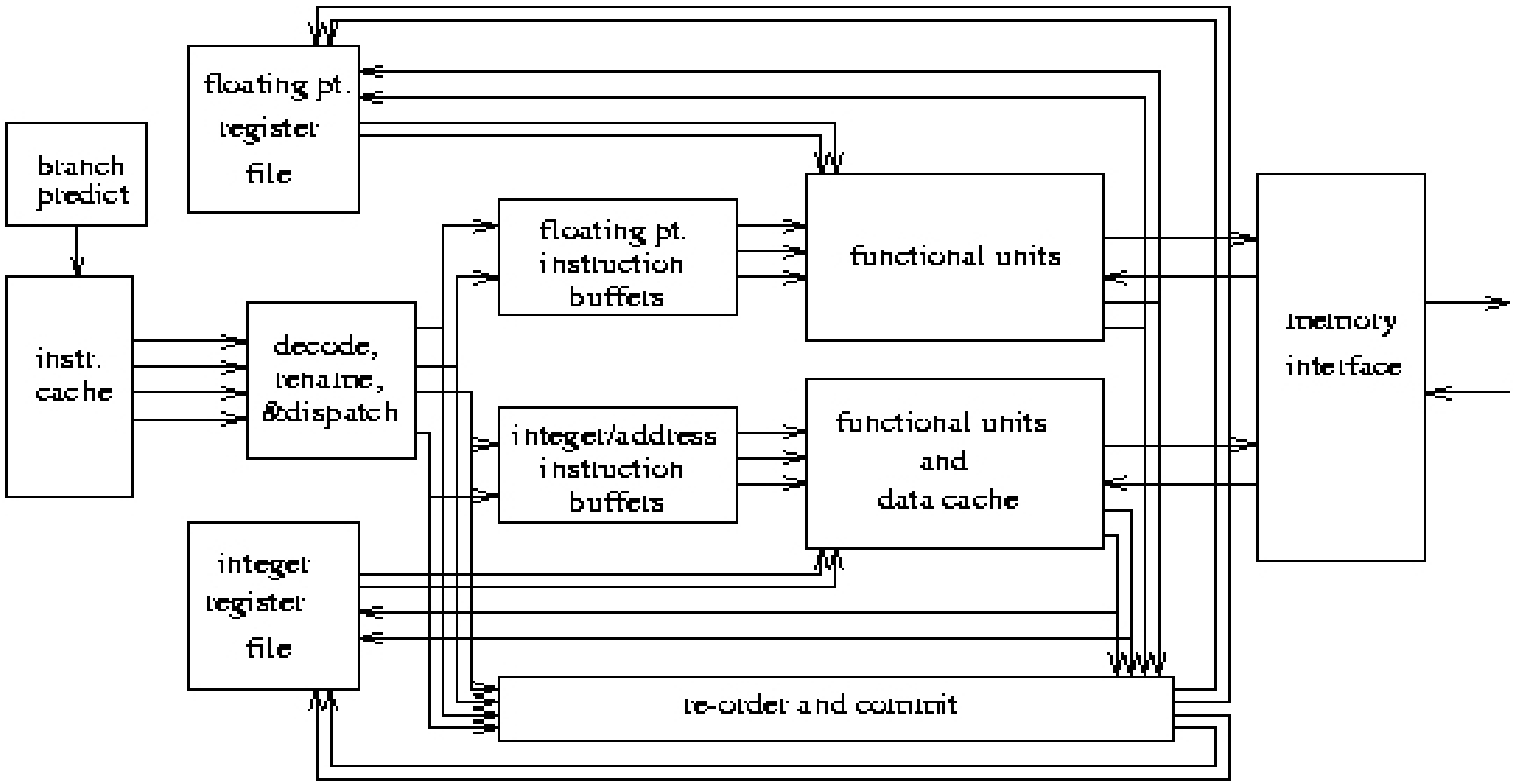
Modeling Superscalar Processors via Statistical Simulation

Daniel J. Sorin, Richard Carl, James E. Smith, David A. Wood

Depts of Comp. Sci. and Elec. and Comp. Engr.
University of Wisconsin-Madison

Goal

- Goal: *quickly* evaluate designs with complex superscalar processors

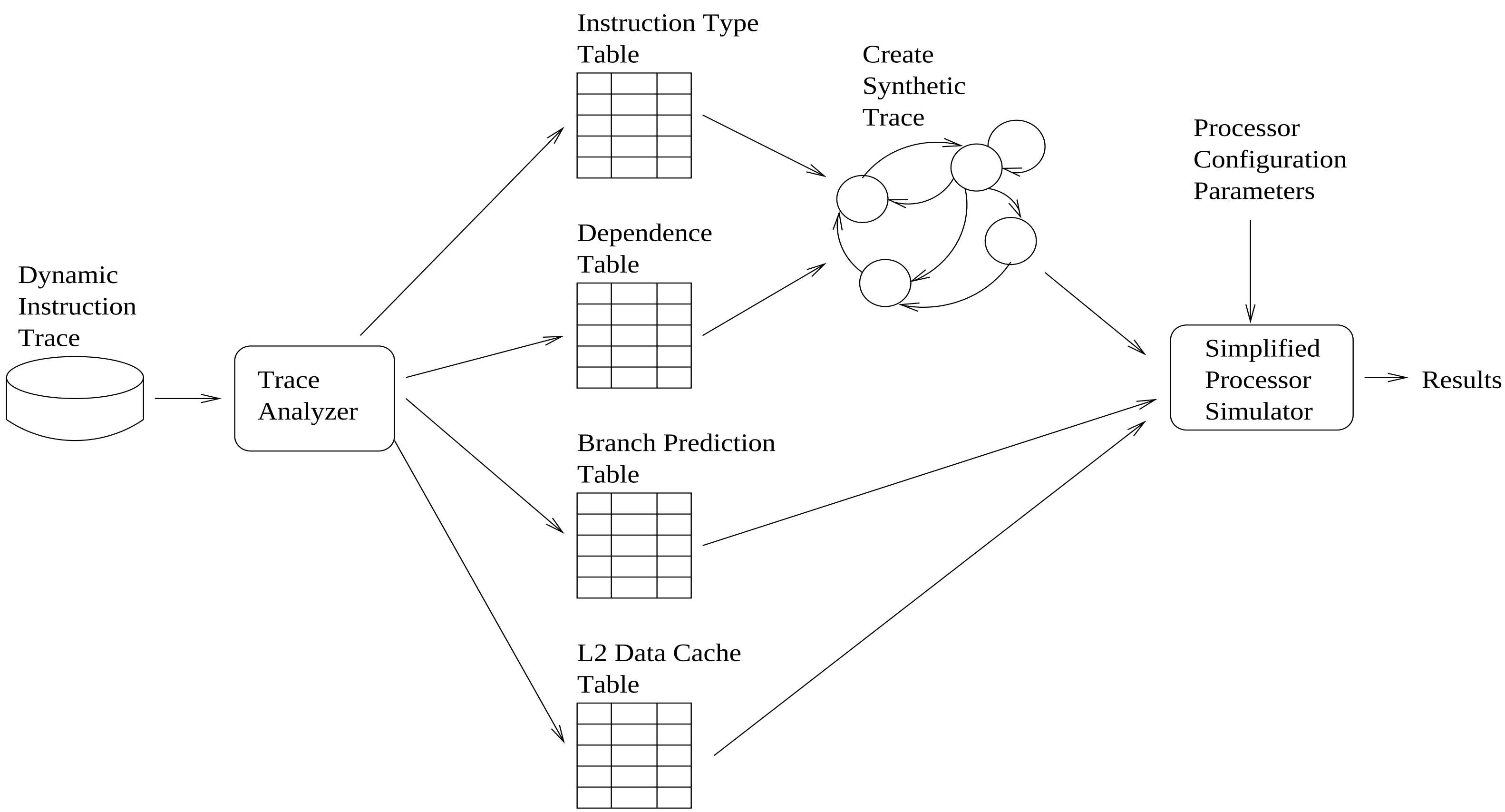


Existing Techniques for Evaluating Designs

- Detailed simulation
    - e.g., SimpleScalar, SimOS, RSIM
    - \- Slow! Takes hours to simulate seconds
    - Speedup techniques
        - direct execution
        - sampled simulation
        - accuracy vs. speed tradeoffs
- Analytical Modeling
    - \+ very fast
    - \- difficult to model superscalar processor

