A Runtime Metric of Design Confidence

For Use in Dynamic Verification & Design Refinement

Joseph Greathouse and Kenneth Zick

December 12, 2006

EECS 578 Final Project

List of Topics We Will Use to Wow You

- Motivation
  - We restate some ideas from class and say what is broken!
- Background work
  - What you need to know about what we need you to know
- Problem Statement
  - We clearly define the problem so that *we* can be the ones to solve it
- Our Contributions
  - Includes our fantastic plan for fixing all the flaws we bring up
- Experimental Results
  - Proof that our work is a great solution!
- Conclusions
  - We will rush through this to finish on time.

Motivations

- Runtime verification (checker processors, etc.)
  - Benefits of this are pretty well-covered in this class.
- Even so, questions about runtime verification:
  - How confident are you in a deployed design?
  - Diagnose a problem in the field: Is your fix good?
    - What if the fix breaks something else?
  - How can you compare replacement designs?
  - What parts of the design are to blame when you detect a failure?
    - How badly broken are they?

Background work

- Formally Verified Checkers/DIVA
  - Find bugs in real-time, correct them with slowdown
- Statistical learning approaches
  - Learn and predict failure rates using runtime statistics
- Dynamically-reconfigurable computing
  - Replace “too-buggy” designs on reconfigurable circuits (e.g. FPGAs)
- Design diversity
  - Multiple versions of a design lessen chance of overlapping bugs

The Official Problem Statement

Find a scheme to quantify the confidence in a design at runtime

- Must be able to identify problematic regions in the design
- Should allow fair comparison of similar systems
- Needs to be constantly updated during system operation

Now For Our Solutions

A Runtime Metric of Design Confidence

Module-Level Probabilistic Diagnosis

A Runtime Metric of Design Confidence

Design confidence? What do you mean by that?

- An estimated probability that a design will operate correctly when run in a specific system environment (e.g. embedded system)
- Concerned with the **probability** of future failure, not number of bugs

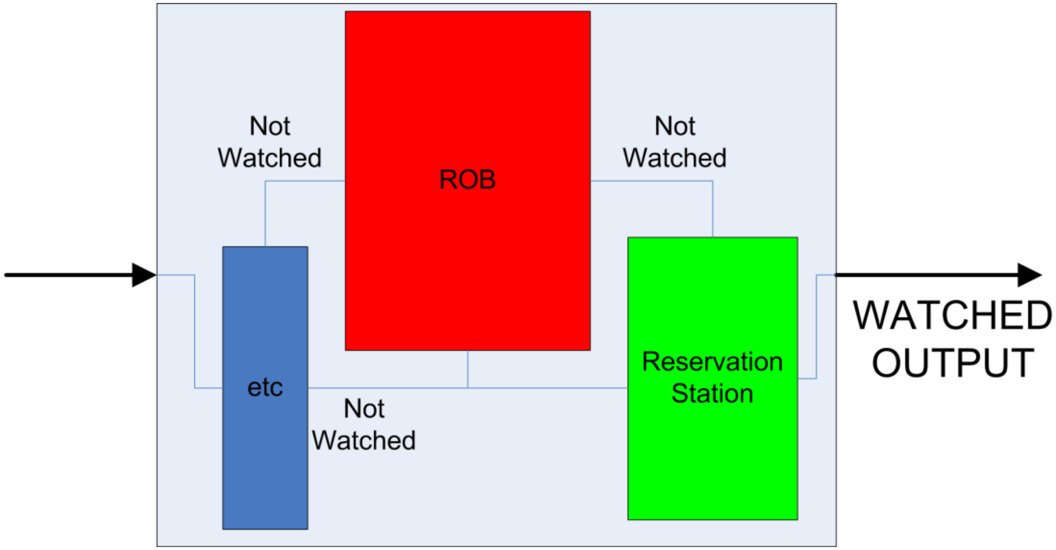
We represent confidence as a scalar value with range:
0 (terrible design) to 1 (we think it's good)

Key aspects of our metric:

