Boosting

Alexander Leidinger

Zwei Artikel von Michael D. Smith, Mark Horowitz, Monica S. Lam
Computer Systems Laboratory
Stanford University

26. Januar 2000

Zusammenfassung

Es wird ein superskalarer Prozessor beschrieben, der die positiven Eigenschaften von statischem und dynamischem Scheduling vereint, um die Leistung nicht numerischer Programme zu erhöhen. Um die Fähigkeit des Compilers, effektiv Befehle über viele Basic-Blocks hinaus zu schedulen, zu nutzen, wird bei dieser Architektur statisches Scheduling benutzt. Da bedingte Sprünge in nicht numerischem Code stark datenabhängig sind, wird hier das Konzept *geboosteter* Befehle eingeführt, deren Ergebnisse abhängig von später auszuführenden Sprungbefehlen akzeptiert werden. Boosting entfernt effektiv die Abhängigkeiten von Sprüngen und macht das Scheduling von Befehlen mit Nebeneffekten so einfach wie das von Befehlen ohne Nebeneffekte. Zur Effizienzsteigerung wird Boosting von der Hardware durch *Schattenstrukturen* unterstützt, welche vorübergehend die Nebeneffekte geboosteter Befehle puffern, bis die bedingten Sprünge von denen die geboosteten Befehle abhängen ausgeführt wurden. Dann werden die gepufferten Nebeneffekte entweder verworfen oder akzeptiert. Die erzielte Leistung durch diese Art von statischem Scheduling ist vergleichbar mit der von dynamischem Scheduling.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Da die Anzahl der parallel ausführbaren Befehle in einem Basic-Block (Befehle zwischen zwei Sprüngen) klein ist, müssen superskalare Prozessoren über diese Blockgrenzen hinausschauen, um die Leistung zu erhöhen. In numerischem Code können Schleifen, welche eine große Prozentzahl aller ausgeführten Sprünge ausmachen, früh erkannt und die Parallelität erhöht werden. Da jedoch viele der Sprünge in nicht numerischem Code datenabhängig sind, können diese nicht frühzeitig vorhergesagt werden. Deshalb ist spekulative Ausführung von Befehlen eine wichtige Quelle von Parallelität in dieser Art von Code.

Befehlsparallelität erreicht man statisch (zur Compilezeit) oder dynamisch (zur Laufzeit). Statisches Scheduling erreicht man mit einem geringen Hardwareaufwand durch Berücksichtigung der Parallelität im Befehlssatz. Für numerische Programme benutzt der Compiler Techniken wie z.B. Softwarepipelining um effizienten Code zu erzeugen, was bei nicht numerischen Programmen nicht zum gewünschten Ergebnis führt. Dynamisch geschedulte superskalare Prozessoren hingegen benötigen einen großen (und damit teuren) Hardwareaufwand um effektiv zu arbeiten.

Um die Leistung nicht numerischer Programme mit vernünftigen Hardwarekosten zu erhöhen, wird eine superskalare Architektur vorgestellt (TORCH genannt), die die Vorteile statischen und dynamischen Schedulings vereint. Der Vorteil statischen Schedulings liegt in der Fähigkeit des Compilers Befehle über viele Basic-Blocks hinweg zu Schedulen. Deshalb wird das Befehlsscheduling bei TORCH vom Compiler ausgeführt. Der Vorteil von dynamischem Scheduling ist die Fähigkeit der spekulativen Ausführung von Befehlen. Darum hat TORCH Hardware, die die Ausführung von Befehlen vor dem Sprung erlaubt, die erst nach dem Sprung ausgeführt würden, eine Operation die hier Boosting genannt wird. Boosting macht agressives Scheduling im Compiler einfach, da Nebeneffekte, die von den bedingten Sprüngen abhängen, umgangen werden.

Um effizient zu boosten, enthält die TORCH Hardware zwei Schattenstrukturen: das shadow Registerfile und den shadow Storebuffer. Diese Strukturen puffern die Nebeneffekte geboosteter Befehle bis die zugehörigen Sprünge abgearbeitet sind. Bei einem korrekt vorhergesagten Sprung akzeptiert die Hardware die entsprechenden Ergebnisse in den Schattenstrukturen. Bei inkorrekt vorhergesagten Sprüngen garantiert die Hardware korrekte Ergebnisse durch verwerfen der nicht benötigten Ergebnisse geboosteter Befehle.

2 Spekulative Ausführung von Befehlen

Spekulative Ausführung ist notwendig, da bedingte Sprünge eine Ausführungsabhängigkeit der nachfolgenden Befehle aufzwingen. Zum Beispiel sind die Befehle in den THEN und ELSE Teilen eines IF Befehles abhängig von der IF Bedingung, da wir nicht herausfinden können, ob der THEN oder ELSE Befehl ausgeführt werden soll, bis die Bedingung der IF Anweisung getestet wurde. Spekulative Ausführung erlaubt die Ausführung des THEN und/oder ELSE Befehles vor der Überprüfung der Bedingung der IF Anweisung.

2.1 Sichere Ausführung spekulativer Befehle

Spekulative Ausführung erhält jedoch leider nicht immer die Korrektheit eines Programmes, so daß man 4 Arten von spekulativer Ausführung unterscheiden muß.

