Master/Slave Speculative Parallelization

Craig Zilles (U. Illinois)
Guri Sohi (U. Wisconsin)

The Basics

- #1: a well-known problem:

  On-chip Communication

- #2: a well-known opportunity:

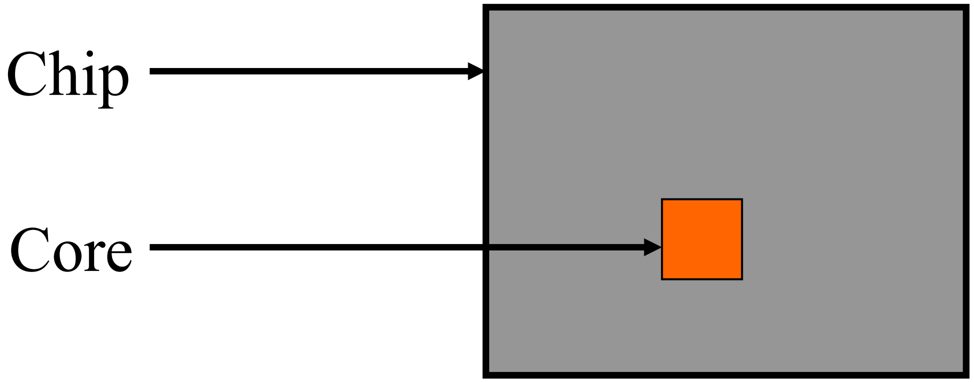
  Program Predictability

- #3: our novel approach to #1 using #2

Problem: Communication

- Cores becoming “communication limited”
  - Rather than “capacity limited”
- Many, many transistors on a chip, but…
- Can’t bring them all to bear on one thread
  - Control/data dependences = freq. communication

Best core << chip size

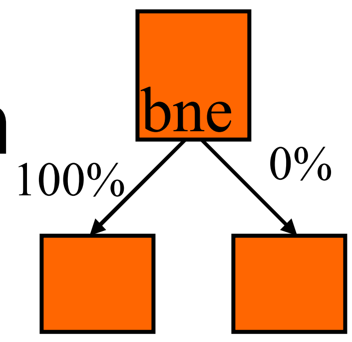


- Sweet spot for core size
  - Further size increases either hurts Mhz or IPC

How can we maximize core's efficiency?

Opportunity: Predictability

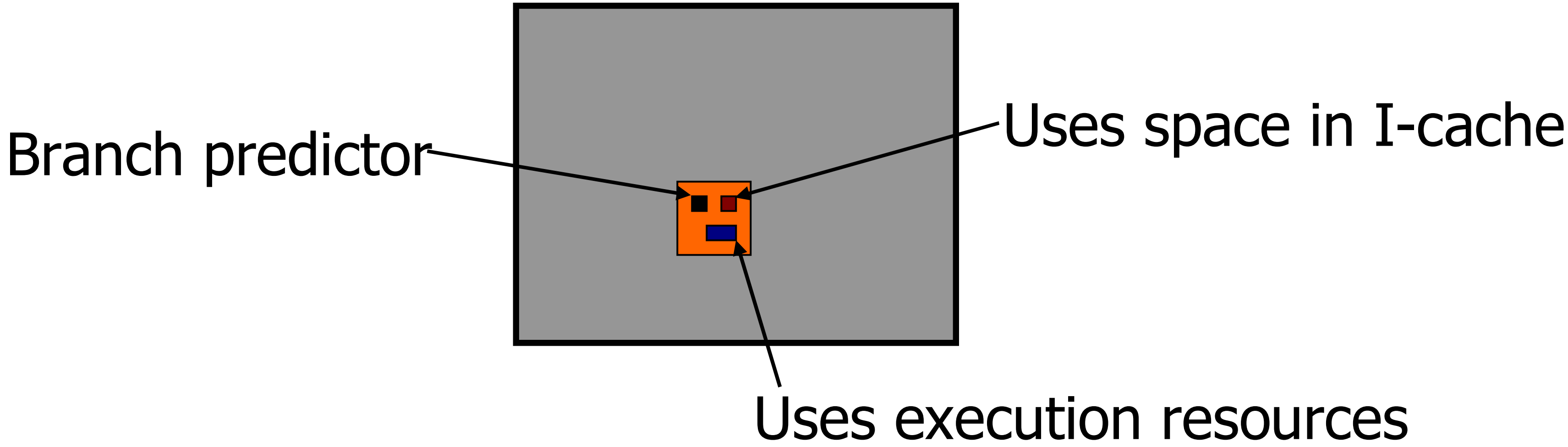
- Many programs behaviors are predictable
  - Control flow, dependences, values, stalls, etc.
- Widely exploited by processors/compilers
  - But, not to help increase effective core size
  - Core resources used to make, validate pred's
- Example: perfectly-biased branch



