Operating System Support for Shared-ISA Asymmetric Multi-core Architectures

Tong Li, Paul Brett, Barbara Hohlt,
Rob Knauerhase, Sean McElderry, Scott Hahn
Intel Corporation

Contact: tong.n.li@intel.com

Overview

- Asymmetric cores for future CMPs
  - Different frequency, cache size, instructions, in order vs. out-of-order, etc.
- Can software handle and benefit from asymmetry?
  - Apps, compilers, libraries, OS, etc.
- Focus: OS support for asymmetry

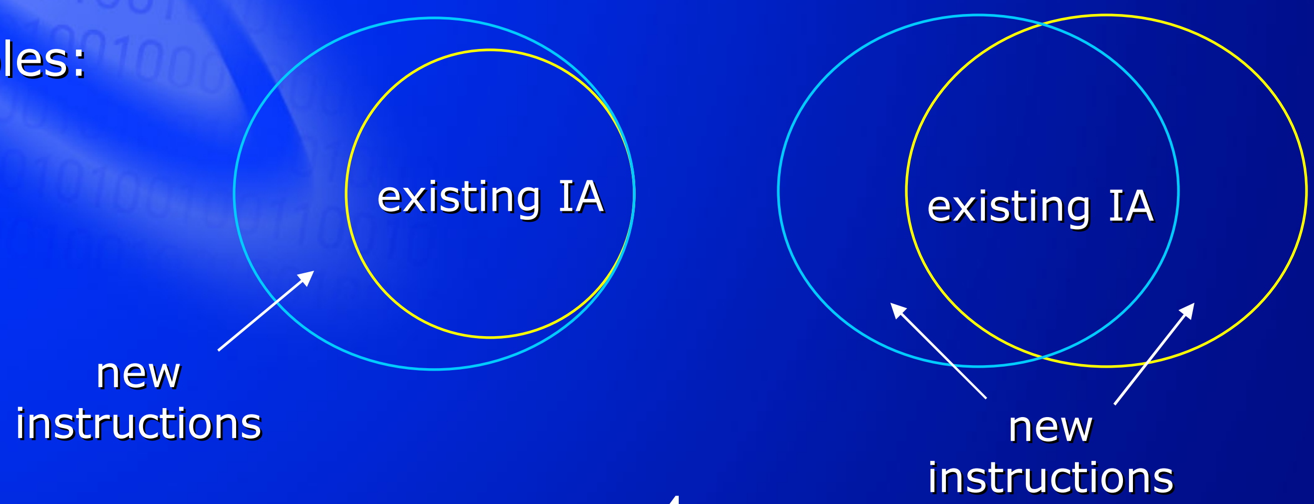
Outline

- Asymmetric architecture design space
  - Types of asymmetry
  - Hardware-software interface
- Case study of OS support
  - OS design and implementation
  - Evaluation
- Conclusion

Types of Asymmetry

- Performance asymmetry
  - Different core speed, cache size, uarch, etc.
- Functional asymmetry
  - Disjoint ISAs, overlapping ISAs (small vs. big overlap, …)
- Focus on instruction-based functional asymmetry
  - Cores have overlapping, but non-identical instruction sets

Examples:



HW-SW Interface

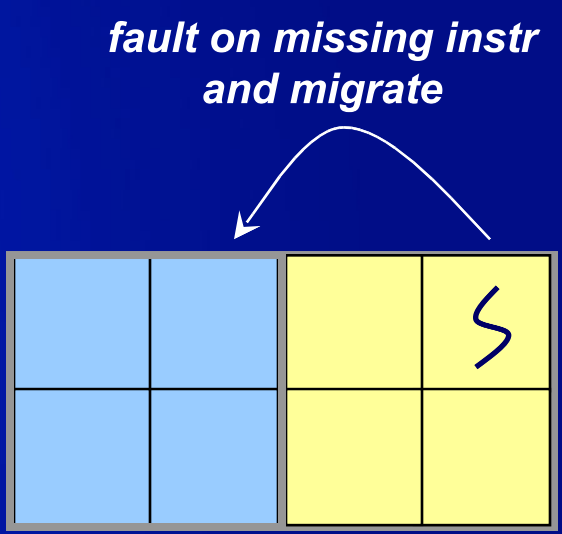
- Virtual-ISA model
  - HW hides asymmetry and exposes a common virtual ISA
  - Easy programming, no OS/app changes, but complex HW
- Coprocessor model (e.g., Cell, CUDA)
  - Subset of cores exposed as coprocessors or peripherals
  - Pros: minimum OS changes (handled by drivers)
  - Cons: OS runs on main cores, master/slave programming, need drivers/libraries, performance and maintenance overhead
- CPU model
  - All cores exposed as CPUs and managed by OS
  - Cons: non-trivial OS changes (core OS changes)
  - Pros:
    - Resource management is the traditional job of OS
    - Transparent support of apps
    - Even better w/ help from apps, compilers, and libraries

