University of Wisconsin – Machine Learning Optimizing Systems (MLOS)
(September 28, 2020)

CHALLENGES AND OPPORTUNITIES IN MACHINE PROGRAMMING (MP)

Justin Gottschlich

Principal Scientist & Director/Founder of Machine Programming Research

Intel Labs

(Incomplete) Active Collaborators: Maaz Ahmad, Todd Anderson, Saman Amarasinghe, Jim Baca, Regina Barzilay, Michael Carbin, Carlo Carino, Alvin Cheung, Pradeep Dubey, Kayvon Fatahalian, Henry Gabb, Craig Garland, Moh Haghighat, Mary Hall, Niranjan Hasabnis, Adam Herr, Jim Held, Roshni Iyer, Nilesh Jain, Tim Kraska, Brian Kroth, Insup Lee, Geoff Lowney, Shanto Mandal, Ryan Marcus, Tim Mattson, Abdullah Muzahid, Mayur Naik, Paul Petersen, Alex Ratner, Tharindu Rusira, Martin Rinard, Vivek Sarkar, Koushik Sen, Oleg Sokolsky, Armando Solar-Lezama, Julia Sukharina, Yizhou Sun, Joe Tarango, Nesime Tatbul, Josh B. Tenenbaum, Jesmin Tithi, Javier Turek, Rich Uhlig, Anand Venkat, Wei Wang, Jim Weimer, Markus Weimer, Fangke Ye, Shengtian Zhou ... and many others.

(Non-Exhaustive) MP Topics

Machine programming uses stochastic and deterministic methods

Extraction of evolving and multi-dimensional code semantics

Novel structural representations of code

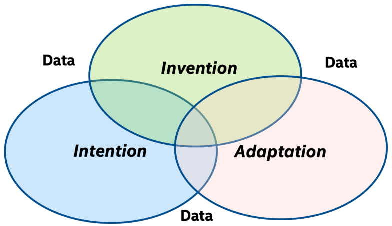
Automation for software and hardware heterogeneity

Intentional programming

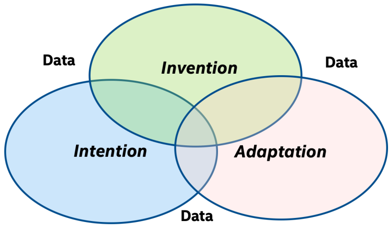
The future of data, communication, and computation for MP

But first – some background

SOME BACKGROUND



INTRODUCING:



MACHINE PROGRAMMING RESEARCH (MPR)

A NEW PIONEERING RESEARCH INITIATIVE @ INTEL

INTEL LABS' MPR GOALS

Automation of software (and hardware) to improve:

1. ***productivity:* minimal human effort***

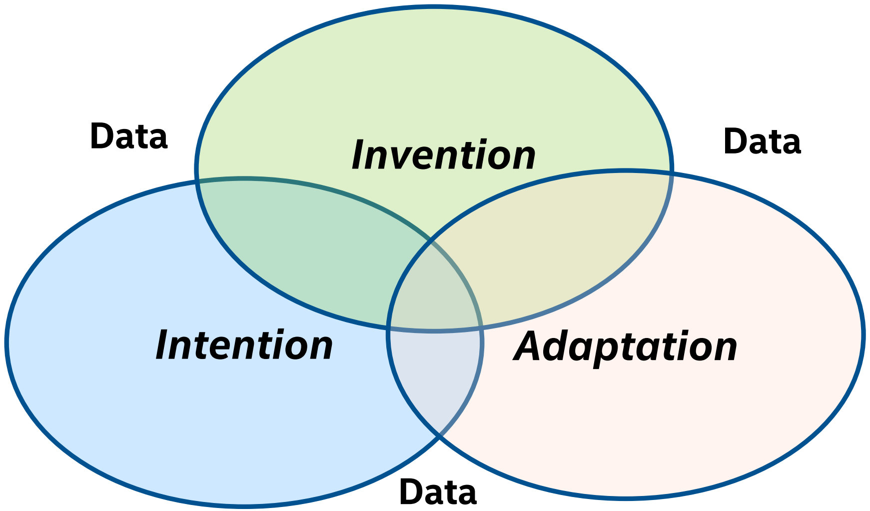
   ****Measured as 1000x+ improvement over human work performed today***

2. ***quality:* better software than the best human programmers***

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

We speculate this end-point to be at least 2+ decades away.

