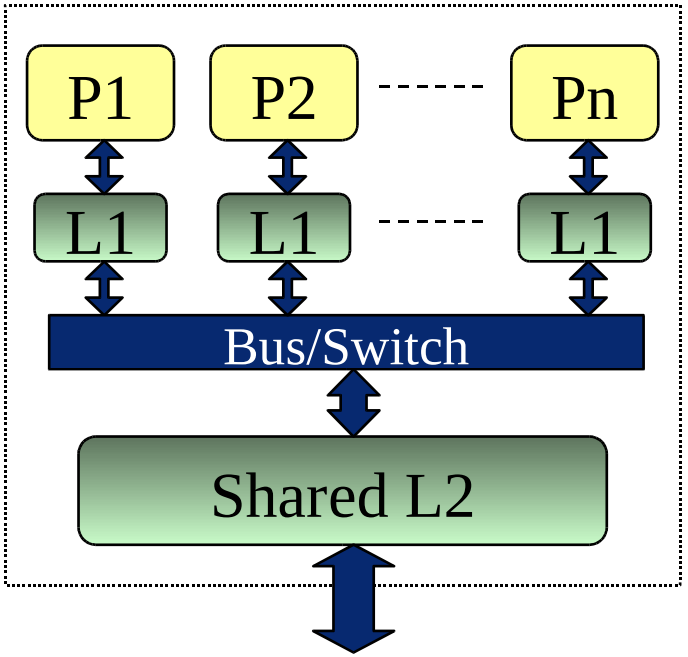
Reducing Misspeculation Overhead for Module-Level Speculative Execution

Fredrik Warg and Per Stenstrom

Department of Computer Science and Engineering
Chalmers University of Technology

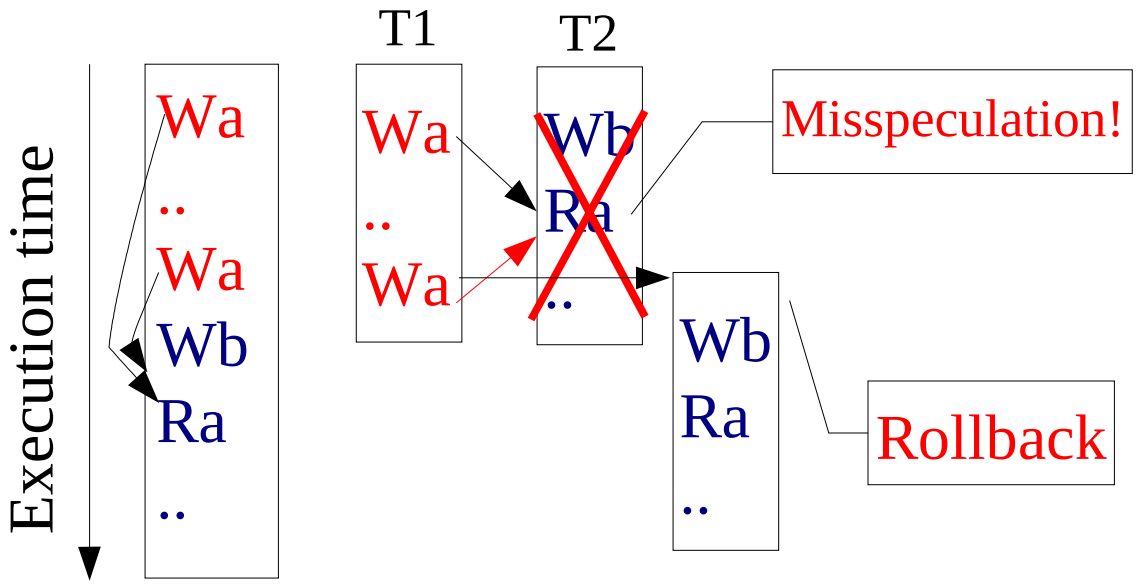
Motivation

- CMP/SMT designs increasingly popular
- Some applications hard to parallelize + large base of existing single-threaded programs



Thread-level speculation: Agressive parallelization of single-threaded applications for multithreaded processors

Thread-level speculation



Overhead: Thread-start, roll-back (restart), extra execution, extra communication