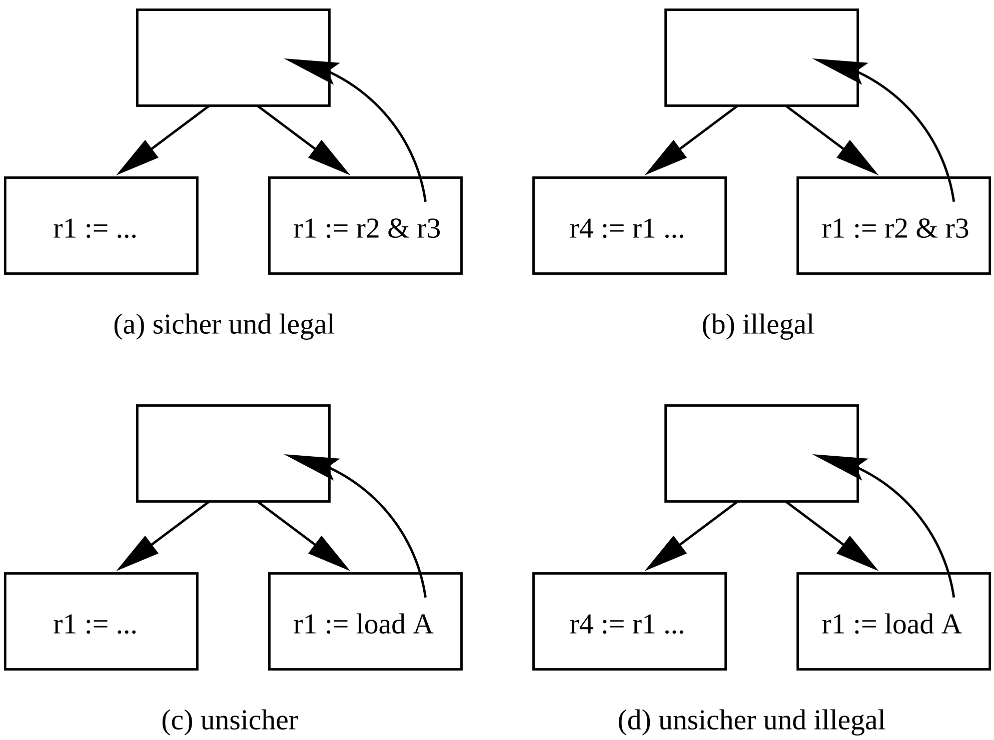

Abbildung 1: Typen spekulativer Ausführung

Für einen spekulativen Befehl kann ein Sprung entweder korrekt oder falsch vorhergesagt werden. Einen Sprung nennt man für einen spekulativen Befehl korrekt vorhergesagt, wenn der Block von der der spekulative Befehl geboostet wurde, nach dem Sprung ausgeführt wird. Andernfalls nennt man den Sprung falsch vorhergesagt. Einen spekulativen Befehl nennt man illegal, wenn diese Operation einen Wert überschreibt, der benötigt wird, wenn der Sprung falsch vorhergesagt ist (Bild 1b). Unsicher nennt man einen spekulativen Befehl, wenn dieser eine Exception verursachen kann. Diese Exception sollte nur auftreten, wenn der Sprung korrekt vorhergesagt ist (Bild 1c [address-exception]). Offensichtlich kann ein spekulativer Befehl unsicher und illegal sein (Bild 1d). Zur korrekten Programmausführung sollten nur sichere und legale spekulative Befehle ausgeführt werden, was durch statisches Scheduling erreicht wird. Ein Compiler kann zwar manche illegale spekulative Befehle, durch Verwendung anderer Register, die nicht zu Konflikten führen, in legale spekulative Befehle umwandeln, jedoch nicht unsichere in sichere. Deshalb wird Hardwareunterstützung benötigt. Die Basis der momentan verwendeten Techniken beruht auf der Hardwarepufferung der Effekte spekulativer Befehle.

Der *sequenzielle Status* der Maschine ist definiert als der Maschinenstatus, der nicht ein Ergebnis eines spekulativen Befehles ist. Der *spekulative Status* demzufolge der Maschinenstatus, der das Ergebnis eines spekulativen Befehles ist. Die Pufferung trennt den sequenziellen und spekulativen Status und gewährt einen Aufschub der spekulativen Nebeneffekte bis der Sprung/die Sprünge ausgeführt wurde(n). Wenn alle Sprünge von denen ein spekulativer Befehl abhängt ausgeführt wurden, müssen Hardwaremechanismen den sequenziellen Status der Maschine mit den spekulativen Effekten dieses Befehles aktualisieren, andernfalls verwirft die Maschine einfach den spekulativen Status und Nebeneffekte des Befehles.

2.2 Boosting

Eine Boosting Architektur enthält Hardwarepuffer, die unsichere und illegale spekulative Befehle in sichere und legale umwandelt. Weiterhin behandelt Boosting alle Aspekte von Exceptionhandling durch die Bereitstellung eines Exceptionmechanismusses für alle Operationen.

Nur die Hardware kann effektiv *alle* Effekte einer spekulativen Operation puffern. Damit die Boostinghardware sicherstellen kann, daß die Semantik des Programmes nicht verändert wird, muß der Compiler seine Annahmen der Hardware mitteilen. Deshalb betitelt der Compiler jeden spekulativen Befehl als *geboosteten Befehl*. Diese Betitelung kodiert die benötigten Informationen, so daß die Hardware unterscheiden kann, wann die Effekte eines geboosteten Befehles nicht länger spekulativ sind. Die Betitelung zeigt von welchem Sprung der geboostete Befehl abhängt und welches die vorhergesagte Sprungrichtung ist. Im Beispiel in Bild 2 zeigt ein „B“ an, daß es sich um einen geboosteten Befehl handelt, wobei ein angehängtes „R“ oder „L“ die Sprungrichtung angibt, von der der geboostete Befehl abhängt.