Speculative Execution

- Execute code before/after branch in parallel
- Branch is fetched, predicted, executed, retired

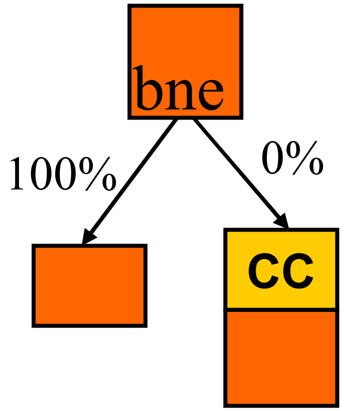
All of this occurs in the core



Not just the branch, but its backwards slice

Trace/Superblock Formation

- Optimize code assuming the predicted path
  - Reduces cost of branch and surrounding code
  - Prediction implicitly encoded in executable



- Code still verifies prediction
  - Branch & slice still fetched, executed, committed, etc.

All of this occurs on the core

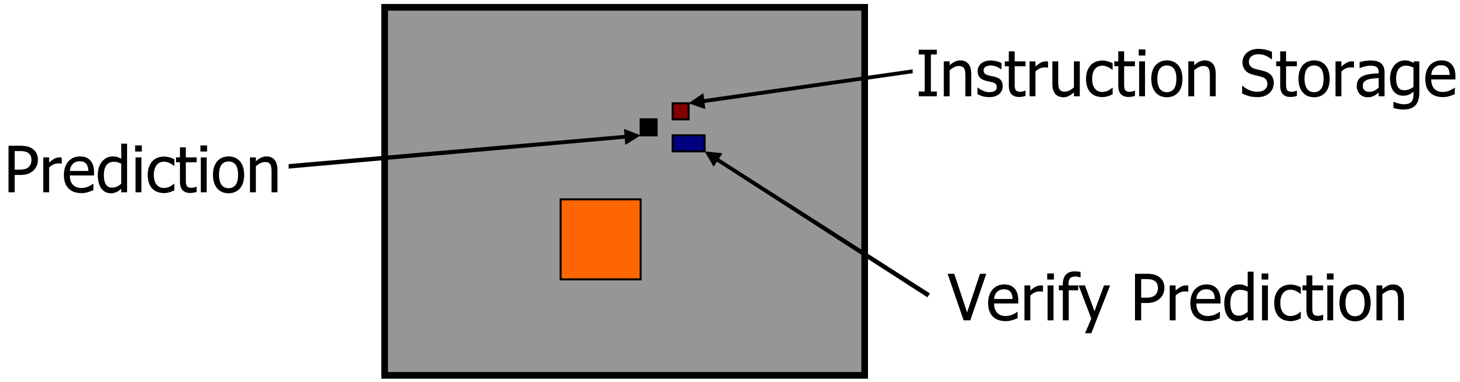
Why waste core resources?

- The branch is perfectly predictable!

The core should only execute instructions that are not statically predictable!

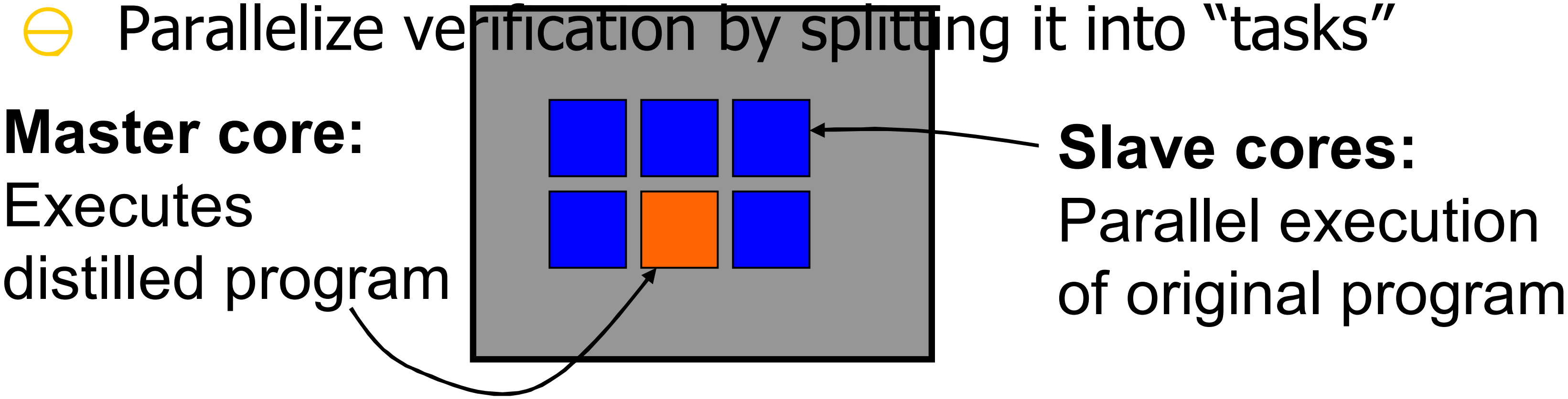
If not in the core, where?

