Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance

Michael D. Powell, Arijit Biswas, Shantanu Gupta, Shubhendu S. Mukherjee

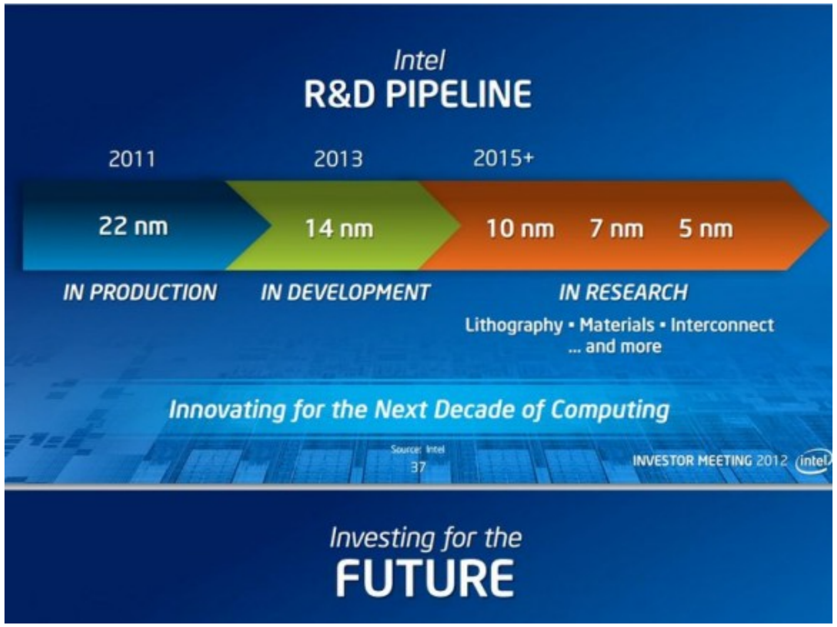
Meghan Cowan Yilei Xu

Outline

- Motivation
- Existing Solutions
- High Level Overview
- Architectural Salvaging
- Hybrid Approach
- Results

Motivation

Transistor size ↓ Die Density ↑



Intel
R&D PIPELINE

2011	2013	2015+		
22 nm	14 nm	10 nm	7 nm	5 nm
IN PRODUCTION	IN DEVELOPMENT	IN RESEARCH		

