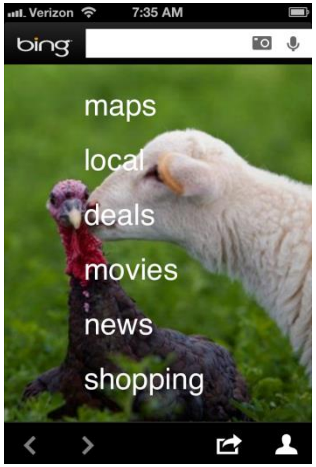
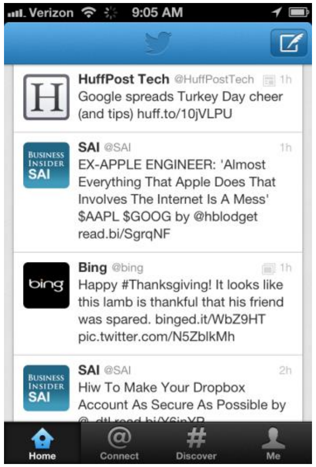
Mantis: Automatic Performance Prediction for Smartphone Applications

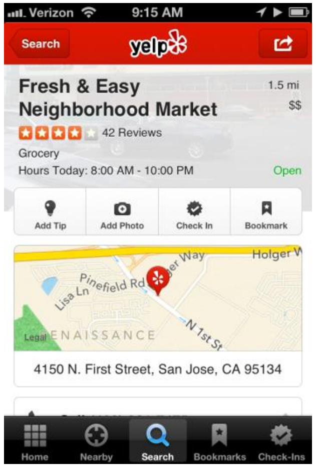
Byung-Gon Chun
Microsoft

Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Ling Huang, Petros Maniatis, Mayur Naik, Yunheung Paik
Seoul National University, UT Austin, Intel, Georgia Tech

Smartphone apps







Performance prediction problem

Predict the execution time of a program on a given input **before running it**

Two kinds of approaches

- Differentiated by features chosen to model program's performance
- Approach 1 : domain-specific program, automatically-extracted features
- Approach 2 : general-purpose program, manually-specified features

Performance predictor design dimensions

	Approach 1	Approach 2	Mantis
Applicability	X	0	0
Automation	0	X	0
Accuracy	△	△	0
Efficiency	△	△	0

Outline

- Motivation
- System overview
- Feature instrumentation
- Profiling
- Prediction modeling
- Predictor code generation
- Evaluation

Key insight of our approach

Program execution runs often contain **features** that **correlate** with **performance** and are **automatically computable** **efficiently**

Automatically computable

```
for (int i=0; i<n; ++i) {
  /* heavy computation */
}
```

Automatically computable

```
for (int i=0; i<n; ++i) {
  if ( a[i] == true ) {
    /* heavy computation */
  }
}
```

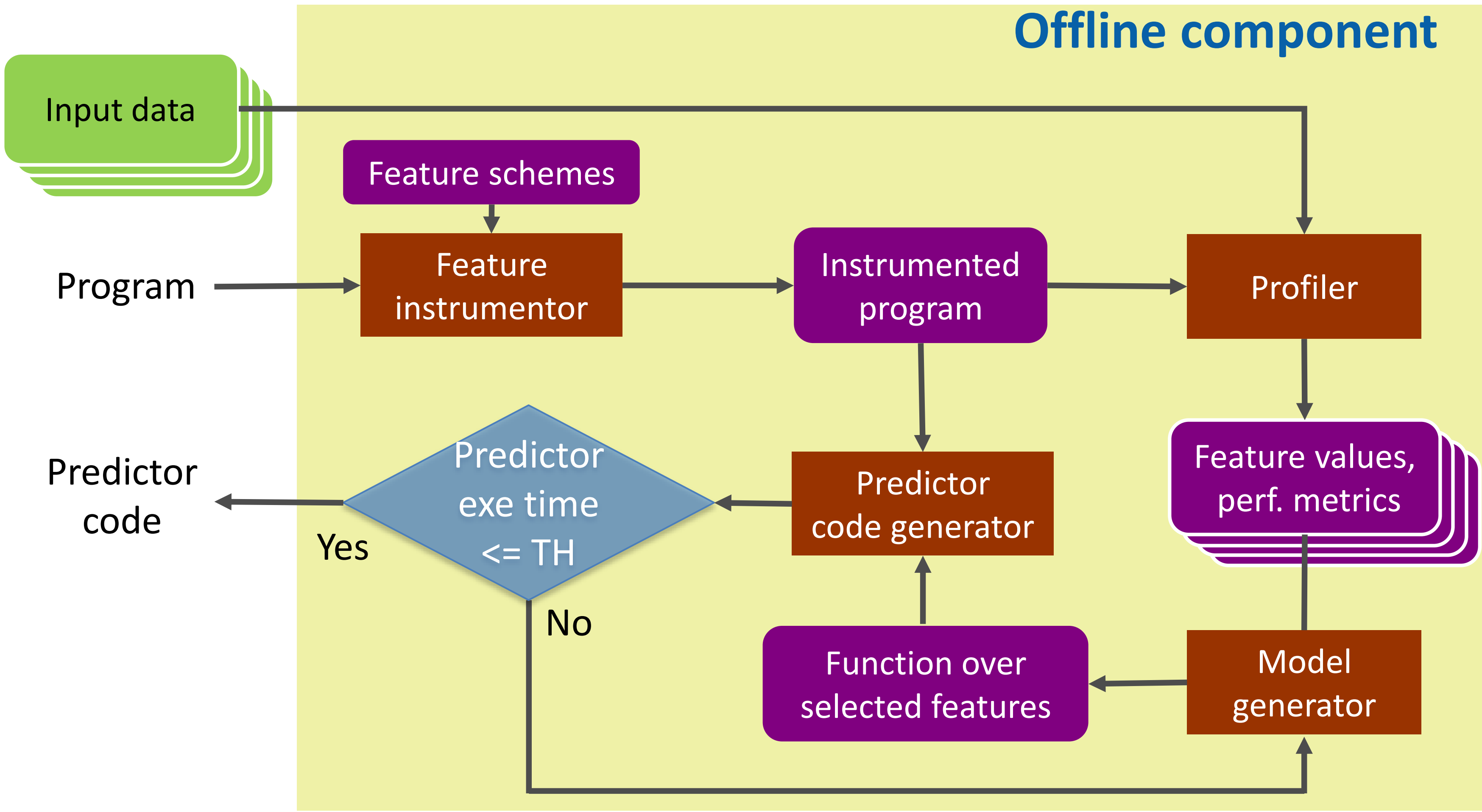
Cheaply computable

```
for (int i=0; i<n; ++i) {
  if ( a[i] == true ) {
    /* heavy computation */
  }
}
```