- Anywhere else on chip!
- Because it is predictable:
  - Doesn't prevent forward progress
  - We can tolerate latency to verify prediction



A concrete example: Master/Slave Speculative Parallelization

- Execute "distilled program" on one processor
  - A version of program with predictable inst's removed
  - Faster than original, but not guaranteed to be correct
- Verify predictions by executing original program
  - Parallelize verification by splitting it into "tasks"

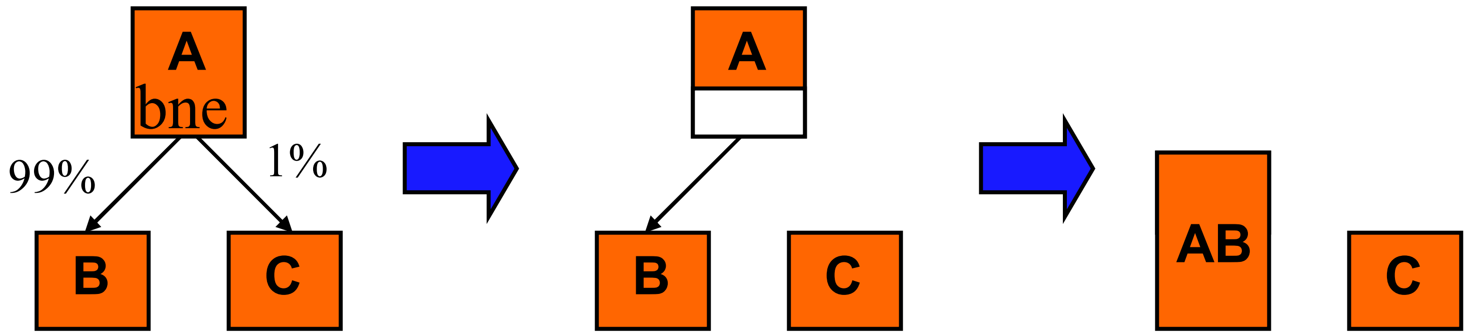


Talk Outline

- Removing predictability from programs
  - “Approximation”
- Externally verifying distilled programs
  - Master/Slave Speculative Parallelization (MSSP)
- Results Summary
- Summary

Approximation Transformations

- Pretend you've proven the common case
  - Preserve correctness in the common case
  - Break correctness in uncommon case
    - Use profile to know the common case



Not just for branches

- Values:

  Load is highly invariant (usually gets 7)

  ~~ld r13, 0(X)~~ addi $zero, 7, r13

- Memory Dependences:

  st r12, 0(A)