Problem: Significant overhead (>300% in our experiments) means inefficient energy and resource usage

Contributions

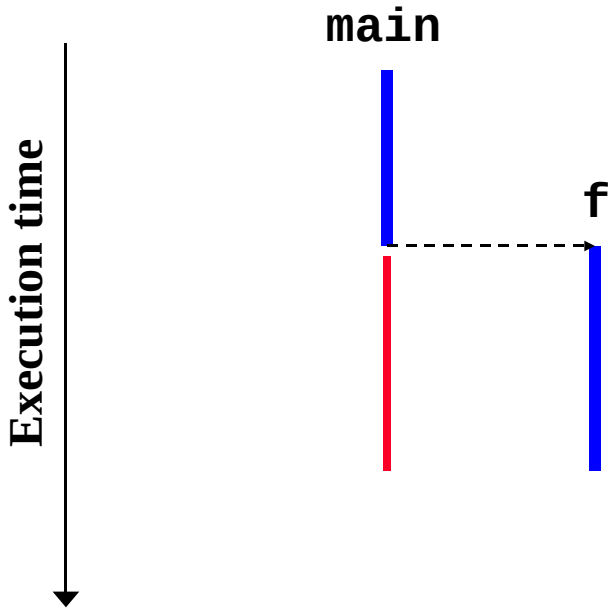
- **Misspeculation prediction** dynamically disables speculation where threads have previously caused misspeculations
  - Removes most misspeculations
  - Improves energy and resource efficiency
- Dynamically detect when misspeculation prediction technique is needed

Overhead reduced six-fold compared to indiscriminate speculation using relatively simple dynamic scheme

Outline

- Background
  - Module/procedure-level parallelism
- The overhead problem
  - Methodology and overhead measurement results
- Misspeculation prediction
  - Basic idea and design space
  - Results
  - On-demand misspeculation prediction
- Conclusions

Module-Level Parallelism



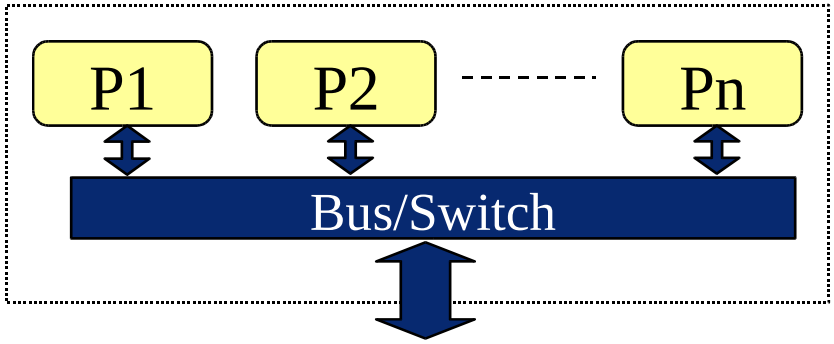

```
main () {
 …
 a=f()
 …
 …
 …
}
```

```
int f() {
 …
 return b
}
```

New threads are spawned for **module continuations.** Return values are predicted