- Failure statistics
- Probabilistic diagnosis

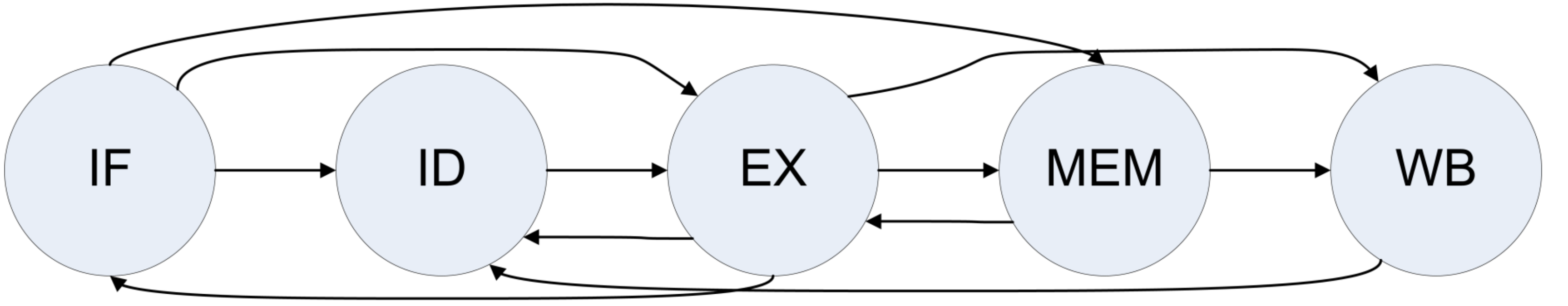
Failure statistics

- Failures detected by runtime checkers
  - Mark each *watched module* when you see an error
- Use failure data to estimate confidence in each module
  - Assumption: Future failures correlated with past failures
  - Statistical technique: parameter learning.
  - Predictions based on maximum likelihood hypothesis
- Must find some way to assign confidence to parts of the design we do not watch.



Probabilistic diagnosis

Create a directed weighted graph of the system:



- 'Causal network'
- Nodes represent design modules
- Links represent signals flowing from one module to another
- How is the weighting determined?

Probabilistic diagnosis

Some methods for determining weights:

1. Expert knowledge (ad hoc method)
   - "If this checker fails, there is probably a bug in IF, or possibly in ID"
2. Systematic analysis of system structure
   - *Compute the contributions of each module to the logic cone that feeds a checker.*
   - Treat modules as a black-boxes and base the weights on fanouts and proportion of interconnections.
   - In our proposal, the weight from module i to module j is:

$$w_{ij} = \begin{cases} 1 \text{ if module j has a checker and i=j} \\ \text{'X' (don't care) if module j has no checker} \\ \sum 1 / (\text{fanout}_s \times (\text{num signals to j})) \text{ for all signals s from } i \rightarrow j \end{cases}$$

Probabilistic diagnosis

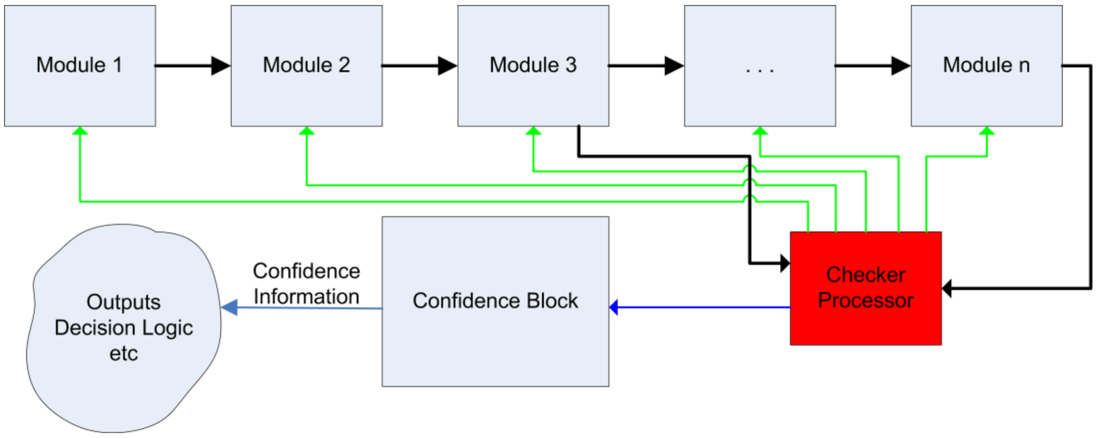
- Weights can be computed at design time. Saved in an *implication matrix*.
- At runtime failure, modules are implicated (blamed) according to the precomputed weights.
- Example: module C gets charged with 0.8 of a failure for every failure caught at module A.

