Hardware acceleration of sequential loops

Pierre Michaud

ALF team
INRIA Rennes – Bretagne Atlantique

December 12, 2011
Outline

1. Sequential accelerators

2. Hardware acceleration of sequential loops
Monster core

• Let’s assume we are given a large silicon area and power budget and we seek to maximize single-thread performance.

• What silicon area and power would we need to double the performance of a 4-wide superscalar core?
  – 4 times the area and power of a 4-wide superscalar?
  – 8 times?

• What about a monster core 128 times bigger than a 4-wide superscalar?
The multi-core era

2005 : superscalar

2009 : 4 cores

Small single-thread performance improvement
The multi-core era

2013 : 16 cores

Small single-thread performance improvement
The many-core era

2017 : 64 cores ?

Small single-thread performance improvement
The many-core era

2021: 256 cores?

Small single-thread performance improvement
What do we choose for 2021?

Do we prefer double parallel performance (maybe) or double sequential performance?
How do we build a monster core?

• 8-wide superscalar?
  – Probably the first step, but that does not make a monster core.

• 16-wide superscalar?
  – Never implemented so far.
  – Can we increase the IPC without impacting the clock cycle?

• Huge NUCA cache?

• What else?
Sequential accelerator

• In the remaining, “core” = “superscalar core”

• Sequential accelerator may consist of several physical cores

• Sequential accelerator is specialized for sequential execution
Proposition A

- Build a very aggressive superscalar core
  - 8-wide, high clock frequency

- Replicate that core (e.g., 8 times)

- Use a single core at a time, power-gate the unused ones

- Migrate the execution periodically to maintain temperature at a safe level
  - P. Michaud, Y. Sazeides, A. Seznec, “Proposition for a sequential accelerator in future general-purpose many-core processors and the problem of migration-induced cache misses”, Computing Frontiers 2010
Proposition B

- Hardware acceleration of sequential loops

- The focus of this presentation
  
  - http://hal.inria.fr/hal-00641350
Hardware acceleration of sequential loops
Dynamic loop

- **dynamic loop = periodic sequence of dynamic instructions**

- Dynamic loop ≠ program loop
  - A program loop does not always generate a dynamic loop
    - control flow behavior inside the loop body may be more or less random
    - e.g., “if” statements inside the loop body
  - A dynamic loop may result from a recursive function
Idea

• Why not build a hardware accelerator for dynamic loops?
  – Loop detector = hardware mechanism for detecting dynamic loops
  – When dynamic loop encountered, migrate the execution to the accelerator
  – When loop exit condition (e.g., branch direction changes), migrate the execution back to the superscalar core
Let’s kill the suspense

percentage of performance increase over baseline
Definitions

- loop length = sequence length (in instructions)
- loop body size $B = \text{smallest period}$
- loop body = first $B$ instructions
Some questions

• What fraction of time programs spend in dynamic loops?

• Are dynamic loops long enough to amortize the penalty of migrating the execution to and from the accelerator?

• What is the typical loop body size?

• First let’s define the loop detector
Loop detector

• Loop detector = loop monitor + loop table

• Loop monitor
  – Observe instructions as they retire from the processor reorder buffer
  – Detect **long** dynamic loops
    • when loop length > 900, count subsequent instructions as loop instructions, until periodic behavior ends

• Loop table
  – Records information about loops already encountered
Remarks

- A loop body may contain several instances of the same static instruction
  - Tiny loop unrolled, same function called multiple times, ...

- There is a **backward jump** somewhere in the loop body

- A loop body may contain several backward jumps
Loop monitor

• Associate a monitor to each static backward jump

• For each retired instruction
  – Update a signature = hash of instructions PCs
  – Increment the body size
  – If body size > MaxBodySize, close the monitor

• Each time we encounter that same backward jump
  – Compare the signature with the body signature
  – If signatures match, add body size to the loop length
  – Reset signature and body size

• Monitor several backward jumps simultaneously
  – 4 monitors are enough in practice; that which first reaches 900 “wins”
Loop table

• If loop length was greater than 900, record entry for that loop in the loop table

• Loop is identified by its body signature

• Next time we encounter that loop, immediately start counting instructions as loop instructions
  – don’t wait till 900 instructions have been retired
Simulations

• Trace-driven, based on Pin (x86)

• Nehalem-like microarchitecture (roughly)

• Hardware cache prefetchers
  – L1, L2, L3

• Benchmarks: SPEC CPU2006
  – 1 trace per benchmark
  – 1 trace = 40 samples of 50 millions instructions (total = 2 billions)
  – Samples regularly spaced throughout whole benchmark execution
CFP2006 loop behavior
Summary

• Long dynamic loops represent a significant part of the execution for half of the CPU2006 benchmarks
  – Up to 90%

• Typical loop length is several thousands instructions

• Body size is diverse, from a few tens up to a few hundreds instructions

• Worth trying to accelerate loops
  – But all loops, with a small or large body
How to accelerate loops?

• Modify a superscalar core?
  – instruction fetch (done), register renaming, what else?

