Master/Slave Speculative Parallelization and Approximate Code

Craig Zilles and Guri Sohi

October 7, 2002

Computer Sciences Department

University of Wisconsin - Madison

Overview

Goal: faster single-threaded program execution

- minding complexity & communication considerations

Concept: Code Approximation

- generate faster, but imperfect copy of program

Execution Model: Master/Slave Speculative Parallelization

- performance of approximate code
- correctness of original program

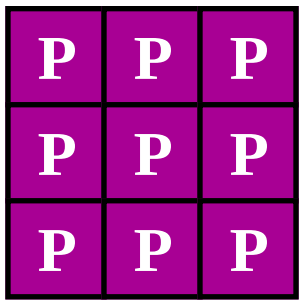
Motivation

Moore's Law:

- many transistors → potential for higher performance
- serious design challenges
  - physical constraints
  - design/verification effort

Chip Multiprocessor (CMP):

- replicate medium-sized processor
  + shorter wires
  + replication reduces design effort
- software challenges
  - must find something for processors to do!



Need Thread-level Parallelism

Some workloads have existing thread-level parallelism

- server, scientific, batch/throughput

... but, writing parallel programs is hard!!

- correct sequential programs are hard enough
- most programmers can't justify the additional effort

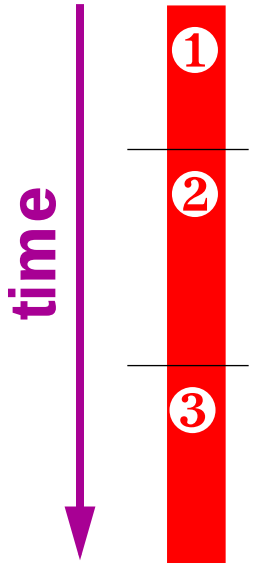
Traditional automatic parallelization not widely applicable

- analyze code, prove equivalency of parallel version
- proving difficult in most code

Relax analysis requirements with dynamic checks

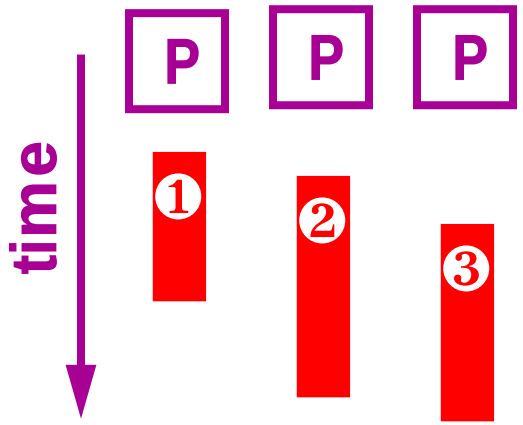
Speculative Parallelization (SP)

Speculative Parallelization

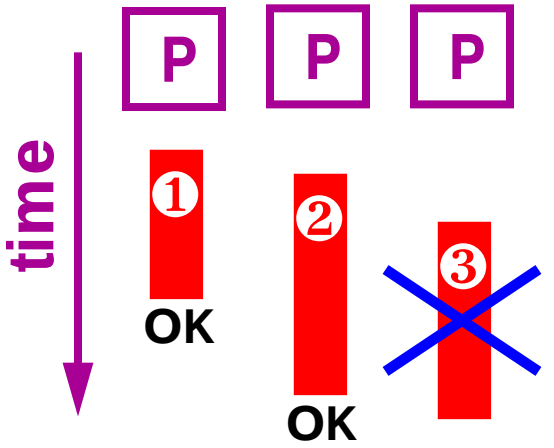


1) break program's execution into "tasks"

2) execute in parallel on separate processors



3) verify speculation



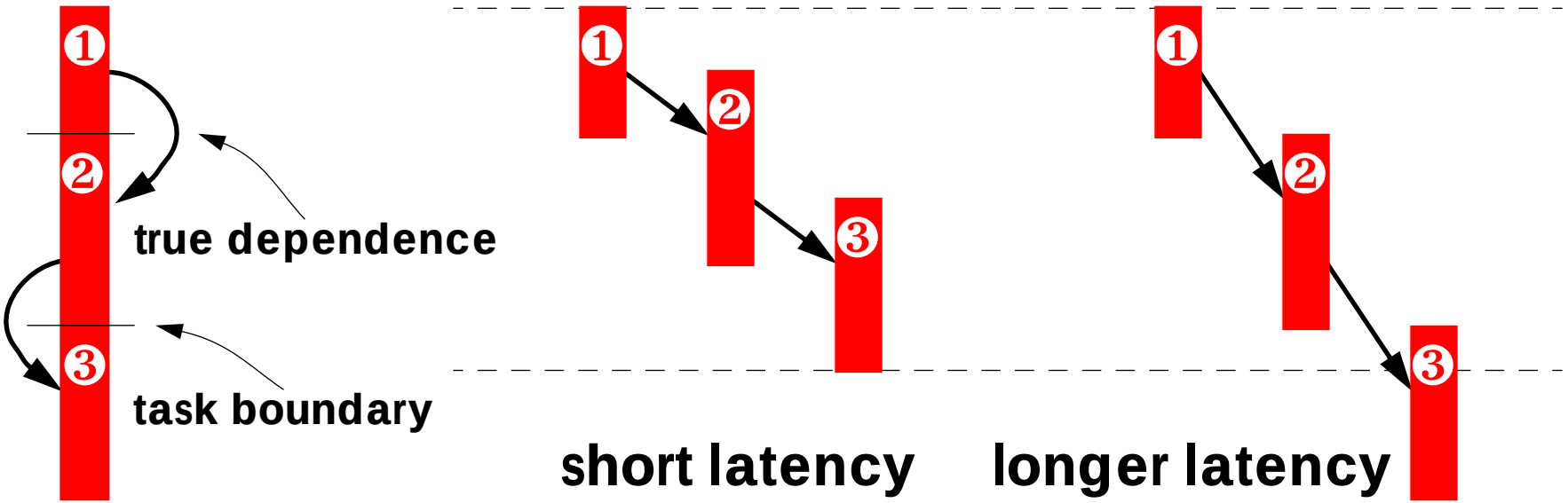
4) commit results, or squash task
(task misspeculation)

... but, Speculation is not a Panacea

Programs are rife with true dependences

In previous SP models:

- inter-processor communication latency serialized by exposed inter-task dependences



Avoiding Serializing Latency

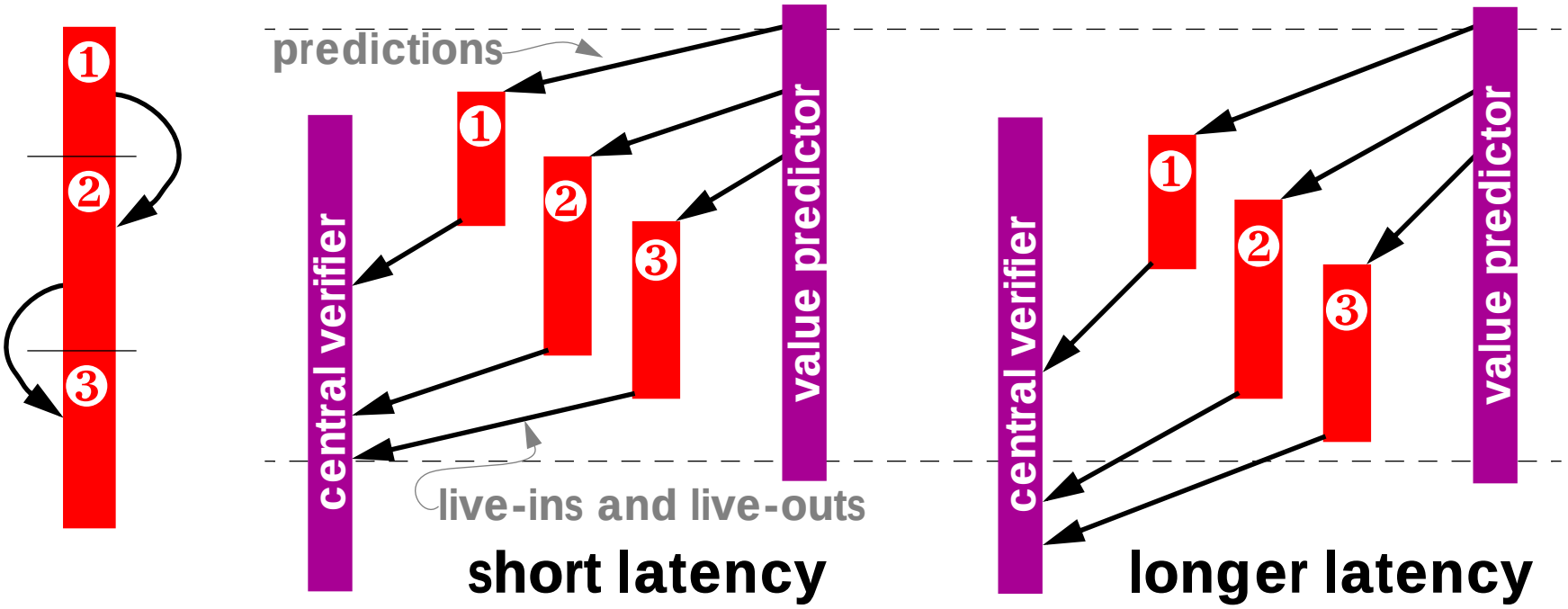
Predict inter-task communication

- assume: an accurate value predictor can be built
- predict all values at task boundaries (including PC)
- execute each task completely independently

Verify predictions

- compare predicted live-ins to architected state
  - basically, the re-use test (i.e., `Sodani & Sohi ISCA 1997`)
- buffer live-ins, live-outs during execution
- when head:
  - check live-ins, commit live-outs

Overlap Communication Latency



If infinite # of processors & perfect prediction accuracy:

- performance independent of communication latency
- value predictor & verifier determine execution rate

What is needed for this execution model?

Value Predictor:

- accurate
- high coverage
- fast

} gives latency tolerance

← determines performance

Verification/Commitment Mechanism:

- fast

Outline

- Overview
- Motivation
- Code Approximation

  "building a better value predictor"

  - the big idea
  - approximation example
- Distilled Programs
- Master/Slave Speculative Parallelization
- Evaluation
- Summary of Thesis Contributions

