# *intelligent* Thread-Level Speculation

Jeremy Singer

# Outline

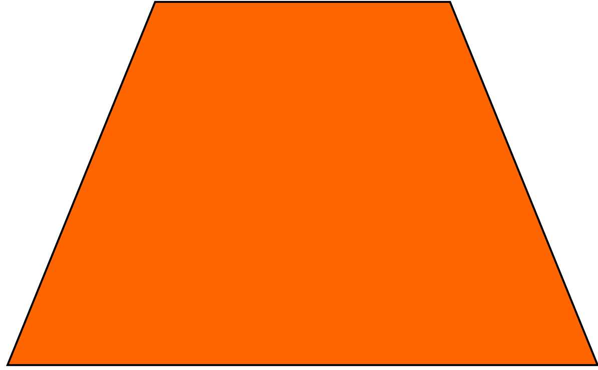
- what is TLS?
- what is intelligence?
- what do we do?

# *i* TLS

# Machine Learning

For

# Dummies



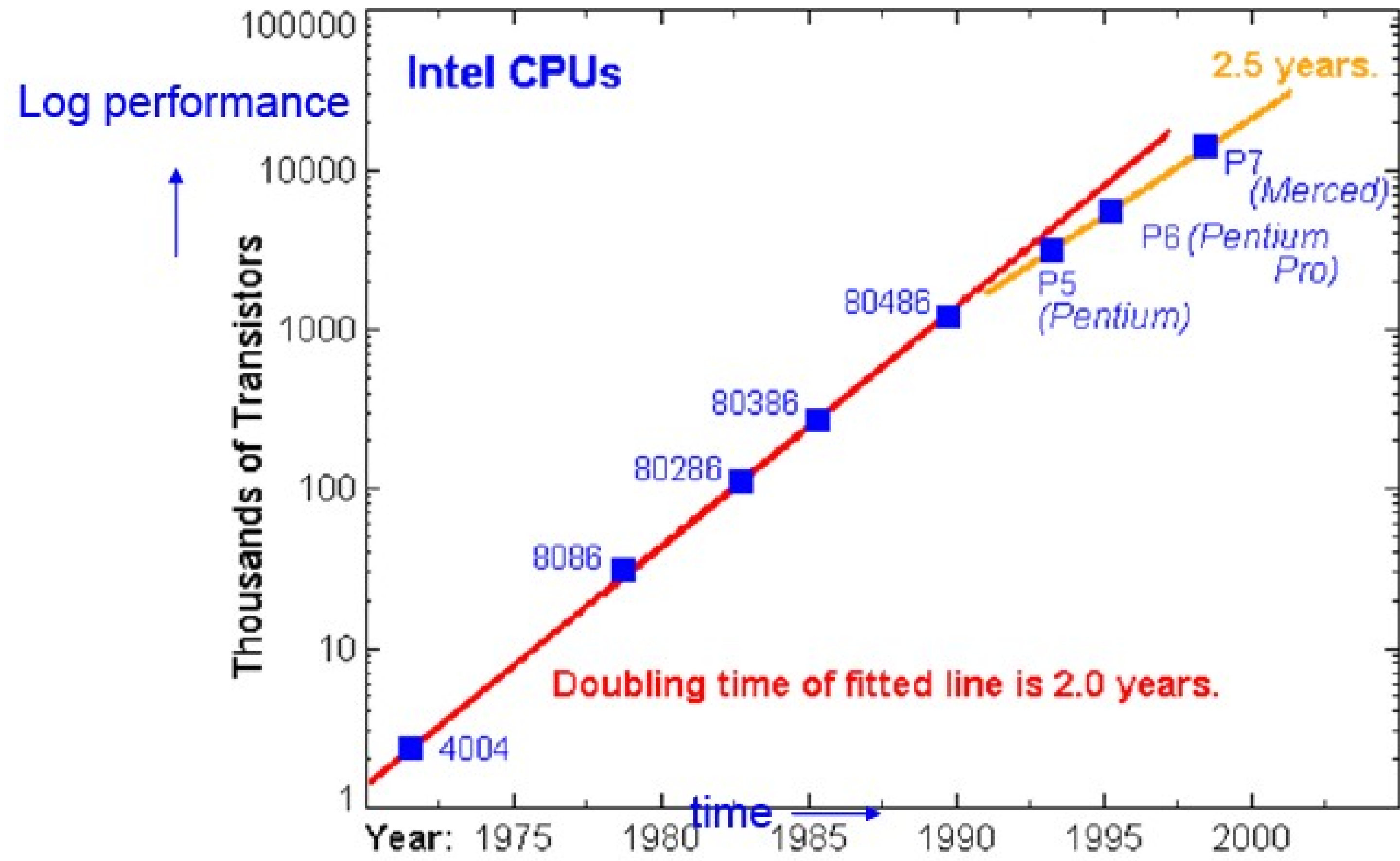
squash prediction

# Thread-Level Speculation in a nutshell

- parallel execution of sequential code
- unsafe optimizations with runtime checking

# Motivation (1)

- Moore's Law in single-core era



# Motivation (1)

- Moore's Law in multi-core era

# Motivation (2)

- Plenty of legacy code around!
- In 1997, the Gartner Group reported that 80% of the world's business ran on COBOL with over 200 billion lines of code in existence. [wikipedia]

# Can't we do auto-parallelization?

- We can try – in some cases it works
- complex static analysis, prove lack of data dependence, insert thread spawn/join points

- essentially ***conservative***
- Is this good enough?

# TLS as an alternative

- insert thread spawn/join points into code where we hope there will not be any data dependence (but we won’t prove it)

- at runtime, system checks for data dependence violations. If these occur, *squash* spec thread and re-execute sequentially

# Method-Level TLS

time

# Read-after-Write Dependence

```
void caller() {
  // do some work …
  callee();
  // do some more work …
  read value of x;
}

void callee() {
  x = 42;
}
```