Example: Implication matrix for system with 3 modules. Only modules A & B have checkers.

Src \ Dest	A	B	C
A	1	0.9	X
B	0.2	1	X
C	**0.8**	0.1	X

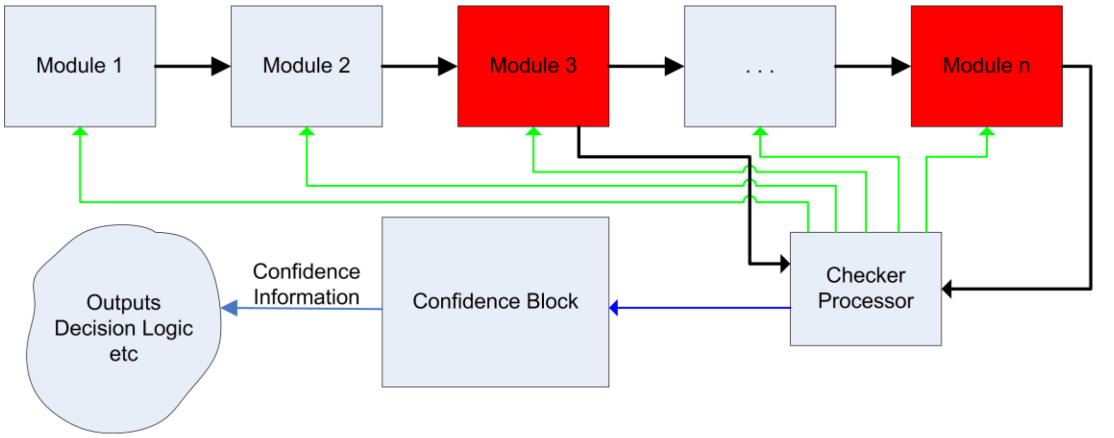
Runtime metric: an example

- Watch for errors with checker processor
- Record error numbers for watched modules
- Statically assign *weighted blame* to all modules based on these error numbers.
- Compute confidence in modules using compiled blame statistics



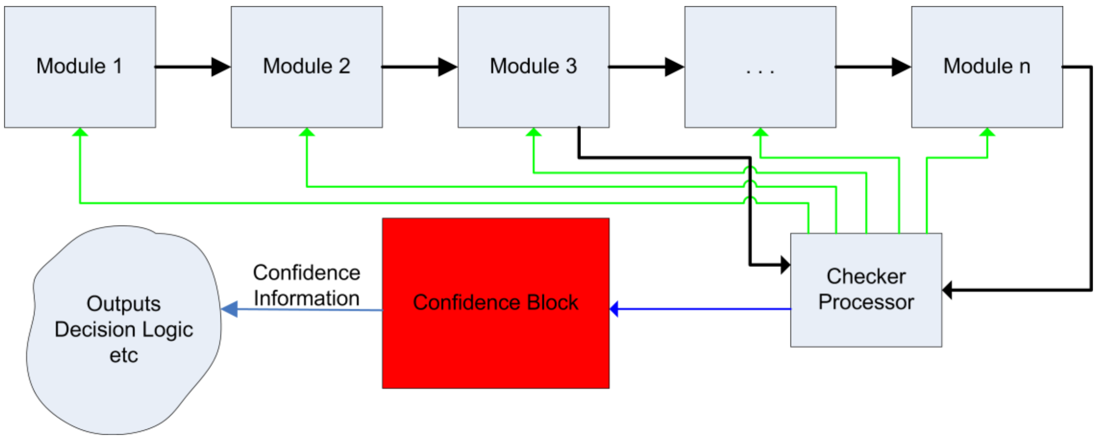
Runtime metric: an example

- Watch for errors with checker processor
- Record error numbers for watched modules
- Statically assign *weighted blame* to all modules based on these error numbers.
- Compute confidence in modules using compiled blame statistics