Methodology

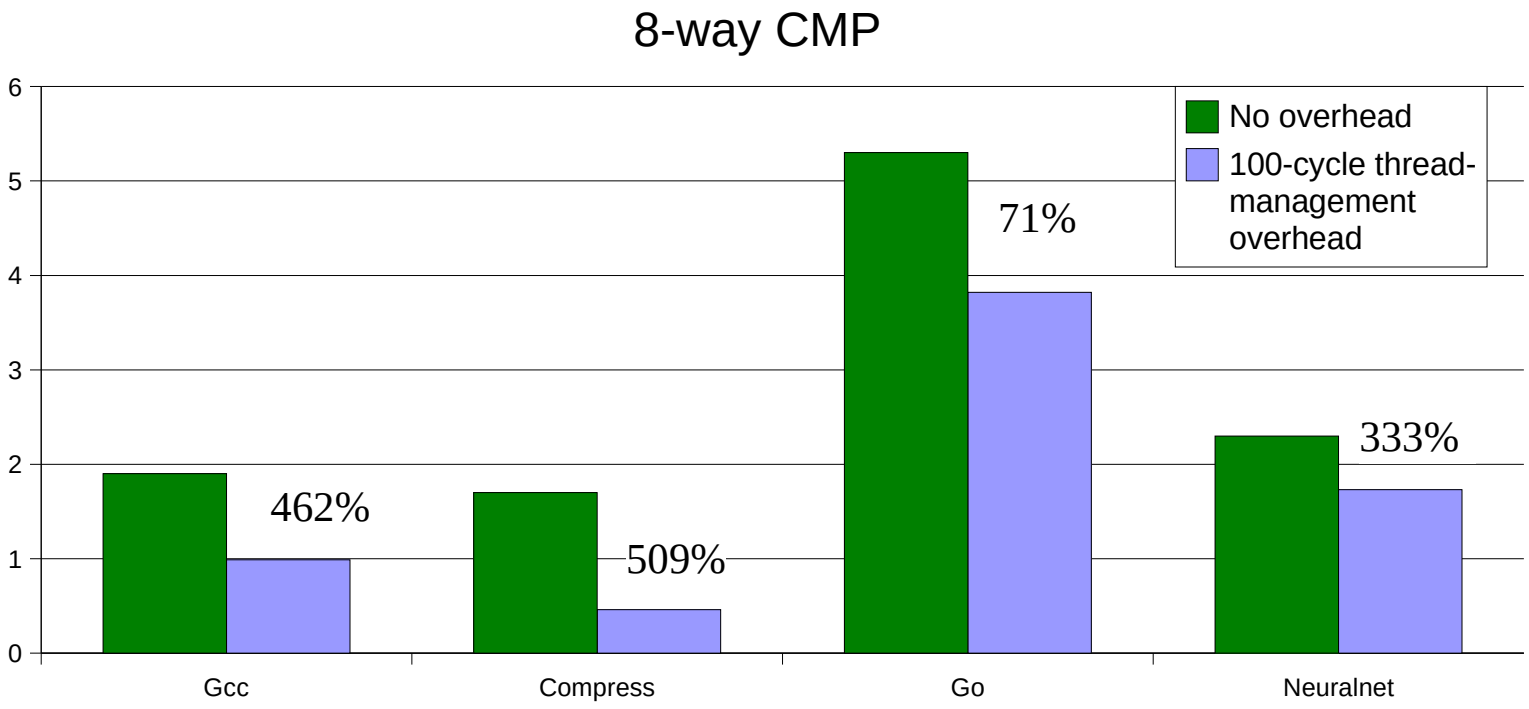
- Annotated sequential trace from full-system simulator
- TLS in custom trace-driven simulator
- 8-way CMP
  - Single-issue cores
  - Ideal memory system
  - Fixed-length thread-mgmt overheads (100 cycle)



CMP enhanced to:

- Detect dependence violations
- Buffer speculative values
- Handle roll-backs

Speedup with Overheads



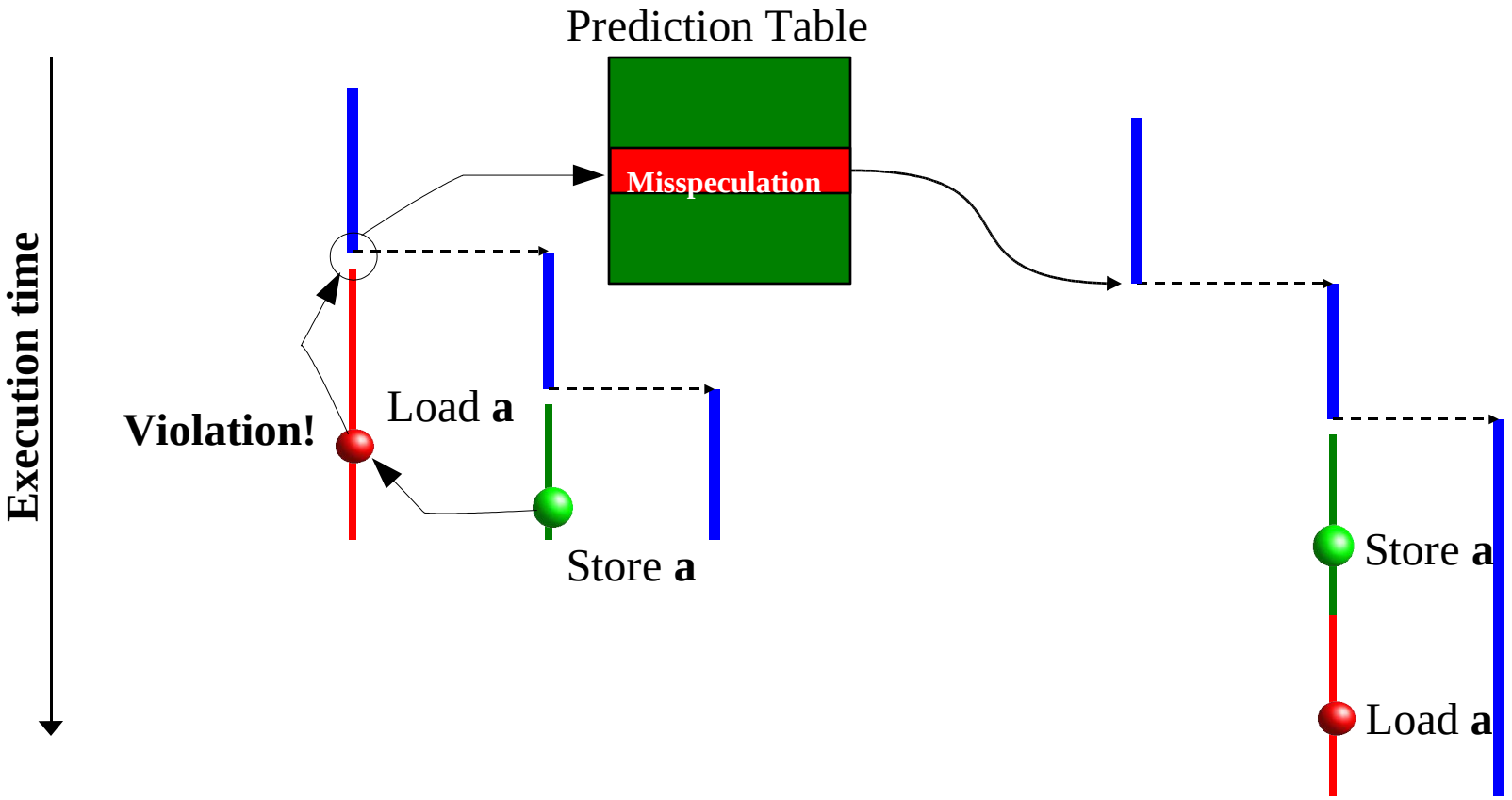
Overhead leads to large inefficiencies, and can severely reduce speedups

Misspeculation Prediction

Problem: Misspeculations are the source of much of the overhead
Solution: Do not start threads expected to cause misspeculation

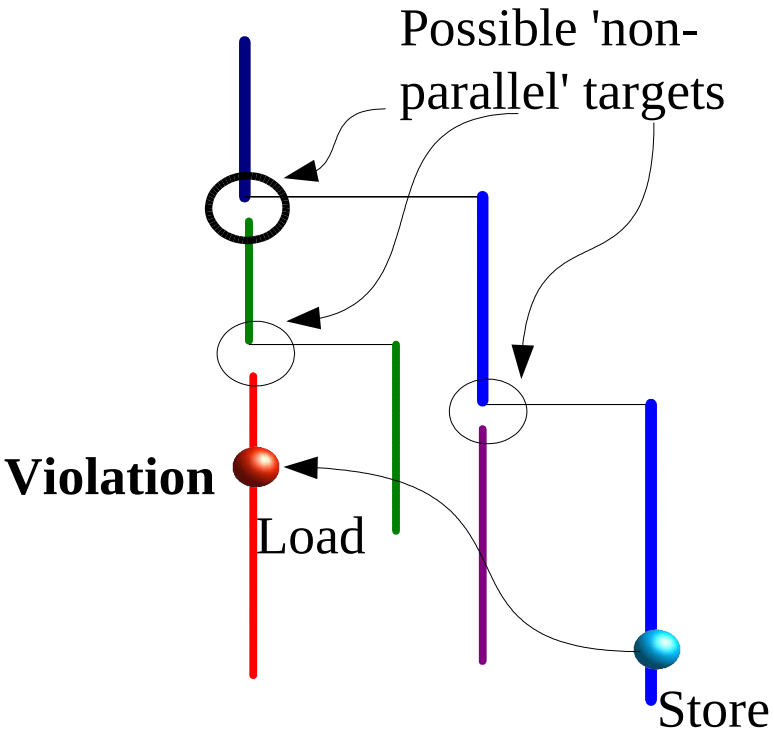
- If creating a thread for a module continuation caused a misspeculation, it is likely to happen again
  - Record misspeculations in a prediction table
  - Prevent thread creation in situations where threads have previously caused misspeculations