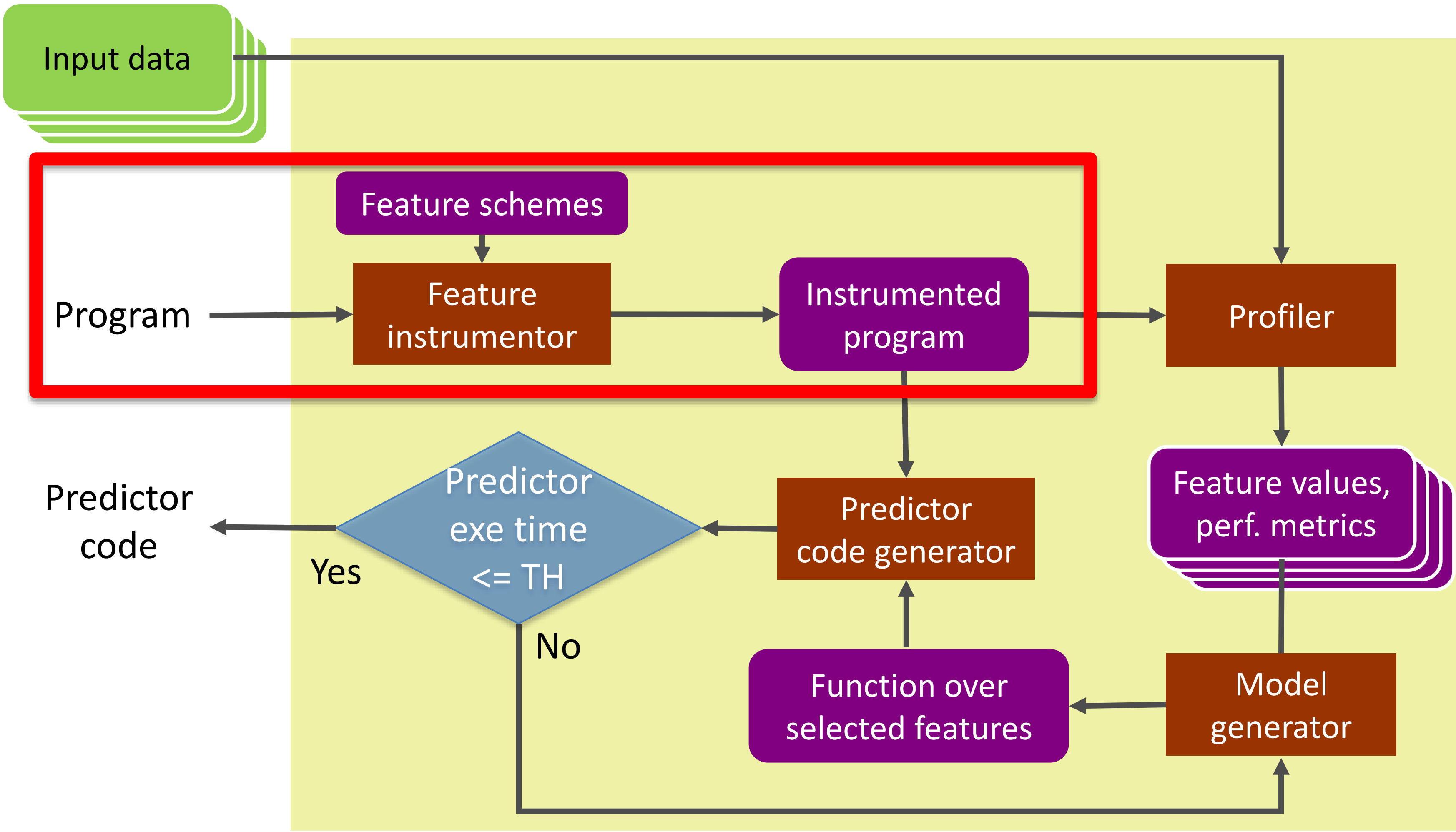
Key questions

- What are good program features for performance prediction?
- How do we model performance with relevant features?
- How do we compute features cheaply?
- How do we automatically generate predictor code?

System architecture



System architecture



Feature instrumentation

- Branch counts
- Loop counts
- Method call counts
- Variable values

Feature instrumentation

- Branch counts

```
// original code
if (flag) {

lightweightCompute();
} else {
  heavyCompute();
}
```

```
// instrumented code
if (flag) {
  ++mantis_branch_cnt1;
  lightweightCompute();
} else {
  ++mantis_branch_cnt2;
  heavyCompute();
}
```

Feature instrumentation

- Loop counts

```
// original code
while(line=readLine())
{
  search(line);
}
```

```
// instrumented code
while(line=readLine())
{
  ++mantis_loop_cnt;
  search(line);
}
```

Feature instrumentation

- Method call counts

```
// original code
process(String arg)
{
  ...
}
```

```
// instrumented code
process(String arg)
{
  ++mantis_method_cnt;
  ...
}
```

