Cooperative Redundant Threads (CRT)

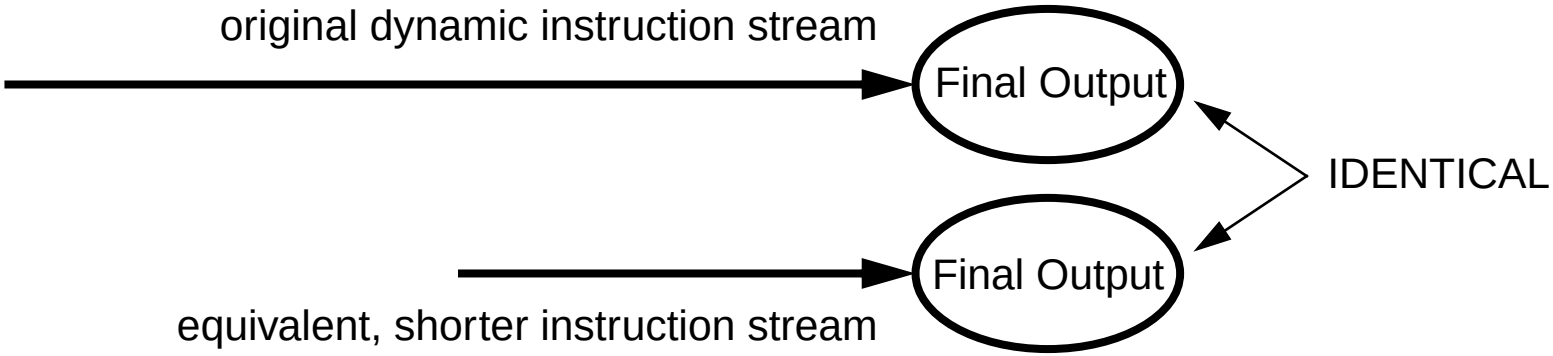
Eric Rotenberg

Karthik Sundaramoorthy, Zach Purser

Dept. of Electrical and Computer Engineering
North Carolina State University
http://www.tinker.ncsu.edu/ericro
ericro@ece.ncsu.edu

Many means to an end

- Program is merely a specification
  - Processor executes full dynamic instruction stream
  - Can construct shorter instruction stream with same overall effect



- Result: as little as 20% of the original dynamic program can produce the same result
  - Tech report - *Exploiting Large Ineffectual Instruction Sequences,* Rotenberg, Nov 1999

Many means to an end

- Key idea
  - Only need a small part of program to make full, correct, forward progress
  - The catch:
    - Speculative
    - Must monitor original program to determine essential component

Cooperative Redundant Threads

1. **Speculatively create a shorter version of the program**
   - Operating system creates two redundant processes
   - Monitor one of the programs for:
     - Ineffectual writes
     - Highly-predictable branches
   - With high confidence, but no certainty, future instances of ineffectual and branch-predictable computation are bypassed in the other program copy

Cooperative Redundant Threads

2. **Run the two versions on a single-chip multiprocessor (CMP) or simultaneous multithreaded processor (SMT)**
   - Names
     - Short program: *Advanced Stream*, or A-stream
     - Full program: *Redundant Stream*, or R-stream
   - A-stream speculatively runs ahead and communicates control/data outcomes to R-stream
   - R-stream consumes outcomes as *predictions* but still redundantly produces same information
     - R-stream executes more efficiently
     - R-stream verifies the speculative A-stream; if A-stream deviates, its context is recovered from R-stream

Cooperative Redundant Threads

- Two potential benefits
  1. Improved single-program performance
     - Faster than running only the original program
  2. Improved fault tolerance
     - Partial redundancy allows *detection* of transient hardware faults
     - Can also *tolerate* faults via the existing recovery mechanism

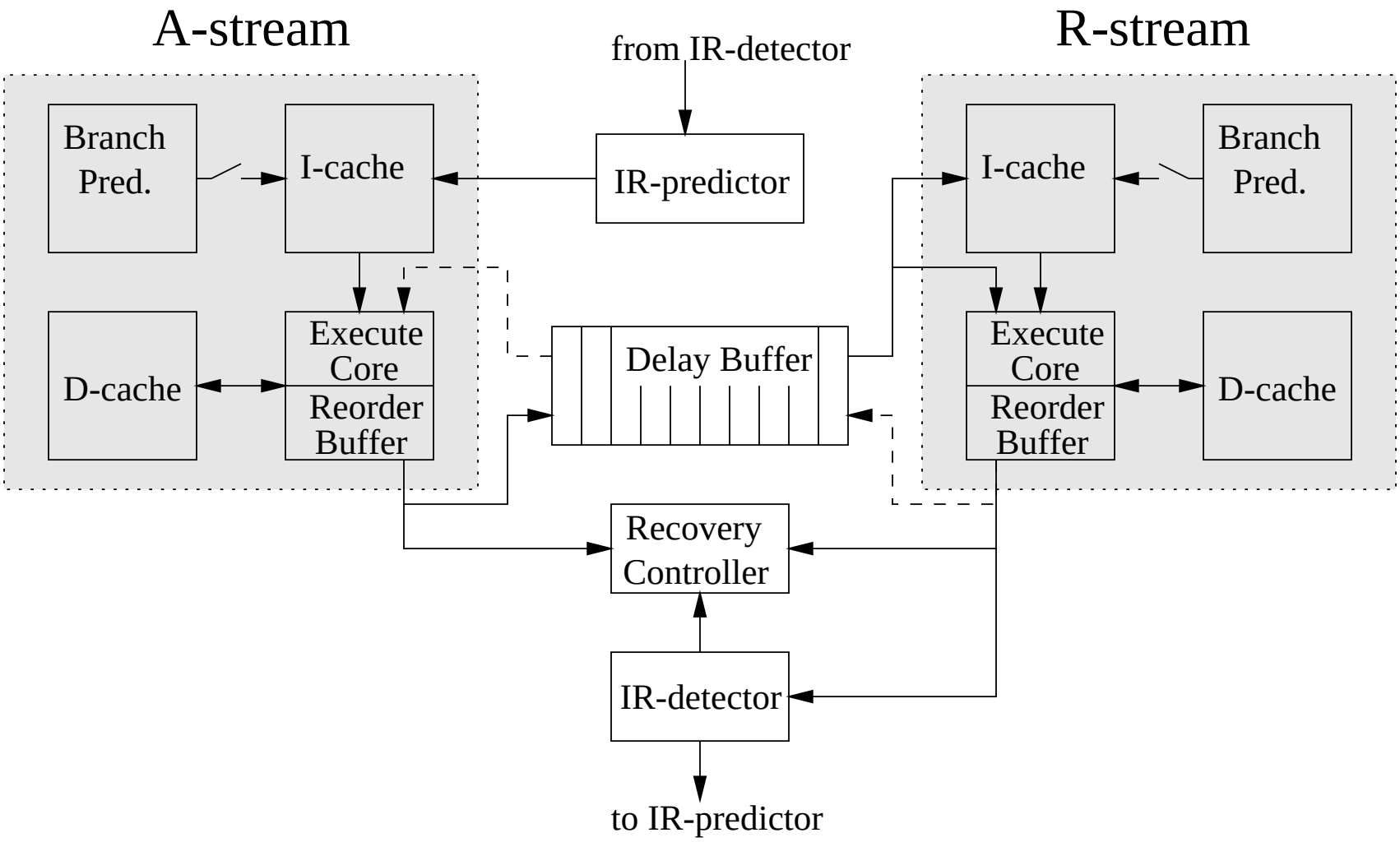
Talk Outline

✓Introduction to CRT

➔Microarchitecture description

- Understanding performance benefits
- Performance results
- Understanding fault tolerance benefits
- The bigger picture: harnessing CMP/SMT processors
- Conclusions
- Future work