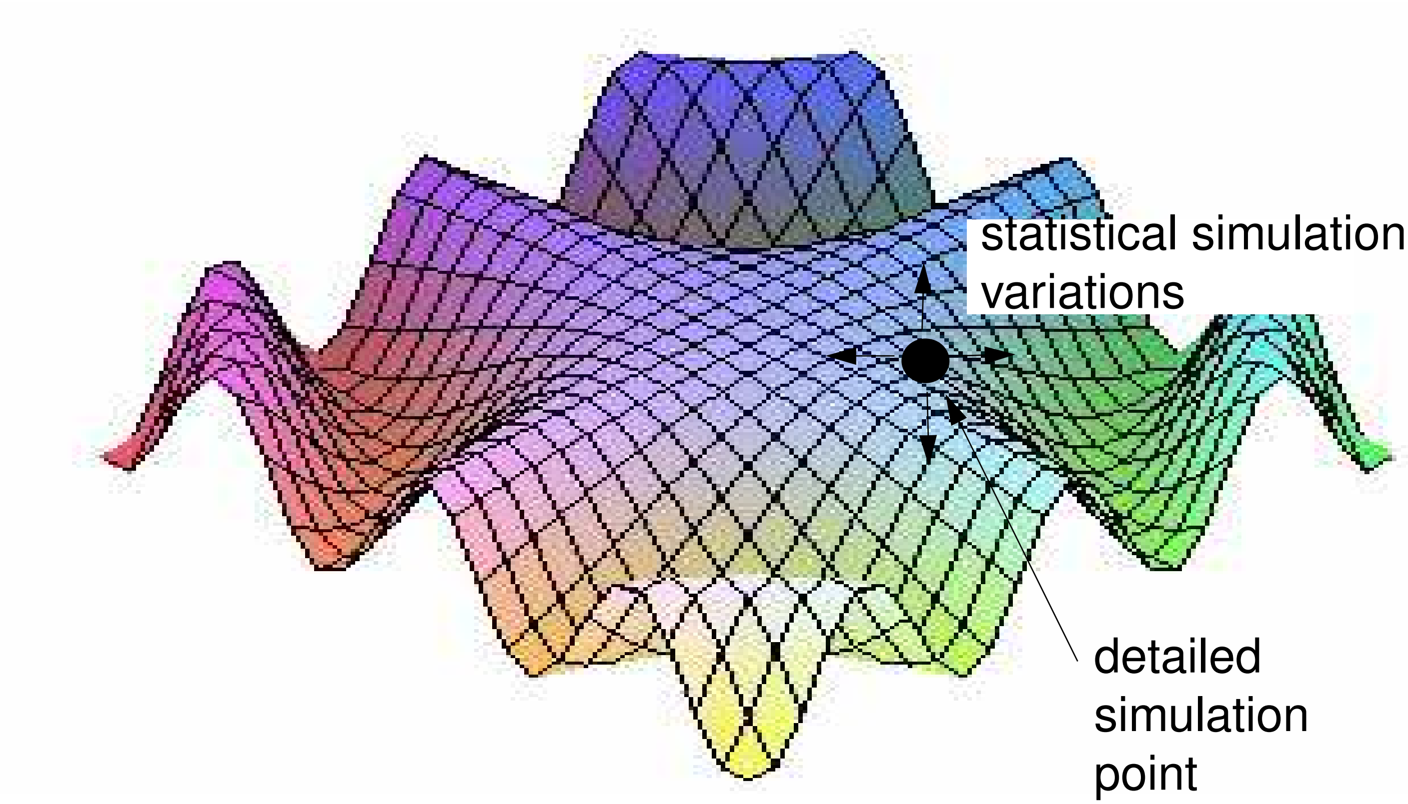
A New Alternative

- Statistical Simulation



Goal of Statistical Simulation

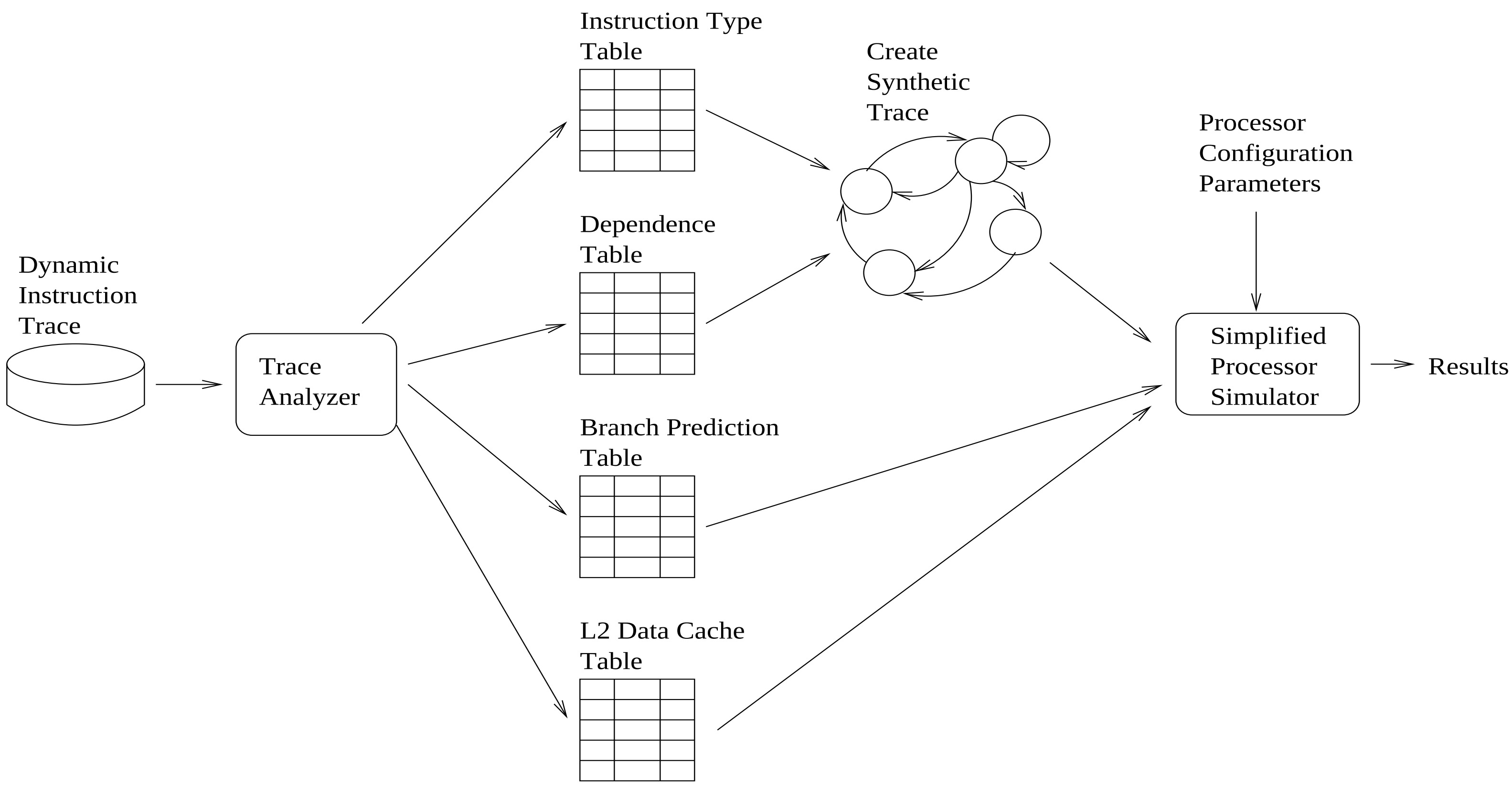
- Quickly explore gradients in design space
- Design point = application and hardware configuration



Outline

- Model
  - statistical characterization of application
  - processor model
- Validations
- Applications
- Future Work

Model - Statistical Characterization of Application/Processor



- What statistics are needed to characterize an application?
  - instruction mix, dependences, cache hit rates, etc.

