Fakultät Informatik Institut für Technische Informatik

EXPLICIT MULTI-THREADING

Mattis Hasler

Dresden, 17.5.2011

Inhalt

- PRAM
- XMT Idee
- Architektur
- Speichertechniken
- Implementierungen
- Leistungsevaluation

PRAM

- PRAM - parallel random access machine
- Idee: feingranulare Instruktionslevelparallelität
- in 90er noch in populären seriell Programmierung Büchern vertreten
- Mitte 90er in den Meisten abgeschafft da "impossible in reality"
- Algorithmen unter der Annahme von unbegrenzter paralleler Hardware

PRAM Beispiel

```
for i = 0 to n pardo{
    C[i] = A[i] + B[i]
}
```

PRAM vs. RAM

RAM

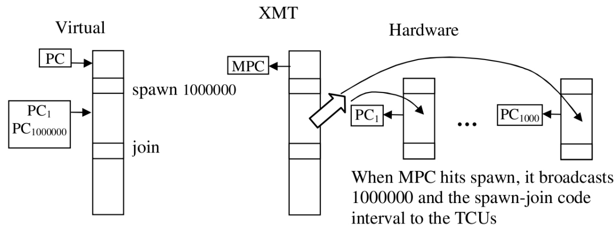
- random access machine
- “normales” serielles Modell
- Operationen werden seriell ausgeführt
- jede OP (logisch, arithmetisch, Speicher) dauert eine Zeiteinheit

PRAM

- parallel random access machine
- beliebig viel Speicherzugriffe und Operationen gleichzeitig möglich
  - Speicherzugriffe dauern trotzdem nur eine Zeiteinheit
- in XMT nehmen wir arbitrary CRCW als Speicherzugriffsmodell an
  - CRCW - concurrent read concurrent write
  - arbitrary - bei mehreren geichzeitigen Schreibzugriffen auf eine Adresse wird ein zufälliger Wert geschrieben

XMT

- eXplicit Multi-Threading
- →Parallelisierung von Prozessen (nicht parallele Ausführung mehrerer Prozesse)
- Wechsel zwischen zwei Modi: Seriell und Parallel
- Abschnitte für Parallelmodus:
  - explizit vom Programmierer zu definieren
  - PRAM ähnlicher Code; PRAM Algorithmen 1 zu 1 übertragbar
  - einfache zu programmieren; ein Hauptziel von XMT
- PRAM Parallelität wird auf Threads gemappt
- `Spawn` und `Join` zum Beschreiben paralleler Abschnitte
  - wenig Overhead
  - →sehr kurze Threads effizient



(a) Program counter + stored program

[2]

no-busy-wait FSM

Problem: falsche Abhängigkeiten von Befehlen führen bei vielen parallelen Architekturen zu unnötigen Verzögerungen. (z.B. Cache Miss in einem VLIW Teilbefehl lässt möglicherweise unabhänige Teilbefehle warten.)
Lösung:

- no-busy-wait FSM
  - jeder Thread bekommt eine eigene virtuelle FSM
  - FSMs können sich nicht gegenseitig beeinflussen
  - Synchronisation nur am Threadende "`Join`"
  - →kein busy-waiting
- FSMs sind virtuell →Anzahl von FSMs unabhänig von der Hardware
- Ausführung von immer k Threads gleichzeitig (bei k Recheneinheiten)
  - ist ein Thread beendet wird der Recheneinheit ein Neuer zugewiesen
  - "langsame" Threads beeinflussen die Ausführung anderer Threads nicht
  - möglich durch "independence of order semantics" (IOS)

XMTC

- C-Extension zur Definition paraller Codeabschnitte
- `spawn` & `join` → n Threads
- `$` ThreadID im parallelen Code
- `ps` - prefix-sum: atomarer Befehl
- XMTC: SPMD - single program multiple data
- `xmtc` - auf gcc basierender Compiler
  - benutzt ein um XMT-Befehle erweitertes MIPS Instruction set

Beispiel: Arraykompaktierung

```
int psBaseReg x = 0;
spawn(0, n - 1){
    int e = 1;
    if(A[$] != 0){
        ps(e,x);
        B[e] = A[$];
    }
}
```

Architektur

Komponenten

- TCU - thread control unit
- MTCU - master thread control unit
- Cluster - Gruppe von TCUs
- PSU - prefix-sum unit
- shared Cache
- interconnection network