Lithography • Materials • Interconnect
... and more

Innovating for the Next Decade of Computing

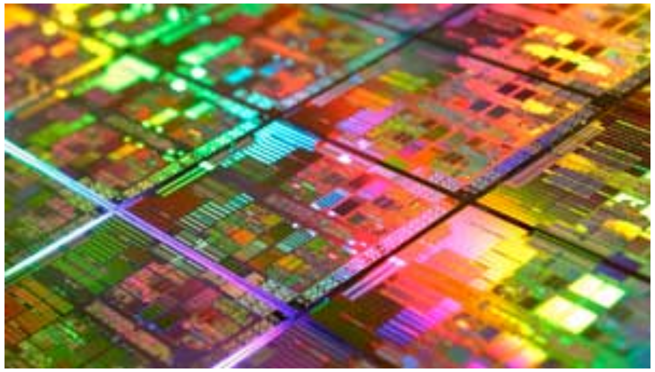
Source: Intel
37

INVESTOR MEETING 2012 intel

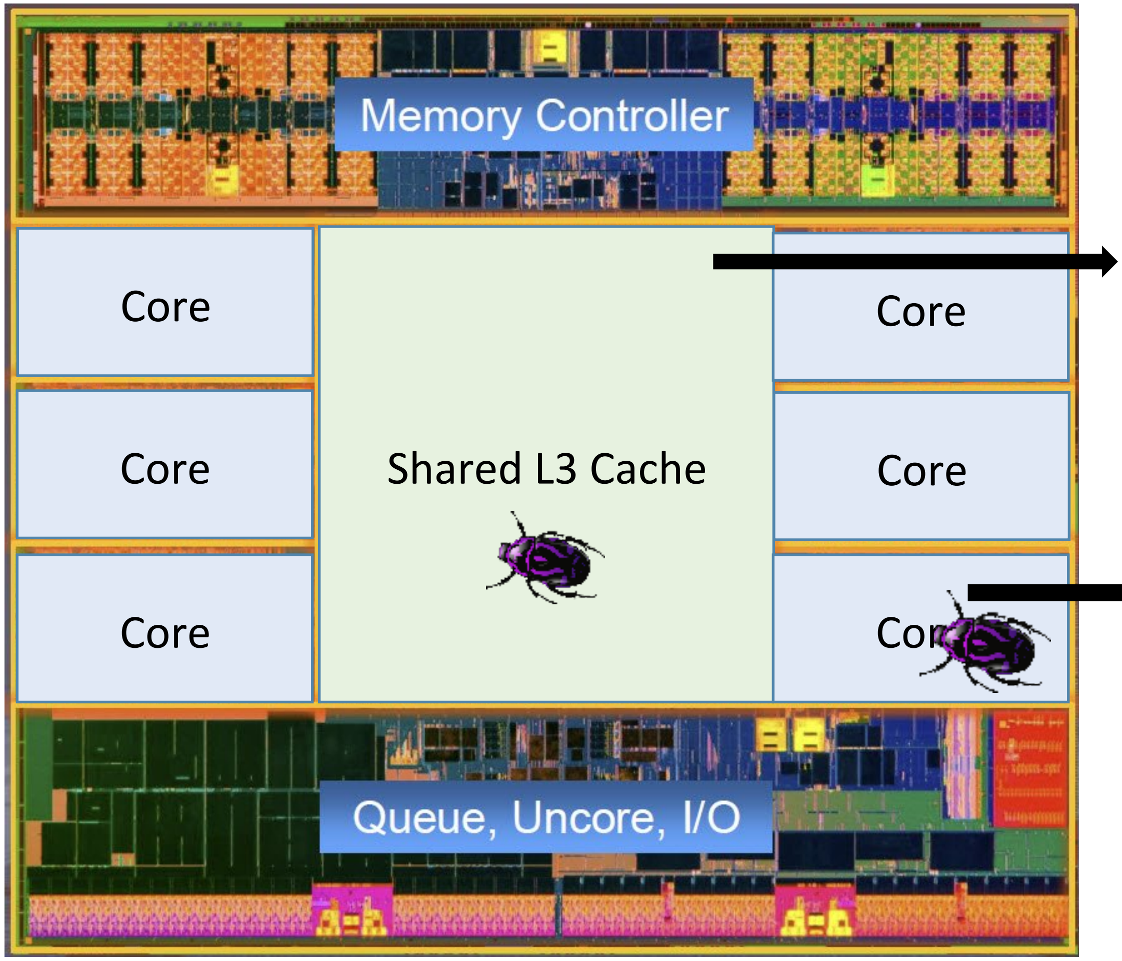
Investing for the
FUTURE

More susceptible to Hard Faults

- Frequently happen
- Difficult to handle



Motivation



Intel Haswell-E

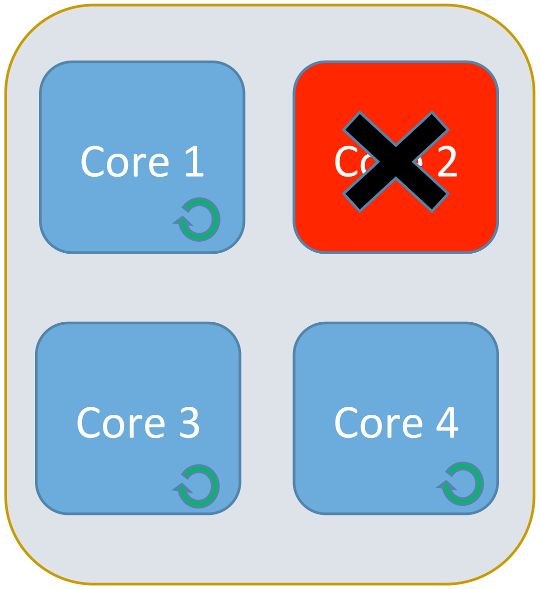
Array sparing, line sparing, etc.