Application Characterization - Instruction Mix

- Collect probabilities that an instruction is of type X given that the instruction y before it is of type Y (14 types overall)

FPdiv	Store	Sub	Beq	Load	Add	?

Prob(Load) = Prob(Load | 1 before is INT)
+ Prob(Load | 2 before is Load)
+ etc.

- Add up probabilities and compare to a random number generator to produce a synthetic instruction

- Assumption of independence reduces storage requirements

Application Characterization - Register Dependences

- Collect probabilities that an instruction depends on instruction y before it given that current instruction is of type X and previous instruction is of type Y

FPdiv	Store	Sub	Beq	Load	Add	FPmul

Prob(FPmul depends on instr 1 before | instr is INT)

Prob(FPmul depends on instr 2 before | instr is Load)

- Compare probabilities to a random number generator to produce a synthetic dependence

- Assumption of independence reduces storage requirements

Application Characterization - L2 Cache Behavior

- Collect probabilities that a LD/ST hits in L2 cache given that instruction y before is of type Y

FPdiv	Store	Sub	Beq	Load	Add	Store

hit in L2?

Prob(Store hits in L2 | instr 1 before is INT)

Prob(Store hits in L2 | instr 2 before is Load)

- Add up probabilities and compare to a random number generator to produce a synthetic cache access result

- Assumption of independence reduces storage requirements

Application Characterization - Branch Prediction

- Collect probabilities that a branch is mispredicted given that instruction y before is of type Y

FPdiv	Store	Sub	Beq	Load	Add	Bgt

↑ mispredict?

Prob(Mispredict | instr 1 before is INT)

Prob(Mispredict | instr 2 before is Load)

- Add up probabilities and compare to a random number generator to produce a synthetic branch prediction result

- Assumption of independence reduces storage requirements

Processor Simulation Model

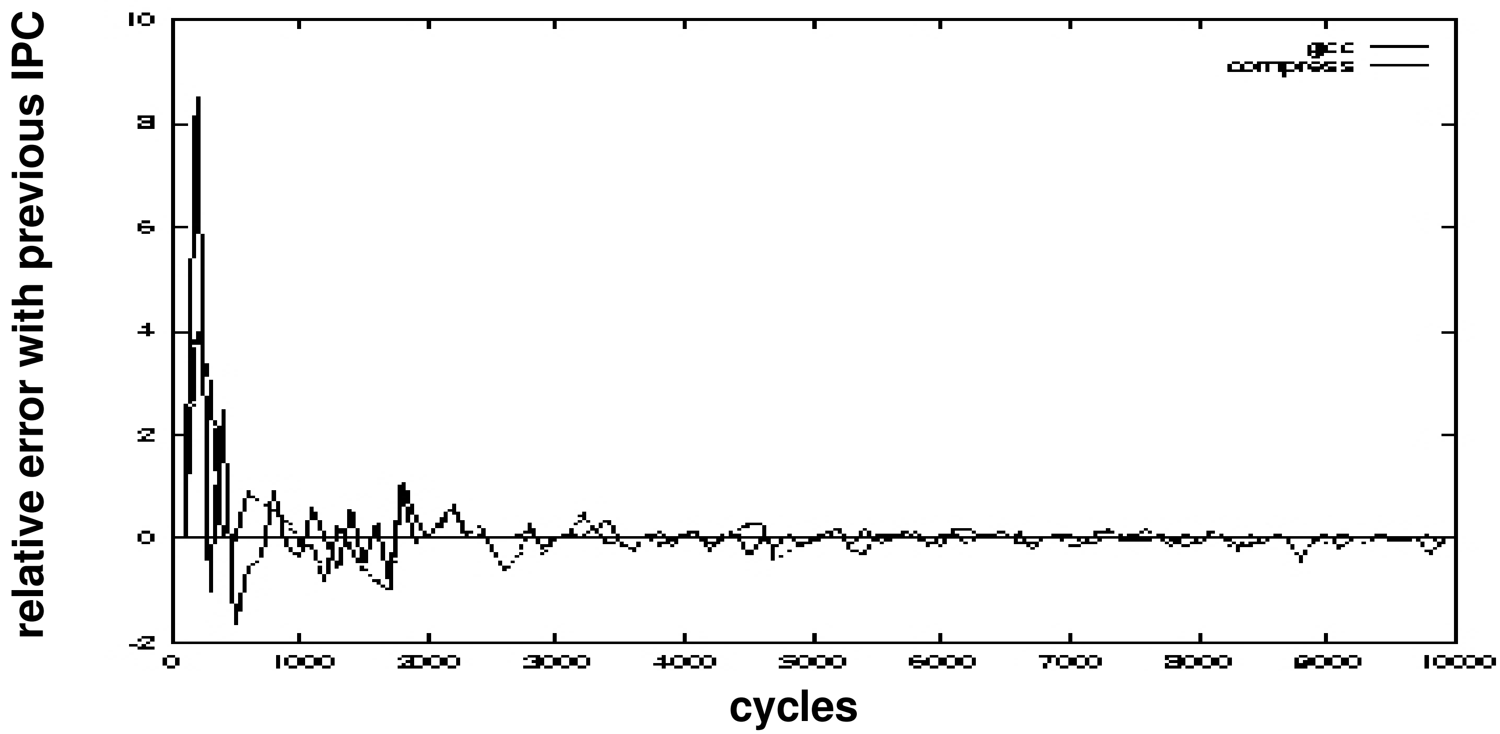
- Simplified version of SimpleScalar 2.0
  - same resources
  - release consistent
  - 14 types of instructions
  - instructions move through pipelines