Misspeculation Prediction Example



Design Space

Indentifying which calls to classify as 'non-parallel'

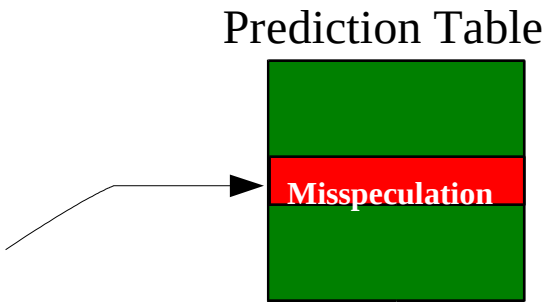


”Same situation” (prediction table index)

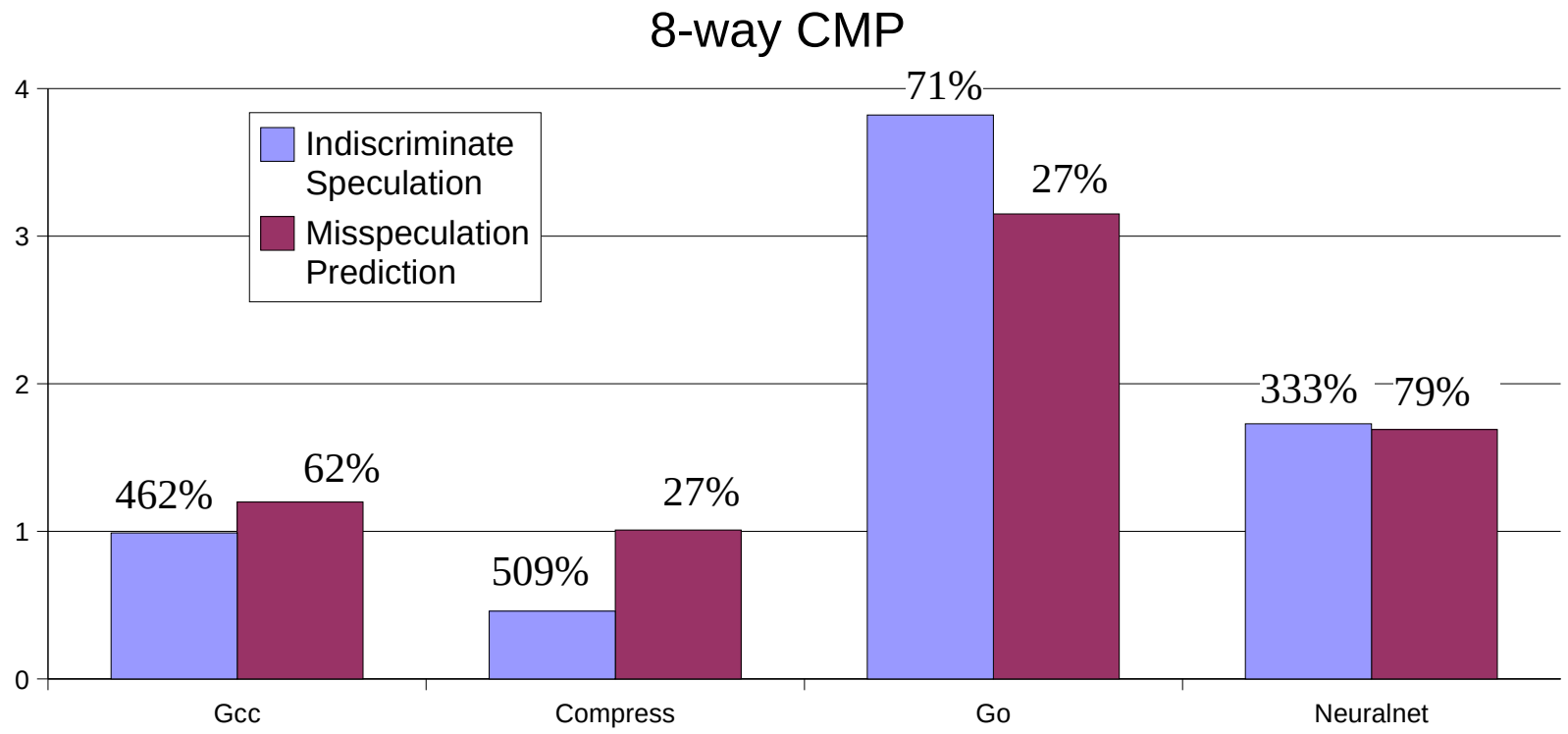
- Call-instruction, **call destination**