Die allgemeinste Art von Boosting benötigt Hardware exponentiell zum Bostinglevel, da für jeden Sprungweg der spekulative Status der Maschine benötigt

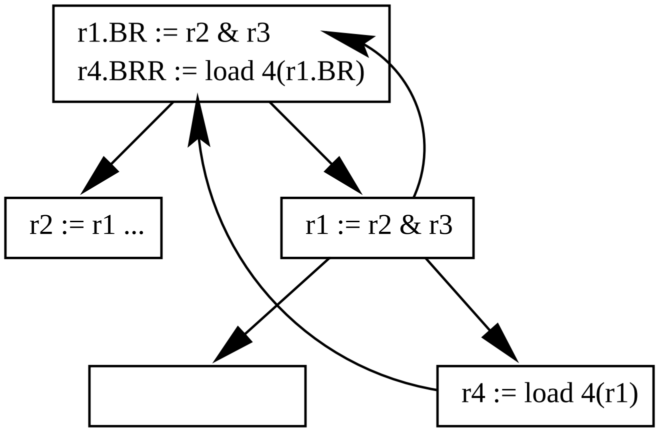

Abbildung 2: Beispiel geboosteter Befehle

wird. Um die Hardware auf einem vernünftigen Niveau zu halten, boostet man nur Befehle der meist genutzten Richtung eines Sprunges (z.B. durch Profiling, wird im folgenden vorausgesetzt). Dadurch können die Sprungbefehle die Vorhersageinformation (Sprung wird ausgeführt oder nicht) kodieren, und jeder geboostete Befehl kann einfach anzeigen, daß er von den nächsten n bedingten Sprüngen abhängt (z.B. der 2. spekulative Befehl in Bild 2 vereinfacht sich von „.BRR“ zu „.B2“). Um das ganze weiter zu vereinfachen geht man davon aus, daß der geboostete Befehl kontrollabhängig von allen Sprüngen ist, über die er geboostet wurde. Durch Kodierung des Boostinglevels als Anzahl der kontrollabhängigen Sprünge kann die Hardware die Kontrollabhängigkeitsinformation rekonstruieren. D.H. ein geboosteter Befehl des Levels n is kontrollabhängig von der Ausführung der nächsten n bedingten Sprünge, und das Ergebniss des geboosteten Befehles wird nur akzeptiert, wenn alle n bedingten Sprünge korrekt vorhergesagt wurden. Diese Bedingung macht es einfacher, die Boostinginformationen zu kodieren und die Hardware zu bauen.

Spekulative Exceptions werden gepuffert bis feststeht, ob der Sprung korrekt vorhergesagt wurde oder nicht. Je nachdem wird dann der Befehl, der die Exception verursachte, erneut ausgeführt und dadurch eine sequenzielle Exception ausgelöst. Eine genauere Beschreibung spekulativer Exceptions befindet sich in der Doktorarbeit von M. D. Smith, Stanford Universität.

3 Notwendige Hardware für Boosting

3.1 Komplette Unterstützung von Boosting

In der MIPS Architektur, auf der hier Boosting aufbaut, sind 3 mögliche Nebeneffekte zu beachten: ein Register wird beschrieben, ein Speicherbereich wird beschrieben, oder eine Ausnahme wird signalisiert. Deshalb werden Schattenstrukturen für das Registerfile und den Storebuffer benötigt (Ausnahmebehandlung siehe Abschnitt 2.2).

Im einfachsten Fall kann man sich die spekulativen Teile der Schattenstrukturen als Kopien der sequenziellen Teile vorstellen. In diesem Fall benötigt man jedoch für jeden Boostinglevel einen eigenen spekulativen Hardwareeintrag (Bild 3b). Obwohl diese Art des Boostings viel Hardware benötigt, ist sie einfach zu implementieren.

