RAS

1. Overview

Reconfigurable system ≙ (partially) change the function of its components, in our case: hardware

Adaptive system ≙ system w/ capability to adapt one or more of its properties

Idea: implement application specific accelerators on demand (exploiting parallelism & operator chaining) using FPGA-like structures: CLB, PSM, ...

Coarse-grained: configuring whole busses, driving ALU-array

Fine-grained: configuring each bit lane, driving ~~Slice~~ LUTs

Self-reconfiguration needs capability of partial reconfiguration

Memory that stores the configuration bitstream.

PROM: only writable once

(E)EPROM: non-volatile, low power, few write-cycles, slow

SRAM: fast, quasi ∞ # of configurations, high power

Hybrid of SRAM & (E)EPROM: auto copy @ bootup to SRAM, @ cost of larger chip area

Practically, the reconfiguration bandwidth is limited by the memory reading speed

Time for reconfiguration: ~ 1ms - 10ms

rather long time for a CPU

Coarse-grained are faster, because of fewer reconfiguration bits @ cost of efficiency when facing bit-level-operations like bit shuffling

2. Special Instructions

Connecting reconfigurable fabric to the CPU:

External, off-chip, loosely coupled
- can be connected to existing ICs
- can execute in parallel
- very high communication overhead; slow interconnection,
- no acces to CPU's internal registers

Attached:
- reconf. fabric connected to data bus of CPU
- requires new IC
- much faster communication
- still a lot overhead for communication and no access to internal registers

Coprocessor
- similar to "attached" but dedicated communication interface besides databus

RFU
- Embedded reconf. fabric in CPU
- low communication overhead w/ high data bandwidth
- requires modifying the core ISA

CPU as IP in FPGA (soft / hard core)
- no new IC needs to be developed
- any CPU realisable
- can be used for simulation of both, ~~tool~~ loosely and tightly coupled systems
- noticably lower frequency
- may require modifying the CPU architecture

Summary: loosely coupled:
- communication overhead
- speedup needs to compensate this
+ easy to be done w/ standard CPU & FPGA
+ can run in parallel

tightly coupled:
+ communication overhead insignificant
+ multiple RFUs can be attached, calculating & configured @ the same time
- new IC needs to be developed
- speedup limited by size of chip

SI Input Data
stream based: continuous data for filter / video encoder, ...
chunk based: working on larger parts of data
element based: bit-reversal / MAC / sin / cos / e^x...
SIs for more efficient memory access, using stream based operations & smart address generation
if SIs access memory, coherency plays a role
SI control
If an SI needs multiple cycles use following solutions:
pipelined: higher throughput but cost of register stages
multi-cycle-operation: allow input only each n cycles
state-machine: SI execution is stateful, each cycle, the state is proceeded
accompany RFUs w/ reservation stations to deal w/ variable length execution times w/o pipeline stalls

Instruction coding
SIs are generated @ compile time
they need an own opcode, sum of them is limited
SIs are divided into two parts.
format identifier: saying, there is an SI coming
SI " : which SI is meant
Can have an Address, that points to the start of a reconfiguration bitstream
Can have an instruction number, that identifies an SI by entries in a table
more bits needed -> fewer available for operands, but more different SIs supported
number of SIs limited by table size, more complex to get the address, but more bits available for operands
Virtual SI identifiers
provide dedicated 5-bit register
SI identifier corresponds to the concatenation of this register entry & the binary bits
Needs helper Instructions to read / write to that register
The resulting SI ID can be used as bitstream address or as table pointer

Operand passing
hardwired: op code only has dest. field, inputs will always be the same
Fixed operand coding
positions & length fixed
Flexible operand coding
opcode specifies length & position, needs a table lookup

Register File access
SIs can have a dedicated register file, or share one w/ other FUs
Data needs to be transferred to the dedicated one
most tightly coupled solutions prefer a shared one

SI detection / generation
manual identification: in C-code with pragmas
static identification: compiler decides, not performant if user input @ runtime is crucial for performance
dynamic identification: profiling @ runtime, time consuming, but mostly the best performance, relies on significant and relevant input data
Identification => analysis of the control / dataflow graph
create SIs by grouping instructions
Performance estimation takes into account: parameters, instruction latency, area overhead, reconfiguration time
Instruction performance check: checks if there was a speedup with the SI

Instruction Merging
Multiple SIs share one configured RFU, saves time & space, but only one SI can run @ a time

Instruction selection
ASIP: globally optimal instructions
here: multiple locally optimal " & schedule the reconfigurations along the control flow graph