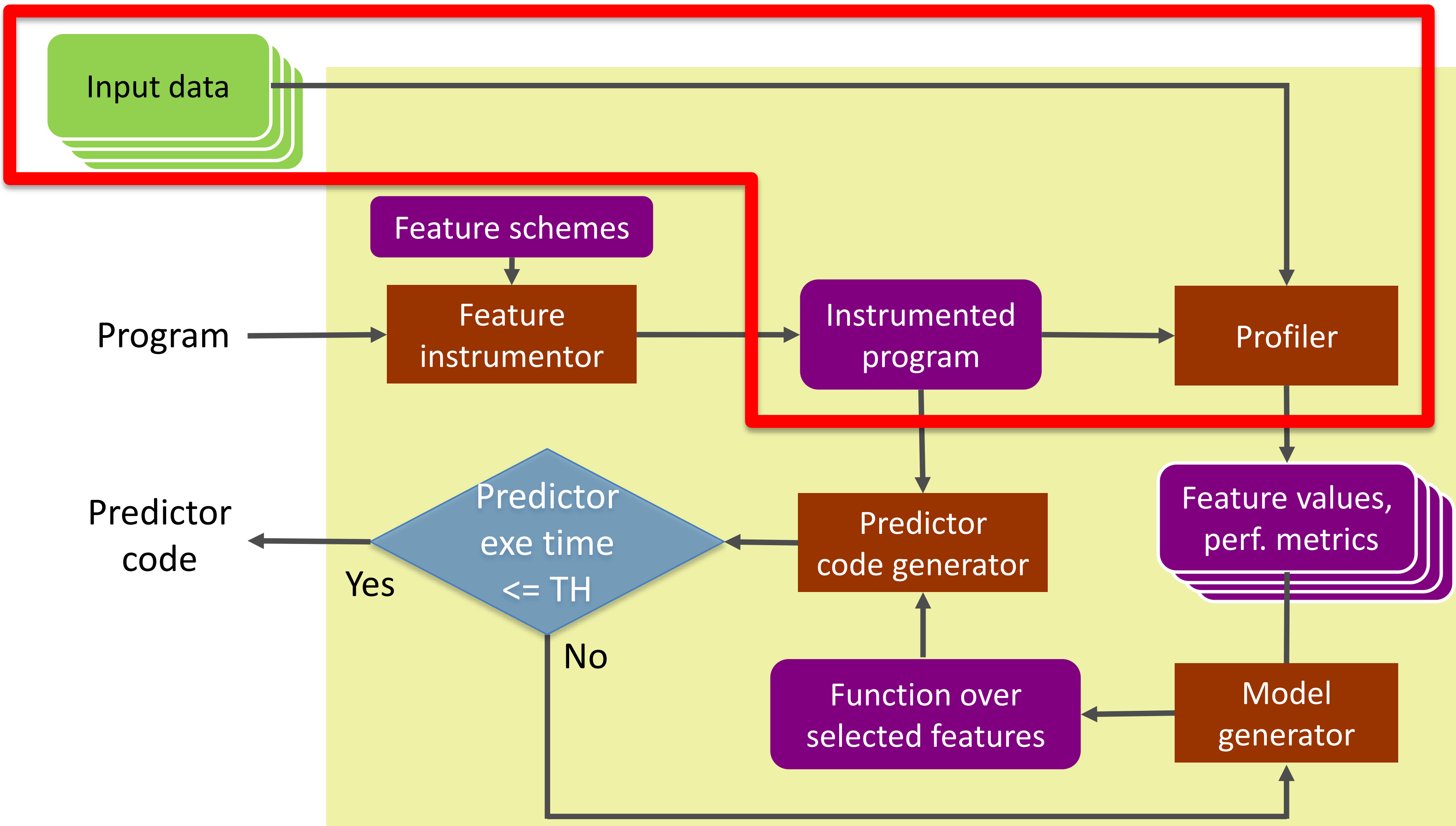
Feature instrumentation

- Variable values

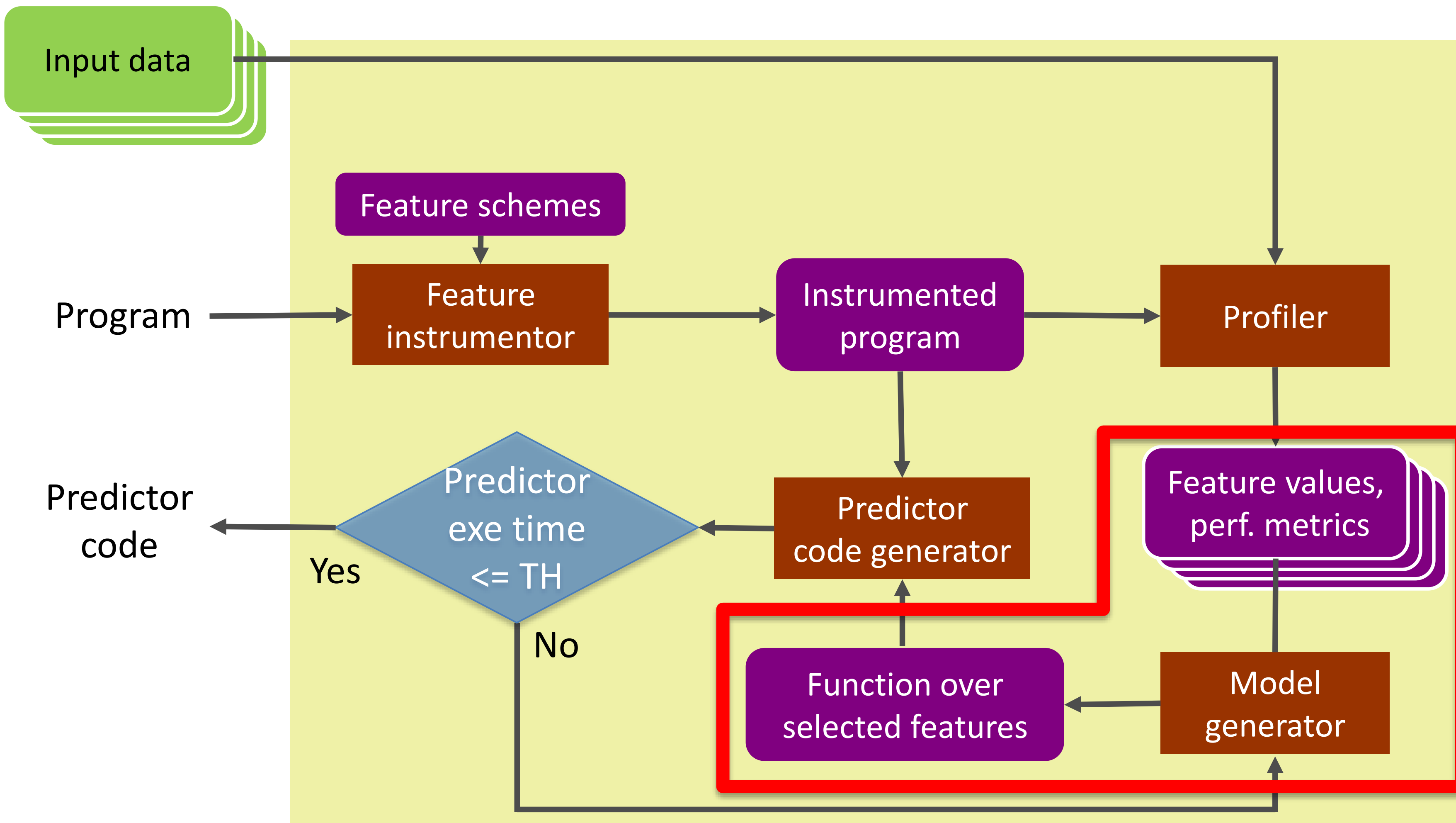
```
// original code
n=preprocess();
compute(n);
```

```
// instrumented code
n=preprocess();
mantis_n_sum += n;
++mantis_n_count;
compute(n);
```

