Programmer Education for the Multicore Era: the Twin-paradigm approach

The single-paradigm dictatorship

- The single-paradigm dictatorship
- Von Neumann vs. FPGA
- The Datastream Machine Model
- Avoiding address computation overhead
- The twin Paradigm approach
- Conclusions

The spirit of the Mainframe Age

- For decades, we've trained programmers to think sequentially, breaking complex parallelism down into atomic instruction steps ...
- ... finally tending to code sizes of astronomic dimensions
- Even in "hardware" courses (unloved child of CS scenes) we mostly teach von Neumann machine design - deepening this tunnel view
- 1951: Hardware Design going von Neumann (Microprogramming)

Program Performance

“Multicore computers shift the burden of software performance from chip designers to programmers.”

... performance drops & other problems in moving single-core to multicore ...

The law of Moore? No, the law of More

Massively decreasing programmer productivity

supercomputing: multi-disciplinary multi-location team works several years: software ready - hardware obsolete

Missing programmer population and methodology: a scenario like before the Mead- & Conway revolution

Why we went multicore

Four walls:
- Instruction-level parallelism
- Memory
- Power
- Complexity

Multicores promised to remove the walls
Thousands of cores; boy, so many challenges...

More cores instead of faster cores

Avoiding the decline from growth industry to replacement business ?
- no, not without redefinition of the field
- not under the single-paradigm dictatorship

sequential-only mind set dominating parallel algorithms mostly missing very difficult to program useful abstractions mostly missing
Very difficult to program

Programming skills needed going far beyond sequential programming
Massive Synchronization overhead
Race conditions: reasons of bugs
Non-determinism: new types of bugs
Language and tool support missing

Useful abstractions mostly missing

Parallelism models machine-specific and low level
shared memory use or message passing (hardware features)
Parallel programming at assembly language level
multi-threading, semaphores, locking (compare & swap)
Performance models are machine-specific
Problems in portability, investment reuse, economics of scale

Which programming model to use?

Stubborn consensus on the von Neumann paradigm
its enforced monopoly-like dominance is the key problem
it is incredibly inefficient (the von Neumann syndrome)
No consensus on parallelism model
data parallelism, message passing, (unstructured) multi-threading, or ?
Many applications use all three or even more
Language & tool support needed to integrate models
unqualified programmer population: Education reform needed

Program Performance

"Multicore computers shift the burden of software performance from chip designers to programmers."
... performance drops & other problems in moving single-core to multicore ...
Since People have to write code differently, we anyway need a Software Education Revolution ...
... the chance to move RC* from niche to mainstream
Missing programmer population and methodology: a scenario like before the Mead-Ã–Conway revolution

Power Consumption of Computers

Has become an industry-wide issue:
incremental improvements are on track, (plain Green Computing)
but we may ultimately need revolutionary new solutions
[Horst Simon, LBNL, Berkeley]
Twin Paradigm Green Computing
More effective by orders of magnitude
Energy cost may overtake IT equipment cost in the near future
Current trends will lead to unaffordable future operation cost of our cyber infrastructure

For a Booming Multicore Era

von-Neumann-only is not the silver bullet
Reconfigurable Computing is indispensable!
From CPU to RPU
Reconfigurable Processing Unit

<table>
<thead>
<tr>
<th>machine model right now</th>
<th>resources</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>property</td>
<td>programming source</td>
</tr>
<tr>
<td>ASIC</td>
<td>hardwired</td>
<td>-</td>
</tr>
<tr>
<td>CPU</td>
<td>hardwired</td>
<td>-</td>
</tr>
<tr>
<td>RPU</td>
<td>programmable</td>
<td>Software (data streams)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>program counter</td>
</tr>
</tbody>
</table>

now accelerators are programmable! non-von-Neumann
we need 2 more program sources

A Multicore Submarine Model?
mapping parallelism just into the time domain: “abstracting” away the space domain is fatal
C is not the silver bullet: it’s inherently serial
There is no easy way to program in parallel
The programmer needs to understand how data flows through cores, accelerators, interconnect and peripherals
The datastream model of the twin-paradigm approach helps to understand the space domain and parallelism
The programmer needs system visualization in the space domain, to understand performance under parallelism
*) and, especially the student