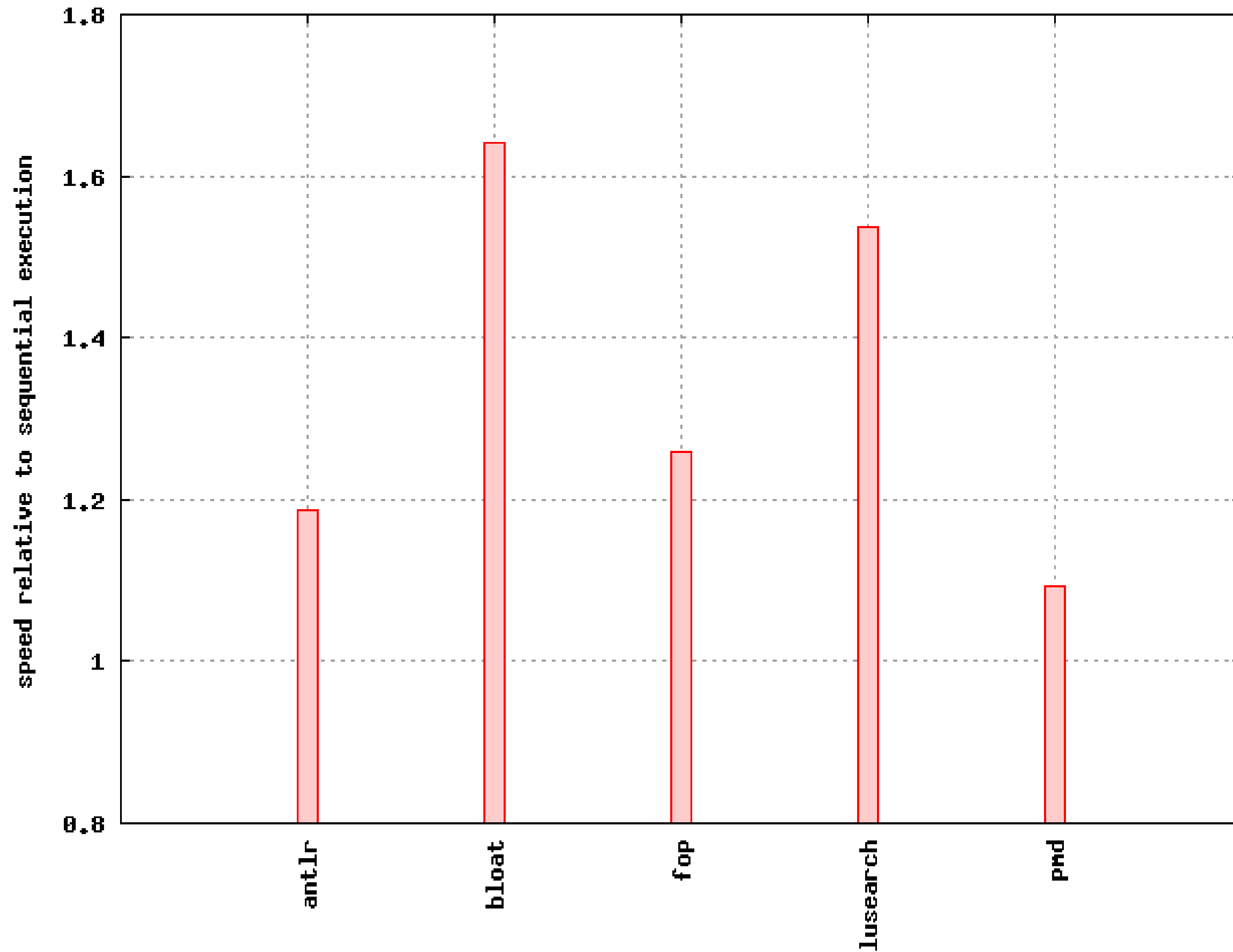
# Squashing

- Undoes speculative execution
- Re-executes caller continuation non-speculatively
- Overhead for squashing
- Perhaps, only spawn when we are fairly sure that squashing is unlikely …
- ***Squash prediction***

# Our implementation

- sequential traces from instrumented JVM
  - give method timings
  - give method call graph
  - give memory reads/writes
- use a custom trace-driven TLS simulator model
- pluggable interfaces for squash prediction, etc

# Limits of Method-Level TLS



# TLS Heuristics

- rule-of-thumb
- devised by human experts
- domain-specific (program-specific)
- to predict good thread spawn points?
