ],8
    (p22) fmpy f68 = f68, f8
    (p26) stfd [ r33 ] = f62,8
    br.ctop.sptk .pipelinedloop
.exit:
```

Unter Beachtung der Latenzen und mit entsprechender Anpassung der Pipelinestufe können Instruktionen auch über *br.cexit*-Anweisungen hinweg bewegt werden, um zum Beispiel ein dichteres Bundling zu erreichen.

Bei Instruktionen auf der letzten Pipelinestufe führt ein Verlegen über die nächste *br.cexit*-Anweisung allerdings zu einer effektiven Verlängerung der Pipeline, der mit einer Erhöhung von ar.ec begegnet werden kann. Bei zweifachem Ausrollen ist auch eine trickreiche Nutzung von *br.cexit* denkbar, indem hier nicht aus der Schleife, sondern zum nächsten Bundle gesprungen wird. *br.cexit* wird dabei also nur genutzt, um bei Bedarf eine weitere Rotation zu erzwingen. Dies gelingt jedoch nur einmal bei $ar.ec = 1$.

Im Gegensatz zu Compilern können handoptimierte Unterprogramme strengere Bedingungen an die übergebenen Parameter stellen und somit Überprüfung und Behandlung von Ausnahmefällen sparen. Um die skalare Vektormultiplikation auch aus dem L2-Cache mit höchster Geschwindigkeit ausführen zu können, muss eine Schleife vierfach ausgerollt werden. Eine effiziente Lösung kann unter folgenden Annahmen hergestellt werden:

- Die Größe des Feldes n ist größer 0, und
- $A[0]$ ist an einer 16-Byte-Grenze ausgerichtet, so dass parallele Ladeoperationen verwendet werden können.
- Ist n nicht ganzzahlig durch vier teilbar, so können dennoch die folgenden $4 - (n \ mod \ 4)$-Werte (also bis zur nächsten Vierer-Grenze) gelesen werden. Bei Speicherschutz auf Seitenbasis ist dies z. B. immer der Fall, andernfalls kann man den Arrays nicht genutzte Füllwerte hintanstellen.

Die Schleife wird dann vierfach ausgerollt, um mindestens zwei Takte für den Schleifendurchlauf zu benötigen. Durch die Verwendung von zwei parallelen Ladeoperationen werden zum einen Bankkonflikte im L2-Cache vermieden, zum anderen sind nur dann ausreichend Slots für Prefetching und den Rücksprung in den vier Bundles verfügbar. Die Speicheroperationen werden so verteilt, dass zwischen ihnen und den Ladeoperationen keine Bankkonflikte auftreten können. Die Ausführung der letzte Pipelinestufe im Epilog wird ausgelagert, um nicht nur Vektoren der Länge $n \cdot 4$ bearbeiten zu können. Der entsprechende Programmcode ist mit vollständigen Bundling-Informationen in Programmliste 1 zu finden.

3.1.2 Mehrfachnutzung von Werten

Oft werden dieselben Werte für mehrere Berechnungen kurz hintereinander benötigt. Ein einfaches Beispiel bietet die eindimensionale lineare Interpolation $B[n] = (A[n-1] + A[n+1])/2.0$, bei der der Wert $A[n]$ zu $B[n-1]$ und zu $B[n+1]$ beiträgt.

Dies kann auf dem Itanium ohne Umkopieren oder Ausrollen der Schleife zur Reduzierung von Ladeoperationen genutzt werden. Mithilfe der rotierenden Register haben Operationen Zugriff auf Registerinhalte, die anderen Iterationen zuzuordnen sind.

Lineare Interpolation mit Mehrfachnutzung ohne Ausrollen

```
// r32 Zeiger auf A[n-1]
// r33 Zeiger auf B[n]
// f8 = 0.5
[ ... Initialisierung ... ]
    ldfd f34 = [ r32 ], 8
    ldfd f33 = [ r32 ], 8
    ;;
.loop:
    (p16) ldfd f32 = [ r32 ],8
    (p19) fadd f50 = f35, f37       // A[n-1] + A[n+1]
    (p21) fmpy f52 = f52, f8
    (p23) stfd [ r33 ] = f54, 8
    br.ctop.sptk .loop
```

Programmliste 1 Skalare Vektormultiplikation mit maximalen Durchsatz

```
{ .mmi
          add r3 = -1,r33
          add r37 = 4000, r32
          extr.u r2 = r33,0,2 // Entspricht Modulo-Teilung durch 4      }
{ .mmi
          add r33 = 0, r32
          add r34 = 8, r32
          mov pr.rot = 1 << 16 ;;                                       }
{ .mmi
          cmp.lt p6 = 1, r2
          cmp.lt p7 = 2, r2
          shr r3 = r3, 2 // Entspricht Ganzzahldivision durch 4         }
{ .mmi
          cmp.lt p8 = 3, r2
          add r35 = 16, r32
          nop.i 0 ;;                                                    }
{ .mmi
          add r36 = 24, r32
          cmp.eq.or p6 = r0, r2
          mov ar.lc = r3                                                }
{ .mmi
          cmp.eq.or p7 = r0, r2
          cmp.eq.or p8 = r0, r2
          mov ar.ec = 7 ;; // Pipelinestufen -1                         }

.loop:
{ .mmf
          (p16) lfetch.nt1 [ r37 ], 32
          (p16) ldfpd f32, f45 = [ r32 ], 16
          (p20) fmpy f36 = f36, f8                                      }
{ .mmf
          (p23) stfd [ r33 ] = f39, 32
          (p23) stfd [ r36 ] = f82, 32
          (p20) fmpy f49 = f49, f8 ;;                                   }
{ .mmf
          (p23) stfd [ r34 ] = f52, 32
          (p23) stfd [ r35 ] = f69, 32
          (p20) fmpy f66 = f66, f8                                      }
{ .mfb
          (p16) ldfpd f62, f75 = [ r32 ], 16
          (p20) fmpy f79 = f79, f8
          br.ctop.sptk .loop ;;                                         }

{. mmi
          stfd [ r33 ] = f39
          (p6) stfd [ r34 ] = f52
          nop.i 0                                                       }
{ .mmi
          (p7) stfd [ r35 ] = f69
          (p8) stfd [ r36 ] = f82
          nop.i 0                                                       }
```

Für die ersten Iterationen müssen dafür in einer Erweiterung des Prologs die benötigten Teile des Zustands hergestellt werden, den vorhergehende Iterationen hinterlassen hätten.

3.1.3 Überlappung von Schleifen

Trotz seiner Eleganz ist die Nutzung von Software Pipelines nicht sinnvoll, wenn nur wenige Iterationen damit ausgeführt werden:

Sind die Operationen eines Schleifenkörpers auf verschiedene Pipelinestufen verteilt, kann unter Vernachlässigung von Latenzen beim Zugriff auf niedrigere Speicherebenen davon ausgegangen werden, dass jeder Durchlauf des Schleifenkörpers, auch während Pro- und Epilog, dieselbe Zeit t benötigt.

In den ersten der Anzahl Iterationen i entsprechend vielen Durchgängen, wird die erste Pipelinestufe jeweils neu beladen. Danach schließen sich weitere e Durchläufe für den Epilog an, in denen keine neuen Berechnungen mehr gestartet und die Pipeline abgeladen wird, wobei e eins kleiner ist als die Anzahl an Pipelinestufen. Die gesamte Ausführungdauer lässt sich nun in den linear abhängigen Anteil $i \cdot t$ und den konstanten Teil $e \cdot t$ zerlegen.

Bei großer Iterationszahl i hat der konstante Mehraufwand der Software Pipeline einen geringen Anteil; die Verarbeitungszeit nähert sich dem Optimum $i \cdot t$ an und macht dieses Verfahren durch einfache Umsetzung und kleine Codegröße zum Mittel der Wahl. Mit zunehmender Pipeline-Länge und abnehmender Iterationszahl überwiegt der Aufwand zum Be- und Entladen der Pipeline und macht dieses Verfahren zunehmend unattraktiver.

Ist eine niedrige Iterationszahl bekannt, kann die Schleife vollständig ausgerollt werden. Zum Teil können auch explizite Pro- oder Epiloge mit anderen Berechnungen vermischt werden.

3.1.3.1 Überlappung ohne Ausrollen

Wird dieselbe Schleife mehrfach hintereinander ausgeführt, zum Beispiel bei verschachtelten Schleifen, können Epi- und Prologe nach einigen Anpassungen häufig effizient überlappt werden.

Der Ansatz ist hierbei, durch Setzen des Epilogzählers auf 1 die Schleife zunächst vor Ausführung des Epilogs zu verlassen. Sind keine weiteren Schleifen derselben Art mehr zu überlappen, wird durch Setzen des Schleifenzählers auf 0 und des Epilogzähler auf den benötigten Wert bei erneuter Rückkehr der Epilog alleine abgeschlossen.

Sind noch weitere Schleifen mit demselben Schleifenkörper auszuführen, werden die Operanden für die erste Pipelinestufe angepasst, der Schleifenzähler entsprechend gesetzt und die erste Pipelinestufe aktiviert. Nach erneutem Setzen des Epilogzählers auf 1 kann die Ausführung wieder aufgenommen werden. Der Prolog der neuen und der Epilog der vorhergehenden Schleife werden dann gleichzeitig ausgeführt.

Diese Art der Verschmelzung funktioniert ohne Anpassungen, wenn Operanden ab der zweiten Stufe entweder über alle Schleifen invariant sind (z. B. Konstanten), von diesen Stufen exklusiv genutzt werden oder auf den rotierenden Registern von Stufe zu Stufe weitergereicht werden, also nicht verzögert oder mehrfach genutzt werden.

Des weiteren muss sichergestellt sein, dass im Prolog keine Werte aus dem Speicher geladen werden, die ein Epilog aufgrund der Überlappung noch nicht ausgeschrieben hat.

Diese strengen Anforderungen erfüllt zum Beispiel die Suche nach Maxima in verschiedenen Zeilen eines mehrdimensionalen Feldes, viele andere Anwendungen jedoch nicht. Ein allgemeines Problem ist die verzögerte Änderung von Operanden wie der Zieladressen von Schreibströmen, die üblicherweise auf der letzten Pipelinestufe nötig wäre. Auch die Mehrfachnutzung von Werten kann mit Einschränkungen entsprechend angepasst werden.

3.1.3.2 Verzögerte Operationen

Um Änderungen, die zwischen zwei Schleifen durchgeführt werden müssten, erst später wirksam werden zu lassen, gibt es je nach Verfügbarkeit von Registern oder freien Slots innerhalb der Schleife grundsätzlich zwei Möglichkeiten.

Zum einen können Änderungen zwischen zwei Schleifen durchgeführt und mit Hilfe der rotierenden Register verzögert werden. Zwischen zwei überlappten Schleifen sind Operationen der ersten Pipelinestufe zuzuordnen, und ihre Resultate wandern mit dieser Iteration durch die Pipeline.

Die Adressen von Schreibströmen können, statt mit Postinkrementen auf statischen Registern, dazu auf den rotierenden Registern erzeugt werden, ähnlich der Adress-Generierung beim Prefetching. Die auf der ersten Pipelinestufe erzeugten Adressen wandern dann mit den zugehörigen Operanden durch die Pipeline. Das folgende Beispiel zeigt, mit wenigen Teilen in Pseudo-Code, wie die skalare Vektormultiplikation mehrerer getrennter Vektoren effektiv verschmolzen werden kann.

Überlappte skalare Vektormultiplikation

```
[ ... Initialisierung ...]
// r33 enthält Startadresse des ersten Vektors
    mov ar.ec = 1
    cmp.ne p6 = r0, r0              // Epilog noch nicht ausgeführt

.loop:
    (p16) ldfd f32 = [ r33 ]
    (p16) add r32 = 8, r33
    (p21) fmpy f35 = f35, f8
    (p23) stfd [ r40 ] = f37
    br.cloop .loop
    (p6) br.cond.sptk .exit         // p6 ist erst nach dem letzten Epilog wahr

if ( weitere Berechnungen )
    add r33 = 0,nächste Startadresse
    mov ar.lc = Anzahl Iterationen −1
    mov ar.ec = 1
    cmp.eq p16, p0 = r0, r0         // Stufe 1 aktivieren
else
    mov ar.ec = 4                   // Stufen−1
    cmp.eq p6, p0 = r0, r0          // nur noch Epilog
fi