Choice of predictor

- **Last-outcome**, 2-bit counter, prediction timeout



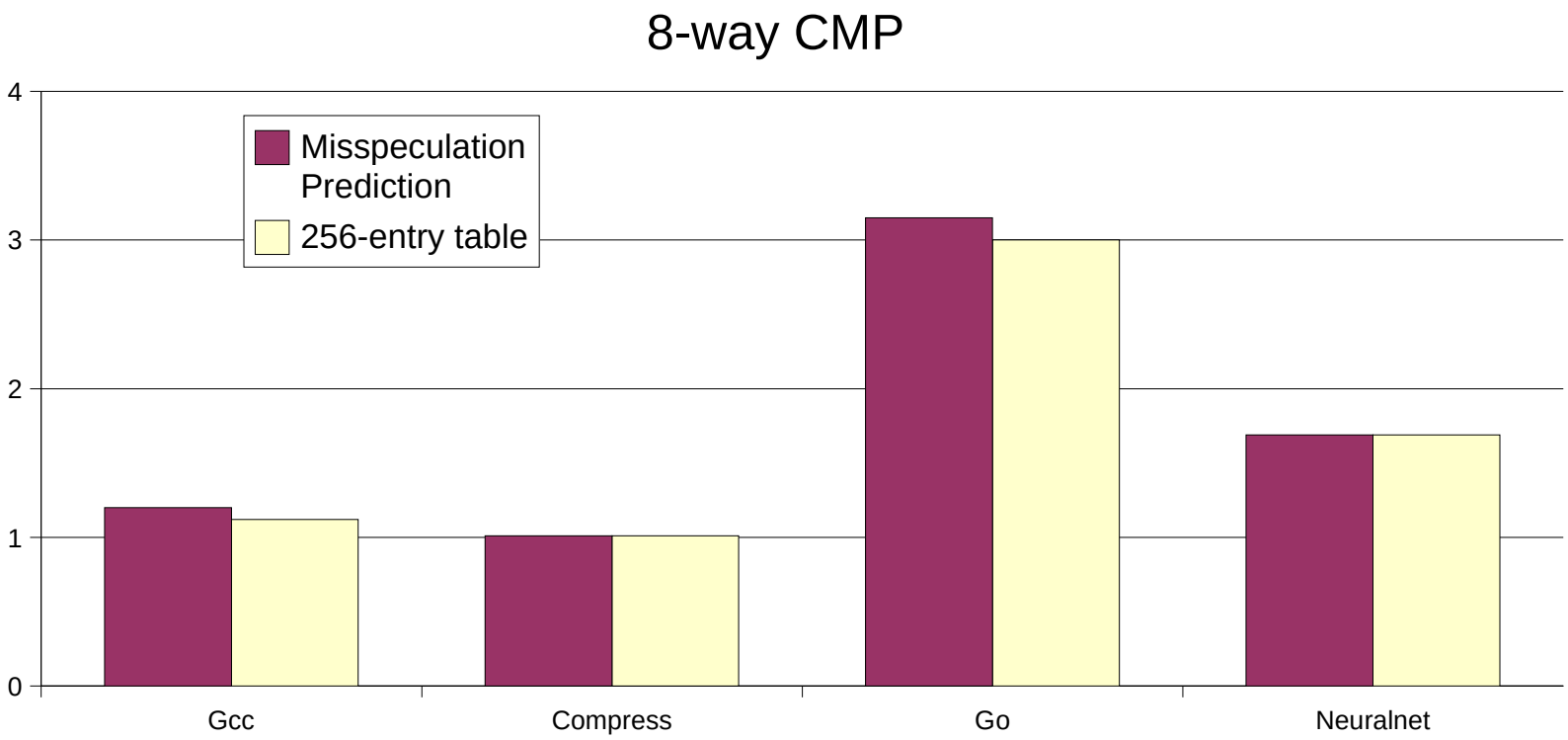
Results: Misspeculation Prediction



Overhead reduced from avg. >300% to avg. ~50%

Speedup improved or unaffected for most applications

Results: Limited Prediction Table



256 bits of of information storage is sufficient

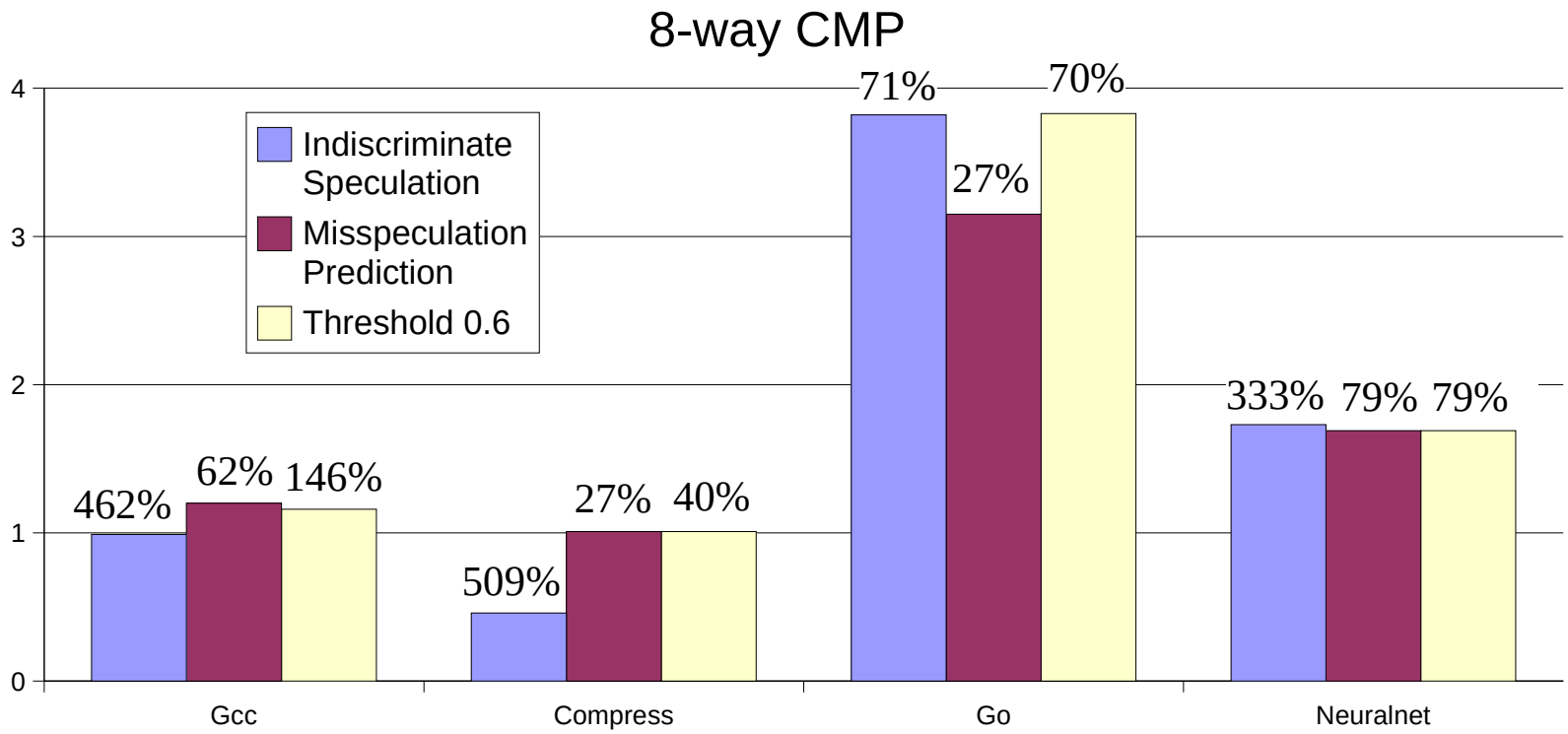
for the prediction table

Enabling Misspeculation Prediction On-demand