  ~~ld r11, 0(B)~~

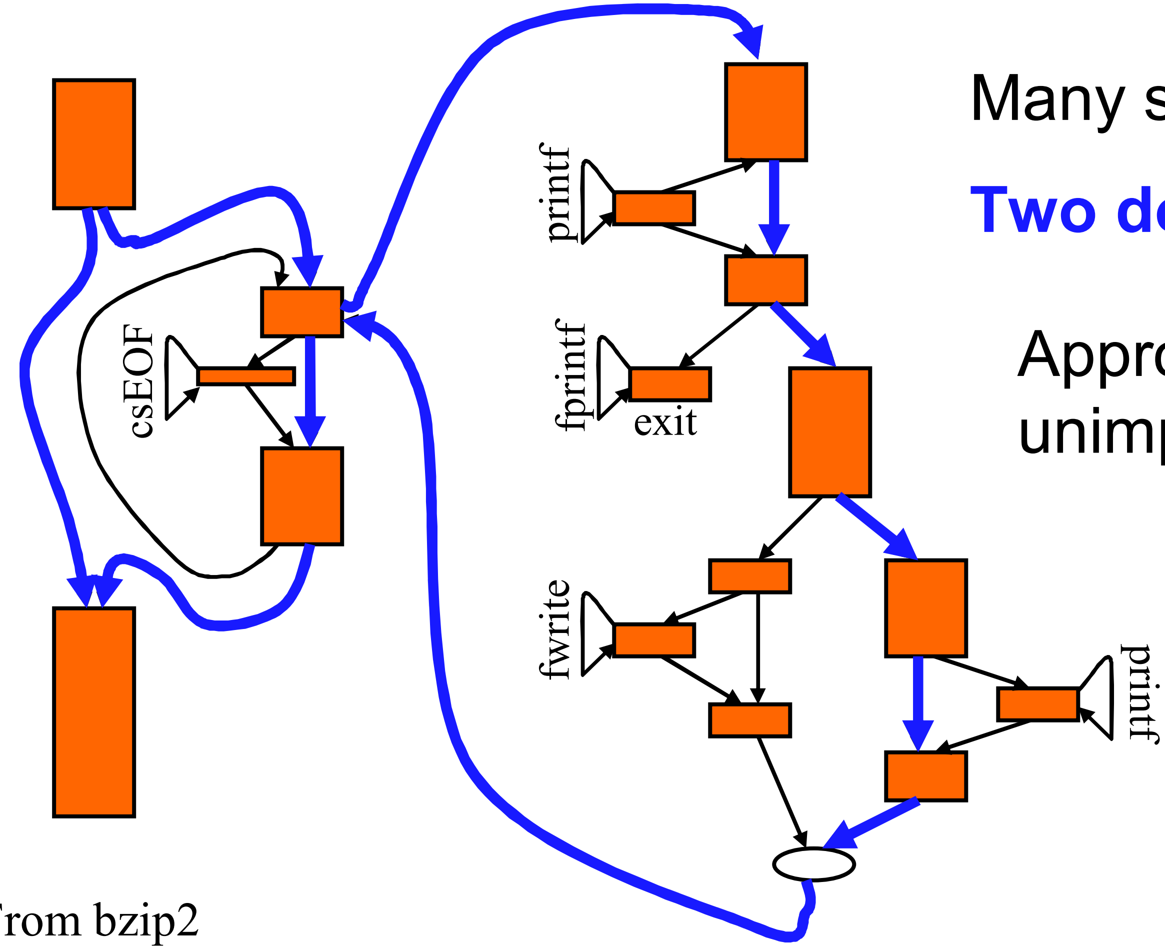
  mv r12, r11

  A and B ~~may~~ never / always alias

  If rarely alias in practice?

  If almost always alias?