    br.cond.sptk .loop
.exit:
```

Bei Software Pipelines, die weniger als die maximal 48 Pipelinestufen und ausreichend freie Slots im Schleifenkörper besitzen, können auch die freien Prädikatregister genutzt werden, um hierüber eine verspätete Ausführung zu veranlassen.

Dabei setzt ein Vergleich im Schleifenkörper einen rotierenden Bereich auf *falsch*. Wird zwischen zwei überlappten Schleifen der Wert auf *wahr* geändert, kann er beim Wandern durch die Pipeline zum Auslösen von Änderungen genutzt werden. Dafür sind zumindest zwei Slots für Löschen des Prädikaten-Bereichs und einen bedingten Sprung nötig. Bei mehrdimensionalen Feldern, die Aufgrund von Padding oder enthaltenen Randwerten nicht in großen Schleifen durchlaufen werden können, ist so leicht ein Sprung zur nächsten Zeile durch Addition eines konstanten Wertes realisierbar.

Über ein verwandtes Verfahren können auch Schleifen mit ausgelagerter Epilogstufe behandelt werden (vgl. 3.1), solange im ausgelagerten Epilog dieselben Operationen wie im Schleifenkörper, aber unter anderen Bedingungen ausgeführt werden:

Bei jedem Durchgang werden durch Kopieren von p16 ein oder mehrere zunächst identische Pipeline-

Steuerungen geschaffen. Die Instruktionen der letzten Pipelinestufe werden dann abhängig von den zunächst identischen Prädikat-Bereichen ausgeführt. Wird der Schleifenkörper zwischen zwei zu überlappten Schleifen verlassen, können diese kopierten Prädikate verändert, und so das Verhalten der wieder eingegliederten Operationen genau gesteuert werden.

3.1.4 Überlappung bei Mehrfachnutzung

Bei der Mehrfachnutzung von Operanden, wie sie bei der eindimensionalen Interpolation in 3.1.2 durchgeführt wurde, greift jede Schleifeniteration auf Operanden vorhergehenden Iteration zu, während in einer Erweiterung des Prologs die für die ersten Iteration benötigten Operanden hergestellt werden.

Können diese kritischen Zugriff auf der ersten Pipelinestufe stattfinden, ist eine Verschmelzung durch Vorladen der Werte zwischen zwei Schleifen weiterhin möglich. Allerdings können so nur bei längeren Schleifenkörpern Speicherlatenzen noch verborgen werden.

Andernfalls können Epi- und Prologe nur teilweise überlappt werden, indem man den Epilogzähler auch zwischen zwei Schleifen größer als 1 setzt. Dabei wird eine „Blase“ in die Software Pipeline eingeführt, solange der Epilog noch Werte aus vorhergehenden Iterationen benötigt. Je nach Aufbau der Schleifen ist hier aber zu bedenken, dass auch Register für Prefetching oder mit Speicheroperanden weiter rotieren.

Wird nur auf Werte der vorhergehenden Iteration zugegriffen, und lässt die Länge der Pipeline ausreichend rotierende Prädikatregister frei bietet sich eine effektivere Lösung: Für Operationen, die auf die problematischen Operanden zugreifen, wird wie im letzten Abschnitt beschrieben eine Kopie der Pipeline-Prädikate erstellt. Nachdem diese Berechnungen bei der ersten Iteration eines neuen Prologs mit falschen Operanden arbeiten würden, wird dies durch Setzen des entsprechenden Prädikats auf *falsch* verhindert. Stattdessen werden die Berechnung mit expliziten Anweisungen vor Wiederaufnahme der Schleifenausführung vorweggenommen, solange die nötigen Operanden noch vorhanden sind.

3.1.5 Vorausschauendes Prefetching

Nachdem Zugriffe auf den Hauptspeicher Latenzen bis zu Hunderten von Takten besitzen, kommt dem gezielten Vorladen von Werten auf dem Itanium eine große Bedeutung zu.

Das bereits in 2.2.4.3 beschriebene Verfahren besticht insbesondere, weil es auch von Compilern leicht angewandt werden kann, und wegen des niedrigen Bedarfs an Prozessorressourcen und Instruktions-Slots.

Die Annahme dabei ist, dass die Schleife einen langen Datenstrom erzeugt. Durch Abschätzung der Dauer einer Iteration, dem dabei vom Datenstrom überstrichenen Adressbereich und der Latenz des Hauptspeichers werden Daten in konstantem Abstand vor den Berechnung angefordert. Operanden- und Vorlade-Bereiche sind also leicht verschoben. (Vgl. hierzu Abb. 3.1)

Bei langen Datenströmen überlappen sich diese Bereiche fast vollständig und die Vorteile des Verfahrens überwiegen. Je weniger Schleifeniterationen allerdings ausgeführt werden, desto geringer wird die Überlappung im Verhältnis.

Dies ist wiederum unbedenklich, wenn die vorgeladenen Daten von direkt folgenden Berechnung zum großen Teil genutzt werden, z. B. bei Operationen auf einem mehrdimensionalen Feld mit geringem Padding oder Randwerten. Andernfalls können durch diese Art des Vorladens sogar ebenso viele unbenötigte wie benötigte Daten vorgeladen, und die Ausführung damit deutlich gebremst werden.

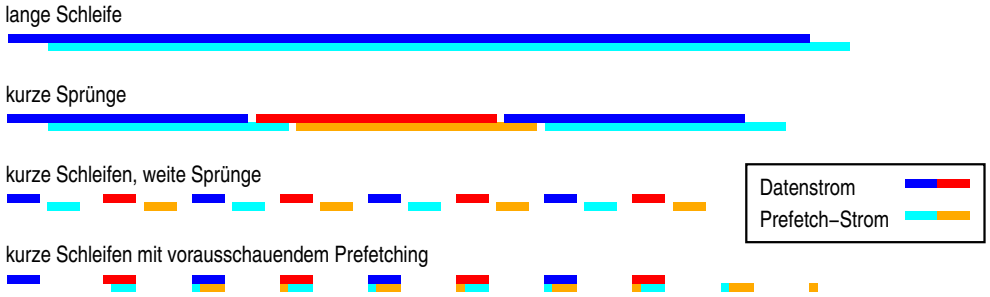
Ist das Zugriffsmuster aufeinander folgender Schleifen gleichmäßig und bekannt, kann dies jedoch zur Anpassung der Prefetch-Ströme genutzt werden. Die Startadresse der Prefetch-Ströme kann dann dort gewählt werden, wo für die Berechnung nach Ablauf der Speicherlatenz (und eines Sicherheitsbereichs) tatsächlich Daten benötigt werden. Diese Adresse kann dann auch im Operandenbereich einer folgenden Schleife liegen.

Außerdem kann bestimmt werden, wann der Datenstrom den Operandenbereich verlässt und wie weit er zum Beginn des nächsten Operandenbereichs springen muss. Besonders einfach ist dies, wenn die Schleifen ähnlich oder am besten gleich lang sind, so dass pro Schleifendurchlauf immer nur ein Sprung nötig ist.

Innerhalb der Schleife bietet sich dann ein Zähler an, der bei einem vorher berechneten Wert die Erhöhung der zum Prefetchen genutzten Register veranlasst, denn Vergleiche mit dem Schleifenzähler *ar.lc* sind nicht direkt möglich.

Idealisiert betrachtet werden dann nur der Prefetch-Entfernung entsprechend viele Daten nach dem letzten Operandenbereich unnötig vorgeladen, wie dies auch bei einer einzigen langen Schleife der Fall gewesen wäre. In der Praxis liegen die Grenzen der Operanden-Bereiche nicht mit den Grenzen von Cachezeilen übereinander, wodurch weitere Daten in die Caches eingelagert werden. Dies ist jedoch keine Besonderheit dieser Optimierungstechnik.

Abbildung 3.1: Prefetching bei verschiedenen Zugriffsmustern



3.2 Optimierung des Mehrgitterlösers

Ein guter Ausgangspunkt zur Optimierung eines Algorithmus kann durch Profiling bereits bestehender Implementierungen oder anhand theoretischer Überlegungen gefunden werden: Was sind die aufwendigsten Bestandteile, welche besitzen das größte Optimierungspotential?

In einem zweiten Schritt sollten dann globale Aspekte betrachtet werden: Dies sind zum einen Design-Fragen, die alle Komponenten betreffen. Zum anderen ergeben sich, zum Beispiel durch strengere Bedingungen an die Eingabedaten oder Parameter, weitere Optimierungsmöglichkeiten am Algorithmus selbst.

Schließlich sollte der Algorithmus auf einzelnen Komponenten aufgeteilt, jeweils eine effiziente Implementierung geplant und nach lokalen Optimierungsmöglichkeiten gesucht werden.

3.2.1 Charakterisierung

Um die einzelnen Bestandteile des Mehrgitterlösers zu charakterisieren, eignen sich sowohl die Anzahl der Rechenoperationen als auch der Speicherdurchsatz. Durch einige Vereinfachungen können hierfür griffige Formeln gefunden werden.

Es wird im folgenden angenommen, dass die Vergröberung eines Gitters $\frac{1}{8}$ der n^3 Unbekannten besitzt, sowie der Speicherbedarf der Randwerte vernachlässigt. Der Speicherdurchsatz wird allein für den Hauptspeicher betrachtet, und davon ausgegangen, dass jeder Wert pro Komponente nur einmal gelesen oder geschrieben werden muss, beim Glätter jeweils pro Iteration. Unter diesen Annahmen ist der Speichertransfer und die Anzahl an Rechenoperationen direkt proportional zur Anzahl der berechneten Unbekannten, sowohl bezogen auf eine oder alle Ebenen (mit Ausnahme des direkten Lösungsschritts bei einer Unbekannten).

Tatsächlich wird der Speicherdurchsatz für die meisten Bestandteile der bestimmende Faktor sein. Zur Übertragung der genutzten Gleitpunktzahlen doppelter Genauigkeit werden bei derzeit gängigen Itanium 2-Systemen ca. drei bis fünf Prozessortakte benötigt.

In Tabelle 3.1 sind die Operationen und nötigen Speicherzugriffe für ein Gitter im Verhältnis zur Anzahl Unbekannter angegeben. Dabei wurden auch die durch Write Misses implizit ausgeführten Ladeoperationen mitberechnet. Eine gängige Optimierung ist die Zusammenlegung der Berechnung des Residuums mit der Restriktion und die sofortige Korrektur durch den interpolierten Fehler; weil dann keine temporären Felder benötigt werden, kann der Bedarf an Speicherplatz und -durchsatz deutlich gesenkt werden.

Tabelle 3.1: Abschätzung Flops und Speicherzugriffe pro Unbekannter auf einer Gitterebene

	Flops		**Speicherdurchsatz**	
Komponente	einfach	Itanium	Lesen	Schreiben
Glätter [1]	$(i+j)\cdot 8$	$(i+j)\cdot 7$	$2\cdot(i+j)$ [2]	$(i+j)$ [3]
Residuum	9	8	3	1
Restriktion	$\frac{30}{8}$	$\frac{27}{8}$	$1+\frac{1}{8}$	$\frac{1}{8}$
Interpolation	$\frac{52}{8}$	$\frac{52}{8}$	$1+\frac{1}{8}$	1
Korrektur	1	1	2	1
Residualnormen [4]	11+?	10	2	0
Residuum & Restriktion	$9+\frac{30}{8}$	$8+\frac{27}{8}$	$2+\frac{1}{8}$	$\frac{1}{8}$
Interpolation & Korrektur	$\frac{52}{8}+1$	$\frac{52}{8}$	$1+\frac{1}{8}$	1

[1] i Vorglättungs- und j Nachglättungs-Iterationen
[2] zwischen 4 und $4\cdot(i+j)$ für $i>0$ und $j>0$
[3] zwischen 2 und $2\cdot(i+j)$ für $i>0$ und $j>0$
[4] Berechnung nur auf feinstem Gitter sinnvoll und nur von einer Norm nötig

Obwohl die Anzahl der Rechenoperationen mit jeder Vergröberung etwas stärker und der Speicherbedarf etwas weniger als auf $\frac{1}{8}$ abnimmt, gibt die geometrischen Reihe $\sum_{k=0}^{\infty}\left(\frac{1}{8}\right)^k=\frac{8}{7}$ eine gute Abschätzung für die Verteilung auf die verschiedenen Gitterebenen: Etwa $87,5$ % der Berechnungen und des Speicherverkehrs entfallen auf das feinste Gitter.

Für einen V-Zyklus lässt sich zudem auch die Verteilung auf die einzelnen Komponenten berechnen. Annahme der Berechnungen in Tabelle 3.2 ist es, dass je zwei Vor- und Nachglättungsschritte ausgeführt und nach jedem V-Zyklus beide Residualnormen berechnet werden. Außerdem werden lesende und schreibende Speicherzugriffe gleich gewichtet.

Tabelle 3.2: Anteil der Komponenten bei einem 2,2-V-Zyklus

Komponente	**Flops** (Itanium)	**Speicher-zugriffe**
Glätter	51 %	66 %
Residua & Restriktion	21 %	12 %
Interpolation & Korrektur	12 %	12 %
Residualnormen	16 %	10 %
Anzahl pro Unbekannter	ca. 62,5	ca. 20,7

Bereits bei einem V-Zyklus mit zwei Vor- und Nachglättungsschritten dominiert dabei der Glätter. Die Berechnung der Normen – die Berechnung der Residua selbst ist dabei am aufwendigsten – ist außerdem auffällig „teuer“, obwohl sie nur zur Bestimmung der Konvergenz und zum Abbruch der V-Zyklen genutzt wird.

3.2.2 Algorithmische Optimierung

Das Red-Black-Gauss-Seidel-Verfahren, das hier als Glätter genutzt wird, setzt während einer Iteration zunächst bei allen roten, dann bei allen schwarzen Unbekannten den neuen Wert so, dass das Residuum dort gleich 0 ist. Wird mindestens eine Iteration zur Vorglättung genutzt, werden an allen schwarzen Punkten Residua berechnet und restringiert, von denen bereits bekannt ist, dass sie im Rahmen der Rechengenauigkeit gleich 0 sind. Die Berechnung von Residuum und Restriktion können sich demnach auf die roten Punkte im groben Gitter beschränken.

