The Future Begins Here

intel® labs

Labs Day 2020 | December 3
A Continuing Shift from People Effort to Machine Effort

- **1950**: Machine code
- **1960**: Low-level systems languages
- **1970**: Personal computers, systems languages
- **1980**: Software & hardware ecosystems, datacenters, design patterns
- **1990**: Machine programming
- **2000**: Machine programming
- **2010**: Machine programming
- **2020**: Machine programming
The Three Pillars of Machine Programming (MP)

MP is the automation of software development

- **Intention:** Discover the intent of a user
- **Invention:** Create new algorithms and data structures
- **Adaptation:** Evolve in a changing hardware/software world

"THE Three Pillars of Machine Programming" (Joint with Intel Labs and MIT)
Building Software is Difficult

<table>
<thead>
<tr>
<th>16.2%</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completed on time and budget, with all the promised functionality</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>52.7%</th>
<th>Partial Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over cost, over time, and/or lacking promised functionality</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31.1%</th>
<th>Complete Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project abandoned or cancelled</td>
<td></td>
</tr>
</tbody>
</table>

... And the Price of Software Failure Can be Immense

Credit: Randi Rost, Intel, The Standish Group report 83.9% of IT projects partially or completely fail, Feb. 20, 2019
Goals of MP:
Make Programmers More Productive and Democratize the creation of Software

How?

Automated Debugging and Development Tools, Improved Programmer Intention Languages

1. **AutoPerf** – state-of-the-art MP parallel performance regression testing system for enhanced software debugging
2. **ControlFlag** - self-supervised idiosyncratic pattern detection system for software control structures for enhanced error detection and correction
3. **Machine Inferred Code Similarity (MISIM)** – to automate code development
Let’s look at two fundamental PROBLEMS

And Let’s work our way backwards
Problem #2

Software Debugging

Ensuring software has stable quality as things change

• Quality = correctness, performance, and security

Changes in SW/HW tend to reduce software quality

• These reductions in quality are known as bugs

Debugging

• The process of observing, root causing, and eliminating bugs

But how expensive is debugging, really?
Productivity – Debugging

RESULTS: The global cost of software development is US$1.25 trillion

Productivity – Debugging

50% of the cost of software development is debugging

University of Cambridge (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.370.9611&rep=rep1&type=pdf)
Debugging: Performance Regression Testing

Test for detecting **performance anomaly** introduced by a change in software

MP = Machine Programming, Ninja = an expert in SW development generally requiring a deep understanding of HW
Debugging: Performance Regression Testing

AutoPerf – state-of-the-art MP parallel performance regression testing system (NeurIPS ‘19)

“A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions” (Alam et al., NeurIPS ’19)
ControlFlag: Self-supervised Idiosyncratic Pattern Detection System for Software Control Structures

Machine Programming Research
Introduction

• Detects possible errors in control structures (if, while, etc.) in code

• Possible error in \( x = 7 \)
  • Should it be: \( x == 7 \)?
    • A: Most likely
    • Because “==” is more common in if than “=“.

• We don’t know programmer’s true intention
  • Lack of program specification or test cases
Wait! Can’t compilers catch this case?

• Yes!
  • No compiler error – valid syntax
  • Compiler can only warn - gcc –Wall

Rules-based Approach
	est.cpp:3:9: warning: suggest parentheses around assignment used as truth value [-Wparentheses]

if (x = 7) y = x;

• Rules-based approach: labor-intensive and require compilable program
How does ControlFlag find errors?

1. **Hypothesis**: *certain* patterns are *uncommon* in control structures
2. **Mine** idiosyncratic patterns in open-source projects on github
3. **Check** user’s pattern against mined patterns
4. Flag if anomalous

Data!

Formulate the problem as an anomaly detection problem
ControlFlag’s Design

Step 1: Pattern mining

- Semi-trust (humans must decide this)
- Learn idiosyncratic patterns in code
- Self-supervision; no labels

Step 1-0: Source Code Repository Selection
- Codebase

Step 1-1: Mine patterns in control structures
- Source code parser

Step 1-2: Build representation for patterns

Step 1-3: Self-supervised clustering using decision tree

Step 2: Scanning for erroneous patterns

- Step 2-0: Target Code Repository
- Codebase

- Step 2-1: Mine patterns in control structures
- Source code parser

- Step 2-2: Build representation for patterns

- Step 2-2: Check pattern in the decision tree

- Pattern in training data?