  \+ does not need to produce values

  \- does not currently model memory dependences

Convergence of Synthetic Simulator

+ Only needs to run for a couple thousand instructions
+ Time to run = *seconds*



Outline

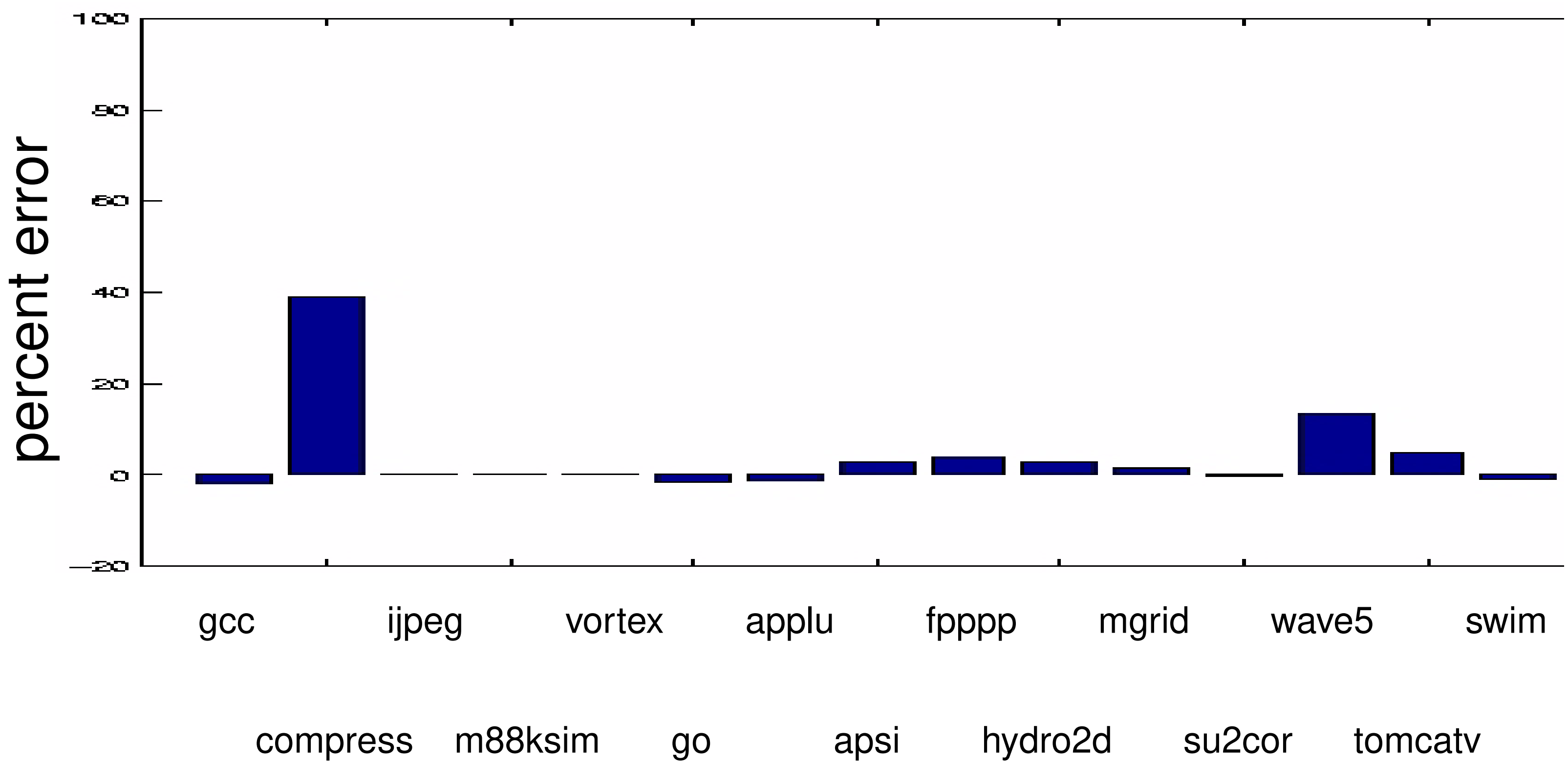
- ~~Model~~
- Validations
  - processor simulation model
  - application characterization
- Applications
- Future Work

Validation

- Spec95 benchmarks
- Results compared to SimpleScalar 2.0
- Progression from all real to all synthetic
  - allows for isolation of modeling inaccuracy

Validation - All Real

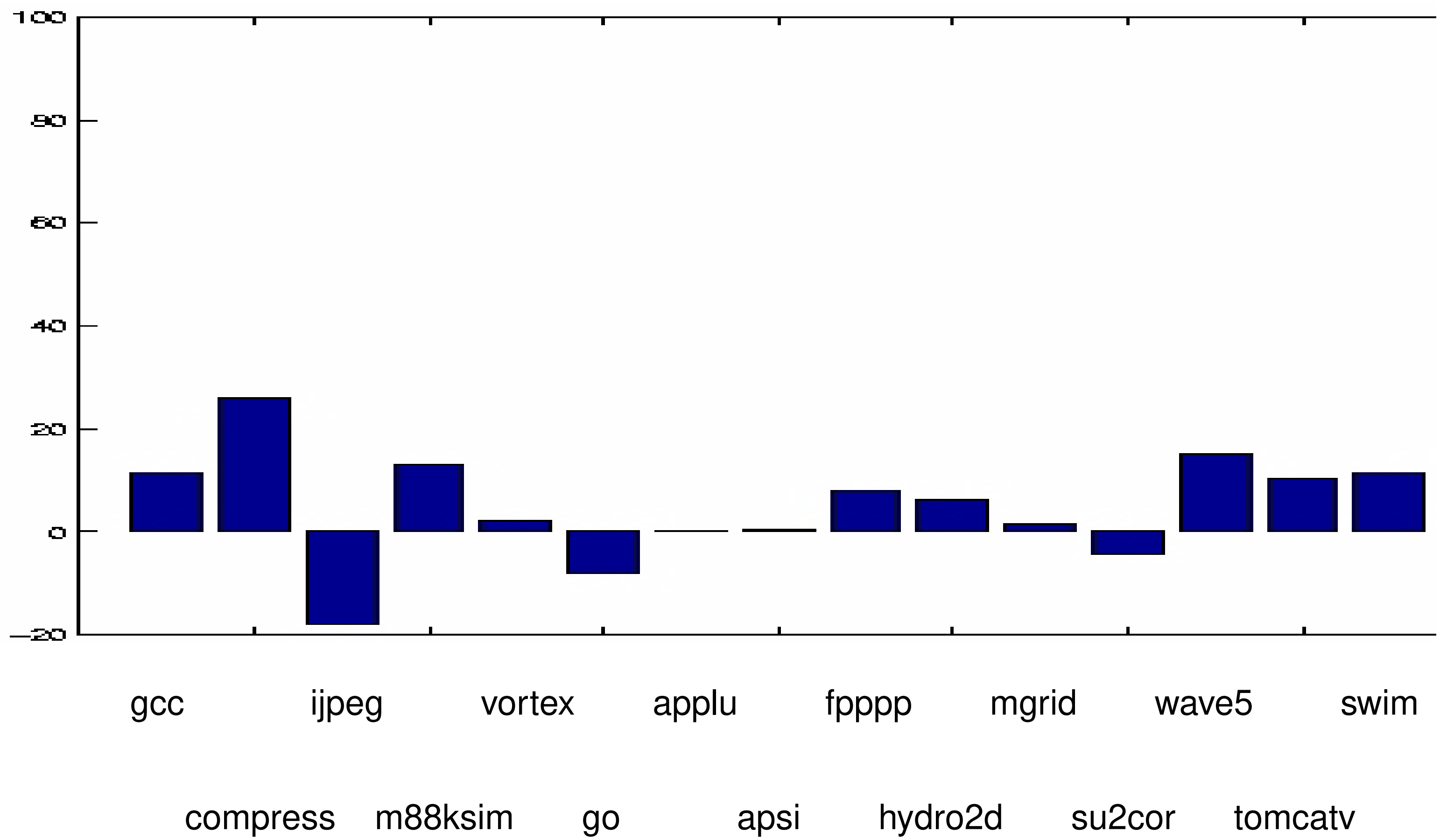
- Allows for validation of processor simulation model
  - Perfect caches, perfect branch prediction



- Compress: memory disambiguation not modeled
(357,825 stalls per 1M instrs vs. < 7,000 for most apps)