- to predict method return values?
- etc…
- How can we create such heuristics automatically? …

# Machine Learning

For

# Dummies

# different kinds of ML

- supervised versus unsupervised
  - we focus on *supervised*
- classification versus regression
  - we focus on *classification*
- online versus offline
  - we focus on *offline*

# how it works

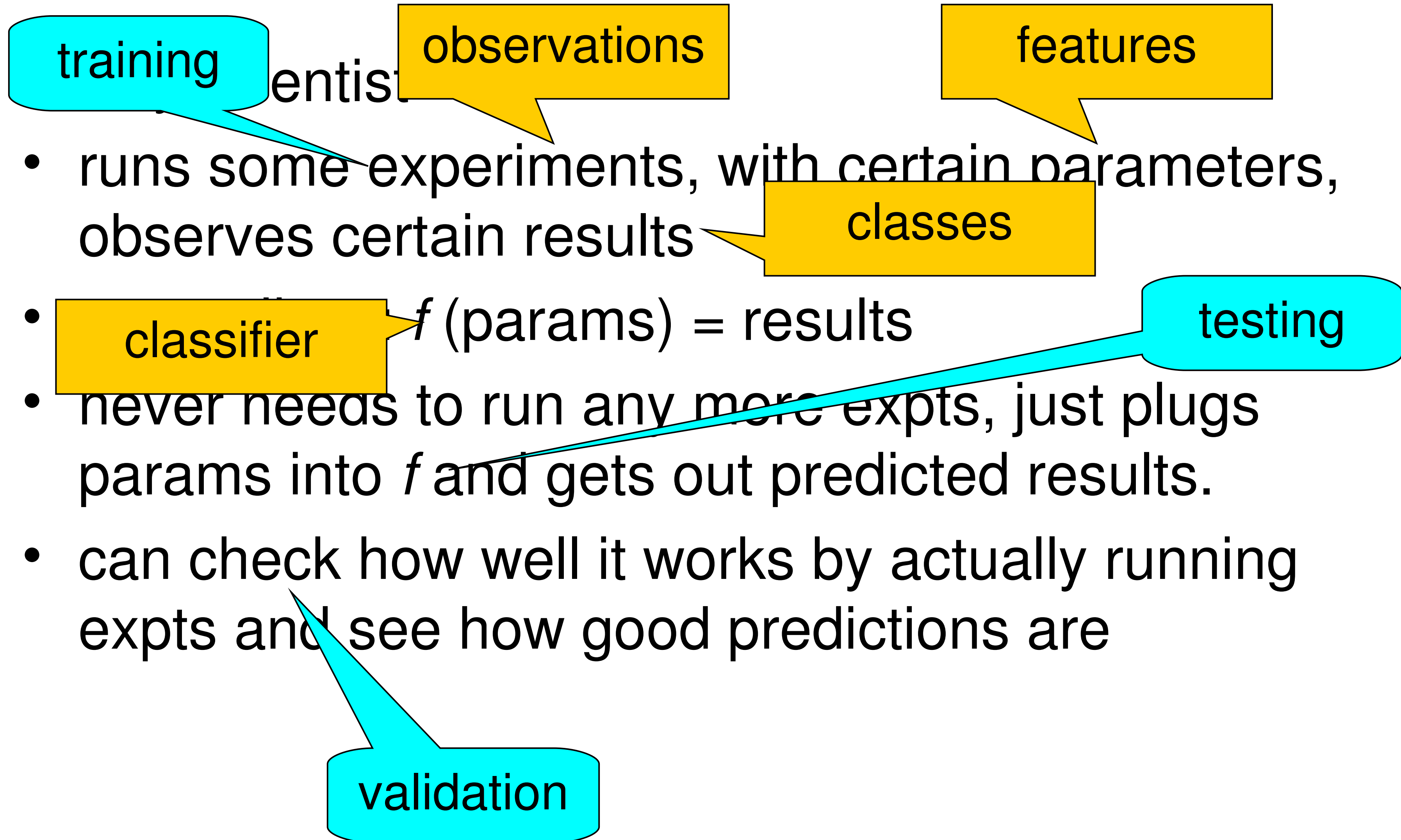
- lazy scientist
- runs some experiments, with certain parameters, observes certain results
- generalizes: $f$ (params) = results
- never needs to run any more expts, just plugs params into $f$ and gets out predicted results.
- can check how well it works by actually running expts and see how good predictions are