- Step 2-3: Suggest possible corrections
ControlFlag Utilizes

- Self-supervised erroneous pattern detection system
- Semi-trusted training data
Evaluation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Package</th>
<th>Number of programs</th>
<th>Lines of Code</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>6000 packages</td>
<td>2.57 Million</td>
<td>1.1 Billion</td>
<td>~38 Million</td>
</tr>
<tr>
<td></td>
<td>OpenSSL</td>
<td>1212</td>
<td>623,000</td>
<td>30,000</td>
</tr>
<tr>
<td>Evaluation</td>
<td>CURL</td>
<td>736</td>
<td>225,000</td>
<td>13,000</td>
</tr>
</tbody>
</table>

- Anomaly threshold – compare frequency of a pattern to the frequencies of its possible corrections (e.g., teh -> the)
- If corrections appear more frequently, then most likely the pattern is an anomaly
Anomaly flagged in CURL-7.73 (latest release)

Potential anomaly: (s->keepon > TRUE)
Location: curl/lib/http_proxy.c:359
Possible corrections:
(s->keepon > TRUE), edit distance 0, occurrences 4
(s->keepon > number), edit distance 2, occurrences 127540
(s->keepon > variable), edit distance 2, occurrences 56475

```
if(s->keepon > TRUE) /* This means we are currently ignoring a response-body */
...
s->keepon = FALSE;
s->keepon = 2;
```

- **CURL is a popular package**, first released in 1997 (Wikipedia), has **18,300 stars on GitHub**
- **TRUE is a Boolean type** compared with `keepon`, which is integer. Greater than Boolean has no intrinsic meaning
- **GCC-10 (latest!)** doesn’t warn in this case
Problem #1

Software Development

- Aim to build new software correctly so problem #2 vanishes
- Programming is difficult
  - How can we help programmers write code (or think) like experts?
- Hardware and software heterogeneity complicates problem
  - Multiplicative complexity

SW = software, HW = hardware
Ninja = an expert in SW development generally requiring a deep understanding of HW
Poor Software Development is Expensive

$2.84T

Credit: Randi Rost, Intel
Poor Software Development is Expensive

$2.84T

Credit: Randi Rost, Intel

FIGURE 3: THE ICEBERG MODEL

- Customer problem reports
- Customer service calls
- Lawsuits/warranty claims
- QA & test department costs
- Service outages

- Finding & fixing internal problems/defects
- Cancelled and delayed projects
- Unaccounted overtime (crisis mode)
- Waste and rework
- Successful cyber attacks
- Staffing problems (e.g., turnover)
- Poor teamwork
- Lack of good planning
- Dubious project value/ROI
- Excessive systems costs
- Lost market opportunities

- Lack of good practices & quality standards
- Understanding complex code
- Technical debt
- Poor quality data
Let’s Help Programmers Develop Software

- Build a code recommendation system (i.e., expert virtual assistant)
- Auto-complete line of code
- Auto-complete algorithm (or select replacement one API call)
- Recommend code changes to improve quality
- Automate bug fixing
- Etc.
- Code recommendation relies on code similarity
- Code similarity is an open research problem

SW = software, HW = hardware
Ninja = an expert in SW development generally requiring a deep understanding of HW
What is Code Similarity?

• **Code similarity** aims to determine if two code snippets are:
  • Semantically similar
  • Irrespective of syntactical differences

• Code similarity helps us to understand:
  • What code is doing
  • Not how it is being done

• This is a non-trivial problem – so we built MISIM
  • Machine Inferred Code Similarity (MISIM)
Machine Inferred Code Similarity (MISIM)

MISIM System Design

MISIM – Machine Inferred Code Similarity – Intel, Georgia Tech, MIT
See cited source for workloads and configurations. Results may vary.
Machine Inferred Code Similarity (MISIM)

Takeaway

▪ Good Early Progress

▪ More accurate than all prior state-of-the-art systems we compared it to (1.5x – 43.4x)

▪ Big Step in Automating many aspects of Machine Programming
MISIM: In the News

Intel, MIT and Georgia Tech Deliver Improved Machine-Programming Code Similarity System

Week 1 Press Coverage Summary
August 5, 2020
Legal Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Results have been estimated or simulated.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.