Transmeta Crusoe: Hardware, Software, and Development

Pardo

AMAS-BT 2009

Transmeta Crusoe

Crusoe: The First Commercial
Dynamic Translation Processor

Original Vision

- x86 is the industry-standard ABI
  - But x86 processors are complex
  - And VLIW is simple, cheap, & fast – but not x86
  - Shade: simulation is “fast”
- So:
  - Build a simple, cheap, fast VLIW
  - Run x86 via simulation
  - Develop HW & SW in parallel
    - Sooner hardware is faster hardware

Today's Talk

- 10,000 metre overview
  - Hardware
  - Code Morphing Software (“CMS”)
  - Supporting software
  - Experiences (“story hour”)
  - Q&A

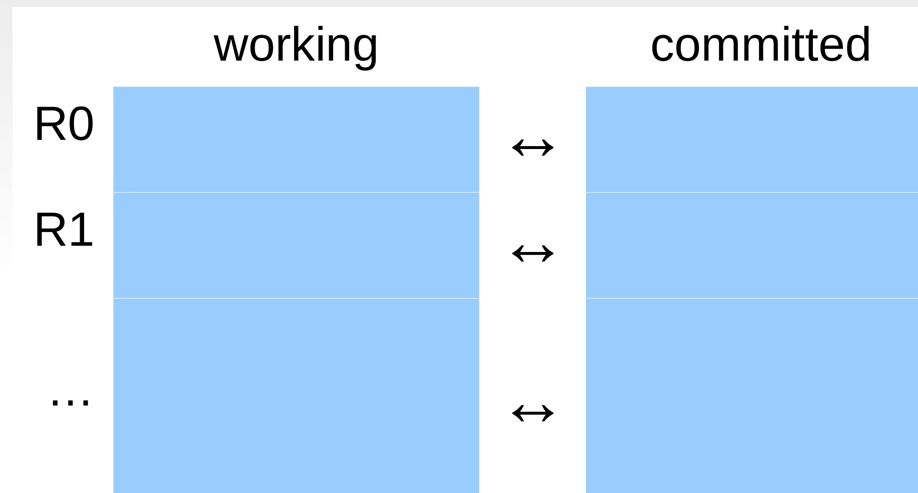
- Today: Crusoe
- Not today: Efficeon

Crusoe Hardware

- VLIW
  - 5-deep pipe
  - 4-wide issue
  - Fast clock (almost)
  - Small chip
- Plus simulation support
  - Generic simulation support
  - x86-specific support
- No x86 instruction decoding (100% simulation)

Generic Simulation Support

- “Shadowed” registers, commit/abort
- Commit: →
- Abort: ←



- Gated store buffer: drain on commit
  - “Shadowed memory” without extra RAM
- Speculate to go fast; if unexpected, abort

More Generic Support

- Goal: out-of-order execution
- Software speculatively reorders loads & stores
- Simple alias hardware: “load and protect”
  - Save address
  - Compare against other loads & stores
- Hardware trap if speculation fails while running
  - Software deals with failed speculation

x86-Specific Support

- x86 condition codes
    - Carry = ...
    - Overflow = …
    - …
    - Varies by instruction
- Complicated to simulate in SW
- “Free” for SW if done in HW
- Non-x86 targets: simulate CCs in SW