Ineffective current solutions

- High area cost
- High performance cost

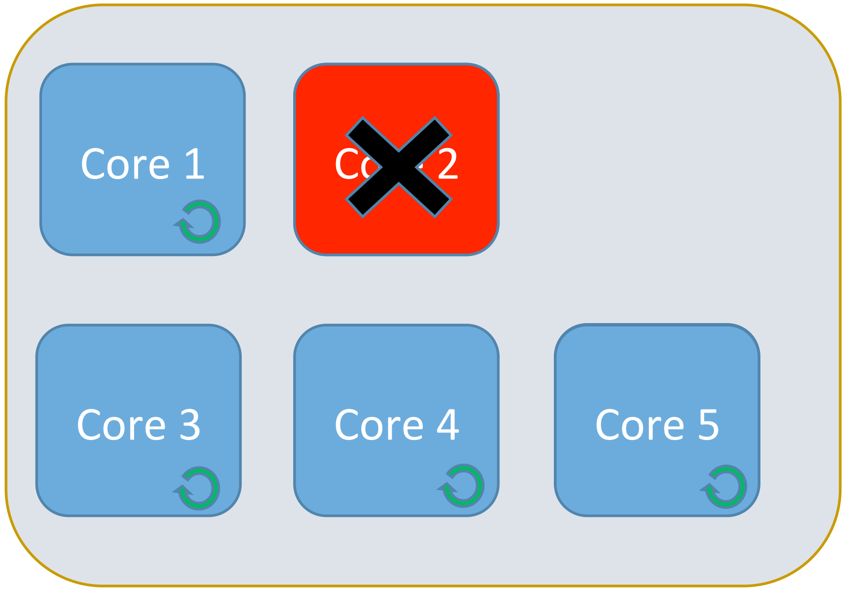
Existing Solutions

Core Disabling



Performance

Core Sparing

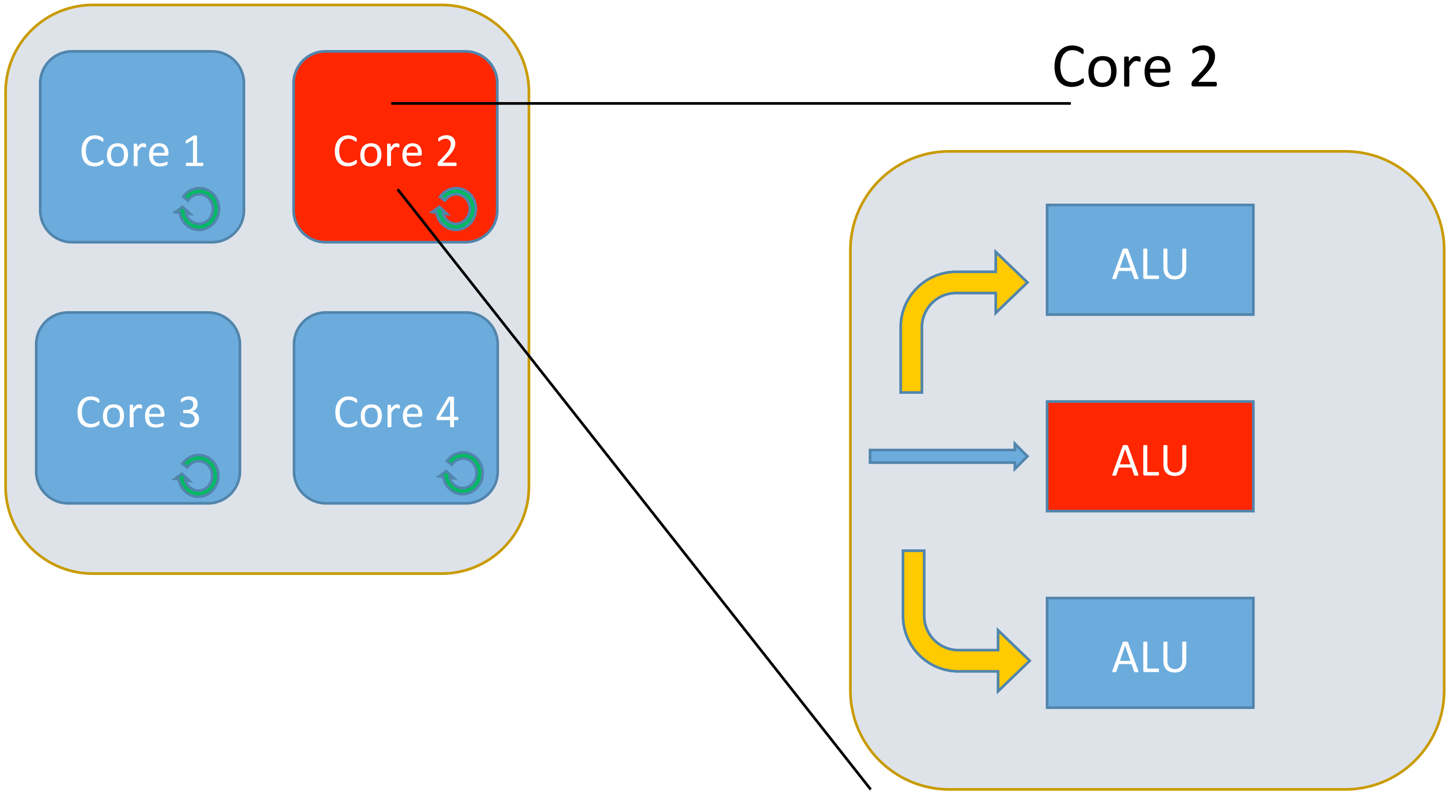


Area

Existing Solutions

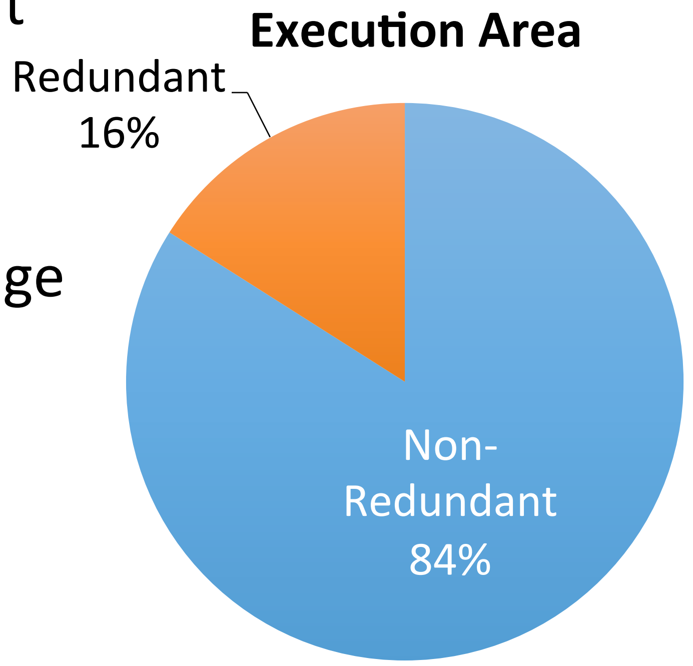
Core Salvaging

Microarchitectural Salvaging
-- Exploit redundancy within a core



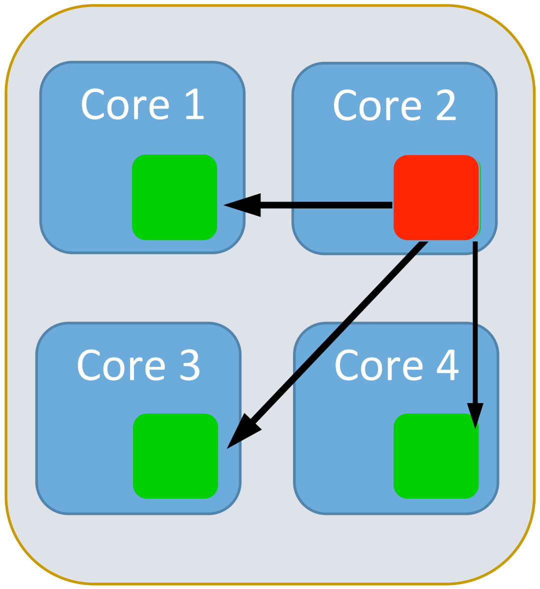
Microarchitectural Salvaging Limitations

- **Low Coverage**
  - CPUs are mostly combinational logic
  - Many logic structures not redundant
- **Complex**
  - Add artificial redundancy for coverage
  - Requires unique salvaging methods