• Loop accelerator
  – Sits beside the superscalar core
  – Specialized for executing periodic sequences of instructions

• Try to remove repetitive work from the loop execution ➔ do that work before executing the loop
  – E.g., register dependency analysis
  – When possible, store information in the loop table for future occurrences of that loop
Global view

- **Loop Detector**
- **Loop Builder**
- **Loop Accelerator**
- **Superscalar Core**
- **DL1**
- **Arch. Regs**
- **L1**
- **L2**
- **L3**
- **Retired Micro-ops**
Transitions

Normal mode:
- execute on core

Build mode:
- for X loop iterations
- still execute on core
- prepare loop

Acceleration mode:
- execute on loop accelerator
  - flush reorder buffer
  - read loop input regs
  - migrate the execution

long loop detected ➔ early loop exit ➔ loop exit ➔ update arch. regs
Proposed loop accelerator

- Get rid of main superscalar bottlenecks
  - Register renaming, dynamic scheduling, bypass network,…

- Scheduling is done during build mode
  - Each static micro-op is mapped to an Execution Unit (EU)
  - All dynamic instances of that micro-op will be executed by that EU, in program order

- Static micro-ops communicate through FIFO queues
  - No registers
  - Data-flow execution
Cyclic register dependency graph

example loop ➔ body = 5 micro-ops

cycles in the graph = long dependency chains

latency-tolerant dependency
Constraint edges

dynamic instances of a static micro-op execute in program order

this does not increase the length of the longest chain
Proposed loop accelerator (cont’d)

- 8 nodes
- 4 EUs per node
- 4-lane pipelined ring
Proposed loop accelerator (cont’d)

- 8 nodes
- 4 EUs per node
- 4-lane pipelined ring

1-cycle micro-ops (ALU, branch, address generation, ...)

S-node  
\[\text{I-node} \rightarrow \text{I-node} \rightarrow \text{L-node} \rightarrow \text{F-node} \]
Proposed loop accelerator (cont’d)

- 8 nodes
- 4 EUs per node
- 4-lane pipelined ring

FP add, FP mul, int mul,…
Proposed loop accelerator (cont’d)

- 8 nodes
- 4 EU per node
- 4-lane pipelined ring
Proposed loop accelerator (cont’d)

- 8 nodes
- 4 EUs per node
- 4-lane pipelined ring

stores
Remarks

• Implement only a **subset of the ISA**
  – Some operations are rare in the loops we have found
  – E.g., integer division

• Build mode determines if the loop cannot be executed by the accelerator

• **Floating-point divisions are not rare**
  – One specific I-node can be configured during build mode to execute FP divs only
I-node / F-node

lane 0
IQ

DISTRIBUTOR

lane 1
IQ

lane 2
IQ

lane 3
IQ

operand Qs

lane 0
CEC
EU

lane 1
CEC
EU

lane 2
CEC
EU

lane 3
CEC
EU

OQ

LEQ

lane 0
lane 1
lane 2
lane 3
Lane 0

Up to 12 static micro-ops may be mapped on that EU

The Cyclic Execution Controller schedules them in program order
Lane 0

Dependent micro-ops can be mapped on the same EU
**Input Queue** buffers copies of values traveling on the lane

The IQ must never get full → early loop exit when this happens
Values coming from other nodes are stored in one of 12 Operand Queues.

A given queue holds up to 32 values from a single static micro-op.
Values produced by the EU are sent to other nodes through the Output Queue.
Thanks to the **distributor**, copies of values produced on a lane can be sent to EUs on other lanes.
Loop Exit Queue contains last 128 values produced by the EU

→ for updating registers after loop exit
L-node, lane 0

- CEC
- LIB
- LEQ
- LOQ
- DL1 cache
- MSHRs
- Port 0
- Port 1
- Port 2
- Port 3

operand Qs
Load address comes from an I-node
L-node, lane 0

64-entry Load Issue Buffer ➔ select one load per cycle
L-node, lane 0

Send data to lane 0 in program order
S-node, lane 0

- **Operand Qs**
- **Store address**
- **Store data**
- **CEC**
- **EU**
- **LSQ**
- **64-entry Loop Store Queue**
Validates stores in program order

A validated store can write in the DL1

Global synchronizer updates an *executed iteration count* that tells which stores can be validated
Global synchronization

- At a given time, different EUs may be in different loop iterations

- Make sure that no store is validated before older instructions have been executed

- When a loop exit occurs, make sure that the values needed to update the architectural registers have not been pushed out of the Loop Exit Queue
  - Prevent an EU to go too far ahead of the others
Global synchronizer
Global synchronization (cont’d)

- Each EU has an iteration counter (IC)

- EU freezes when IC = ICMAX
  - ICMAX not the same for all EUs
  - For the S-node, ICMAX = ICMIN
  - For other nodes, ICMAX ≥ ICMIN

- When the IC reaches ICMIN, the EU sends a signal to the synchronizer

- After the synchronizer has received a signal from each EU, it sends back an unfreeze signal to all EUs