>> Outline <<
• The single-paradigm dictatorship
• Von Neumann vs. FPGA
• The Datastream Machine Model
• Avoiding address computation overhead
• The twin Paradigm approach
• Conclusions

The first Reconfigurable Computer
• prototyped 1884 by Herman Hollerith
• a century before FPGA introduction
• data-stream-based
• 60 years later the von Neumann (vN) model took over

Languages turned into Religions
Java is a religion – not a language

widening the semantic gap
[Harold „Bud“ Lawson]

unnecessary complexity inside

Burroughs 5500: language-friendly stock machine
IBM 260/370 & intel x86
highly complex instruction set
MULTICS (GE, Honeywell): well manageable (impl. in PL/1)
UNIX: complexity problems, compatibility problems
Pascal killed by C, coming as an infection, along with UNIX
KARL killed by VHDL, an infection coming along with Ada

• teaching to students the tunnel view of language designers
• falling in love with the subtleties of formalisms
• instead of meeting the needs of the user
Appeals to people who do not know what they are doing. Mastering even small complexity creates a deep feeling of satisfaction without solving the real problem. The transition from machine level to higher level languages led to the biggest productivity gain ever made. It’s alarming that today’s megabytes of code are compiled from languages at low abstraction levels (C, C++, Java).

The power efficiency is disputable. GPUs can only be used in certain ways. such speed-ups by GPUs only for embarrassingly parallel applications, effective only at problems that can be solved using stream processing, programmer has to learn irrelevant graphics concepts, data copy from main memory to video memory is slow.

Speed up factors by GPGPUs (1)

(1) NVIDIA Zone Home web site [NVIDIA Corp.]

compared to Configware migration, Intel Xeon "Nehalem" vs. NVIDIA Fermi GPU: 50x

- Massive Energy Saving factors: ~10% of speedup factor

Software vs. FPGA (2)
**RC*: Demonstrating the intensive Impact**

<table>
<thead>
<tr>
<th>Application</th>
<th>Speed-up factor</th>
<th>Power</th>
<th>Size</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA and Protein</td>
<td>8723</td>
<td>779</td>
<td>22</td>
<td>253</td>
</tr>
<tr>
<td>DES breaking</td>
<td>28514</td>
<td>3439</td>
<td>96</td>
<td>1116</td>
</tr>
</tbody>
</table>

*RC = Reconfigurable Computing*

---

**Why such Speed-up Factors ...**

... with FPGAs: a much worse technology!
- massive wiring overhead
- massive reconfigurability overhead
- routing congestion growing with FPGA size

The „Reconfigurable Computing Paradox”
- main reason: NO von Neumann Syndrome!
- more recently also: more “platform FPGAs”

---

**>> Outline <<**

- The single-paradigm dictatorship
- Von Neumann vs. FPGA
- The Datastream Machine Model
- Avoiding address computation overhead
- The twin Paradigm approach
- Conclusions

---

**Reconfigurability per se is not the key**

It’s the paradigm coming along with it

Note: no instruction fetch at run time!

Data streams instead of instruction streams

Enabling technology for data sequencers brings further performance improvements

A non-reconfigurable example is the BEE project (Bob Broderson et al., UC Berkeley)

---

**“data stream”: an ambiguous definition**

Reconfigurable Computing is not instruction-stream-based
- it’s data-stream-based
- it’s different from the operation of the (indeterministic) „dataflow machine”
- other definitions also from multimedia area
- usable definition from systolic array area

---

**Introducing Data streams**

Systolic array

(pipe network) DPA

Input: data stream

H. T. Kung et al. (1979, 1980)

Flowware defines: which data item at which time at which port

---
Classic Systolic Array Synthesis

algebraic methods
i.e., linear projections
yields only uniform
arrays w. linear pipes
only for applications with
regular data dependencies

The counterpart of the
von Neumann machine

course-grained

(r)DPA

data counters: located at memory
(Not at data path)

Distributed

RAM

ASM: Auto-
Scheduling

Memory

Who generates the Data
Streams?