Beim eingesetzten Glätter hängt zudem jedes neue Ergebnis ausschließlich von den umgebenden Punkten und der rechten Seite der Gleichung ab, jedoch nicht vom vorherigen Wert. Wird mindestens ein Nachglättungsschritt ausgeführt, können sich Interpolation und Korrektur deshalb auf die schwarzen Punkte des feinen Gitters beschränken.

3.2.3 Speicherlayout

Während die Einsparung von Rechenoperationen eine gute Voraussetzung für höhere Ausführungsgeschwindigkeit ist, bildet der Hauptspeicherdurchsatz die vermutlich niedrigere Hürde. Bei einem üblichen Speicherlayout liegen rote und schwarze Punkte jedoch untrennbar vermischt in einer Cachezeile und können deshalb getrennt weder gelesen noch geschrieben werden. Die algorithmischen Optimierungen des letzten Abschnitt können somit nicht voll genutzt werden.

Einen guten Lösungsansatz bietet das in [Stü05] vorgeschlagene Speicherlayout mit einer im dortigen Kapitel 5.1.2 vorgeschlagenen Änderung.

Anstatt die Gitter direkt in mehrdimensionale Felder zu übertragen, werden die Punkte nach Farbe getrennt in sogenannten Halbfeldern abgespeichert, wobei die Farb-Definition auch auf Randwerte verallgemeinert wird. Die erste Unbekannte jeder einfarbigen Halbzeile wird bei den verwendeten Gleitpunktzahlen doppelter Genauigkeit an einer 16-Byte-Grenze ausgerichtet, und alle Zeilen auf eine gleiche, gerade Länge gepaddet.

Randwerte werden effektiv an das Ende der vorhergehenden Zeile verschoben. Der Gitterpunkt $0, 0, 0$ liegt damit logisch sogar am Ende der -1-ten Zeile. Es empfiehlt sich also, den logischen Nullpunkt – zumindest des roten Halbfeldes – entsprechend 16 Byte hinter den Beginn des allozierten Speicherbereichs zu legen. (Siehe Abb. 3.2)

Vor dem Ende von Zeilen und zwischen Ebenen können gleichmäßig Füllbereiche zur Vermeidung von Cachekonflikten eingefügt werden, deren Größe ein Vielfaches von 16 Byte betragen muss.

Dieses Speicherlayout ermöglicht zum einen den getrennten Zugriff auf rote und schwarze Punkte. Zum anderen können dank der Speicherausrichtung parallele Ladeoperation sinnvoll genutzt werden, wodurch hoher Durchsatz zum L2-Cache leichter erreichbar wird, und Slots für weitere Optimierungen frei werden.

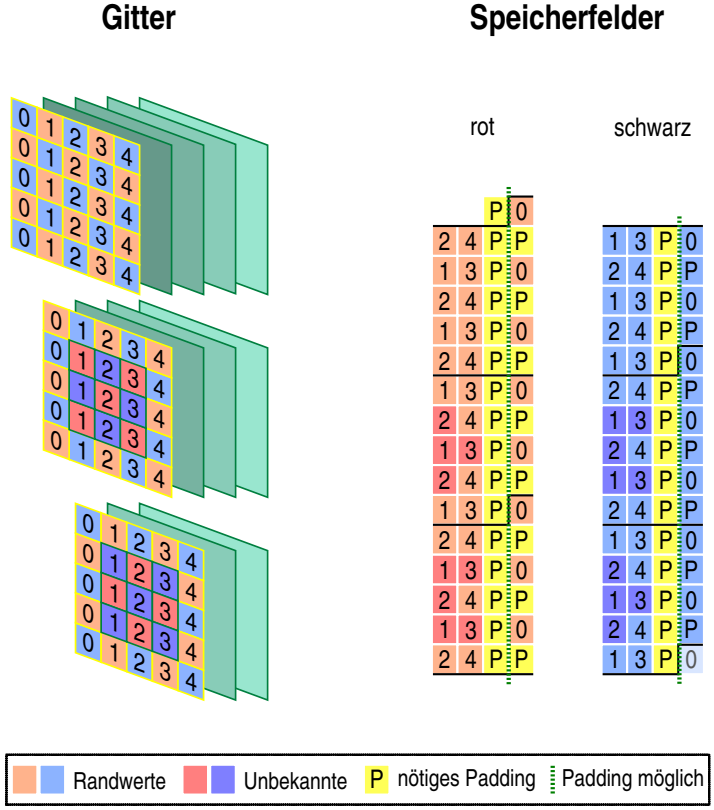
Nachteilig ist die kompliziertere Adressierung und die Erhöhung der Anzahl an Datenströmen. Mit handoptimiertem Code kann dem jedoch leichter begegnet werden als in Hochsprache.

3.2.4 Allgemeine Optimierungen

Das Konzept des Itanium-Prozessors, eine einfache aber leistungsfähige Prozessorplattform zu schaffen, verlagerte einen großen Teil der Komplexität auf die Code-Erzeugung. Deshalb wurden nur die für die Geschwindigkeit unrelevanten Teile, in diesem Fall Initialisierung, die Ablaufsteuerung des V-Zyklus und die Geschwindigkeitsmessung, in C++ geschrieben.

Die Berechnungen selbst wurden per Hand in Assembler kodiert, und zumindest bei den inneren Schleifen explizites Bundling durchgeführt. Erst so ist die Anwendung vieler oben besprochener Optimierungstechniken möglich. Alle Komponenten führen die Berechnungen bis zum Wechsel in andere Ebenen überlappt aus und sind dabei zu vorausschauendem Prefetching fähig.

Abbildung 3.2: Speicherlayout der ersten drei Ebenen bei $3 \times 3 \times 3$ Unbekannten



Viele dieser Optimierungen können von heutigen Compilern nicht durchgeführt und auch in keiner gebräuchlichen höheren Programmiersprache nachgebildet werden. Auch mit Compiler-Flags und Pragma-Anweisungen im Programmcode kann hier nur sehr begrenzt Abhilfe geschaffen werden.

Höhere Programmiersprachen sollen zudem einen Algorithmus in einer allgemeinen, von Computern interpretierbaren Form darstellen können, ohne die ausführende Plattform genau kennen zu müssen. Sie bieten deshalb keine Anweisungen oder Konzepte, die sich nur auf speziellen Prozessoren sinnvoll über- und einsetzen lassen.

Je speziellere, stark auf eine Plattform zugeschnittene Optimierungen genutzt werden sollen, desto fraglicher wird der Einsatz einer höheren Programmiersprache. Da dieselben Optimierungen auf anderen Plattformen nicht nutzbar sind, bringt die Portierbarkeit kaum Vorteile. Stattdessen wird man von einem speziellen Compiler, möglicherweise in einer bestimmten Version, abhängig, der beim gegebenen Quellcode auch die gewünschten Optimierungen durchführt.

3.2.5 Optimierung des Glätters

Der Glätter ist die rechen- und unoptimiert auch speicherintensivste Komponente des Mehrgitterlösers, seiner Optimierung kommt somit zentrale Bedeutung zu.

Für die Berechnung eines neuen Wertes werden am Itanium sieben Gleitpunktinstruktionen benötigt, wobei acht Flops nach üblicher Rechenart ausgeführt werden. Ohne Ausrollen werden hierfür vier, mit Ausrollen für die Berechnung von zwei Werten sieben Takte benötigt.

Dabei müssen fünf Werte geladen werden, der Wert von $V(x+1, y, z)$ kann bei der nächsten Berechnung als $V((x+2)-1, y, z)$ durch die rotierenden Register wiederverwendet werden. Zudem muss das Ergebnis wieder in den Speicher geschrieben werden, was zunächst zu einem Store Miss und Einlagerung der Cachezeile führt.

Aufgrund des Speicherlayouts sind unterschiedliche „Stempel“ zur Berechnung neuer Werte nötig, je nach Farbe der ersten Unbekannten bzw. des Randwertes. Diese Unterscheiden sich durch Ausrichtung und Ort der beiden Werte $V(x \pm 1, y, z)$. Beide Fassungen können jedoch nach entsprechender Initialisierung mit demselben Schleifenkörper berechnet werden, wenn für diese mittleren Werte einfache Ladeoperationen genutzt werden, die keine Speicherausrichtung verlangen.

Für die Berechnung von zwei Werten sind so zwei einfache und fünf parallele Ladeoperationen sowie zwei Speicheroperationen nötig.

3.2.5.1 Implizites Blocking

Werden soweit möglich neue Werte in einem Bereich von zwei mal zwei Zeilen gleichzeitig berechnet, also in den Zeilen y, z bis $y + 1, z + 1$, kann dies zur Reduktion von Speicher- und Gleitpunktoperationen genutzt werden.

In jeweils zwei Zeilen wird eine der Stempelarten benötigt. Dadurch können zumindest in zwei Zeilen parallele Ladeoperationen verwendet werden.

Die geladenen Werte aus den vier inneren Zeilen tragen zu jeweils vier Berechnungen bei ($V(x, y, z)$ z. B. wird für die Berechnung von $V(x + 1, y, z)$, $V(x - 1, y, z)$, $V(x, y + 1, z)$ und $V(x, y, z + 1)$ benötigt), so dass die Berechnung von acht Werten mit nur noch vier einfachen und 14 parallelen Ladeoperationen auskommt; es sind weiterhin acht Schreiboperationen nötig.

Auch eine halbe Gleitpunktberechnungen kann pro Unbekannter gespart werden, wenn Zwischenergebnisse günstig genutzt werden. So tragen die Summen aus den Zeilen $y + 1, z$ und $y, z + 1$ sowohl zu den Werten in x, y als auch $x + 1, y + 1$ bei, und ebenso aus y, z und $y + 1, z + 1$ zu $y, z + 1$ und $y + 1, z$. Zur Berechnung von acht Werten sind dann nur 52 statt 56 Gleitpunktberechnungen nötig.

Durch die große Anzahl freier Slots können das vorausschauende Prefetching (siehe 3.1.5) und viele Befehle zur Schleifenüberlappung im Schleifenkörper mit ausgeführt werden.

Die gleichzeitige Berechnung in vier Zeilen entspricht einem impliziten räumlichen Blocking, nachdem immer zwei Ebenen gleichzeitig berechnet werden. Bei einer ungerade Anzahl von Zeilen oder Ebenen muss allerdings auch Code zur Berechnung von zwei bzw. einer Zeile vorhanden sein.

3.2.5.2 Explizites Blocking

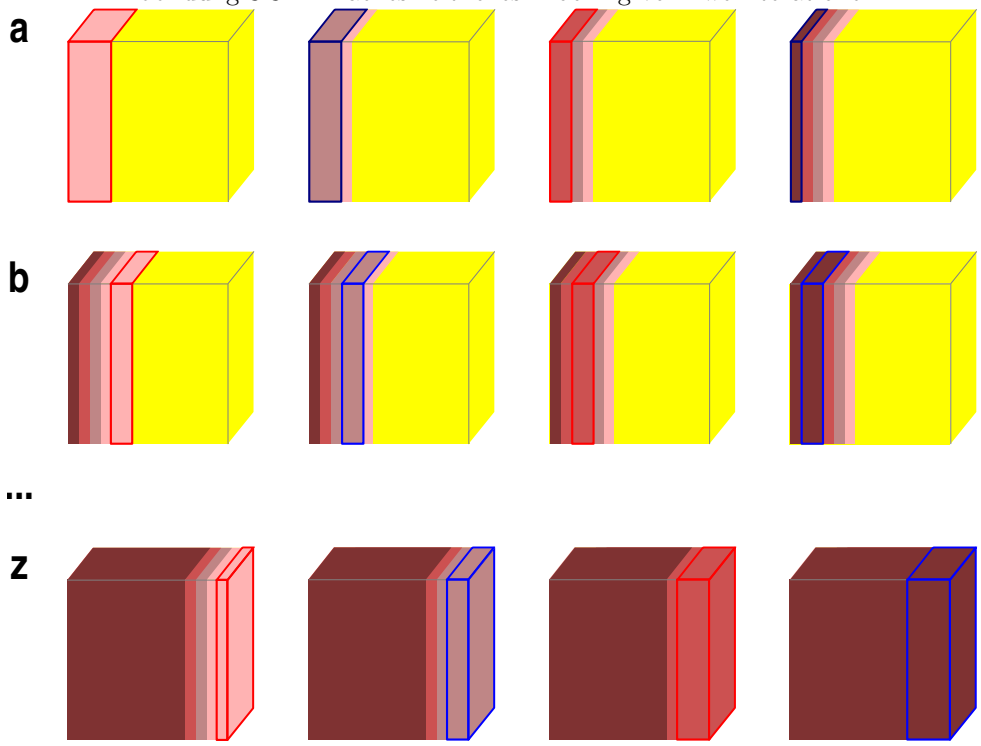
Durch Verwendung von nur räumlichem Blocking kann bei größeren Problemen im günstigsten Fall erreicht werden, dass – in diesem Fall je Halbiteration – alle benötigten Werte nur einmal in die Cachehierarchie geladen und bei Änderung in den Hauptspeicher zurück geschrieben werden müssen. Für das verwendete Speicherlayout würden für die erste rote Halbiteration demnach $V_{schwarz}$ und F_{rot} gelesen, und V_{rot} geschrieben und dafür implizit ebenfalls gelesen werden. Minimal könnte so für den Glätter ein Speichertransfer von vier Gleitpunktwerten pro Unbekannter und Halbiteration erreicht werden.

Hier besteht allerdings die Möglichkeit, durch zeitliches Blocking mehrere Halbiterationen auszuführen, solange diese Daten in den Caches verweilen können. Dadurch kann der Datendurchsatz in den Hauptspeicher weiter verringert werden. Am Itanium eignen sich dafür größere Blöcke einfacher Struktur. Die großen und schnellen Caches können dann mit überlappten Schleifen effizient durchlaufen werden. Bei dieser Implementation ist deshalb ein Paar aus zwei Ebenen die primäre Blockgröße.

Beim zeitlichen Blocken von i Iterationen (bereits $i = 1$ Iterationen bestehen aus 2 Halbiterationen) wird zunächst der Zustand hergestellt, dass in den ersten $2 \cdot i$ Ebenen die erste rote Halbiteration, in den ersten $2 \cdot i - 1$ Ebenen auch die erste schwarze Halbiteration und bei $i > 1$ immer so weiter, bis in Ebene 1 die i-te schwarze Halbiteration und damit die gewünschten Berechnungen dort abgeschlossen wurden. (Abb. 3.3 a)

Hiervon ausgehend, kann zunächst die rote Halbiteration in den Ebenen $2 \cdot i + 1$ und $2 \cdot i + 2$, danach die erste schwarze Halbiteration in den Ebenen $2 \cdot i$ und $2 \cdot i + 1$ usw. berechnet werden. Der

Abbildung 3.3: Einfaches zeitliches Blocking von zwei Iterationen



Block der ersten roten Halbiteration wird dabei immer um zwei Ebenen versetzt, und es bildet sich zwischen den ersten Ebenen, in denen alle i Iterationen bereits abgeschlossen sind, und den letzten noch unberührten Ebenen eine Art „Treppe“. (Abb. 3.3 b)

Am Ende müssen, entsprechend dem Aufbau der Treppenstruktur zu Beginn, in einer speziellen Randbehandlung die noch ausstehenden Berechnungen abgeschlossen werden. (Abb. 3.3 z)

Ist die Größe der Ebenen und die Anzahl der geblockten Iterationen i klein genug, können alle Daten von ihrer ersten bis zur letzten Verwendung in den Caches gehalten werden. Abhängig von i, der Cache- und Ebenengröße kann es jedoch nötig sein, eine weitere Blocking-Ebene einzuführen.

Die Blöcke der zweiten Blocking-Ebene, im folgenden als Überblöcke bezeichnet, unterteilen das gesamte Gitter jeweils in der x-z-Ebene, bestehen also aus einem bestimmen Zeilenbereich aller Ebenen. Auch auf dieser Blocking-Ebene sind spezielle Randbehandlungen in den Blöcken notwendig, die die Zeilen y, z mit $y = 1$ oder $y = y_{max} - 1$ enthalten.

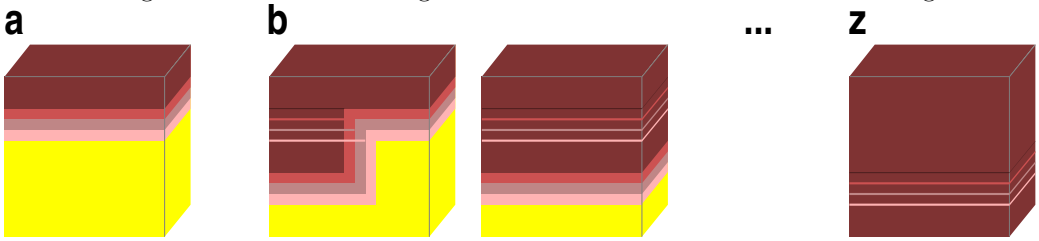
Grundsätzlich kann innerhalb aller Überblöcke das oben beschriebene Blocking-Verfahren untergeordnet genutzt werden. Dabei wird im ersten Überblock immer nur für die erste rote Halbiteration die volle Höhe genutzt und diese je Halbiteration um eine Zeile erniedrigt. Dieser Überblock benötigt deshalb auch eine größere Mindesthöhe. (Abb. 3.4 a)

Für die mittleren Blöcke werden zwar in jeder geblockten Halbiteration gleich viele Zeilen berechnet, diese findet jedoch jeweils um eine Zeile nach oben verschoben statt. (Abb. 3.4 b)

Am unteren Ende ist wieder ein Abschluss der fehlenden Berechnungen nötig. (Abb. 3.4 z)