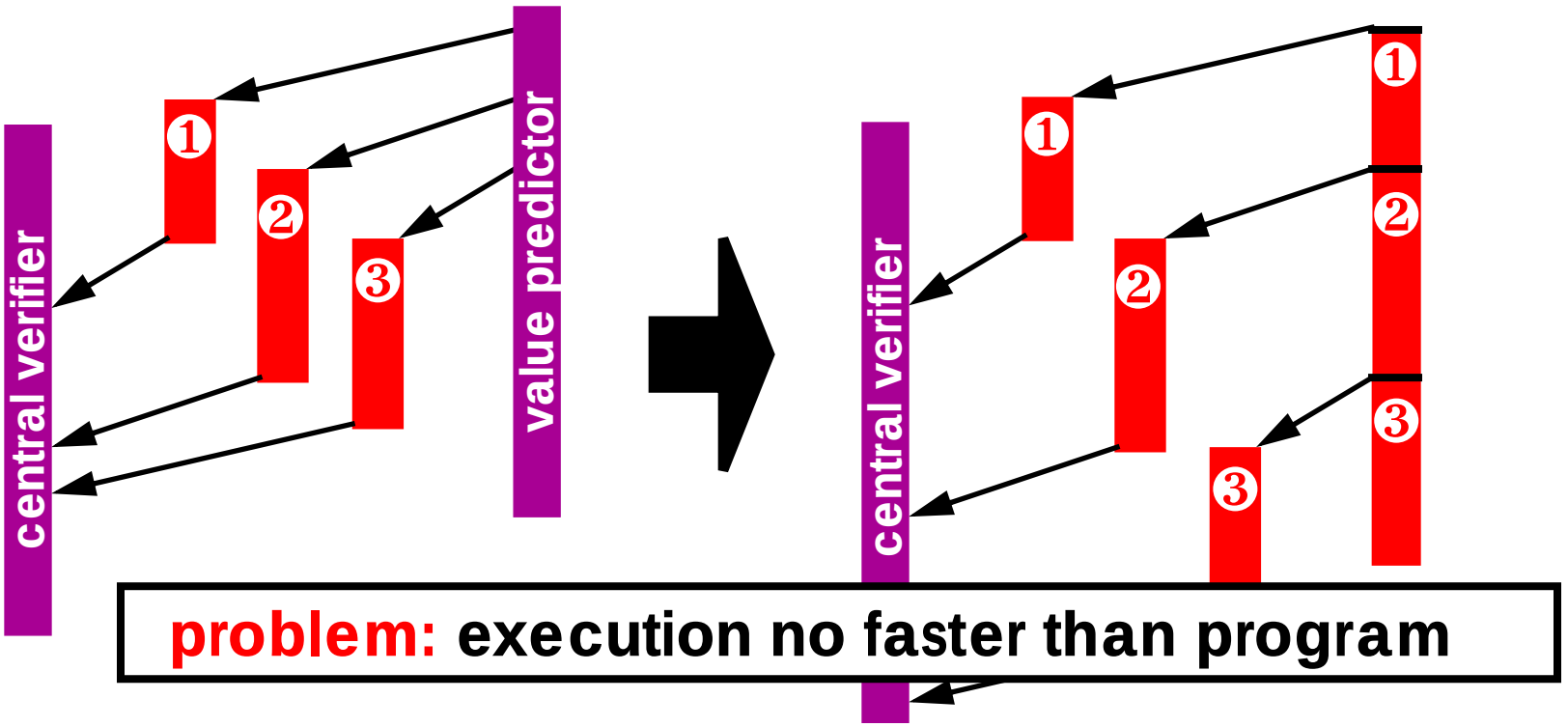
Building a Decent Value Predictor

Hardware value predictors have

- mediocre accuracy, coverage, or both

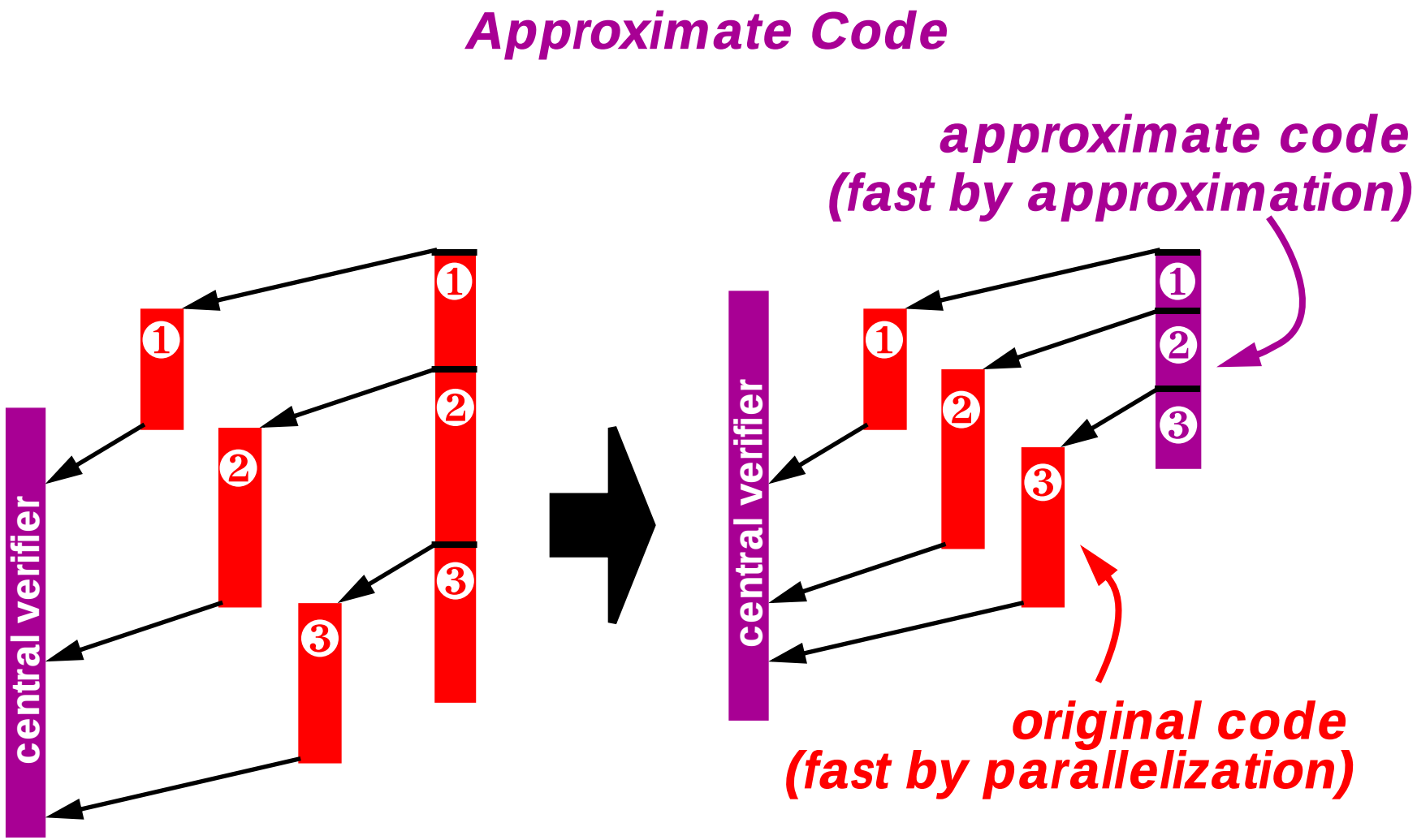
Insight #1/tautology:

Execution of program correctly computes the necessary program values



Building a Decent Value Predictor

Insight #2: Predictions need not be correct all of the time



Approximate Code - The Big Idea

In most programs large discrepancy between

- **what could happen**
- **what does happen**

Program Paths: `[Ball & Larus]`

- **>2^{32} potential acyclic paths**
- **<1000 paths cover >90% execution**

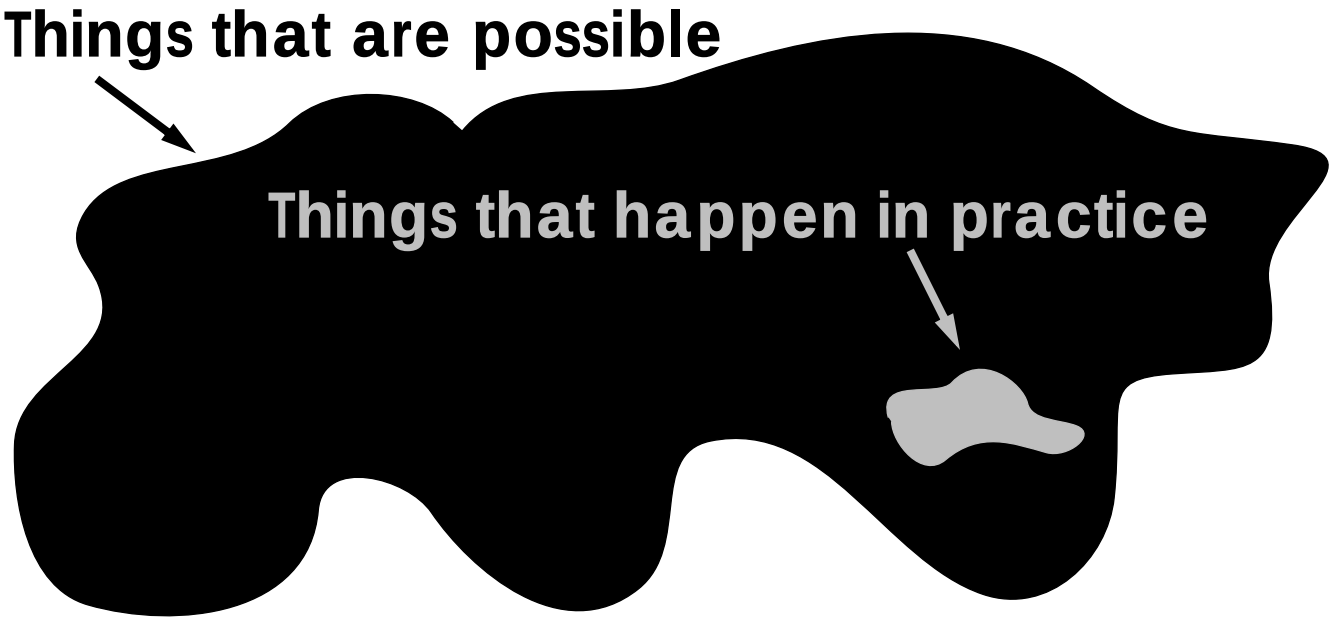
Pointer Analysis: `[Mock, et al.]`

- **dynamic points-to-sets 5 times smaller than static**

Approximate Code - The Big Idea

In most programs large discrepancy between

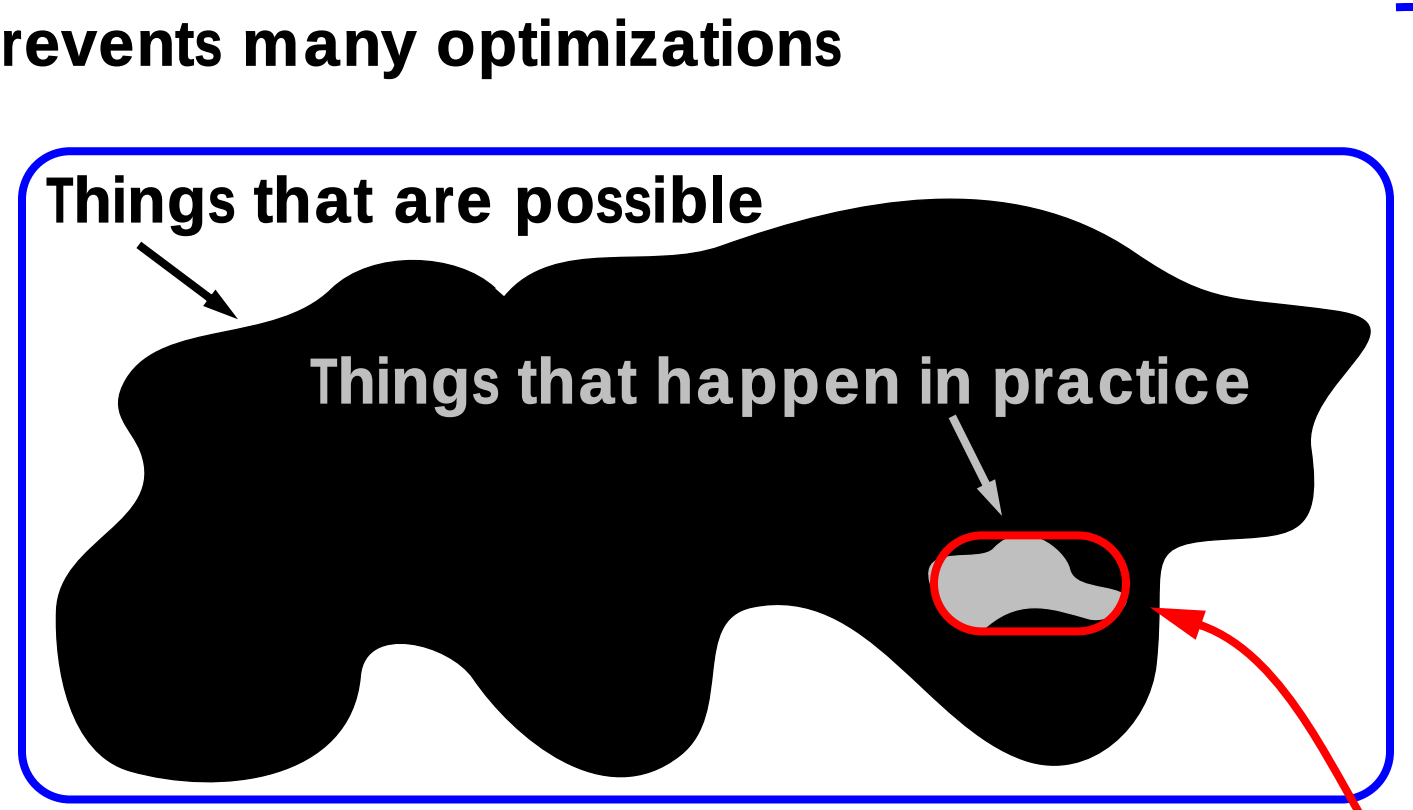
- what could happen
- what does happen



Approximate Code - The Big Idea

Traditionally compiled code

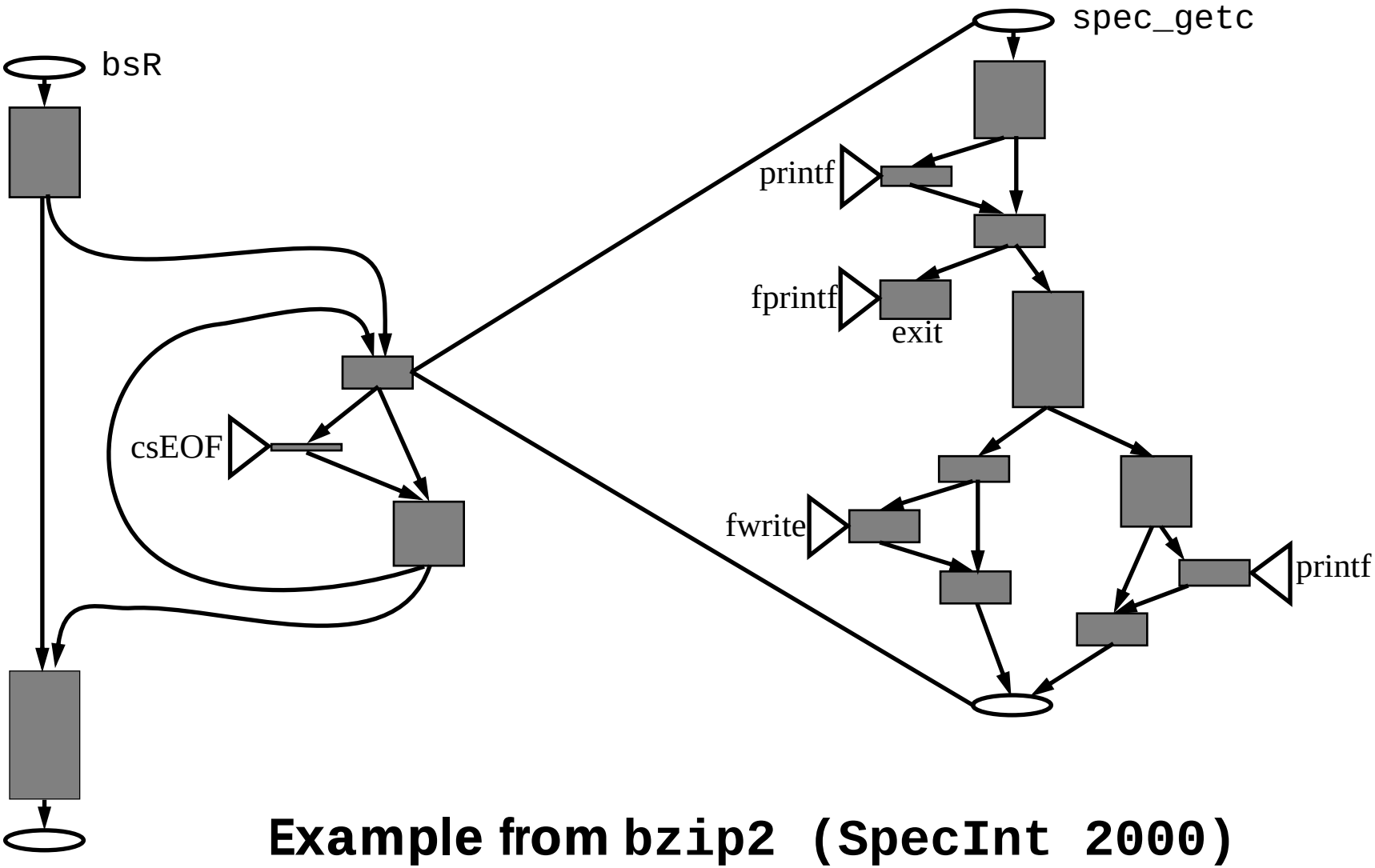
- must conservatively preserve all possible behaviors
- prevents many optimizations



