intel

Machine Programming & Data-Driven Dependable and Secure Software Systems

Justin Gottschlich

Principal AI Scientist & Director/Founder of Machine Programming Research (Intel Labs)

Adjunct Assistant Professor (University of Pennsylvania)

Steering Committee Chair, ACM Machine Programming Symposium (MAPS)

Contributors:
Todd Anderson, Saman Amarasinghe, Regina Barzilay, Michael Carbin, Alvin Cheung, Pradeep Dubey, Henry Gabb, Niranjan Hasabnis, Adam Herr, Jim Held, Tim Kraska, Insup Lee, Geoff Lowney, Shanto Mandal, Tim Mattson, Pranav Mehta, Abdullah Muzahid, Paul Petersen, Alex Ratner, Martin Rinard, Vivek Sarkar, Koushik Sen, Armando Solar-Lezama, Joe Tarango, Nesime Tatbul, Josh B. Tenenbaum, Jesmin Tithi, Javier Turek, Abdul Wasay, Rich Uhlig, Anand Venkat, Fangke Ye, Xin Zhang, Shengtian Zhou … and many others.

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 noninfringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Overview

- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
  - Separation of Intention is Critical
- The Bifurcated Space of MP
  - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
  - ControlFlag: a Self-Supervised Systems for MP
  - MISIM: a Code Semantics Similarity System

Overview

- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
    - Separation of Intention is Critical
- The Bifurcated Space of MP
    - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
    - ControlFlag: a Self-Supervised Systems for MP
    - MISIM: a Code Semantics Similarity System

Definition: Machine Programming (MP) *is the automation of software and hardware development*

Machine Programming Research (MPR)

A New Pioneering Research Initiative at

intel labs

Intel Labs' MPR Goals

Machine Programming (MP) is the automation of software and hardware development

Time:

Reduce development time of all aspects of software development

*Measured as 1000x+ improvement over human work performed today

Quality:

Better software than the best human programmers*

*Measured as superhuman correctness, performance, security, etc.

Intel Labs' MPR Goals

Machine Programm … *d hardware development*

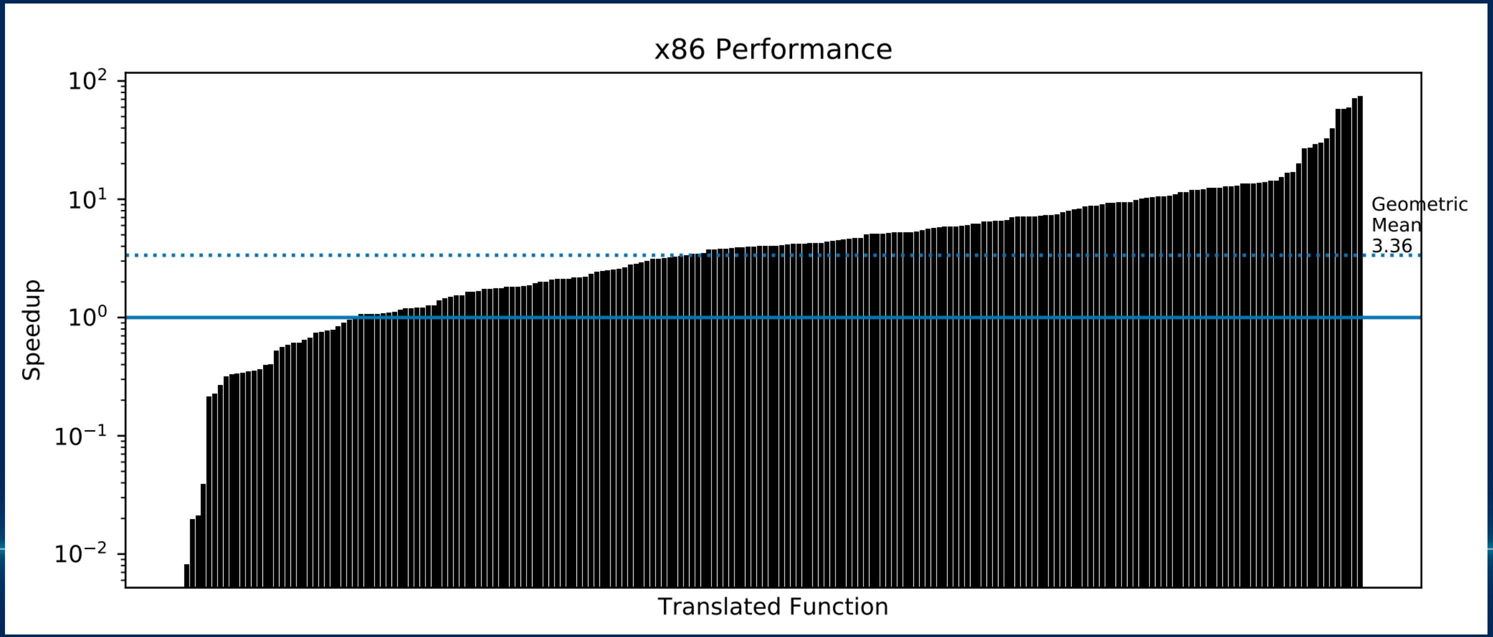
Concrete Data Point:

"Automatically Translating Image Processing Libraries to Halide" **(Ahmad et al., 2019)***

***Funded by Intel's CAPA Research Center**

Time:

Quality:



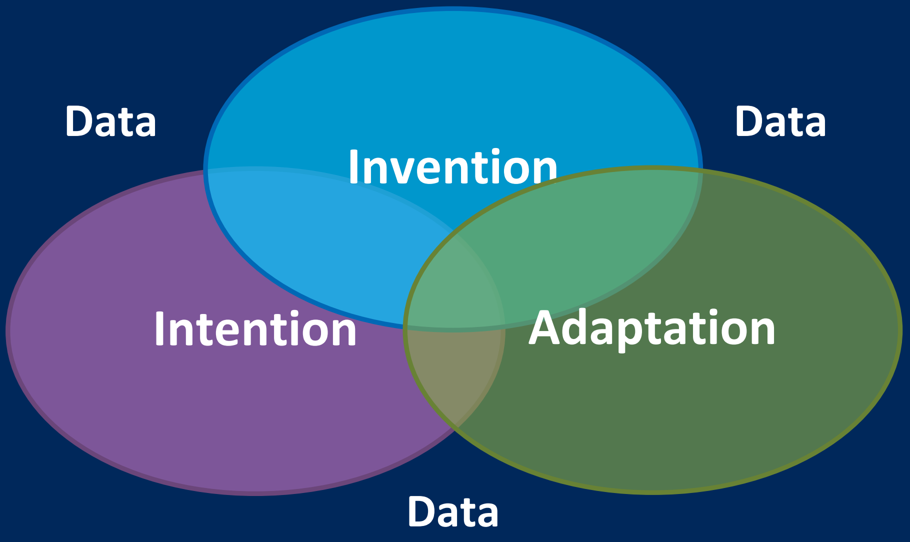
Overview

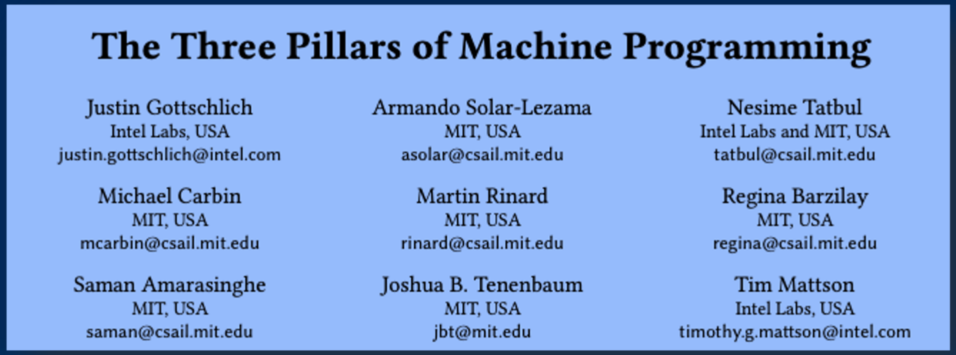
- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
  - Separation of Intention is Critical
- The Bifurcated Space of MP
  - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
  - ControlFlag: a Self-Supervised Systems for MP
  - MISIM: a Code Semantics Similarity System

The Three Pillars of Machine Programming

Machine Programming (MP) is the automation of software and hardware development

- **Intention:** Discover the intent of a programmer; lift meaning from software
- **Invention:** Create new algorithms and data structures; compositional novelty
- **Adaptation:** Evolve in a changing hardware/software world




The Three Pillars of Machine Programming

Justin Gottschlich
Intel Labs, USA
justin.gottschlich@intel.com

Armando Solar-Lezama
MIT, USA
asolar@csail.mit.edu

Nesime Tatbul
Intel Labs and MIT, USA
tatbul@csail.mit.edu

Michael Carbin
MIT, USA
mcarbin@csail.mit.edu

Martin Rinard
MIT, USA
rinard@csail.mit.edu

Regina Barzilay
MIT, USA
regina@csail.mit.edu

Saman Amarasinghe
MIT, USA
saman@csail.mit.edu

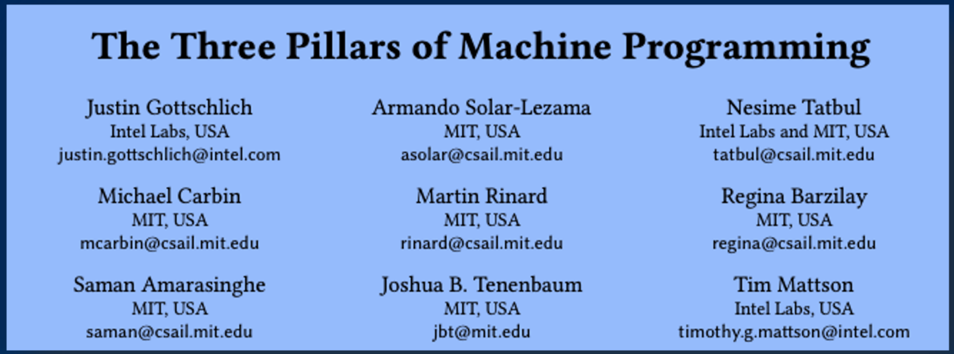
Joshua B. Tenenbaum
MIT, USA
jbt@mit.edu

Tim Mattson
Intel Labs, USA
timothy.g.mattson@intel.com

The Three Pillars of Machine Programming

Machine Programming (MP) is the automation of software and hardware development

- **Intention:** Discover the intent of a programmer; lift meaning from software
- **Invention:** Create new algorithms and data structures; compositional novelty
- **Adaptation:** Evolve in a changing hardware/software world