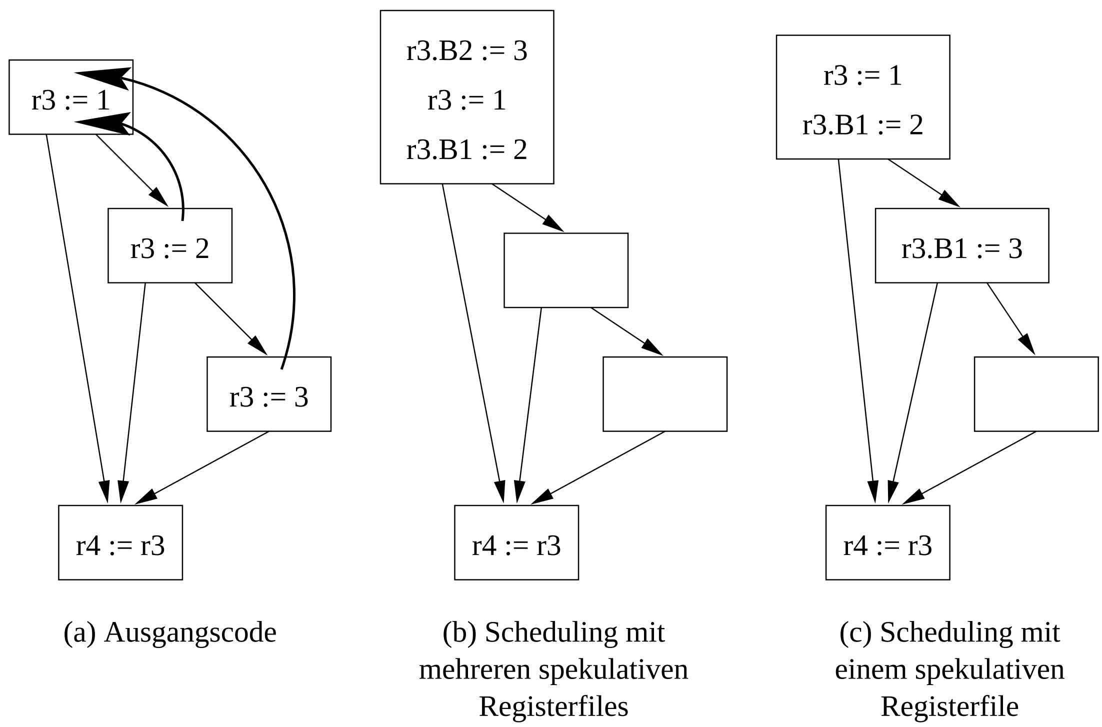

Abbildung 3: Verschiedene Registerfilekonfigurationen

Da es ausreicht, die Daten lediglich logisch in den sequenziellen Zustand zu kopieren, kann man eine Technik anwenden, die ähnlich zum Registerrenaming ist. Für jedes sequenzielle Register erzeugt man eine Liste bestehend aus Register, Zähler und einem Belegt-Bit, wobei der Zählerwert den Boostinglevel angibt. Ein Boostinglevel von 0 ist hierbei der sequenzielle Status. Bei einem geboosteten Befehl wird ein freier Eintrag als Belegt gekennzeichnet und in den Zähler wird der Boostinglevel eingetragen. In das Register kommt nach Abarbeitung des Befehles das Ergebnis. Nach einem Sprung wird jeder Zähler dekrementiert, wobei darauf geachtet werden muß, daß der Zähler des sequenziellen Registers nur heruntergezählt wird, wenn der Boostinglevel 1 Daten enthält (Belegt-Bit gesetzt). Enthält er diese nicht, bleibt der Zähler des sequenziellen Registers auf 0 und das Belegt-Bit des Boostinglevels 1 wird gelöscht.

3.2 Partielle Unterstützung von Boosting

Da die komplette Unterstützung von Boosting einen nicht unerheblichen (und damit teuren) Hardwareaufwand mit sich bringt, lohnt es sich mehrere Optionen der partiellen Unterstützung von Boosting anzusehen.

In Option 1 wird der spekulative Storebuffer entfernt. Somit kann der Scheduler keine Speicherbefehle boosten, was auch Berechnungen beinhaltet, welche ein Ergebnis aus dem Speicher laden, welches vorher durch einen anderen Befehl dort gespeichert wird. Unter der Annahme, daß genug Register vorhanden sind und der Compiler einen effektiven Registerallocator hat, sollten solche Abläufe selten auftreten, weswegen die Einbußen dieser Einschränkung minimal sein sollten.

In Option 2 werden die verschiedenen spekulativen Registerfiles in einem einzelnen spekulativen Registerfile zusammengelegt, welches in der Lage ist mehrere Bosstinglevel zu handhaben. Ohne einen unterschiedlichen Speicherplatz für jeden möglichen Boostinglevel beziehen sich r3.B1 und r3.B2 in Bild 3b auf den selben physikalischen Speicherplatz. Der Compiler muß deshalb Code für Bild 3c generieren. Wenn diese Art der Abhängigkeit selten auftritt, bzw. die geringe Überlappung wie in Bild 3c ausreicht, dann sollten die Einbußen dieser Einschränkung ebenfalls minimal sein.

Option 2 reduziert signifikant den Hardwareaufwand, den man für mehrere Boostinglevel benötigt. Sie braucht lediglich zwei Register, einen Zähler und ein Valid-Bit für jedes sequenzielle Register. Bild 4 zeigt wie die dafür benötigte Hardware verschaltet ist.

Der Zähler handhabt den aktuellen Boostinglevel des Wertes im spekulativen Register. Dieser Zähler wird jedesmal dekrementiert, wenn ein korrekt vorhergesagter Sprung ausgeführt wird. Steht der Zähler auf eins und das Register enthällt korrekte Daten wird beim nächsten korrekt vorhergesagten Sprung aus dem sequenziellen das spekulative Register und umgekehrt. Diese Hardware implementiert den spekulativen Status und fügt lediglich ein einzelnes Gatter dem Registerzugriffspfad hinzu.

In Option 3 wird das komplette spekulative Registerfile entfernt. Um trotzdem Befehle boosten zu können, wird die Pipelinekontrolle erweitert, so daß der Scheduler in den „Schatten" des bedingten Sprunges hinein boosten kann. Dieses ist eigentlich eine Erweiterung von „squashing branches" bei denen Befehle in den dem Sprung nachfolgenden Zyklen verworfen werden, wenn der Sprung falsch vorhergesagt wurde. Hier werden in solchen Fällen jedoch lediglich geboo-