Approximate Code

- can focus on common-case behavior
- enables many optimizations

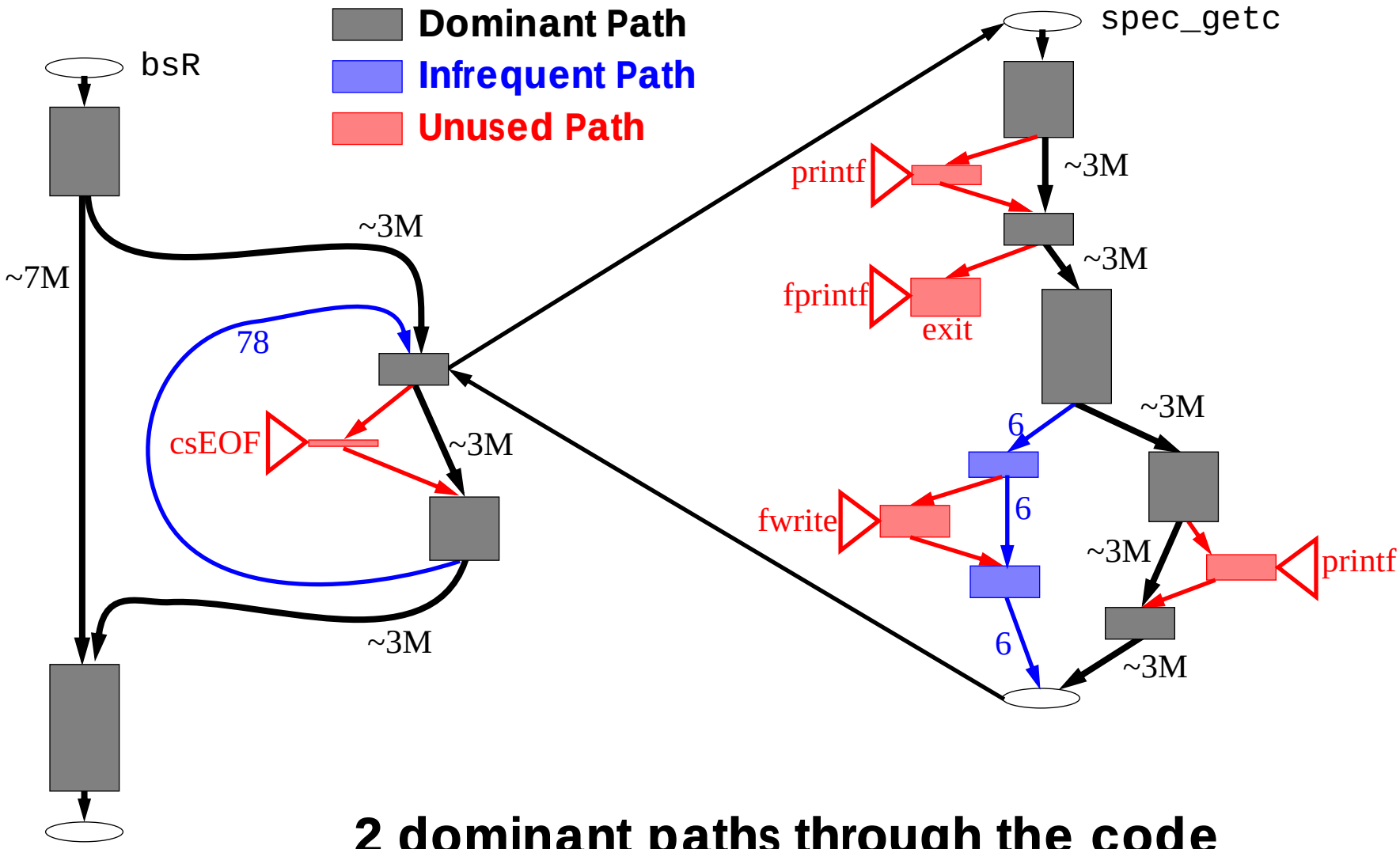
Approximation Example



Example from `bzip2` (SpecInt 2000)

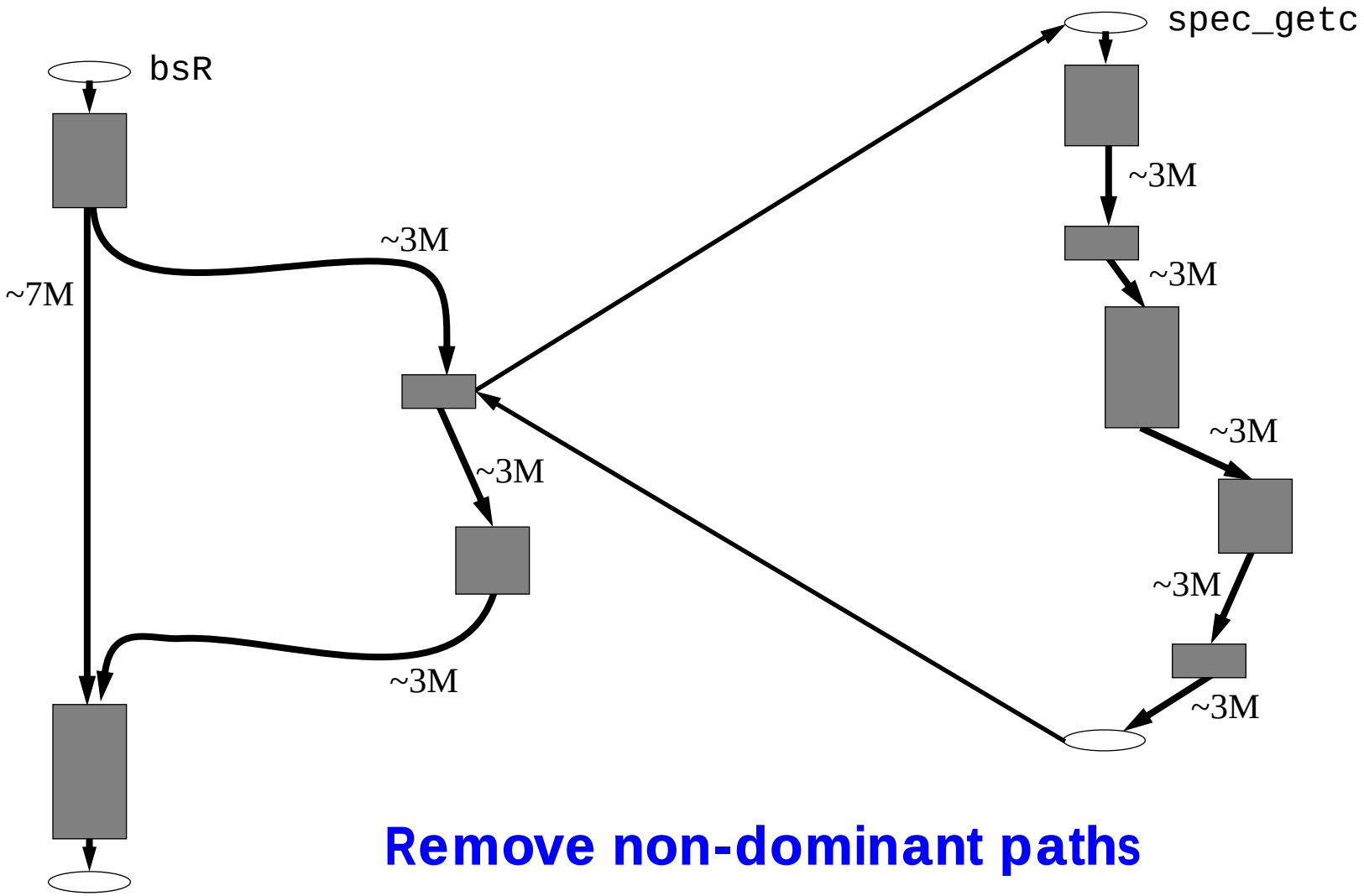
(represents 3% of total execution)

Approximation Example



2 dominant paths through the code

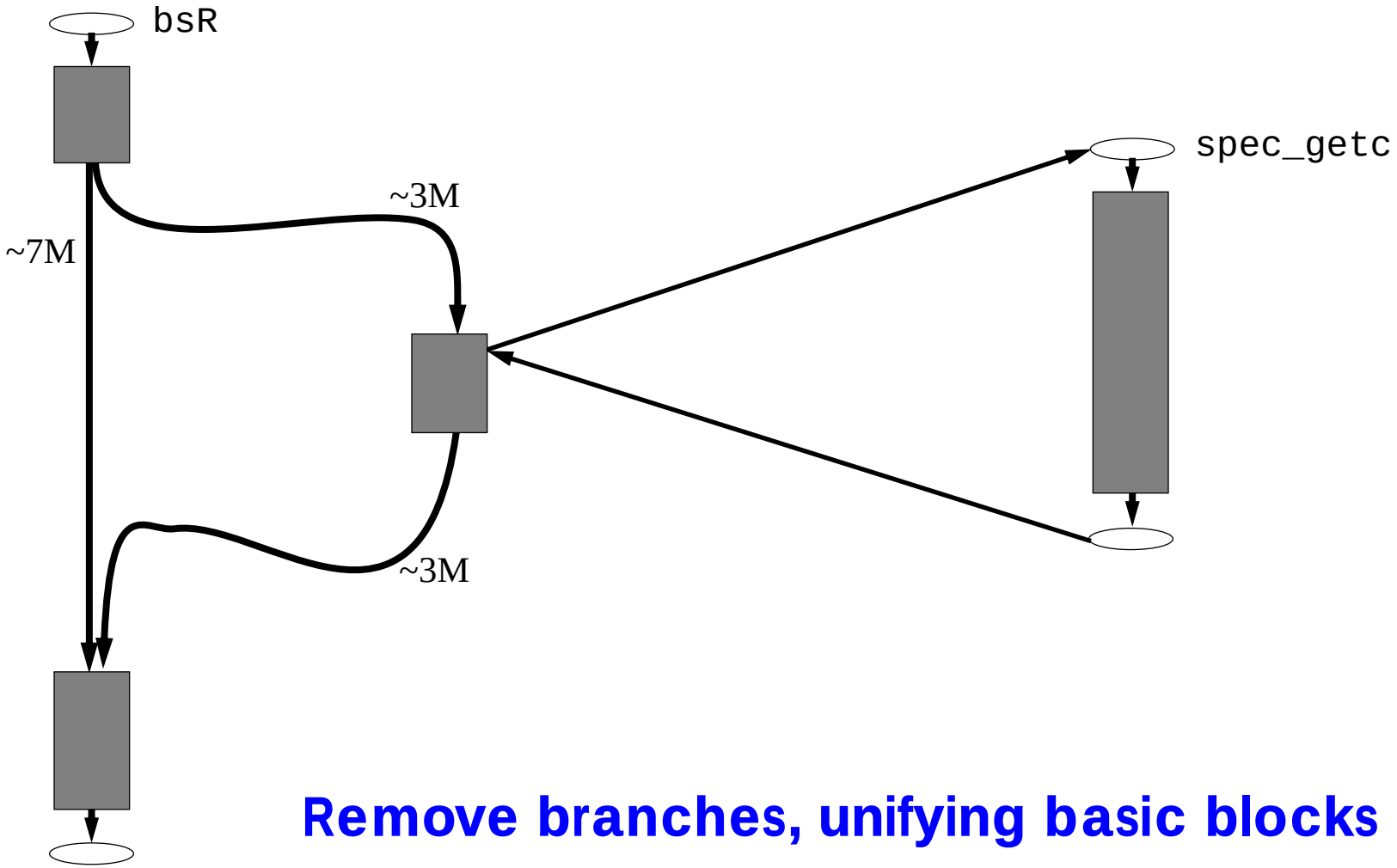
Constructing Approximate Code



Remove non-dominant paths

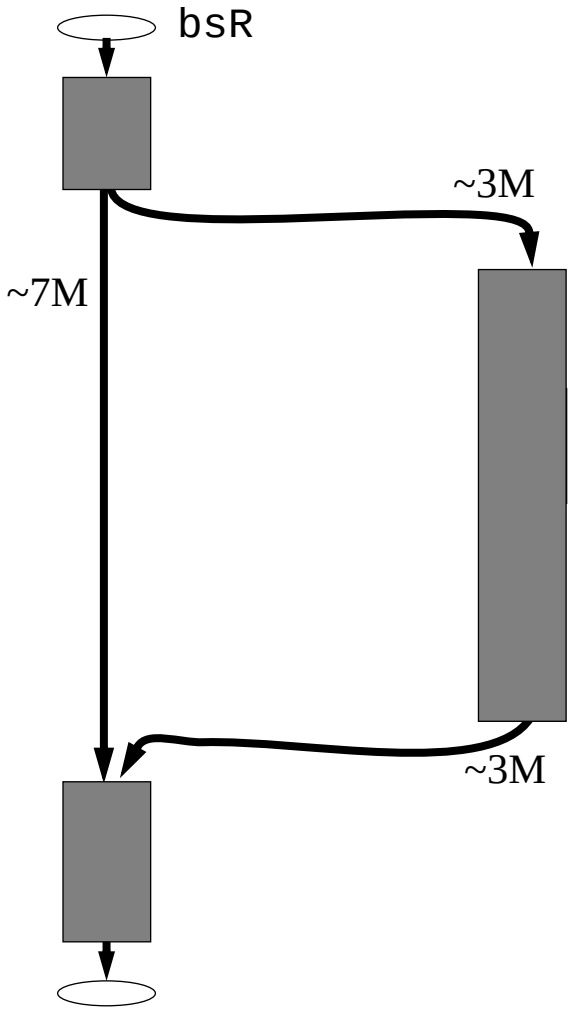
(code will be incorrect for removed paths)

Constructing Approximate Code, cont.