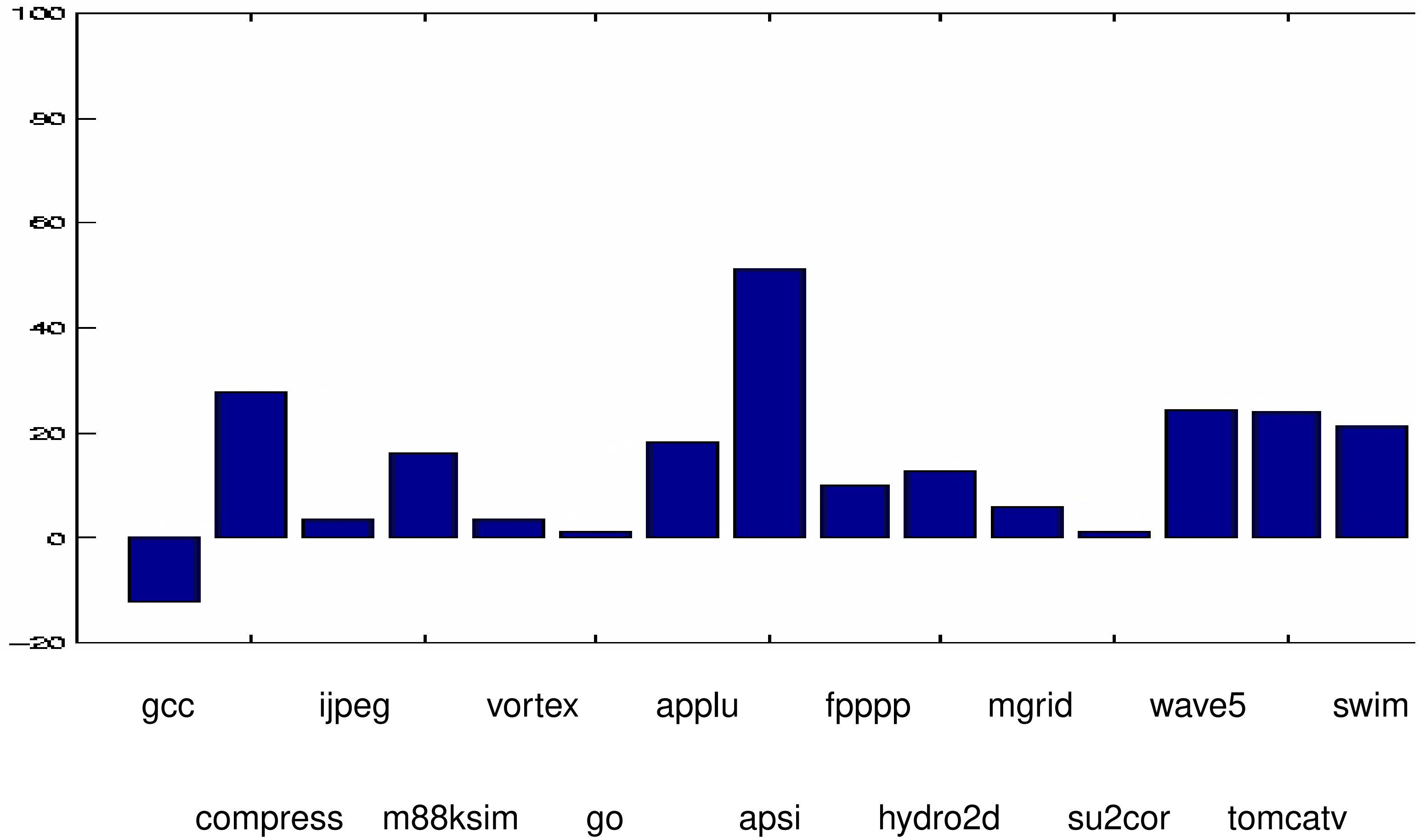
Validation - Synthetic Dependences

- Real instructions, perfect caches and branches



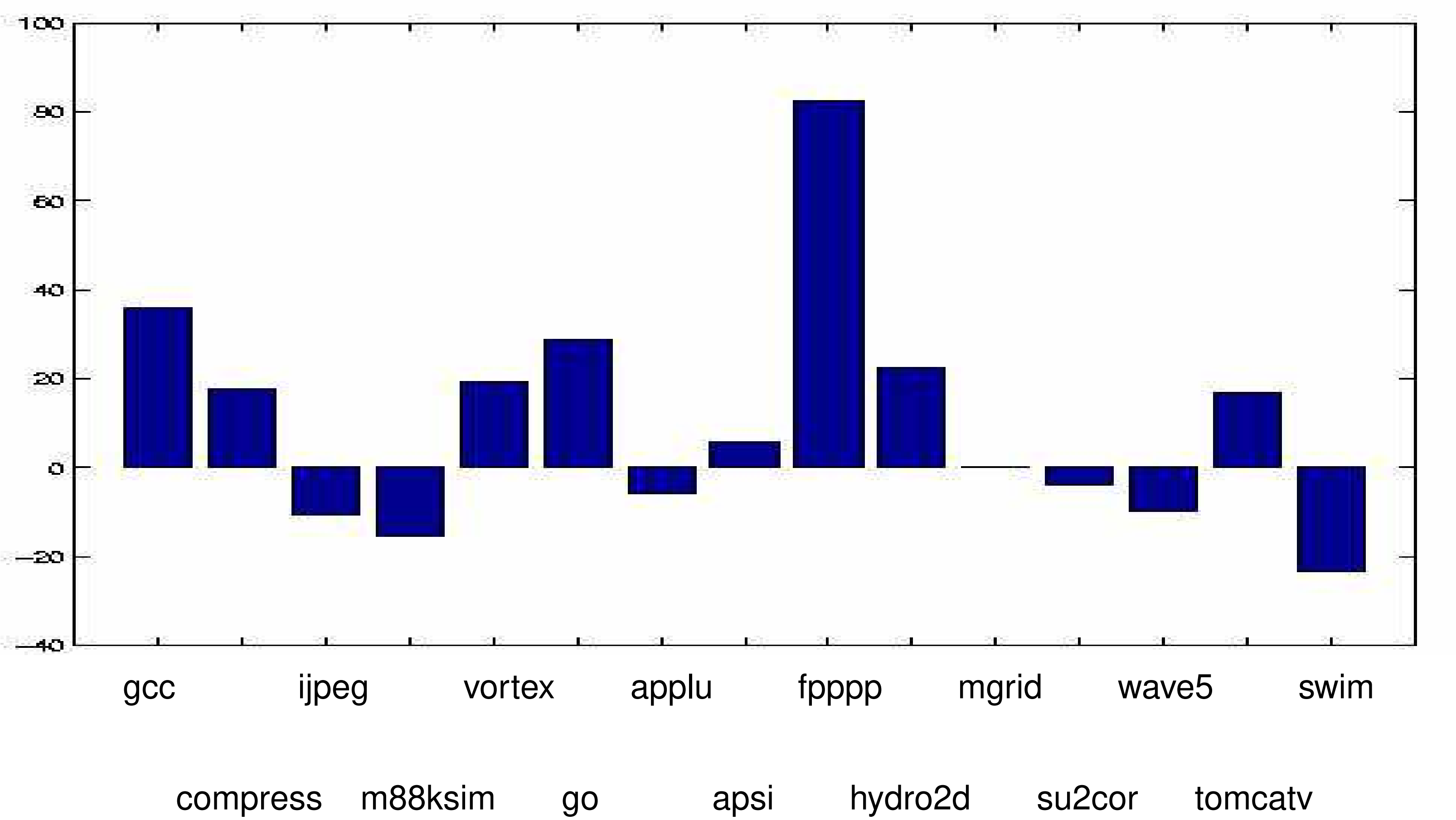
Validation - Synthetic Instructions and Dependences