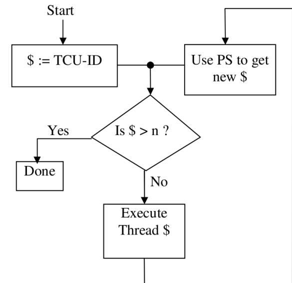
Speichertechniken

- value broadcastiong
- prefetching
- readonly buffer

TECHNISCHE UNIVERSITÄT DRESDEN

TCU

- thread control unit
- führt einen virtuellen Thread aus
- "besorgt" sich neuen Thread wenn Eigener beendet ist
- lokales Registerfile
- prefetch buffer
- einfache Pipeline
- einfache Logikeinheiten (Addierer...)
- einfache Kernstruktur →kleiner Kern
- →viele kleine Kerne besser als wenig Große



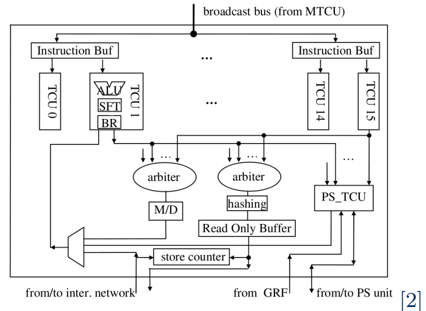
[2]

MTCU

- master thread control unit
- übernimmt die Ausführung im seriellen Modus
- `Spawn` & `Join`
- sendet Instruktionen für paralleles Arbeiten an lokale Instruktionsbuffer (broadcast)
- value broadcasting
- L0 - eigener Level 1 Cache während der seriellen Ausführung
  - paralleler Cache ist Level 2

Cluster

- Gruppe von TCUs
- gemeinsam genutzte Logikeinheiten (arbitered)
- load/store Einheit
- readonly buffer
- instruction buffer
- lokale prefix-sum Einheit



[2]

Prefix-Sum Unit

- hierarchisch aufgebaut
  - lokale PSU im Cluster gibt gesammelte Anfrage an gobale PSU weiter
- behandelt `ps` Befehl atomar
- hocheffizient implementiert, um das System nicht auszubremsen
- Ausführungszeit unabhänig von Anzahl der Anfragen

Shared Caches

- Cache-Kohärenz Protokolle kompliziert
- →keine lokalen Caches (TCU, cluster)
- mehrere chipglobale Cache-Einheiten
- Einheiten teilen Speicher gleichmäßig auf
- “hashing” um Last auf Cache-Einheiten gleichmäßig zu verteilen

Value Broadcasting

Problem: ein Wert den (fast) jeder Thread braucht wird von jedem Thread einmal geladen →ineffizient
Lösung: Value Broadcasting

- MTCU lädt den Wert kurz vor Beginn des parallelen Teils
- load immediate (`li`) mit dem geladenen Wert wird in den parallelen Code eingefügt
- MTCU sendet Code per Broadcast zu allen TCUs

Nachteil: der Wert belegt ein Register in jeder TCU
Vorteil: der Wert muss nur einmal geladen werden

Prefetching

Vermeidung von Wartezeiten beim `load`

- der Compiler fügt an geeigneter Stelle prefetch-Befehle ein
- nichtblockierender prefetch-Befehl bringt den Wert in TCUs prefetch buffer
- späterer `load` auf die selbe Adresse holt den Wert aus prefetch buffer garantiert in einem Takt

Nachteil: Overhead in Prozessor (prefetch belegt den Prozessor 1 Takt)
Vorteil: Prozessor nicht untätig wenn es zu delays kommt

Readonly Buffer

Beschleunigung des mehrfachen Lesens eines Wertes

- Compiler ersetzt bestimmte load-Operationen mit einer XMT Variante
- dadurch wird der Wert zusätzlich im readonly buffer des Clusters abgelegt
- spätere load-Operation auf dieselbe Adresse werden vom readonly buffer behandelt

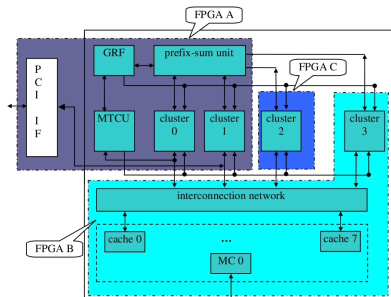
Nachteil: - Compiler muss Fälle in denen der readonly buffer benutzt werden kann erkennen und spezielle load-Operation einfügen
Vorteil: - Entlastung des interconnection networks und der Cache-Einheiten
- alle TCUs des Clusters profitieren von einem Wert der im readonly buffer liegt