Example Microarchitecture



Creating the A-stream

- A-stream creation

1. IR-predictor: *instruction-removal* predictor
   - Built on top of conventional branch predictor
   - Generates next PC in a new way
     - Next PC reflects skipping past any number of instructions that would otherwise be fetched/executed
     - Also indicates which instructions within fetch block to discard
2. IR-detector: monitor R-stream, detect candidate instructions for future removal
   - IR-detector indicates removal info to IR-predictor
   - Repeated indications cause IR-predictor to remove future instances

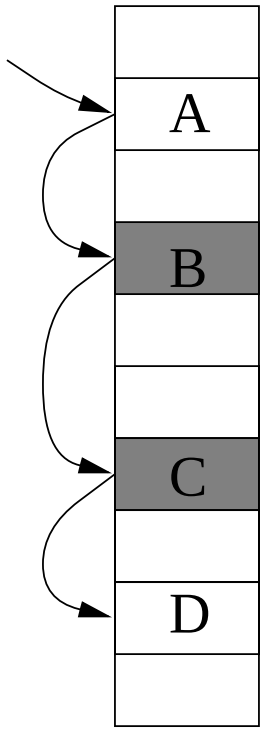
IR-predictor (Base)

- Indexed like *gshare*
- Each table entry contains info for one dynamic basic block
  - Tag
  - 2-bit counter to predict branch
  - Per-instruction resetting confidence counters
    - Updated by IR-detector
    - Counter incremented if instr. detected as removable
    - Counter reset to zero otherwise
    - Saturated counter => instruction removed from A-stream when next encountered

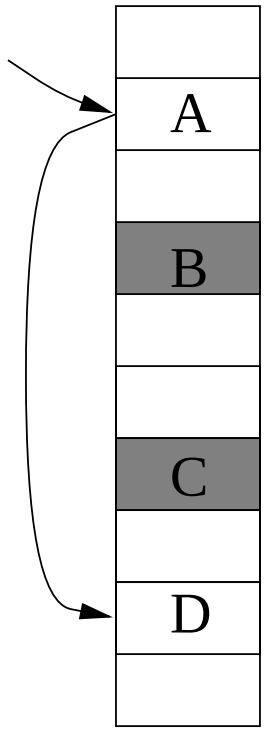
IR-predictor (Improved)

- Reducing fetch cycles in the A-stream

base IR-predictor



improved IR-predictor



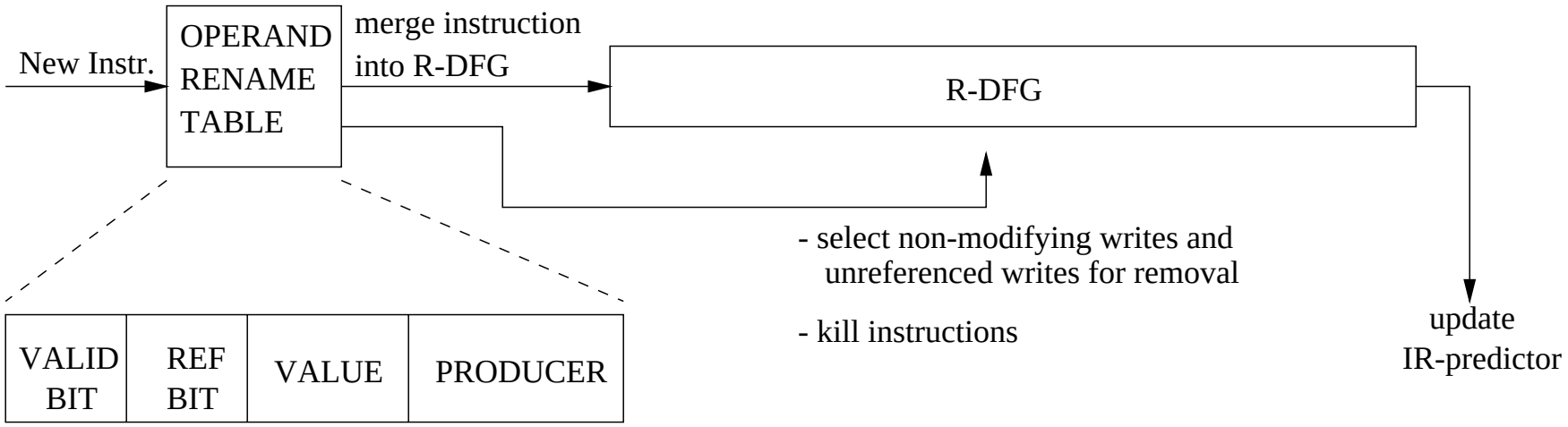
IR-predictor (Improved)

- Bypassing fetch => same effect as taken branch!
- Previous example
  - "Convert" branch ending block A to a taken branch whose target is D
- At least two possible methods
  1. Include converted target (D) and implied intervening branch outcomes (B,C) in block A's entry
  2. Include intervening branch outcomes (B,C) in block A's entry, but separate BTB to store numerous targets per static branch

IR-detector

- Monitor retired R-stream instructions for three triggering conditions
  1. Unreferenced writes
  2. Non-modifying writes
  3. Correctly-predicted branches
- Select triggering instructions as *candidates* for removal
- Also select their computation chains for removal
  - Can remove an instruction if all consumers are known (value has been killed) and all are selected for removal
  - Facilitated by reverse data flow graph (R-DFG) circuits

IR-detector



Delay Buffer

- A simple FIFO queue for communicating outcomes
  - A-stream pushes
  - R-stream pops
- Actually two buffers
  - Control flow buffer
    - Complete history of control flow as determined by A-stream
    - Instruction-removal information (for matching partial data outcomes w/ instructions in R-stream)
  - Data flow buffer
    - Partial history of data flow, for instructions executed in A-stream
    - Source/dest. register values and memory addresses

IR-mispredictions

- Instruction-removal misprediction (IR-misprediction)
  - Instructions were removed from A-stream that shouldn't have been removed
  - Undetectable by A-stream
  - IR-mispredictions corrupt A-stream context and must be resolved by the R-stream

Handling IR-mispredictions

- Three things needed
  1. Detect IR-mispredictions
     - Both R-stream and IR-detector perform checks
  2. Get ready for state recovery
     - Backup IR-predictor (branch predictor)
     - Flush delay buffer, flush ROB_A, flush ROB_R
     - PC_A=PC_R
  3. ...