Mark the intermediate representation of a program with information for the compiler
the backend has to take care of scheduling and assigning operand registers
can be done by hand, too => inline assembly
In case of asynchronous CPU and coprocessing: insert synchronization commands

Thrashing
loop w/ 2 SIs that don't fit in the FPGA: permanent reconfiguration
Solution: predetermine the SIs done by the FPGA
needs to be reviewed, if size of FPGA changes (like recompilation @ VLIW processors)
Other solution: provide "fallback" in core-ISA (cISA)
unimplemented-exception triggers execution in coreISA decided by a runtime-system
Flow: thrown exception → Trap table, identify trap → trap handler, identify SI ID → cISA → return → return → return ∑ 38 cycles, if exception is thrown
cISA execution may bridge a reconfiguration time
Other solution:
Conditional branch, that checks if an implementation is available, ~ 6-10 cycles @ every SI-call
Parameters for choosing between cond.branch or TH:
how often is the SI executed?
how long is the SI execution time?
SI executed often & cISA slow? → trap handler
SI executed relatively seldom & SI execution time short? → cond.

Trap handler needs to fetch operands of the SI call & identify SI
=> reread the instruction word
→ extract the 10 bit SI ID (load, and, shift)
load the 5 bit from the dedicated register, shift it, or it
Same for the parameters ... large OH
there are helper functions to improve this

4 Prefetching

Reconfiguration time: PRISC : 100 - 600 cycles
XiRISC : 128 + cycles
Garp : ~ 50 µs
Molen : 2 - 12 ms

If an SI is to be executed, but it is not configured, significant performance drops occur.
If there is free space in the FPGA, and the call of a SI is predicted, start the configuration in advance
=> Prefetching (can evoke thrashing !!)
Goal: minimize performance loss
Prefetching can target the FPGA directly or the configuration cache

Parameters for Prefetching
- Temporal Distance
- Probability
- # SI calls

} Depends on control flow

Abort prefetching in case of false-prefetching
If it hadn't started: remove from prefetch-queue
otherwise wait until it is done (row based FPGAs can abort after row completion)

Static Prefetching

@ compile time: embed prefetching instructions in to the code => static decision

needs profiling before compilation

sort all SI that are reachable @ every node in CFG by probability of execution

going upward again, add to every node in CFG the reachable SIs

take into account the size of the FPGA

eliminate redundant prefetching instructions

Clock Frequency Variation

static prefetching works well, if the CFG is well known @ compile time & user input isn't crucial for it.

=> Reconfiguration time & execution time (RT & ET) are well-known

ET can be increased, if the following instruction needs to wait for a configuration completion => lower clock frequency

# of different clock nets limited => cluster clock frequencies to groups

Dynamic Prefetching

If CFG doesn't provide reliable edge probabilities, use dynamic prefetching, since static decisions may screw up

idea: learn from previous executions => adaption @ runtime

requires online monitoring to obtain feedback about actual executions

Hybrid Prefetching
combines static & dynamic prefetching:
some decisions can be made better by the compiler / programmer

Prefetching is partitioned into two parts:
1. Forecasting the # of executions
2. Selecting the SI implementations (cISA or FPGA)

Error-back propagation approach change the prediction for the future
Taking into account the initial forecast, the actually monitored forecast and the new forecast => fine-tuning
α := parameter that determines the strength of error-back propagation ($\alpha \in [0,1]$)
$\alpha = 0$ no error-back-propagation => static prefetching
$\alpha = 1$ overwrite former forecasts
γ = strength of future prediction affecting the error-back propagation => shift back in time ($\gamma \in [0,1]$)
λ := weight how many forecasts are affected ($\lambda \in [0,1]$)

Dynamic Prefetching uses a static approach as starting set

Area Models
Need to consider, if the selected SIs fit into FPGA @ same time
=> Placement conflict => affects prefetching decision
Prefetching may not start too early
Area Model is given by the FPGA: some reduce the potential for conflicts @ cost of efficiency / flexibility
can also be reduced by relocatable implementations or configuration caches

2D area model
Highest Flexibility by highest complexity (routing / placing)
introduces external fragmentation (unusable gaps between the configured modules)

1D area Model
reduces external fragmentation, but introduces internal fragmentation
easy relocation
rather slim shape can make internal routing difficult

1D horizo scanning
only memorize the front line, can't see gaps behind configured modules

1D stuffing
finds the gap (complexer algo)

1D fixed-size area model
no external fragmentation
high internal "
Conflicts only occur when too many IS are demanded @ same time
easy communication connection