Why the SA scene has missed to
invent the new machine paradigm

reductionist
approach:
"it's not our job"

"systolic"

(it's not algebraic)

Algebraic Synthesis Methods

array
applications
Pipeline properties
mapping
scheduling (data stream
formation)

uniform
resources
regular data
dependencies

shape

KressArray principles

- take systolic array principles
- replace classical synthesis by simulated annealing
- yields the super systolic array
- a generalization of the systolic array
- no more restricted to regular data dependencies
- now reconfigurability makes sense

Generalization of the Systolic Array

discard algebraic synthesis methods
use optimization algorithms instead,
for example: simulated annealing
the achievement: also non-linear
and non-uniform pipes, and even
more wild pipe structures possible
now reconfigurability really makes sense

flowware history:

1980: data streams
(King, Leiserson)
1995: super systolic rDPA
(Rainer Kress)
1996+: SCC (LANL), SCORE,
ASMPC, Kress (UCB).
Super-systolic Synthesis

The key is mapping, rather than architecture

<table>
<thead>
<tr>
<th>array</th>
<th>applications</th>
<th>pipeline properties</th>
<th>mapping</th>
<th>scheduling (data stream formation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>systolic</td>
<td></td>
<td>uniform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>super-systolic</td>
<td></td>
<td>only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dependencies</td>
<td></td>
<td>linear</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend: backbus connect

Coarse-grained Reconfigurable Array example

image processing: SNN filter (mainly a pipe network)

coming close to programmer’s mind set (much closer than FPGA)

note: kind of software, perspective, but without instruction streams → datastreams + pipelining

question after the talk: but you can’t implement decisions!

The wrong mind set......

S = R + (if C then A else B endif); 

section of a very large pipe network: 

symptom of the hardware / software chasm 

and the configure / software chasm

S = R + (if C then A else B endif);

embarrassing remark - not knowing this solution:

“but you can’t implement decisions!”

Simple conservative CPU example

C = 1

memory

cycles

5

100

if C then read A else B endif

read instruction

1

100

if not C then read B else read A endif

read instruction

1

100

add & store

new result

1

100

store result

1

50

C = 1

section of a major pipe network with 500 rDPUs

The wrong mind set - the wrong road map!

introducing hardware description languages

(data stream formation)

The decision box becomes a (de)multiplexer

This is so simple: why did it take decades to find out?

The wrong mind set - the wrong road map!
Programming Language Paradigms

<table>
<thead>
<tr>
<th>Language Category</th>
<th>Computer Languages</th>
<th>Languages for Anti-Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both Deterministic</td>
<td>Procedural Sequencing</td>
<td>Traceable, Checkpointable</td>
</tr>
<tr>
<td>Operation Sequence</td>
<td>Read data instruction, jump (to instr. addr.), branch, loop nesting</td>
<td>Read data from data stream, jump (to data addr.), data loop, loop nesting</td>
</tr>
<tr>
<td>State Register</td>
<td>Program Counter</td>
<td>Data Counter(s)</td>
</tr>
<tr>
<td>Address Computational</td>
<td>Memory Cycle Overhead</td>
<td>Overhead Avoided</td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>Parallel Memory Task Access</td>
<td>Interleaving Only</td>
</tr>
<tr>
<td>Language Features</td>
<td>Data Stream Only</td>
<td>No Restrictions</td>
</tr>
</tbody>
</table>

Double Dichotomy

1) Paradigm Dichotomy
- von Neumann Machine
  - Instruction Stream: (Software-Domain)
  - Data Stream: (Flowware-Domain)
- Datastream Machine
  - Data Stream: (Software-Domain)
  - Structure: (Configure-Domain)

2) Relativity Dichotomy
- Time: Procedure
  - Space: Structure

Relativity Dichotomy

Time Domain: Procedure Domain
- 2 phases:
  1) Programming Instruction Streams
  2) Run Time