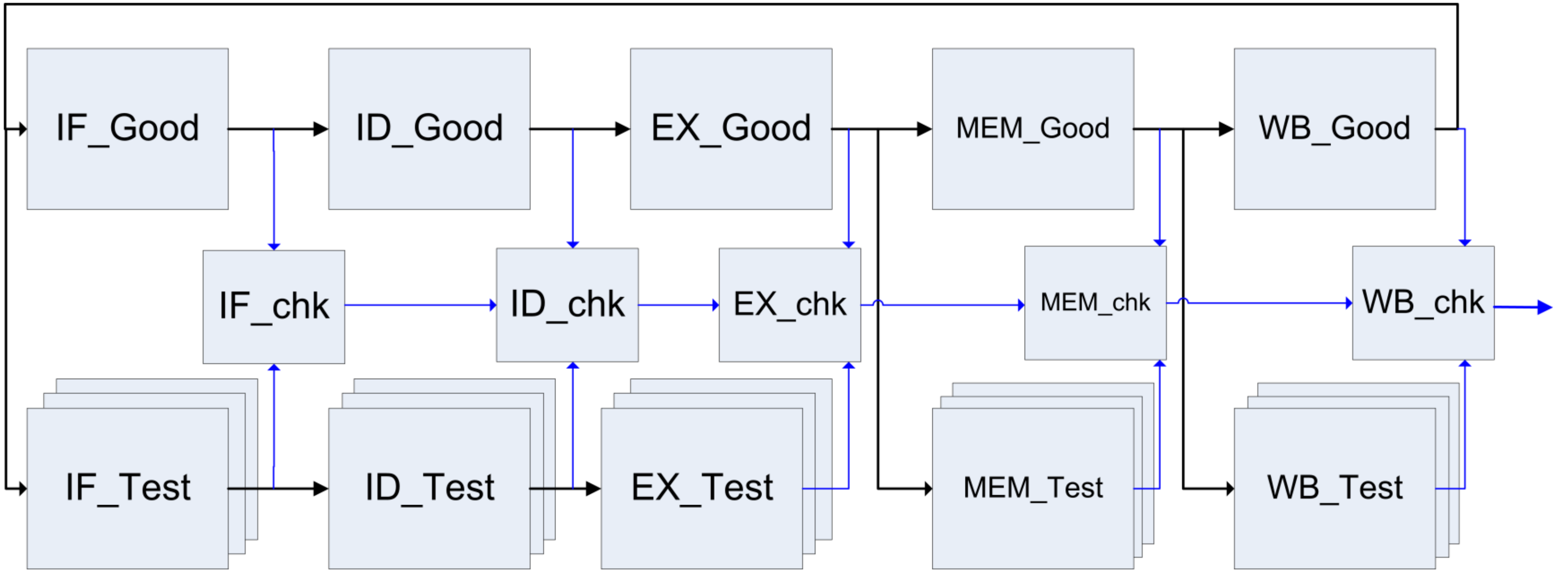
Runtime metric: an example

- Watch for errors with checker processor
- Record error numbers for watched modules
- Statically assign *weighted blame* to all modules based on these error numbers.
- Compute confidence in modules using compiled blame statistics



Experimental Setup

- Five-stage pipeline. One known-good (checker), one under test
- Multiple versions of each stage under test (one version active at a time)
- All stages under test have design defects
- Test suite: 50,000 vectors of directed tests
- Good stages maintain correct architectural state of bad pipe