THE THREE PILLARS OF MACHINE PROGRAMMING (MP)



Justin Gottschlich, Intel Labs
Armando Solar-Lezama, MIT
Nesime Tatbul, Intel Labs
Michael Carbin, MIT
Martin Rinard, MIT
Regina Barzilay, MIT
Saman Amarasinghe, MIT
Joshua B Tenenbaum, MIT
Tim Mattson, Intel Labs

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

Summarized ~90 works.

Key efforts by Berkeley, Google, Microsoft, MIT, Stanford, UW and others.

WHY CALL IT MACHINE PROGRAMMING?

WHY CALL IT MACHINE PROGRAMMING?

Names matter. Should <u>infer</u> meaning from name.

WHY CALL IT MACHINE PROGRAMMING?

Names matter. Should infer meaning from name.

Why isn't this seminar series called?

- AI for Computer Science
- Neural Networks for Optimization
- Machine Learning for Software

None of these name ***precisely*** match the intention of this seminar series (as I understand it ☺).

WHY CALL IT MACHINE PROGRAMMING?

- Likewise, our alternatives were:
  - Program Synthesis
  - AI/ML for Code
  - Software 2.0

WHY CALL IT MACHINE PROGRAMMING?

- Likewise, our alternatives were:
  - Program Synthesis (historical w/ formal methods; not always synthesizing)
  - AI/ML for Code (it's not just AI/ML – this is important)
  - Software 2.0 (what does this mean?)
    - And Software 3.0, and 4.0, and 5.0?

- The ***machine programming*** name was coined to avoid confusion and broaden scope.

(NON-EXHAUSTIVE) TOPICS

MACHINE PROGRAMMING USES STOCHASTIC AND DETERMINISTIC METHODS

EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS

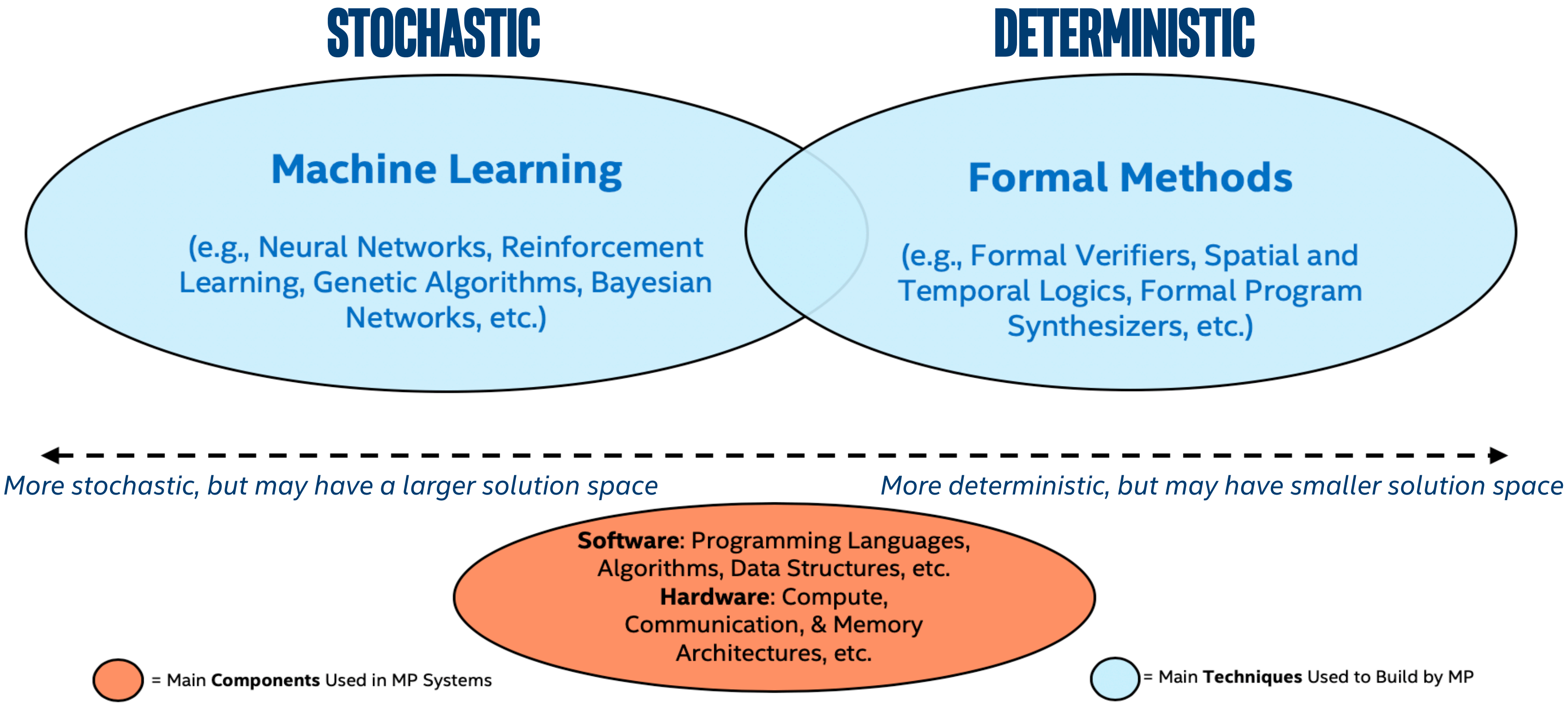
NOVEL STRUCTURAL REPRESENTATIONS OF CODE

AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY

INTENTIONAL PROGRAMMING

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

MP USES STOCHASTIC AND DETERMINISTIC METHODS



MP USES STOCHASTIC AND DETERMINISTIC METHODS

EMERGING SOLUTIONS USING A FUSION OF BOTH

MP USES STOCHASTIC AND DETERMINISTIC METHODS

EMERGING SOLUTIONS USING A FUSION OF BOTH

Learning to Infer Program Sketches

Maxwell Nye [1,2] Luke Hewitt [1,2,3] Joshua Tenenbaum [1,2,4] Armando Solar-Lezama [2]

Abstract

Our goal is to build systems which write code automatically from the kinds of specifications humans can most easily provide, such as examples and natural language instruction. The key idea of this work is that a flexible combination of pattern recognition and explicit reasoning can be used to solve these complex programming problems. We propose a method for dynamically integrating these types of information. Our novel intermediate representation and training algorithm allow a program synthesis system to learn, without direct supervision, when to rely on pattern recognition and when to perform symbolic search. Our model matches the memorization and generalization performance of neural synthesis and symbolic search, respectively, and achieves state-of-the-art performance on a dataset of simple English description-to-code programming problems.

way to combine these language constructs to construct an expression with the desired behavior.

A moderately experienced programmer might immediately *recognize*, from learned experience, that because the output list is always a subset of the input list, a `filter` function is appropriate:

```
filter(input, <HOLE>)
```

where `<HOLE>` is a lambda function which filters elements in the list. The programmer would then have to reason about the correct code for `<HOLE>`.

Finally, a programmer very familiar with this domain might immediately recognize both the need for a `filter` function, as well as the correct semantics for the lambda function, allowing the entire program to be written in one shot:

```
filter(input, lambda x:  x%2==0)
```

ICLR 2019

An Abstraction-Based Framework for Neural Network Verification

Yizhak Yisrael Elboher[1], Justin Gottschlich[2], and Guy Katz[1(✉)]

[1] The Hebrew University of Jerusalem, Jerusalem, Israel
{yizhak.elboher,g.katz}@mail.huji.ac.il
[2] Intel Labs, Santa Clara, USA
justin.gottschlich@intel.com

CAV 2020

Abstract. Deep neural networks are increasingly being used as controllers for safety-critical systems. Because neural networks are opaque, certifying their correctness is a significant challenge. To address this issue, several neural network verification approaches have recently been proposed. However, these approaches afford limited scalability, and applying them to large networks can be challenging. In this paper, we propose a framework that can enhance neural network verification techniques by using over-approximation to reduce the size of the network—thus making it more amenable to verification. We perform the approximation such that if the property holds for the smaller (abstract) network, it holds for the original as well. The over-approximation may be too coarse, in which case the underlying verification tool might return a spurious counterexample. Under such conditions, we perform counterexample-guided refinement to adjust the approximation, and then repeat the process. Our approach is orthogonal to, and can be integrated with, many existing verification techniques. For evaluation purposes, we integrate it with the recently proposed Marabou framework, and observe a significant improvement in Marabou's performance. Our experiments demonstrate the great potential of our approach for verifying larger neural networks.