The Three Pillars of Machine Programming

Justin Gottschlich
Intel Labs, USA
justin.gottschlich@intel.com

Armando Solar-Lezama
MIT, USA
asolar@csail.mit.edu

Nesime Tatbul
Intel Labs and MIT, USA
tatbul@csail.mit.edu

Michael Carbin
MIT, USA
mcarbin@csail.mit.edu

Martin Rinard
MIT, USA
rinard@csail.mit.edu

Regina Barzilay
MIT, USA
regina@csail.mit.edu

Saman Amarasinghe
MIT, USA
saman@csail.mit.edu

Joshua B. Tenenbaum
MIT, USA
jbt@mit.edu

Tim Mattson
Intel Labs, USA
timothy.g.mattson@intel.com

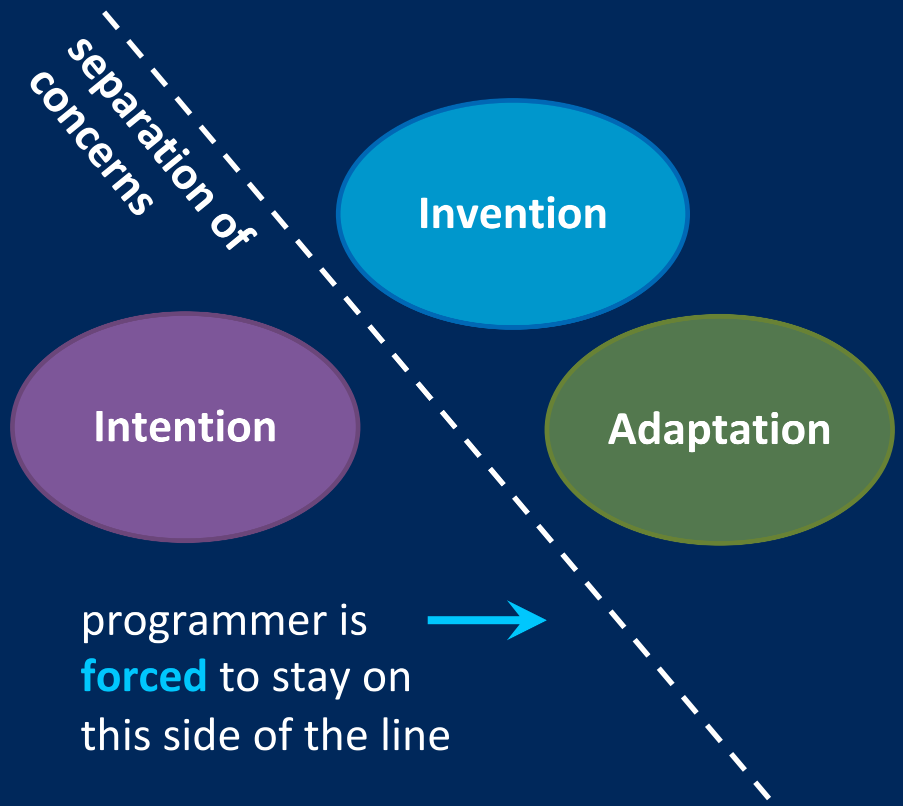
Separation of Intention is Critical

- Requires user only supply **core idea** (improving productivity)
- Enables machine to explore a **wider range** of possible solutions (improving MP-generated solutions)
- Enables automatic SW **adaptation & evolution**

We anticipate this separation will give rise to:

- **Intentional** Programming Languages

  Example: Halide/Verified Lifting (Adobe Photoshop)

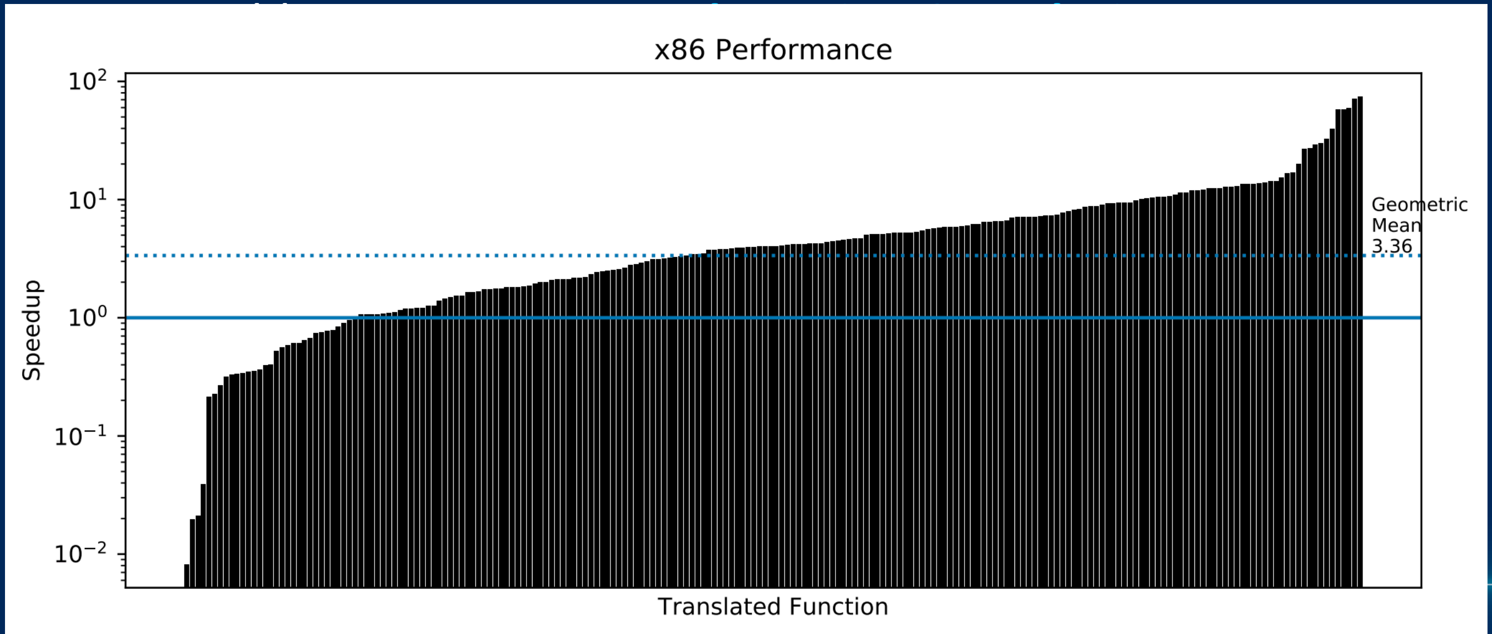


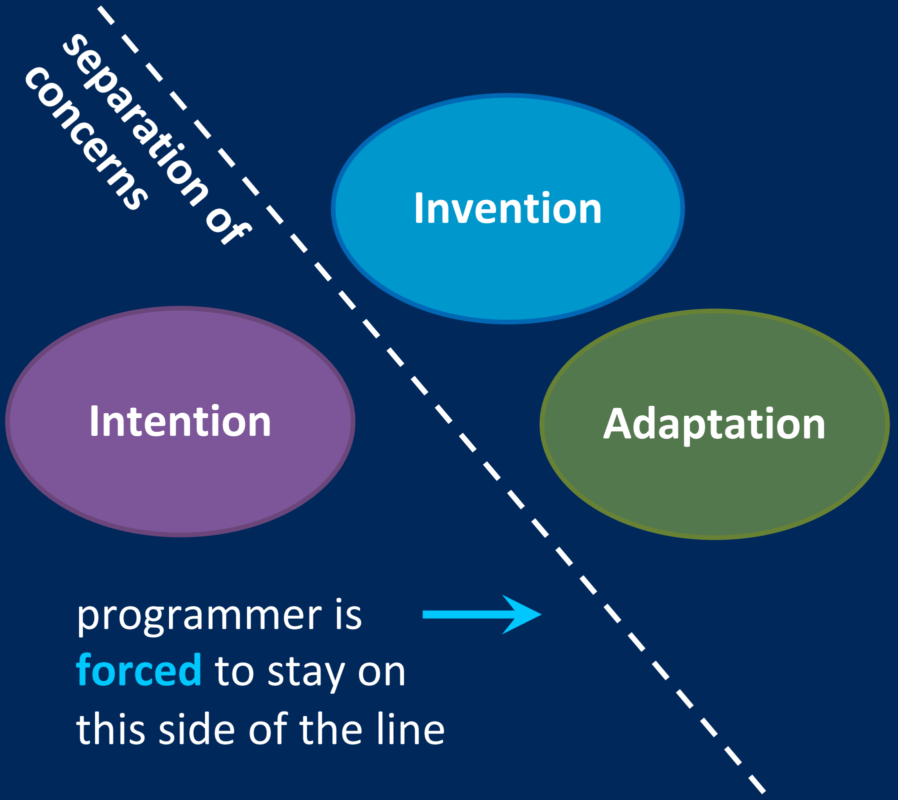
Separation of Intention is Critical

Leverages Separation of Intention from Invention & Adaptation

"Automatically Translating Image Processing Libraries to Halide" **(Ahmad et al., 2019)**

-
- sible





Overview

- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
  - Separation of Intention is Critical
- The Bifurcated Space of MP
  - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
  - ControlFlag: a Self-Supervised Systems for MP
  - MISIM: a Code Semantics Similarity System

The Bifurcated Space of Machine Programming

	Stochastic	Deterministic
Techniques used in MP systems	**Machine Learning** **(Neural networks, reinforcement learning, genetic algorithms, Bayesian networks, etc.)**	**Formal Methods** **(Formal verifiers, spatial and temporal logics, formal program synthesizers, etc.)**

Progressively more approximate ← → **Progressively more precise**

Components used in MP systems

Software

Programming languages, algorithms, data structures, etc.

Hardware

Compute, communications, and memory architectures, etc.

The Bifurcated Space of Machine Programming

Techniques used in MP systems

Stochastic

Machine Learning

(Neural networks, reinforcement learning, genetic algorithms, Bayesian networks, etc.)

Deterministic

Formal Methods

(Formal verifiers, spatial and temporal logics, formal program synthesizers, etc.)

proximate - - - - - - - - - - → **Progressively more precise**

Stochastic MP systems tend to improve w/ *more* iid data

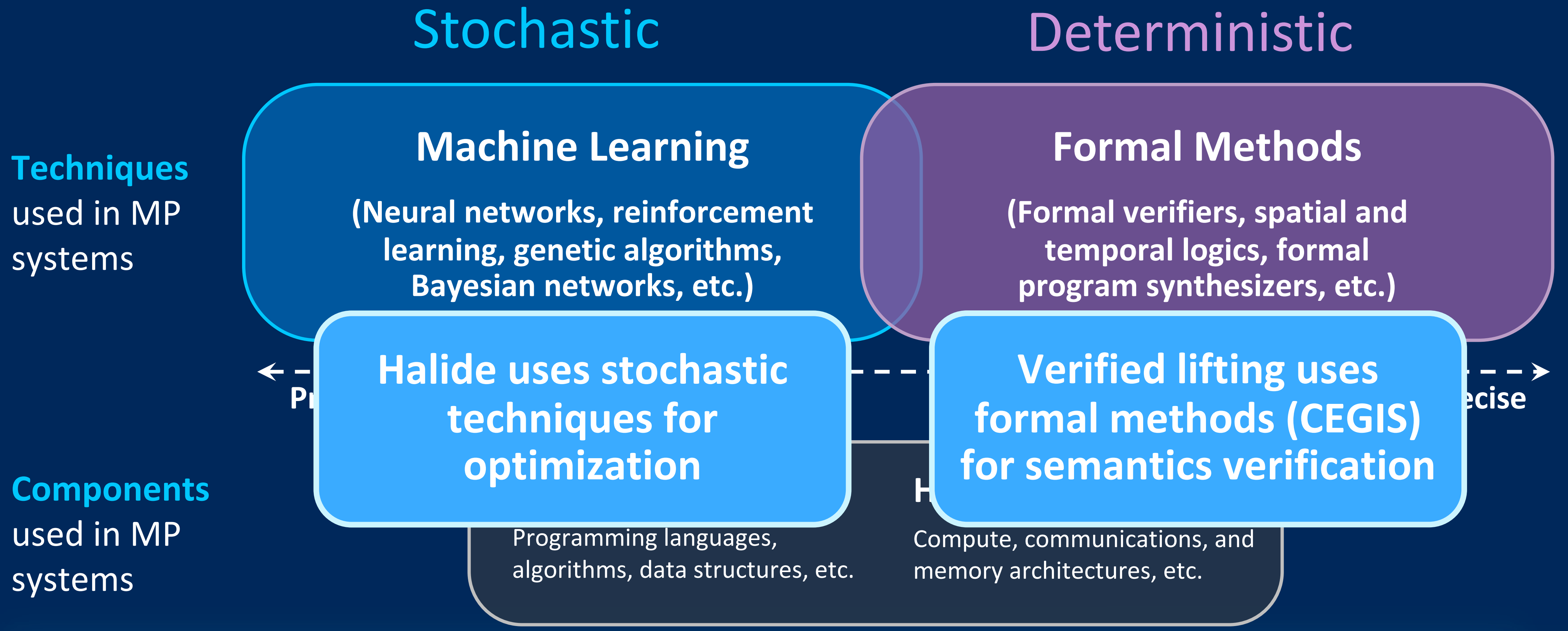
are

ming languages,
ms, data structures, etc.

