A Step Towards Transparent Integration of Input-Consciousness into Dynamic Program Optimizations

Kai Tian, Eddy Z. Zhang, Xipeng Shen

Computer Science Department

The College of William and Mary
Program Optimizations

1950s
Static

1980s
Profile

1990s
Dynamic
[Java, C#, etc.]

1990s
[Java, C#, etc.]

- JIT in JikesRVM

```
while (...) {
    foo();
}
```

input

opt(-1)

foo()

foo()

opt(0)

foo()

foo()

opt(1)

10--82% speedup on 10 Java programs.
Building Up the “Magic Crystal Ball”

[OOPSLA’10]

A statistical learning process on 100s of offline profiling runs.

A main barrier for practical usage.

input feature vector $\mathbf{v}$

$\langle$ feature 1, feature 2, ..., feature k $\rangle$

Predictive model construction

input-behavior models

$\{ \text{beh}_1 = f_1(\mathbf{v}); \text{beh}_2 = f_2(\mathbf{v}); \ldots \}$

raw input

Input characterization

uses cross-loop correlations
Building Up the “Magic Crystal Ball”

[OOPSLA’10]

Goal of This Work

1. Folding whole process into production runs transparently.
2. Creating an auto-evolving input-conscious dynamic optimization system.

Input

raw input feature vector \( v = \langle \text{feature 1}, \text{feature 2}, ..., \text{feature k} \rangle \)

raw input behavior models

\{ \text{beh}_1 = f_1(v); \text{beh}_2 = f_2(v); ... \}

Predictive model construction

A statistical learning process on 100s of offline profiling runs.

A main barrier for practical usage.
Prior Sampling Schemes

- Interval-based sampling
  - E.g., Arnold+:PLDI02 [yielding points in Jikes RVM]
- Bursty sampling
  - E.g., Hirzel+:FDDO01
- Bernoulli sampling
  - E.g., Liblit+:PLDI03

- Not suitable for obtaining absolute frequency (e.g., loop tripcounts)
- Weak overhead guarantee
  - Only consider overhead from instrumented code
  - But ignore instrumentation itself or checks
Data Collection: Underlying Vehicle

- Cross-user cross-run sampling [Liblit+:PLDI’03]
Overhead Sources

- Data to collect
  - Loop tripcounts, key variables (interface variables) values, etc.

```java
while (...){
    tripCnt[l]++;
    ...
    ...
}
```

<table>
<thead>
<tr>
<th></th>
<th>Base compiler/interpreter</th>
<th>Optimizing compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capable to instrument</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Overhead</td>
<td>Low</td>
<td>High (e.g. 23X)</td>
</tr>
</tbody>
</table>

- Overhead sources
  - Execution of the inserted counter-increase instruction
  - Extra compilation time incurred by optimizing compiler

K. Tian, Z. Zhang, X. Shen

College of William and Mary
main(){
    ...
    foo();
}

foo(){
    while (...){
        ...
    }
    while (...){
        ...
    }
}

Need method info to estimate compilation overhead.
But it is unknown before a compilation.

Cross-run info.

Need loop trip-counts to estimate execution overhead.
But they are unknown before execution.

Conservativeness.
**Inspection-Instrumentation**

When a method is loaded.

- **Size known from prior runs?**
  - **Y**
    - \( C_j = \text{estCompileOverhead}(S)/\text{exTime}; \]
    - \( \text{totalCost} + C_j < H? \]

  - **N**
    - No instrumentation. Default compilation.
    - Save its size.

- **N**

  - Let \( L_i \) be the smallest loop such that \( \text{totalCost} + 1/S(L_i) < H \).
  - Instrument \( L_i \) and all loops larger than it. \( \text{totalCost} = +1/S(L_i) \).

**Proposition:**
If loop \( L_i \) and all loops larger than \( L_i \) are instrumented, the incurred execution overhead must be less than \( 1/S(L_i) \), regardless of loop trip-counts.

\[ \text{totalCost: global, initially 0.} \]
\[ \text{exTime: estimated execution time.} \]
Proposition:
If loop $L_i$ and all loops larger than $L_i$ are instrumented, the incurred execution overhead must be less than $1/S(L_i)$, regardless of loop trip-counts.
Inspection-Instrumentation

Size known from prior runs?
No instrumentation.
Default compilation.
Save its size.

Proposition:
If loop \( L_i \) and all loops larger than \( L_i \) are instrumented, the incurred execution overhead must be less than \( 1/S(L_i) \), regardless of loop trip-counts.

\[ NC_j = \text{estCompileOverhead}(S)/\text{exTime}; \]
\[ \text{totalCost} + C_j < H? \]
\[ \text{totalCost} += C_j; \]

Compile with optimizing compiler, with loops selectively instrumented.

Let \( L_i \) be the smallest loop such that \( \text{totalCost} + 1/S(L_i) < H \).