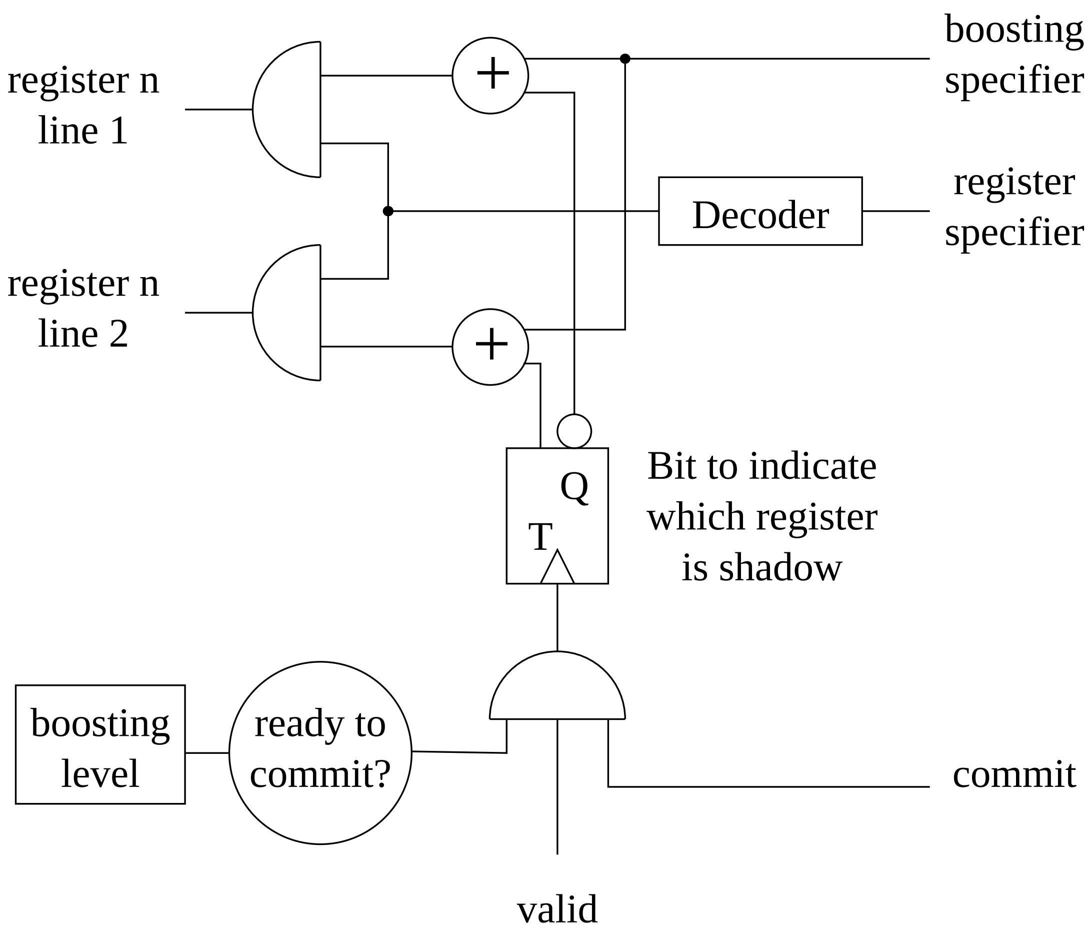

Abbildung 4: Hardware der Option 2

stete Befehle verworfen. Dieses erfordert den geringsten Hardwareaufwand, hat jedoch die größten Restriktionen.

4 TORCH

TORCH basiert auf der MIPS R2000 RISC Architektur. Wie der R2000 benutzt TORCH verzögerte Sprünge um die Latenz zwischen der Ausführung des Sprunges und dem Fetchen des ersten Befehles des Sprungzieles zu minimieren. Weiterhin codiert TORCH die statische Sprungvorhersagenformation in jeden bedingten Sprung. Die Sprungvorhersage in TORCH basiert auf Sprung-Profile-Statistiken, weil die Genauigkeit der Sprungvorhersage direkten Einfluß auf die Leistung hat. Profiling zeigt den meist genutzten Sprungpfad, dessen Befehle von TORCH um lediglich einen Level geboostet werden. Außerdem minimiert bei TORCH der Compiler die Effekte unsicherer Befehle durch Registerbelegung während dem Befehlsscheduling.

4.1 Hardware

TORCH besteht aus zwei Einheiten, eine für Integer- und Logikoperationen und eine für Fließkommaberechnungen. Beide enthalten jeweils eine Schattenstruktur, welche aus dem sequenziellen und dem spekulativen Registerfile besteht, wobei jedes Registerfile den entsprechende Status der Maschine enthält. Befindet sich ein Befehl gerade in der Pipeline wenn die Bedingung des dazugehörigen Sprunges feststeht, so wird, wenn der Sprung korrekt vorhergesagt wurde, das Ziel des geboosteten Befehles das sequenzielle Register, oder wenn es falsch vorhergesagt wurde wird der Befehl verworfen. Desweiteren existiert ein shadow Storebuffer, welcher einen sequenziellen und einen spekulativen Teil enthält. Dieser ist für die spekulative Ausführung von Schreibzugriffen auf den Speicher zuständig

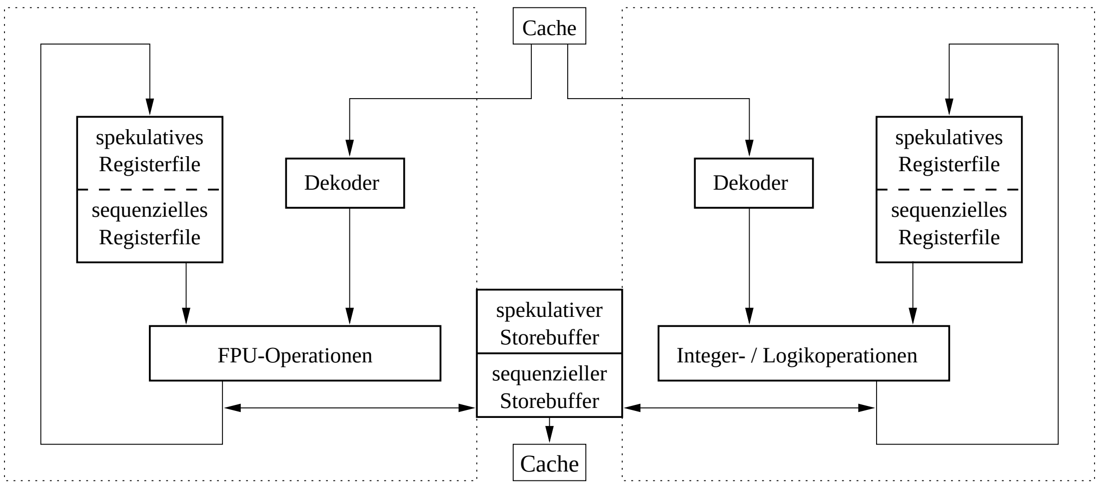

Abbildung 5: Block Diagramm der TORCH Hardware

5 MATCH

MATCH basiert auf einer Studie von M. Johnson.