FPGA Prototyp

- 3 FPGAs (2 $\times$Vitrex-4 LX200, 1 $\times$Virtex-4 LX100) auf PCI Entwicklungsboard
- 75MHz Systemtakt mit 1GB DDR Speicher

Struktur

- 1 MTCU
- 4 Cluster mit je 16 TCUs
- 8 Cache-Einheiten
- interconnection network
- 1 globales Registerfile
- 1 prefix-sum Einheit



[2]

Angestrebter ASIC

- 1024 TCUs in 64 Clustern
- 64 Cache-Einheiten
- 800 MHz Systemtakt
- 400 MHz RAM clock
- separierter Instruction- und Datacache in der MTCU für Kompatibilität mit seriellen Programmen

Evaluationssetup

- parallel
  - XMT FPGA @ 75MHz
  - XMT ASIC @ 800MHz (Emulated)
- seriell
  - AMD Opteron 2.6GHz
  - 64KB + 64KB L1, 1MB L2
  - PC-3200 DDR DRAM
- benchmarks
  - mmul - Matrixmultiplikation
  - qsort - Quicksort
  - BFS - breath first search
  - DAG - längsten Pfad in azyklischen Graphen finden
  - add - Arraysummation
  - comp - Arraykompaktierung
  - BST - Schlüsselsuche in binären Bäumen
  - conv - Faltung

Ergebnisse

- speedups parallel vs. seriell in Takten:
  - obere Schranke 64 da XMT FPGA 64 TCUs hat, also max. 64 Threads parallel ausgeführt werden

Größe	mmul	qsort	BFS	DAG	add	comp	BST	conv
groß	35.7	20.8	15.7	19.6	26.0	15.3	7.0	38.9
klein	45.9	16.8	18.1	22.0	24.0	23.8	10.0	36.9

[2]

- absolute Ausführungszeit

Prozessor	mmul	qsort	BFS	DAG	add	comp	BST	conv
Opteron	117.4	2.644	0.659	2.594	0.143	0.105	0.479	1.776
XMT FPGA	64.74	6.483	0.604	3.101	0.193	0.267	1.469	6.884
XMT ASIC	13.71	1.634	0.351	1.494	0.067	0.051	0.305	0.647

[2]

Zusammenfassung

- wissenschaftliche Architektur
  - kein Betriebsystem
  - in parallelen Abschnitten keine Funktionsaufrufe
- Multi-Threading noch nicht mal ansatzweise möglich
- Vorteile
  - einfach zu programmieren (ein Hauptziel von XMT)
  - auch bei sehr kurzen parallelen Abschnitten sehr effizient
- Nachteile
  - alter (serieller) Code kann nur eingeschränkt verwendet werden
  - viele essentielle XMT-spezifische Konstrukte momentan nur von Hand möglich (prefetching, readonly buffer, value broadcasting)
  - nicht parallelisierbare Prozesse können Hardware nicht ausnutzen

ENDE

U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism (Extended Abstract). In Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1998.

G.G. Caragea, A.B. Saybasili, X. Wen and U. Vishkin. Performance potential of an easy-to-program PRAM-On-Chip prototype versus state-of-the-art processor. Proc. 21st ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 163-165, Calgary, Canada, August 10-13, 2009.

X. Wen and U. Vishkin. PRAM-On-Chip: First Commitment to Silicon. Brief announcement in Proc. 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 301-302, June 9-11, 2007.

Anhang - Benchmark Datensatzgröße

Größe der Eingabedaten:

App.	Large size			Small size		
	input size	memory usage		input size	memory usage	
		parallel	serial		parallel	serial
mmul	2000x2000	48MB	48MB	128x128	192KB	192KB
qsort	20 million	360MB	200MB	100 thousand	1.8MB	1MB
BFS	V=1M, E=10M	220MB	100MB	V=100K, E=1M	21.6MB	9.6MB
DAG	V=1M, E=17M	368MB	160MB	V=50K, E=600K	13.4MB	6.0MB
add	50 million	200MB	200MB	3 million	12MB	12MB
comp	20 million	208MB	208MB	2 million	20.8MB	20.8MB
BST	16.8M nodes, 512K keys	205MB	205MB	2.1M nodes, 16K keys	25.3MB	25.3MB
conv	image:1000x1000 filter:32x32	8MB	8MB	image:200x200 filter:16x16	320KB	320KB

[2]