7 Reliability

FPGAs can adapt to deal w/ permanent and temporary faults

Types of faults

Permanent faults, can occur @ fabrication time w/o being detected, can also " " runtime due to aging

Intermittent faults, can appear due to temp being high/low

Transient faults, particle strikes can flip a bit

NTBI breakdown of Si-H bonds due to stress
cooldown after stress

HCI build up of trapped charges @ gate-channel-interface-region
progressive reduction of carrier mobility
V_{th} +, speed −

TDDB over time, conductive paths through dielectric emerge
→ short-circuit

All leads back to failure of demand's scaling

Single Event Upset (SEU)

In ASIC, not dramatic

In configuration of FPGA, catastrophic

Fault Detection

Modular redundancy
problem w/ error accumulation
radiation-proof voter needed

Concurrent Error Detection
parity data / checksums

RESO (Recomputation w/ shifted operands)

DWC (Duplicate w/ compare)

Offline BIST (Build-in-self-tot)

TPG (Test pattern generator) w/ ORA (output response analyzer)

Online BIST

divide FPGA into parts, some for testing, some for performing user functions

↳ may require clock speed reduction due to longer wires

STARs (Self-Testing AReas)

horizontal and vertical BIST Areas

roving

Fault tolerance approaches

1. STAR parking
2. if logic cell can still use the block, do not reconfigure otherwise remap logic cell to spare working block
3. STAR stealing: if no spares are left, shrink STAR try to maintain at least one roving STAR

Scrubbing

repairs configuration memory through updating it

via JTAG, ICAP or SelectMAP

can't scrub LUTs that are used as RAM

" " BRAM

Blind scrubbing

continuous overwriting, even if no SEU is detected

fast, easy, minimal additional HW

Readback scrubbing

Read configuration memory contents

on error, rewrite configuration memory

can log SEUs

Internal scrubbing
read configuration frame from ICAP and apply CRC
1-bit correcting, 2-bit detecting

Partial Reconfiguration scrubbing
used for partial reconfiguration fault ~~detection~~ correction
previously introduced scrubbing would 'repair' the newly partly configured area
golden-bitstream: reference bitstream in radiation-hardened memory that PR and scrubbing write/read

3 Fine-Grained Reconfigurable Processors
PRISM loosely coupled (own board)
16bit Interconnect
Reconf time ~1s
proof of concept
PRISM II
GCC Frontend /VHDL
Coprocessor-Like fabric
64bit Bus faster data movement
3 XILINX FPGAs
GARP
Aimes to overcome: - reconfiguration OH
- Memory accesses
- binary compatibility
Just a draft, one chip
RF is coprocessor-like, main core still needed to be changed
SW-controlled runtime configuration
asynchronous to core

2D-Mesh of blocks (24 x)
partially recont. supported, but blocks all rows
Instruction set extensions for programming
Compiler detects kernels and places/route HW
hyperblock scheduling to increase ILP
building new DFG

MOLEN

Coprocessor, inspired by Garp
Reconfigurable micro-code (PM-code)
p-set / c-set
helper instructions for executing, reconfiguring, moving data...
p-set, c-set, pretetch, break, execute-pretetch

PRISC

tightly coupled FU, small Reconf. logic
one new instruction: EX FU, same timings, integrated in pipeline
11bit FU-addressing => max 2048 FUs
targets logical expression, table lookup, branch prediction
optimization, loop & jump optimization

OneChip

tightly coupled, supports more complex SIs than PRISC
1 large PFU, but memory contains multiple configs, so
it can be changed quickly & on demand

OneChip98

extension of OneChip with memory access for
the FPU
out-of-order superskalar execution
multiple PFU

XiRisc

VLIW processor w/ RFU

5 stage - RISC pipeline

Pipelined configurable gate array PiCoGA

multi-cycle-operations & stateful-computation

XiSystem

eFPGA added, 1bit granularity

5 Coarse-Grained Reconfigurable Processors

Chameleon SoC

coarse grained, but effort is done to improve efficiency

FPGA / ASIC / GPP connected via NoC

Montium core on chip to accelerate DSP-like tasks

Processing Part Array made up of SRAM, Regs, ALU, Bus

Sequencer that handles the flexibility of the PP array results in a vast amount of control signals ⇒ sequencer hierarchy

problem: tool-chain & compiler ⇒ kernels hand-mapped

ADRES

tightly coupled reconf. fabric & core cpu (VLIW core)

2D array of FUs

instructions first pass the VLIW core, then enter the RFU

compiler: DREAC uses if-conversion & hyperblock generation

Initiation Interval, filling tasks onto RFUs spreading over iterations

Investigation of Mesh / #multipliers / #mem-ports / register file placement & amount