Space Domain: Structure Domain
- 3 phases:
  1) Reconfiguration of Structures
  2) Programming Data Streams
  3) Run Time

Time Iterative to Space Iterative

Often the space dimension is limited to 1 time step, 1 CPU

Loop Transformation Examples

Sequential Processes:
1) 1-8 body endloop
2) 1-16 body endloop
3) 1-2 trigger endloop
4) 1-8 trigger endloop

Resource Parameter Driven Co-Compilation

Pipeline:
- Loop 1-16 body endloop
- Loop 1-8 body endloop
- Loop 9-16 body endloop

Loop Unrolling:
- Loop 1-8 body endloop
- Loop 1-2 trigger endloop

Strip Mining
- [D. Loveman, J. ACM, 1977]
### Pros for streaming

- Streamlined, low-overhead communication
- (More) deterministic behaviour
- Good match for many simple media-rich applications

### Cons

- Control-dominated applications
- Shunt yard problem

We need to find out which applications types and programming models students should exercise for the flowware approach.

#### The new paradigm: how the data are traveling

- No, not by instruction execution
- Transport-triggered: an old hat pipeline, or chaining
- Asynchronous (via handshake)
- Systolic array
- Wavefront array

---

#### How the data are moved

- DMA,
- vN move processor (Jack Lipovski, EUROMICRO, Nice, 1975)
- Henk Corporaal coins the term "transport-triggered"
- MoM: GAG-based storage scheme methodology (Merz?)*
- Application-specific distributed memory (Catthoor et al.)

---

#### The Paradigm Shift to Data-Stream-Based

#### The Method of Communication and Data Transport

- The von Neumann syndrome complex pipe network on RTDA
- by Software by Configure

---

#### Illustrating the von Neumann paradox trap

The instruction-stream-based approach

The data-stream-based approach

The data-stream-based approach has no von Neumann bottleneck

---

#### JPEG zigzag scan pattern

Flowware language example (MoFl): programming the datastream (an animation)
• The single-paradigm dictatorship
• Von Neumann vs. FPGA
• The Datastream Machine Model
• Avoiding address computation overhead
• The twin Paradigm approach
• Conclusions

Significance of Address Generators

• Address generators have the potential to reduce computation time significantly.
• In a grid-based design rule check a speed-up of more than 2000 has been achieved, compared to a VAX-11/750
• Dedicated address generators contributed a factor of 10 - avoiding memory cycles for address computation overhead

Generic Address Generator (GAG)

Generalization of the DMA

Acceleration factors by:

• address computation without memory cycles
• storage scheme optimization methodology, etc.

GAG & enabling technology published 1989, survey:
[M. Herz et al.: IEEE ICECS 2003, Dubrovnik]
patented by TI 1995

ASM: Auto-Sequencing Memory

Generalization of the DMA

Acceleration factor: generic address generator (GAG) for address computation without memory cycles

... partly explaining the RC paradox

GAG & enabling technology published 1989, survey:
[M. Herz et al.: IEEE ICECS 2003, Dubrovnik]

Migration benefit by on-chip RAM

Some RC chips have hundreds of on-chip RAM blocks, orders of magnitude faster than off-chip RAM
so that the drastic code size reduction by software to configure migration can beat the memory wall
multiple on-chip RAM blocks are the enabling technology for ultra-fast anti machine solutions

Acceleration Mechanisms

• parallelism by multi bank memory architecture
• auxiliary hardware for address calculation
• address calculation before run time
• avoiding multiple accesses to the same data.
• avoiding memory cycles for address computation
• optimization by storage scheme transformations
• optimization by memory architecture transformations
The Datastream Machine Model

The twin Paradigm approach

Dual paradigm mind set: an old hat
(mapping from procedural to structural domain)

Software mind set:
instruction-stream-based:
flow chart ->
control instructions

Mapped into a Hardware mind set:
action box = Flipflop, decision box = (de)multiplexer

- C. G. Bell et al: The Description and Use of Register-Transfer Modules (RTMs); IEEE Trans-C21/5, May 1972