Dieses zweistufige Blocking-Verfahren nimmt dabei in Kauf, dass die Zeilen an den Überblockrändern in y-Richtung mehrfach geladen werden müssen, die restlichen Werte aber solange benötigt in den Caches verweilen können. Die Anzahl der Überblöcke sollte demnach möglichst klein gewählt werden, soweit die Anzahl der zeitlich geblockten Iterationen i, die Zeilenlänge und die Cachegröße dies zulassen.

Abbildung 3.4: Zeitliches Blocking von zwei Iterationen mit weiterer Blocking-Ebene



Abhängig von der Problemgröße

- sollte wenn möglich auf ganze Ebenen geblockt werden, aber
- ein einfacher Glätter genutzt werden, wenn das Problem für Blocking zu klein ist, und
- bei zu großen Ebenen eine zweite Blocking-Ebene eingeführt werden.

3.2.5.3 Erweitertes Prefetching

Insbesondere zeitlich geblockter Code weist einen stark schwankenden Bedarf an Daten aus dem Hauptspeicher auf.

Beim optimierten Speicherlayout und dem Blocken auf jeweils zwei ganze Ebenen fällt unter Vernachlässigung der Randwerte folgender Hauptspeichertransfer an: Die erste rote Halbiteration benötigt für jede Unbekannte im Schnitt zwei neue Werte aus dem Hauptspeicher ($V_{schwarz}$ und F_{rot}), ein weiterer wird aufgrund eines Write Miss eingelagert (V_{rot}). Bei der folgenden schwarzen Halbiteration muss nur noch der entsprechende Wert der rechten Gleichungsseite aus $F_{schwarz}$ geladen werden, weiteren Halbiterationen können im Ideal komplett auf den Caches arbeiten.

Die Verdrängungssemantik der Caches verstärkt dies, weil beim Einlagern vieler neuer Zeilen auch viele geänderte Zeilen ausgeschrieben werden.

Als Beispiel kann auf dem Testsystem (siehe 4.1) ein Wert in minimal $3,25$ Takten berechnet werden, währenddessen maximal 13 Byte übertragen werden können. Die erste rote Halbiteration ist damit sicher, die erste schwarze aufgrund der Verdrängung vermutlich von der Speicherbandbreite, weitere geblockte Berechnungen allerdings primär durch die Rechenleistung beschränkt.

Sind für jeden der n Abschnitte eines Algorithmus die Anzahl der Operationen o und die Ausführungsgeschwindigkeit T bekannt, so ergibt sich die durchschnittliche Ausführungsgeschwindigkeit zu

$$T_{ges} = \frac{o_{ges}}{t_{ges}} = \frac{\sum_{i=1}^{n} o_i}{\sum_{i=1}^{n} \frac{o_i}{T_i}}$$

und bei $o_i = o$ zu

$$T_{ges} = \frac{\sum_{i=1}^{n} o}{\sum_{i=1}^{n} \frac{o}{T_i}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{T_i}},$$

was der Berechnung des harmonischen Mittelwerts entspricht.

Durch gezieltes Prefetching kann der Speicherdurchsatz geglättet und auf die einzelnen Abschnitte der Berechnung verteilt werden, so dass nach Möglichkeit entweder alle durch Speicher- oder Prozessordurchsatz beschränkt sind.

Dazu werden nicht nur die in Kürze benötigten Operanden in den L2-Cache, sondern auch später benötigte Werte in die unterste Cacheebene vorgeladen. Bei diesem Blocking-Verfahren muss das Vorladen der acht benötigten Halbebenen möglichst gleichmäßig auf die vorherigen $i \cdot 2$ Halbiterationen verteilt werden. Die Operanden der Berechnungen selbst sind dabei schon vorher in die

Cachehierarchie gebracht worden und müssen bestenfalls vorher vom L3- in den L2-Cache geholt werden.

3.2.5.4 Parallelisierung

Durch Parallelisierung kann die Ausführungsgeschwindigkeit dann gesteigert werden, wenn die serielle Berechnung durch die Rechengeschwindigkeit begrenzt ist oder aus anderen Gründen noch Bandbreite zum Hauptspeicher verfügbar ist. Dies ist insbesondere beim Glätter der Fall, wenn mehrere Iterationen zeitlich geblockt werden.

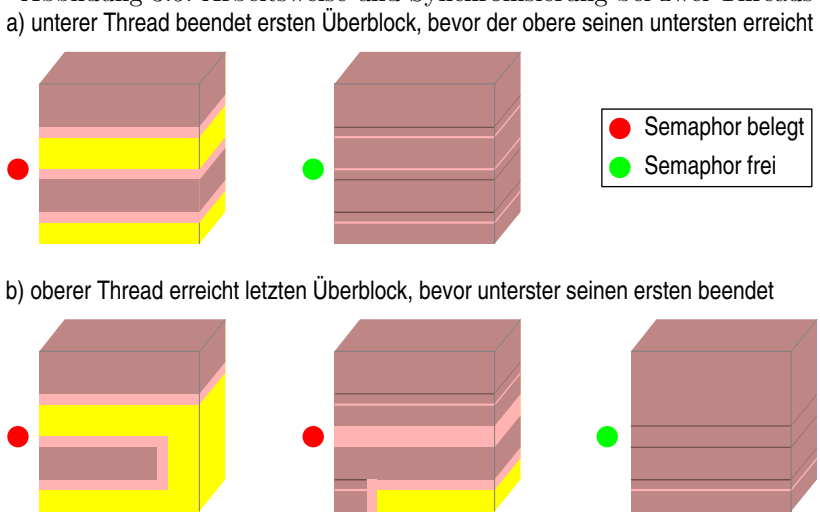
Da das Testsystem nur zwei Prozessoren besitzt, wurde sich dabei auch auf zwei Threads beschränkt, das Prinzip lässt sich allerdings auch für mehr Prozessoren verallgemeinern. Vielprozessorsysteme besitzen jedoch meist einen verteilten Speicher, so dass weitere Optimierungen sinnvoll sind, die den Rahmen dieser Arbeit übersteigen.

Der parallele Glätter baut direkt auf den Glätter mit Überblöcken auf. Das Gitter wird dabei in der x-z-Ebene etwa halbiert und jedem Teil ein Thread zugewiesen. Diese können in ihrem Gebiet weitgehend nach bisherigem Muster arbeiten, eine besondere Behandlung ist nur in der Mitte nötig, wo beide Bereiche zusammentreffen.

Dieser Übergangsbereich wird durch einen Semaphor abgesichert. Der erste Überblock des „unteren“ Thread beginnt dort, und da oberhalb noch keine Werte berechnet sind, muss in der Mitte eine Treppenstruktur gebildet werden. Ist dieser Überblock abgeschlossen, gibt dieser Thread den Semaphor und damit den Zugriff auf diese Treppenstruktur frei.

Der „obere“ Thread startet mit dem Überblock bei Zeile 1,1. Beginnt er seinen letzten Überblock, der dann am Übergang der Bereiche endet, überprüft er den Semaphor; ist dieser bereits freigegeben, kann er den Übergangsbereich mit dem Überblock schließen. Andernfalls ist er gezwungen, auch von seiner Seite eine Treppenstruktur aufzubauen und diese erst nach Verfügbarkeit des Semaphors zu vervollständigen. (vgl. Abb. 3.5)

Abbildung 3.5: Arbeitsweise und Synchronisierung bei zwei Threads



3.2.6 Optimierung der Residualberechnungen

Wie in 3.2.2 festgestellt, müssen nur an den roten Punkten Residua berechnet werden, da sie nach einem Glättungsschritt an allen schwarzen Punkten im Rahmen der Rechengenauigkeit gleich 0 sind.

Bei einer naiven Implementierung würden alle roten Residua berechnet und in ein temporäres rotes Halbfeld geschrieben. Dafür müssen dann V_{rot}, $V_{schwarz}$ und F_{rot} gelesen und das temporäre R_{rot} geschrieben (und wegen des Write-Allocate-Verhaltens auch gelesen) werden.

Nachdem allerdings auch ein Nachglättungsschritt vorausgesetzt wird, um Interpolation und Korrektur vereinfachen zu können, werden die roten Werte nach der Berechnung des Residuums nicht mehr benötigt. Die berechneten Residua können deshalb an die Stelle der entsprechenden Unbekannten geschrieben werden.

Die Berechnung der Residualnormen wird nur am Ende von V-Zyklen durchgeführt. Die dafür berechneten Residua müssen nicht geschrieben werden, sondern können in zwei Registern gesammelt werden. Je Norm und berechnetes Residuum ist dafür nur eine Gleitpunktoperation am Itanium nötig, da die Quadratsumme für die euklidische Norm über ein Fused Multiply Add aufsummiert werden kann, und auch eine Instruktion zur Bestimmung des absoluten Maximalwertes von zwei Unbekannten existiert.

Eine weitere Verbesserung gegenüber dem üblichen Verfahren ergibt sich, wenn die Residualberechnungen mit dem geblockten Glätter zusammengelegt und analog zu einer $(i+1)$-ten roten Halbiteration durchgeführt werden. Dabei können die Residua entweder an Stelle der roten Unbekannten geschrieben oder zur Berechnung der Normen genutzt werden.

Können auch für diese zusätzlichen Operationen noch alle benötigten Ebenen im Speicher gehalten werden, erhöht sich der Hauptspeicherdurchsatz des einfach geblockten Glätters nicht. Bei der Verwendung von Überblöcken muss jedoch eine geringere Höhe genutzt werden, was jedoch in Kauf genommen werden kann.

3.2.7 Optimierung der Restriktion

Die optimierte Restriktion berechnet zwar ebenso viele Werte, muss dafür jedoch nur die Residua in einem roten Halbfeld beachten. Dennoch stellen Daten aus dem feinen Gitter den Großteil des dominierenden Speicherdurchsatzes dar, so dass das rote Halbfeld nach Möglichkeit genau einmal durchlaufen werden sollte.

Die gewählte Lösung durchläuft des feinen Gitters jeweils in Blöcken von zwei mal zwei Zeilen, nämlich $2 \cdot y - 1, 2 \cdot z - 1$ bis $2 \cdot y, 2 \cdot z$. Dabei werden werden die Beiträge der Residua auf die vier Zeilen $y - 1, z - 1$ bis y, z im groben Gitter verteilt (vgl. 3.6).

Eine Zeile im groben Gitter wird dabei insgesamt vier Mal durchlaufen, davon jeweils zweimal direkt nacheinander. Während das erste mal der vorherige Inhalt überschrieben wird, muss der bisherige Wert danach gelesen und neue Beiträge hinzuaddiert werden. Durch konsequente Mehrfachnutzung von Operanden und Zwischenergebnissen werden – neben Mehraufwand an den Enden der Zeile – elf Rechenoperationen für jede Restriktion benötigt.

Auch an den Seiten in y- und z-Richtung ist dabei eine spezielle Randbehandlung nötig. Dort kann derselbe Schleifenkörper eingesetzt werden, indem sich die Berechnungen in den vier Zeilen des groben Gitters über Prädikate einzeln an- und abschalten lassen. So werden zwar alle Zeilen durchlaufen, jedoch nur $1, 1$ bis $y_{max} - 1, z_{max} - 1$ geschrieben.

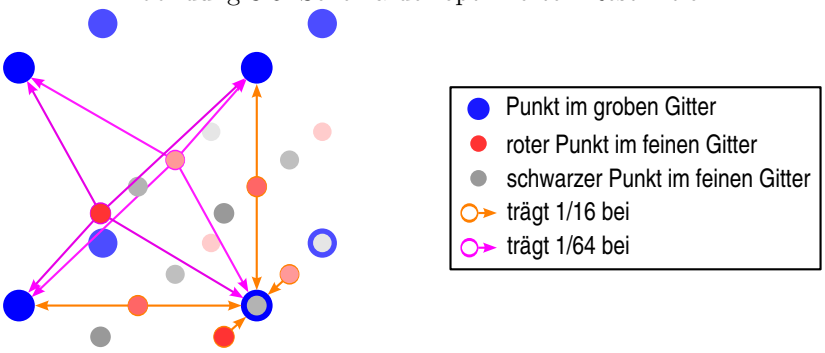
Besonders kleine Probleme können jedoch nicht mit der überlappten Software Pipeline berechnet werden, da Epi- und Prolog so weit überlappen würden, dass die beim ersten und dritten Durchlauf einer Zeile berechneten Werte beim zweiten und vierten noch nicht geschrieben sind.

Sind die Ebenen klein genug, stehen die Werte des groben Gitters für alle vier Durchgänge in den Caches zur Verfügung. Andernfalls kann dies durch ein einfaches räumliches Blocking ermöglicht werden, wobei die Felder – ähnlich wie beim Glätter – in der x-z-Ebene unterteilt werden.

3.2.8 Optimierung von Interpolation und Korrektur

Interpolation und Korrektur sind das Gegenstück zur Restriktion. Nachdem ein Nachglättungsschritt erwartet wird, müssen dabei nur die schwarzen Unbekannten des feinen Gitters korrigiert

Abbildung 3.6: Schema der optimierten Restriktion

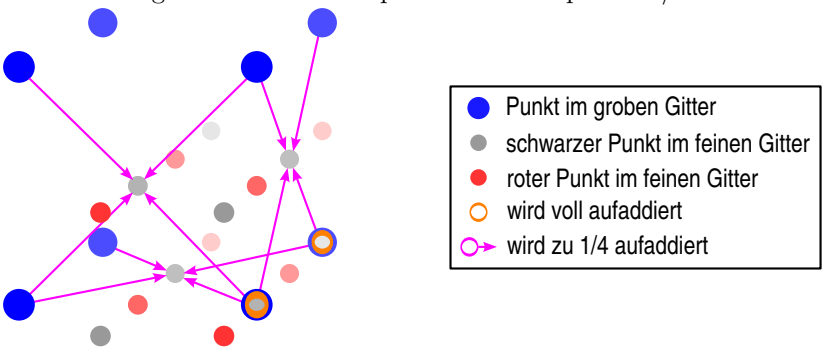


werden.

Wie bei der Restriktion wird die Interpolation in je zwei mal zwei Zeilen des groben und des feinen Gitters berechnet. Die Werte des feinen Halbfeldes werden dabei je einmal gelesen und korrigiert geschrieben (vgl. Abb 3.7). Die Zeilen des groben Gitters müssen insgesamt viermal, je zweimal direkt hintereinander, geladen werden.

Pro Interpolation und Korrektur sind im Schnitt fünf Instruktionen nötig.

Abbildung 3.7: Schema der optimierten Interpolation/Korrektur



Da bei der Interpolation das feine schwarze Halbfeld sowohl gelesen als auch geschrieben werden muss, ist der Anteil des groben Gitters am Datenaustausch im Verhältnis gering. Deshalb macht sich räumliches Blocking kaum bemerkbar.

Eine Randbehandlung ist immer dann notwendig, wenn in y- oder z-Dimension eine ungerade Anzahl von Unbekannten vorhanden ist – wie es beim Mehrgitterverfahren immer der Fall ist. Dies wurde ähnlich wie bei der Restriktion gelöst, indem durch Setzen von Prädikaten die „Korrektur“ von Randwerten verhindert werden kann.

3.2.9 Optimierter V-Zyklus im Überblick

Obwohl derselbe Algorithmus durchgeführt wird, unterscheidet sich der optimierte V-Zyklus von der direkten Umsetzung, wie sie in 2.1.2 beschrieben wurde.

Analog zu Tabelle3.1 in 3.2.1 kann der optimierte V-Zyklus nach Rechen- und Speicheroperationen charakterisiert werden wie in Tabelle 3.3. Dabei wird zusätzlich angenommen, dass ein Halbfeld genau halb so viele Unbekannte wie das gesamte Gitter enthält.

Die daraus resultierende Verteilung auf die entsprechenden Komponenten ist Tabelle 3.4 zu entnehmen. In Vergleich zu Tabelle 3.2 erkennt man deutlich, dass die Anzahl der Gleitpunktoperationen moderat, der Speicherzugriffe jedoch deutlich gesenkt werden konnte. Zudem können die rechen-

intensiven Anteile, also Glätter mit und ohne Residualberechnungen, auch auf zwei Prozessoren verteilt werden.

Optimierter V-Zyklus in Pseudo-Code