System architecture



System architecture



Performance modeling

- Expect a small set of features to explain performance among lots of features
- SPORE-FoBa (Huang et al. 2010)

Performance modeling

```
SelectedFeatures = Prune_by_FoBa(Features)
// FoBa (Zhang 2008)
```

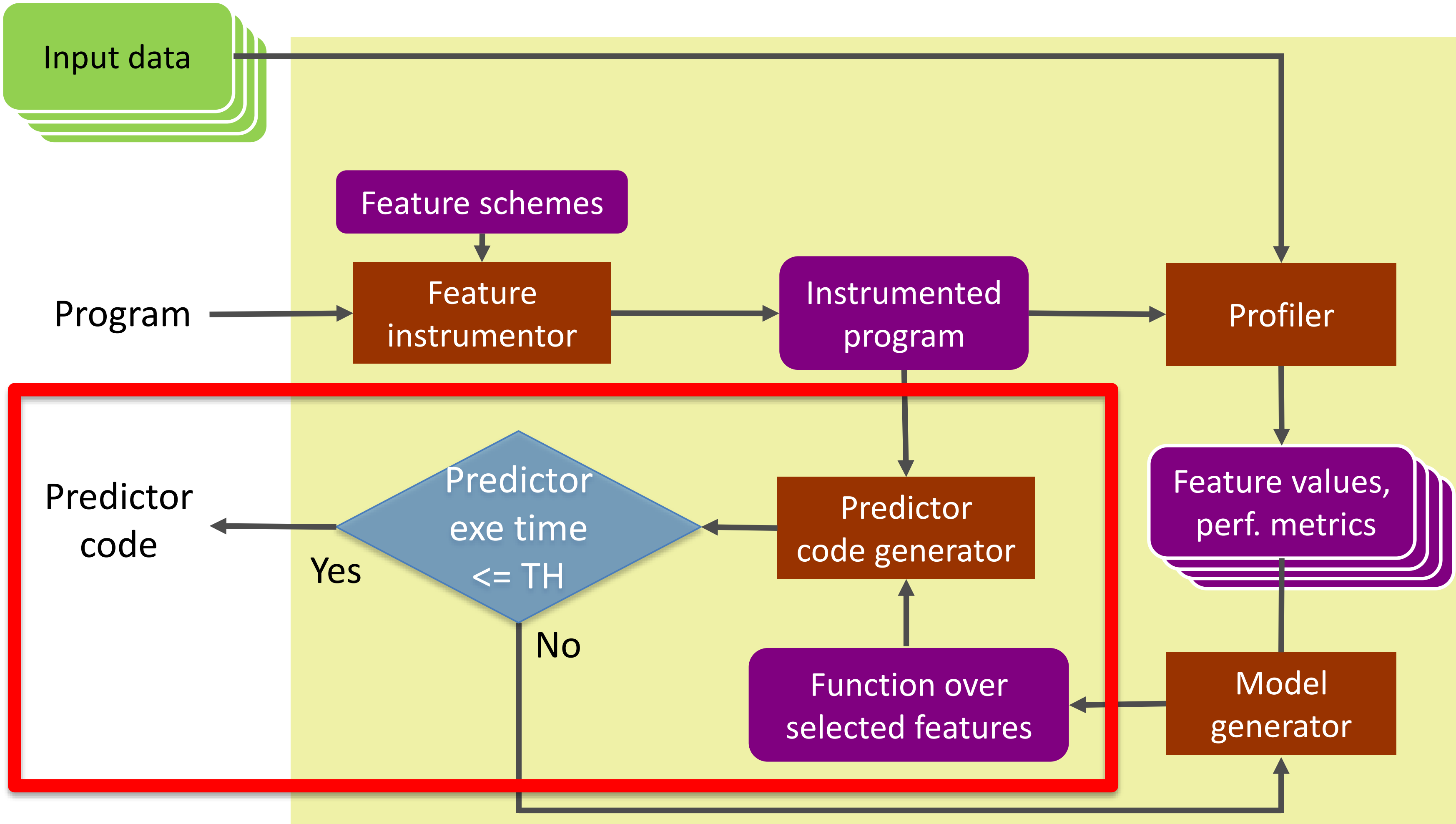
```
Terms = PolynomialExpansion(SelectedFeatures,
d)
// e.g., SelectedFeatures = {x1, x2}
```

// $(1 + x1 + x2)^2$ => 1, x1, x2, $x1^2$, $x2^2$, x1x2

// $\hat{y} = b_1 + b_2 x1 + b_3 x2 + b_4 x1^2 + b_5 x2^2 + b_6 x1x2$

```
PerfModel = Choose_by_FoBa(Terms)
```