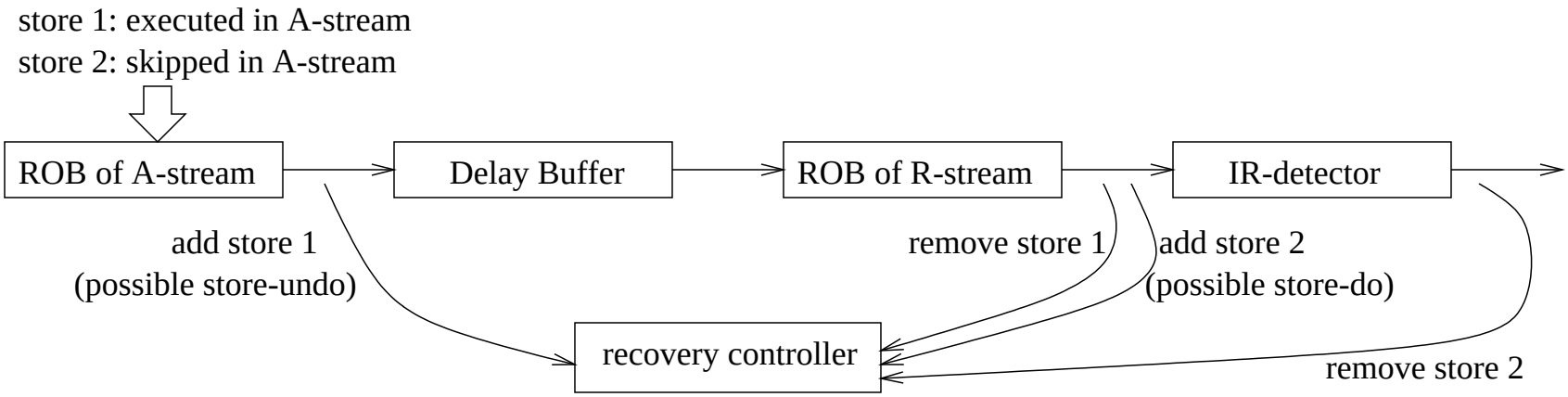
Handling IR-mispredictions (cont.)

3. Pinpoint corrupted architectural state in A-stream and recover state from R-stream
   - Entire register file copied from R-stream to A-stream
   - Recovery controller maintains list of potentially tainted memory addresses
   - Communicate restore values via Delay Buffer, reverse direction

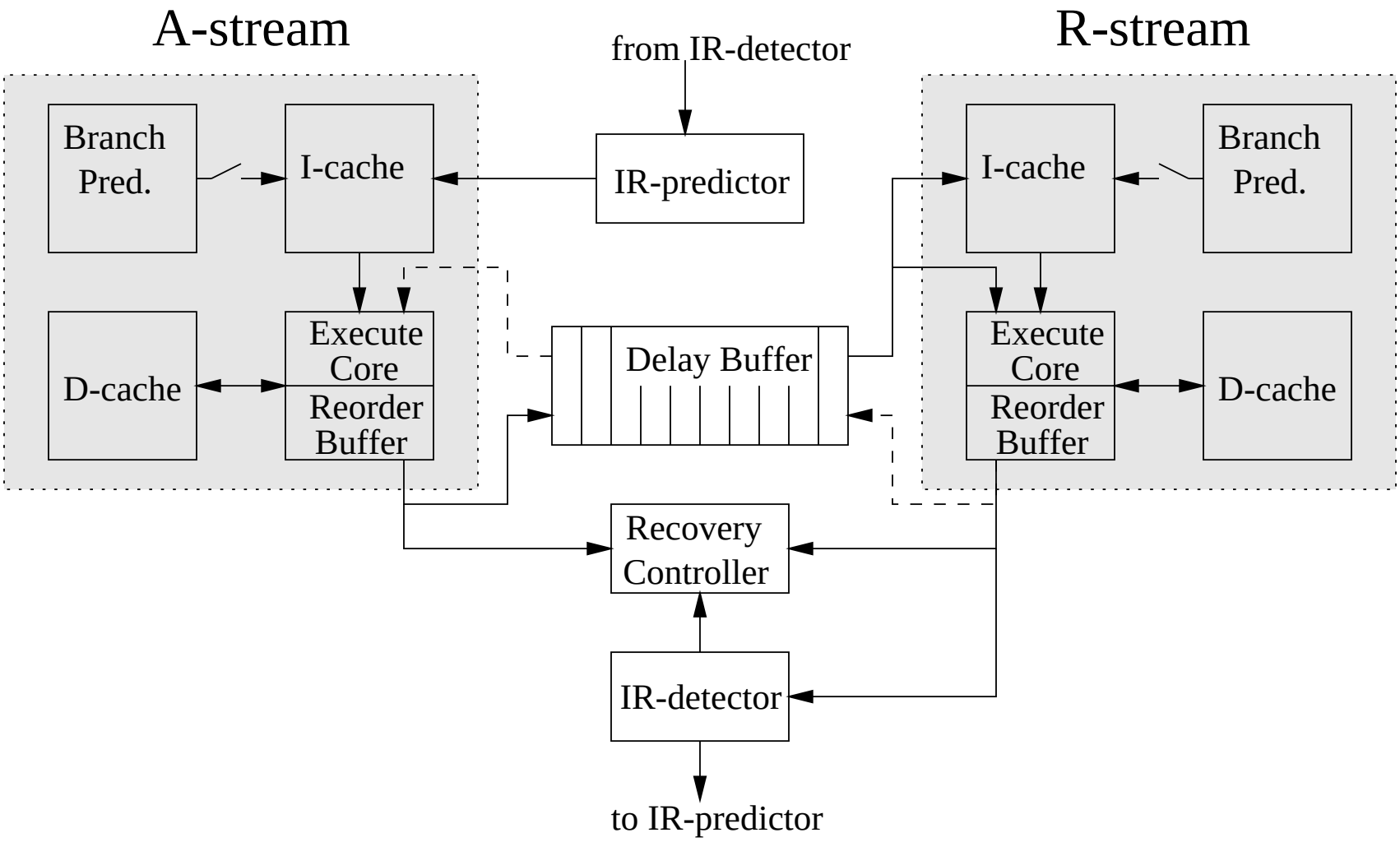
IR-misprediction Detection

- Usually surface as branch/value mispredictions in R-stream
- Some IR-mispredictions take long time to show symptoms
- IR-detector can detect a problem sooner
  - Compare *predicted* & *computed* removal information
  - Checks are redundant with R-stream checks, but recovery model requires “last line of defense”
  - “Last line of defense” bounds state in recovery controller

IR-misprediction Recovery



Review


A-stream
from IR-detector
R-stream
Branch Pred.
I-cache
IR-predictor
I-cache
Branch Pred.
D-cache
Execute Core
Reorder Buffer
Delay Buffer
Execute Core
Reorder Buffer
D-cache
Recovery Controller
IR-detector
to IR-predictor