Key Idea

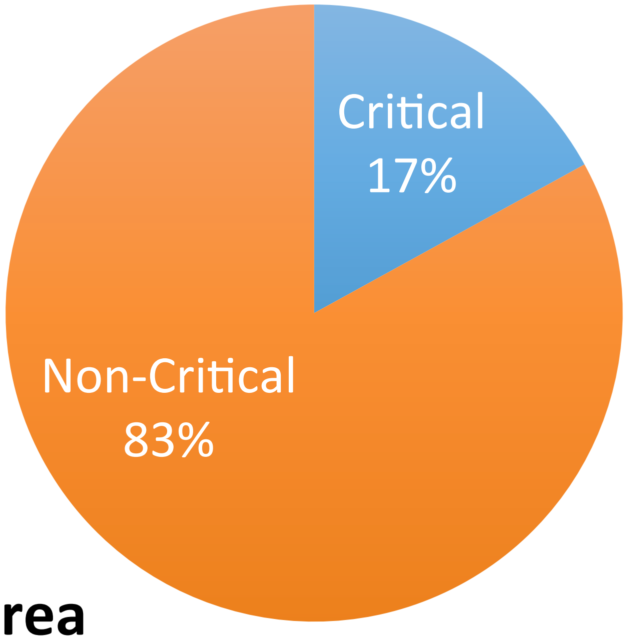
- CPU die can be ISA compliant while individual cores aren't
- **Cross-core redundancy**: Other cores have the same resources



Use **Architectural Salvaging** to borrow another core's resources

Potential

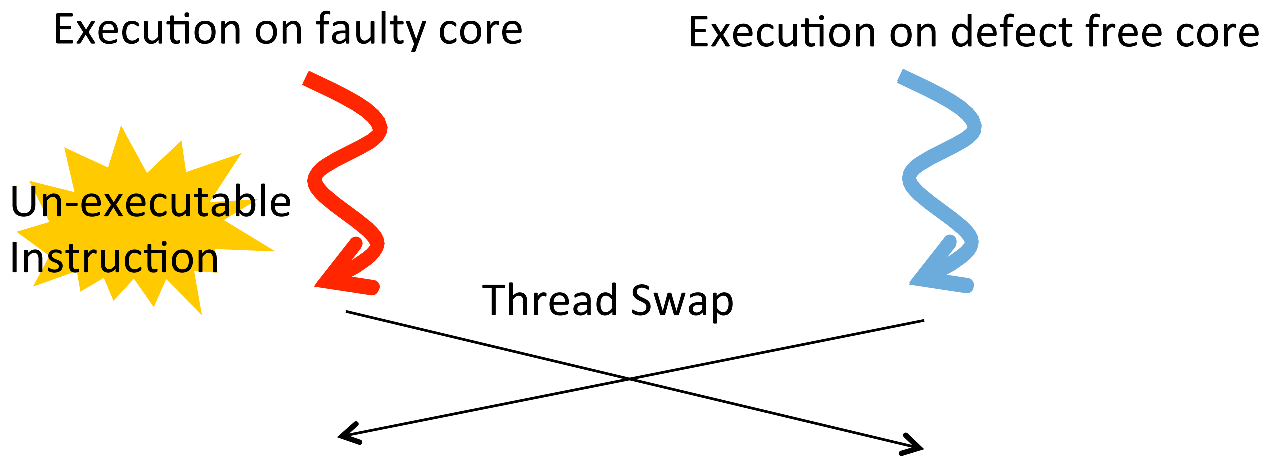
- Efficiently cover lots of area without replication
- High percentage of area is non-critical structures
  - Not needed for basic functionality
- Example Complex decoder
  - Infrequently used
  - Large and not replicated



Execution Area

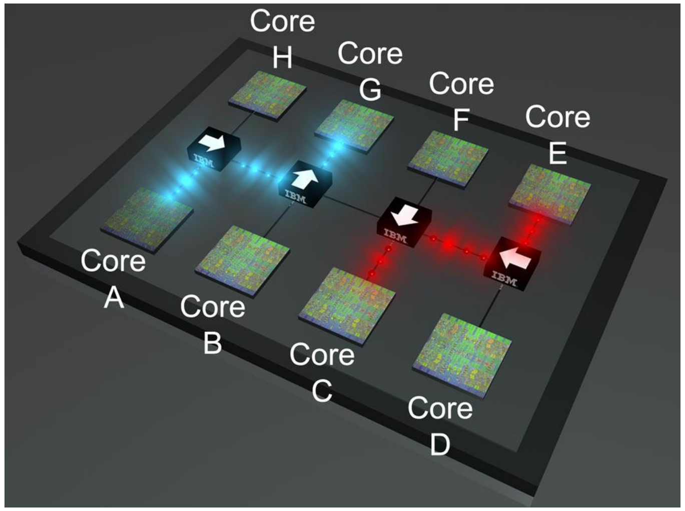
Architectural Salvaging

- Don’t require individual cores be fully functional
  - At least support critical instructions (load, add, etc.)
- Track defects
  - Migrate thread to another core when it can’t execute



Implementation

- Minimal Core Changes
  - Detect defective instructions
  - Use existing thread migration capability



- OS Transparency
  - Make the APIC ID programmable
  - Migrate the APIC ID with the thread

Optimizations

Temporarily fallback to Core Disabling

- Triggered when migrating too frequently
- Thread migration has a performance cost
  - ~100s cycles + pipeline flush

Small Array Problem

- Examples: decode queues, RS, branch predictor
- Most of the area is dedicated to support logic