Outline

- Asymmetric architecture design space
  - Types of asymmetry
    - Focus on instruction-based asymmetry
  - Hardware-software interface
    - Focus on CPU model
- Case study of OS support
  - OS design and implementation
    - fault-and-migrate
  - Evaluation
- Conclusion

Fault-and-Migrate

- Core raises exception when running unsupported instruction
  - #UD fault on IA (invalid opcode)
- OS migrates thread to core that supports the instruction
  - Change Linux thread affinity mask to only cores with the support
  - Leverage Linux existing migration mechanism
  - Different policies could control when thread can migrate back



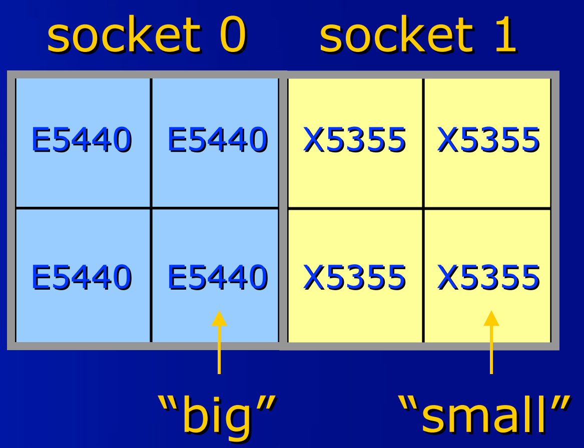
Fault-and-migrate (cont)

Policies to control “migrate back”

- Always
  - Restore affinity mask after one quantum on new core
  - When and where to migrate controlled by existing OS policy
- Counter-based
  - Hardware counts “faulty” instructions on new core
  - Restore affinity mask if zero “faulty” instructions for a quantum on new core

Research Platform

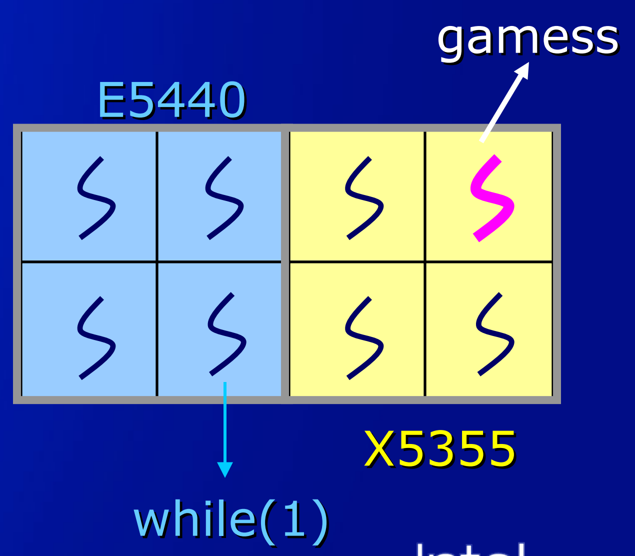
- Dual-socket Intel® Xeon® system
  - E5440 + X5355 quad-core
  - 2.83 vs. 2.66 GHz
  - 6 vs. 4 MB L2
  - SSE4.1 vs. none



- Modified BIOS to allow OS boot
  - Both Windows and Linux boot out-of-box
  - SSE4.1 apps fail half of time
- Implemented fault-and-migrate in Linux 2.6.20
- Changed global variables to per-CPU to support frequency asymmetry (cpu_khz, cyc2ns_scale)