MP USES STOCHASTIC AND DETERMINISTIC METHODS

FORMAL METHODS FOR INCREASED DETERMINISM OF NEURAL NETS

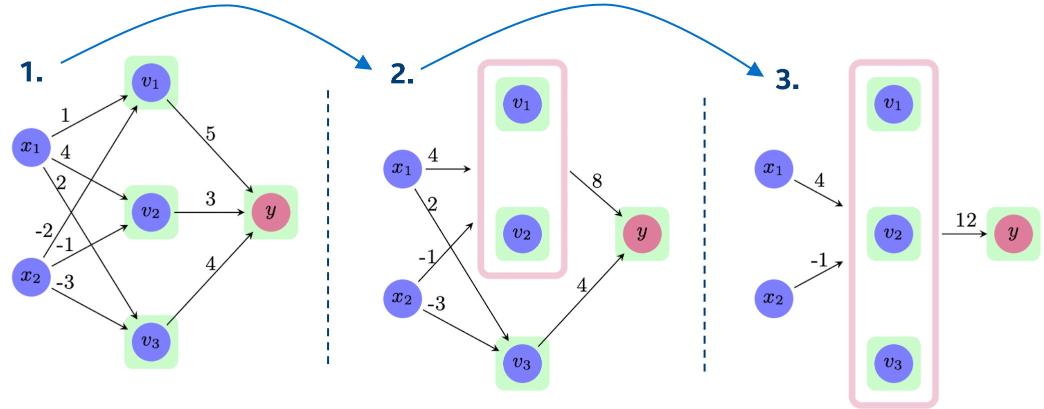
An Abstraction-Based Framework for Neural Network Verification

Yizhak Yisrael Elboher[1], Justin Gottschlich[2], and Guy Katz[1(✉)]

[1] The Hebrew University of Jerusalem, Jerusalem, Israel
{yizhak.elboher,g.katz}@mail.huji.ac.il
[2] Intel Labs, Santa Clara, USA
justin.gottschlich@intel.com

Abstract. Deep neural networks are increasingly being used as controllers for safety-critical systems. Because neural networks are opaque, certifying their correctness is a significant challenge. To address this issue, several neural network verification approaches have recently been proposed. However, these approaches afford limited scalability, and applying them to large networks can be challenging. In this paper, we propose a framework that can enhance neural network verification techniques by using over-approximation to reduce the size of the network—thus making it more amenable to verification. We perform the approximation such that if the property holds for the smaller (abstract) network, it holds for the original as well. The over-approximation may be too coarse, in which case the underlying verification tool might return a spurious counterexample. Under such conditions, we perform counterexample-guided refinement to adjust the approximation, and then repeat the process. Our approach is orthogonal to, and can be integrated with, many existing verification techniques. For evaluation purposes, we integrate it with the recently proposed Marabou framework, and observe a significant improvement in Marabou's performance. Our experiments demonstrate the great potential of our approach for verifying larger neural networks.

CAV 2020



$$y = 5R(x_1 - 2x_2) + 3R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$$

$$y = 8R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$$

$$y = 12R(4x_1 - x_2)$$

NEURON COALESCENCE VIA MATHEMATICAL TRANSITIVITY

USING COUNTEREXAMPLE-GUIDED ABSTRACTION REFINEMENT

MACHINE PROGRAMMING + DEEP LEARNING = NEURAL PROGRAMMING?

Neural Programming: use of neural networks as a replacement of code.

MACHINE PROGRAMMING + DEEP LEARNING = NEURAL PROGRAMMING?

Neural Programming: use of neural networks as a replacement of code.

Learning to Optimize

Ke Li **Jitendra Malik**
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720
United States
{ke.li,malik}@eecs.berkeley.edu

ICLR 2017

Abstract

Algorithm design is a laborious process and often requires many iterations of ideation and validation. In this paper, we explore automating algorithm design and present a method to *learn* an optimization algorithm, which we believe to be the first method that can automatically discover a better algorithm. We approach this problem from a reinforcement learning perspective and represent any particular optimization algorithm as a policy. We learn an optimization algorithm using guided policy search and demonstrate that the resulting algorithm outperforms existing hand-engineered algorithms in terms of convergence speed and/or the final objective value.