Talk Outline

✓Introduction to CRT

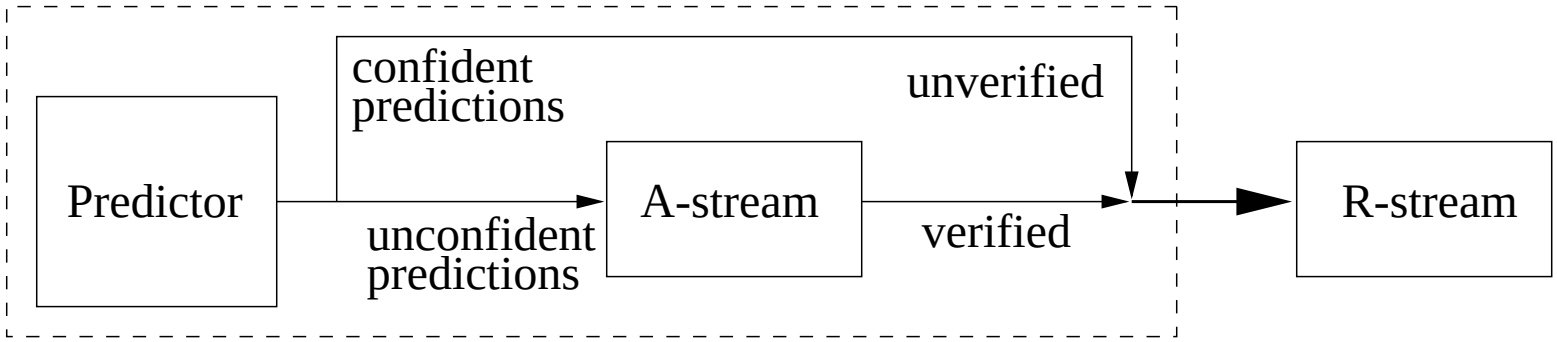
✓Microarchitecture description

➔Understanding performance benefits

- Performance results
- Understanding fault tolerance benefits
- The bigger picture: harnessing CMP/SMT processors
- Conclusions
- Future work

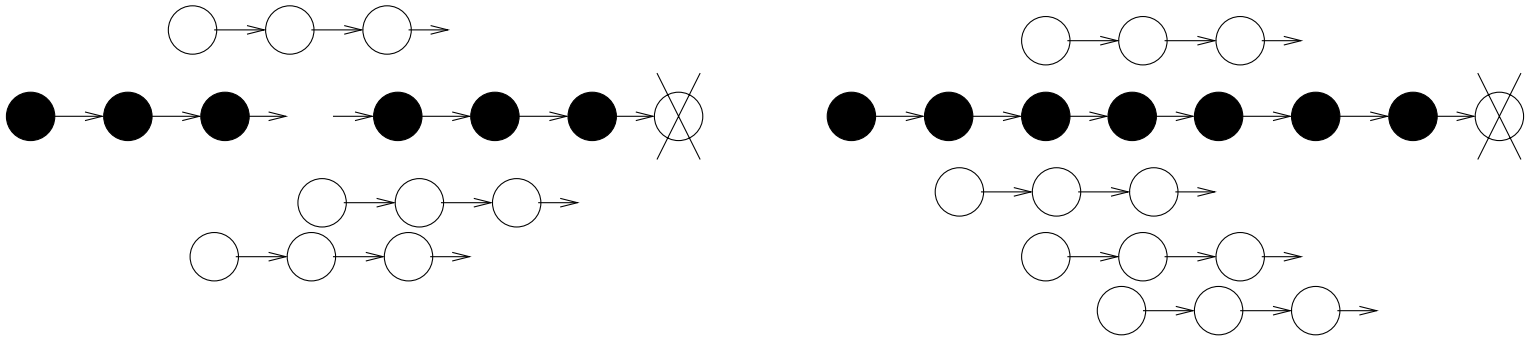
Understanding Performance

- A-stream's perspective
  - Performance is better simply because *program is shorter*
  - R-stream plays secondary role of validation
- R-stream's perspective
  - Better branch prediction
    - [**Pre-execution**: Roth&Sohi, Zilles&Sohi, Farcy et. al.]
    - A-stream is a helper thread
  - Better value prediction (program-based, not history-based)



Understanding Performance (cont.)

- What if fetch & execution bandwidth were unlimited?
  - Critical path through program = serialized dependence chains of mispredicted branches
  - A-stream cannot reduced this critical path!



Understanding Performance (cont.)

- Reasoning about instruction fetch and execution
  - More execution bandwidth devoted to R-stream (more units, bigger ROB) => A-stream less effective
  - UNLESS A-stream can also bypass instruction fetching
  - Raw instruction fetch bandwidth not as easily increased
    - Branch predictor throughput
    - Taken branches
    - (trace predictors and trace caches...)
    - Having a second program counter is great alternative *if it can run ahead*

Talk Outline

✓Introduction to CRT

✓Microarchitecture description

✓Understanding performance benefits

➔Performance results

- Understanding fault tolerance benefits
- The bigger picture: harnessing CMP/SMT processors
- Conclusions
- Future work

Experimental Method

- Detailed execution-driven simulator
  - Faithfully models entire microarchitecture
    - A-stream produces possibly bad control/data, R-stream checks A-stream and recovers, etc.
    - Simulator validation: independent functional simulator checks timing simulator (R-stream retired instr.)
  - Simplescalar ISA and compiler
    - Inherit inefficiency of MIPS ISA and gcc compiler
- SPEC95 integer benchmarks, run to completion (100M - 200M instructions)

Single Processor Configuration

single processor	
instruction cache	size/assoc/repl = 64kB/4-way/LRU
	line size = 16 instructions
	2-way interleaved
	miss penalty = 12 cycles
data cache	size/assoc/repl = 64kB/4-way/LRU
	line size = 64 bytes
	miss penalty = 14 cycles
superscalar core	reorder buffer: 64, 128, or 256 entries
	dispatch/issue/retire bandwidth: 4-/8-/16-way superscalar
	n fully-symmetric functional units (*n* = issue bandwidth)
	n loads/stores per cycle (*n* = issue bandwidth)
execution latencies	address generation = 1 cycle
	memory access = 2 cycles (hit)
	integer ALU ops = 1 cycle
	complex ops = MIPS R10000 latencies