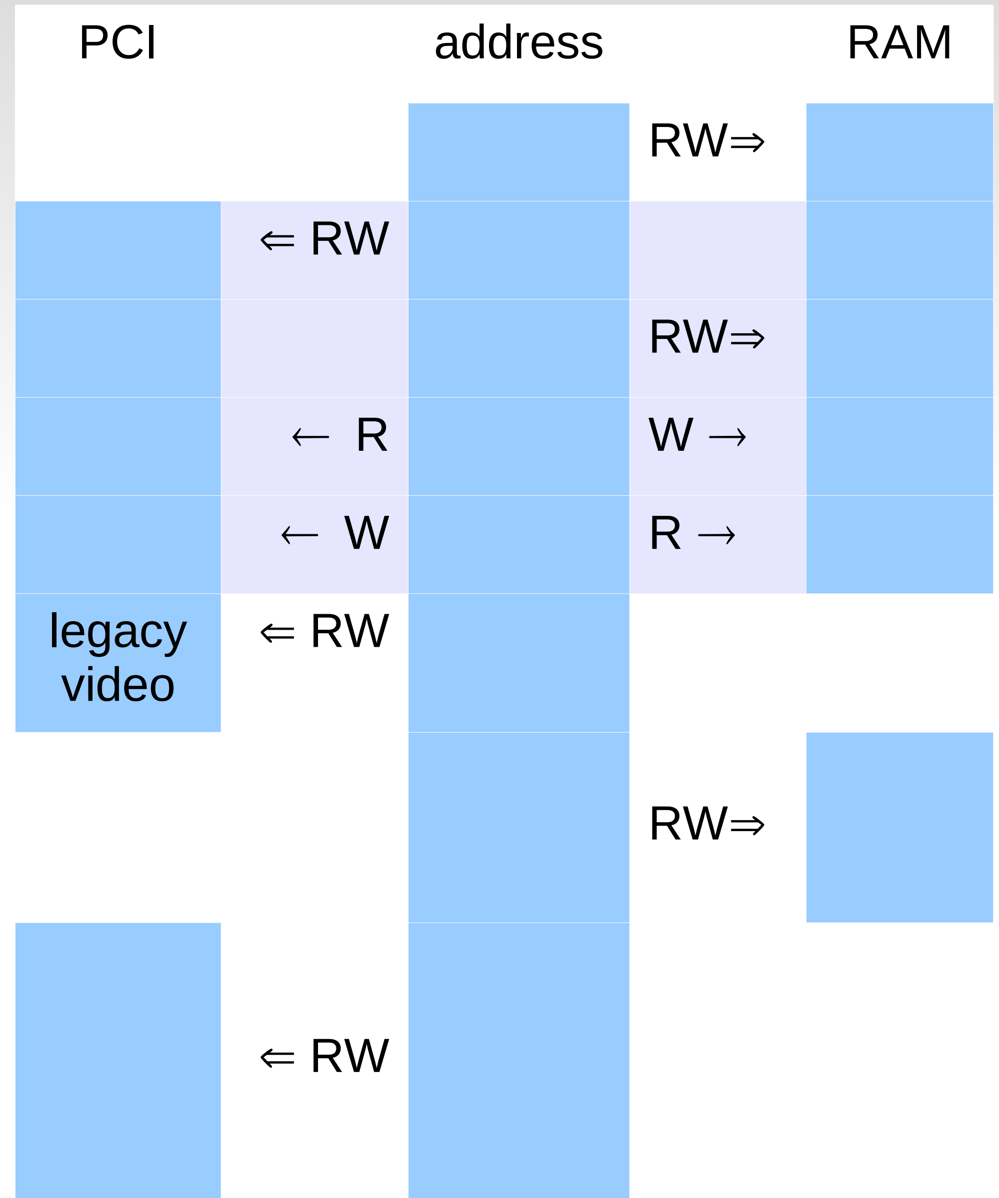
More x86 Support

- Low memory: A20M

  Some addresses:

  Read: memory or I/O

  Write: memory or I/O
- Steered separately!
- Holes!
- Hardware steering
  - Software control



Low Power

- Simple: low power
- x86 instruction decode
  - P4: 1/4 of area, 1/3 of power (worst case)
  - Crusoe: x86 decode in software...
    - Time spent decoding is power *and* performance
    - Good overall as long as decode is “not too often”
- LongRun
  - Drop the frequency → drop the voltage
  - $P \sim V^2 F$: 90% CPU speed → 70% dynamic power
  - Over 90% performance: memory stays at 100%

Software: Simulate The Whole x86

- Code Morphing Software (CMS)
- Crusoe's "Microcode"

- Original vision: simple translator
- Shade
  - Translation: 100 I/I
  - Performance: 3:1 integer, 1:1 FP

Simulator Complications

- Compared to Shade:
  - x86: more complex behavior, SMC harder
  - User + kernel
  - VLIW scheduling
  - Reuse rates (or lack thereof)
- Crusoe translation: 10,000 I/I
- Peak performance: often better than 1:1 int/FP
- But: translation costs, memory bottlenecks, ...
- Performance: varies with time, application
- Humans remember “slow”

Results (Generalizations)

- Very good reliability
  - New HW, new SW, new strategy, new people
- Cost: good
  - ~1/2 Intel/AMD parts
- Power: good
  - ~1/3 Intel/AMD parts
- Performance: ummm...
  - Next slide!

Performance Generalizations

- Crusoe much faster than low-power parts
- But: a lot slower than Intel 15W mobile parts
- Compute-bound: often faster at much lower Watts
- Memory/cache traffic: slower
- Low reuse: translation overhead $\rightarrow$ slower
- PCI graphics, not AGP
- Non-overlapped compute and I/O
- Humans notice delays, not asymptotes
- Variable: within and across applications

How To Go Faster?