A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions

Mejbah Alam
Intel Labs
mejbah.alam@intel.com

Justin Gottschlich
Intel Labs
justin.gottschlich@intel.com

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

Javier Turek
Intel Labs
javier.turek@intel.com

Timothy Mattson
Intel Labs
timothy.g.mattson@intel.com

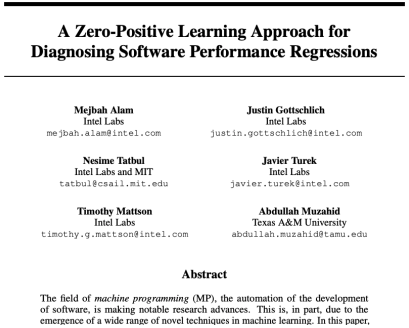
Abdullah Muzahid
Texas A&M University
abdullah.muzahid@tamu.edu

NEURIPS 2019

Abstract

The field of *machine programming* (MP), the automation of the development of software, is making notable research advances. This is, in part, due to the emergence of a wide range of novel techniques in machine learning. In this paper,

AUTOPERF: PERFORMANCE REGRESSION TESTING



A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions

Mejbah Alam
Intel Labs
mejbah.alam@intel.com

Justin Gottschlich
Intel Labs
justin.gottschlich@intel.com

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

Javier Turek
Intel Labs
javier.turek@intel.com

Timothy Mattson
Intel Labs
timothy.g.mattson@intel.com

Abdullah Muzahid
Texas A&M University
abdullah.muzahid@tamu.edu

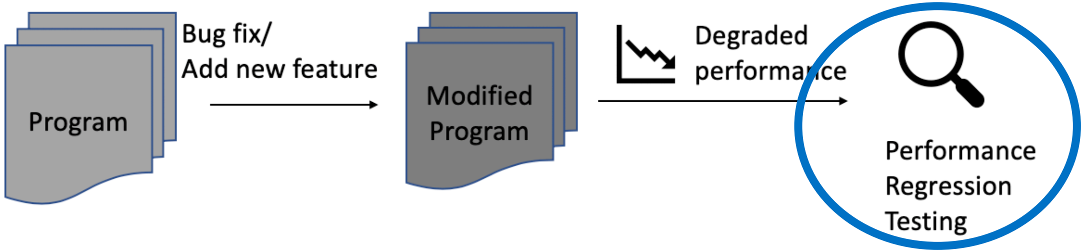
Abstract

The field of *machine programming* (MP), the automation of the development of software, is making notable research advances. This is, in part, due to the emergence of a wide range of novel techniques in machine learning. In this paper,

NEURIPS 2019

AutoPerf ***invents*** and ***adapts*** these tests

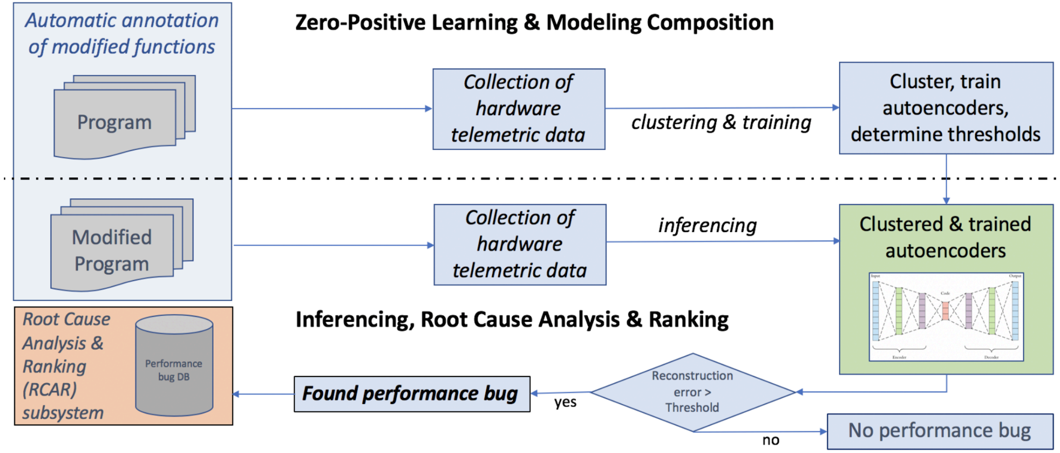
Performance Regression Testing



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

AUTOPERF: PERFORMANCE REGRESSION TESTING

- **Uses zero-positive learning (ZPL), autoencoders, hardware telemetry**
- **Emits no false negatives (no missed performance bugs)**
- **Negligible (4%) performance overhead using hardware performance counters (HWPCs)**



AutoPerf System Design

How is this neural programming?

ML ***invents*** the regression tests and ***adapts*** them to the specialized hardware to analyze performance.

NN ***is*** the code/test.

MACHINE PROGRAMMING + DEEP LEARNING = NEURAL PROGRAMMING?

SOME CONCERNS W/ NEURAL PROGRAMMING

ONLY IMPROVED BY RETRAINING?