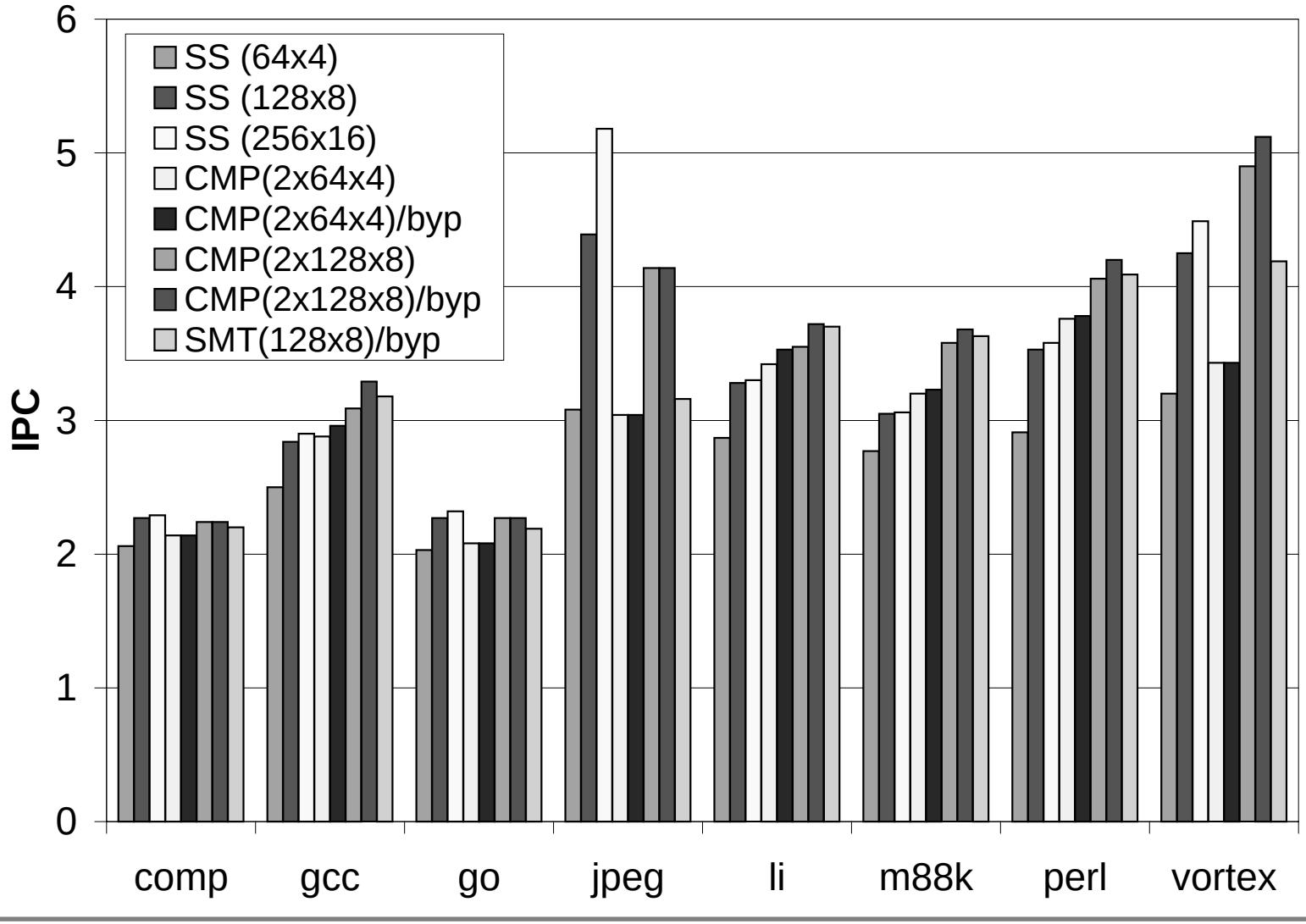
New Component Configuration

new components for cooperating threads	
IR-predictor	2^{20} entries, *gshare*-indexed (16 bits of global branch history)
	16 confidence counters per entry
	confidence threshold = 32
IR-detector	R-DFG = 256 instructions, unpartitioned
delay buffer	data flow buffer: 256 instruction entries
	control flow buffer: 4K branch predictions
recovery controller	number of outstanding store addresses = unconstrained
	recovery latency (*after* IR-misprediction detection): • 5 cycles to start up recovery pipeline • 4 register restores per cycle (64 regs performed first) • 4 memory restores per cycle (mem performed second) • ∴ minimum latency (no memory) = 21 cycles

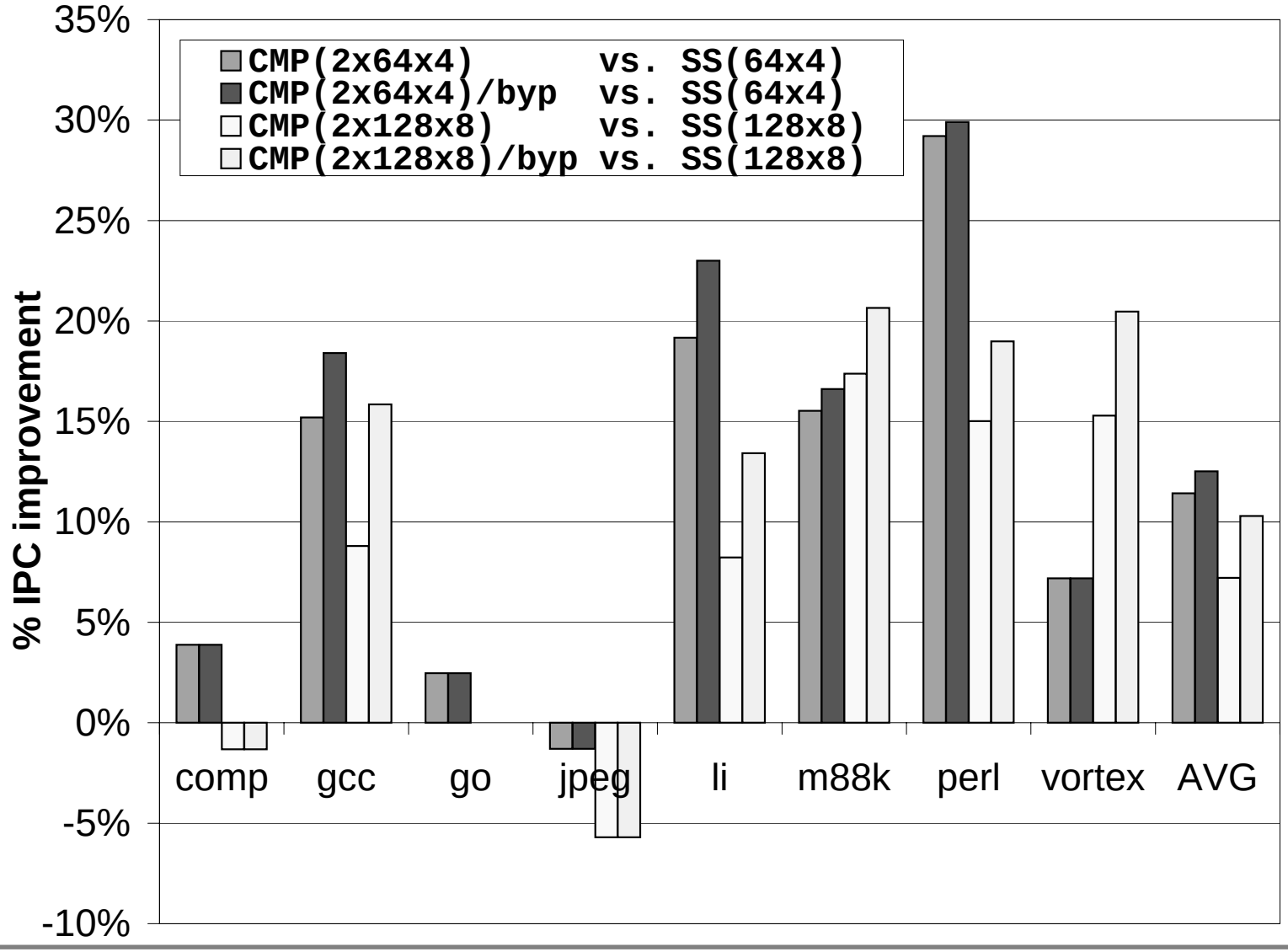
Models

- SS(64x4): single 4-way superscalar proc. with 64 ROB entries.
- SS(128x8): single 8-way superscalar proc. with 128 ROB entries.
- SS(256x16): single 16-way superscalar proc. with 256 ROB entries.
- CMP(2x64x4): CRT on a CMP composed of two SS(64x4) cores.
- CMP(2x64x4)/byp: Same as previous, but A-stream can bypass instruction fetching.
- CMP(2x128x8): CRT on a CMP composed of two SS(128x8) cores.
- CMP(2x128x8)/byp: Same as previous, but A-stream can bypass instruction fetching.
- SMT(128x8)/byp: CRT on SMT, where SMT is built on top of SS(128x8).