System architecture



Predictor code generation: static program slicing

- A slice: a subprogram of the given program that yields the same value of variable v at program point p

program

```
int x;
if (b1) {
    x = 10;
} else {
    if (b2) {
        x = 20;
    } else {
        x = 30;
    }
}
x = 40;
Print(x);
```

slice

```
int x;
x = 40;
Print(x);
```

Predictor code generation: static program slicing

program

```
Reader r = new Reader(file);
String s;
while((s = r.readLine()) != null) {
    mantis_loop_cnt++; // feature inst
    process(s);        // expensive comp
}
```

slice

```
Reader r = new Reader(file);
String s;
while((s = r.readLine()) != null) {
    mantis_loop_cnt++; // feature inst
}
```

Predictor code generation: static program slicer challenges

- Inter-procedural analysis
- Alias analysis
- Concurrency analysis
- Executable slices

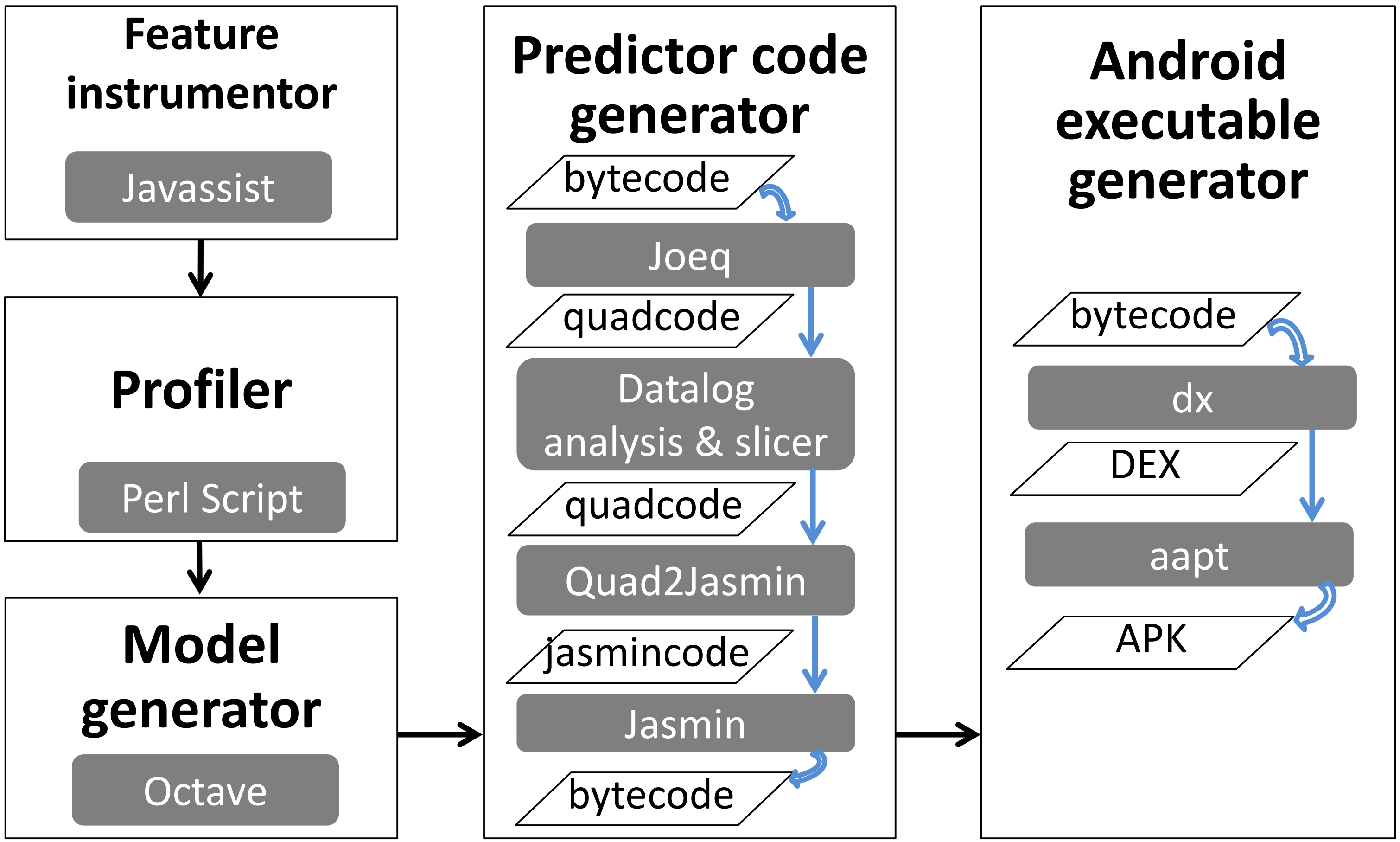
Predictor code generation

- Intraprocedural: construct Program Dependency Graphs (PDGs) (HRB 1988)
- Interprocedural: construct a System Dependency Graph (SDG) (HRB 1988)
- Context sensitivity: augment the SDG with summary edges by running the SummaryEdge algorithm (RHS+ 1994)
- Run the two-pass reachability algorithm on the augmented SDG (HRB 1988)
- Translate intermediate code to final code