Instrument \( L_i \) and all loops larger than it. \( \text{totalCost} = +1/S(L_i) \).

\( \text{totalCost} \): global, initially 0.
\( \text{exTime} \): estimated execution time.

• Cross-run info and conservative estimation are the key.
• Obtaining accurate profile with guaranteed efficiency.

Shortest history time with outliers filtered.

When a method is loaded:

- Cross-run info and conservative estimation are the key.
- Obtaining accurate profile with guaranteed efficiency.
Randomization

• Scheme
  • Each user has a $skipN$, with a random initial value.
  • After each run
    • $skipN = (skipN + instrumentedN) \mod totalMethods$
  • The first $skipN$ methods are simply excluded from instrumentation.

• Effects
  • Maximize overall coverage rate by cross-user synergy
  • Maximize each user’s coverage by systematic profiling
Enhancing Sparsity Tolerance

- Input characterization through cross-loop correlation analysis

<table>
<thead>
<tr>
<th></th>
<th>run$_1$</th>
<th>run$_2$</th>
<th>run$_3$</th>
<th>run$_4$</th>
<th>run$_5$</th>
<th>run$_6$</th>
<th>run$_7$</th>
<th>run$_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop$_1$</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>loop$_2$</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>loop$_3$</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

- Correlation propagation

\[
\begin{align*}
\text{loop}_1 \text{ correlates } \text{loop}_3 \\
\text{loop}_2 \text{ correlates } \text{loop}_3
\end{align*}
\Rightarrow \text{loop}_1 \text{ correlates } \text{loop}_2
\]
Input-Conscious Evolvable Optimization System

- Cross-run continuous data collection and model evolvement
- Optimizing with confidence
  - Confidence of predictive models: cross validation
  - Selective prediction and optimization
Evaluation

- Methodology
  - Execution setting
    - Emulating 100 virtual users
    - Randomly selecting an input for each run
  - Platform
    - Xeon E5310
    - JikesRVM 3.1.0, Linux 2.6.22, -Xmx512m
  - Benchmarks
    - 18 benchmarks from Dacapo, JVM98, Grande
    - 58 inputs per program on average
Overhead

[ 20 runs each setting on a random input ]

Exec. time

Difference is significant only when P-value<0.05.
Cross-Run Evolvement

![Graphs showing accuracy, confidence, and speedup over number of runs for different inputs and optimizations.]

- Accuracy
- Confidence
- Speedup

Input:
- Temporarily set to a value of 1.
- Foo() function with parameters depending on the specific optimization (opt(1), foo()).
Overall Speedup Comparison

- Input-oblivious cross-run [Arnold+:OOPSLA05]
- Offline-prof input-aware [Tian+:OOPSLA10]
- Our method
Future Work

- Management of data communication and profiles
- Differences in platforms and libraries
- Software update
- Server applications
Related Work

• **Sampling**
  - [Arnold+:PLDI02, Hirzel+:FDDO01, Liblit+:PLDI03, Bond+:PLDI10]

• **Cross-run program optimizations**
  - CPO framework [Wisniewski+: PAC2’04]
  - Continuous compilation - CoCo [Childers+: NSFNGS’03]
  - Repository based optimization [Arnold+: OOPSLA’05]

• **Input based optimizations**
  - Manual spec (XICL) to capture input features [Mao+: CGO’09]
  - Input-centric framework [Tian+: OOPSLA’10]
  - Manual specify input-features and adapt to inputs
    - Machine learning based compilation [Leather+: CGO’09]
    - Adaptive Sorting [Li+: CGO04]
    - Library and frameworks: STAPL [Thomas+: PPOPP’05]
Related Work

• Sampling
  • [Arnold+: PLDI'02, Hirzel+: FDDO'01, Liblit+: PLDI'03, Bond+: PLDI'10]

• Cross-run program optimizations
  • CPO framework [Wisniewski+: PAC'04]
  • Continuous compilation - CoCo [Childers+: NSFNGS'03]

• Repository based optimization
  • [Arnold+: OOPSLA'05]

• Input based optimizations
  • Manual spec (XICL) to capture input features [Mao+: CGO'09]
  • Input-centric framework [Tian+: OOPSLA'10]
  • Manual specify input-features and adapt to inputs
    • Machine learning based compilation [Leather+: CGO'09]
    • Adaptive Sorting [Li+: CGO'04]
    • Library and frameworks: STAPL [Thomas+: PPOPP'05]

New sampling scheme with strong overhead control and accurate profiling results.

First input-aware optimization system that evolves continuously, confidently, and transparently.
Take-Aways

• It is feasible to obtain accurate profiles with strong efficiency guarantees in production runs.
  • Opens the doors to post-deployment enhancement

• Completely transparent input-aware, continuous optimizations are feasible and beneficial.
Thanks!