# proper ML terminology

- observations
- features
- class
- classifier
- training
- testing
- validation

- don’t overlap train/test sets

# how it works

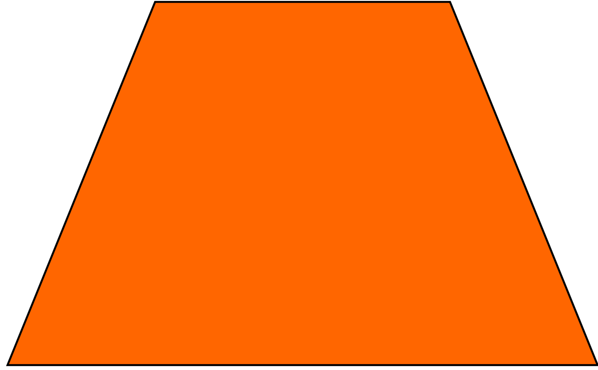


# What does ML give you?

- various *learning algorithms* that generate classifiers
- quantitative methods to compare performance of different classifiers

# What ***doesn't*** ML give you?

- a way to choose the optimal learning algorithm
  - requires ML expert
- a way to choose appropriate features to characterize the problem
  - requires domain expert



squash prediction

# Why is Squash Prediction so Important?

- Thread squashes are expensive
  - require rollback
  - redo computations
- Statically – avoid inserting spawn instrs at compilation time
- Dynamically – suppress spawns when we think they are unlikely to succeed

# Existing Heuristics for Squash Prediction

- Whaley and Kozyrakis, ICPP 2005
- Simple heuristics based on method runlength or number of store instructions

# ML to auto-generate heuristics

- For method-level TLS, statically insert spawns at all method call points
- At runtime, set dynamic policy to spawn everywhere
- See which method calls lead to squashes…

# Challenges

- How do we characterise dynamic method call sites?
  - Which features tell us where squashes occur?

- At each call site, take some measurements on *caller* and *callee* methods

# Static Features

| *category* | *name* | *description* |
|---|---|---|
| Calling | NoParams | takes no arguments |
| | NoReturn | returns `void` |
| | **Recursive** | calls itself recursively |
| | SameName | calls another method with same name |
| | **Leaf** | does not issue any method calls |
| OO | ObjectCreator | creates `new` objects |
| | FieldReader | reads field values from an object |
| | FieldWriter | writes values to field of an object |
| | TypeManipulator | uses type cast or instanceof |
| Control Flow | **StraightLine** | no branches in method body |
| | Looping | loop(s) in method body |
| | Exceptions | may throw an unhandled exception |
| Data Flow | **LocalReader** | reads local var(s) |
| | LocalWriter | writes local vars |
| | **ArrayCreator** | creates new array |
| | **ArrayReader** | reads values from array |
| | **ArrayWriter** | writes values to array |