5.1 Hardware

MATCH ist sehr ähnlich zu TORCH. Da MATCH jedoch das Befehlsscheduling mit seiner Hardware erledigt, ist der komplexeste Teil von MATCH seine Kontrollogik, welche in Bild 6 nicht zu sehen ist.

Vor jeder Funktionseinheit gibt es eine Reservation-Station, welche zusammen mit ihrer Kontrollogik die Out-of-Order Ausführung ermöglicht. Außerdem enthält MATCH noch einen Reorder-Buffer für die In-Order-Completion. Desweiteren wird noch ein Banch-Target-Buffer eingesetzt, um bedingte Sprünge zu beschleunigen, wobei hier Verzögerungen durch Befehls-misalignment nicht verhindert werden können.

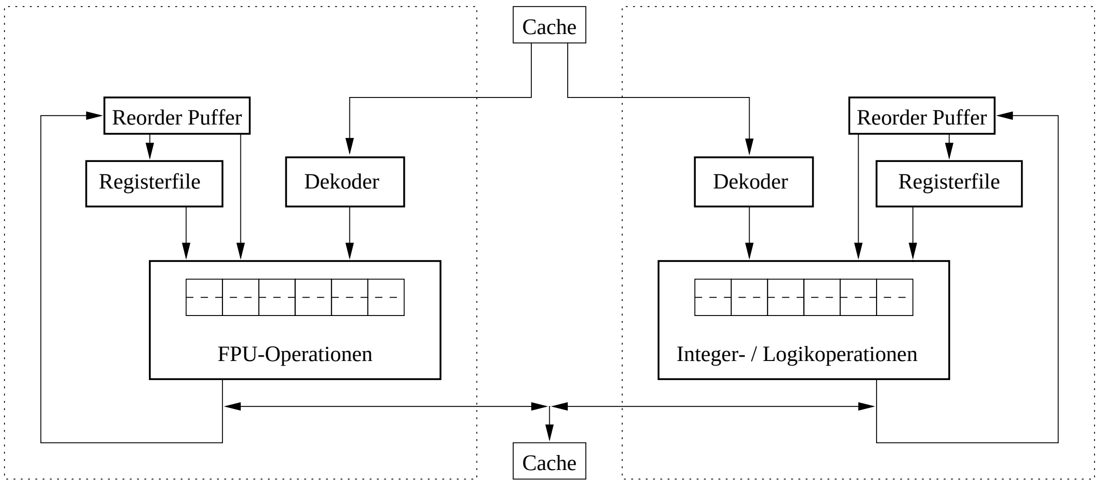

Abbildung 6: Block Diagramm der MATCH Hardware

6 Messungen

Im weiteren wird davon ausgegangen, daß ideale Caches und effektive Registerumbenennung beide Prozessoren in gleicher Weise beeinflussen. Deshalb gehen wir hier der Einfachheit halber von diesen idealen Bedingungen aus. Desweiteren wird davon ausgegangen, daß die Fetcheffektivität und nicht die Anzahl der Funktionseinheiten ausschlaggebend für die ILP in nicht numerischen Programmen ist.

Beide superskalare Prozessoren haben zwei ALU's und jeweils eine Funktionseinheit nach Tabelle 1, deren Charakteristika ebenfalls in Tabelle 1 beschrieben ist.

Funktionseinheit	Ausführungslatenz (Zyklen)	Ergebnislatenz (Zyklen)
Integer ALU	1	1
Barrel Shifter	1	1
Load Pipe	1	2
Store Pipe	1	–
Branch Unit	1	2
FP Addierer	1	2
FP Multiplizierer	1	5
FP Dividierer	1	19
FP Konverter	1	2

Tabelle 1: Ausführungs- und Ergebnislatenz

Tabelle 2 beschreibt die für den Benchmark verwendeten Programme, welche alle durch den Compiler hochoptimiert wurden, und nicht FPU-intensiv sind.

Programmname	Statische Befehle	Dynamische Befehle	Beschreibung
awk	26,9K	60,7M	pattern scanning/processing
ccom	58,1K	19,4M	front-end of a C compiler
espresso	43,6K	340,5M	logic minization
irsim	89,4K	54,3M	simulator for VLSI layouts
latex	53,2K	100,6M	document preparation system

Tabelle 2: Benutzte Programme

6.1 Ergebnisse

Alle Ergebnisse zeigen die Geschwindigkeitssteigerung gegenüber einem skalaren Prozessor. Um eine Referenz zu haben, zeigt Tabelle 3 die maximale theoretische Geschwindigkeitssteigerung die erreicht werden kann, wenn Load/Store Befehle der einzige Flaschenhals des Befehlsschedulings ist. Berechnet wurde dies durch:

$$\frac{\text{total serial clock cycles}}{\text{number of dynamic load/store instructions in the program}}$$

D.h. das Fetchen von 2–4 Befehlen pro Takt reicht aus, um das theoretische Maximum an Geschwindigkeitssteigerung zu erreichen. Deshalb werden bei TORCH und MATCH lediglich zwei oder vier Befehle pro Zyklus gefetcht.

awk	ccom	espresso	irsim	latex	ϕ
3,91	3,03	4,19	2,84	2,88	3,28

Tabelle 3: Maximale theoretische Geschwindigkeitssteigerung basierend auf Load/Store Häufigkeit

6.2 Berechtigung von Boosting

Um Boosting richtig einschätzen zu können, zeigt Tabelle 4 die Geschwindigkeitssteigerung die durch herkömliches statisches Scheduling erreicht wird.

	awk	ccom	espresso	irsim	latex	ϕ
Fetch 2	1,17	1,11	1,22	1,11	1,19	1,16
Fetch 4	1,17	1,11	1,22	1,12	1,21	1,16
1 IPC	1,19	1,23	1,08	1,27	1,13	1,18
unbegrenzt	1,24	1,32	1,35	1,24	1,41	1,31