- Perfect caches and branch prediction



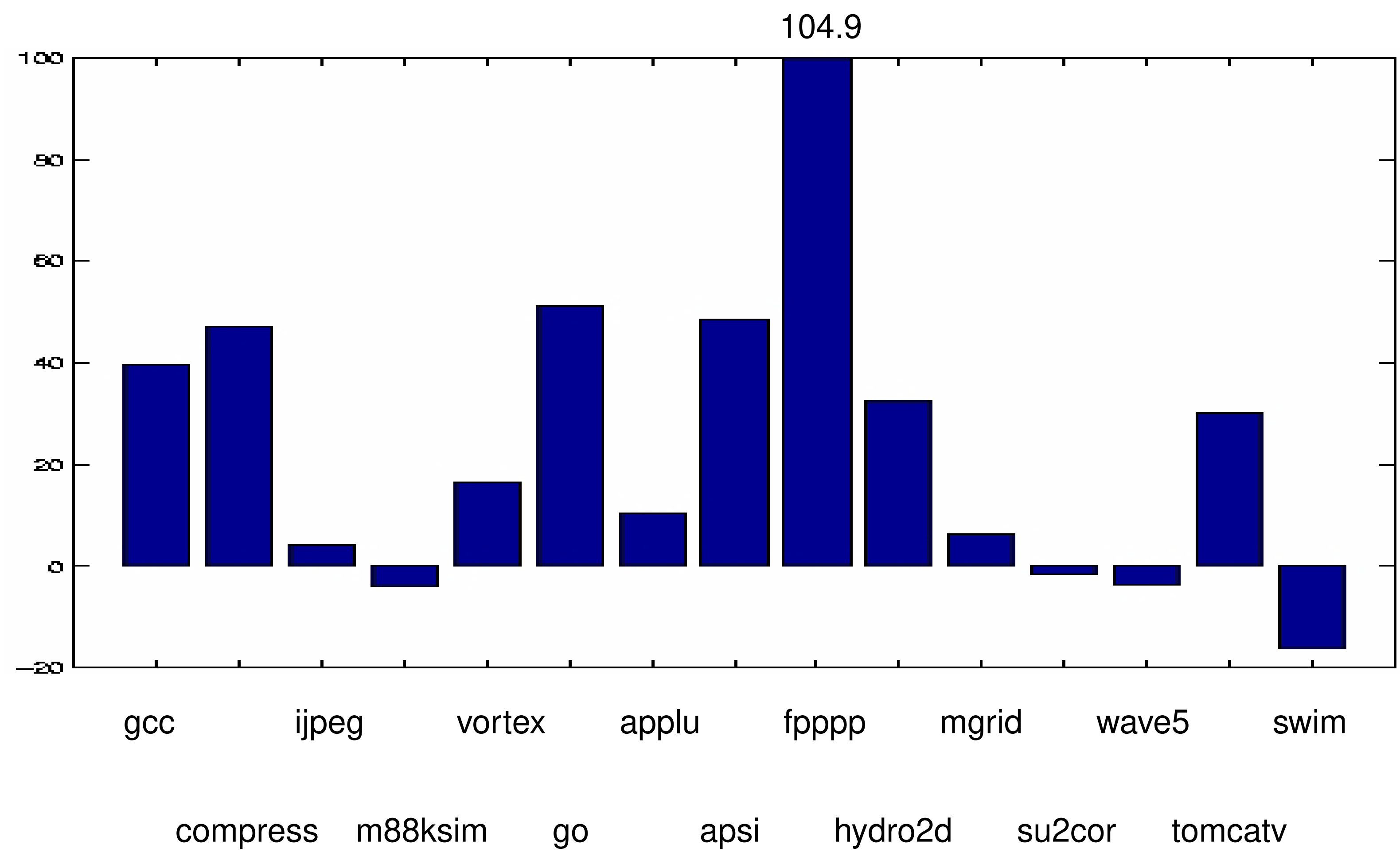
Validation - Prob Caches and Branches

- Real instructions and real dependences



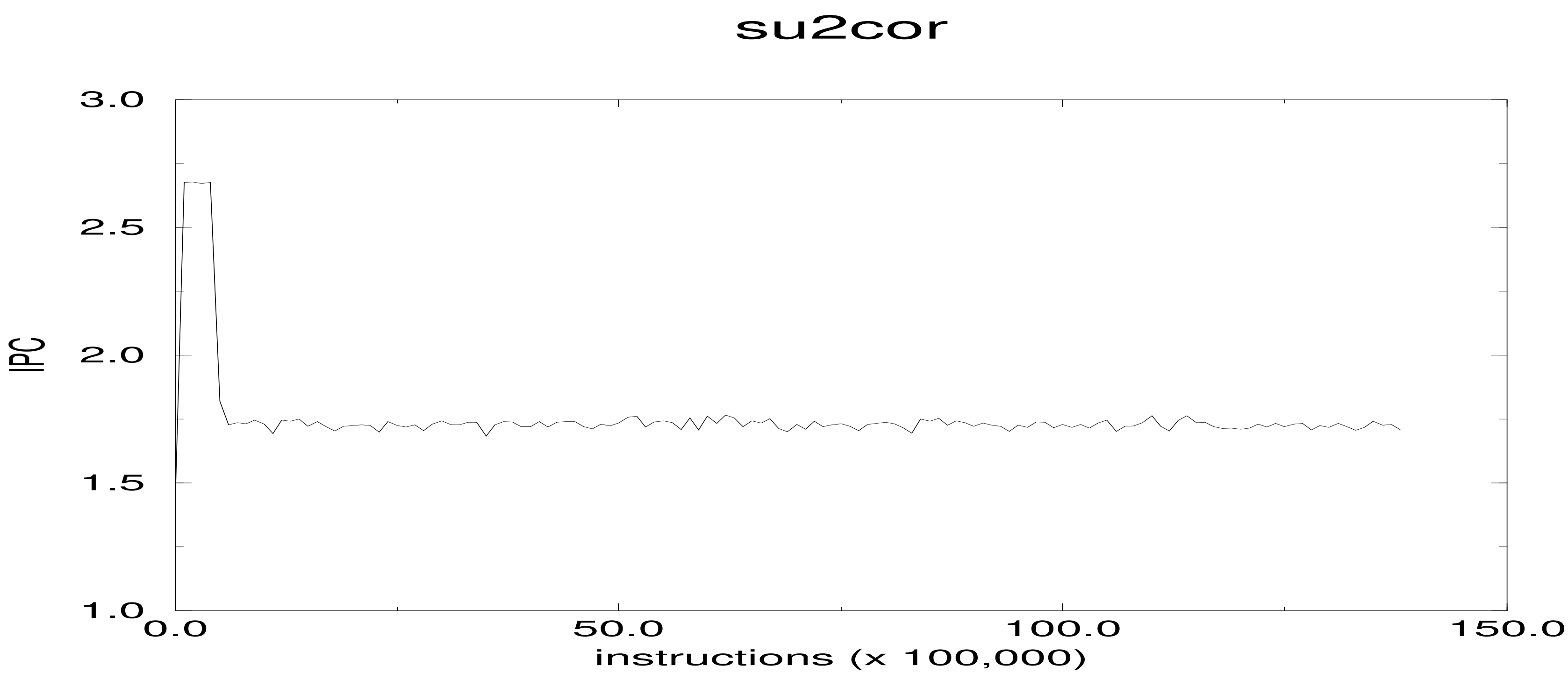
Validation - All Synthetic

- Prob caches and branch prediction