# Dynamic Features

- Method runlength
  - Measured in term of instrs, bytes, or time
- Num reads/writes/allocs
  - Split into different memory spaces
- Num calls

# Data set characteristics

- Large amounts of data (many call sites)
- Repetition (many recurring call sites)
- Ambiguity (call sites have different behaviour)
- Imbalance (more commits than squashes)

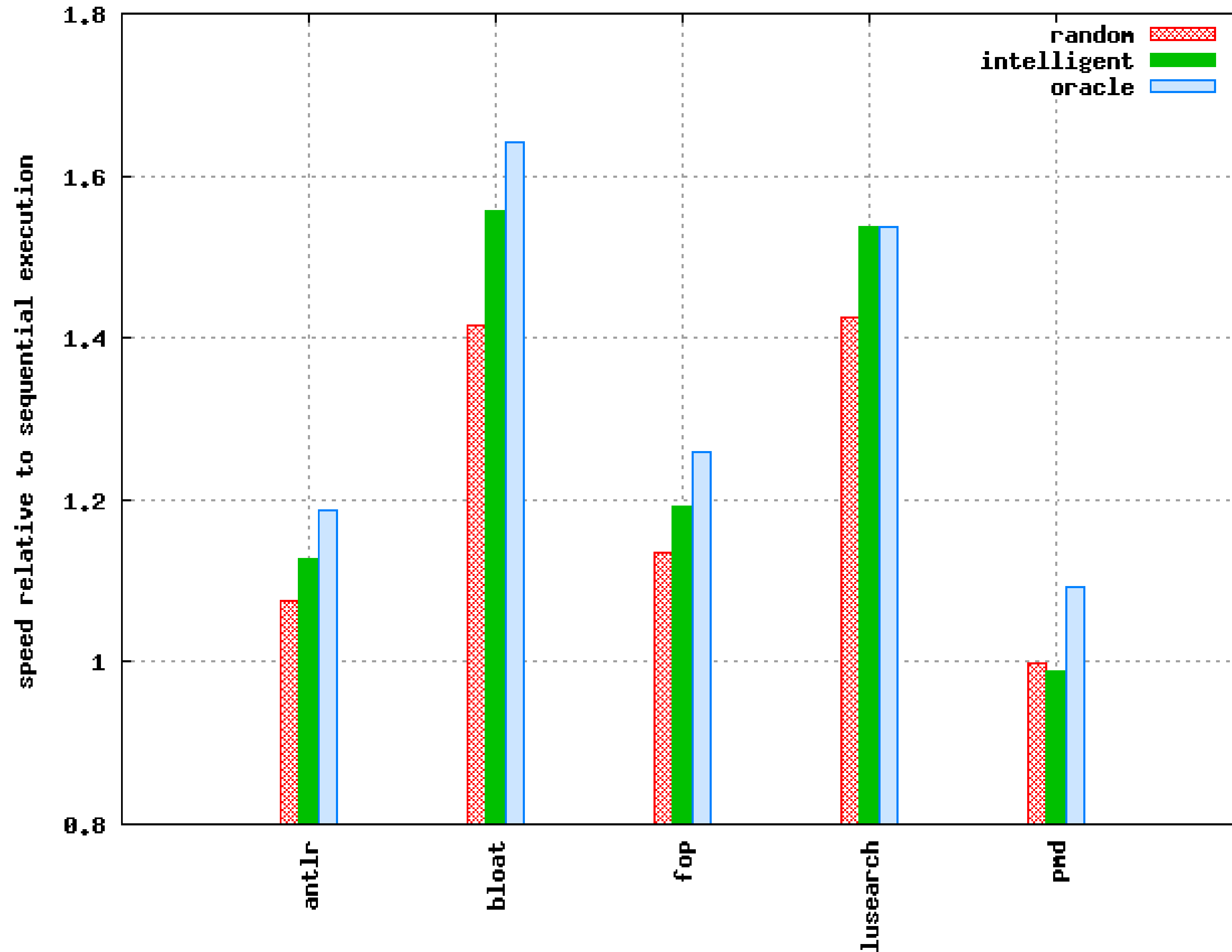
- Suggested learning algorithm – *association rule mining*