Tabelle 4: Geschwindigkeitssteigerung (TORCH) nur durch Basic Block Scheduling

Die Daten der Zeilen Fetch 2 und Fetch 4 (welche die Anzahl der gefetchten Befehle pro Takt angeben) zeigen, daß diese nicht-numerischen Programme keine großen Abhängigkeiten innerhalb der Basic-Blöcke (BB) haben. Außerdem zeigten sie, daß in den Programmen die ILP innerhalb der BB sehr gering ist, und so schon das Fetchen von nur zwei Befehlen ausreicht. Zeile 3 zeigt, daß BB-Scheduling in etwa mit der Ausführung von einem Befehl pro Zyklus korrespondiert.

Die letzte Zeile in Tabelle 4 zeigt die maximal mögliche Geschwindigkeitssteigerung der Programme, wenn der Prozessor unendliche viele Funktionseinheiten hätte und man unendlich viele Befehle gleichzeitig Fetchen könnte.

6.3 Vergleich von TORCH und MATCH

Um beide superskalaren Prozessoren vergleichen zu können, wird davon ausgegangen, daß sie die Sprungvorhersage gleich handhaben. Weiterhin wird in Tabelle 5 davon ausgegangen, daß alle bedingten Sprünge genommen werden (d.h. es wird gesprungen).

	awk	ccom	espresso	irsim	latex	ϕ
Fetch 2						
TORCH	1,42	1,37	1,61	1,45	1,44	1,45
MATCH	1,34	1,28	1,45	1,31	1,34	1,34
MATCH +	1,54	1,46	1,58	1,51	1,48	1,51
Fetch 4						
TORCH	1,45	1,41	1,69	1,53	1,49	1,51
MATCH	1,40	1,37	1,58	1,43	1,40	1,43
MATCH +	1,73	1,71	1,88	1,73	1,70	1,75

Tabelle 5: Geschwindigkeitssteigerung bei korrekt vorhergesagten Sprüngen (MATCH+ mit Load/Store Reorganisation)

Wie man in Tabelle 5 sieht, ist statisches Scheduling mit Boosting effektiver als dynamisches Scheduling. Wenn zwei Instruktionen pro Takt gefetcht werden kommt dieser Effektivitätsvorteil daher, daß der Compiler in der Lage ist, Befehle im Speicher auszurichten und somit Befehlsmisalignment verhindert wird. Außerdem limitiert ein kleiner Fetchblock in MATCH auch das Befehlsfenster und somit das dynamische Scheduling. Obwohl ein größerer Fetchblock diese Effekte reduziert, ist TORCH trotzdem schneller.

Da der Scheduler in TORCH lediglich einen einzigen Boostinglevel zuläßt und der Compiler noch verbessert werden kann, läßt sich die Effektivitätssteigerungen von TORCH gegenüber dem skalaren Prozessor noch erhöhen, außerdem wurde in TORCH noch keine Load/Store Reorganisation (Priorisierung von Load Befehlen zur effektiveren Auslastung von Funktionseinheiten) implementiert.

Wenn man realistische Sprungvorhersagetechniken zuläßt, kommt man zu Ergebnissen wie in Tabelle 6, wobei in Tabelle 7 die Genauigkeit der Sprungvorhersage abgelesen werden kann. Hierbei ist der Branch-Target-Buffer in MATCH ein assoziativer 4-Wege Puffer mit 2048 Einträgen.

	awk	ccom	espresso	irsim	latex	ϕ
Fetch 2						
TORCH	1,49	1,52	1,70	1,55	1,55	1,56
MATCH	1,41	1,41	1,51	1,40	1,42	1,43
MATCH +	1,63	1,59	1,62	1,63	1,61	1,62
Fetch 4						
TORCH	1,52	1,57	1,79	1,64	1,63	1,63
MATCH	1,48	1,50	1,66	1,53	1,49	1,53
MATCH +	1,86	1,97	1,97	1,94	1,95	1,94

Tabelle 6: Geschwindigkeitssteigerung unter reeller Vorhersage von Sprüngen (MATCH+ mit Load/Store Reorganisation)

	awk	ccom	espresso	irsim	latex	ϕ
Predict Take	79,3	68,5	77,6	75,6	66,8	73,2
BTB	87,1	80,8	80,0	88,1	81,5	83,4
Profiling	91,0	92,6	86,9	92,1	90,6	90,6

Tabelle 7: Sprungvorhersage Genauigkeit (%)

6.4 Vergleich verschiedener Boostingoptionen

Im folgenden werden die Programme und Simulationsinformationen aus Tabelle 8 verwendet. Die Geschwindigkeitssteigerung gegenüber einem MIPS R2000 wird durch

$$\frac{\text{Anzahl der R2000 Zyklen}}{\text{Anzahl der superskalaren Zyklen}}$$

definiert.

	Total R2000 Cycles	Avg. R2000 IPC	Branch Prediction Accuracy
awk	47,2M	0,89	82,0%
compress	29,3M	0,87	82,7%
eqntott	0,7M	0,95	72,1%
espresso	97,8M	0,89	75,7%
grep	28,6M	0,81	97,9%
nroff	66,4M	0,82	96,7%
xlisp	1,0M	0,89	83,5%

Tabelle 8: Benchmarkprogramme und ihre Simulationsinformationen

Tabelle 9 zeigt Effektivitätssteigerungen verschiedener Boostingoptionen in

Prozent, normalisiert zu dem einfachen superskalaren Prozessor ohne Boosting jedoch mit statischem Befehlsscheduling.