❌ Array Salvaging

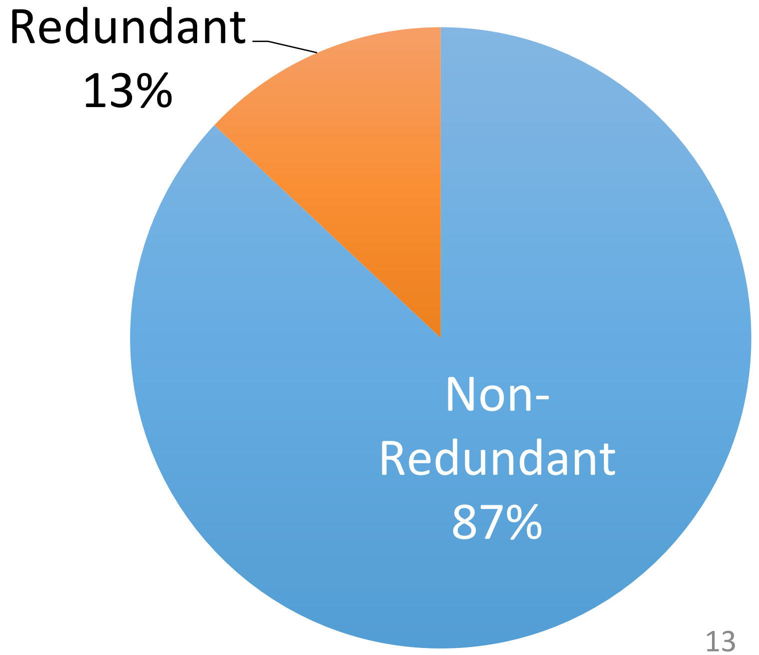
- No natural redundancy

❌ No backups

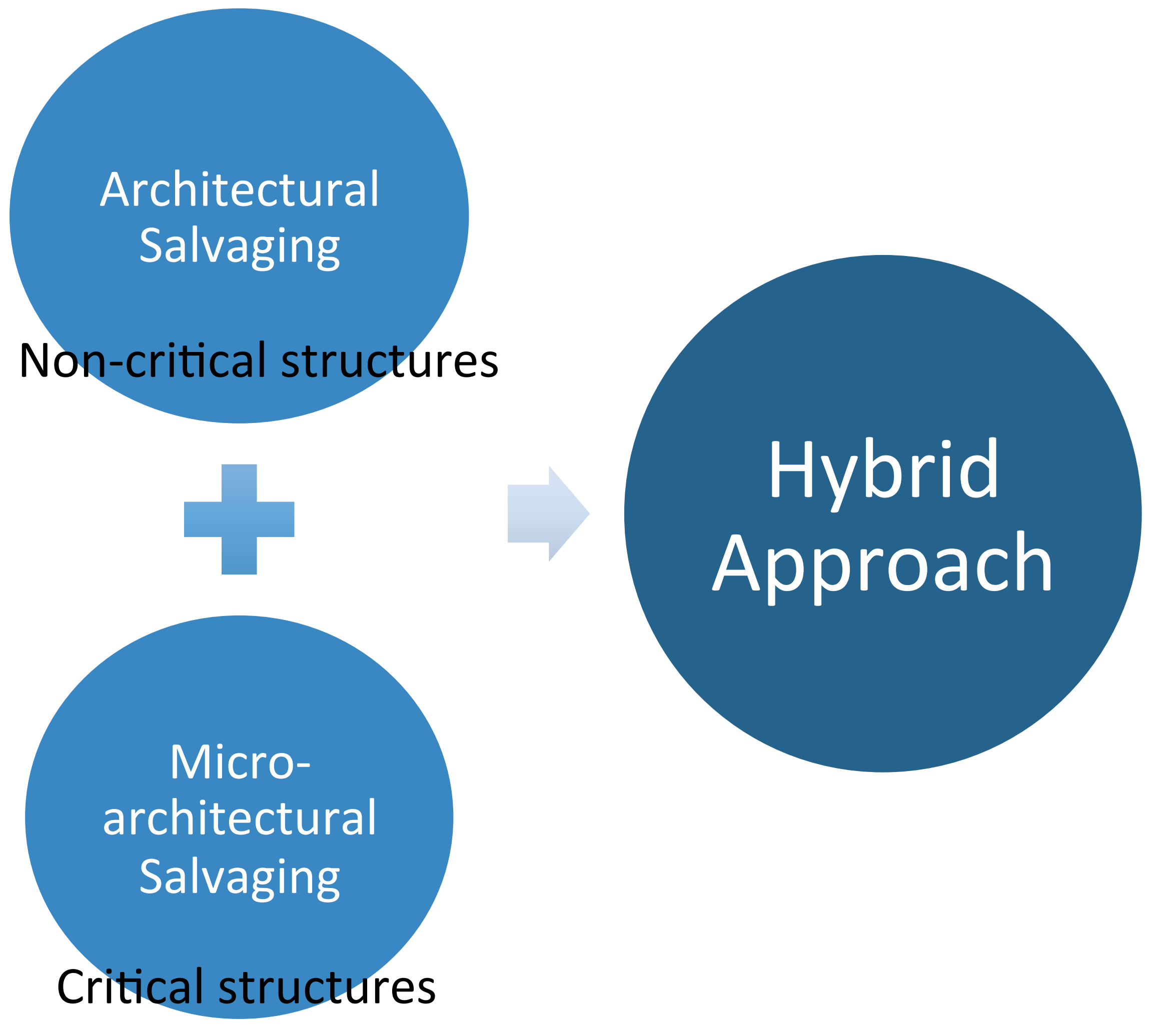
- Often critical structures

❌ Architectural Salvaging

Decode Queue Area