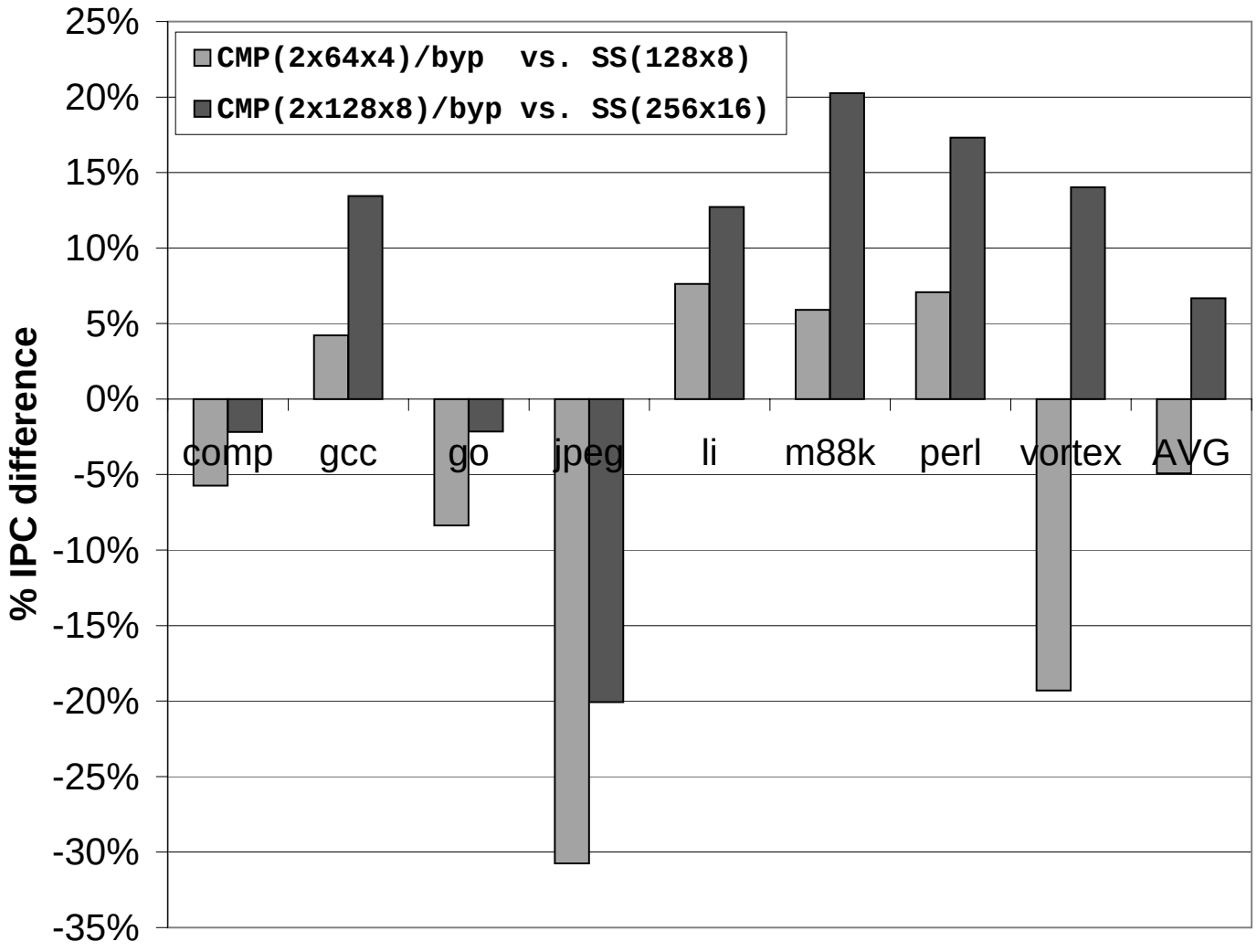
IPC Results



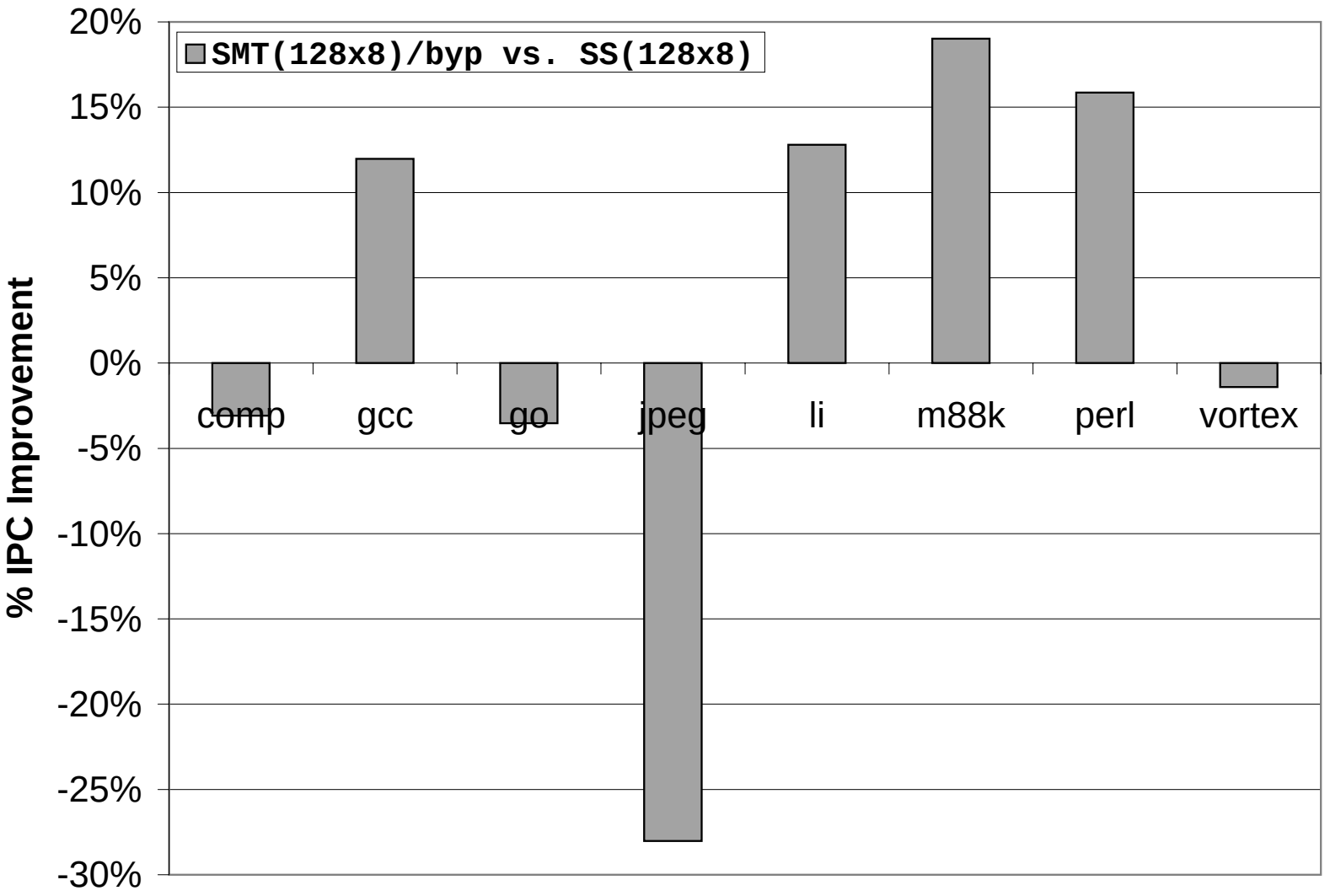
Using a Second Processor for CRT



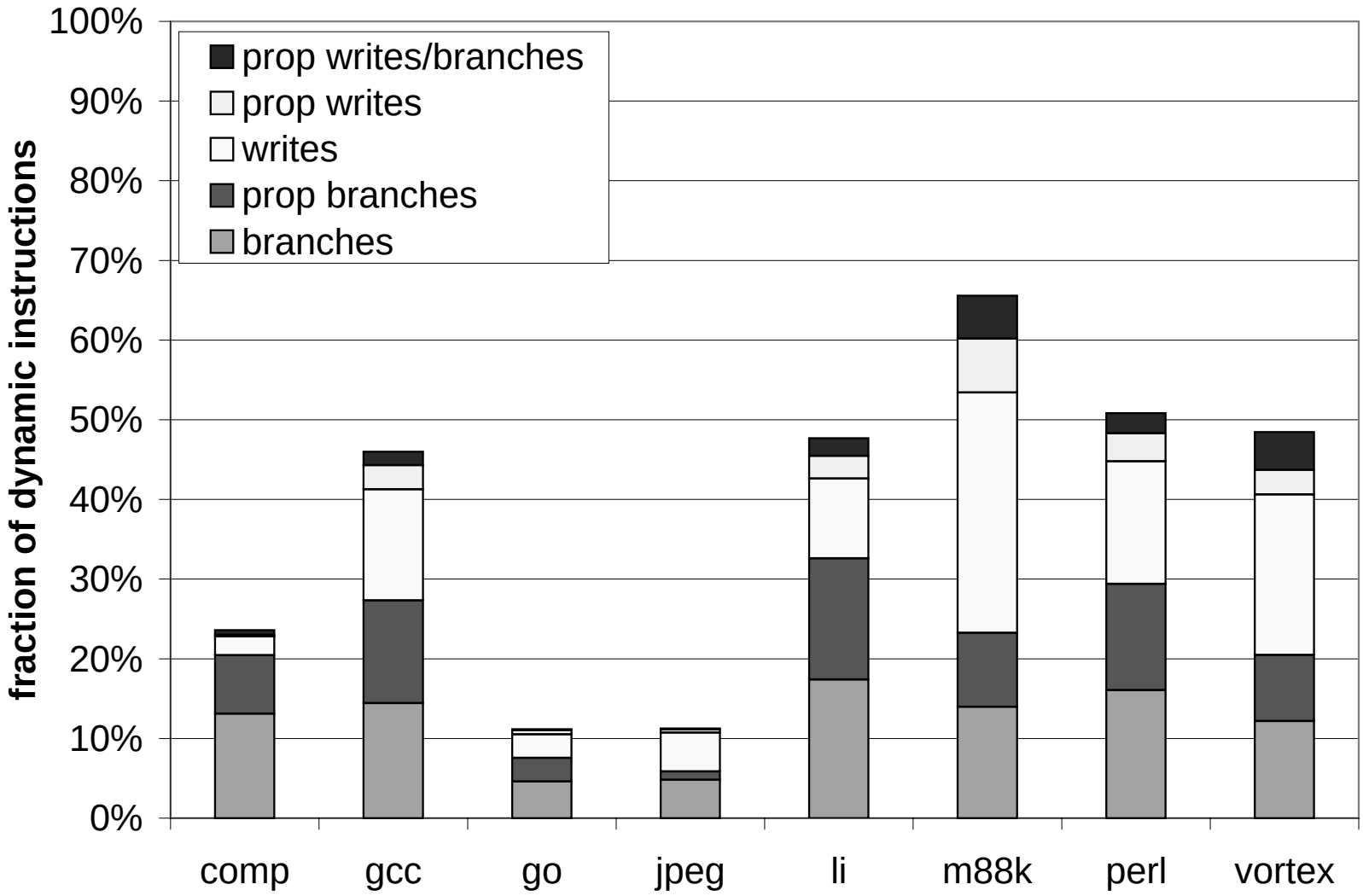
CRT on 2 small cores VS. 1 large core



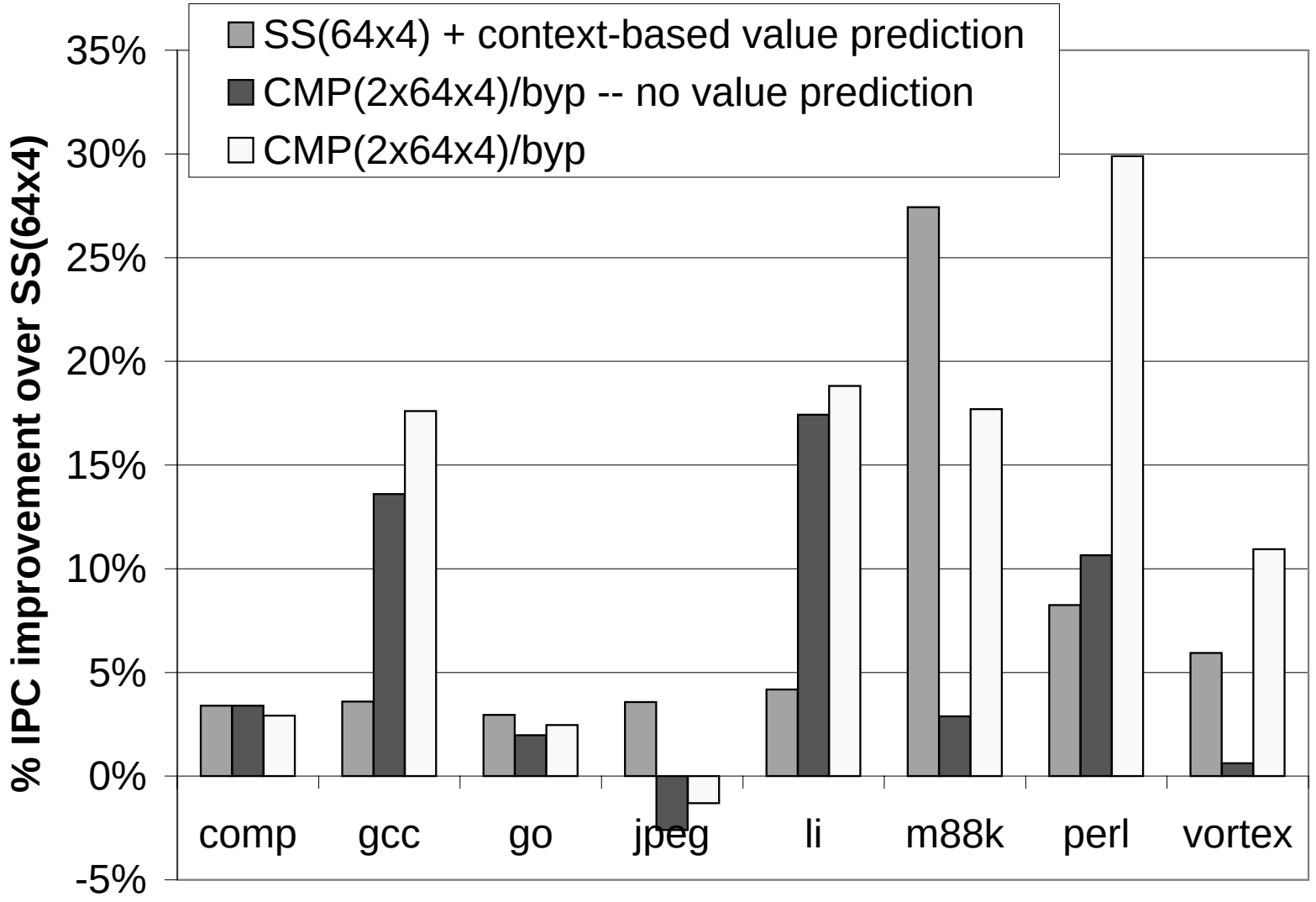
SMT Results



Instruction Removal



Prediction Benefits



Talk Outline

✓Introduction to CRT

✓Microarchitecture description

✓Understanding performance benefits

✓Performance results

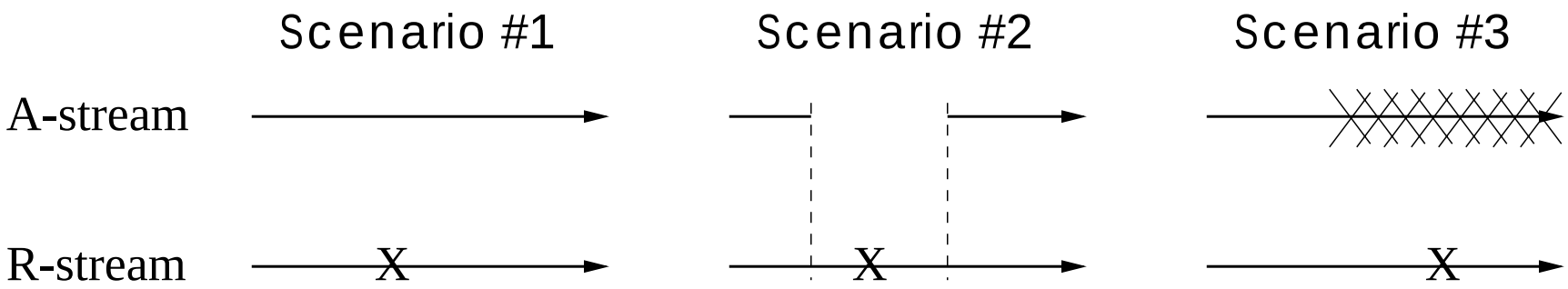
➔Understanding fault tolerance benefits

- The bigger picture: harnessing CMP/SMT processors
- Conclusions
- Future work

Fault Tolerance

- [FTCS-29 - *AR-SMT*, Rotenberg, June 99]
- Formal analysis left for future work
- Assumptions
  - Single transient fault model
  - Fault eventually manifests as a bad value, appearing as a misprediction in R-stream
- *Time redundancy* provides certain guarantees
  - Single fault may cause simultaneous *but different* errors in both A-stream and R-stream
  - Streams are shifted in time: guarantees the two redundant copies of an instruction will not both be affected

Fault Tolerance



Fault Tolerance

- Scenario #1
  - Fault detectable, but indistinguishable from IR-misprediction!
  - Must assume IR-misprediction

1. Don't make any special considerations
   - If fault does not flip R-stream arch. state, don't care about source of problem — recovery works! (pipeline coverage)
   - Otherwise, the system is bad and we are unaware of it
2. Try to distinguish faults
   - If no prior unresolved IR-predictions, it's a fault — invoke software (e.g., restart)
   - Otherwise, default to 1) above
3. ECC on R-stream register file, D$: always fault tolerant