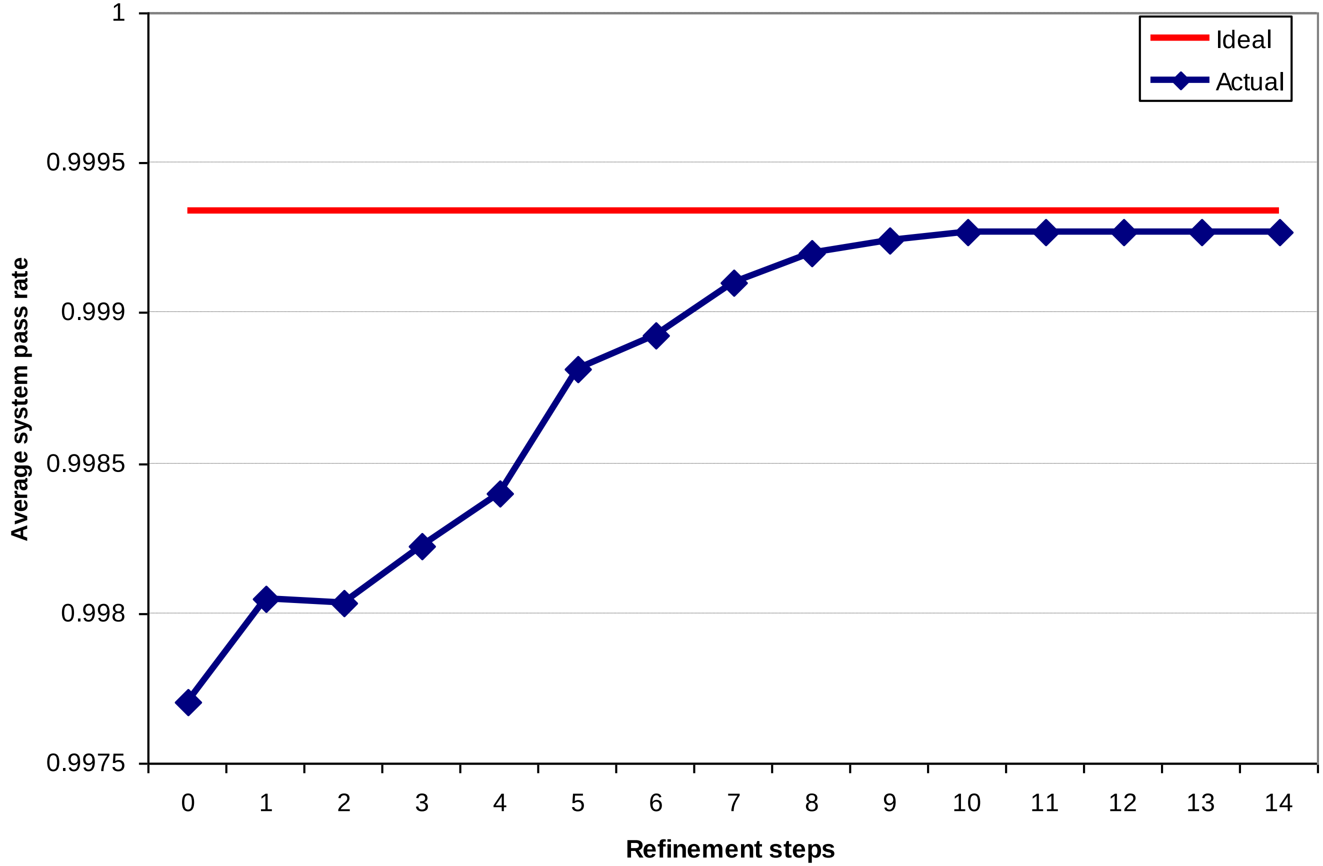
Experiment 1: Can design confidence be used to find a good system configuration?

- Checkers on every stage (i.e. assume full visibility)
- Select among two buggy versions of each stage
  - $2^5 = 32$ possible system configurations

- Initialize all confidence values to 1.0

- Simple decision procedure:
  - Compare failure rates of current version vs. others of same stage. Repeat for all modules.
  - Swap versions that lead to biggest increase in module confidence