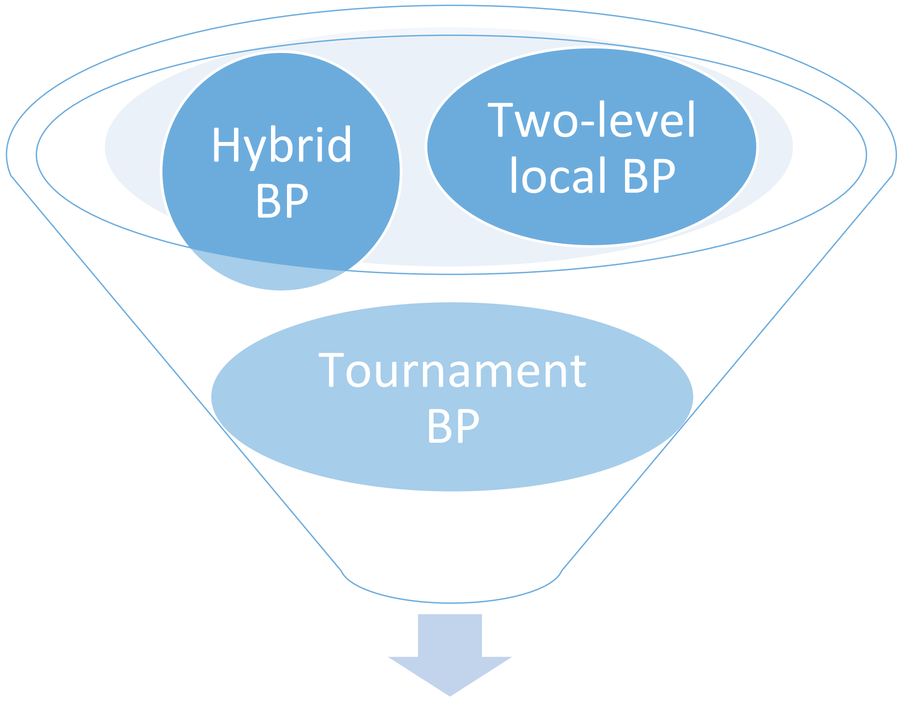


Solution - Hybrid Approach

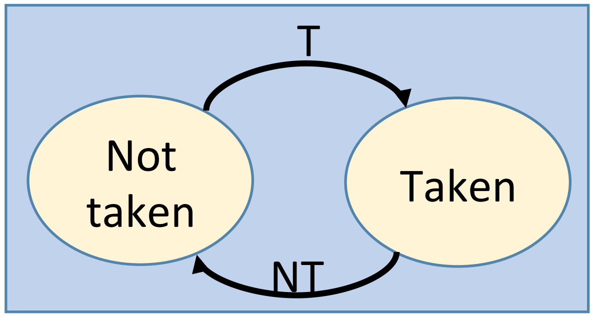


Hybrid Approach

- Add secondary structure
  - Small compared to primary structure
  - Minimal functionality