Hardware

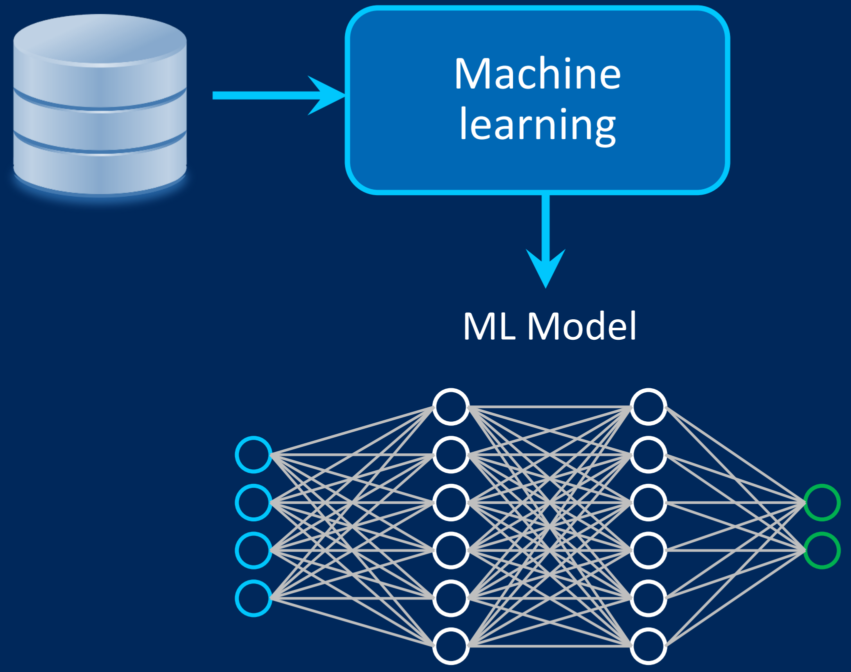
Compute, communications, and memory architectures, etc.

The Bifurcated Space of Machine Programming

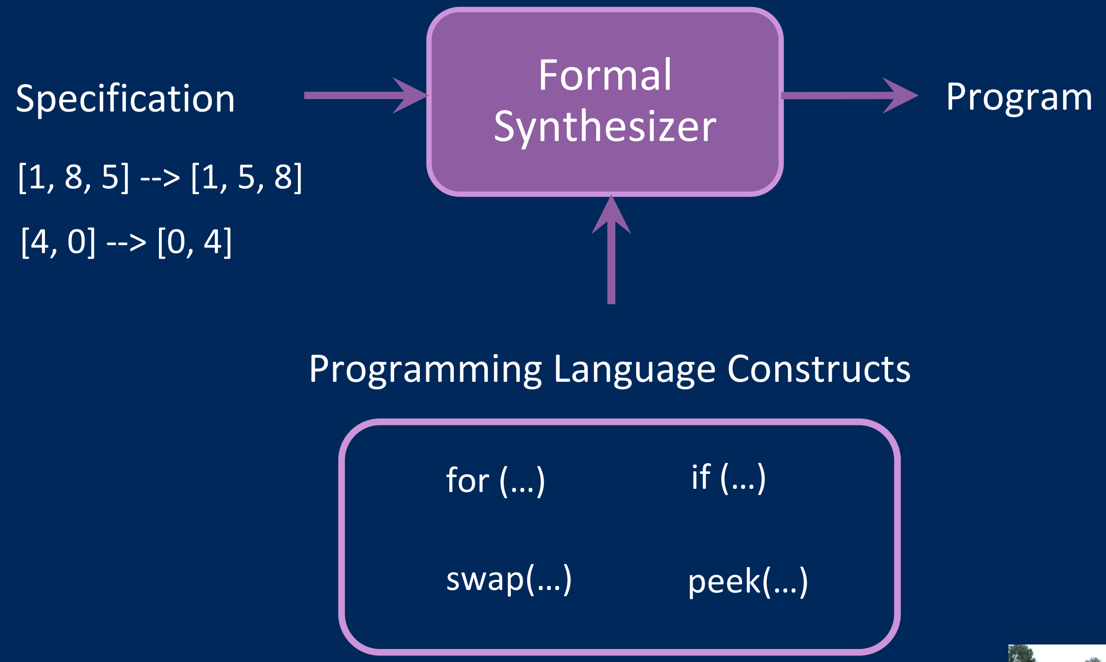


Concretizing The Two Sides of MP with Neuro-Symbolism

Stochastic (Neuro)



Deterministic (Symbolic)



Credit: Jeevana Inala & Armando Solar-Lezama

Overview

- Machine Programming Research @ Intel
- Discussion of The Three Pillars of MP
  - Separation of Intention is Critical
- The Bifurcated Space of MP
  - Stochastic and Deterministic
- Machine Programming Emphasis @ Intel
  - ControlFlag: a Self-Supervised Systems for MP
  - MISIM: a Code Semantics Similarity System

Numerous MP Efforts @ Intel

Debugging / Profiling / Productivity

- ControlFlag, MISIM, & AutoPerf

Automated Performance Extraction

- Inteon's Tiger Shark (Intel venture)
- MP-based general-purpose compiler (e.g., ML-learned code optimizations)

And Many More …

Numerous MP Efforts @ Intel

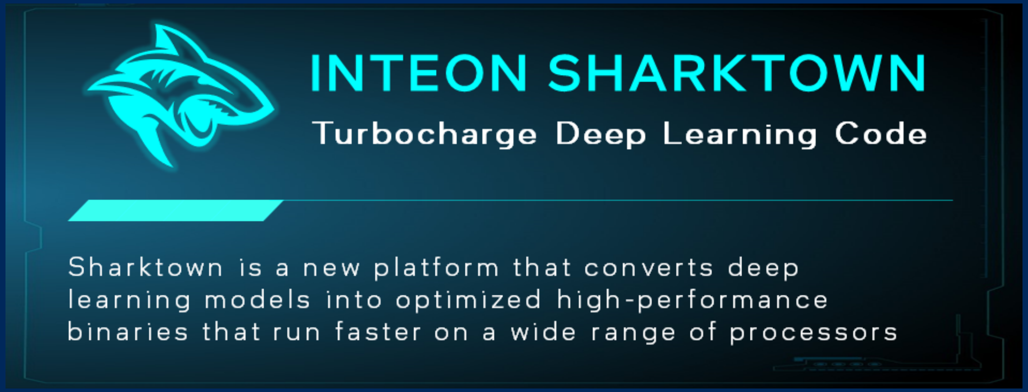
Debugging / Profiling / Productivity

- ControlFlag, MISIM, & AutoPerf

*Beats SOTA by ~2x with 400k labeled data samples
**Beats SOTA by ~5x with 1M labeled data samples
(independently confirmed by IBM/MIT)

Automated Performance Extraction

- Inteon's Tiger Shark (Intel venture)
- MP-based general-purpose compiler (e.g., ML-learned code optimizations)



And Many More …

*"MISIM: A Neural Code Semantics Similarity System Using the Context-Aware Semantics Structure" by Ye et al. (https://arxiv.org/abs/2006.05265)

**"CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks" by Puri et al. (https://arxiv.org/abs/2105.12655)

Numerous MP Efforts @ Intel

Debugging / Profiling / Productivity

- ControlFlag, MISIM, & AutoPerf

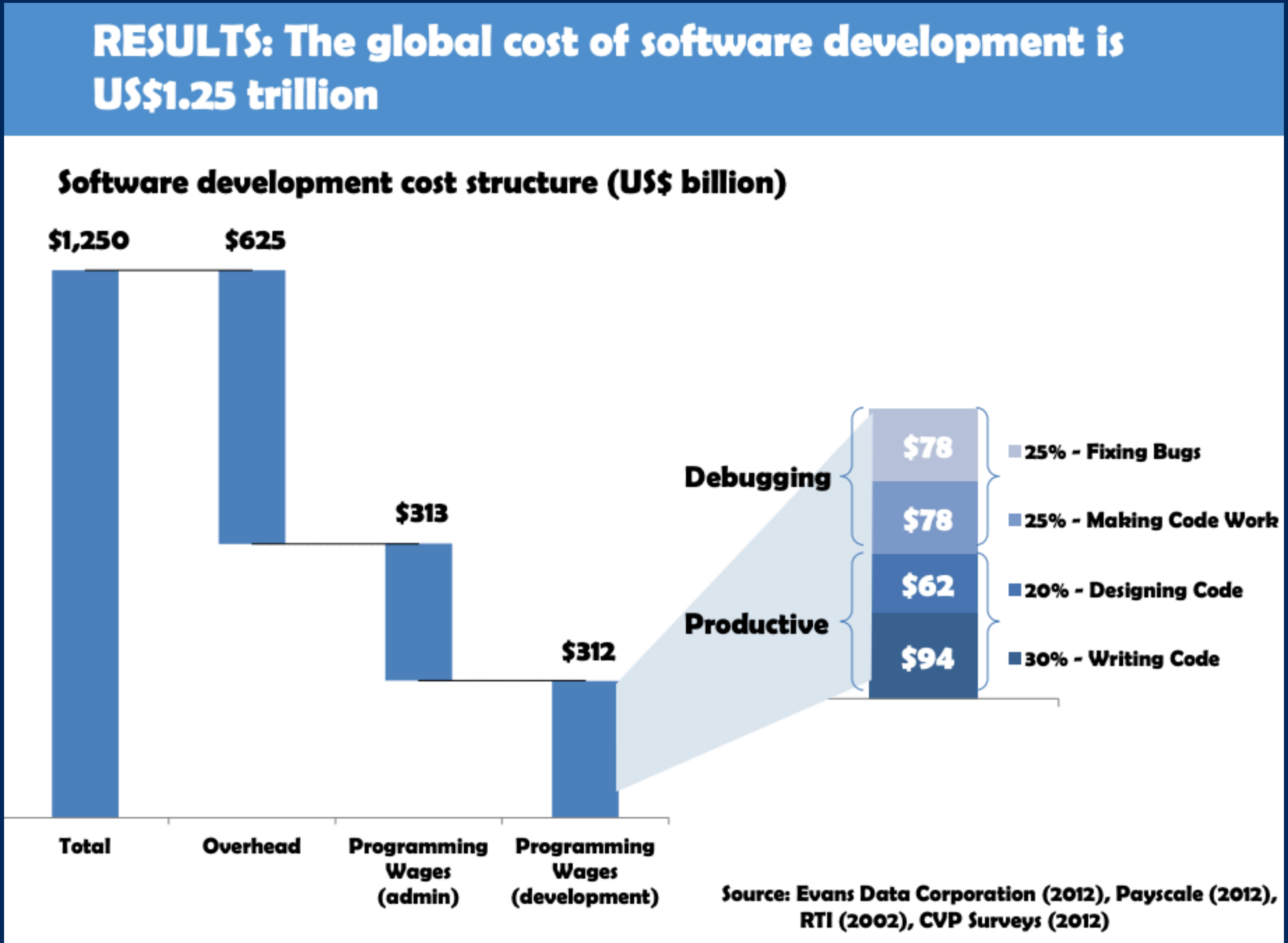
What can we build without labeled data?