Enables Traditional Optimizations



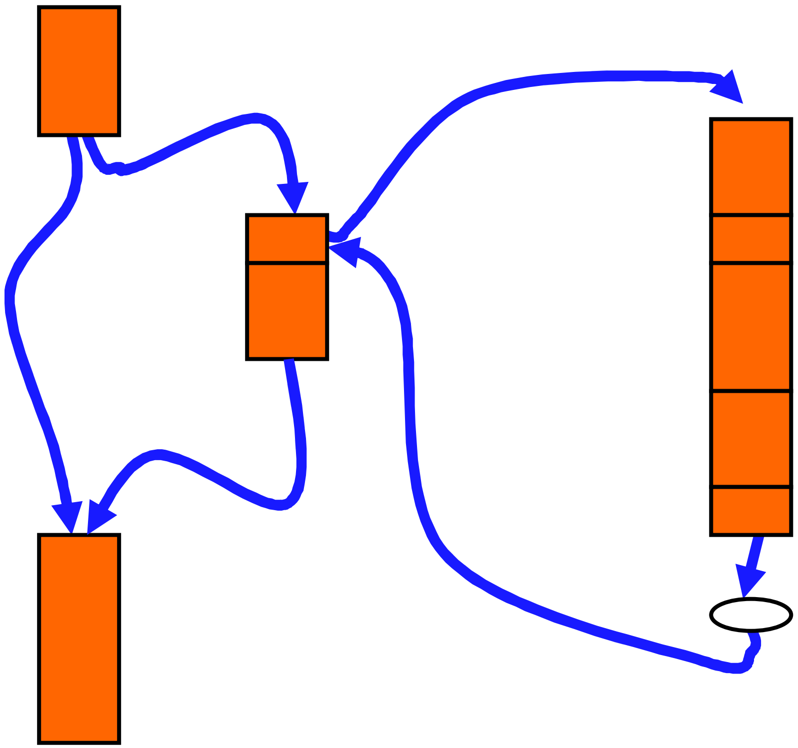
Many static paths

Two dominant paths

Approximate away unimportant paths

From bzip2

Enables Traditional Optimizations



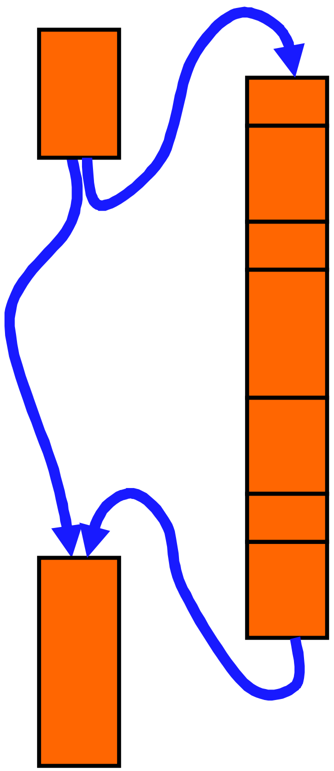
From bzip2

Many static paths

Two dominant paths

Approximate away unimportant paths

Enables Traditional Optimizations



From bzip2

Many static paths

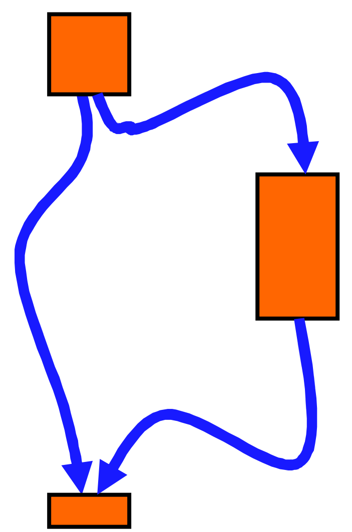
Two dominant paths

Approximate away unimportant paths

Very straightforward structure

Easy for compiler to optimize

Enables Traditional Optimizations



Many static paths

Two dominant paths

Approximate away unimportant paths

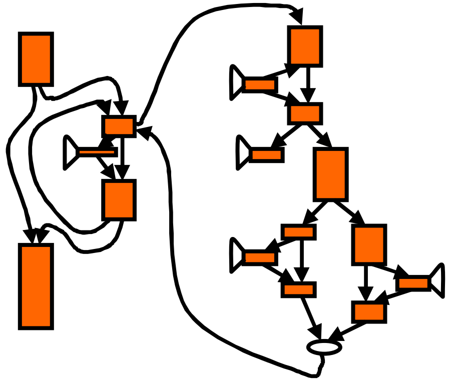
Very straightforward structure

Easy for compiler to optimize

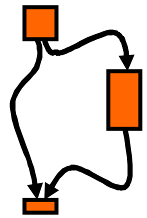
From bzip2

Effect of Approximation

Original Code



Distilled Code



- Equivalent 99.999% of the time, better execution characteristics
  - Fewer dynamic instructions: ~1/3 of original code
  - Smaller static size: ~2/5 of original code
  - Fewer taken branches: ~1/4 of original code
  - Smaller fraction of loads/stores
- Shorter than best non-speculative code
  - Removing checks: code incorrect .001% of the time

Talk Outline

- Removing predictability from programs
  - "Approximation"
- Externally verifying distilled programs
  - Master/Slave Speculative Parallelization (MSSP)
- Results Summary
- Summary

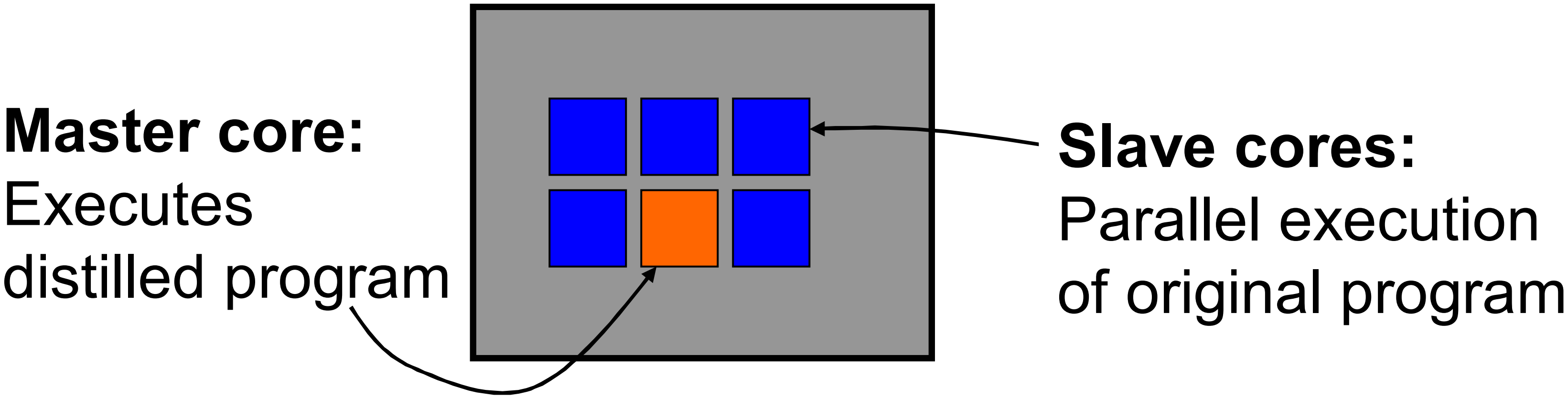
Goal

- Achieve performance of distilled program
- Retain correctness of original program