```
globale anzahl Pre mit Pre größer 0
globale anzahl Post mit Post größer 0

funktion Vzyklus(halbfeld Vrot, halbfeld Vschwarz, halbfeld RSrot, halbfeld RSschwarz) {
    wenn (Anzahl_Unbekannte(V) == 1) {
        Glätte(Vrot,Vschwarz,RSrot,RSschwarz,1)
    } sonst {
        neues grobes halbfeld Erot
        neues grobes halbfeld Eschwarz
        neues grobes halbfeld Rrot
        neues grobes halbfeld Rschwarz

        Glätte&Residuum(Vrot,Vschwarz,RSrot,RSschwarz,Pre)

        Setze_Anfangslösung(Erot,Eschwarz)

        Restriktion(Vrot,Rrot,Rschwarz)

        Vzyklus(Erot,Eschwarz,Rrot,Rschwarz)

        Interpoliere&Korrigiere(Erot,Eschwarz,Vschwarz)

        wenn (feinste gitterebene) {
            Glätte&Residualnorm(Vrot,Vschwarz,Rrot,Rschwarz,Post)
        } sonst {
            Glätte(Vrot,Vschwarz,RSrot,RSschwarz,Post)
        }
    }
}
```

Tabelle 3.3: Abschätzung Flops und Speicherzugriffe pro Unbekannter auf einer Gitterebene nach Optimierung

Komponente	**Flops** Itanium	**Speicherdurchsatz** Lesen	Schreiben
Glätter & Residuum	$7 \cdot i + \frac{8}{2}$ [2]	2	1
Restriktion	$\frac{11}{8}$	$\frac{1}{2} + \frac{1}{8}$	$\frac{1}{8}$
Interpolation & Korrektur	$\frac{5}{2 \cdot 2}$	$\frac{1}{2} + \frac{1}{8}$	$\frac{1}{2}$
Glätter & Residualnormen [1]	$7 \cdot j + \frac{10}{2}$ [2]	2	1
Glätter [1]	$7 \cdot j$ [2]	2	1

[1] Berechnung auch der Residualnorm nur beim feinsten Gitter
[2] i Vorglättungs- und j Nachglättungs-Iterationen

Tabelle 3.4: Anteil der Komponenten bei einem optimierten 2,2-V-Zyklus

Komponente	**Flops** (Itanium)	**Speicher-zugriffe**
Glätter & Residuum	46 %	39 %
Restriktion	3 %	9 %
Interpolation & Korrektur	3 %	13 %
Glätter & Residualnormen	44 %	34 %
Glätter	5 %	5 %
Anzahl pro Unbekannter	ca. 43,6	ca. 8,7

Kapitel 4

Bewertung

4.1 Vorstellung des Testsystems

Als Testmaschine stand eine zx6000 Workstation von Hewlett Packard zur Verfügung. Diese enthält zwei Itanium 2-Prozessoren mit Madison-Kern, die jeweils mit einem Takt von 1,4 GHz arbeiten und einen L3-Cache von 1,5 MB besitzen. Die Bandbreite der Caches hängt alleine von der Prozessortaktung ab (siehe hierzu 2.2.3.2).

Die Prozessoren teilen sich 10 GB SD-RAM mit doppelter Datenrate, der einen theoretischen Durchsatz von 6,4 GB/s auf beide Prozessoren verteilen kann. In der Praxis liegen die Transferraten jedoch niedriger, Tabelle 4.1 führt bei einem Datenstrom typische Transferraten auf, wobei die reine Schreibgeschwindigkeit nicht direkt gemessen werden kann.

Tabelle 4.1: Typischer Hauptspeicherdurchsatz des Testsystems

Read	5,6 GB/s (5,2 GiB/s)
Write & Allocate	2,4 GB/s (2,2 GiB/s)
Write	4,2 GB/s (3,9 GiB/s)

1 GB = 1000^3 Byte, 1 GiB = 1024^3 Byte

Zur Übersetzung der Komponenten wurden die Intel Compiler in der aktuellen Version 9.0 genutzt. Durch ein Batch-System konnte exklusiver Zugriff auf die Maschine gesichert werden, alle Geschwindigkeitsmessungen basieren auf echter Laufzeit.

4.2 Optimierung des Glätters

Der Optimierung des Glätters kommt eine zentrale Bedeutung zu. Er dominiert nicht nur die Geschwindigkeit des gesamten Mehrgitterlösers, sondern verspricht durch zeitliches Blocking großes Potential zur Beschleunigung, da bereits eine Iteration aus zwei Halbiterationen besteht. Dadurch bietet er auch eine gute Möglichkeit, Einsatz und Wirkung der verschiedenen Optimierungstechniken zu untersuchen.

4.2.1 Adressierung

Bei C werden mehrdimensionale Felder nur in zur Übersetzungszeit bekannter Größe programmiersprachlich unterstützt, entweder im statischen Speicherbereich oder auf dem Stapel. Bei mehrdimensionalen Feldern in dynamischem Speicher oder in variablen Größen muss die Position des Datums in einem großen eindimensionalen Feld explizit berechnet werden.

In Abbildung 4.1 werden verschiedene Möglichkeiten der Adressierung bei einer naiven Implementierung des Red-Black Gauss-Seidel gegenübergestellt. Für den Mehrgitterlöser wird ein Glätter benötigt, der auf Gittern verschiedener Größe in dynamisch alloziertem Speicher arbeiten kann (C, dynamisch). Im Gegensatz zu programmiersprachlich dreidimensionalen Gittern (C, statisch 3D-Array) misslingt dem Compiler jedoch eine effektive Umsetzung.

Der Grund für die schlechte Performanz bei dynamischer Adressierung lässt sich erst durch einen Blick auf den erzeugten Assembler-Code finden: Der Compiler ist dabei – unabhängig von Compiler-Flags zur aggressiven Optimierung – stets in der Lage, Software Pipelines mit guter Adressierung und passendem Prefetching zu erzeugen. Unerklärlich scheitert er jedoch bei der Aufteilung der Pipelinestufen und bei der Berücksichtigung von Speicher- und Instruktionslatenzen. Ob der Compiler Aliasing der benötigten Felder V und F ausschließen kann oder eine spekulative Ladeoperation für die Werte aus F nutzen muss, hat keinen messbaren Einfluss.

Eine schnelle, alternative Adressierung ist auch über eindimensionale Felder möglich. Der berechnete Punkt in V und der benötigte Wert der rechten Seite werden dabei über eine Laufvariable h, die Nachbarwerte in V relativ zu h adressiert. Zur Adressierung innerhalb der Zeile ist hier in Hochsprache nur die Änderung der Laufvariable nötig. Sind Zeilen- und Ebenengrößen bekannt, kann der Compiler somit noch effizienteren Code erzeugen (C, optimiert), erreicht aber nur eine minimale Geschwindigkeitssteigerung bei unbekannten Größen.

Durch die direkte Umsetzung in Assembler ist eine ebenso schnelle Berechnung auch bei zur Übersetzungszeit nicht bekannten Dimensionsgrößen möglich. Adressierung und Prefetching der Datenströme innerhalb der Schleifen wird dabei wie bei den compiler-erzeugten Versionen realisiert, und keine spezielle Optimierungstechnik angewandt.

Bei kleinen Problemgrößen leiden alle Versionen unter den dann im Verhältnis vielen Berechnungen zur Vorbereitung der nächsten Schleife und dem großen Aufwand zum Be- und Entladen der Software Pipelines. Noch deutlich vor Erreichen der ohne Ausrollen theoretisch erreichbaren 2,8 GFlop/s bricht die Geschwindigkeit deutlich ein, wenn zwischen zwei Halbiterationen nicht mehr alle Daten im L3-Cache gehalten werden können (oberhalb ca. 50^3). Der letzte Abfall ist dann festzustellen, wenn die Daten einer Ebene die Cachegröße übersteigen (jenseits 200^3). Die Größe des L2-Caches ist hierbei unerheblich; solange er die Daten der zuletzt berechneten Zeile noch fassen kann (theoretisch bis über Zeilenlänge 5000), kann der L3-Cache die benötigten Daten scheinbar schnell genug liefern.

4.2.2 Speicherlayout

Das optimierte Speicherlayout mit farblich getrennten Feldern verfolgt zwei verschiedene Ziele.

Nach mindestens doppeltem Ausrollen der Schleifen können parallele Ladeoperationen genutzt werden, wodurch Bankkonflikte im L2-Cache vermieden und mehr Slots verfügbar werden. Letztere können jedoch bei einer einfachen Umsetzung nicht sinnvoll genutzt werden, so dass dieser Vorteil erst bei weiteren Optimierungsschritten sichtbar wird.

Viele Komponenten des Mehrgitterlösers können ihren Speichertransfer deutlich reduzieren, wenn Halbfelder einzeln gelesen oder geschrieben werden können. Auch für den ungeblockten Glätter kann der begrenzende Hauptspeicherdurchsatz so verringert werden..

Neben der komplexeren Adressierung ist jedoch der leicht erhöhte Speicherbedarf zur Speicherausrichtung ein Nachteil, der insbesondere bei kleineren Problemen ins Gewicht fällt. Nachdem die Füllwerte mit Nutzwerten gemeinsam in Cachezeilen liegen, steigt der Speichertransfer unterhalb des L2-Caches leicht an und weniger Nutzdaten können in den Caches gehalten werden (vgl. Abb. 4.2).

4.2.3 Schleifenoptimierung

Zunächst sollte eine möglichst schnelle Berechnung auf Daten in den Caches möglich sein, bevor der üblicherweise begrenzende Hauptspeicherdurchsatz durch Blocking-Techniken verringert wird.

Abbildung 4.1: Adressierungsprobleme beim Glätter

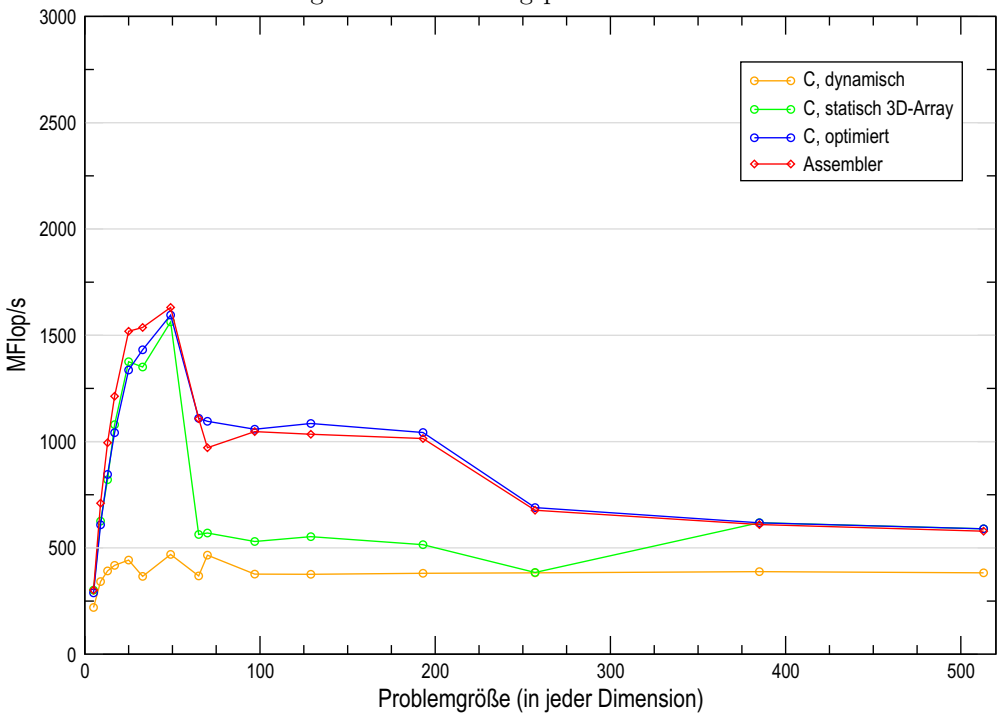
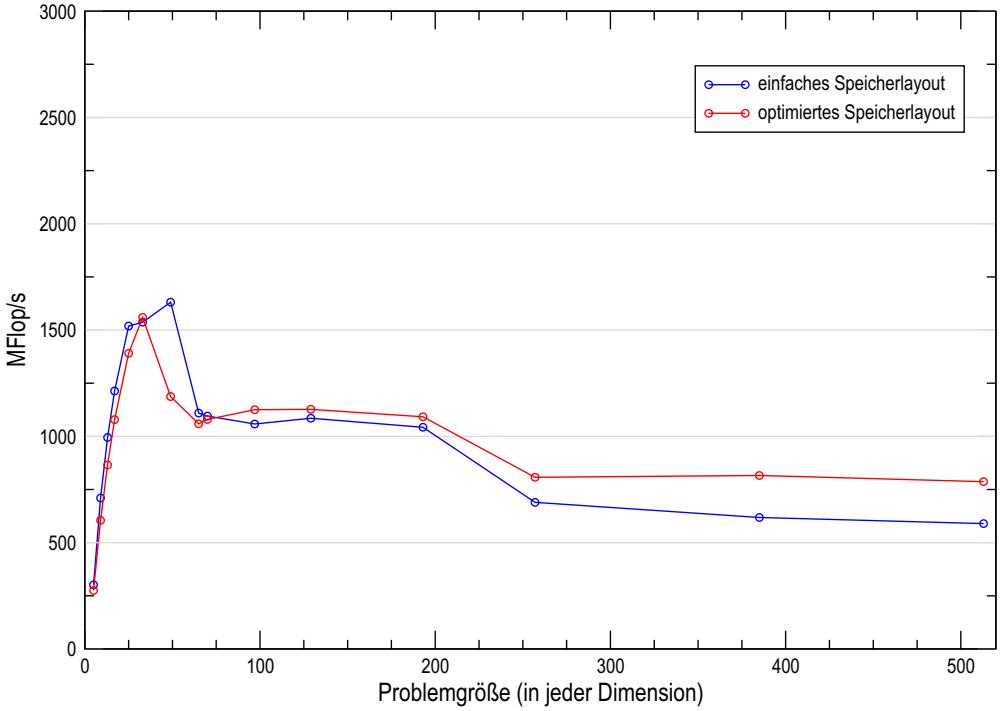


Abbildung 4.2: Glätter mit optimiertem Speicherlayout



In Abbildung 4.3 werden dem einfachen Glätter in Assembler zwei Versionen mit weiteren Optimierungen gegenübergestellt. Um die Geschwindigkeit unabhängig vom Hauptspeicher messen zu können, sollten alle benötigten Daten in den Caches gehalten werden können. Dazu wurde die Problemgröße nur in x-Dimension variiert und in y- und z-Richtung jeweils eine Dimensionsgröße von 12 inklusive Randwerte gewählt.

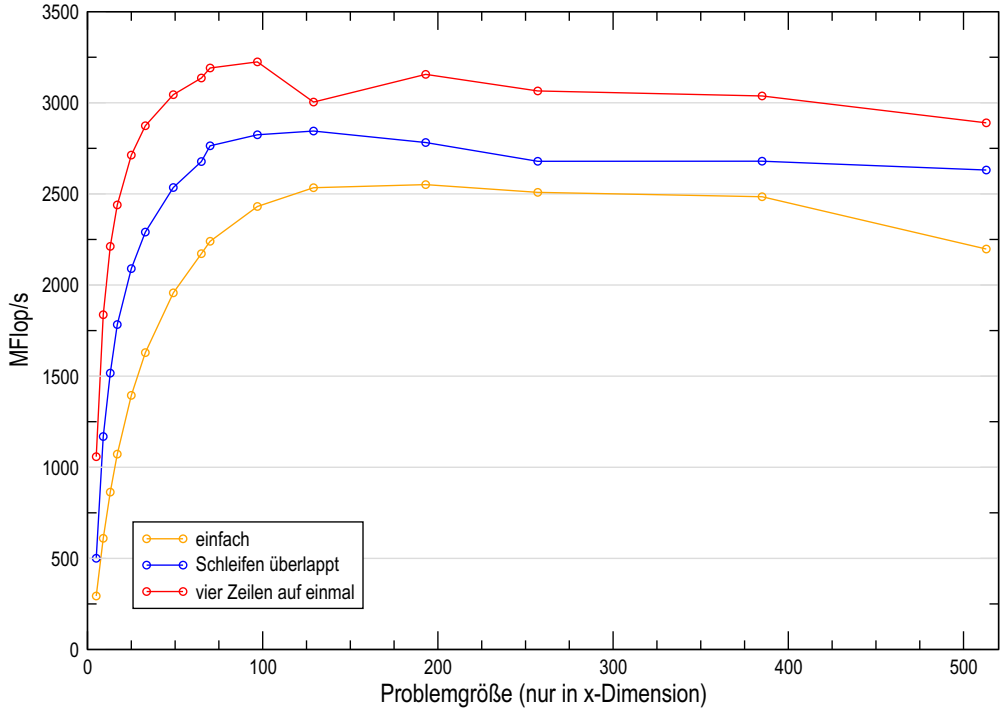