Experiment 1 Results

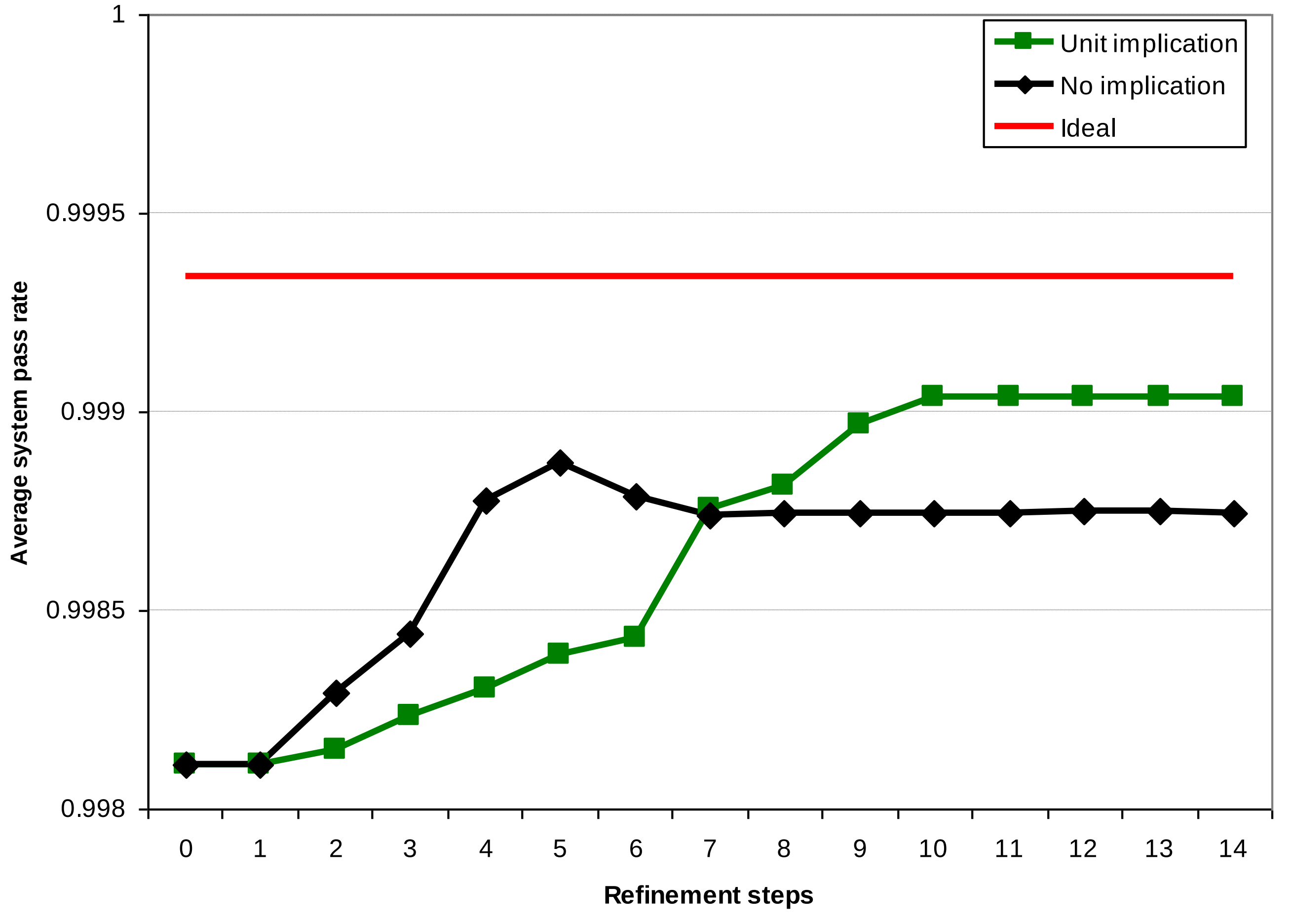


- We ran all possible starting configurations
- Results: System pass/fail rate improves significantly over time
  - Fail rate decreases to 36 fails every 50,000 cycles.
  - Optimal configuration is found for 2/3 of starting configurations

Experiment 2: When checking is limited, can we use simple probabilistic diagnosis?

- Partial checking: only 3 of 5 modules have checkers
- Watch signals that affect architectural state.
- Set weights to 1 for unchecked source modules.
- Again we ran from all possible starting configurations

Experiment 2 Results



- Results: even with simple implication (w=1), probabilistic diagnosis can provide some benefit (at least on this tiny example)!

Experiment 3: Hypothesis: proportional weighting will work even better

- Now try probabilistic diagnosis with *proportional* weighting

- Try two proportional weighting schemes:

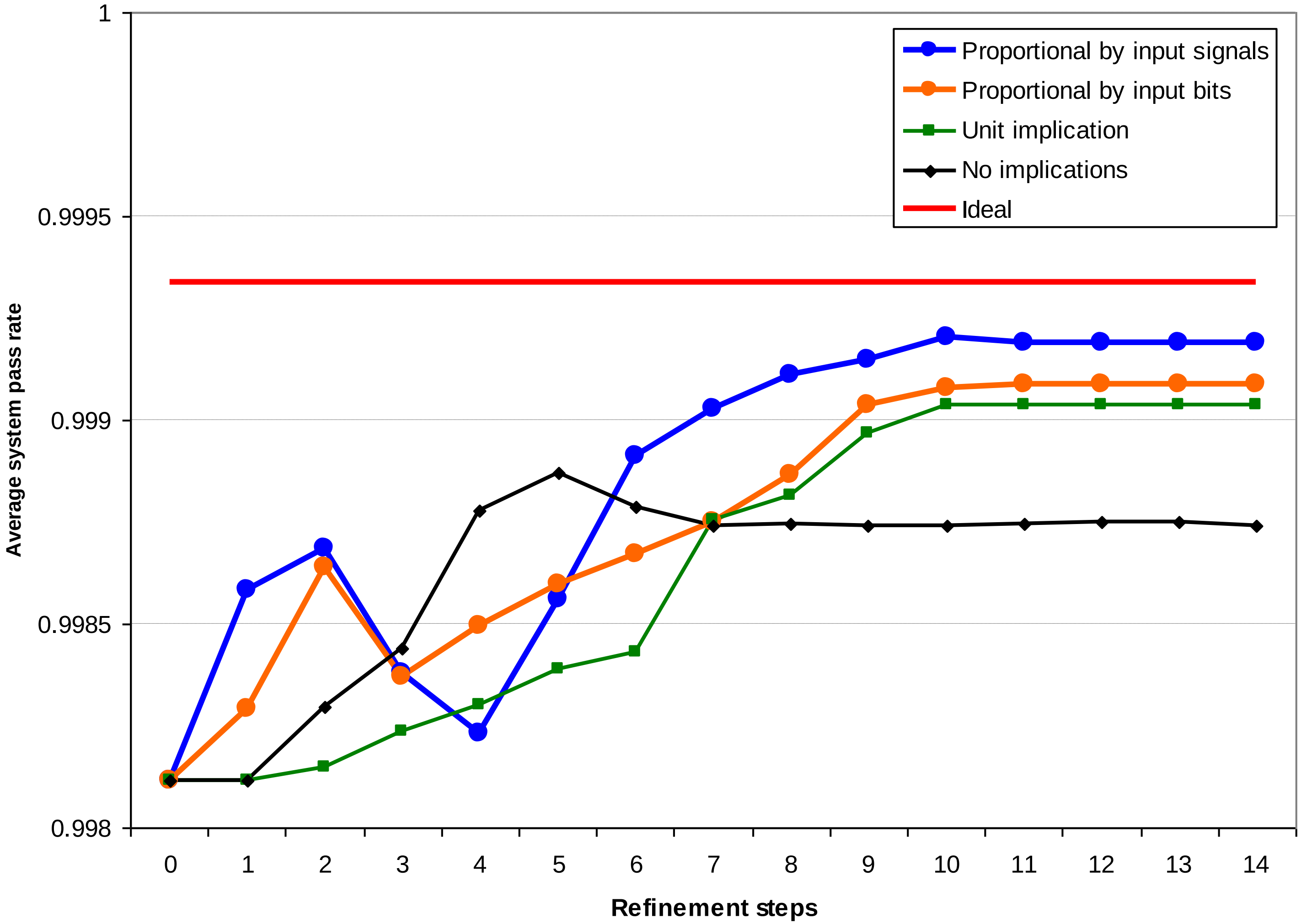
Implication matrix for weighting #1
(Based on total # of input bits)

Src \ Dest	IF	ID	EX	M	WB
IF	x	x	.321	0	.474
ID	x	x	.465	.015	.044
EX	x	x	1	.985	.007
M	x	x	.214	1	.474
WB	x	x	0	0	1

Implication matrix for weighting #2
(Collapse data busses into single input signal)

Src \ Dest	IF	ID	EX	M	WB
IF	x	x	.702	0	.111
ID	x	x	.277	.4	.667
EX	x	x	1	.6	.111
M	x	x	.021	1	.111
WB	x	x	0	0	1

Experiment 3 Results



Lessons learned

- Cross-pollination can yield useful ideas
  - Software reliability originally modelled on hardware reliability
  - Due to design complexity, HW may now benefit from SW reliability ideas
- Many of these concepts have a surprisingly long history.
- Assertions are not as great as we initially thought
- Industry has related projects with related ideas
  - e.g. Sun Niagara II, IBM autonomic computing
- Possible future work:
  - Scalability, new heuristics, sophisticated weighting, non-proc. systems

Thank you!