- Upon receiving the unfreeze signal, each EU subtracts ICMIN from its IC, and the store validator adds ICMIN to the executed iteration count
Loop exit

• Typical loop exit ➔ a branch changes its direction

• An EU that detects a loop exit condition sends a signal to the synchronizer
  – The signal contains the loop exit point = the IC value and the micro-op rank inside the body

• The synchronizer forwards the loop exit point to all the EUs and to the store validator
  – EUs freeze when they reach the loop exit point
  – Store validation stops at the loop exit point
Memory dependencies

- Instruction window can be very large
  - example: body size = 100, ICMAX=32 $\Rightarrow$ 3200 instructions

- Exploit loop properties
  - spatial locality is often very good
  - constant-stride accesses are frequent
  - store-load dependency generally repeats on each iteration
Memory independence checker

• Accessed by loads and stores

• At store validation, tells if the store might conflict with an already executed load
  – exploits spatial locality and constant-stride accesses to keep structures reasonably sized
  – see tech report for details

• If answer is no, everything is ok

• If answer is yes, may be a memory order violation → trigger an early loop exit
  – the store is a safe loop exit point
Store-load bypassing

transform the loop body during build mode
Memory dependency (cont’d)

• Assumes distance between store and load is less than a loop body
  – sufficient in most cases
  – similar method can be used for distances between 1 and 2 bodies
    • beneficial only for 456.hmmr

• ‘check’ micro-op executed by I-node, triggers early loop exit if fails

• Must verify that stores between the dependent store-load pair do not conflict

• ➔ Bypassed-Store Table accessed by each store at validation
  – one entry per bypassed store
  – if conflict detected ➔ early loop exit
Loop body reduction

- Micro-ops with no input dependencies inside the loop body produce the same result on each iteration
- May remove these redundant micro-ops during build mode

- Recursive definition: micro-ops depending only on redundant micro-ops are themselves redundant
  - they produce constant results after a fixed number of iterations

- Keep redundant stores (memory consistency)

- Can remove redundant loads, but record their address in a Removed-Loads Table check conflicts at store validation

- Typically, between 10% and 20% of micro-ops can be removed
Static mapping

• On which EU do we put a static micro-op?

• Very important for performance
If we put micro-ops A and C on the same EU, we create an artificial CRDG cycle of size 3.
Artificial CRDG cycle

CRDG cycle may increases loop execution time very much

When possible, try to avoid creating artificial CRDG cycles
EU sharing

Iteration rate is limited by the most loaded EU
EU sharing

Iteration rate is limited by the most loaded EU

Try to balance number of micro-ops per EU
Lane segment sharing

Iteration rate is limited by the most loaded lane segment
Lane segment sharing

Iteration rate is limited by the most loaded lane segment

Try to balance number of dependencies per lane segment
Mapping heuristic

• Try to balance EU sharing
  – important for loops with small body

• Try to detect natural and artificial CRDG cycles
  – artificial CRDG cycles can be avoided if detected
  – natural cycles $\Rightarrow$ try to put micro-ops on same EU

• Try to put dependent micro-ops on the same EU
  – $\Rightarrow$ that dependency does not use any lane segment

• Try to minimize number of lane segments used
Configuration simulated

- Loop detector \textit{MaxBodySize} = 128
- Build mode \rightarrow 5 iterations
- Transition penalty
  - loop table hit \rightarrow 100 cycles
  - loop table miss \rightarrow 10000 cycles
- Loop accelerator
  - 4 lanes, 8 nodes
  - 4 loads / cycle
  - 2 stores validated / cycle
  - up to 12 static micro-ops per EU
  - 12 operand queues per EU (queue depth = 32)
  - load issue buffer \rightarrow 64 loads
  - unfreeze signal latency = 4 cycles
Performance

percentage of performance increase over baseline
Local loop speedup
Impact of disabling features

percentage of performance loss when disabling a single feature
Some future research

• How does performance scale if we increase the number of nodes and/or the number of lanes?
• Can we over-clock the loop accelerator?
• Is a pipelined ring the best choice?
• Is it a good choice not to provide any direct data path between EUs on same node?
• What about applications other than the SPECs?
• How could a compiler exploit a hardware loop accelerator?
Poor man’s loop acceleration?

- Some ideas may be useful even if you don’t buy the loop accelerator
- ➔ loop detector
- ➔ loop body reduction
- ➔ store-load bypassing

Can we improve dynamic instruction scheduling in a superscalar core if we know that we are in a dynamic loop?

you pay the overhead only once for several thousands instructions
Conclusions

- **Long** dynamic loops are frequent
  - > 30 % of all the instructions executed by the SPEC CPU2006 suite

- Try doing the repetitive work only once, before executing the loop

- The proposed loop accelerator avoids some important superscalar bottlenecks
  - no registers, no register renaming
  - no dynamic scheduling
  - huge instruction window ➔ latency tolerance

- This is preliminary research, many questions still pending
Questions ?