Automated Performance Extraction

- Inteon's Tiger Shark (Intel venture)
- MP-based general-purpose compiler (e.g., ML-learned code optimizations)

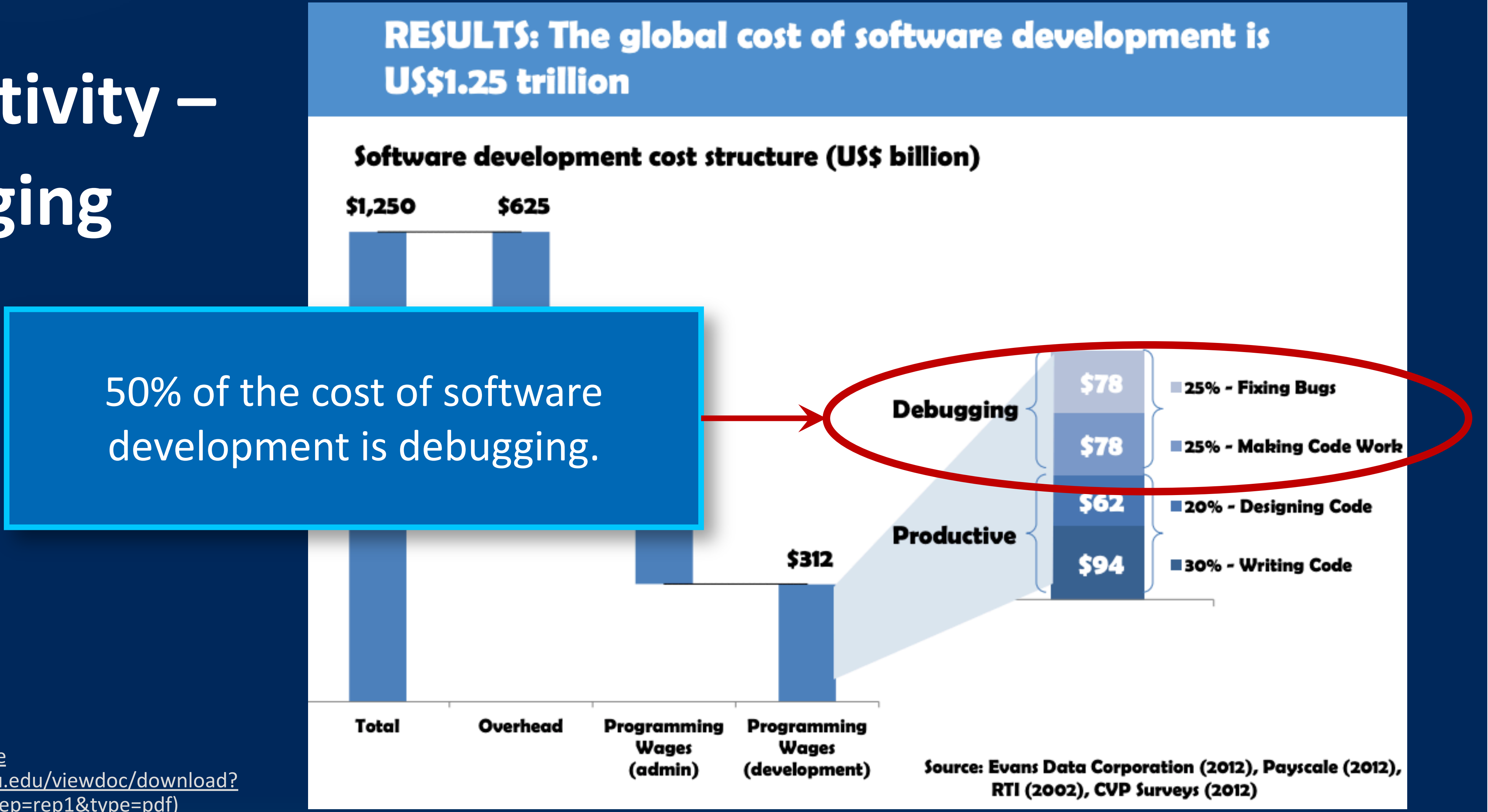
And Many More …

Productivity – Debugging



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

Productivity – Debugging



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

Debugging: Finding Code Anomalies

What is a code anomaly?

- A piece of code that is irregular

Why care about code anomalies?

- Anomalous code can lead to defects, technical debt, delayed software development (hard to understand code), loss of customer trust

Anomaly found in CURL (~30-year-old software)

CURL developers rewrite flagged piece of code found with ControlFlag

Re: Potential confusion in http_proxy.c and a recommendation

•*Contemporary messages sorted*: [by date] [by thread] [by subject] [by author] [by messages with attachments]
From: Daniel Stenberg via curl-library <curl-library_at_cool.haxx.se>
Date: Mon, 9 Nov 2020 23:51:20 +0100 (CET)

On Mon, 9 Nov 2020, Hasabnis, Niranjan via curl-library wrote:

> We believe that using "if (s->keepon > 1)" would eliminate this confusion
> and capture the intended semantics precisely.

I think you've pointed out code that could be written clearer, yes. But I think an even better improvement to this logic would be to use an enum or defined values that include all three used values as state names.

What do you think about my proposal over at:
https://github.com/curl/curl/pull/6193

https://curl.se/mail/lib-2020-11/0028.html

```
        }
-         s->keepon = FALSE;
+         s->keepon = KEEPON_DONE;
          break;
        }

-       if(s->keepon > TRUE) {
+       if(s->keepon == KEEPON_IGNORE) {
          /* This means we are currently ignoring a response-body */

6 lib/urldata.h

@@ -802,7 +802,11 @@ struct proxy_info {
  /* struct for HTTP CONNECT state data */
  struct http_connect_state {
    struct dynbuf rcvbuf;
-   int keepon;
+   enum keeponval {
+     KEEPON_DONE,
+     KEEPON_CONNECT,
+     KEEPON_IGNORE
+   } keepon;
    curl_off_t cl; /* size of content to read and ignore */
    enum {
      TUNNEL_INIT,    /* init/default/no tunnel state */
```

Anomaly found in CURL (~30-year-old software)

CURL developers rewrite flagged piece of code found with ControlFlag

Re: Potential confusion in http_proxy.c and a recommendation

•*Contemporary messages sorted*: [by date] [by thread] [by subject] [by author] [by messages with attachments]
From: Daniel Stenberg via curl-library <curl-library_at_cool.haxx.se>
Date: Mon, 9 Nov 2020 23:51:20 +0100 (CET)

On Mon, 9 Nov 2020, Hasabnis, Niranjan via curl-library wrote:

> *We believe that using "if (s->keepon > 1)" would eliminate this confusion*
> *and capture the intended semantics precisely.*

I think you've pointed out code that could be written clearer, yes. But I think an even better improvement to this logic would be to use an enum or defined values that include all three used values as state names.

What do you think about my proposal over at:
https://github.com/curl/curl/pull/6193

https://curl.se/mail/lib-2020-11/0028.html

```
355 355         }
356       -       s->keepon = FALSE;
    356   +       s->keepon = KEEPON_DONE;
357 357           break;
358 358         }
359 359
360       -     if(s->keepon > TRUE) {
    360   +     if(s->keepon == KEEPON_IGNORE) {
361 361           /* This means we are currently ignoring a response-body */
```

```
6 lib/urldata.h

@@ -802,7 +802,11 @@ struct proxy_info {
802 802    /* struct for HTTP CONNECT state data */
803 803    struct http_connect_state {
804 804      struct dynbuf rcvbuf;
805      -   int keepon;
    805  +   enum keeponval {
    806  +     KEEPON_DONE,
    807  +     KEEPON_CONNECT,
    808  +     KEEPON_IGNORE
    809  +   } keepon;
806 810      curl_off_t cl; /* size of content to read and ignore */
807 811      enum {
808 812        TUNNEL_INIT,    /* init/default/no tunnel state */
```

Limitations in Existing Code Anomaly Detectors

Tools & techniques to identify software defects

- Testing (unit tests, QA, etc.)
- Static analysis
  - Compilers, linters

Limitations

- Continuous manual effort to maintain and update (i.e., adding new rules as things evolve)
- Manual efforts can be error-prone

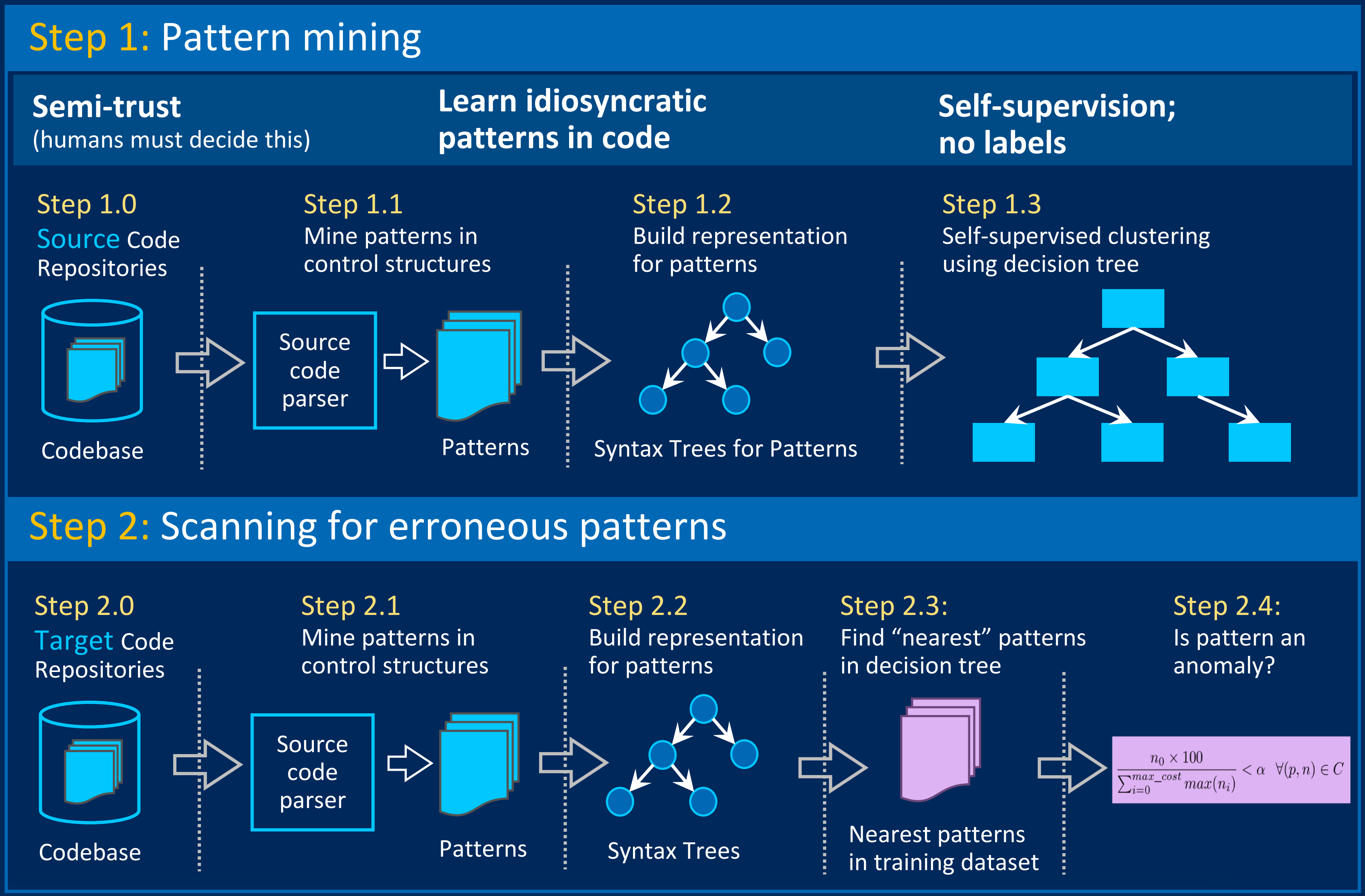
ControlFlag

A Self-Supervised Anomalous Code Detection System

Technical Lead:

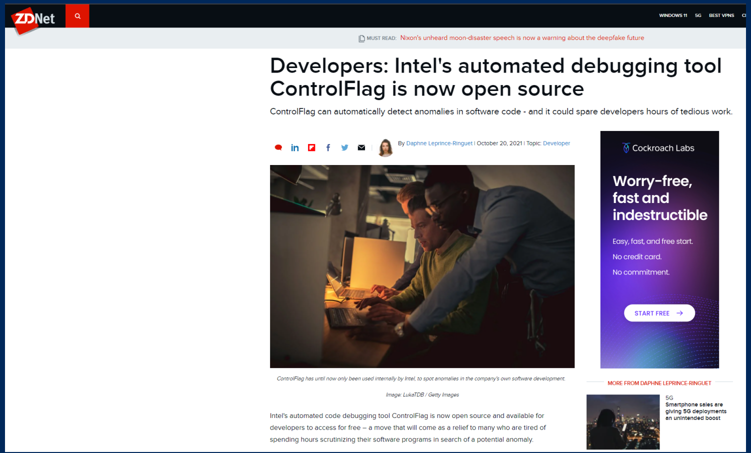
Dr. Niranjan Hasabnis

Intel Labs



"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

ControlFlag In The News

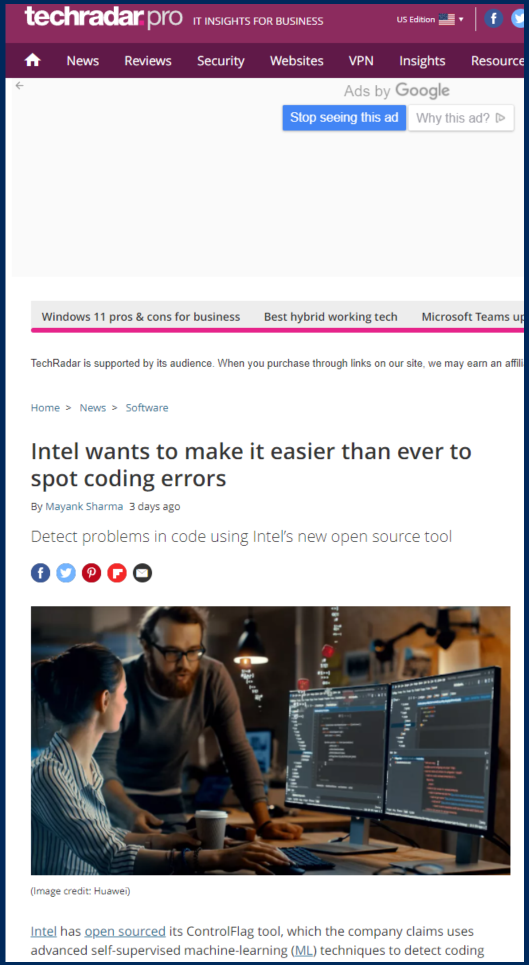


Developers: Intel's automated debugging tool ControlFlag is now open source

ControlFlag can automatically detect anomalies in software code - and it could spare developers hours of tedious work.

By Daphne Leprince-Ringuet | October 20, 2021 | Topic: Developer

Intel's automated code debugging tool ControlFlag is now open source and available for developers to access for free – a move that will come as a relief to many who are tired of spending hours scrutinizing their software programs in search of a potential anomaly.



techradar pro IT INSIGHTS FOR BUSINESS

News Reviews Security Websites VPN Insights Resource

Home > News > Software

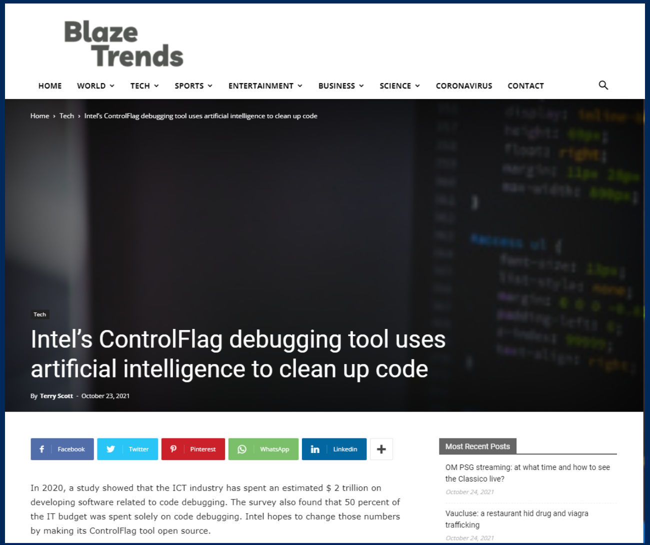
Intel wants to make it easier than ever to spot coding errors

By Mayank Sharma 3 days ago

Detect problems in code using Intel's new open source tool

(Image credit: Huawei)

Intel has open sourced its ControlFlag tool, which the company claims uses advanced self-supervised machine-learning (ML) techniques to detect coding

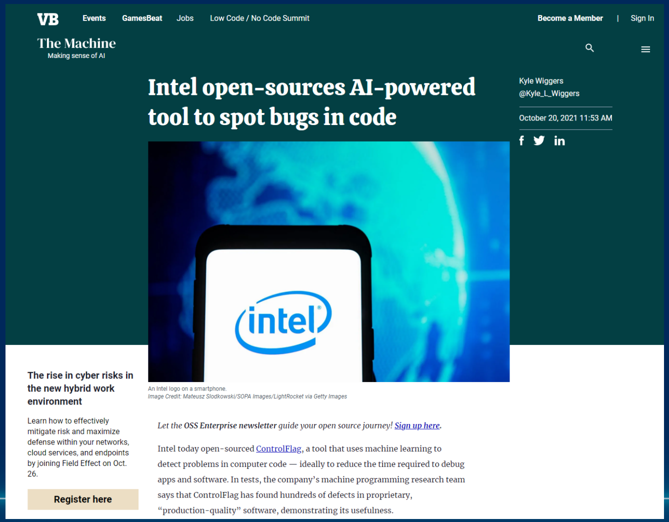


Blaze Trends

Intel's ControlFlag debugging tool uses artificial intelligence to clean up code

By Terry Scott - October 23, 2021

In 2020, a study showed that the ICT industry has spent an estimated $ 2 trillion on developing software related to code debugging. The survey also found that 50 percent of the IT budget was spent solely on code debugging. Intel hopes to change those numbers by making its ControlFlag tool open source.



VB The Machine Making sense of AI

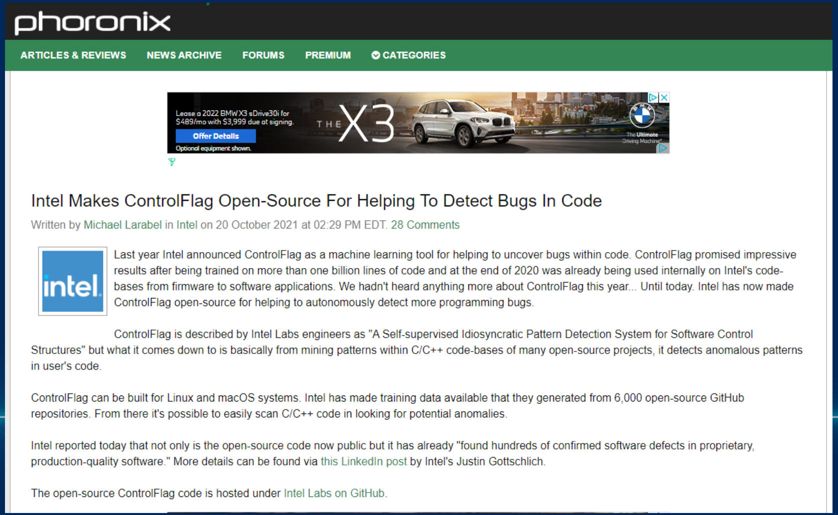
Intel open-sources AI-powered tool to spot bugs in code

Kyle Wiggers @Kyle_L_Wiggers

October 20, 2021 11:53 AM

Let the OSS Enterprise newsletter guide your open source journey! Sign up here.

Intel today open-sourced ControlFlag, a tool that uses machine learning to detect problems in computer code — ideally to reduce the time required to debug apps and software. In tests, the company's machine programming research team says that ControlFlag has found hundreds of defects in proprietary, "production-quality" software, demonstrating its usefulness.



phoronix

Intel Makes ControlFlag Open-Source For Helping To Detect Bugs In Code

Written by Michael Larabel in Intel on 20 October 2021 at 02:29 PM EDT. 28 Comments

Last year Intel announced ControlFlag as a machine learning tool for helping to uncover bugs within code. ControlFlag promised impressive results after being trained on more than one billion lines of code and at the end of 2020 was already being used internally on Intel's code-bases from firmware to software applications. We hadn't heard anything more about ControlFlag this year... Until today. Intel has now made ControlFlag open-source for helping to autonomously detect more programming bugs.

ControlFlag is described by Intel Labs engineers as "A Self-supervised Idiosyncratic Pattern Detection System for Software Control Structures" but what it comes down to is basically from mining patterns within C/C++ code-bases of many open-source projects, it detects anomalous patterns in user's code.

ControlFlag can be built for Linux and macOS systems. Intel has made training data available that they generated from 6,000 open-source GitHub repositories. From there it's possible to easily scan C/C++ code in looking for potential anomalies.

Intel reported today that not only is the open-source code now public but it has already "found hundreds of confirmed software defects in proprietary, production-quality software." More details can be found via this LinkedIn post by Intel's Justin Gottschlich.

The open-source ControlFlag code is hosted under Intel Labs on GitHub.

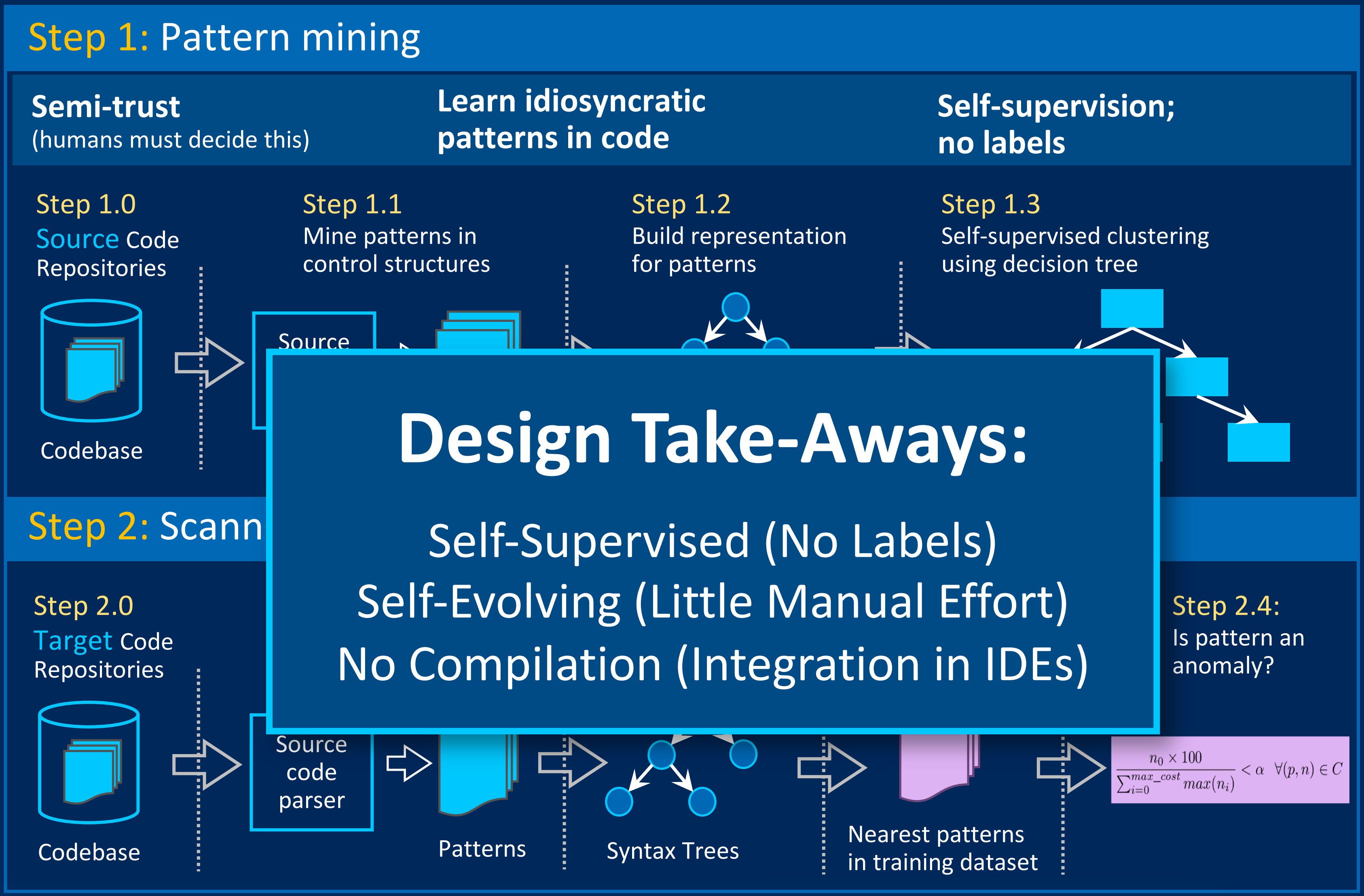
ControlFlag

A Self-Supervised Anomalous Code Detection System

Technical Lead:

Dr. Niranjan Hasabnis

Intel Labs



"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

Anomalies in Production-Quality, Open-Source Software

Evaluation: Setup

Training repository selection

- 6000 GitHub repos for C language having more than 100 stars
- 2.57M programs
- 1.1B Lines of code
- 38M patterns

Test repositories

- openssl, curl, ffmpeg
- git, vlc, lcx, lz4, reactos

Repo	GitHub stars	Found Anomalies	Scanned Expressions	Types of anomalies found
IoLanguage/io	2.3K	**5**	1635	Confusing expressions; missing parenthesis
Git/git	38.9K	**6**	6341	Confusing expression; character comparison using greater than or less than
Rubinius/rubinius	3K	**2**	10135	Character comparison using greater than or less than; missing parenthesis
FreeRADIUS/ freeradius-server	1.5K	**3**	20621	Character comparison using greater than or less than
Davidfstr/rdiscount	755	**4**	472	Character comparison using greater than or less than; missing parenthesis
Libharu/libharu	1.2K	**1**	2785	Character comparison using greater than or less than
Macournoyer/tinyrb	454	**3**	4369	Character comparison using greater than or less than
Rhomobile/rhodes	1K	**14**	76128	Confusing expressions; missing parenthesis; character comparison using greater or less than

"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

Anomaly Found in Proprietary & Deployed Software

Anomaly flagged by ControlFlag: in 12
if (address % 4)

```
1. void func() {
2.  uint32_t* p32;
3.  uint16_t* p16;
4.  uint32_t index = 0, other_index = <function_call>;

5.  for(j = 0; j < ..; j++) {
6.   if (array[j+1] == some_value) {
7.      index = j + 1;
8.      break;
9.   }
10. }

11. p16 = &array1[other_index].array2[index];

12. if ((uint32_t) &array1[other_index].array2[index] % 4) {
13.   p32 = (uint32_t*)(p16 - 1);
14.   *p32 = (*p32 & 0x0000FFFF) | (uin32_t) (mask);
15. } else {
16.   p32 = (uint32_t*) p16;
17.   *p32 = (*p32 & 0xFFFF0000) | some_other_mask;
18. }
19. }
```

An Example of ControlFlag's Finding

Three defects:

1. Duplicate expression in lines 11 and 12
2. Possible out-of-bounds memory access (memory error) in line 14
3. Information leak, security vulnerability in line 11

"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

RESULTS: Summary of 1st Proprietary Repo Analysis

Identified 104 potential defects

- 812 scanned files (.C and .H)
- 353K scanned lines of code
- 4600 scanned expressions

3 hours total analysis time (approx.)

- 56 Intel CPU cores

Description	Count	Comments
Anomalies that are critical bugs	**2**	Type error; memory error; security vulnerability
Anomalies that can lead to unwanted side-effects	**39**	Missing NULL check; possible divide by 0; missing return value check
Anomalies that point to confusing programming style	4	Double parenthesis around expressions, when not required
Anomalies that point to improvements in programming styles	59	Not using named constants; constant on right hand of equality;
Total unique anomalies reported	104	Not including false positives

"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

RESULTS: Summary of 2nd Proprietary Repo Analysis

Identified 191 potential defects

- 19K scanned files (.C and .H)
- 10.9M scanned lines of code
- 18.7K scanned expressions

8 hours total analysis time (approx.)

- 12 Intel CPU cores

Description	Count	Comments
Bugs found (confirmed by group)	5	Bitwise operation instead of Boolean logic operation
Confusing programming styles that could lead to bugs	22	Overly complex code E.g., ((xxxx[pstate].yyy & 0x1) >> 0)
Syntactic improvements to code according to standard style guides	164	Stylistic deviations from standards
Total unique anomalies reported	191	Not including false positives

"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

RESULTS: Summary of 2nd Proprietary Repo Analysis

Identified 191 potential defects

- 19K scanned f
- 10.9M scanned
- 18.7K scanned

8 hours total anal

- 12 Intel CPU core

Description	Count	Comments
Bugs found (confirmed by group)	5	Bitwise operation instead of Boolean logic operation
		plex code [pstate].yyy & 0x1) >> 0)
		viations from standards
		ng false positives

Working on a larger scan of ~65M lines of code, which identified 25,000 anomalies.

Number of files (.C and .H)	126,896
Number of expressions	1,374,028
Number of lines of code	64,690,054

Intel's partner is working to integrate ControlFlag as a permanent component of their continuous integration process.

"*ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures*" by Hasabnis & Gottschlich, MAPS '21

CODE SEMANTICS

CODE SEMANTICS

What are code semantics?

The <u>meaning</u> behind the syntax.

Why should we care?

Many reasons: code comprehension and reasoning (Microsoft/GitHub Co-Pilot), bug detection, etc.

CODE SEMANTICS

What are code semantics?

The <u>meaning</u> behind the syntax.

Why should we care?

Many reasons: code comprehension and reasoning (Microsoft/GitHub Co-Pilot), bug detection, etc.

Formally, at the highest level

For some set of inputs, *I*

And two programs P_i and P_j

If programs, P_i and P_j are executed using inputs *I* and produce an identical set of outputs *O*

We say they are ***semantically equivalent***

CODE SEMANTICS

```
Program A

  int a;
  // algorithm
  while (!cin.eof()) {
    while (!cin.eof() && !isdigit(cin.peek()))
      cin.get(); // ignore
    // print out result
    if (cin >> a)
        cout << a << endl;
  }
```

```
Program B

  char *p, *head, c;
  p = (char *) malloc(sizeof(char) * 30);
  head = p; scanf("%c", p);
  while (*p != '\n') { p++; *p = getchar();}
  *p = '\0'; p = head;
  for (; *p != '\0'; p++) {
    if(*p <= '9' && *p >= '0'){printf("%c",*p);}
    else if(*(p+1) < 58 && *(p+1) > 47){putchar('\n');}
  }
```

These code snippets are semantically equivalent (according to our prior definition)

CODE SEMANTICS

Program A

```
int a;
// algorithm
while (!cin.eof()) {
  while (!cin.eof() && !i
    cin.get(); // ignore
  // print out result
  if (cin >> a)
      cout << a << endl;
}
```

Program B

```
char) * 30);
p = getchar();}
){printf("%c",*p);}
p+1) > 47){putchar('\n');}
```

My Opinion:

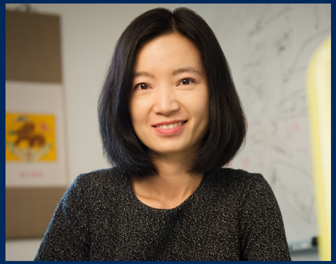
The ***Most*** Important Critical Open Problem for MP is Code Semantics Similarity

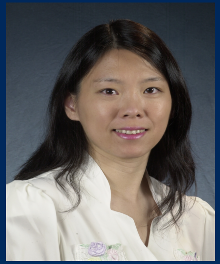
(this is a strong claim, I generally don't make such claims unless I feel strongly about something)

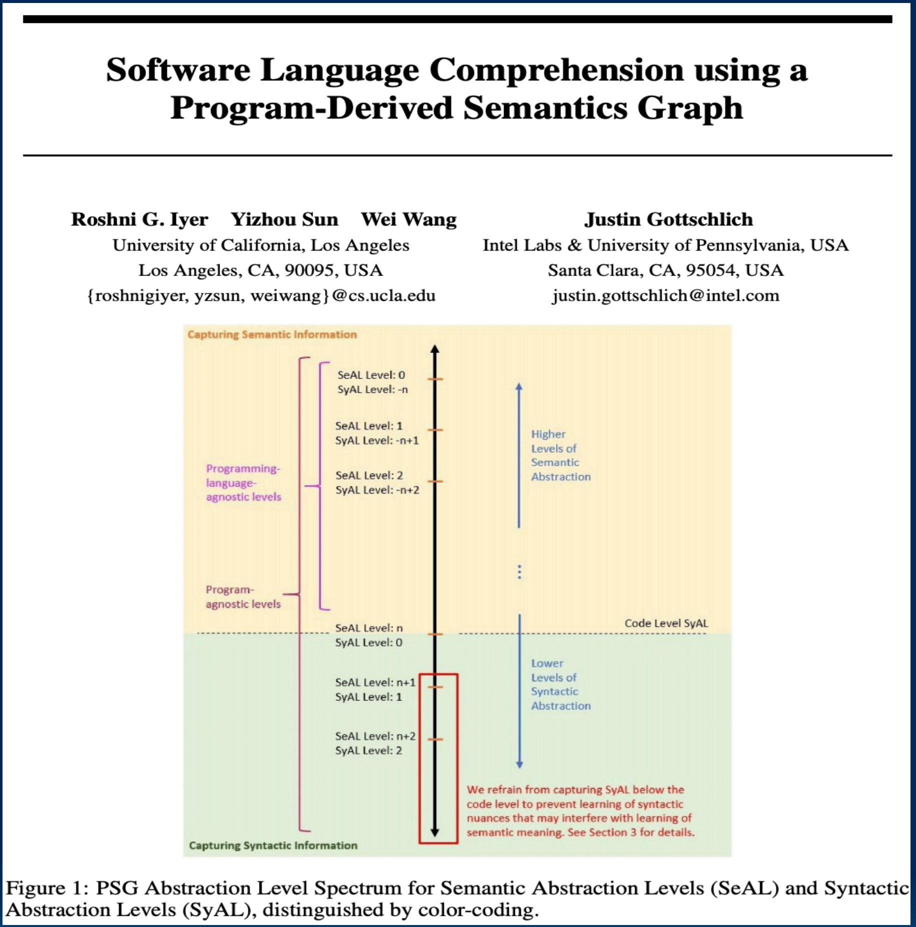
These code snippets are semantically equivalent (according to our prior definition)

CODE SEMANTICS: PROGRAM-DERIVED SEMANTICS GRAPH (PSG)

PSG is a graphical, hierarchical representation of code semantics






Software Language Comprehension using a Program-Derived Semantics Graph

Roshni G. Iyer Yizhou Sun Wei Wang
University of California, Los Angeles
Los Angeles, CA, 90095, USA
{roshnigiyer, yzsun, weiwang}@cs.ucla.edu

Justin Gottschlich
Intel Labs & University of Pennsylvania, USA
Santa Clara, CA, 95054, USA
justin.gottschlich@intel.com

Figure 1: PSG Abstraction Level Spectrum for Semantic Abstraction Levels (SeAL) and Syntactic Abstraction Levels (SyAL), distinguished by color-coding.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Computer-Assisted Programming Workshop, Vancouver, Canada. Copyright 2020 by the author(s).

PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY

Software Language Comprehension using a Program-Derived Semantic Graph Preprint, April, 2020

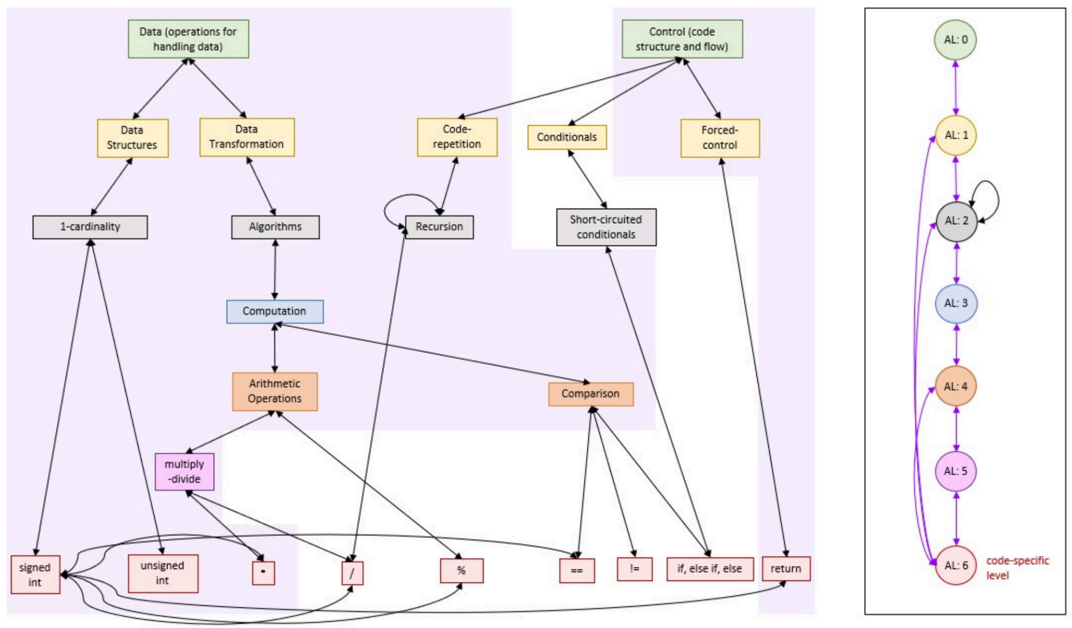


Figure 5: PSG of Recursive Power Function. The shaded region denotes overlap in the nodes of the PSG for the iterative power function shown in Figure 6. These total 17 of the 24 total nodes, a 70.83% overlap.

Implementation 1

```
signed int recursive_power(signed int x, unsigned int y)
{
   if (y == 0)
      return 1;
   else if (y % 2 == 0)
      return recursive_power(x, y / 2) *
         recursive_power(x, y / 2);
   else
      return x * recursive_power(x, y / 2) *
         recursive_power(x, y / 2);
}
```

Implementation 2

```
signed int iterative_power(signed int x, unsigned int y)
{



   }
   return val;
}
```

PSG = PROGRAM-DERIVED SEMANTICS GRAPH

PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY & ITERATIVELY

Software Language Comprehension using a Program-Derived Semantic Graph — Preprint, April, 2020

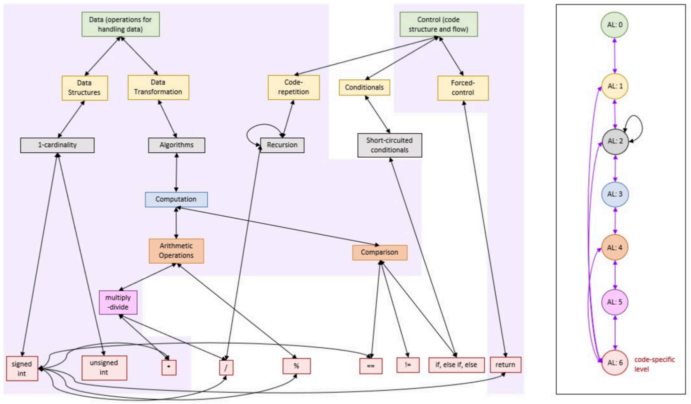


Figure 5: PSG of Recursive Power Function. The shaded region denotes overlap in the nodes of the PSG for the iterative power function shown in Figure 6. These total 17 of the 24 total nodes, a 70.83% overlap.

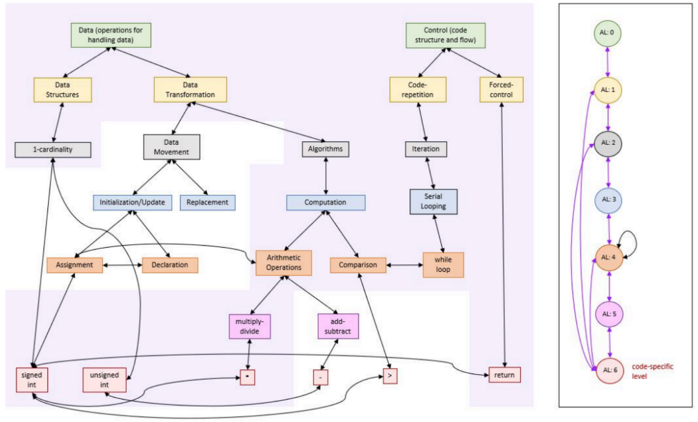


Figure 6: PSG of Iterative Power Function. The shaded region denotes overlap in the nodes of the PSG for the recursive power function shown in Figure 5. These total 19 of the 27 total nodes, a 70.37% overlap.

Implementation 1

```
signed int recursive_power(signed int x, unsigned int y)
{
    if (y == 0)
        return 1;
    else if (y % 2 == 0)
        return recursive_power(x, y / 2) *
            recursive_power(x, y / 2);
    else
        return x * recursive_power(x, y / 2) *
            recursive_power(x, y / 2);
}
```

Implementation 2

```
signed int iterative_power(signed int x, unsigned int y)
{
    signed int val = 1;
    while (y > 0) {
        val *= x;
        y -= 1;
    }
    return val;
}
```

PSG = PROGRAM-DERIVED SEMANTICS GRAPH

PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY & ITERATIVELY

Software Language Comprehension using a Program-Derived Semantic Graph

Preprint, April, 2020

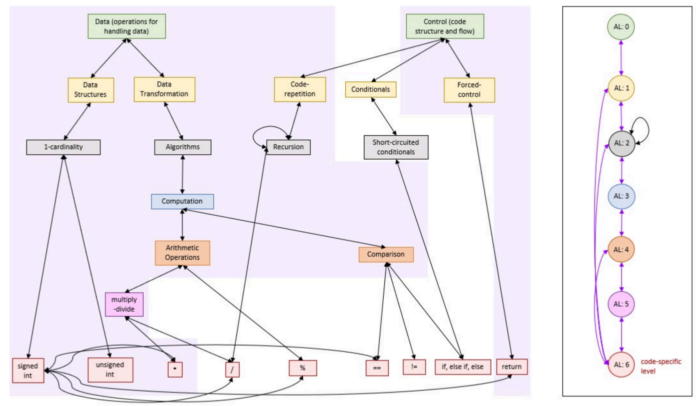


Figure 5: PSG of Recursive Power Function. The shaded region denotes overlap in the nodes of the PSG for the iterative power function shown in Figure 6. These total 17 of the 24 total nodes, a 70.83% overlap.

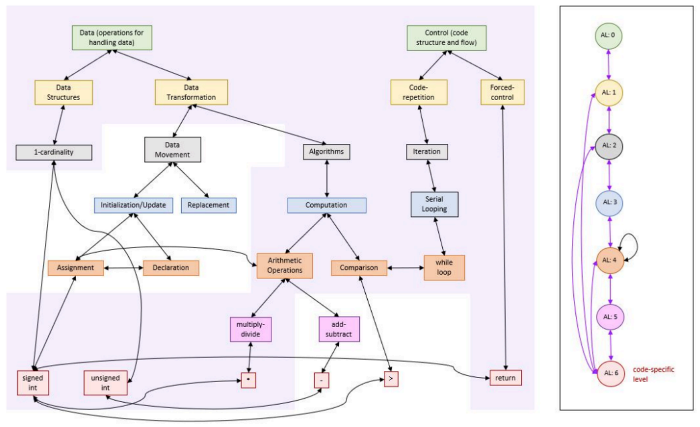


Figure 6: PSG of Iterative Power Function. The shaded region denotes overlap in the nodes of the PSG for the recursive power function shown in Figure 5. These total 19 of the 27 total nodes, a 70.37% overlap.

Compared to Aroma's simplified parse tree (OOPSLA '19), PSG has greater graph node matching.

```
        return 1;
    else if (y % 2 == 0)
        return recursive_power(x, y / 2) *
            recursive_power(x, y / 2);
    else
        return x * recursive_power(x, y / 2) *
            recursive_power(x, y / 2);
}
```

Implementation 2

```
signed int iterative_power(signed int x, unsigned int y)
{
    signed int val = 1;
    while (y > 0) {
        val *= x;
        y -= 1;
    }
    return val;
}
```

PSG = PROGRAM-DERIVED SEMANTICS GRAPH

PSG OF EXPONENTIATION (POWER) IMPLEMENTED RECURSIVELY & ITERATIVELY

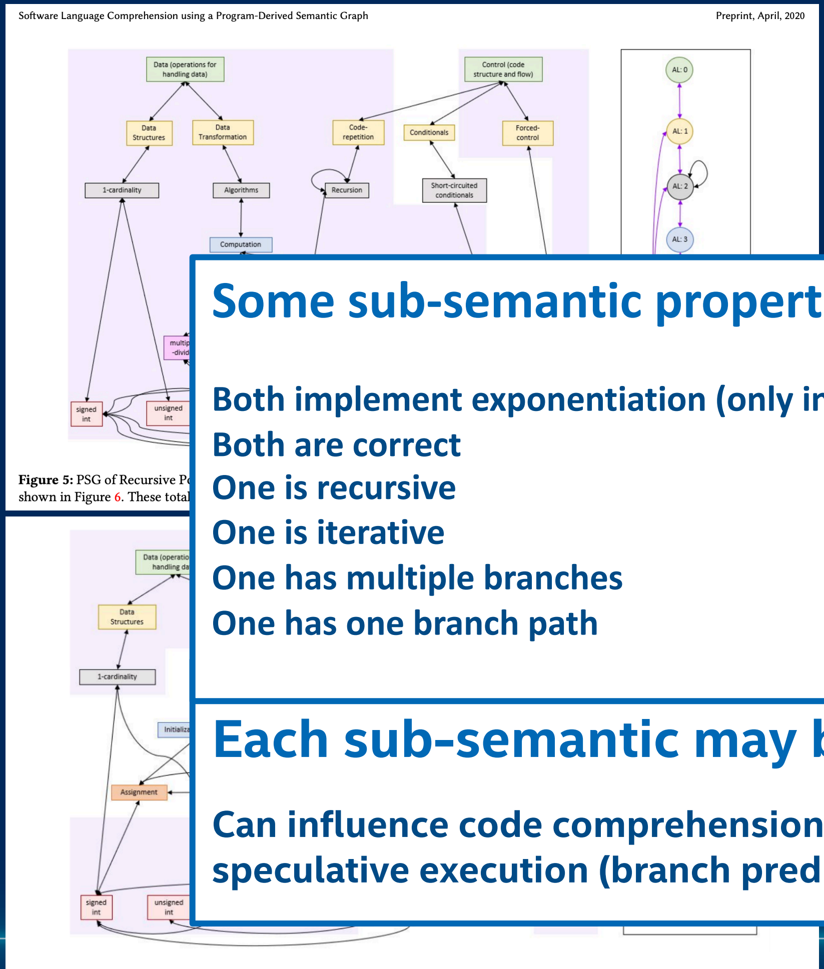


Figure 5: PSG of Recursive P... shown in Figure 6. These total...

Figure 6: PSG of Iterative Power Function. The shaded region denotes overlap in the nodes of the PSG for the recursive power function shown in Figure 5. These total 19 of the 27 total nodes, a 70.37% overlap.

Some sub-semantic properties

Both implement exponentiation (only integers)
Both are correct
One is recursive
One is iterative
One has multiple branches
One has one branch path

Each sub-semantic may be useful

Can influence code comprehension, call stacks, speculative execution (branch prediction), etc.