Outline

- Motivation
- System overview
- Feature instrumentation
- Profiling
- Prediction modeling
- Predictor code generation
- Evaluation

Mantis prototype



Evaluation

- Prediction accuracy

Prediction error =
|ActualTime –PredictedExecutionTime| / ActualTime

Prediction time =
Predictor_Running_Time / ActualTime

- Prediction under background load
- Mantis offline component processing time

Evaluation

- Prediction accuracy
  - Benefit of non-linear terms
  - Benefit of slicing
- Predictor execution time
  - Benefit of slicing
- Prediction on different hardware platforms
- Prediction under background load
- Mantis offline stage processing time

Experiment setup

- Applications: Encryptor, Path Routing, Spam Filter, Chess Engine, Ringtone Maker, and Face Detection
- Galaxy Nexus running Android 4.1.2

- 1000 randomly generated inputs for each application : 95-100% basic-block coverage
- 100 inputs for training
- 5% : predictor execution time threshold

Prediction error and time

Application	Prediction error (%)	Prediction time (%)
Encry		
Path F		
Spam		
Chess		
Ringtone Maker	2.2	0.20
Face Detection	4.9	0.62

2.2-11.9% error by executing predictor costing at most 1.3% of app execution time

Prediction error and time

Application	Prediction error (%)	Prediction time (%)	No. of detected features	No. of chosen features
Encryptor	3.6	0.18	28	2
Path Routing	4.2	1.34	68	1
Spam Filter	2.8	0.51	55	1
Chess Engine	11.9	1.03	1084	2
Ringtone Maker	2.2	0.20	74	1
Face Detection	4.9	0.62	107	2

Benefit of slicing
Baselines: PE and BE

- Partial Execution (PE) :
  runs the instrumented program only until we obtain the chosen feature values

- Bounded Execution (BE) :
  runs the instrumented program for amount of time the Mantis predictor runs

Mantis vs. Partial Execution (PE)

Application	Mantis pred. time (%)	PE pred. time (%)
Encryptor	0.20	100.08
Path Routing	1.30	17.76
Spam Filter	0.50	99.39
Chess Engine	1.03	69.63
Ringtone Maker	0.20	0.04
Face Detection	0.61	0.17

Mantis vs. Bounded Execution (BE)

Application	Mantis pred. error (%)	BE pred. error (%)
Encryptor	3.6	56.0
Path Routing	4.2	64.0
Spam Filter	2.8	36.2
Chess Engine	11.9	26.1
Ringtone Maker	2.2	2.2
Face Detection	4.9	4.9

Related work

- Predicting performance or resource consumption in databases, cluster computing, networking, program optimization, etc.
- Non-trivial features: program complexity, hardware simulation specificity, cooperative bug finding
- Worst-case behavior prediction in embedded/real-time systems