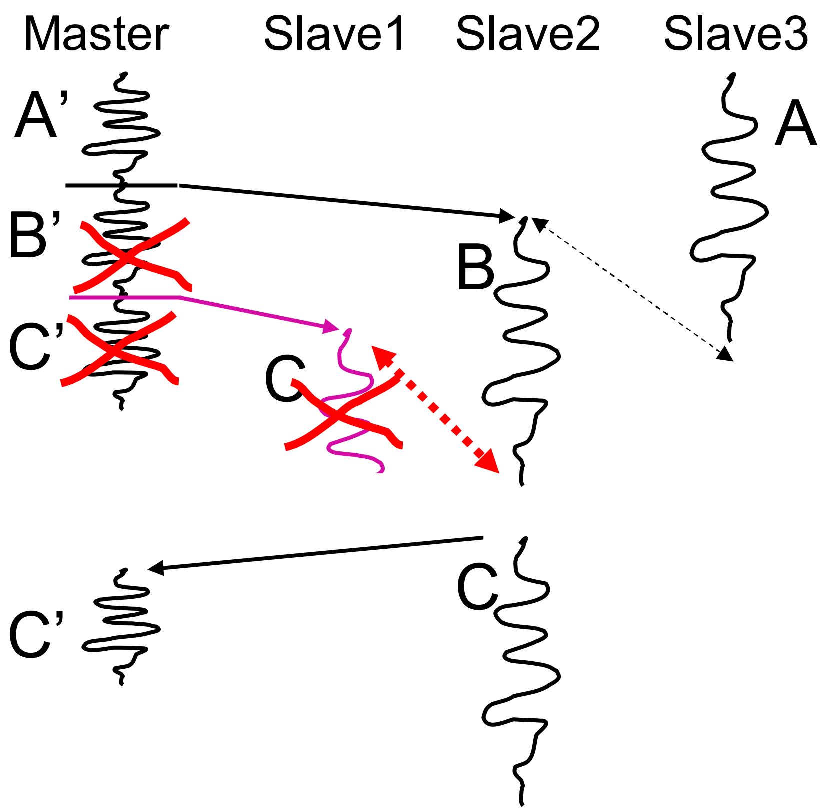
- Approach:
  - Use distilled code to speed original program

Checkpoint parallelization

- Cut original program into “tasks”
  - Assign tasks to processors
- Provide each a checkpoint of registers & memory
  - Completely decouples task execution
  - Tasks retrieve all live-ins from checkpoint
- Checkpoints taken from distilled program
  - Captured in hardware
  - Stored as a “diff” from architected state


Master core:
Executes
distilled program
Slave cores:
Parallel execution
of original program

Example Execution



Start Master and Slave from architected state

Take checkpoint, use to start next task

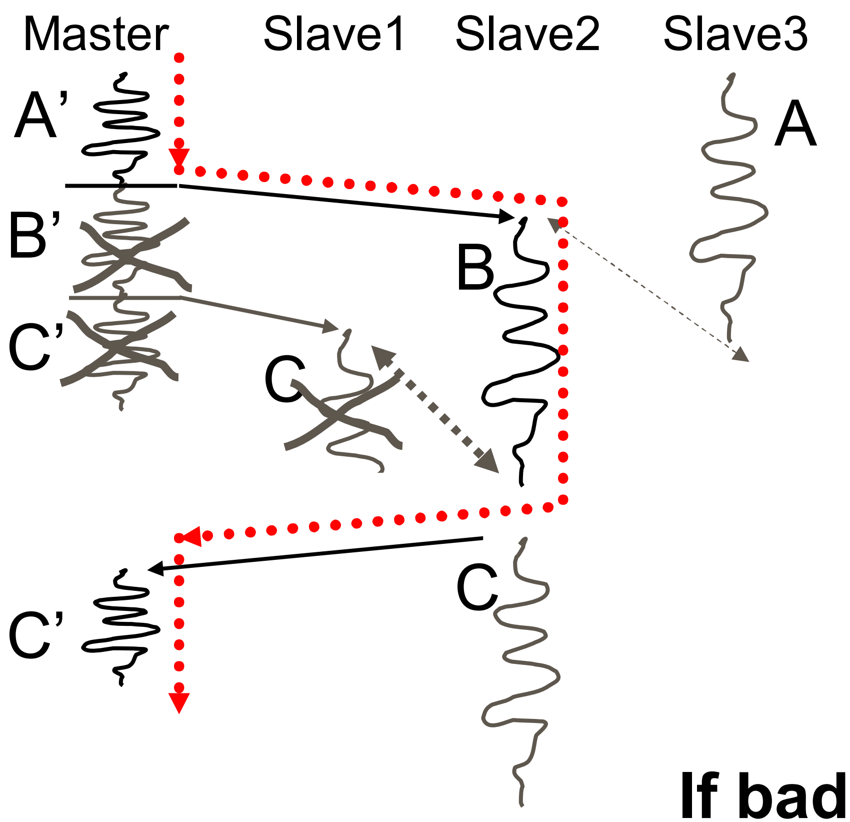
Verify B's inputs with A's outputs; commit state

Bad Checkpoint @ C

Detected at end of B

Squash, restart from architected state

MSSP Critical Path



If checkpoints correct:

- through distilled program
- no communication latency
- verification in background

Bad checkpoints:

- through original program
- interprocessor comm.