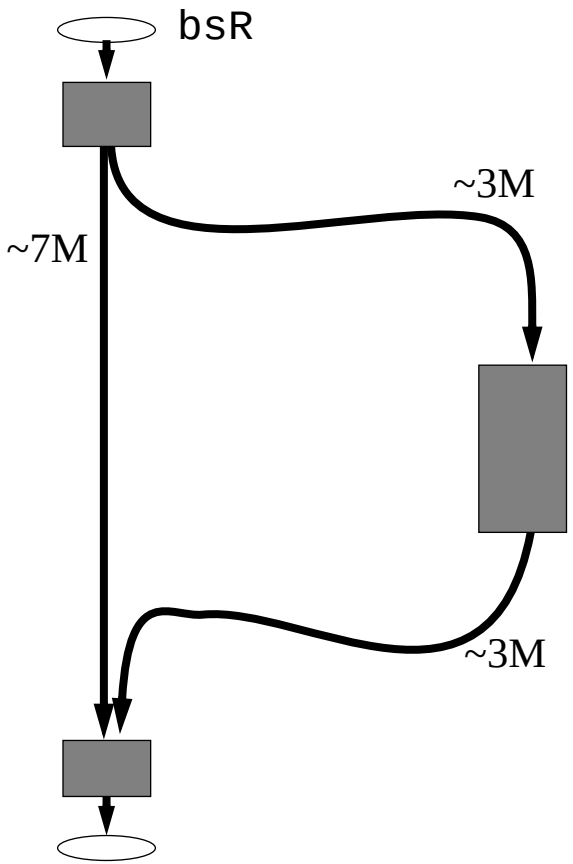
Remove branches, unifying basic blocks

Constructing Approximate Code, cont.



Inline function

Constructing Approximate Code, cont.



Apply traditional optimizations

simplified code → additional opportunities

Constructing Approximate Code

Root optimizations (speculative/unsafe)

- remove uncommon cases

Supporting optimizations (non-speculative/safe)

- new benefit enabled by root optimizations

Root Optimizations:

- biased branch elimination
- long dependence store elim.
- null operation elim.
- indirect-to-direct call conv.

eliminate instructions

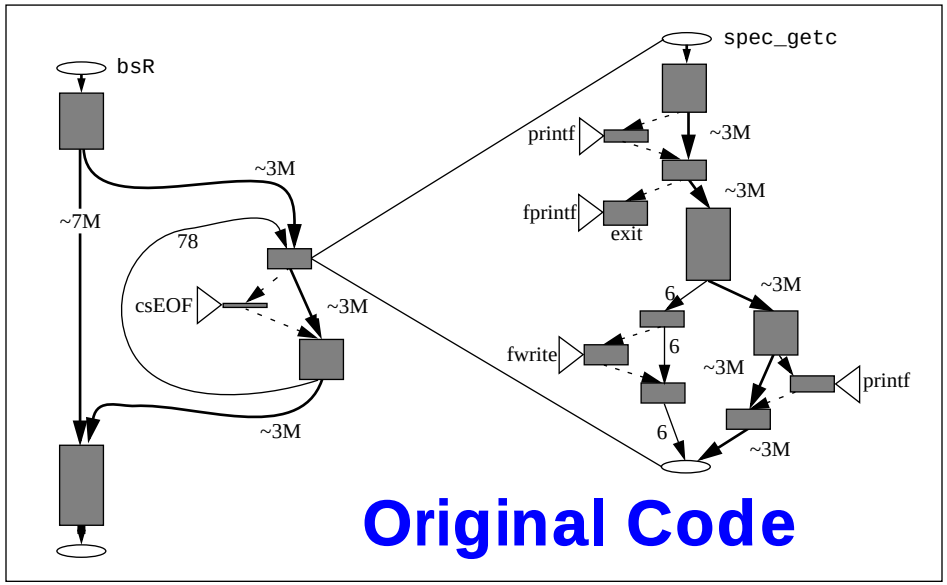
Supporting Optimizations:

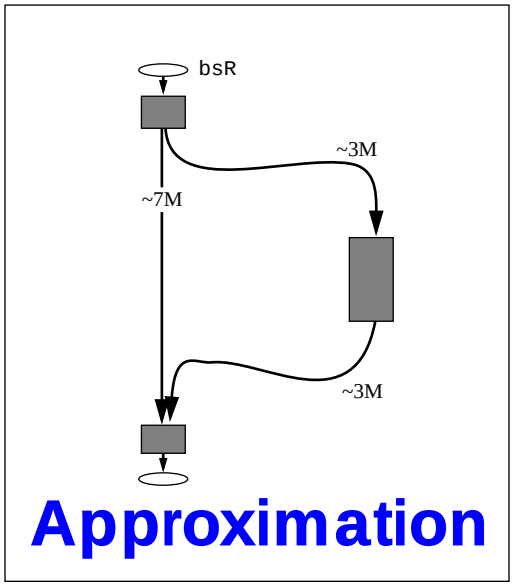
existing literature, e.g.,

- dead code elimination
- inlining
- register re-allocation
- partial redundancy elim.
- etc.

transform remaining code

Approximate Code





equivalent 99.999% of 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**

... but still, it is incorrect 0.001% of the time.

- **use approx. and orig. code together → MSSP**

Outline

- Overview
- Motivation
- Code Approximation
- **Distilled Programs**

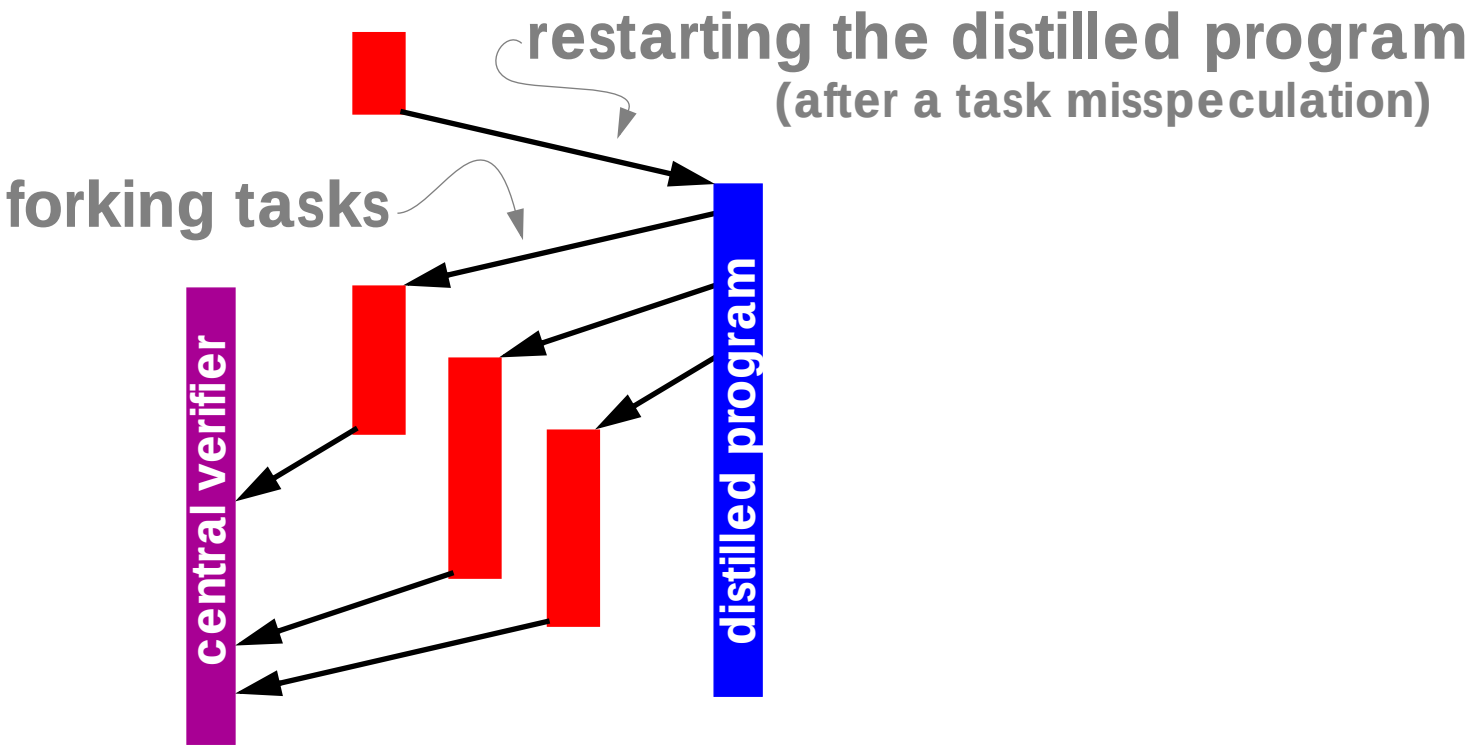
  "approximate code for MSSP"

  - **supporting transitions to original program**
  - **distilled program structure**
- **Master/Slave Speculative Parallelization**
- **Evaluation**
- **Summary of Thesis Contributions**

Using Approximate Code in MSSP

"Distilled Program"

- approximate version of program
- serves as the value predictor
- supports forking to/from original code



Supporting Transitions

Simplification

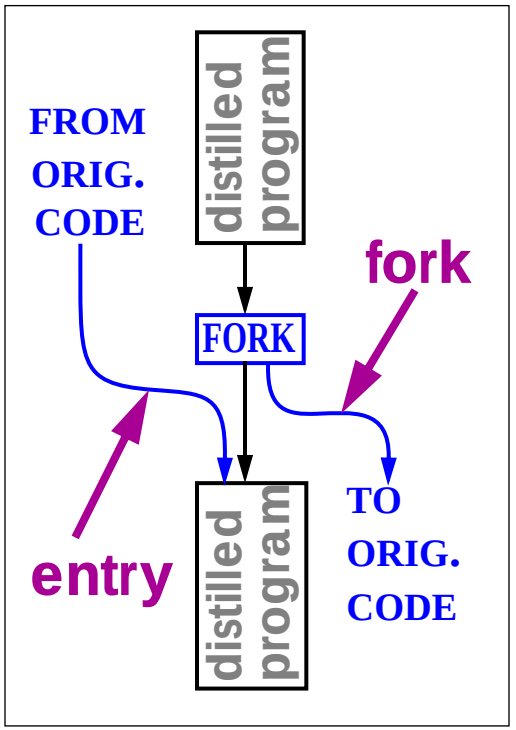
- only support transitions at defined instruction bounds
- static task boundaries
  - selection method only slightly novel, and dry (in thesis)

Encode task bounds in distilled program

- know values needed at transitions
- facilitates optimization (e.g., DCE)

Two “requirements”:

- mapping program counters
- mapping program state



Mapping Program Counters

Distilled program image is distinct from original program

- no implicit mapping between programs
- must create explicit map

Four cases: fork, entry, indirect branches and link (e.g., JAL)

Forks: (from distilled to original program)

- encode PC in distilled program (e.g., branch target)

Entries: (from original to distilled program)

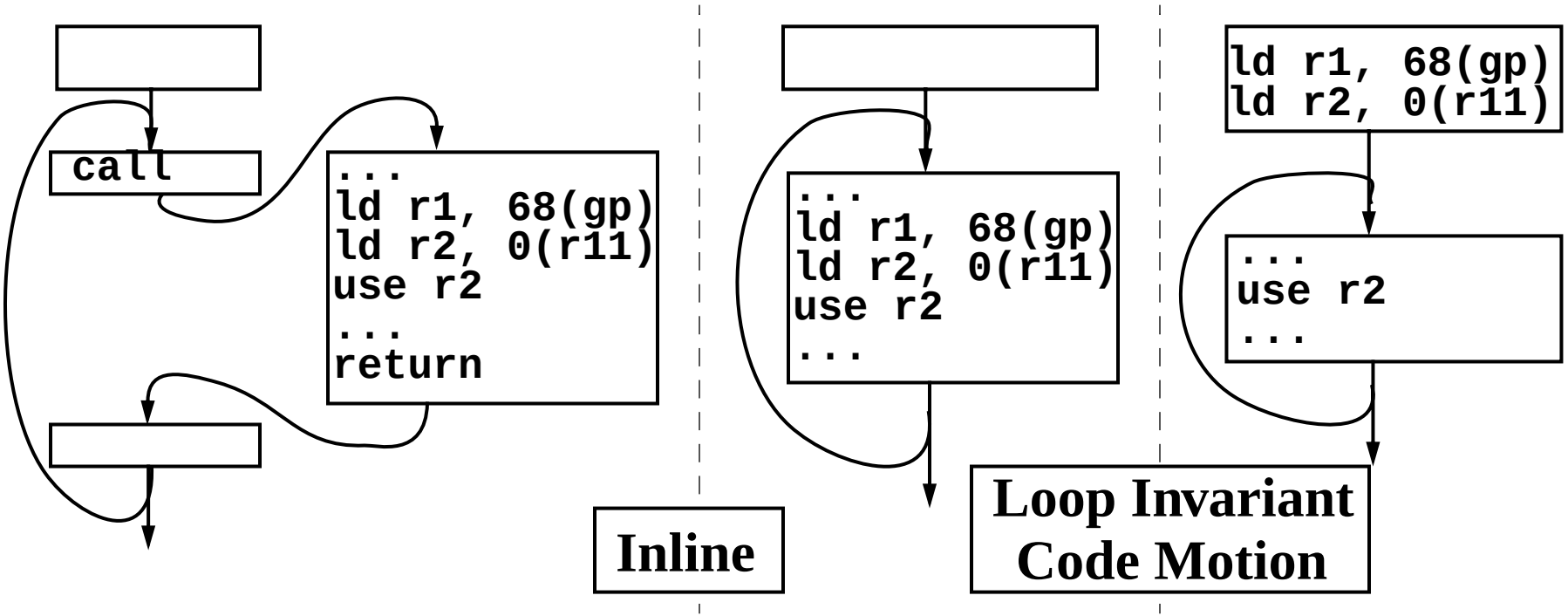
- use table lookup
  - want to avoid modifications to original program
- much like page table/TLB

Mapping Program State

Goal is to maximize performance of distilled program

Useful to re-map program state

- **avoid computing intermediates**
- **re-allocate registers (limited resource used better)**