# Example rules using static features

- Caller void & local writer
- &&
- Callee type manipulator & looping
- Implies SQUASH

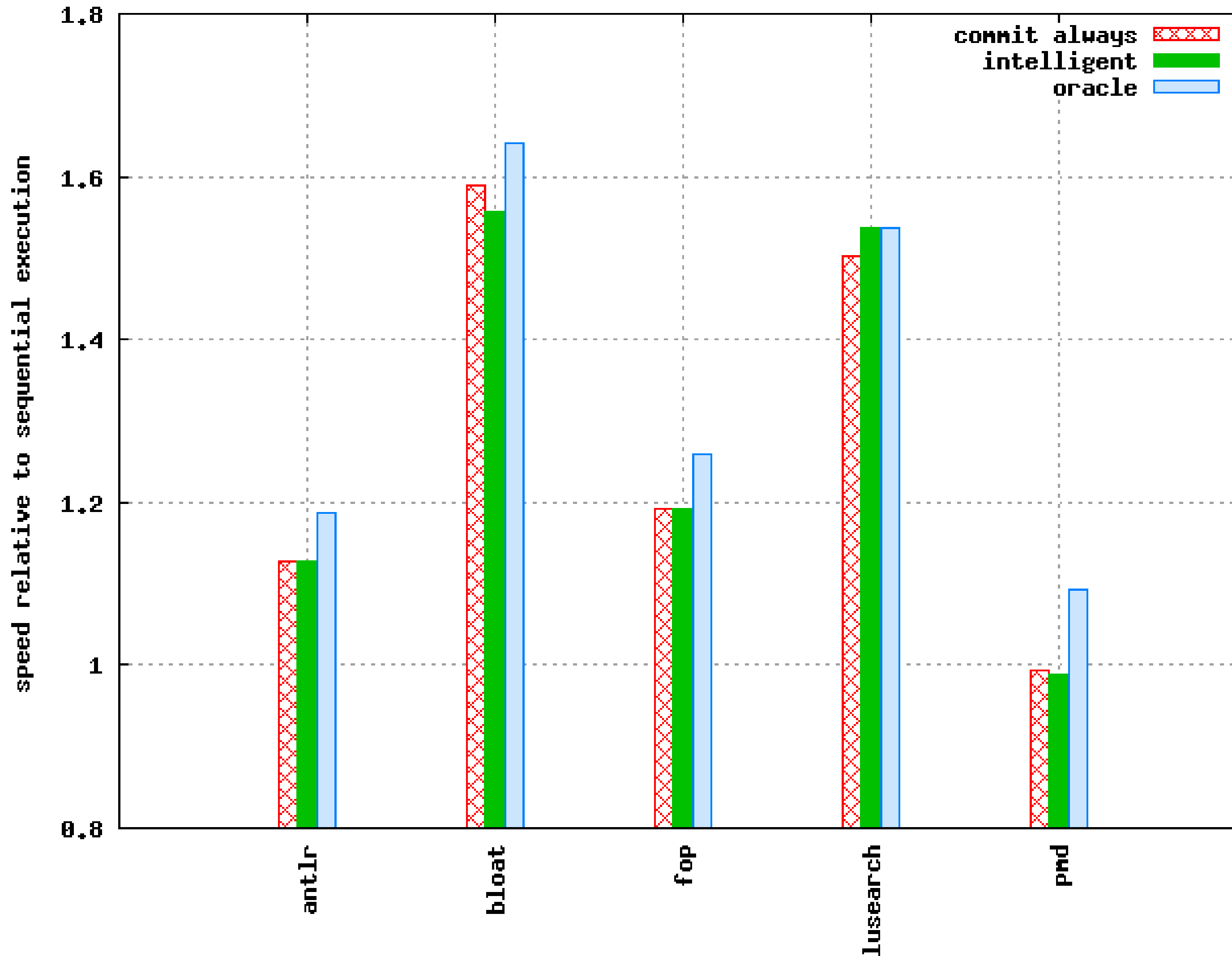
- ~ 100 rules for each benchmark
- Rank by confidence and support