MT-ADRES (multi-threaded)

also exploits task-level parallelism

split the ALU-Array into sub-arrays (recursively also possible)

assign sub-blocks to tasks
statically prepared @ compile time

PipeRench

Co-processor for data-stream applications
Fast reconfigurations (pipelined!)
runtime scheduling for HW utilization
virtualizes HW, enables later upgrades easily
pipelined reconfiguration maps larger HW-designs onto its smaller HW
pipeline stages become part of large HW configuration
constraints like in common pipeline
- data dependencies
- can't skip stage
- can't communicate with previous stages

6 Adaptive Reconfigurable Processors

RISPP

tightly coupled, fine grained
modular SIs
runtime system decides which reconfiguration shall be performed and when to perform them
SI-containers
'Atom's: smallest block that can be reconfigured, computational data path
'Molecule' implementation of a SI, made up of Atoms
Atoms can be shared and added to improve performance
0 Atoms => cISA execution

Runtime System for RISPP

Decode: detects SIs in binary & makes forecasts & triggers prefetching

Execution control: determines currently fastest implementation of an SI and triggers execution or trap handler

Monitoring: counts execution of each SI

Prediction: for fine-tuning of forecasts of prefetching

Selection: select molecules to implement the SIs

Reconfiguration Sequence scheduling: determine the recont. sequence of the atoms that are required

Replacing: determines Atoms to be replaced, if schedule designates (Instruction set Selection) → NP-hard!

Problem of selecting Atoms ~ similar to knapsack-problem use greedy algos, not the standard one, it might implement SI multiple times & costs for an implementation depends on other choices

- remove all other SI implementations, that are already covered
- recalc weights after adding molecule

After O(n) iterations, the first molecule is set, and recont. may start

Reconfiguration Sequences:

FSFR (First select, first reconfigure)

selection order is reconf. order + relevant first - other SIs don't have speedup for long time

ASF (Avoid software first)

always choose Molecule w/ least missing atoms + all SIs might be in HW quickly - valuable SIs may still wait a long time

SJF (smallest job first)
always choose the nearest HW update
- not considering #SI executions or benefit from SI

HEF
always choose Atom w/ highest benefit / upgrade cost
+ good :)

Ersetzung von Atomen (auf einmal Deutsch ...)
Bélády's algorithm: longest time not used will be replaced: knowledge of future not available
MinDeg algo used

WARP
Fine grained, loosely coupled coprocessor
no compiler required, works on binary
detects hotspots during runtime & implements SIs online
Flow: 1. Execute binary normally
2. profiler detects critical regions
3. partition them in HW
4. configure RFU & update binary
(5. ???
6. profit)

Small cache for tracking sbbs (short backward branches)
on Miss: add entry to cache
LFU-replacement strategy
counter overflow: half all other entries (shift to right)
non-intrusive, binary does not need to be modified
optimization possible: store counters in another cache so that counting does not access the whole instruction caches

Synthesis tool needs to execute online
memory & runtime overcome: W-FPGA is simplified
smaller LUTs, less LUTs per CLB, simplified PSM
algos are leightweight greedy
hard wired components like Data Address Generator &
Loop Control Hardware, MAC, ...
each CLB has 1 PSM, nearest neighbor & second
nearest neighbor (4 short - 4 long channels)
Online Synthesis:
1. Decompilation (binary → CFG)
2. Partitioning (select critical kernels)
3. HLS (create RTL)
4. LLS (place & route)
5. Binary update (overwrite jump to old execution w/
jump to SI, SI handler stalls pipeline
Routing: Riverside-on-chip-router
assign routing costs & route them in a channel w/ least
costs. ignore all other routes, rip up illegal ones,
route the one w/ highest priority / cost
re-route all others again
build conflict graph / make graph coloring
overall: needs to run for a long time (hours...) to achieve
good speedups

DIM

tightly coupled, course grained
works on binaries, no extra compile needed
on-the-fly Synthesis, no lengthy synthesis algos
creation of SIs during execution of normal instructions
+ caching of SIs

BT
- starts @ instruction after branch
- stops @ another branch or unsupported instruction
- in between: each instruction is placed on the reconfigurable array, on the fly extention w/ every assembly instruction
- if # instructions found > 3 : cache the SI

BT ≙ binary translation

hot spot is executed, gets cached, and will stay there, because it is executed often

Coarse grained Array of DIM consists of ALUs, LSUs, Mults, ... connected by MUXs

- Limited room for optimization on-the-fly
- depends on compiled code
+ no customer overhead