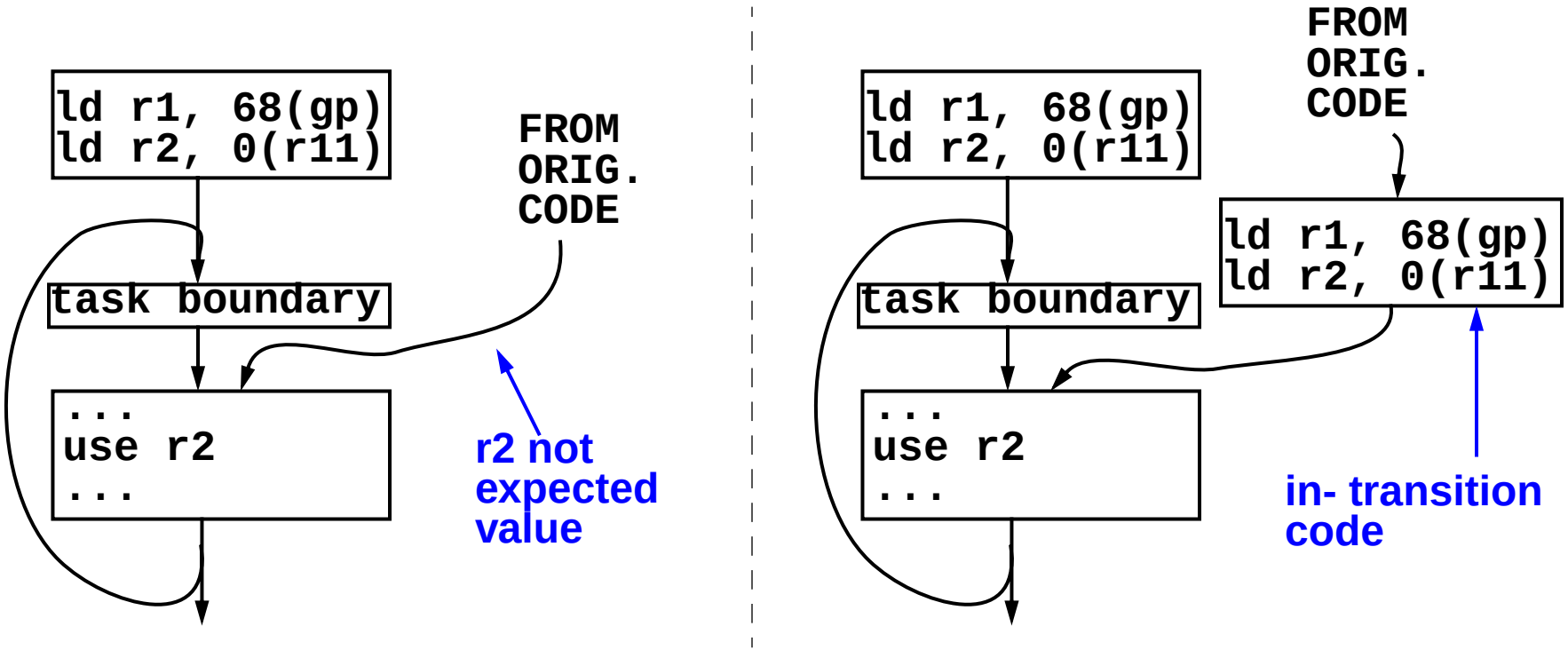


Mapping Program State, cont.

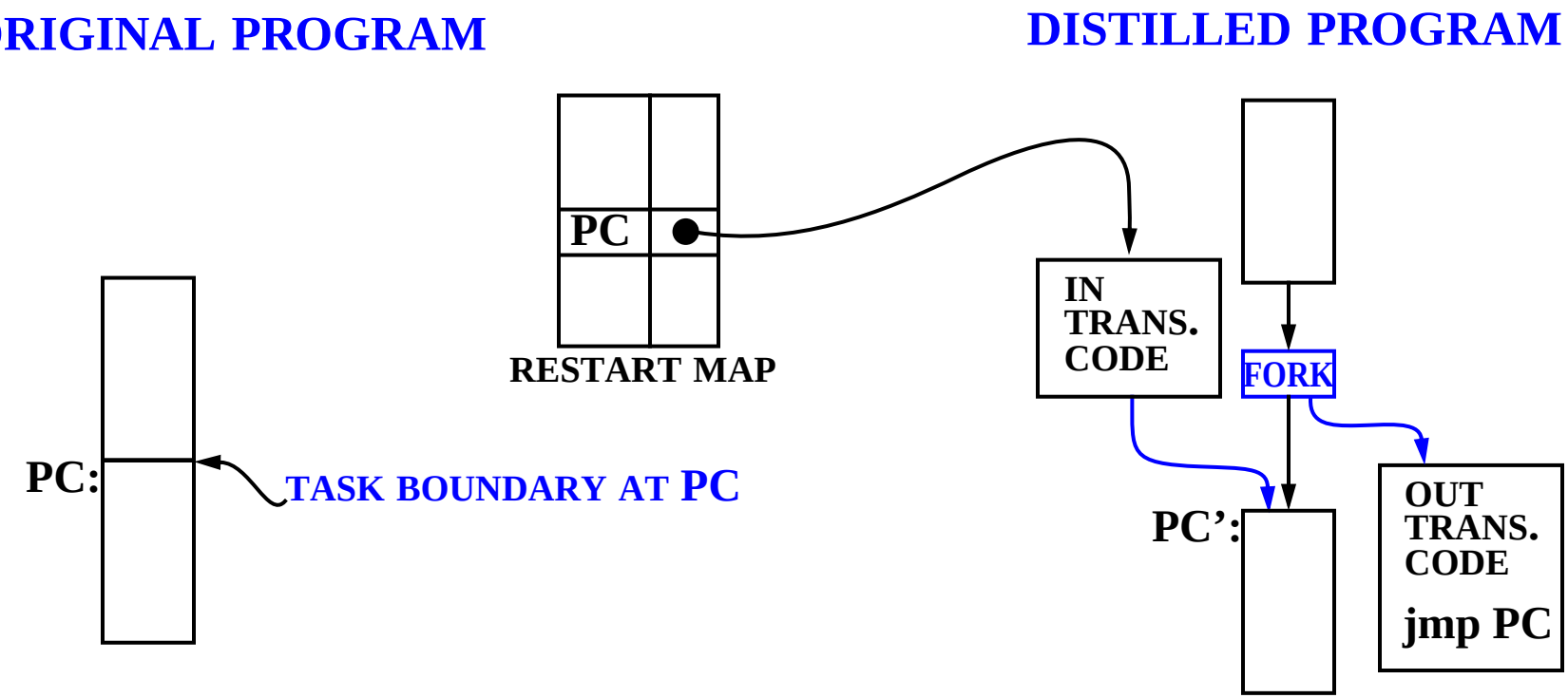
Problem: If re-mapping crosses the task boundary...
...values not in expected locations

Solution: Transition Code

- code snippet executed during transition to map state
- reminiscent of VLIW fix-up code (but can be wrong)



Distilled Program Structure



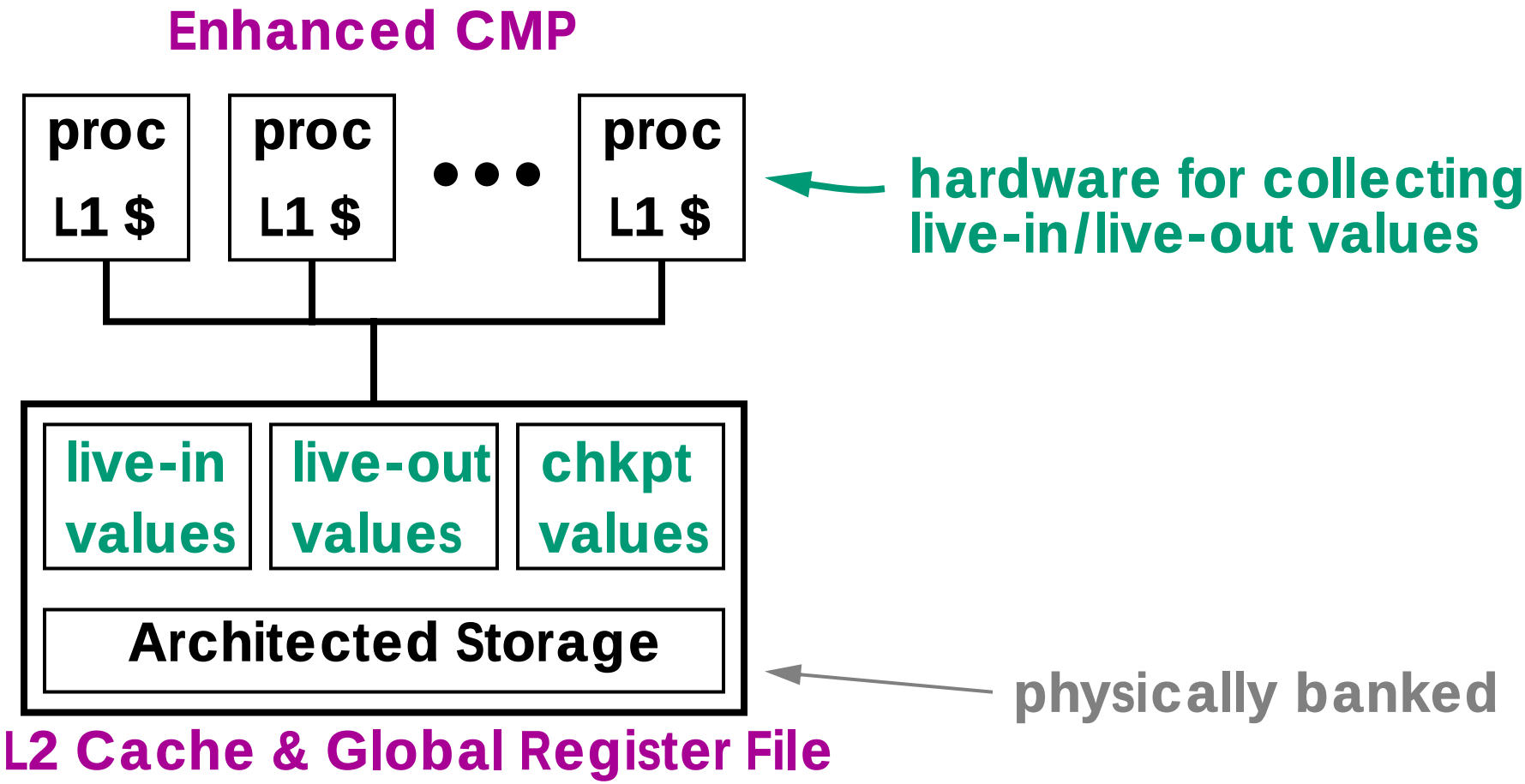
Outline

- Overview
- Motivation
- Code Approximation
- Distilled Programs
- Master/Slave Speculative Parallelization
  - hardware organization
  - forking tasks
  - assembling checkpoints
  - verification/commitment
- Evaluation
- Summary of Thesis Contributions

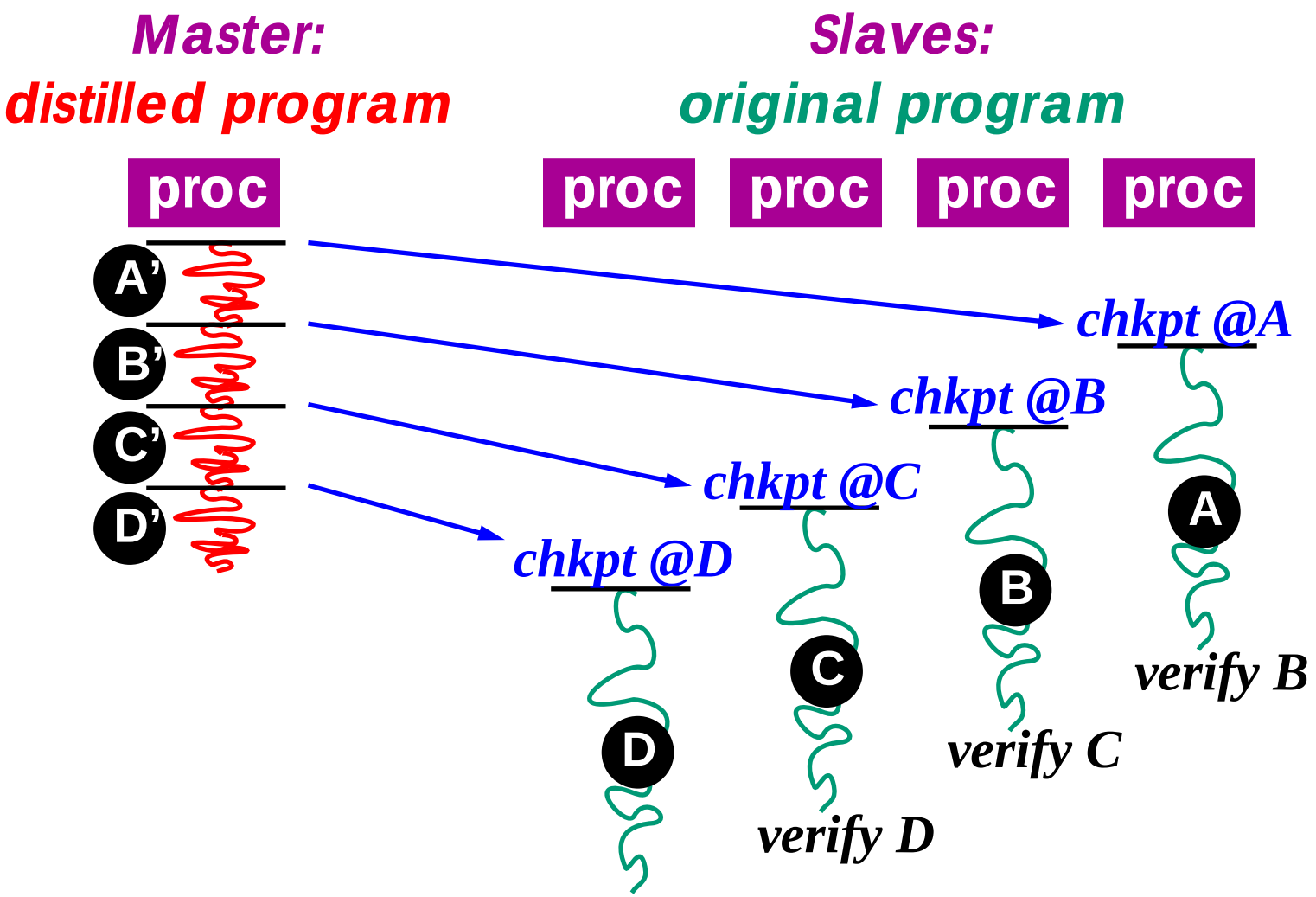
Master/Slave Speculative Parallelization

Okay, how does MSSP work?