Im Gegensatz zur einfachen Version kann durch die Überlappung der Schleifen und durch Wiederverwendung der Werte im mittleren Datenstrom v. a. bei kurzen Schleifen eine deutlich höhere Geschwindigkeit erreicht werden.

Die dritte Version enthält einen voll optimierten Schleifenkörper, wie er auch beim geblockten Code zum Einsatz kommt. Die Software Pipeline berechnet je zwei Werte in vier benachbarten Zeilen, nutzt geladene Werte mehrfach und kann voll überlappt werden. Durch die Mehrfachnutzung von Zwischenergebnissen kann diese Schleife durch eingesparte Berechnungen effektiv über 3,4 GFlop/s statt 3,2 GFlop/s auf dem Testsystem erreichen.

Durch den niedrigeren Bedarf an Speicheroperationen wird außerdem die Bandbreite des L2-Caches geschont, was zusätzlich dem Austausch mit dem L3-Cache zugute kommt. Das implizite Blocking reduziert den Speicherverkehr zwischen den Caches weiter, bei großen Problemen auch in den Hauptspeicher.

Ein Nachteil dieser Technik ist jedoch, dass bei ungerader Anzahl von Zeilen und bzw. oder Ebenen weiterer effizienter Code zur Randbehandlung bereit gehalten werden muss. Hierzu wird eine spezielle Schleife genutzt, die Berechnungen in zwei benachbarten Zeilen durchführt. Durch Prädikate kann sie auch zur Berechnung einzelner Zeilen mit effektiv niedrigerer Geschwindigkeit genutzt werden. Die verschiedenen Schleifentypen können zwar selbst, jedoch nicht untereinander überlappt werden.

Abbildung 4.3: Schleifenoptimierungen beim Glätter

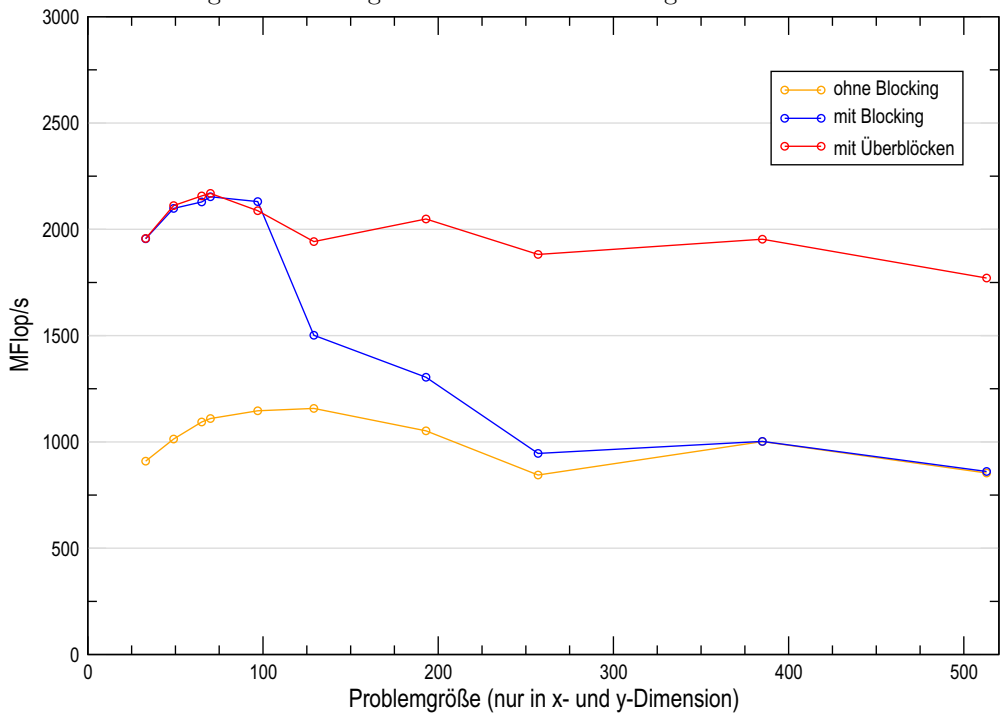


4.2.4 Blocking

Um die optimierten Berechnungen ausreichend mit Operanden zu versorgen, kann der Hauptspeicherdurchsatz durch Blocking-Techniken reduziert werden.

In Abbildung 4.4 sind dem optimierten Glätter (vgl. letzter Abschnitt) zwei Glätter gegenübergestellt, die je drei Iterationen zeitlich blocken: Zum einen auf Basis von (zwei) ganzen Ebenen, zum anderen mit der Möglichkeit, Überblöcke einzusetzen.

Abbildung 4.4: Blocking und zusätzliche Blocking-Ebene beim Glätter



Um auch bei kleineren Ebenen nicht nur auf den Caches zu arbeiten, wurde kein kubisches Gebiet, sondern eine Balkenform genutzt; zu einer vorgegebenen Ebenengröße mit $x = y$ wird z so gewählt, dass sich ein Speicherbedarf von ca. 2 GB für alle vier Halbfelder zusammen ergibt. Der erste Messpunkt entspricht demnach einem Gitter mit $33 \times 33 \times 123248$, der letzte mit $513 \times 513 \times 510$ Punkten inklusive Randwerte.

Alle Glätter erreichen ihre höchste Geschwindigkeit, wenn die Ebenen zwar möglichst groß sind, aber die Daten für den aktuellen Durchgang noch von ihrer ersten bis zu ihrer letzten Nutzung im L3-Cache gehalten werden können. Beim ungeblockten Glätter sind dies nur die Operanden der letzten Doppelebene, beim zeitlichen Blocking jedoch die Daten mehrerer Halbiterationen.

Die Geschwindigkeit des einfach geblockten Glätters sinkt bei zunehmender Ebenengröße deutlich ab, wenn nur von der zweiten bis zur letzten geblockten Halbiteration Werte wiedergenutzt werden können. Können die Daten der letzten berechneten Doppelebene nicht mehr vollständig in den Caches gehalten werden, wird derselbe Speicherbedarf und dieselbe Geschwindigkeit erreicht, wie dies ungeblockt der Fall wäre.

Der zu einer weiteren Blocking-Ebene fähige Glätter zerteilt bei Bedarf das Gitter in der x-z-Ebene, wobei am Übergang von Überblöcken Bereiche doppelt durchlaufen werden müssen. Je größer die Ebenen und die Anzahl der zeitlich geblockten Iterationen ist, desto mehr Überblöcke sind nötig und immer größere Teile der Halbfelder müssen mehrfach geladen werden, so dass die effektive Ausführungsgeschwindigkeit langsam absinkt. Ineffizient wird dieses Verfahren meist erst bei Problemgrößen, die wegen Speicher- und Zeitbedarfs typischerweise auf Vielprozessorsystemen mit verteiltem Speicher bearbeitet werden.

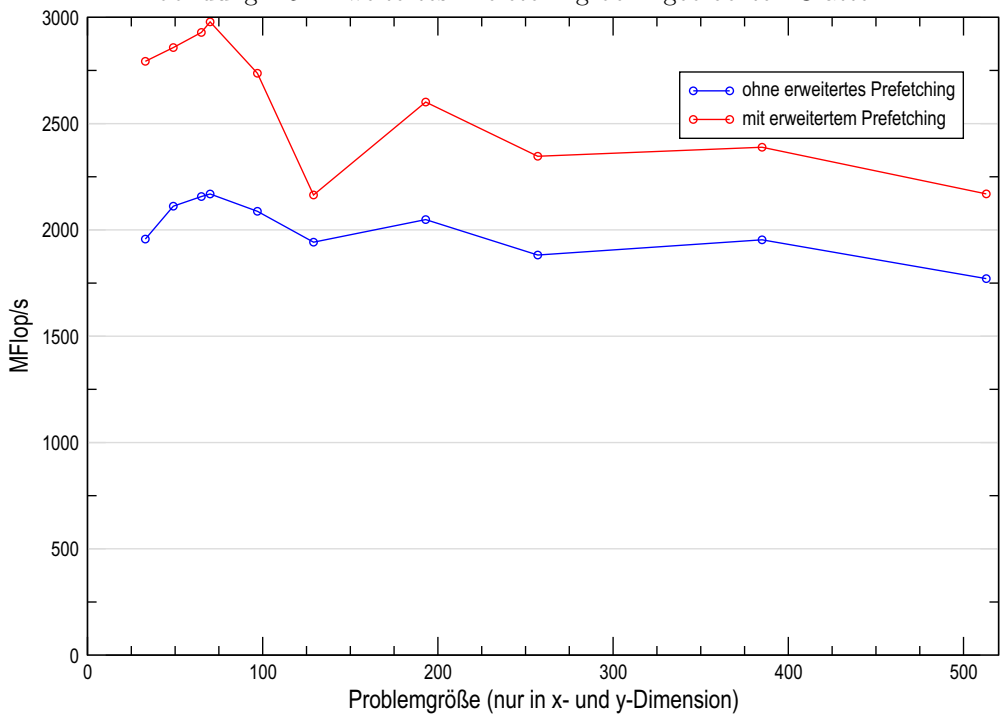
Die Entscheidung, ob und in welcher Höhe Überblöcke genutzt werden, ist für die Geschwindigkeit entscheidend. In diesem Fall wurde dies durch eine plausible Faustformel gelöst, die durchwegs zu guten Ergebnissen führte. Insbesondere für den Einsatz im Mehrgitterlöser könnten die optimalen Werte für die benötigten Größen und geblockten Iterationen auch experimentell ermittelt und jeweils vom Glätter in einer Tabelle nachgeschlagen werden.

4.2.5 Erweitertes Prefetching

Durch das erweiterte Prefetching soll der Speicherbedarf der einzelnen geblockten Halbiterationen möglichst gleichmäßig verteilt werden, so dass nach Möglichkeit alle Berechnungen entweder nur vom Speicher- oder nur vom Prozessordurchsatz abhängen.

Abb. 4.5 vergleicht die Geschwindigkeit mit und ohne erweitertes Prefetching bei drei zeitliche geblockten Iterationen. z wurde so groß gewählt, dass etwa 2 GB Speicher benötigt wurden.

Abbildung 4.5: Erweitertes Prefetching beim geblockten Glätter



Bei allen Ebenengrößen kann eine deutlich höhere Geschwindigkeit erreicht werden. Am größten ist diese, wenn keine Überblöcke benötigt werden. Bei diesen ist es schwieriger, alle benötigten Daten vorzuladen, da der Überblock zum Teil deutlich mehr Zeilen enthält, als bei jeder Halbiteration berechnet werden.

Die Geschwindigkeitssteigerung hängt dabei von der Anzahl geblockter Iterationen und von der Rechen- und Speichergeschwindigkeit ab. Bei nur einer zeitlich geblockten Iteration ist die Geschwindigkeit v. a. vom Hauptspeicher abhängig. Dort kann nur bei kleinen Ebenen eine moderate Beschleunigung erreicht werden, vermutlich weil der Datenaustausch zwischen den Caches günstiger ist (unter 20 %).

Das zeitliche Blocken von zwei oder drei Iterationen profitiert jeweils ähnlich stark (z. T. über 30 % innerhalb der Caches, über 20 % außerhalb) und am meisten vom geglätteten Datenstrom. Ausgehend vom zu erwartenden Hauptspeicherbedarf und der Anzahl an Berechnungen, dürfte bei zwei geblockten Iterationen die Geschwindigkeit vom Speicherdurchsatz abhängen, bei drei jedoch eher durch die Rechengeschwindigkeit und den Austausch zwischen den Caches bestimmt sein.

Werden mehr Iterationen zeitlich geblockt, sinkt der Geschwindigkeitsgewinn, da auch ohne das erweiterte Prefetching größere Teile vom Hauptspeicherdurchsatz unabhängig waren.

4.2.6 Padding

Bei den bisherigen Ergebnissen fallen gelegentliche, punktuelle Geschwindigkeitseinbrüche bei den mehrgitter-typischen Größen auf. Die (Halb-)Zeilenlänge liegt hier nahe an den für Cachekonflikte

anfälligen Zeilenlängen von 2^n.

Durch Padding können diese Konflikte deutlich gemindert werden, welches das Speicherlayout zwischen Ebenen und Zeilen in Vielfachen von 16 Byte ermöglicht. Eine genaue Analyse der Konflikte ist leider nicht gelungen, allerdings können günstige Padding-Werte für eine Prozessorkonfiguration experimentell bestimmt und bei der Erzeugung der Gitter genutzt werden (vgl. Tab. 4.2). Abb. 4.6 zeigt die Verbesserung von gezieltem Padding beim dreifachen zeitlichen Blocking.

Abbildung 4.6: Gezieltes Padding beim Glätter

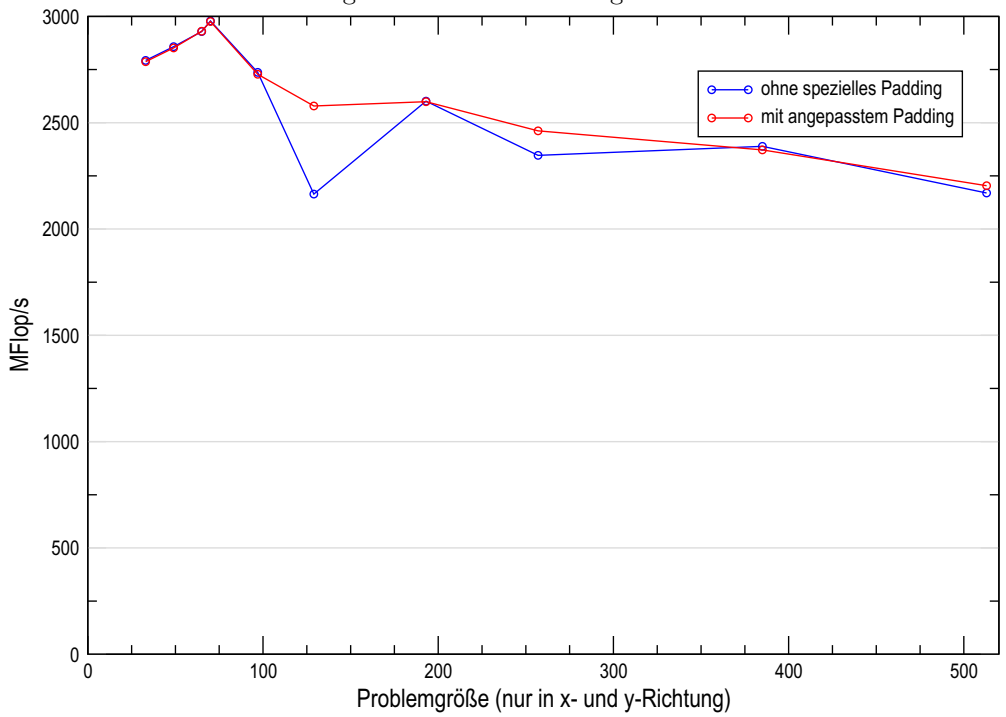


Tabelle 4.2: Padding-Tabelle für das Testsystem

	Zeitliches Blocking				
Zeilenlänge	keines	1 ×	2 ×	3 ×	4 ×
129	-	-	2	1	1
257	-	2	2	2	1
513	2	1	2	-	-

je 16 Byte

4.2.7 Überblick: Serieller Glätter

In Abbildung 4.7 sind die durch die Optimierungen jeweils auf einem Prozessor erreichbaren Geschwindigkeiten dargestellt, wobei wieder kubische Gittergrößen verwendet wurden.

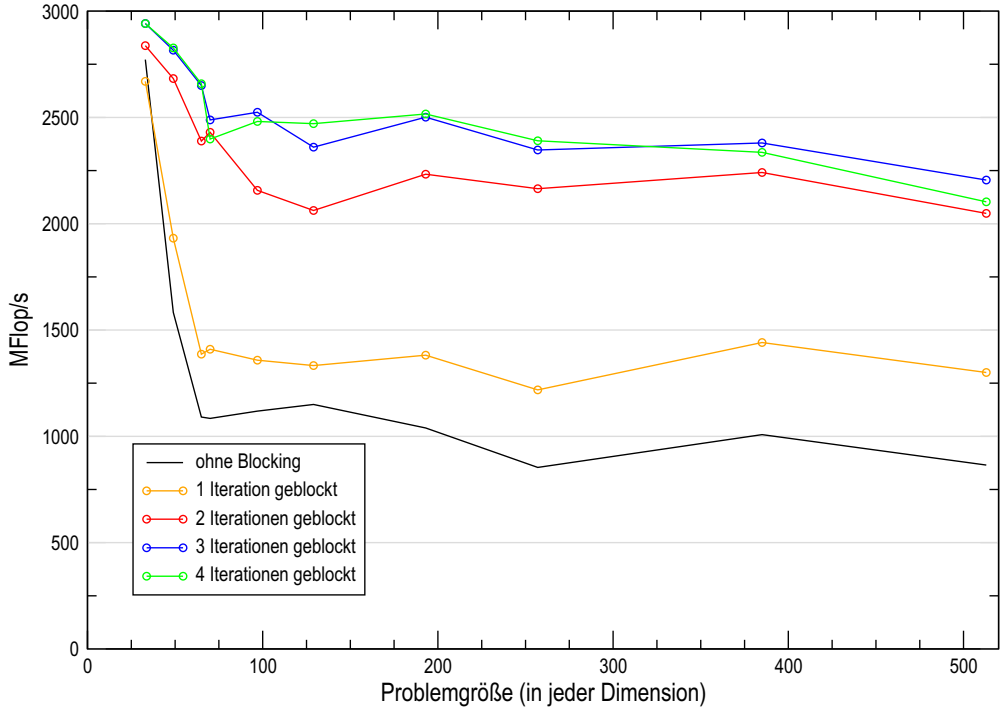
Das zeitliche Blocken der beiden Halbiterationen nur einer Iteration bringt dabei nur geringe Vorteile gegenüber einem ungeblockten Glätter, da letzterer aufgrund des Speicherlayouts pro Halbiteration weniger Transfer mit dem Hauptspeicher benötigt.

Durch das Blocken von zwei Halbiterationen konnte dem gegenüber ein großer Geschwindigkeitsgewinn erreicht werden. Durch Verwendung der Überblöcke steigt der Transfer aus dem Hauptspeicher nur leicht an, der Prozessor hat jedoch immer noch Kapazitäten jenseits der verfügbaren Operanden.

Bei drei zeitlich geblockten Iterationen ist nur noch geringer Gewinn zu erreichen, da hier die Rechengeschwindigkeit und die Bandbreite der Caches – der L2-Cache muss mit dem Prozessor und dem L3-Cache Daten austauschen – die beschränkenden Faktoren darstellen.

Beim zeitlichen Blocken von vier kann das erweiterte Prefetching gleichmässiger als bei drei Iteration verteilt werden, nämlich das Vorladen von acht Ebenen auf acht statt sechs Halbiterationen. Dennoch kann nur in Ausnahmefällen eine höhere Geschwindigkeit erreicht werden.