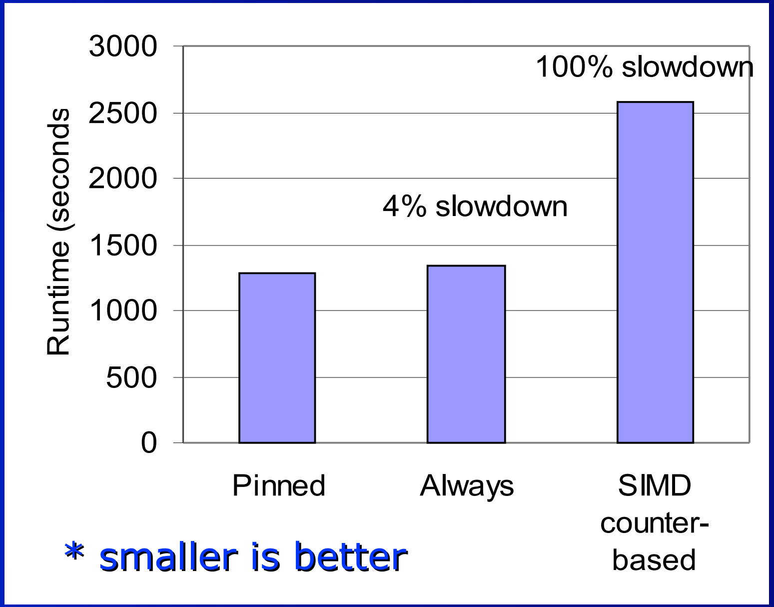
Functional Evaluation

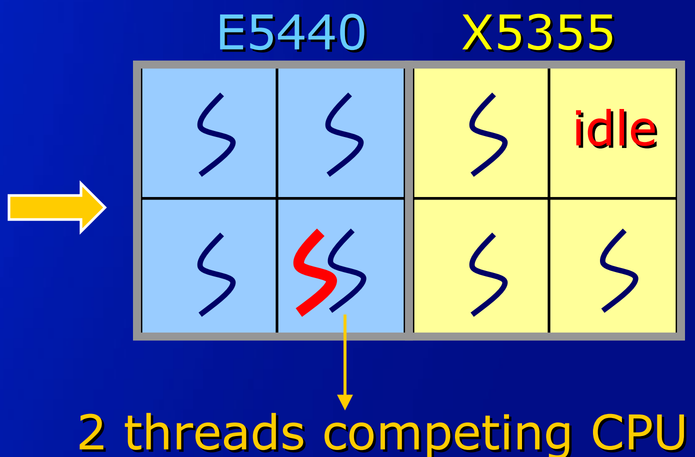
- All cores at 2.66Ghz to isolate instruction asymmetry
  - L2 cache still different (6 MB vs. 4 MB)
- 8-thread workload
  - 7 "while(1)" loops, no SSE4.1, each pinned to a core, but leaving 1 X5355 (small) idle
  - Then, run gamess (SPEC CPU2006, compiled w/ SSE4.1)
- Expected behavior
  - Gamess starts on X5355, faults on SSE4.1, migrates back and forth between X5355 and E5440