Compared to Aroma's simplified parse tree (OOPSLA '19), PSG has greater graph node matching.

```
    return 1;
  else if (y % 2 == 0)
    return recursive_power(x, y / 2) *
      recursive_power(x, y / 2);
  else
    return x * recursive_power(x, y / 2) *
      recursive_power(x, y / 2);
}
```

Implementation 2

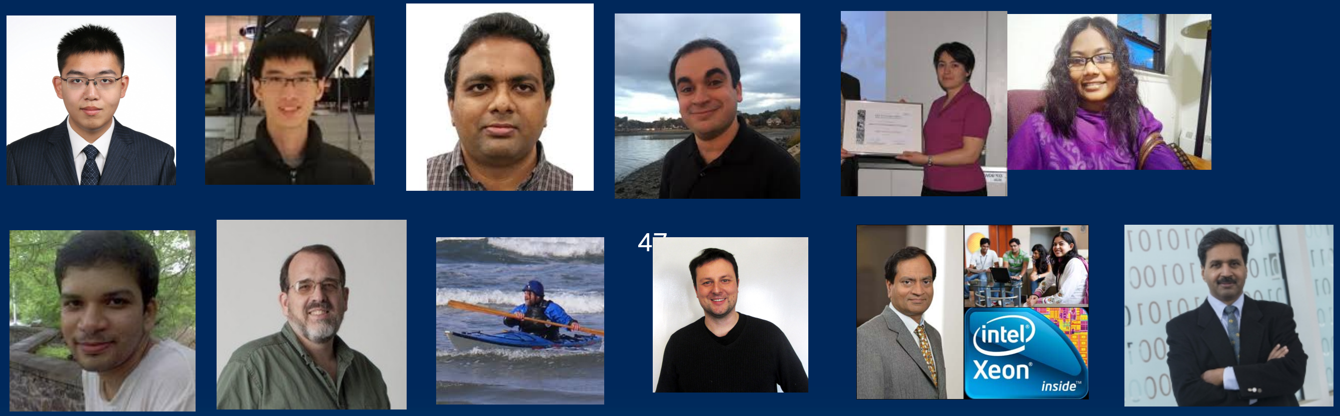
```
signed int iterative_power(signed int x, unsigned int y)
{
  signed int val = 1;
  while (y > 0) {
    val *= x;
    y -= 1;
  }
  return val;
}
```

PSG = PROGRAM-DERIVED SEMANTICS GRAPH

MISIM (MACHINE INFERRED CODE SIMILARITY)

Code semantics similarity system using:

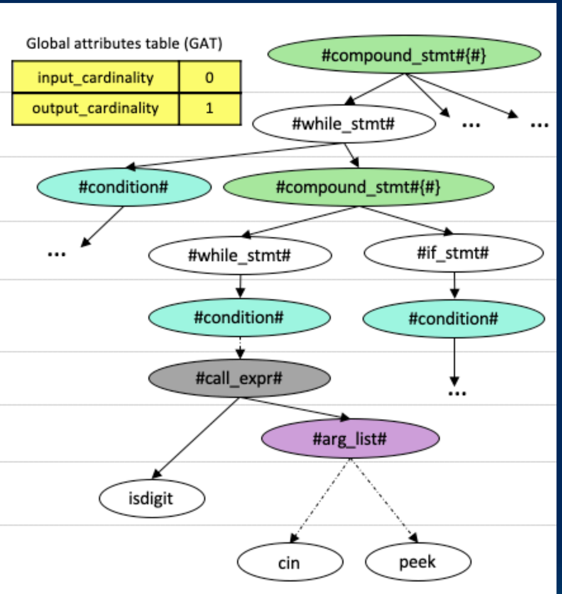
- **Determinism:**
  - new code representation (context-aware semantics structure (CASS))
- **Stochasticism:**
  - learned neural scoring algorithm



Machine Inferred Code Similarity (MISIM)

MISIM has two core novelties: one is deterministic, one is stochastic

[Deterministic] Novel code representation: context-aware semantics structure (CASS)



CASS (1-2-0-0-1)

[Stochastic] Novel learned neural scoring algorithm

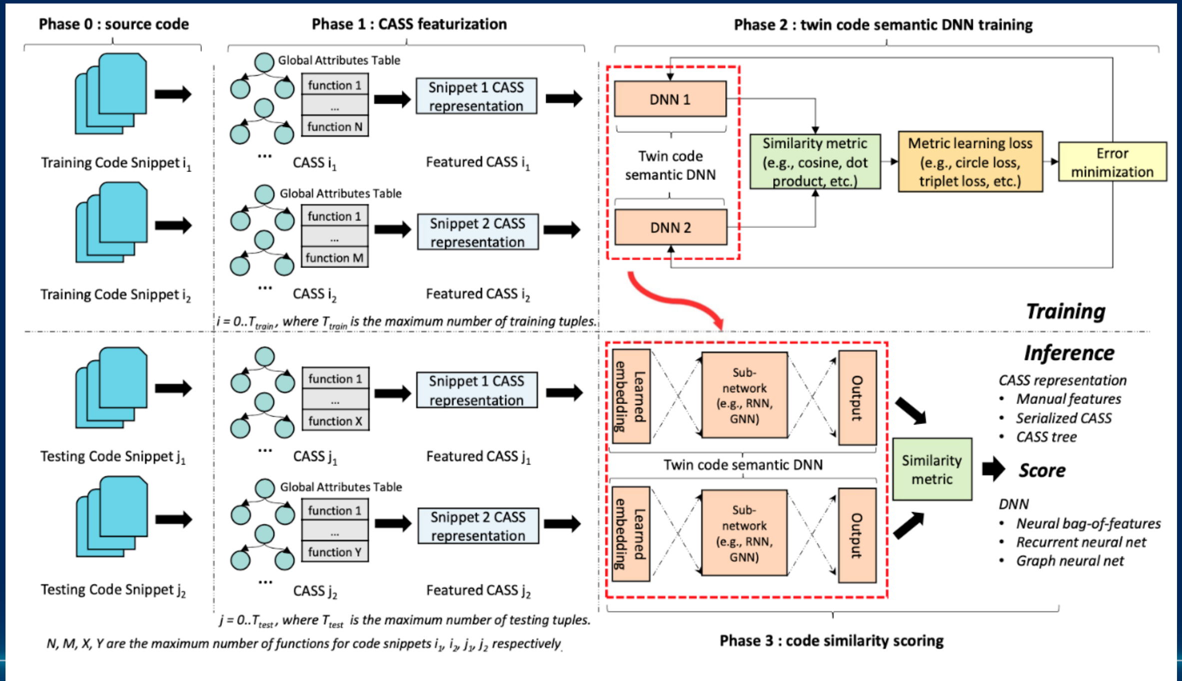


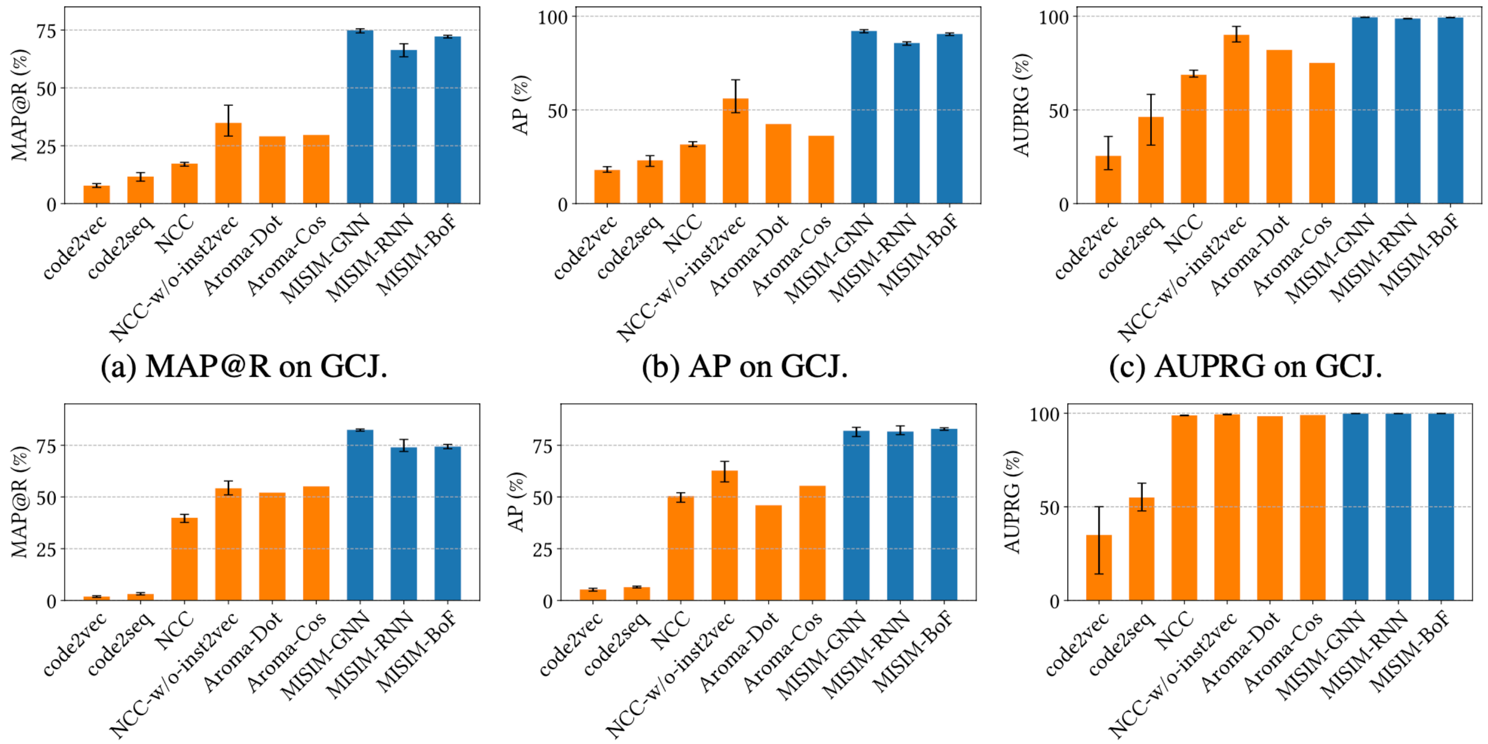
Figure 2. Overview of the MISIM System.

MISIM'S ACCURACY

Other systems =

MISIM =

- Compared to SOTA: code2vec, code2seq, NCC, and Aroma.
- Tested on ~19M LOC, 350,000 full C/C++ programs, 400 unique classes.



(a) MAP@R on GCJ. (b) AP on GCJ. (c) AUPRG on GCJ.

(d) MAP@R on POJ-104. (e) AP on POJ-104. (f) AUPRG on POJ-104.

MISIM'S ACCURACY

- IBM/MIT's Project CodeNet analysis (2021)

  https://arxiv.org/pdf/2105.12655.pdf

Table 3: Similarity MAP@R score from CodeNet (credit: [Puri et al., 2021]).

	C++1000	C++1400
Aroma	0.17	0.15
MISIM	0.75	0.75

- The C++1000 dataset consists of 1000 classes with 500k programs
- The C++1400 dataset consists of 1400 classes with 420k programs

- MISIM performed 4.4-5.0x better than Aroma for Project CodeNet across ~1M programs

- We are using MISIM (and similar systems) in-house for an upcoming new MP system

Conclusion

- Machine Programming Research charter
- Discussion of The Three Pillars of MP
  - Separation of intention, lifting code semantics
  - Intentional programming languages
- The Bifurcated Space in MP
  - Stochastic and Deterministic
- ControlFlag: A Self-Supervised Systems for MP
- MISIM: A Code Semantics Similarity System

Future and Open Invitation for Collaboration

Future directions

- Growing MP investment across all of Intel
- MPR is hiring PhD+ researchers; please reach out to me

Industrial and academic collaborations

- Teaching MP fundamentals at Berkeley and MIT, Fall 2021
- New Intel/NSF Machine Programming Research Center
- MAPS '22: **Program Chair Prof. Dr. Charles Sutton (Google AI)**

Stay current with MP and our open-sourcing

- Intel's Website, LinkedIn, Twitter, and YouTube MP Channel
- ControlFlag's open-source link:
  - https://github.com/IntelLabs/control-flag