Abbildung 4.7: Überblick optimierter serieller Glätters



4.2.8 Parallelisierung

Müssen sich die vorhandenen Prozessoren einen Speicherbus teilen, so ist die Parallelisierung von Algorithmen nur dann effizient, wenn ein Prozessor nicht dauerhaft die volle Bandbreite nutzen kann.

Die Parallelisierung insbesondere des einfach zeitlich geblockten Glätters (siehe Abb. 4.8) bringt somit kaum Leistungszuwachs. Ab drei geblockten Iterationen war der serielle Glätter insgesamt nicht mehr durch die Speicherbandbreite begrenzt, so dass durch Parallelisierung freie Kapazitäten genutzt werden können.

Bei kleinen Ebenen ist die Parallelisierung jedoch aufgrund des verhältnismäßig großen Mehraufwandes für die automatisch eingeführten Überblöcke und die Synchronisierung nicht sinnvoll.

In Abbildung 4.9 sind die Messungen des parallelen Glätters mit ein bis vier zeitlich geblockten Iterationen dargestellt. Je weniger unabhängige Berechnungen durch den Hauptspeicher begrenzt sind, desto stärker profitieren sie von Parallelisierung.

Die beim zeitlichen Blocken von zwei Iterationen gering ausfallende Steigerung ist ein Hinweis, dass das explizite Prefetching den Durchsatz relativ gut verteilen konnte.

Abbildung 4.8: Parallelisierung des Glätters

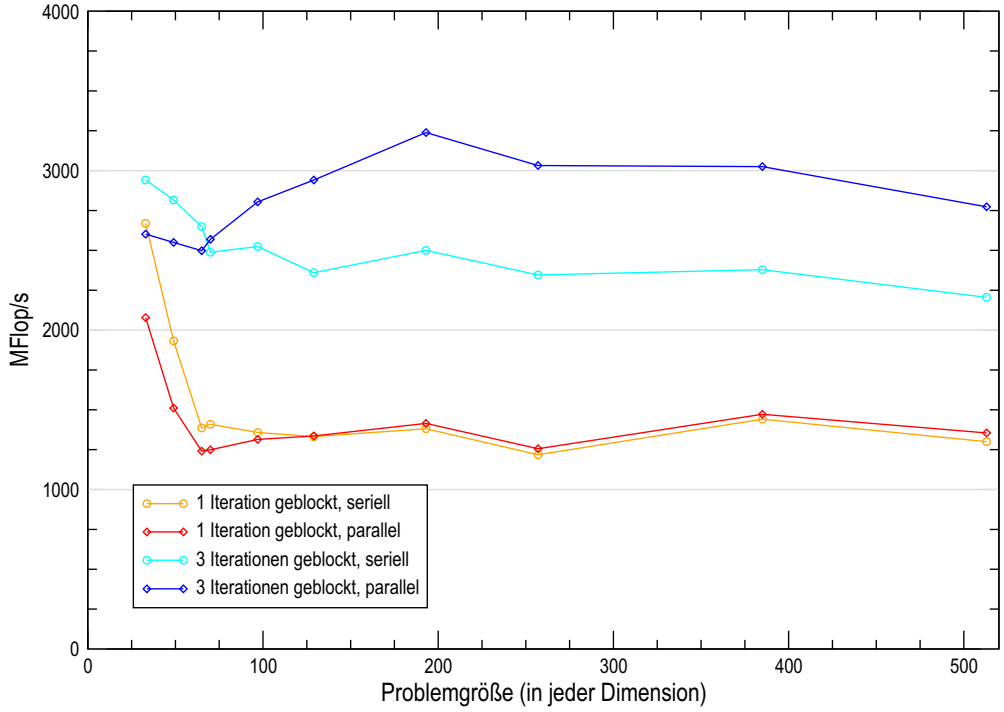
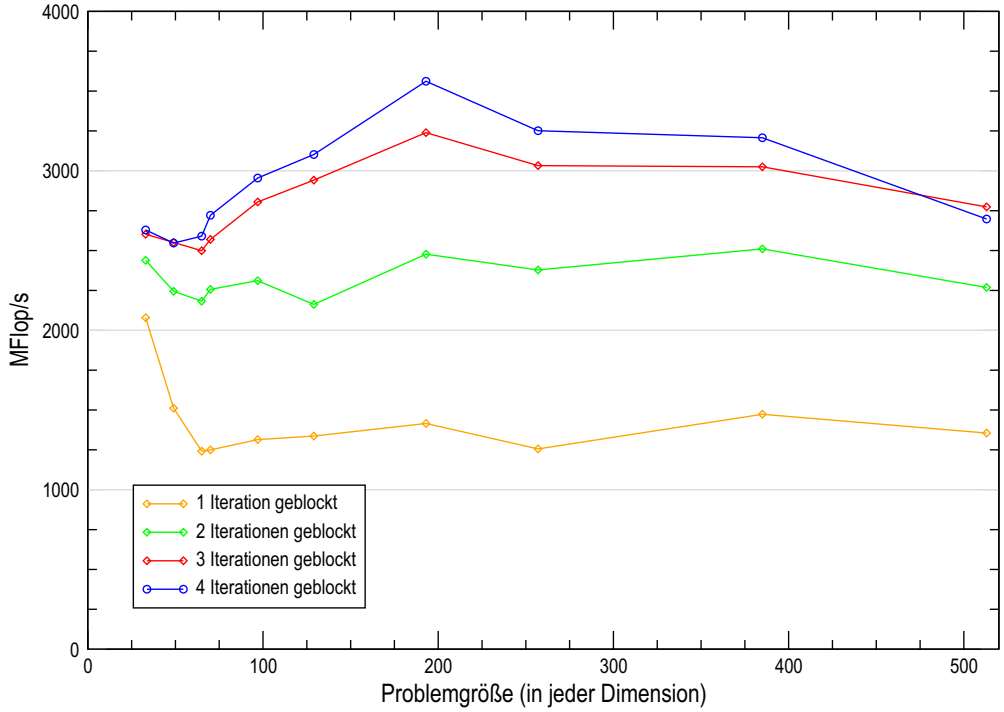


Abbildung 4.9: Überblick Ergebnisse für parallele Glätter



4.3 Optimierung des V-Zyklus

4.3.1 Ausführungsdauer

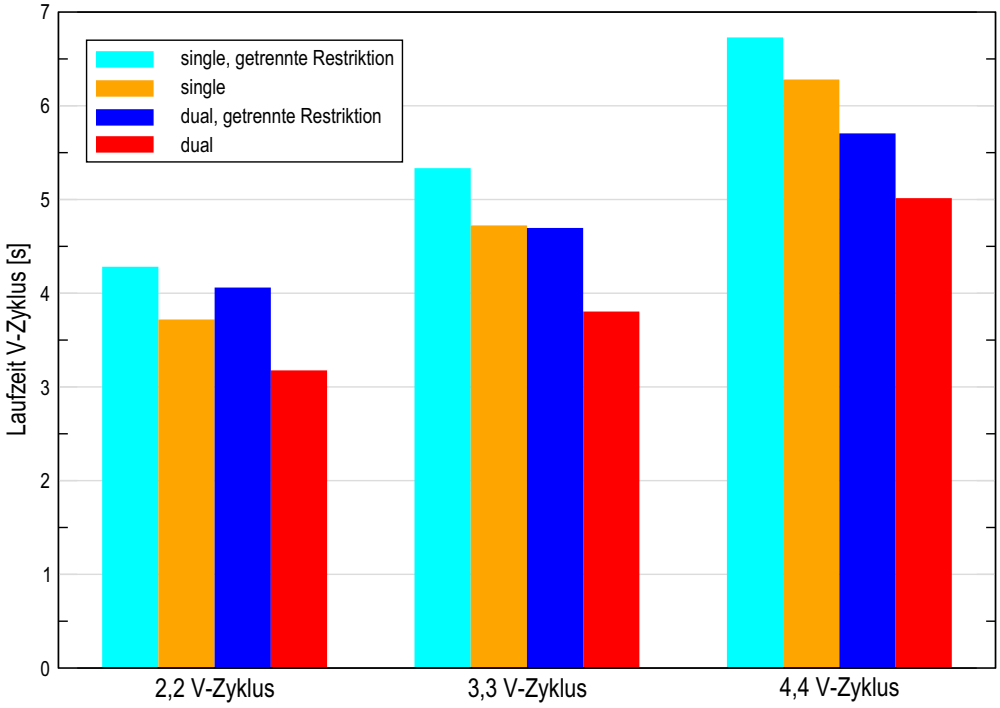
Nachdem der Glätter deutlich optimiert werden konnte, gewinnt auch die Geschwindigkeit anderer Komponenten des Mehrgitterlösers an Bedeutung. Im Abbildung 4.10 sind die Laufzeiten inklusive Residualnormberechnung für V-Zyklen mit je zwei, drei und vier Vor- und Nachglättungsschritten bei einer Problemgröße von $513 \times 513 \times 513$ aufgezeichnet. Zudem sind jeweils die serielle und parallele Version mit oder ohne in den Glätter verschmolzenen Residualberechnungen, also von Residuum und Residualnorm, gegenübergestellt.

Durch Vergleich der Zeiten kann auf den Einfluss der Faktoren geschlossen werden. Beim 2,2-Zyklus wird durch Zusammenfassung des Glätters mit den Residualberechnungen der größere Geschwindigkeitsgewinn erreicht. Diese Variante profitiert zudem mehr von der Parallelisierung: Die Residualberechnung wird wie eine weitere geblockte Halbiteration ohne erweitertes Prefetching durchgeführt. Bei einem Prozessor finden dabei kaum Zugriffe auf den Hauptspeicher statt, so dass die freie Bandbreite weiteren Prozessoren zur Verfügung steht.

Beim 3,3-Zyklus hält sich der Einfluss der beiden Optimierungen einzeln auf dem Testsystem in etwa die Waage. Die Reduzierung des Hauptspeichertransfers durch Kombination von Glätter und Restriktion auf der einen und die höhere Rechengeschwindigkeit durch Parallelisierung auf der anderen Seite wirken sich bei diesem Verhältnis von Taktfrequenz und Speicherbandbreite etwa gleich aus.

Der 4,4-Zyklus profitiert v. a. von der Parallelisierung, auch weil Zusammenfassung mit der Residualberechnung zu einer Reduzierung der Überblockhöhe führt. Insbesondere ist der Glätter hier noch stärker als beim 3,3-Zyklus durch die Rechengeschwindigkeit beschränkt, was durch die Parallelisierung gelöst werden kann.

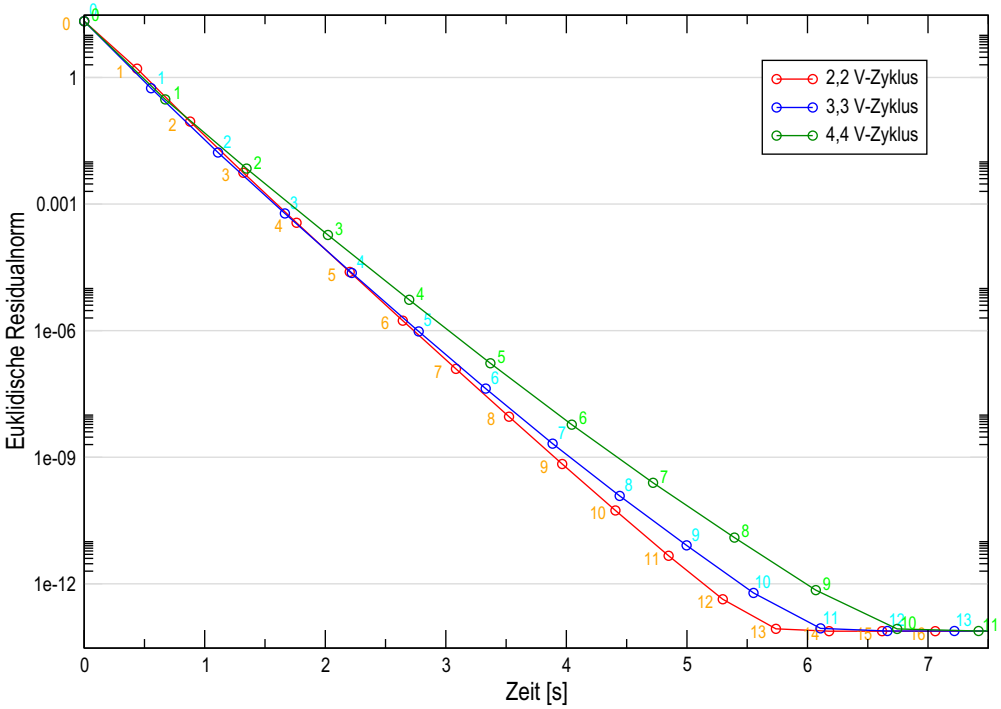
Abbildung 4.10: Laufzeiten bei Gitter $513 \times 513 \times 513$



4.3.2 Konvergenz

Bei der Anwendung des Mehrgitterlösers ist das schnelle Erreichen genauerer Lösungen entscheidend, und nicht die Anzahl dabei durchgeführter Rechenoperationen. In Abbildung 4.11 ist für ein konkretes Problem die euklidische Residualnorm gegen die Laufzeit des seriellen Mehrgitterlösers aufgezeichnet worden.

Abbildung 4.11: Konvergenz bei Gitter $257 \times 257 \times 257$, 1 Prozessor



Nachdem die optimierte Berechnung der Normen nur Residua an roten Gitterpunkten berücksichtigt, wurde außerhalb der gemessenen Zeit für den V-Zyklus die Berechnung an den schwarzen Punkten ergänzt. Unterschiede sind dabei jedoch erst bei den kleinsten erreichten Normen festzustellen, wenn die Rechenungenauigkeit eine größere Rolle spielt.

Zum Erreichen der minimalen Residualnorm werden bei zwei Vor- und Nachglättungsschritten zwölf, bei vier nur zehn V-Zyklen benötigt. Dennoch kann ein 2,2-Zyklus so schnell ausgeführt werden, dass er trotz schlechterer Konvergenzrate früher zum Ziel kommt.

Abbildung 4.12 stellt die entsprechenden Werte für den parallelen Löser dar. Die verschiedenen V-Zyklen erzielen dabei die gleichen Ergebnisse und dieselbe Konvergenz, allerdings verändert sich die Ausführungsdauer.

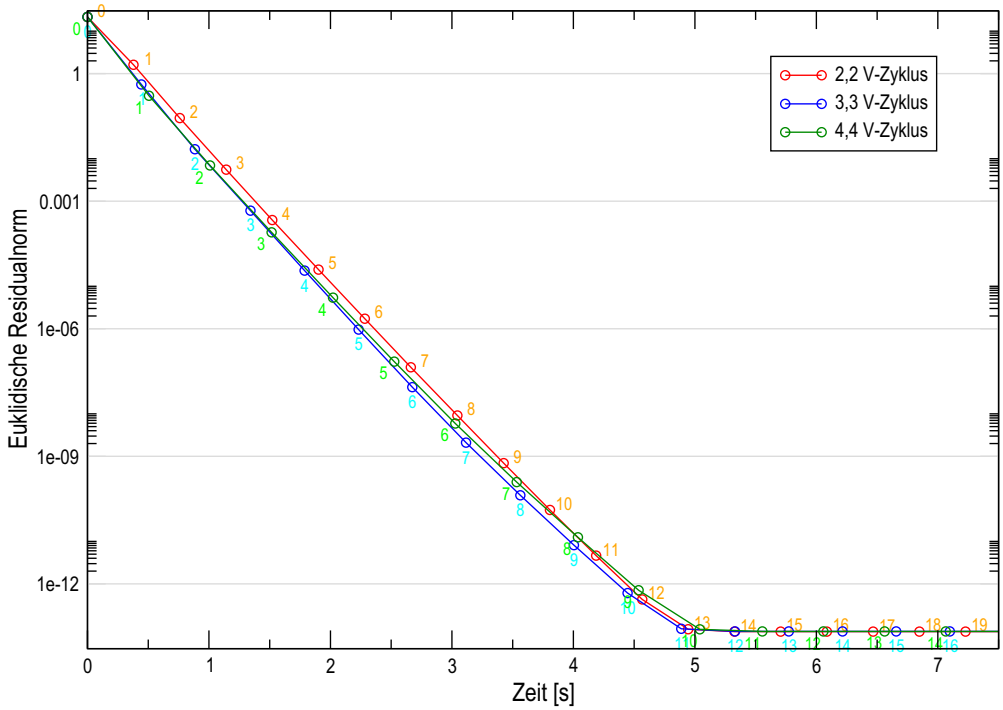
Hier können mit dem 3,3-Zyklus am schnellsten genaue Ergebnisse erreicht werden, da der 2,2-Zyklus am wenigsten und der 4,4-Zyklus nicht ausreichend von der Parallelisierung profitiert.

Die beste Wahl der Vor- und Nachglättungsschritte ist dabei ein Kompromiss zwischen Laufzeit eines V-Zyklus und erreichter Konvergenz. Während die Konvergenzrate allein von der Anzahl der Vor- und Nachglättungsschritte abhängt, wird die Ausführungsdauer durch ein Zusammenwirken von Rechenleistung und Speicherbandbreite bestimmt. Deshalb kann auf einem anderen Itanium 2-System ein anderer V-Zyklus optimal sein.

4.3.3 Verteilung der Rechenzeit

Die Verteilung der Rechenzeit auf die einzelnen Komponenten und Gittergrößen ist in den Tabellen 4.3 für den schnellsten seriellen und in 4.4 für den schnellsten parallelen V-Zyklus dargestellt. Am

Abbildung 4.12: Konvergenz bei Gitter $257 \times 257 \times 257$, 2 Prozessoren



Ende der Spalten und Zeilen sind die Anteile jeweils aufsummiert.

Die Berechnungen auf dem feinsten Gitter benötigen dabei den größten Anteil, der mit ca. 85 % sehr nahe an den abgeschätzten 87,5 % liegt. Der Vorglätter mit Residuums- und der Nachglätter mit Residualnormberechnung sind dabei 70 % der gesamten Zeit aktiv und stellen über 80 % des Aufwandes auf dem feinsten Gitter dar.

Tabelle 4.3: Verteilung der Rechenzeit in % bei 2,2 V-Zyklus, Gitter $257 \times 257 \times 257$, 1 Prozessor

	Gittergröße								
Komponente	3	5	9	17	33	65	129	257	
Vorglätter m. Residuum		$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	0,4	4,0	34,8	39,4
Restriktion		$\ll$0,1	$\ll$0,1	$\ll$0,1	$\ll$0,1	0,2	0,9	7,4	8,5
Initialisieren von V	$\ll$0,1	$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	0,4	3,8		4,2
Interpolation		$\ll$0,1	$\ll$0,1	$\ll$0,1	$\ll$0,1	0,1	1,0	8,1	9,3
Nachglätter	$\ll$0,1	$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	0,4	3,2		3,7
Nachglätter m. Normen								35,0	35,0
	$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	0,2	1,5	12,9	85,3	100,0

4.4 Abschließende Vergleiche

4.4.1 Leistungsfähigkeit des Itanium 2

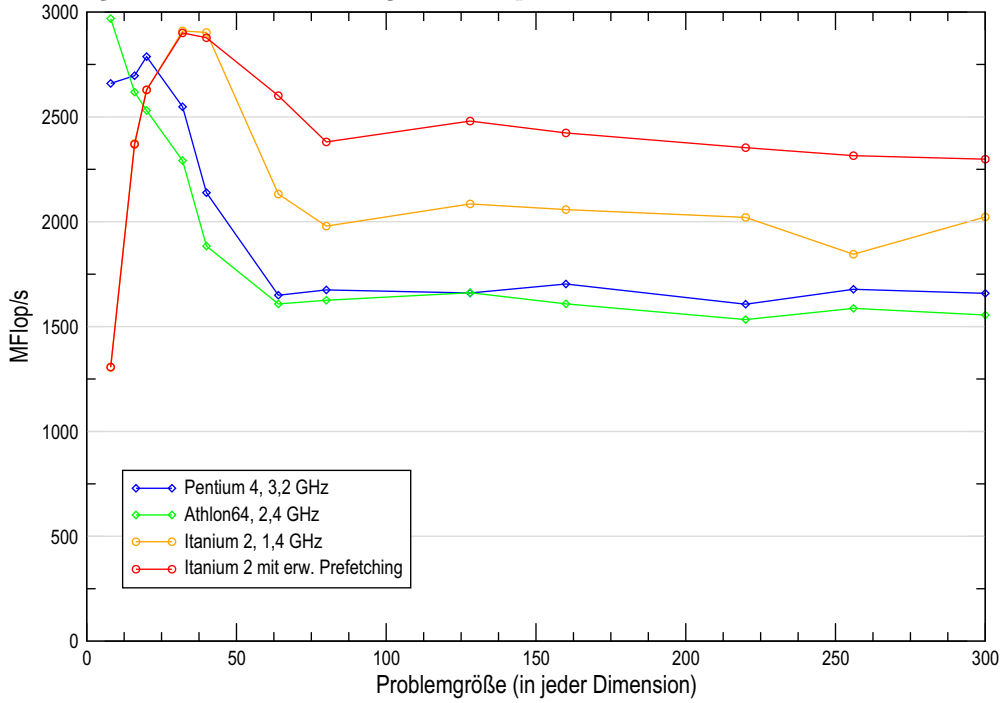
Um die Leistungsfähigkeit des Itanium 2-Prozessors einordnen zu können, wurde der auf den Itanium optimierte Glätter dem für x86-Prozessoren optimierten Red-Black Gauss-Seidel aus [Stü05] in Abbildung 4.13 gegenübergestellt. Dieser führt ein sehr ähnliches, zweistufiges Blocking-Verfahren durch. Als Vergleichssysteme dienten ein Intel Pentium 4 mit Prescott-Kern und 3,2 GHz Taktung sowie ein AMD Athlon64 mit 2,4 GHz. Die Geschwindigkeit wurde jeweils für das zeitliche Blocking von drei Iterationen gemessen.

Tabelle 4.4: Verteilung der Rechenzeit in % bei 3,3 V-Zyklus, Gitter $257 \times 257 \times 257$, 2 Prozessoren

Komponente	**Gittergröße** 3	5	9	17	33	65	129	257	
Vorglätter m. Residuum		$\ll$0,1	$\ll$0,1	<0,1	0,1	0,7	3,8	34,9	39,5
Restriktion		$\ll$0,1	$\ll$0,1	<0,1	$\ll$0,1	0,2	0,9	7,3	8,3
Initialisieren von V	$\ll$0,1	$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	0,4	3,8		4,3
Interpolation		$\ll$0,1	$\ll$0,1	$\ll$0,1	$\ll$0,1	0,1	1,1	8,2	9,3
Nachglätter	$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	<0,1	0,6	3,4		4,1
Nachglätter m. Normen								34,4	34,4
	$\ll$0,1	$\ll$0,1	$\ll$0,1	<0,1	0,2	1,9	13,0	84,8	100,0

Aufgrund der Beschränkungen des Vergleichslösers wurde in jeder Dimension eine gerade Anzahl von Unbekannten gewählt, so dass die Ergebnisse für den Itanium 2 zum Teil minimal besser als bei den vorherigen Tests ausfallen.

Abbildung 4.13: Itanium 2 im Vergleich bei optimierten Red-Black-Gauss-Seidel-Glättern



Beide x86-Prozessoren besitzen eine komplexe Out-of-Order-Logik, die die komplizierten Instruktionen des Befehlssatzes intern in einfache Befehle zerteilt und diese bei auftretenden Latenzen aggressiv umordnen und spekulativ ausführen kann. Bei sehr kleinen Problemgrößen kann sie eine deutlich höhere Geschwindigkeit als der In-Order-Kern des Itanium 2 erreichen.