Fault Tolerance

- Scenario #2
  - Affected R-stream instruction doesn't have redundant A-stream equivalent, nothing to compare with
  - May propagate and detect later, but possibly too late
  - Currently: no coverage for scenario #2 (future work)
- Scenario #3
  - IR-misprediction detected before fault can cause problems
- Summary
  - Can (potentially) tolerate all faults that affect redundantly executed instructions

Talk Outline

✓Introduction to CRT

✓Microarchitecture description

✓Understanding performance benefits

✓Performance results

✓Understanding fault tolerance benefits

➔The bigger picture: harnessing CMP/SMT processors

➔Conclusions

➔Future work

Bigger Picture

1. Multithreaded processors will be prevalent in the future.
2. There is vast, untapped potential for harnessing multithreaded processors in new ways.
3. A single multithreaded processor can and should flexibly provide many capabilities.
4. A multithreaded processor can and should be leveraged without making fundamental changes to existing components/mechanisms.

CRT is a concrete application of these principles.

Conclusions

- CRT: flexible, comprehensive functionality within a single strategic architecture
  - multiprogrammed/parallel workload performance (CMP/SMT)
  - single-program performance with improved reliability (CRT)
  - high reliability with less performance impact (AR-SMT / SRT)
- Performance results
  - 12% average improvement harnessing otherwise unused PE
  - CRT on 2 small cores has comparable IPC to 1 large core, but with faster clock and more flexible architecture
  - Majority of benchmarks show significant A-stream reduction (50%); CRT on 8-way SMT improves their performance 10%-20%