- demonstrate using an example implementation



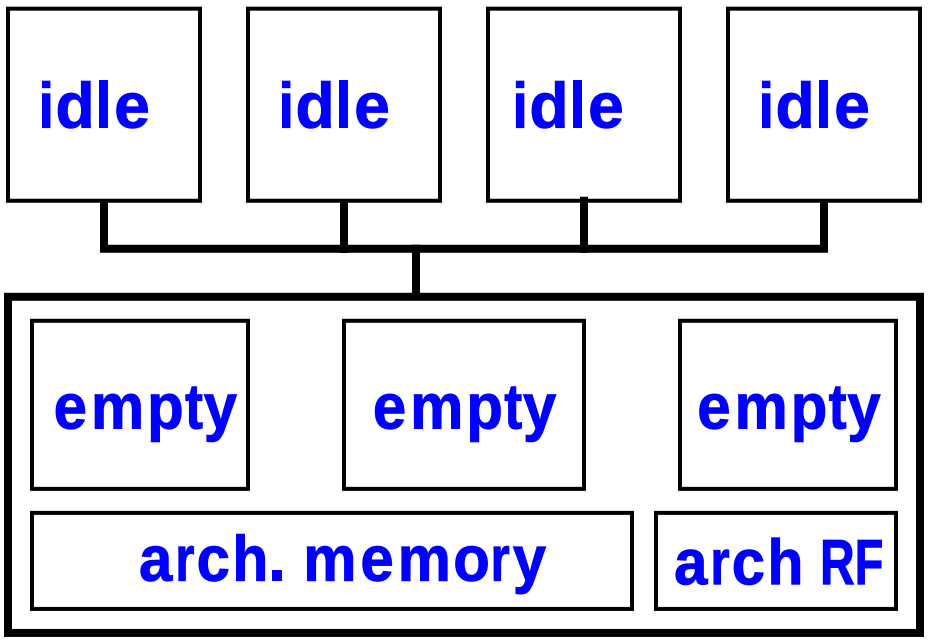
MSSP Overview



Step Zero

At beginning of program, or after task misspeculation

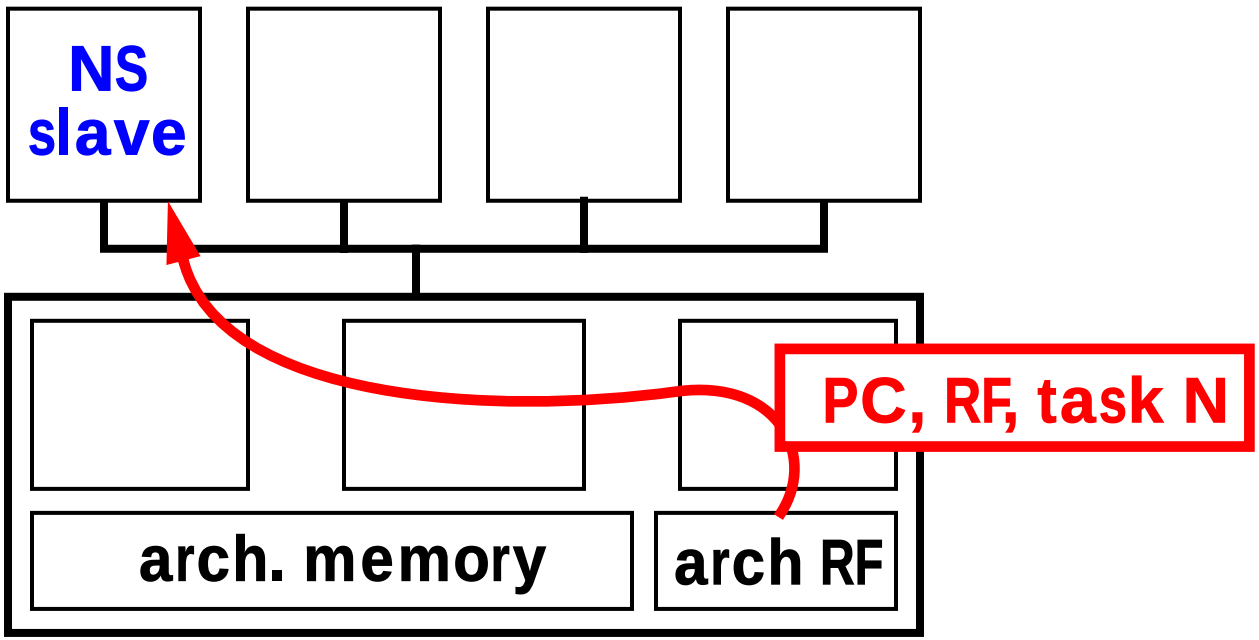
- all processors idle
- architected memory in L2 (or below)
- architected register state in Global Register File (GRF)
- all live-in, live-out, chkpt buffers empty



Step One: Restarting

Start Non-speculative Slave:

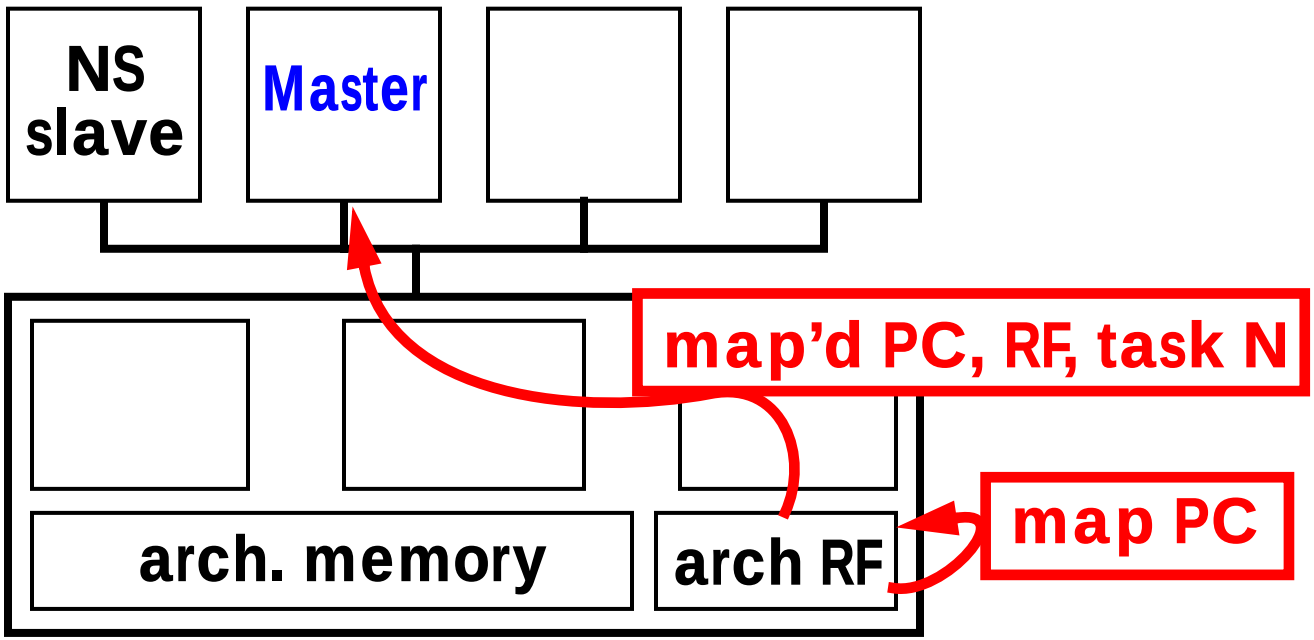
- select processor
- send PC, copy of register file, task #
- execute task non-speculatively (i.e., normal uniproc.)



Step One: Restarting, cont

Start Master:

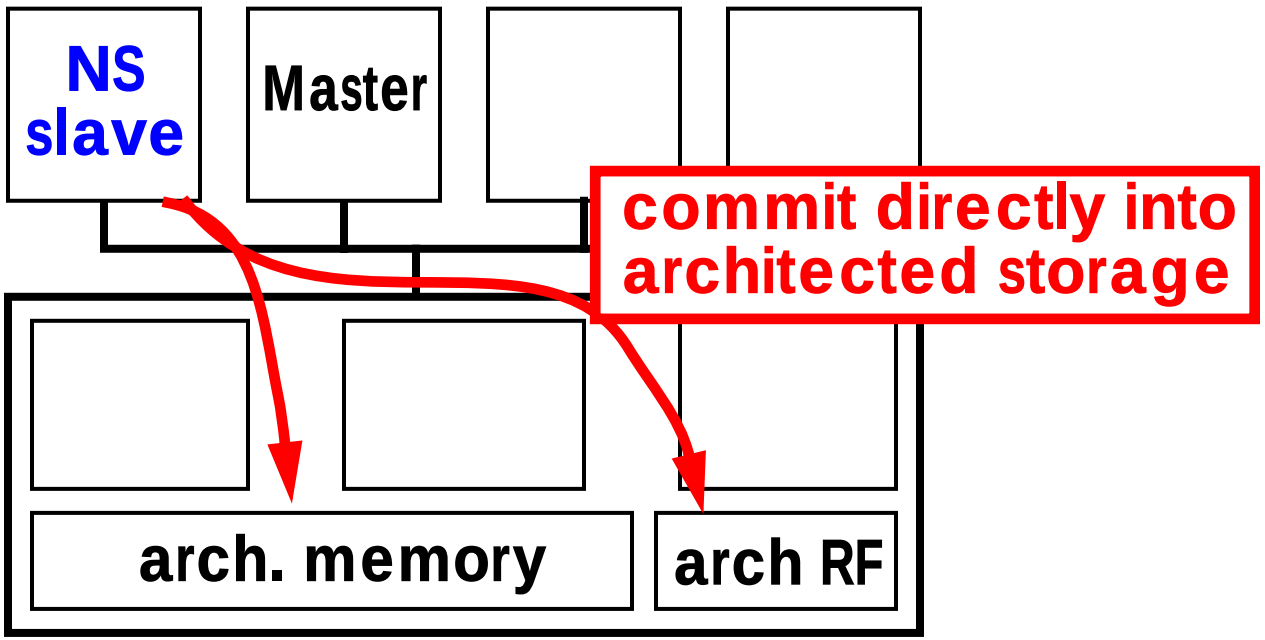
- map PC using table look-up
- select processor
- send mapped PC, copy of register file, task #
- execute in-transition code (if any)
- begin executing distilled program



Step Two: Execution

Non-speculative slave execution:

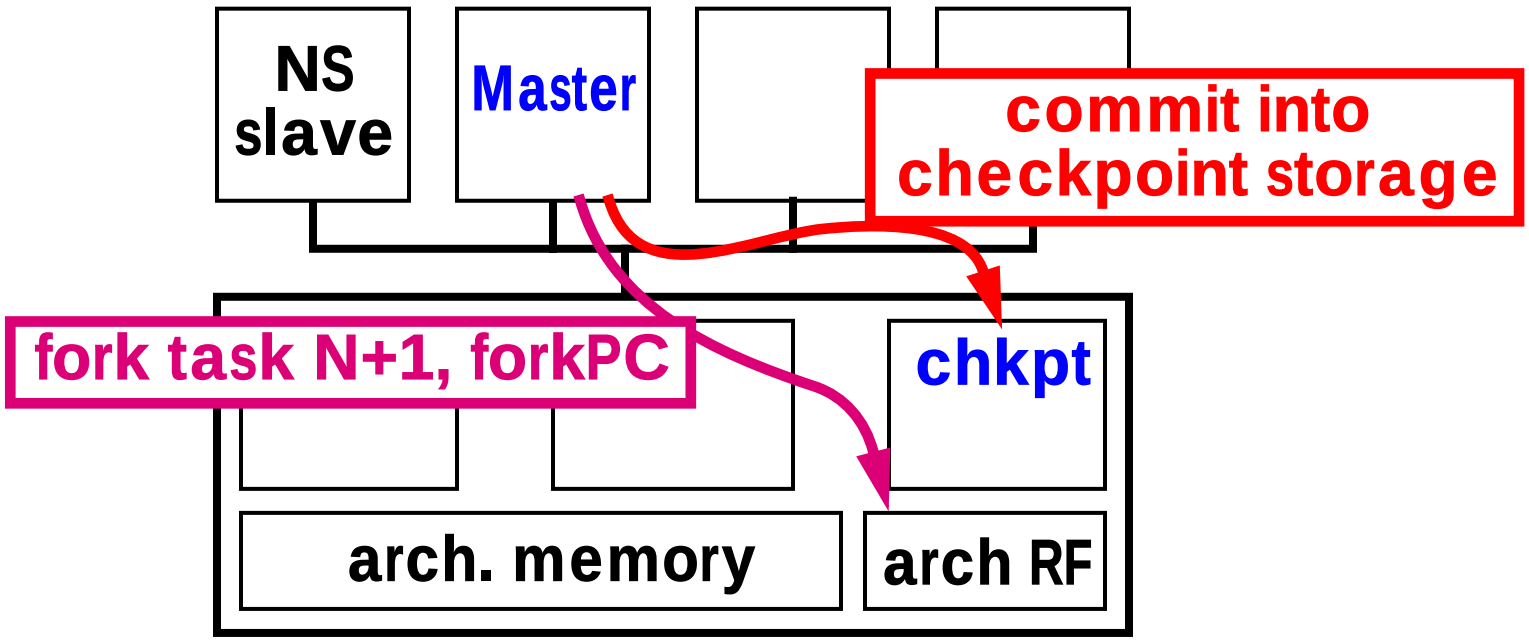
- send live-outs to L2/GRF
  - immediately update architected state



Step Two: Execution, cont.