# Performance of Rule-Based Squash Predictor (1)



# Performance of Rule-Based Squash Predictor (2)

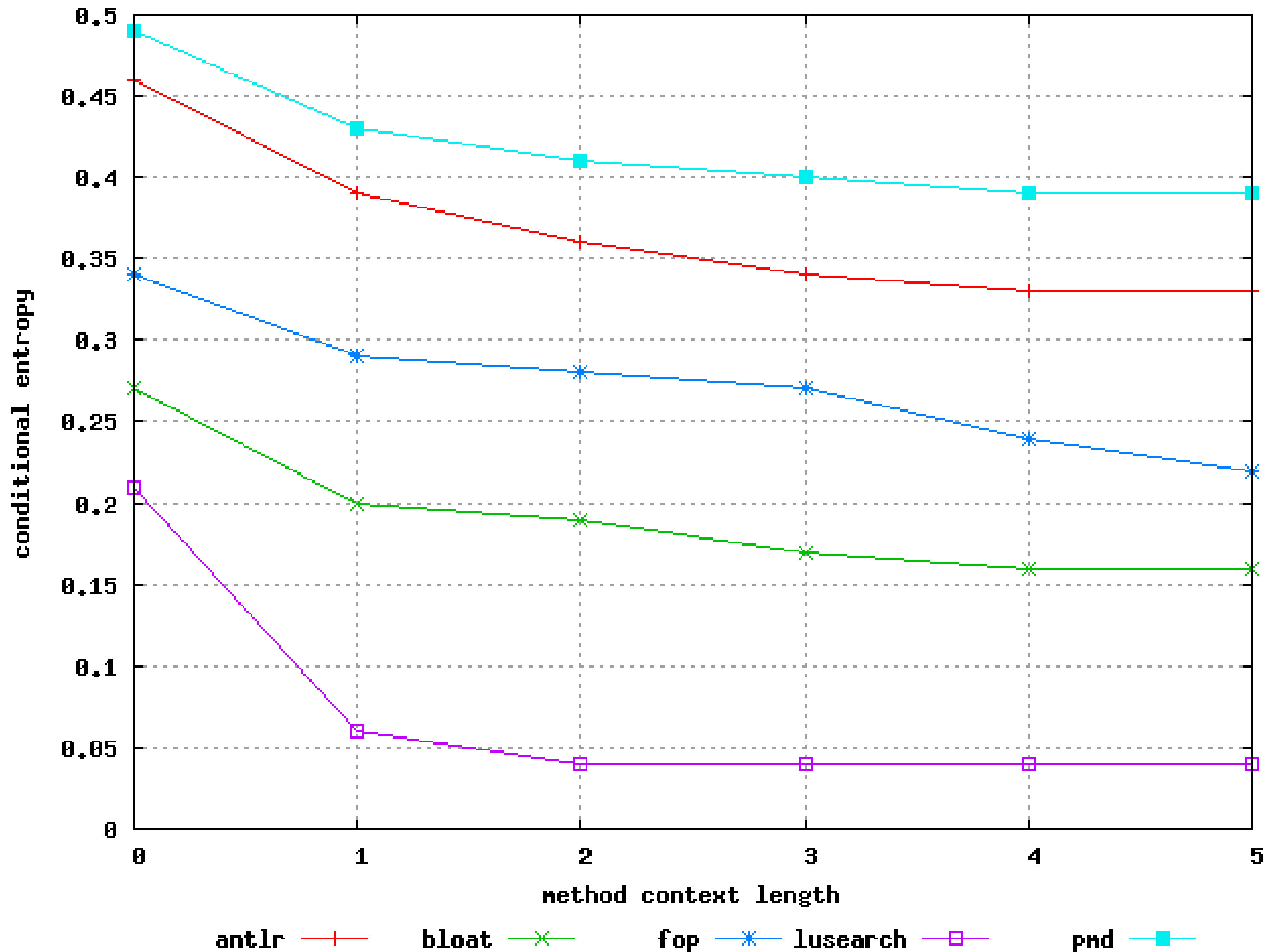
•



# Where are we up to?

- Now we are looking at dynamic features
- Issues with how often we update dynamic info for call sites

# Amount of Context Required



# Future work

- Hybrid prediction.
- Some static squash prediction
  - (cheap, sometimes not accurate)
  - to avoid inserting spawn instrs at compile time
- Some dynamic squash prediction
  - (expensive, hopefully more accurate)
  - to suppress spawn instrs at run time

# Conclusions

- TLS has many heuristics
- We aim to use machine learning to generate better heuristics
- Intelligent squash prediction – work in progress!