Die Caches der drei Prozessoren unterscheiden sich dabei nicht nur in ihrer Größe (siehe Tabelle 4.5), Bandbreite und Latenz, sondern auch in ihrer Anbindung. Der Athlon64 nutzt für alle Daten primär den L1-Cache, in den L2-Cache gelangen nur geänderte oder aus dem L1-Cache verdrängte Zeilen. Die Caches haben zudem eine symmetrische Bandbreite.

Der L1-Cache des Pentium 4 spielt eine weniger zentrale Rolle. Er arbeitet als Write-Through-Cache, so dass die Schreibbandbreite vom L2-Cache abhängt – Store Misses werden sogar direkt an den L2-Cache weitergereicht und führen zu keiner Einlagerung. Zudem besitzt die Gleitpunkteinheit – ähnlich zum Itanium – spezielle Datenleitungen in den L1- und L2-Cache, durch die ausgerichtete 16-Byte-Vektoren mit doppelter Bandbreite geladen werden können.

Durch diese Ladeoperationen erreicht der Pentium 4 auch eine deutlich höhere Bandbreite zu den

Tabelle 4.5: Vergleich der Caches: Pentium 4, Athlon64 und Itanium 2

Cache	Pentium 4	Athlon64	Itanium 2
L1 data	8 kB	64 kB	16 kB [1]
unified L2	1 MB	1 MB	256 kB
unified L3	–	–	1,5 MB

[1] wird von FPU nicht genutzt

Caches als der Athlon64. Der Athlon64 hat jedoch eine effektivere Speicheranbindung und erreicht beim Blocken weniger Iterationen die höhere Geschwindigkeit.

Sobald nicht mehr wenigstens die Daten der geblockten Halbiterationen im L1-Cache gehalten werden können, sinkt die Geschwindigkeit der x86-Prozessoren deutlich ab. Ab einer Größe von ca. $40 \times 40 \times 40$ können die Felder nicht mehr vollständig in den Caches gehalten werden und die Geschwindigkeit bleibt durch das Blocking annähernd konstant.

Der Itanium 2 steigert seine Geschwindigkeit hingegen mit zunehmender Problemgröße, solange alle Daten im L3-Cache gehalten werden können. Durch die Cachehierarchie nimmt zwar die Latenz zu, die Bandbreite aber kaum ab.

Während beide x86-Prozessoren Hardware-Prefetcher einsetzen und ihr Verhalten durch Prefetch-Instruktionen nur bedingt geändert werden kann, kann beim Itanium 2 eine gute Kontrolle über den Aufenthalt der Daten in den Caches ausgeübt werden.

4.4.2 Vergleich mit anderen Mehrgitterlösern

Tabelle 4.6 zeigt Eckdaten einiger am Lehrstuhl für Systemsimulation entwickelten Mehrgitterlöser, die auf dem Test-Itanium 2 gewonnen wurden. Mit Ausnahme eines weiteren experimentellen Lösers handelt es sich dabei um Codes zum praktischen Einsatz.

Tabelle 4.6: Vergleich mit anderen Mehrgitterlösern

Code	Zeit / V-Zyklus [s]	Konvergenzrate	Speicherbedarf
Optimiert [1]	0,058	0,03 - 0,08	38 MB
HHG	0,251	0,05 - 0,08	86 MB
ParExPDE	0,88	0,032	1.3 GB
Expde3D	25,8		750 MB
„Nil's Code“	0,22	0,12	

[1] 3,3 V-Zyklus parallel

HHG steht für Hybrid Hierarchical Grids und stellt eine Bibliothek zum Diskretisieren und Lösen von Partiellen Differentialgleichungen über Finite Elemente zur Verfügung. Dabei dienen unstrukturierte Gitter als Ausgangspunkt und werden auf eine strukturierte Art und Weise verfeinert. Eine parallele Ausführung ist über die Message Passing Infrastructure (MPI) möglich.

Expde3D ist ein Framework zum Lösen von Partiellen Differentialgleichungen über Expression Templates auf block-strukturierten Gittern und stellt auch Multigrid-Operationen zur Verfügung.

ParExPDE ist die parallelisierte Weiterentwicklung von Expde3D.

„Nil's Code“ ist ein experimenteller Mehrgitterlöser in Fortran, der im Rahmen der Studienarbeit [Thü02] entstanden ist.

Die großen Unterschiede sind hier nicht Zeichen unterschiedlicher Qualität, sondern entstammen den unterschiedlichen Zielsetzungen. Dabei stellt jedes Ergebnis einen Kompromiss dar; Allgemeinheit, Skalierung bei großen Problemen, Benutzerfreundlichkeit, Portierbarkeit und Wartbarkeit sind nur einige der nicht in der Tabelle aufgeführten Faktoren.

Kapitel 5

Ausblick

Die Arbeit konnte am sicherlich extremen Beispiel des Itanium 2 das hohe Potential hardwareorientierter Optimierungen demonstrieren. Zumindest auf der IA-64-Architektur scheinen dreidimensionale Probleme auch gut beherrschbar zu sein.

Mit ein Grund dafür ist die gute Kontrolle über Speicherlokalität und Prefetching, durch die Latenzen sehr gut verborgen und Speicherschübe vermieden werden können. In diesem Maße bietet dies derzeit jedoch keine andere verbreitete Architektur.

Etlichen Aspekten konnte leider nicht die nötige Aufmerksamkeit geschenkt werden: So sind u. a. die Ursachen für bestimmte Cachekonflikte nicht im Detail geklärt. Auch konnte keine Untersuchung oder gar Optimierung des Datenverkehrs innerhalb der Caches durchgeführt werden.

Vollkommen ausgelassen wurde die Suche nach Strategien zur effizienten Parallelisierung bei verteiltem Speicher, sei es auf Clustern oder Shared-Memory-Systemen. Erst dann ist es sinnvoll, den noch experimentellen Code zu einem vollwertigen Löser auszubauen.

Für die meisten Anwendungen ist eine ähnlich zeitintensive Optimierung nicht durchführbar, so dass vieles dem Compiler überlassen werden muss. Neuere Programmiersprachen wie Java entfernen sich jedoch weiter von der ausführenden Hardware und beschäftigen sich insbesondere mit Abstraktion und der Beherrschung großer Softwareprojekte. Möglicherweise erzwingen jedoch neue Entwicklungen der Chip-Architektur ein hardware-orientierteres Programmier-Paradigma.

Die derzeitige Entwicklung geht dort zur Integration mehrerer Prozessorkerne auf einen Chip. Dabei werden zum einen mehrere identische Recheneinheiten gruppiert, wie bei den aktuellen Prozessorgenerationen der x86-Architektur und dem bald erscheinenden Itanium 2 Montecito. Andere Hersteller stellen einem Allzweckprozessor auch spezialisierte Koprozessoren zur Verfügung, wie dies beim von IBM, Sony und Toshiba entwickelten Cell-Prozessor der Fall ist.

Hier stellt sich die Frage, ob mit heute gebräuchlichen Programmiersprachen parallele Ausführung der Normalfall werden kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BHM00] Briggs, W. L., V. E. Henson und S. F. McCormick: *A Multigrid Tutorial*. SIAM, 2 Auflage, 2000.

[Int01a] Intel Corporation: *Intel Itanium Architecture Assembly Language Reference Guide*, 2001. ftp://download.intel.com/software/opensource/tools/ia-64/asm_lan.pdf.

[Int01b] Intel Corporation: *Intel Itanium Processor Reference Manual for Software Developement and Optimization*, November 2001.

[Int01c] Intel Corporation: *Intel Itanium Software Conventions & Runtime Architecture*, Mai 2001. ftp://download.intel.com/design/Itanium/Downloads/24535803.pdf.

[Int03] Intel Corporation: *IA-64 Assembler – User's Guide*, 2003. ftp://download.intel.com/design/Itanium/Downloads/asmusrgd.pdf.

[Int04] Intel Corporation: *Intel Itanium 2 Processor Reference Manual for Software Developement and Optimization*, Mai 2004. ftp://download.intel.com/design/Itanium2/manuals/25111003.pdf.

[Int06a] Intel Corporation: *Applicationn Architecture*. In: *Intel Itanium Architecture Software Developer's Manual*, Nummer 1 in *3*. 2006. ftp://download.intel.com/design/Itanium/manuals/24531705.pdf.

[Int06b] Intel Corporation: *Instruction Set Reference*. In: *Intel Itanium Architecture Software Developer's Manual*, Nummer 3 in *3*. 2006. ftp://download.intel.com/design/Itanium/manuals/24531905.pdf.

[Int06c] Intel Corporation: *System Architecture*. In: *Intel Itanium Architecture Software Developer's Manual*, Nummer 2 in *3*. 2006. ftp://download.intel.com/design/Itanium/manuals/24531805.pdf.

[Kow04] Kowarschik, M.: *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. Doktorarbeit, Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, Juni 2004.

[Pfä00] Pfänder, H.: *Cache-optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern*. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, Januar 2000. Diplomarbeit.

[Stü05] Stürmer, M.: *Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren*. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, August 2005. Studienarbeit.

[TGF06] Treibig, J., T. Gradl und Ch. Freundl: *Comparison of FEM-Codes at LSS.* Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, April 2006. Vortrag.

[Thü02] Thürey, N.: *Cache Optimizations for Multigrid in 3D.* Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, Juni 2002. Studienarbeit.

[Wei01] Weiss, C.: *Data Locality Optimizations for Multigrid Methods on Structured Grids.* Doktorarbeit, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, Dezember 2001.