Master execution:

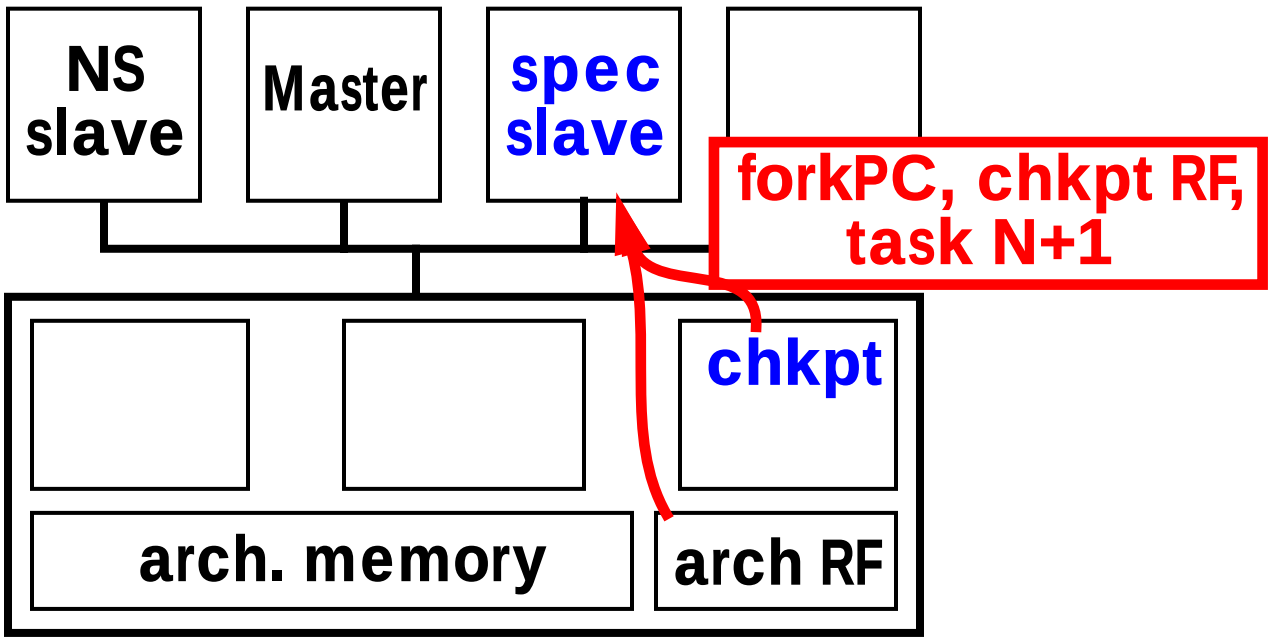
- bundle up live-outs, send to L2/GRF
  - stored as checkpoint values
- encounter a fork instruction
  - increment task number
  - send a message to GRF with fork PC



Step Three: Forking Speculative Slaves

Start Speculative Slave:

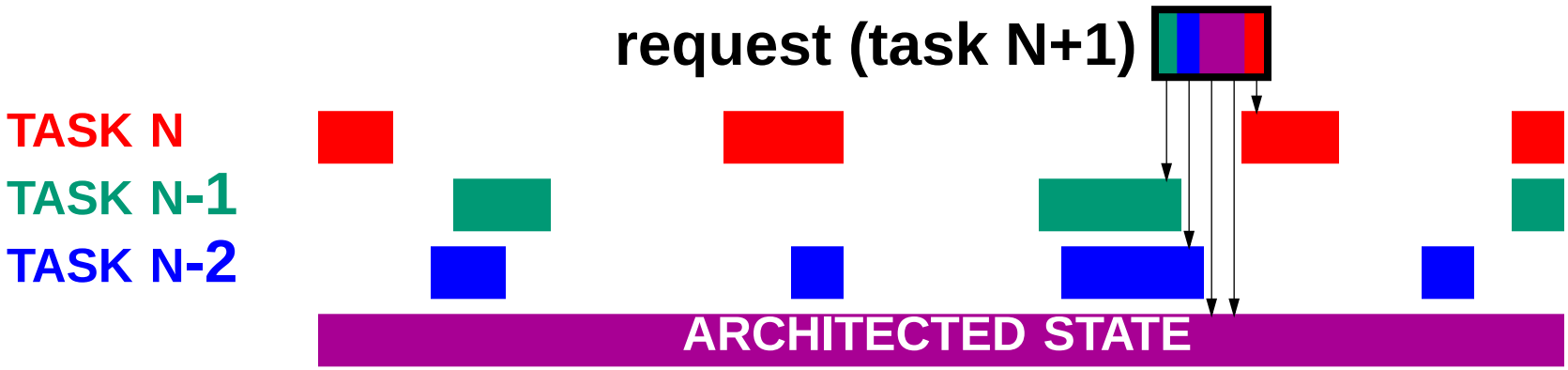
- select processor
- send fork PC, reg. file checkpoint, task #



Interlude: Assembling Checkpoints

Similar to the ARB from Multiscalar

- Checkpoint values tagged with task #



- reads get the most recent value older than request
  - architected state supplied if no chkpt value

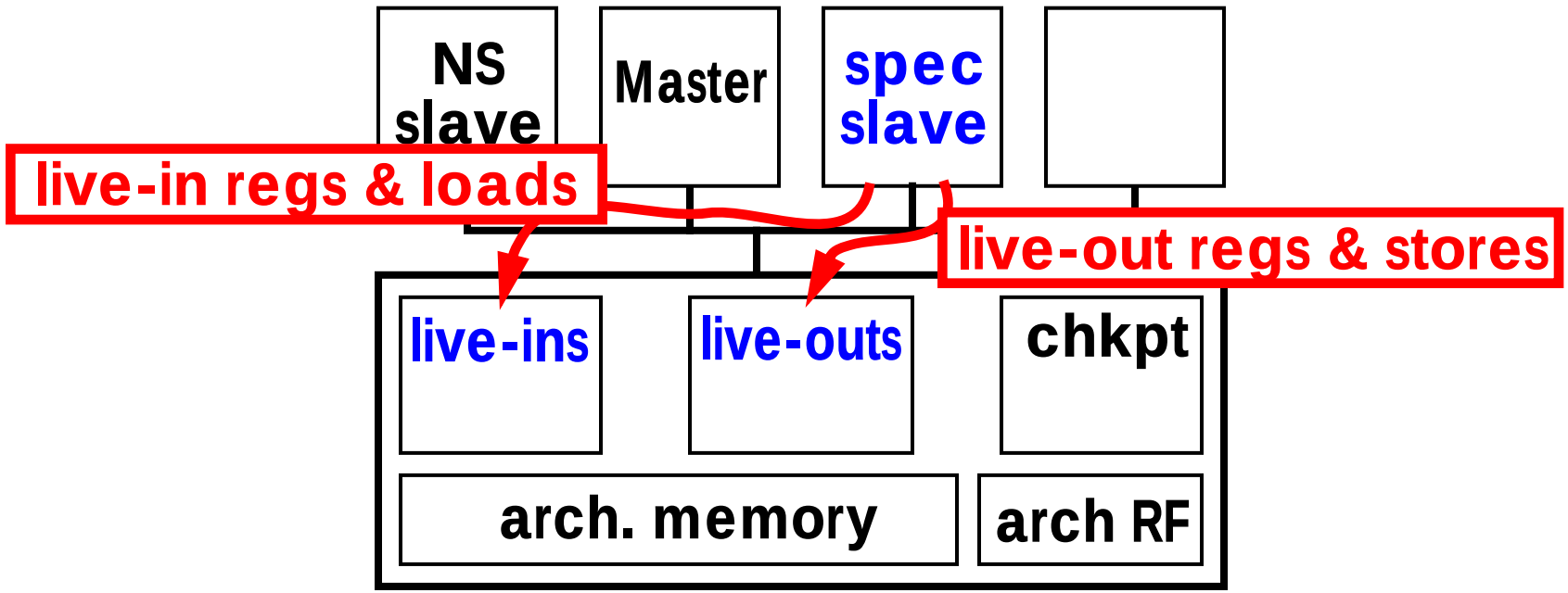
Assembly performed by the L2/GRF

- register file is assembled at beginning of task
- memory assembled a block at a time, on demand
  - cached in the local (L1) caches

Step Four: Executing Speculative Slaves

Speculative Slave Execution:

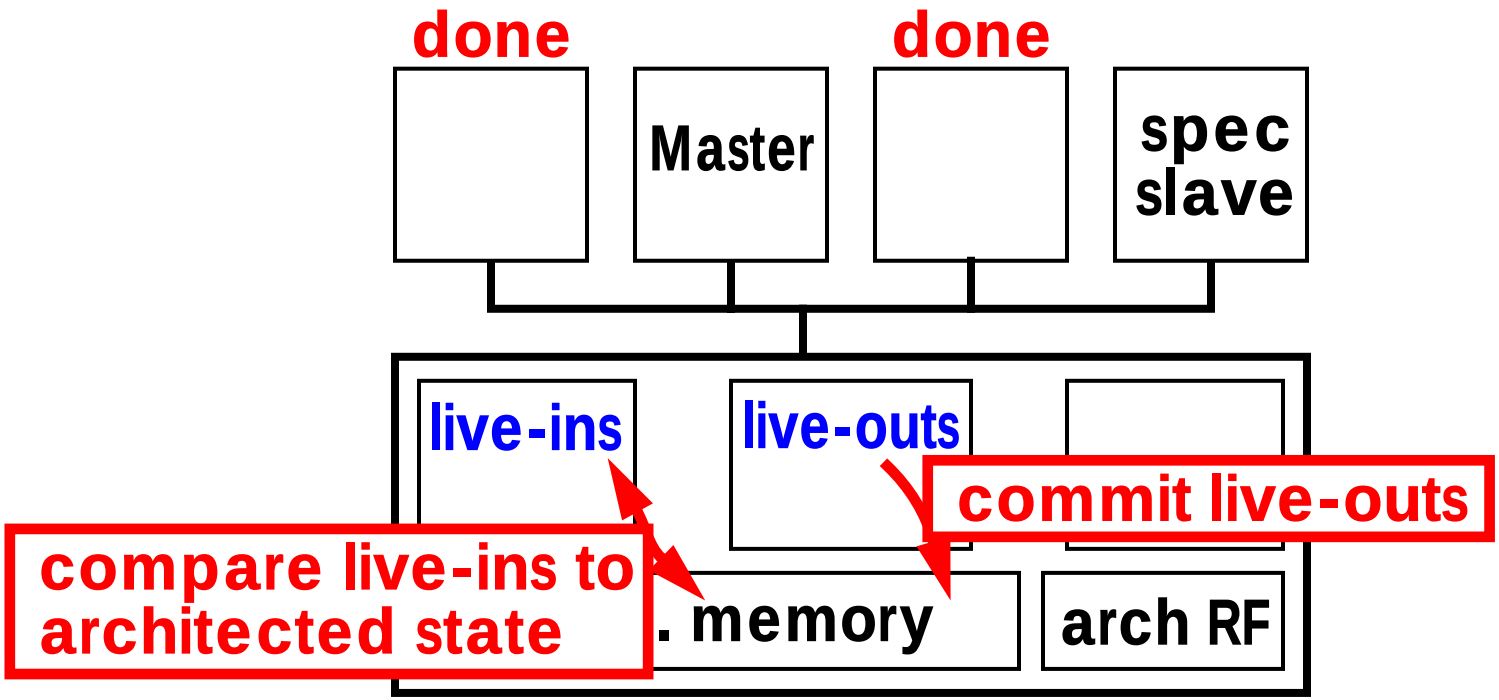
- execute out-transition code, jump to original code
- execute task
- collecting live-in and live-out values, send to L2/GRF
  - buffer as speculative live-ins and live-outs



Step Five: Verification & Commitment

Verification/Commitment:

- compare live-ins to architected state
- if matches, commit live-outs to architected state
- process should appear atomic
  - avoid memory ordering violations
- de-allocate task's live-ins and chkpt values



Step Five: Verification & Commitment, cont.

If live-ins do not match?

- task misspeculation!
- purge live-ins, live-outs, chkpts
- squash all executing tasks (master & slave)
- restart

Two notes:

- process more complicated with distributed L2
  - two-phase commit
- process can be pipelined
  - overlaps communication latency

Outline

- Overview
- Motivation
- Code Approximation
- Distilled Programs
- Master/Slave Speculative Parallelization
- Evaluation
  - goal
  - methodology
  - results
- Summary of Thesis Contributions

Evaluation

Goal:

Develop first-cut infrastructure to see big picture

Questions to be answered:

- is the architecture latency tolerant?
- what optimizations are important?
- how much hardware is required?
- can you get performance improvements?
  - i.e., should I bother continue studying it?

goal is not to exactly quantify performance

Methodology: Distiller

Distiller prototype:

- binary-to-binary “translator”
  - Alpha architecture memory images
- static, off-line for simplicity
- approximate run-time optimization
  - accurate profile info by self training

- root optimizations:
  - biased branch, null op, long dep/silent store elimination
- supporting optimizations:
  - dead code elimination, inlining, register re-allocation, save/restore elimination, simple constant folding, simple loop unrolling, code layout
  - many more possible

Methodology: Simulator

Execution-driven simulator:

- derived from SimpleScalar toolkit
- not de-coupled functional/timing
  - gives me some confidence of the results

Model: CMP of 8 Alpha 21264 processors

- 4-way OOO superscalar core (128 entry window)
- 64kB L1 2-way SA caches, 13 cycle pipeline
- 2MB L2 cache, banked 8 ways
- 10 cycle min. inter-processor communication latency
  - point-to-point: network contention modelled
- 100 cycle memory access (after L2 miss)

Results Summary

Distilled programs can be accurate:

- 1 task misprediction per 10,000 instructions (or better)

Speedup depends on distillation: varies by benchmark

- 1.25 h-mean (ranges from 1.0 to 1.75)

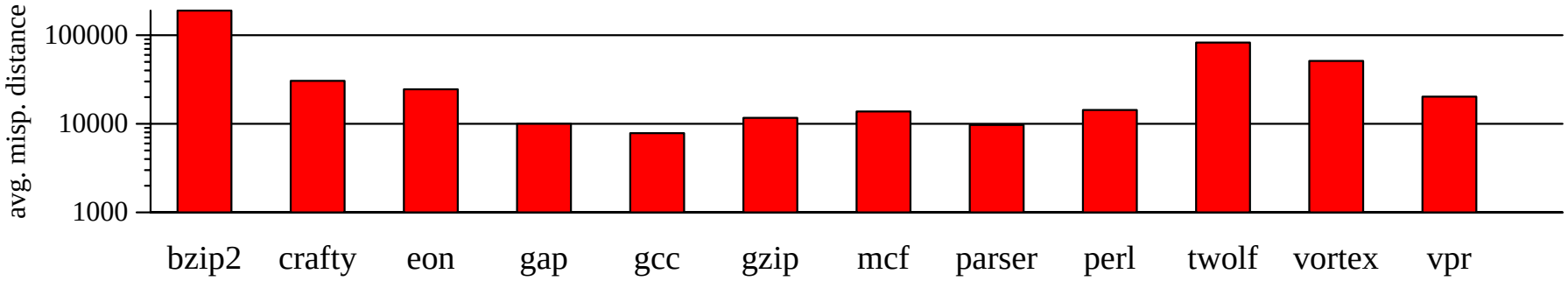
Supporting optimizations are important

- speedup: 1/3 root, 1/3 DCE, 1/3 other supporting

Latency tolerant, modest storage requirements

- 10% performance lost (comm. latency 5 $\rightarrow$ 20 cycles)
- 24kB storage at L2 for non-architectural state