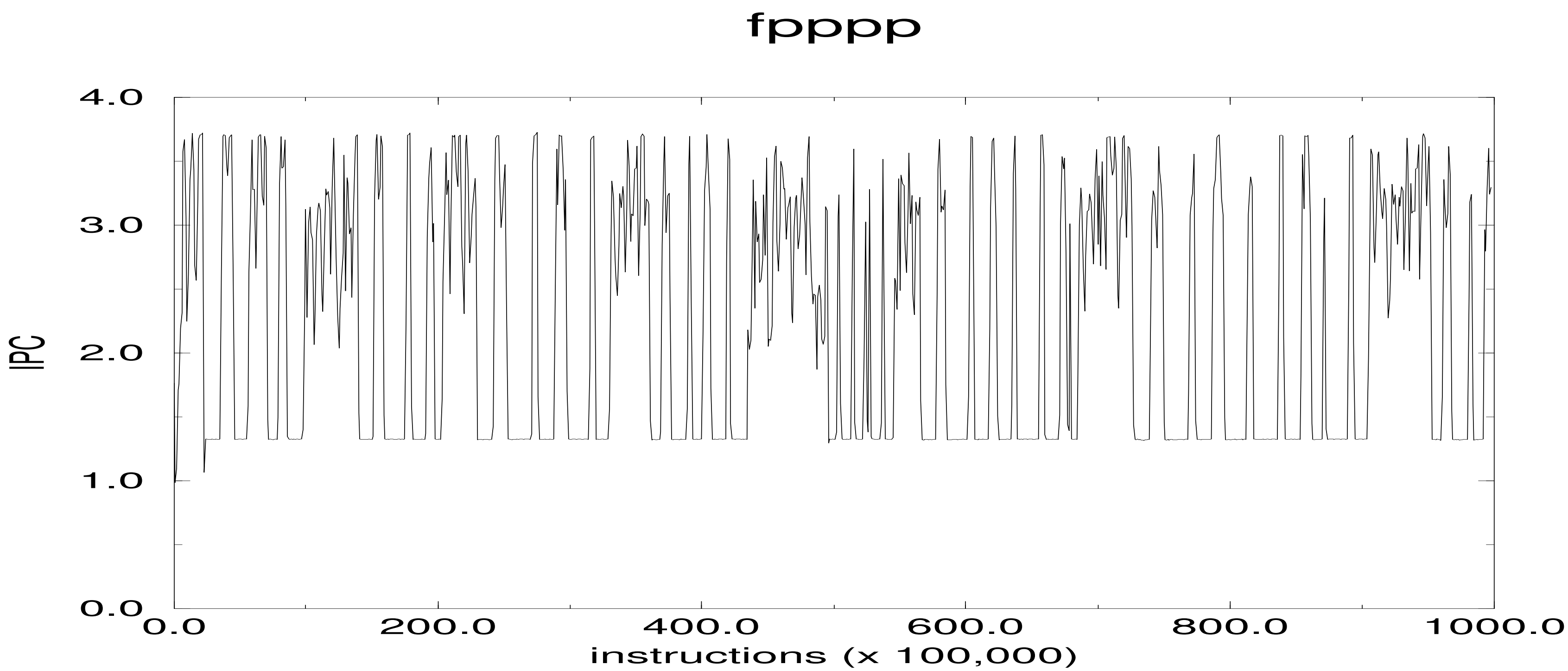
Non-Stationarity

- What if statistics vary over time?
- Context-based statistics do not solve problem
- Can only model stationary pieces of code



Non-Stationarity

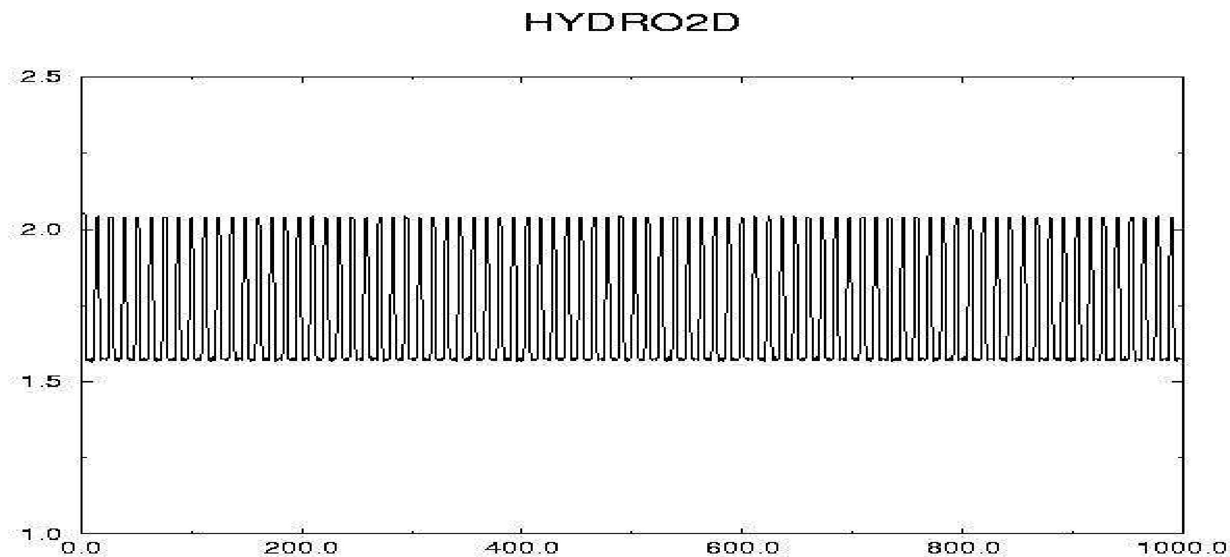
- But what about fpppp?



