Cross-Input Learning and Discriminative Prediction in Evolvable Virtual Machines

Feng Mao, Xipeng Shen

Computer Science Department
The College of William and Mary, Williamsburg, VA
A Bird vs A Program

- Commons?
  (Both may crash.)
  Both do computation.

- Which one is smarter?
  A bird learns to fly better & better.
  Can a program learn to run faster & faster?
Program Optimizations

- Still, the 1 millionth run is as fast as the first-time run
- Programs are “dumb”
Dynamic Optimizations

Observe → Behavior → Optimize

Adaptive to runtime behaviors

Widely used in Java, C#, etc.
Adaptivity vs Proactivity

Adaptivity

- dynamic optimization

Proactivity

- offline prof.
- static compilation
Drawbacks of Reactivity

Delays in optimizations

Inferior decisions for local view

Limited analysis & transformation

Significant influence on performance [Arnold+:OOPSLA05, Gu+:CGO08]
Objective

Adaptivity

Proactivity

dynamic optimization

proactive dynamic opt.

offline prof.

static compilation
Proactive Dynamic Optimization

- Avoid delay in optimizations
- Optimize over global view
Proactive Dynamic Optimization

• Core: Input-Behavior Models

  \[ \text{Behavior} = f(\text{Input}) \]

• Strategy

  Input-centric learning cross production runs
  ---programs evolve through life.

• Challenges
  • Input complexity
  • Construction and uses of \( f() \)
Outline

Proactive dynamic optimization

- Input characterization
- Incremental learning
- Discriminative prediction
- Evaluation
Outline

- Input characterization
- Incremental learning
- Discriminative prediction
- Evaluation

Proactive dynamic optimization
Input Characterization

To extract important features from raw inputs
Challenges

- Input attributes rather than values matter
  - e.g., data distribution
- Complex input syntax & semantics
  - e.g., a graph or a tree or a signal
- Interplay among input components
  - overshadow, equivalence, default values, etc.

Domain knowledge needed; automatic solutions are difficult.
XICL-Based Synergy

- eXtensible Input Characterization Language

Diagram:
- arbitrary input
- XICL Translator
- feature vector

XICL spec by programmers

The College of William and Mary
XICL

- A mini-language
- Two primary constructs
  - Options
  - Operands
- Extensible
  - Allowing programmer-defined feature extraction methods

```
gcc -c -o main.o main.c
```
XICL Example

SYNOPSIS: route [options] FILE

OPTIONS:
- -e, --echo: print intermediate results. It is off by default.
- -n NUM: find NUM shortest paths. NUM is 1 by default.

Users defined feature:
public class mEDG
implements XFMethod () {
    ...
}

Example command-line:
route -n 3 graph1
where, graph1 contains 100 nodes and 1000 edges.

Feature vector produced by XICL translator:
(0, 3, 100, 1000)
XICL Translator

- Implemented in Jikes RVM 2.9.1
- Extensibility support
  - Instantiation of interface `XFMethod`.
- Efficiency support
  - Use of variable values as input features
- Interactivity support
  - Interface for handling interactive applications

Details in the paper.
Outline

- Input characterization
- Incremental learning
- Discriminative prediction
- Evaluation

Proactive dynamic optimization
Construction of Input-Beh Model

Behavior = f (Input)

• Behavior
  • Program-level behaviors
  • Optimization decisions
    • High-level
      • Compilation levels, inlining, etc.
    • Low-level
      • Unrolling levels, tile size, etc.
Target in This Work

- JIT compilation levels in Jikes RVM
- Four possible levels

-1 0 1 2

more sophisticated
more overhead

Default scheme

- -1 at the first encounter of a method
- Higher-level recompilation if it is hot
- Hotness attained through runtime sampling
Incremental Learning

• Goal
  To transparently build $f()$ across production runs:
  
  \[ \text{compilation level} = f(\text{input features}) \]

• Technique: Classification Trees (CT)
Example CT

X3 does not appear in the tree
Advantages of CT

- Automatic feature selection
- Handles both discrete and numeric features
- Efficient to build and use
- Good interpretability
Proactive Dynamic Optimization

$\text{Behavior } = f(\text{Input})$

- But prediction could be wrong...
Outline

- Proactive dynamic optimization
- Input characterization
- Incremental learning
- Discriminative prediction
- Evaluation
Discriminative Prediction

- Predict only when confident
- Confidence of input-behavior models
  - Decayed average of prediction accuracy

\[ conf = (1 - \gamma) \times conf + \gamma \times acc \]

Details in the paper.
Evaluation

- Jikes RVM 2.9.1
- Intel Xeon E5310 with Linux 2.6.9
- Benchmarks
  - 11 programs from 3 suites
  - 5 to 62 inputs per program
<table>
<thead>
<tr>
<th>Programs</th>
<th># inputs</th>
<th>Running time (s)</th>
<th>Input features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>compress</td>
<td>20</td>
<td>0.94</td>
<td>9.33</td>
</tr>
<tr>
<td>db</td>
<td>54</td>
<td>0.59</td>
<td>98.16</td>
</tr>
<tr>
<td>mtrt</td>
<td>62</td>
<td>0.26</td>
<td>6.37</td>
</tr>
<tr>
<td>antlr</td>
<td>65</td>
<td>0.15</td>
<td>0.19</td>
</tr>
<tr>
<td>bloat</td>
<td>55</td>
<td>0.08</td>
<td>41.46</td>
</tr>
<tr>
<td>fop</td>
<td>57</td>
<td>0.59</td>
<td>1.53</td>
</tr>
<tr>
<td>euler</td>
<td>6</td>
<td>0.93</td>
<td>7.79</td>
</tr>
<tr>
<td>moldyn</td>
<td>12</td>
<td>0.11</td>
<td>63.05</td>
</tr>
<tr>
<td>montecarlo</td>
<td>14</td>
<td>9.07</td>
<td>15.81</td>
</tr>
<tr>
<td>search</td>
<td>5</td>
<td>2.74</td>
<td>210.36</td>
</tr>
<tr>
<td>raytracer</td>
<td>12</td>
<td>3.10</td>
<td>236.57</td>
</tr>
</tbody>
</table>
Results on Mtrt
Results on $Mtrt$

![Graph showing results on Mtrt with markers for prediction accuracy and confidence over different numbers of runs.](image-url)
Results on $Mtrt$

Number of runs

pred acc
confidence
speedup

The College of William and Mary
Job Length vs Speedup (Mtrt)

Benefits

• Avoid recompilations
• Generate efficient code early

Both are more obvious on relatively long runs
Comparison

• Repository-based optimization (Rep) [Arnold+: OOPSLA’05]
  • Build a repository across runs
  • Produce an optimization strategy per method to maximize the average performance of history runs
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Evolve</th>
<th>Rep</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-run learning</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Input-specific optimizations</td>
<td>Yes</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>Avoidance of Recompilation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Proactivity</td>
<td>Yes</td>
<td>Weak</td>
<td>No</td>
</tr>
</tbody>
</table>
Speedup Comparison

Benchmarks

Mirt  Compress  Db  Antlr  Bloat  Fop  Euler  MolDyn  MonteCarlo  Search  RayTracer

Evolve  Rep

The College of William and Mary
Conclusion

- Proactive dynamic optimizations
- Core: Input-Centric Behavior Analysis
- Learn cross runs incrementally & transparently
- Program evolves continuously
Thanks!