Distilled Program Accuracy



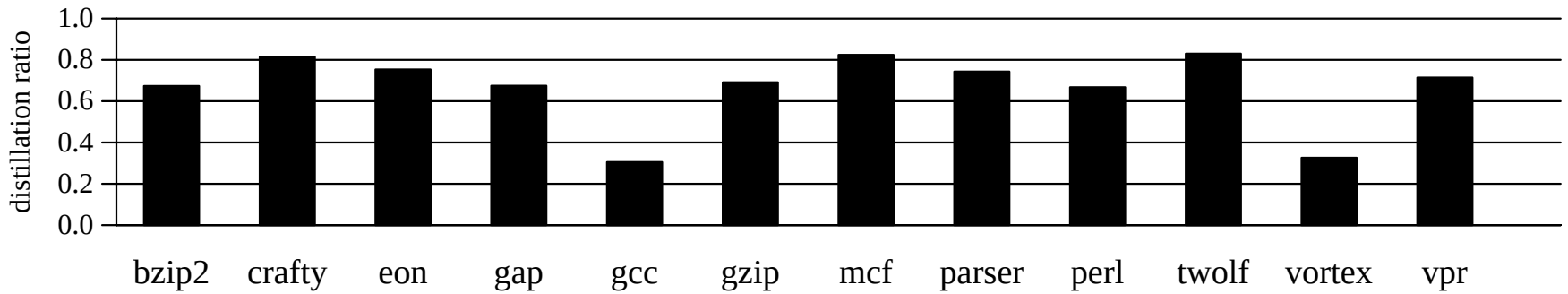
average distance between task misspeculations

distilled programs can be very accurate

Distillation ratio

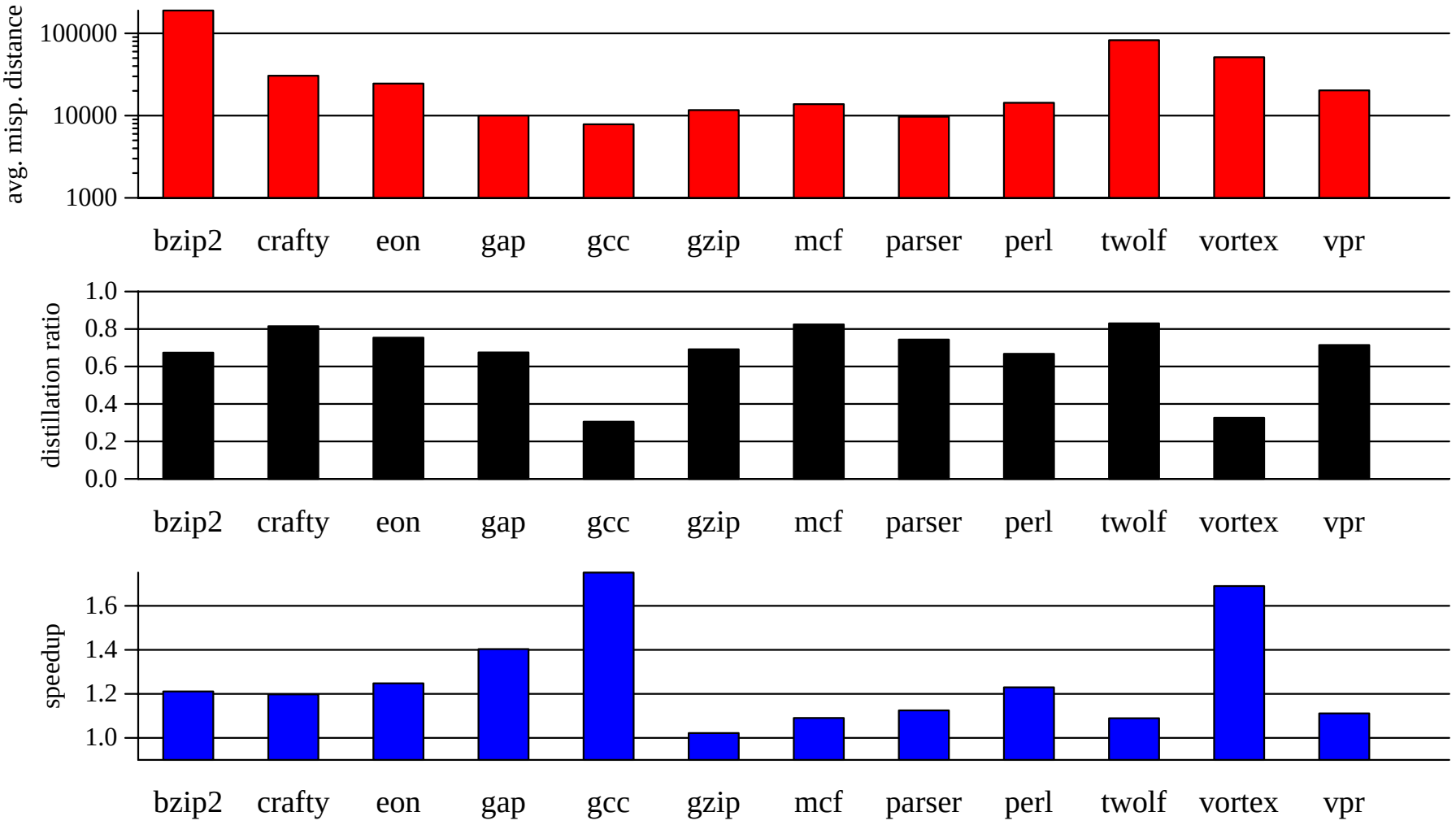
$$\frac{\text{instructions executed by master (dist. program)}}{\text{instructions executed by slave (orig. program)}}$$

(not counting nops)



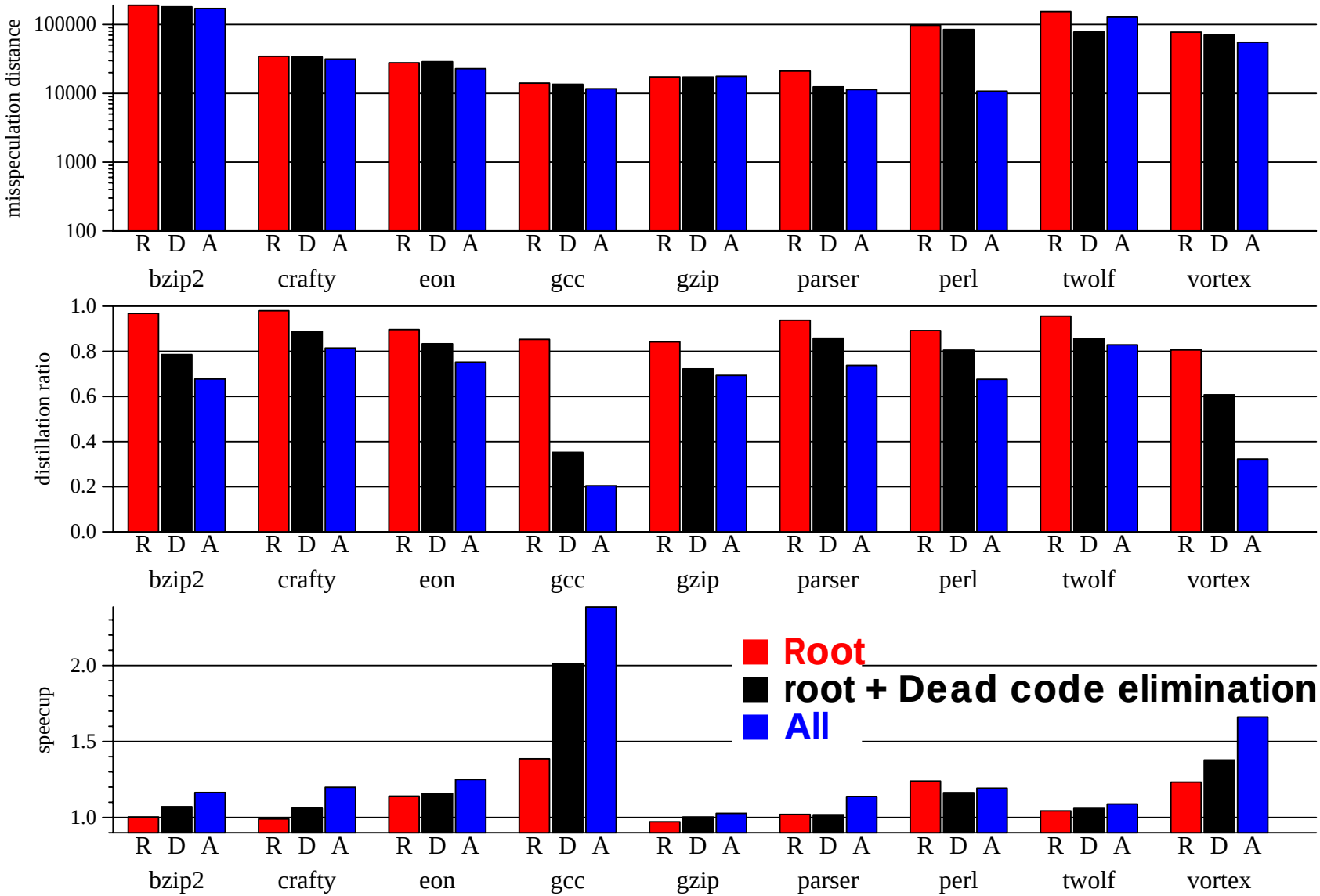
large variance between benchmarks

Performance



performance benefits scale with distillation ratio

Importance of Supporting Optimizations

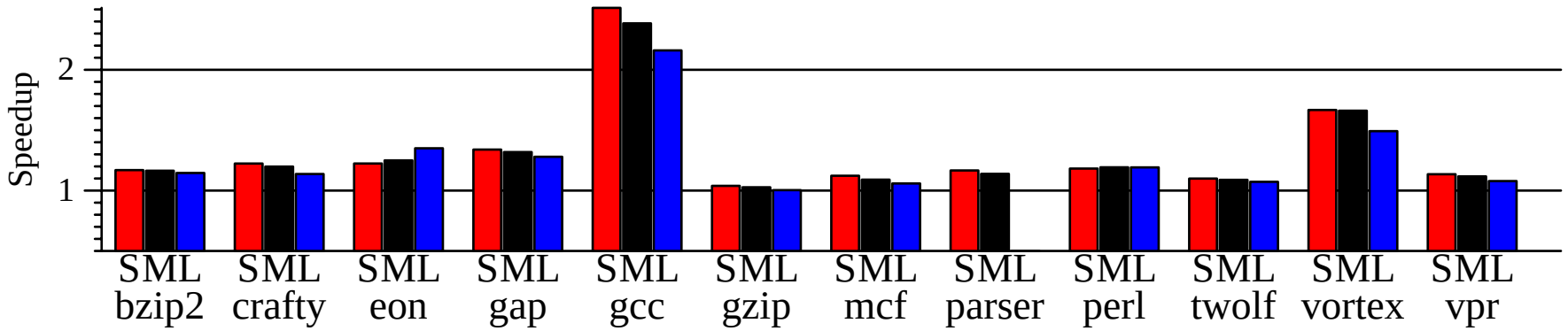


supporting opts. provide much of the speedup

Inter-processor Communication Latency Sensitivity

Vary the communication latency:

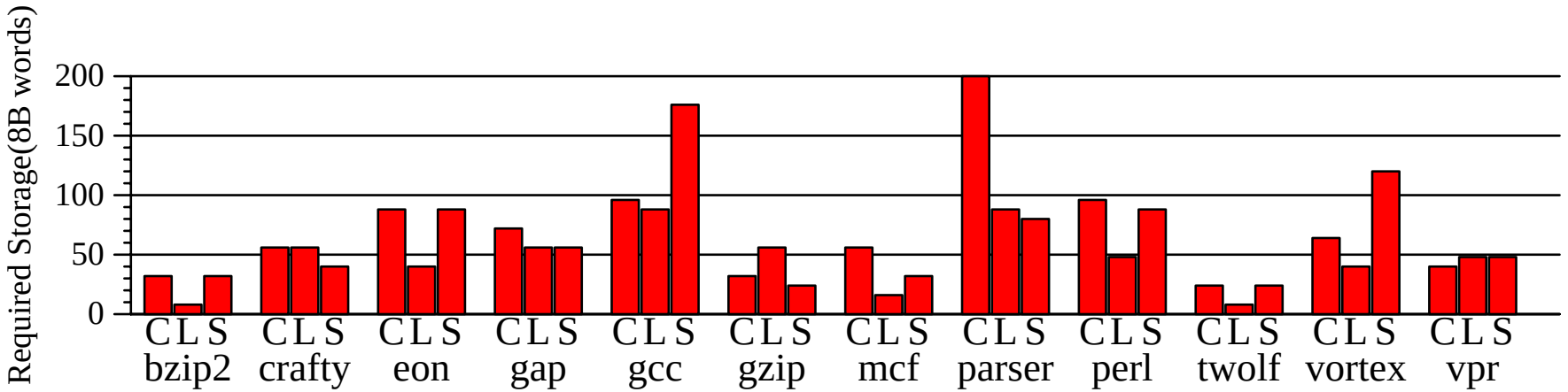
- **S: 5 cycles, M: 10 cycles, L: 20 cycles**



Most benchmarks ~10% slower w/4x latency

Largely insensitive to communication latency

Hardware Storage Requirements



Track non-architectural storage used

- C: checkpoint, L: memory live-ins, S: live-out stores

24kB storage (total at L2) seems sufficient

- About 1% of the bits of a 2MB L2 cache

Modest amount of speculation buffering

Conclusion

Architecture displays desired characteristics:

- latency tolerance
- modest hardware requirements
- support for legacy binaries
- distilled program need not be verified

Performance could be better

- accuracy is good
- distillation will improve with additional effort

Promising avenue for continued research