Conclusion

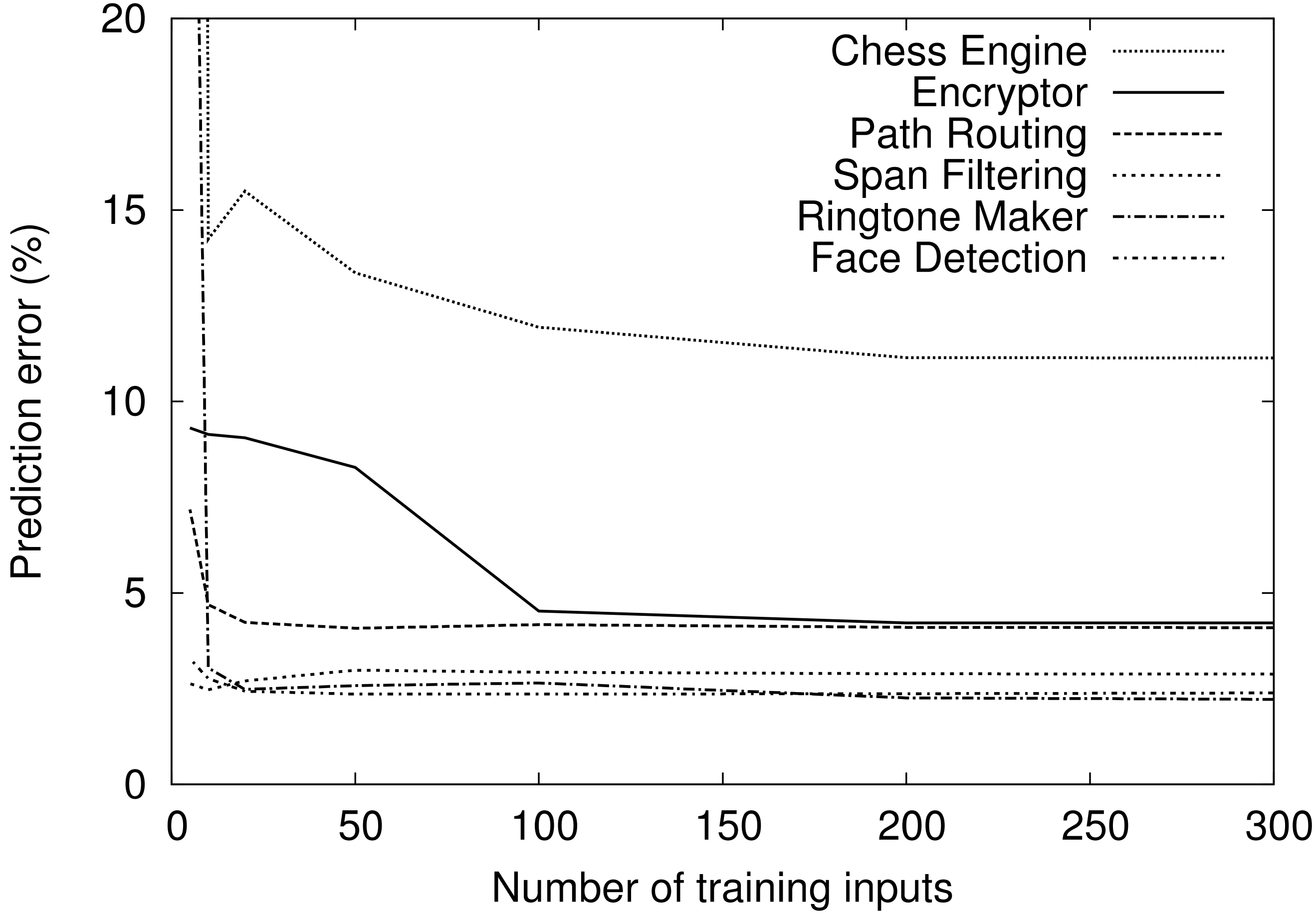
- Mantis: a framework that automatically generates accurate and efficient program performance predictors
  - Extracts information from program executions
  - Models performance with machine learning
  - Generates predictors with program analysis
  - Uses even features that occur late in execution

Backup Slides

Selected features and generated models

Application	Selected features	Generated model
Encryptor	matrix-key size(f_1) loop count of encryption (f_2)	$c_0f_1^2f_2 + c_1f_1^2 + c_2f_2 + c_3$
Path Routing	build map loop count (f_1)	$c_0f_1^2 + c_1f_1 + c_2$
Spam Filter	inner loop count of sorting (f_1)	$c_0f_1 + c_1$
Chess Engine	no. of second-level game-tree nodes (f_1), no. of chess pieces (f_2)	$c_0f_1^3 + c_1f_1f_2 + c_2f_2^2 + c_3$
Ringtone Maker	cut interval length (f_1)	$c_0f_1 + c_1$
Face Detection	width (f_1), height (f_2)	$c_0f_1f_2 + c_1f_2^2 + c_2$

Prediction errors varying the number of input samples



Prediction error and time of Mantis running with Galaxy S2 and Galaxy S3

Application	Galaxy S2		Galaxy S3	
	Prediction error (%)	Prediction time (%)	Prediction error (%)	Prediction time (%)
Encryptor	4.6	0.35	3.4	0.08
Path Routing	4.1	3.07	4.2	1.28
Spam Filter	5.4	1.52	2.2	0.52
Chess Engine	9.7	1.42	13.2	1.38
Ringtone Maker	3.7	0.51	4.8	0.20
Face Detection	5.1	1.28	5.0	0.69

Prediction error under background CPU-intensive loads

Application	Mantis pred. error (%) for the x% background CPU load			
	x=0	x=50	x=75	x=99
Encryptor	3.6	7.5	10.5	21.3
Path Routing	4.2	5.3	5.8	6.7
Spam Filter	2.8	4.7	5.2	5.8
Chess Engine	11.9	13.5	15.3	15.8
Ringtone Maker	2.2	2.3	3.0	3.1
Face Detection	4.9	5.3	5.6	5.8

Predictor code generation: static program slicer challenges

- Inter-procedural analysis
  - Context-sensitive inter-procedural algorithm
- Alias analysis
  - Flow- and context-insensitive may-alias analysis with object allocation site heap abstraction
- Concurrency analysis
  - May-alias
- Executable slices
  - A set of rules we identified

Mantis offline stage processing time (in seconds)

Application	Prof.	Model gen.	Slicing	Test	Total	Iter.
Encryptor	2373	18	117	391	2900	3
Path Routing	363	28	114	14	519	3
Spam Filter	135	10	66	3	214	2
Chess Engine	6624	10229	6016	23142	46011	83
Ringtone Maker	2074	19	4565	2	6659	1
Face Detection	1437	13	6412	179	8041	4