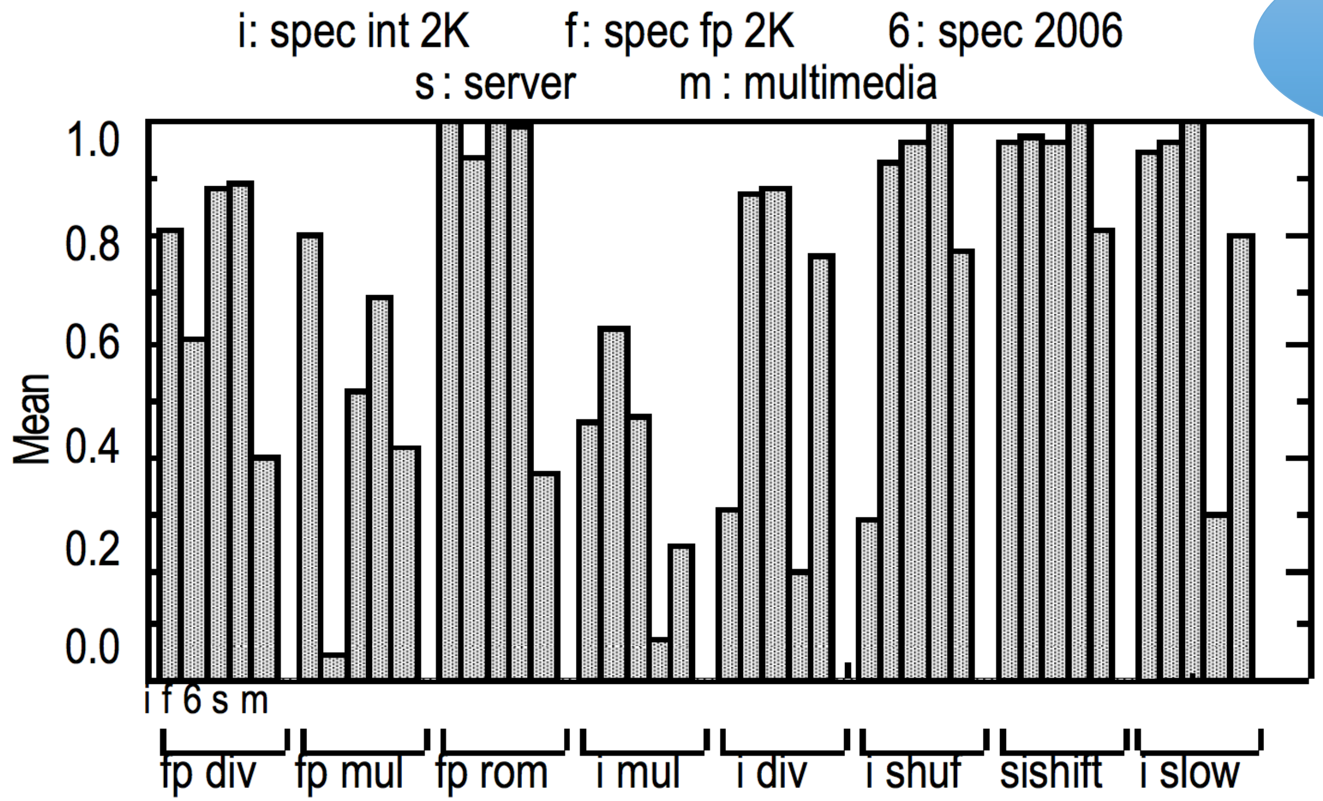
Complex Branch Predictors



Simple bimodal Branch Predictor Backup*

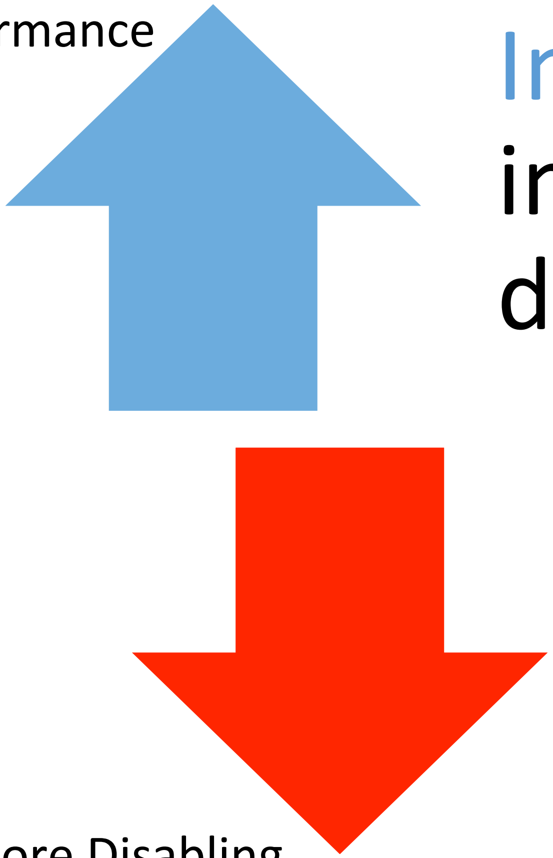
Infrequent Instruction Classes

Fraction of non-overlapping 100K instruction windows that **do not** contain the instruction class