Phase Behavior

- Hydro2d exhibits 2 distinct phases
  - validated each separately

IPC

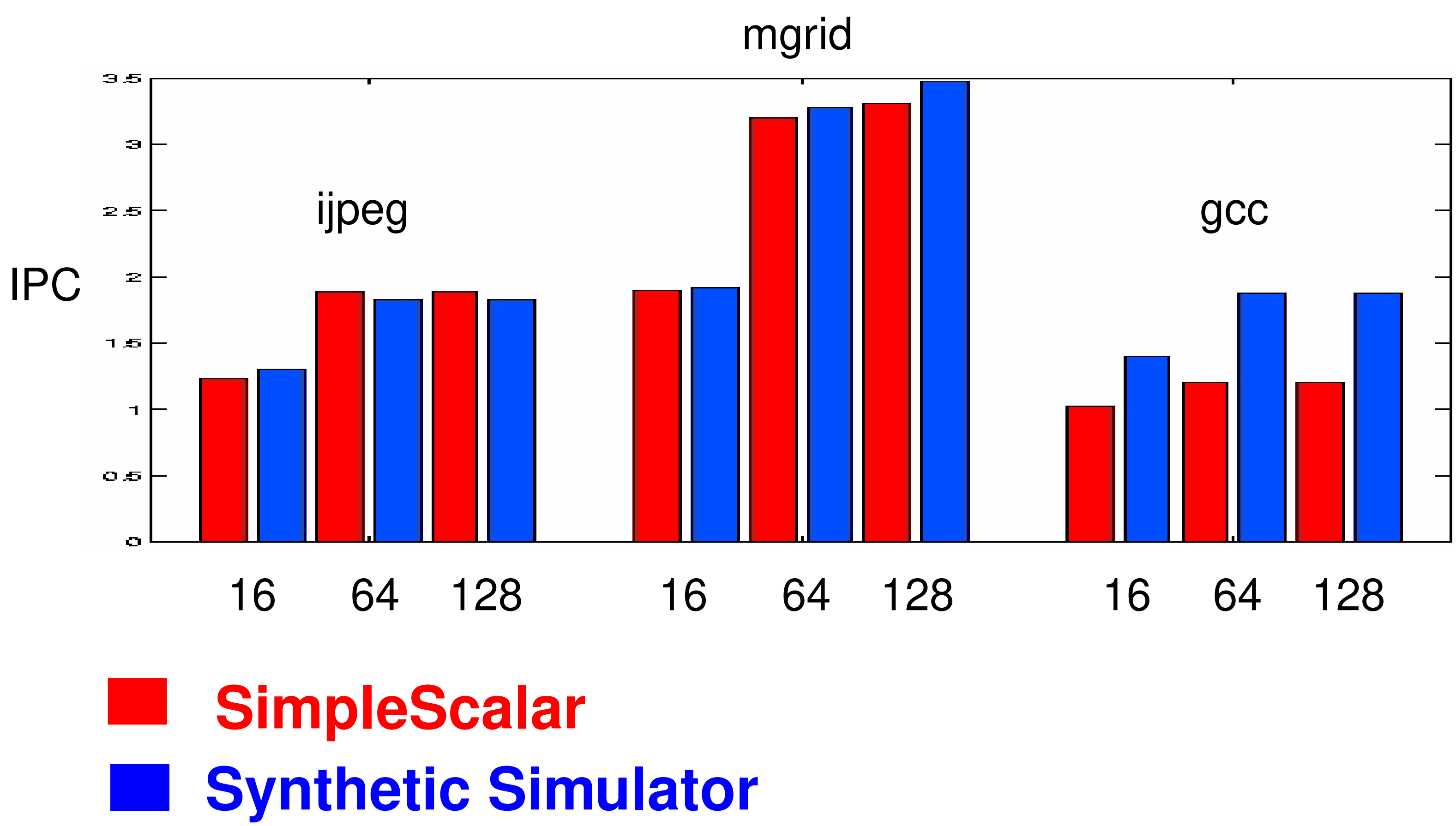


Outline

- ~~Model~~
- ~~Validations~~
- Applications
  - Effects of varying width of processor
  - Effects of varying size of reorder buffer
- Future Work

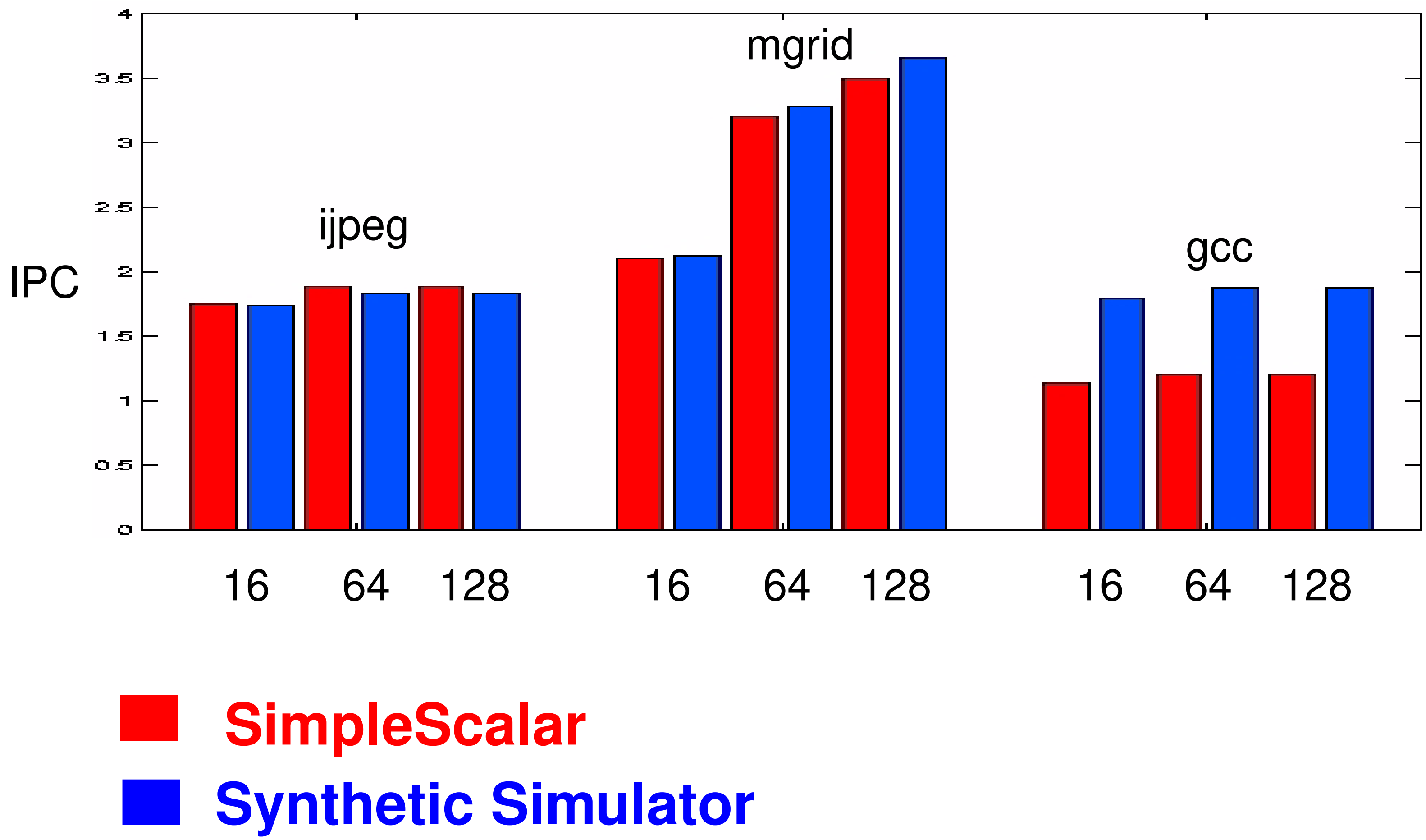
Effects of Varying Width of Processor

- Synthetic simulation captures trends in performance



Effects of Varying Size of Reorder Buffer

- Synthetic simulation captures trends in performance



Conclusions

- Synthetic simulation is a useful tool for exploring gradients in the design space
  - fast
  - accurately predicts trends in performance in applications shown here
- Still room for improvement in absolute accuracy
- Worth exploring other hybrid simulation techniques
  - partially real, partially statistical
  - e.g., use real addresses with synthetic trace

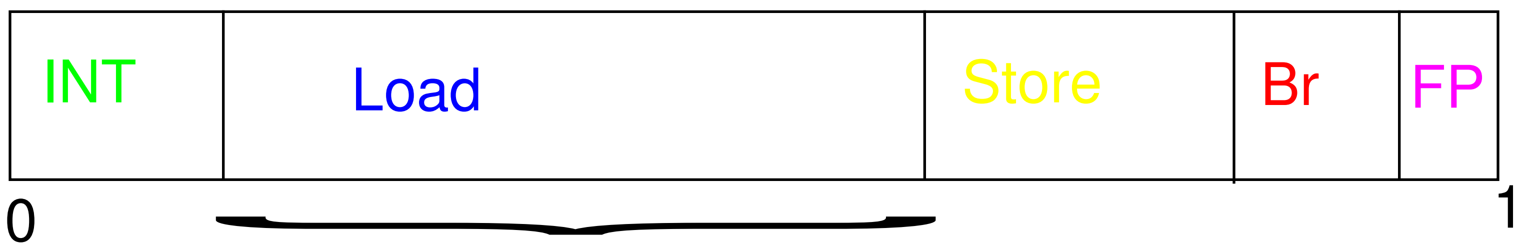
Future Work

- Improving absolute accuracy
  - Modeling Store -> Load dependences
  - More analysis of which parameters are most critical to accurate application characterization
- Automatic division of applications into stationary phases
- Extending to multiprocessor system
  - model coherence events statistically
  - plug synthetic processor into analytical model of system

Creating Synthetic Instructions

- Look at previous N instructions that have been created
  - For each, compute prob that current instruction is of type X given that this instruction is of type Y

- Sum up probs and normalize to 1 to get prob ranges



$= \sum_{y}$ Prob that current instr is a Load given that instr y before is of type Y

- Generate a Uniform [0,1] random variable, and choose whichever type of instr this value falls into