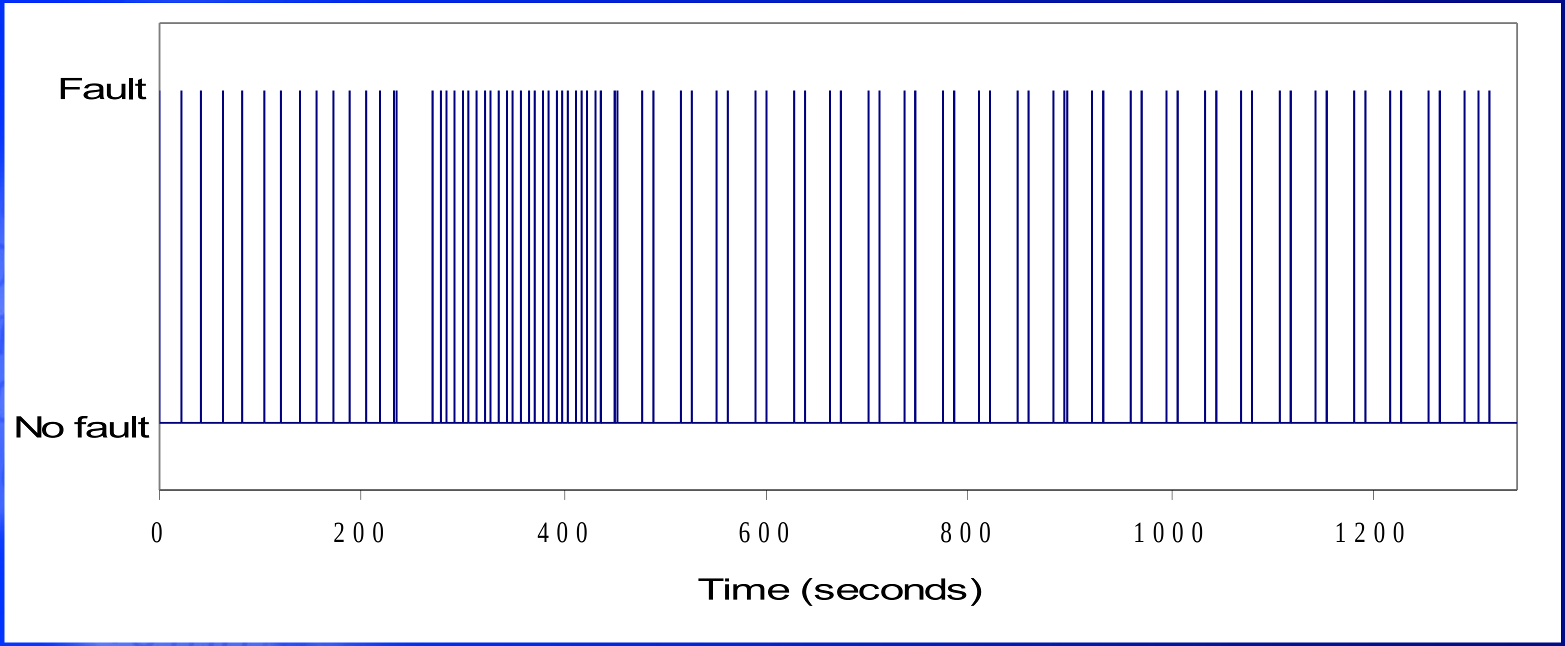
Performance Evaluation

- Pinned: gamess pinned to a E5440 (big) core
- Always: restore affinity mask after one quantum on new core
  - 4% includes cost of smaller L2 and 2 threads competing on new core
  - Actual cost of FaM can be smaller
- SIMD counter-based: restore mask after one quantum of zero SIMD
  - SIMD event counts more than SSE4.1
  - Gamess does SSE2 all the time, so never migrates back
  - SSE4.1 counter would help





Gamess SSE4.1 Faults



- One vertical line means one fault at that time
- Totally only 120 faults in 1341.51 seconds
- Most faults sparsely spread, suggesting benefits of "migrate back"

Migration Overhead

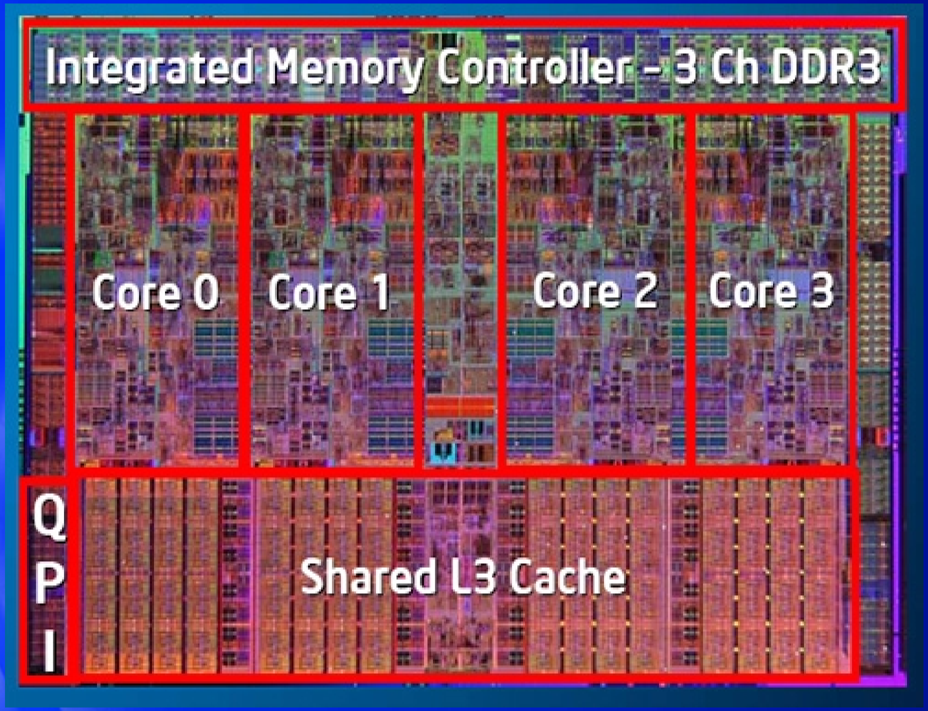
- Overhead = migration cost + cold cache penalty
  - 1st component negligible: 5.5 μs (avg) from early P4 study
  - Focus on 2nd component
- Measuring cold cache penalty
  - touch_cache() walks memory in a hard-to-predict way
  - Measure max penalty of various working set sizes
- Same-socket < 5.2 μs
  - Cores share L2
- Cross-socket < 1.7ms

pin self to core 0
touch_cache() // warm up cache

// no migration	// migration
start = current time	migrate to core 1
touch_cache()	start = current time
stop = current time	touch_cache()
t0 = stop - start	stop = current time
	t1 = stop - start

cold cache penalty = t1 – t0

Migration Overhead (cont)



4-core
8-thread
Nehalem
processor

- Future CMPs likely have shared caches
- Expect low migration overhead

Conclusion

- OS *can* support asymmetric cores
  - Fault-and-migrate enables application transparency
  - Demonstrated with real hardware and OS
- Next step is to do it better
  - HW support to improve OS management
    - Asymmetry discovery
    - Missing instruction notification
    - Missing instruction counting
  - More sophiscated OS policies
    - Intelligent migration policies