- Lower translation cost
- Faster translations
- Faster memory
- Faster graphics (AGP vs. PCI)
- Overlap compute and I/O (DMA in SW)
- Faster VLIW – more Hz, more issue width

- Efficeon (Crusoe successor) much better!

Performance (More Generalizations)

- Crusoe FP was fast...

  2000 Crusoe CPU-bound FP ~ 2009 Atom

  At same Watts

- 2004 Efficeon ~ 2009 Atom

  At same Watts

Supporting Software

- Crucial to making Crusoe!
  - Reference Simulator
  - Fast VLIW simulator
  - Tests
  - Farm
  - Debug tools
  - Build Tools
  - Performance Tools
- Theme: automate, automate, automate

Reference Simulator

- Defines “What is an x86”
- Standard of comparison for CMS+VLIW
- The standard changes constantly
  - The reference is hard to pin down...
- Correctness is important
- Speed is important
  - Boot and run tests, OSes, etc.
  - Often before trying on CMS+VLIW

Fast VLIW Simulator

- Run CMS years before working HW
- Remove CMS workarounds before fixed HW
- Reproducible debugging
    - “Software leads hardware”

- ~30 I/I – running on an x86, simulating VLIW
- Lots of useful features (below)
- Itself a study in fast simulator construction

Feature: Narrowing

- Simulators support checkpoint and restart
- “Reverse execution”: gotox *NUM*
- “Cosimulation”: run two simulators together

  Run N x86 instructions on each

  Stop and compare all state
- “Nexus”: binary search for first difference
  - Show exactly which bits diverged and why
  - Automatic
  - Often: fix it right now

Tests

- Conventional: hand-written VLIW, x86
- Unconventional: pseudo-random tests
  - Biased random – guide to “interesting” cases
  - But still many benefits of randomness
  - Instruction-level, system-level
- “Test” means “checkable”
  - Wrong answer not detected by benchmark
  - A crash is obviously wrong!
  - Divergence under cosimulation
- ~Everything is a test

Farm

- Machines
  - PCs to run simulators
  - Real VLIW hardware
- Automation infrastructure
  - Allocate any machine for any purpose
  - Reallocate with any CMS, BIOS, disk image
  - Live debug of CMS error from 1,000 km away
- More automation
  - Run to failure, snapshot, human gets “later”
  - Gets the failing instruction, which bits are wrong

Debug Tools

- VLIW HW debugger
  - UI same as VLIW simulator debugger
  - State save immediately on reset (after crash)
  - Download state for offline examination
  - Single-step through nested fault handling
- (Reverse HW execution started, not finished.)

Build Tools

- Fast build/test server
  - Full and incremental are run in parallel
  - Check in early and often
  - Fast feedback – a few minutes
- On failure: binary search of checkins
  - Automation

Performance Tools

- CMS instruments translations (Shade!)
- Postmortem “fly over” of whole execution
- Visualize big trends across time: retranslation rates, I/I efficiency, cache/memory stalls, I/O, code paths…
- Telescope “zoom in”
  - Some: phases with similar trend data
  - Lots: trace data for individual translations
- Fast
- Find lots of performance bugs quickly

Story Hour – Lessons

- New HW, new SW, new tools, modified BIOS
  - Bound to be lots of stories
  - Some unusual & educational experiences
- Bug1: MS-DOS boot – ~30,000 instructions in
  "The timer interrupt handler is called too often"
  Hardware bug, change CMS to work around it
- Some "critical" HW bugs made invisible to user
- Keep working in parallel w/ HW fix & tapeout

Story Hour 2

- Bug2: ~100,000 in: state smash on interrupts
- CMS nonshadowed resource rollback hazard
  - read X ; write X ; rollback ; read X
- After rollback: still have new X. Oops!
- Fix
- Add rule checkers
  - “Never again.”
  - Everybody “learns” from one mistake

What Would I Do Different? Technical Lessons

- Getting reliable HW was a big delay
  - I thought software would be the problem!
  - Experienced HW team, but Crusoe different
    - ISA not fixed, changing (for performance)
    - Big project “rules of thumb” don't work so well
    - Interactions different in a small team
- Better performance studies from “go”
  - Is “Problem X” due to x86, VLIW, CMS, ...?
  - (Fewer late HW changes!)
- More software inspection

Summary

- Crusoe:
  - Met many requirements and goals
  - Good reliability, needed better performance
  - (Efficeon)
- Support – Crusoe is only half the story
  - Tools are crucial
  - Automation is crucial
- Paper: lots more details
  - Including more “story hour”

Conclusion

Crusoe: The First Commercial Dynamic Translation Processor