  - Benefits: resolving mispredictions in advance + quality value prediction
  - Demonstrated importance of bypassing instruction fetching

Future Work

1. CRT
   - Understanding performance
   - Microarchitecture design space
   - Pipeline organization
   - Fault tolerance
   - System-level issues
   - Adaptivity
2. Fundamental variations of CRT
   - Streamlining R-stream
   - Other A-stream shortening approaches
   - Scaling to N threads
   - Approximate A-streams
3. Other novel CMP/SMT applications

Related Work

- Fault tolerance in high-perf. commodity microprocessors
  - AR-SMT [Rotenberg, FTCS-29]
  - DIVA [Austin, MICRO-32]
  - SRT [Reinhardt, Mukherjee, ISCA-27]
  - FTCS-29: panel on using COTS in reliable systems
  - [Rubinfeld, *Computer*]
- Much prior work exploiting repetition, redundancy, predictability in programs
  - Instruction reuse, block reuse, trace-level reuse, computation reuse
  - Value speculation, silent writes
  - Motivation for creating shorter A-stream

Related Work

- Understanding backward slices, pre-execution
  - [Farcy et. al., MICRO-31], [Zilles&Sohi, ISCA-27], [Roth et. al., ASPLOS-8 / Tech Reports]
    - Explicitly identify difficult computation chains, possibly for pre-execution
    - Instruction-removal is “inverted” with same effect: A-stream is less-predictable subset but runs ahead
    - A-stream — entire, redundant program instead of many specialized kernels
- Speculative multithreading [e.g., Multiscalar, DMT]
  - Replicated programs: no forking/merging of spec. thread state needed
- DataScalar: redundant programs to eliminate memory reads