Problem: Speedup suffers for som apps w/ misspec prediction
Solution: Only enable it when squashing is too frequent

- Two global counters
  - Roll-back (restart) counter
  - Thread-start/restart counter
- Enable misspeculation prediction when restarts to threadstarts+restarts ratio exceeds some threshold

Results: Misspec Pred w/ threshold



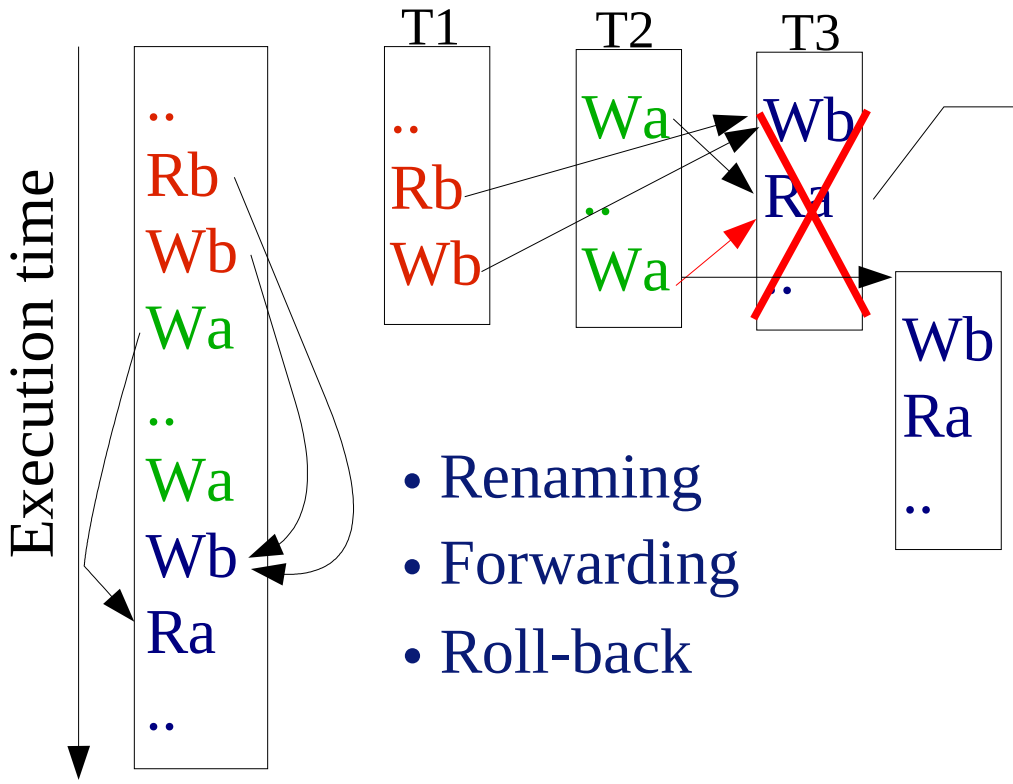
Overhead up from avg. ~50% to ~90%,

but no speedup penalty (ratio = 0.6)

Conclusions

- Overheads can become very large
  - We need to adress overhead, not just speedup
  - Misspeculation reduction techniques needed
- Our misspeculation prediction technique:
  - Requires no changes in the applications
  - Reduces average overhead up to six-fold compared to indiscriminate MLP
  - Small impact on speedup
  - Requires very little storage to implement

Extra: Thread-level speculation



- Renaming
- Forwarding
- Roll-back

Overhead:

- Thread-start
- Roll-back
- Extra execution
- Extra communication