	Squashing	Boost1	MinBoost3	Boost7
awk	11,2%	16,4%	17,2%	18,1%
compress	9,1%	10,6%	10,6%	10,6%
eqntott	8,0%	14,4%	16,0%	16,0%
espresso	9,8%	18,0%	21,3%	23,0%
grep	15,4%	27,7%	40,8%	40,8%
nroff	11,4%	24,4%	31,7%	36,6%
xlisp	6,7%	13,3%	12,5%	14,2%
G.M	9,9%	17,0%	19,3%	20,5%

Tabelle 9: Effektivitätssteigerungen gegnüber statischem Scheduling

Squashing ist hierbei die unter Abschnitt 3.2 beschriebene Option 3, welche keine Schattenstrukturen enthält und somit die kaum Zusatzhardware enthält.

Boost1 entspricht der Boostinghardware in TORCH.

MinBoost3 erlaubt in einem Registerfile Boosting über 3 Sprünge hinweg und entspricht Option 2 in Abschnitt 3.2.

Boost7 erlaubt einen unlimitierten Boostinglevel von 7 und entspricht der Hardware von Abschnitt 3.1. Da Boost7 lediglich eine vernachlässigbare Geschwindigkeitssteigerung gegenüber Boost1 und MinBoost3 bringt und dazu noch einen großen Hardwareaufwand erfordert, hat diese Implementation ein schlechtes Preis/Leistungsverhältnis.

Der Dekoder einer Boost1 Maschine mit 32 sequentiellen Registern enthält nur 33% mehr Transistoren als ein normaler Dekoder für 64 Register (50% mehr Transistoren für eine MinBoost3).

6.5 Vergleich von MinBoost3 mit einem dynamischen Scheduler

Bild 7 vergleicht MinBoost3 mit einem dynamischen Scheduler, da MinBoost3 das beste Preis/Leistungsverhältnis der verschiedenen Implementierungsmöglichkeiten hat. Der dynamische Scheduler fetcht und dekodiert zwei Befehle pro Zyklus, hat insgesamt 30 Plätze in den Reservation-Stations und 16 Einträge im Reorderbuffer um spekulative Out-of-Order Ausführung zu ermöglichen. Desweiteren benutzt er einen assoziativen 4-Wege Branch-Target-Buffer mit 2048 Einträgen.

Die untere Hälfte des Balkens bei MinBoost3 zeigt die Geschwindigkeitssteigerung wenn Register vor dem Befehlsscheduling alloziert werden. Die obere Hälfte repräsentiert ein Model mit unendlich vielen Registern. Bei dem dynamischen

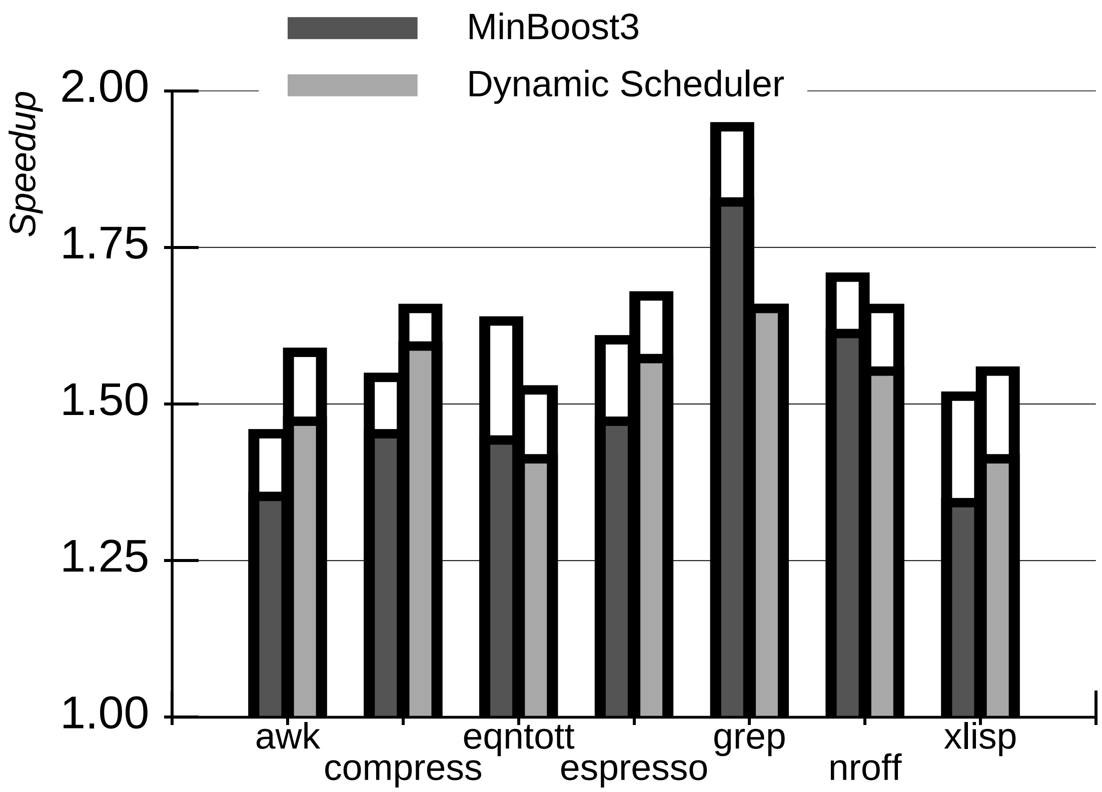

Abbildung 7: Effektivitätsvergleich mit einem dynamischen Scheduler

Scheduler zeigt die untere Hälfte die Effektivität ohne, die obere Hälfte mit Registerumbenennung.

Wie man sieht, erreicht man schon mit geringen Kosten eine hohe Effektivität.