Good candidates

Expectations

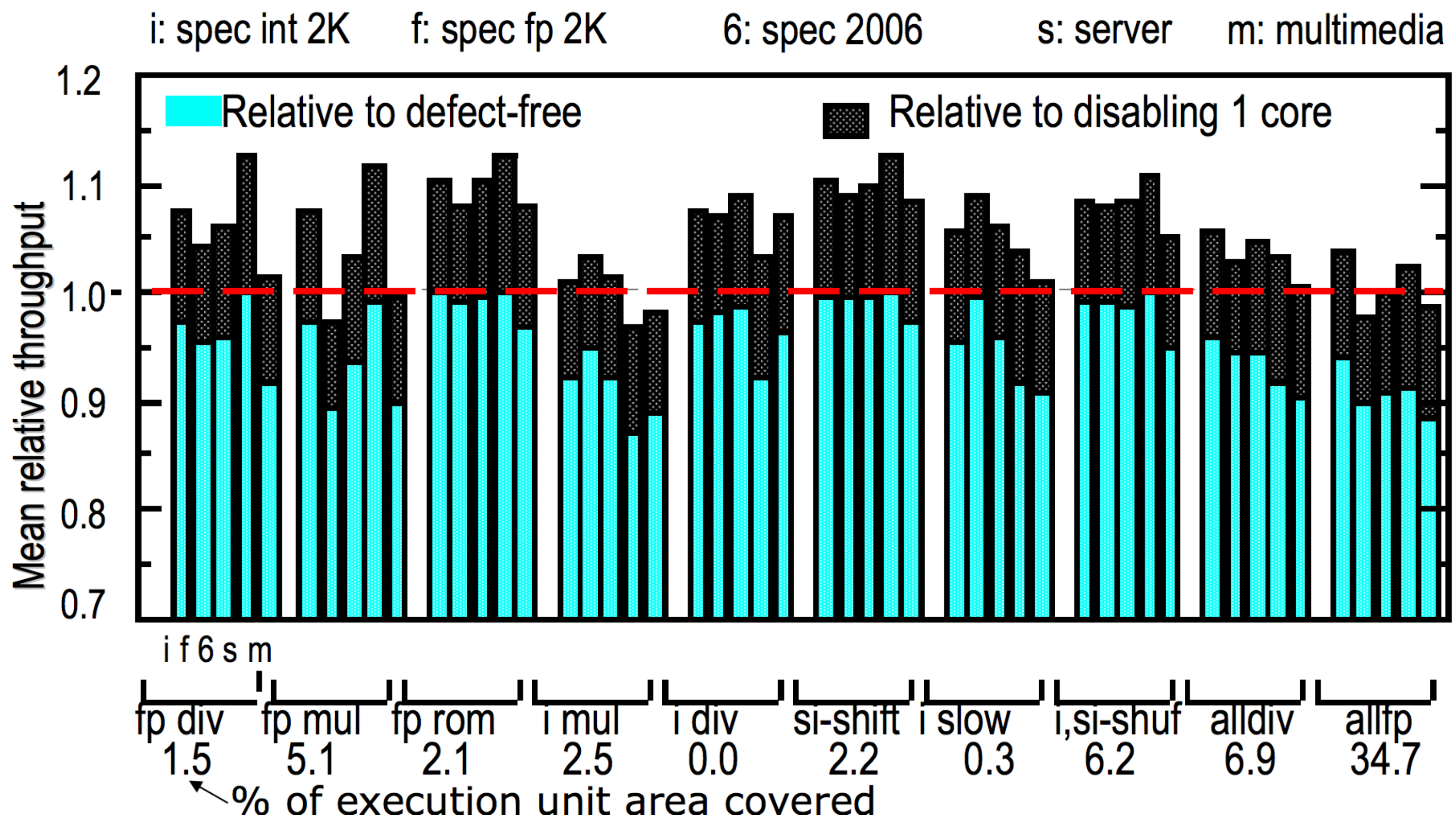


Infrequent
instruction
defected

Frequent
instruction
defected

Throughput – 8 Core Die

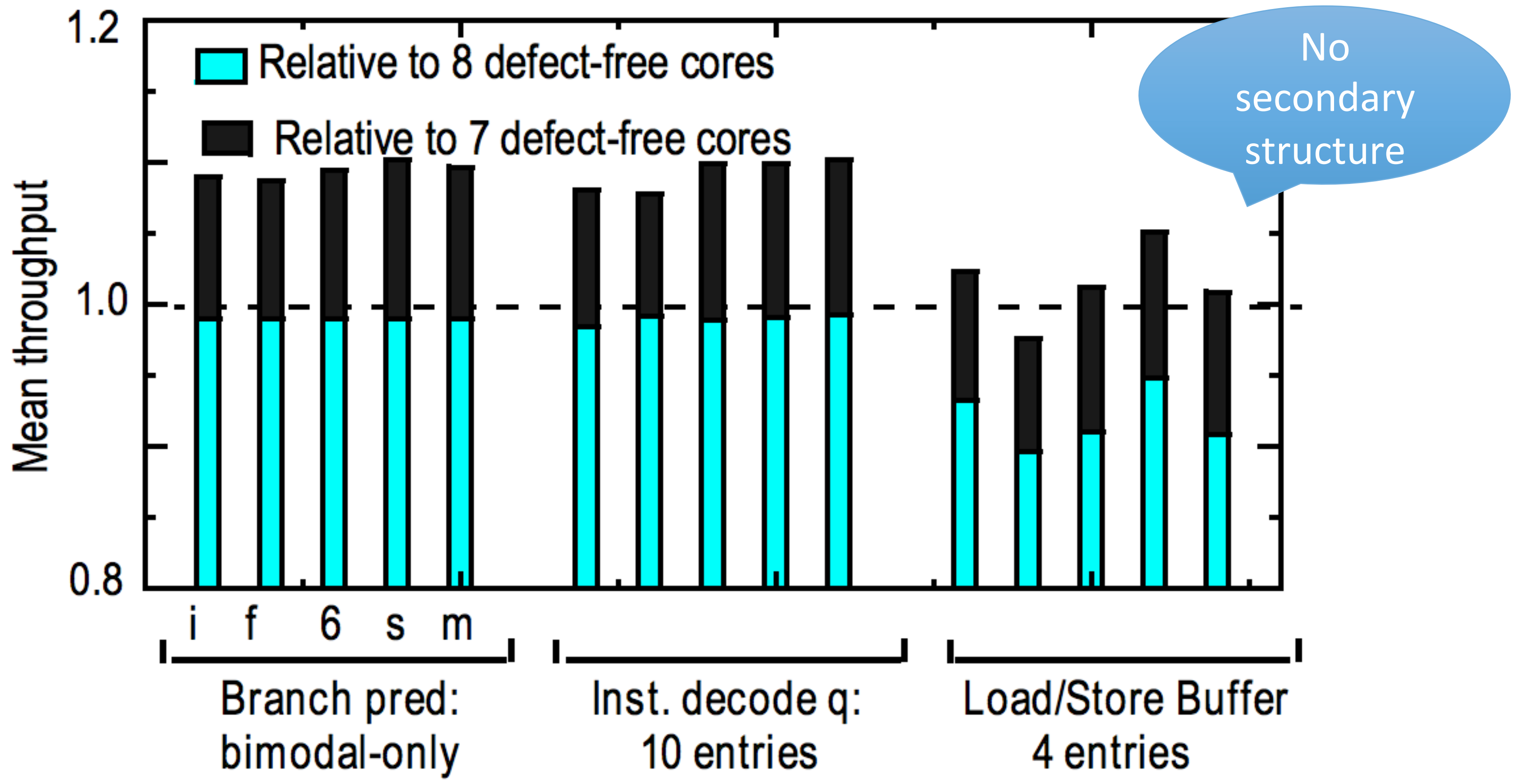
On average 5-7% better throughput than core disabling



]

Hybrid Approach Throughput

Throughput using the secondary structure > Core disabling



Conclusion

- Hard faults in the CPU are challenging to tolerate
- Microarchitectural salvaging has limitations:
    - Complex
    - Low coverage
- Architectural salvaging offers:
    - Higher coverage for non-critical structures
    - Minimal architecture changes by thread migration
- Hybrid Approach:
    - Achieve coverage for critical structures

Questions?

Discussion Points

1. Can architectural core salvaging work with multiple defective cores?

2. Are the results convincing that core salvaging is worth the effort?