Nick Tredennick’s Paradigm Shifts explain the differences

Software Engineering
CPU

resources: fixed
algorithm: variable
1 programming
source needed

Configware Engineering

resources: variable
algorithm: variable
2 programming
sources needed

>> Outline <<

- The single-paradigm dictatorship
- Von Neumann vs. FPGA
- The Datastream Machine Model
- Avoiding address computation overhead
- The twin Paradigm approach
- Conclusions

The single paradigm mind set: an old hat
(research from software compilation
fundamentally different
configware compilation
vs.
from software compilation
configware

Software Engineering

CPU

resources: fixed
algorithm: variable
1 programming
source needed

Configware Engineering

resources: variable
algorithm: variable
2 programming
sources needed

>> Outline <<

- The single-paradigm dictatorship
- Von Neumann vs. FPGA
- The Datastream Machine Model
- Avoiding address computation overhead
- The twin Paradigm approach
- Conclusions

© 2009, reiner@hartenstein.de

15 May 2013
Our Contemporary Computer Machine Model

<table>
<thead>
<tr>
<th>Machine model</th>
<th>resources</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>property</td>
<td>property</td>
</tr>
<tr>
<td></td>
<td>programming source</td>
<td>programming source</td>
</tr>
<tr>
<td>ASU accelerator</td>
<td>hardwired</td>
<td>hardwired</td>
</tr>
<tr>
<td>CPU</td>
<td>hardwired</td>
<td>programmable</td>
</tr>
<tr>
<td>RPU accelerator</td>
<td>programmable</td>
<td>state register</td>
</tr>
</tbody>
</table>

data counters of reconfigurable address generators in ASU (auto-sequencing) data memory blocks

twin Paradigm Dichotomy
the same language primitives!

Compilation: Software vs. Configware

Software Engineering

Configure Engineering

Software compiler

Source program

Placement & routing

Source program

Configware compiler

Flowware compiler

Compiling: Software vs. Configware

Co-Compilation

Co-Compiler for Hardwired Anti Machine

Relativity Dichotomy

The biggest payoff will come from putting Old Ideas into Practice and teaching people how to apply them properly.

David Parnas

Time to Space Mapping
**Dual Paradigm Application Development**

- High level language
- Software/configware co-compiler
- Software code
- Instruction-stream based
- Configware code
- Data-stream based
- Reconfigurable accelerator
- Hardwired accelerator
- Dual mode pool

---

**Ways to implement an Algorithm**

- Hardware
- Software
- Configware
- Flowware
- Mixed

**Flowware**

Flowware means parallelism resulting from time to space migration.

Flowware: scheduling data streams - from a generalization of the systolic array supports any wild free form of pipe networks: spiral, zigzag, fork and join, and even more wild, unidirectional and fully or partially bidirectional, Fifos, stacks, registers, register files, RAM blocks...

---

**Software Education (R)evolution:**

- Step by step, not overthrowing the SE scene
- By simultaneous dual domain co-education:
  - Traditional qualification in the time domain
  - Lean qualification in the space domain
  - Lean hardware modeling qualification at a higher level of abstraction
- Viable methodology for dual rail education (only a few % curricula need to be changed)

---

**RC versus Multicore**

- RC: Speed-up often higher by orders of magnitude
- RC: Energy-efficiency often higher: Very much, or, by orders of magnitude?

**This is the silver bullet**

We need both: Multicore and RC
We need new courses

We urgently need a Mead- & Conway-like text book

We need undergraduate lab courses with HW / CW / SW partitioning

We need new courses with extended scope on parallelism and algorithmic cleverness for HW / CW / SW co-design

“We urgently need a Mead- & Conway-like text book”

[R. H., Dagstuhl Seminar 03301, Germany, 2003]

here its foreruner: but not yet twin paradigm

Software Education Revolution for using Multicore - and RC*(SERUM-RC*)

*) Reconfigurable Computing

“We urgently need a Mead- & Conway-dimension text book on twin-paradigm programming education”

thank you