If bad checkpoints are rare:

- performance of distilled program
- tolerant of communication latency

Talk Outline

- Removing predictability from programs
  - "Approximation"
- Externally verifying distilled programs
  - Master/Slave Speculative Parallelism (MSSP)
- Results Summary
- Summary

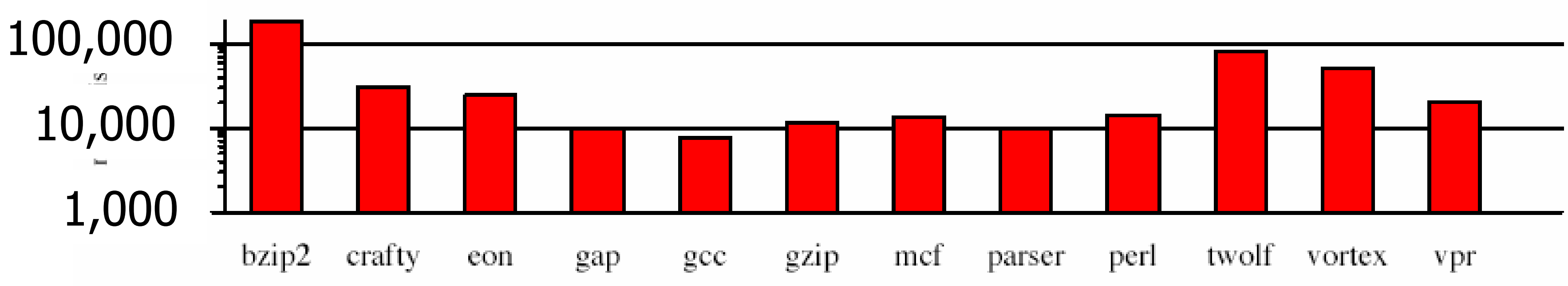
Methodology

- First-cut distiller
  - Static binary-to-binary translator
  - Simple control flow approximations
  - DCE, inlining, register re-allocation, save/restore elimination, code layout…
- HW model: 8-way CMP of 21264's
  - 10 cycle interconnect latency to shared L2
- Spec2000 Integer benchmarks on Alpha

Results Summary

- Distilled Programs can be accurate
  - 1 task misspeculation per 10,000 instructions
- Speedup depends on distillation
  - 1.25 h-mean: ranges from 1.0 to 1.7 (gcc, vortex)
  - (relative to uniprocessor execution)
- Modest storage requirements
  - Tens of kB at L2 for speculation buffering
- Decent latency tolerance
  - Latency 5 -> 20 cycles: 10% slowdown

Distilled Program Accuracy



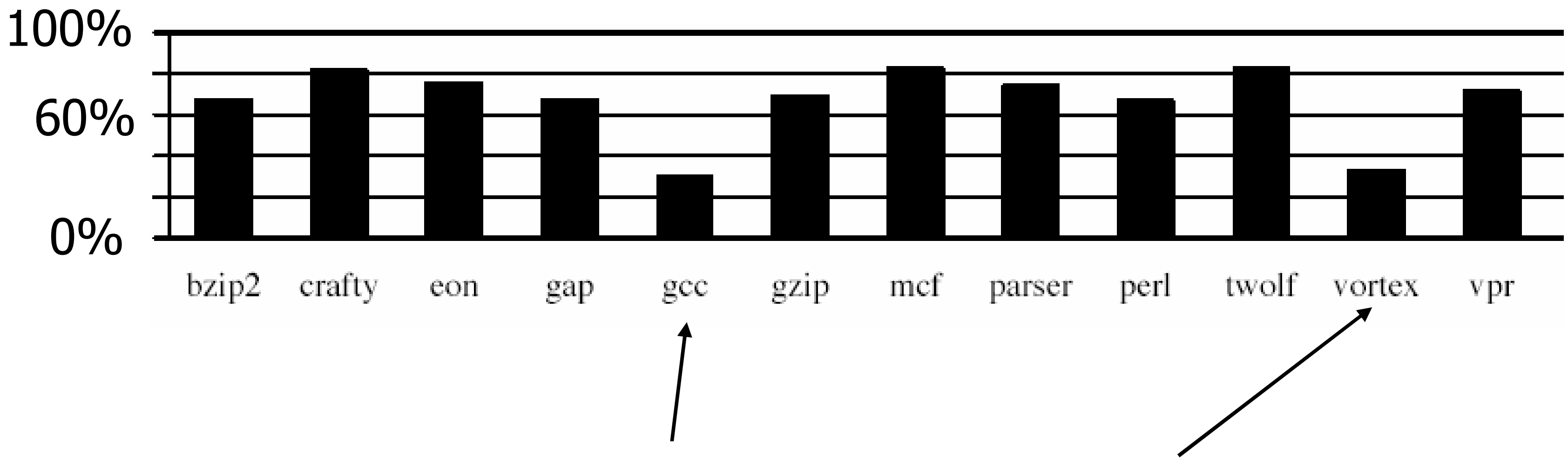
Average distance between task misspeculations:

> 10,000 original program instructions

Distillation Effectiveness

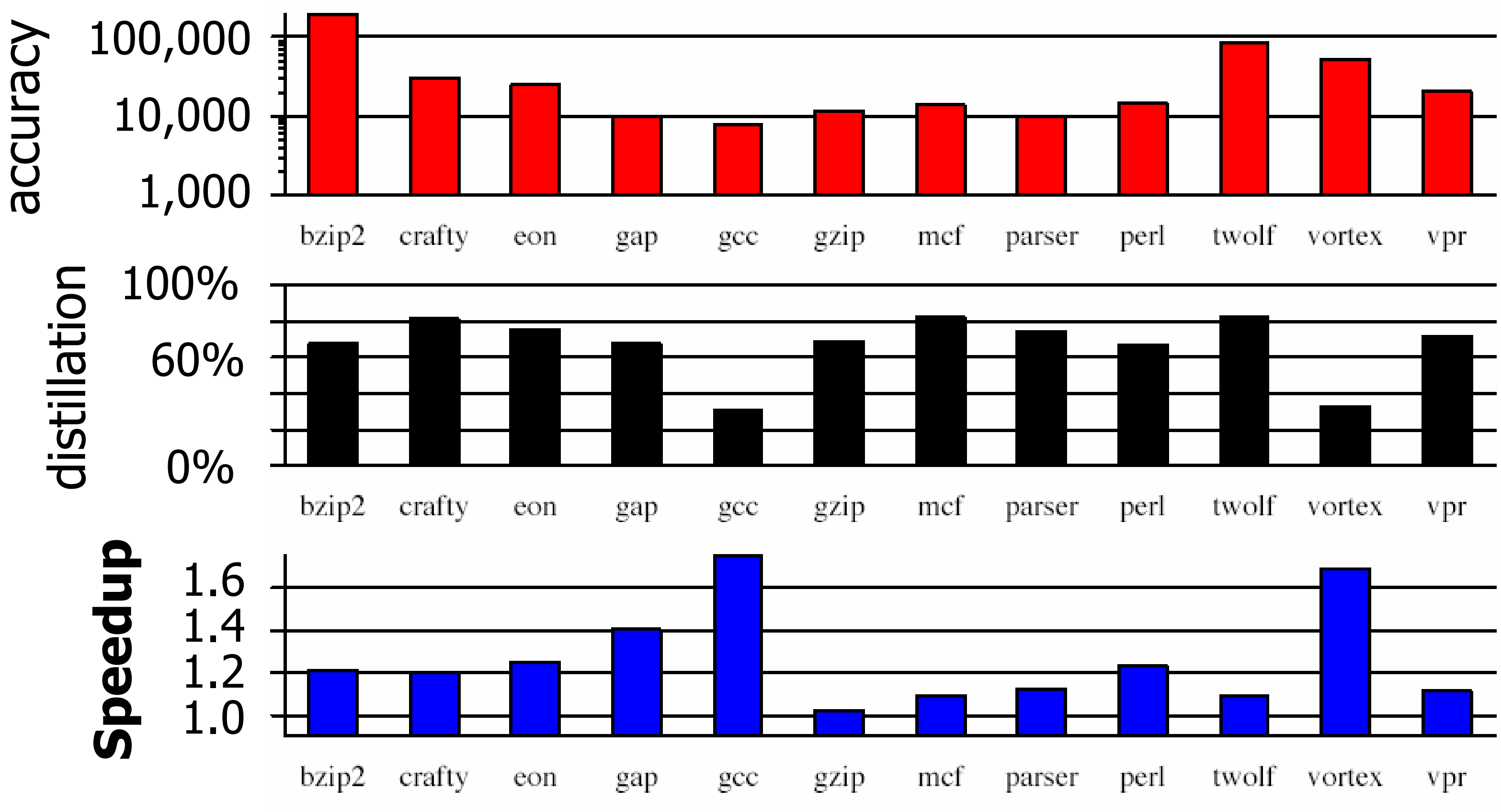
$$\frac{\text{Instructions retired by Master}}{\text{Instructions retired by Slave}}$$ (distilled program) / (original program

(not counting nops)



Up to two-thirds reduction

Performance



Performance scales with distillation effectiveness

Related Work

- Slipstream
- Speculative Multithreading
- Pre-execution
- Feedback-directed Optimization
- Dynamic Optimizers

Summary

- Don’t waste core on predictable things
  - “Distill” out predictability from programs
- Verify predictions with original program
  - Split into tasks: parallel validation
  - Achieve the throughput to keep up
- Has some nice attributes (ask offline)
  - Can support legacy binaries, latency tolerant, low verification cost, complements explicit parallelism