UNDERSTANDABLE, INTERPRETABLE, DEBUGGABLE?

THERE ARE OTHER MP APPROACHES THAT GENERATE ACTUAL CODE

(WE'LL SEE SOME EXAMPLES TODAY)

(NON-EXHAUSTIVE) TOPICS

MACHINE PROGRAMMING USES STOCHASTIC AND DETERMINISTIC METHODS

EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS

NOVEL STRUCTURAL REPRESENTATIONS OF CODE

AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY

INTENTIONAL PROGRAMMING

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS

EXTRACTION OF ~~EVOLVING AND MULTI-DIMENSIONAL~~ CODE SEMANTICS

EXTRACTION OF CODE SEMANTICS

- **Why care about code semantics?**

EXTRACTION OF CODE SEMANTICS

- **Why care about code semantics?**

HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS

Elizabeth Dinella*
University of Pennsylvania

Hanjun Dai*
Google Brain

Ziyang Li
University of Pennsylvania

Mayur Naik
University of Pennsylvania

Le Song
Georgia Tech

Ke Wang
Visa Research

ICLR 2020

ABSTRACT

We present a learning-based approach to detect and fix a broad range of bugs in Javascript programs. We frame the problem in terms of learning a sequence of graph transformations: given a buggy program modeled by a graph structure, our model makes a sequence of predictions including the position of bug nodes and corresponding graph edits to produce a fix. Unlike previous works built upon deep neural networks, our approach targets bugs that are more diverse and complex in nature (i.e. bugs that require adding or deleting statements to fix). We have realized our approach in a tool called HOPPITY. By training on 290,715 Javascript code change commits on Github, HOPPITY correctly detects and fixes bugs in 9,490 out of 36,361 programs in an end-to-end fashion. Given the bug location and type of the fix, HOPPITY also outperforms the baseline approach by a wide margin.

HOPPITY: CODE REPAIR AS GRAPH TRANSFORMATIONS

- **Example of Hoppity's bug repair graph transformation**

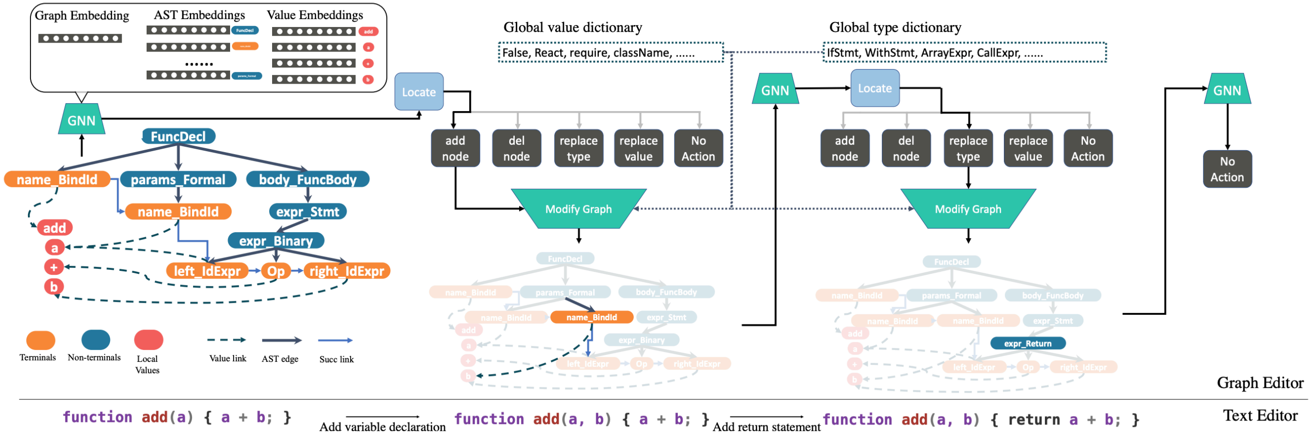


Figure 2: Code repair as graph transformation. Each step the source code graph is edited via one of the operator module until STOP is triggered by controller.

- **How does Hoppity *find* bug fixes to learn from?**

HOW DOES HOPPITY FIND BUG FIXES TO LEARN FROM?

Given a commit, we download the Javascript file before and after the change: $(src_{buggy}, src_{fixed})$. Commits can contain many types of changes such as feature additions, refactorings, bug fixes, etc. In an attempt to filter our dataset to only include bug fixes, we use a heuristic based on the number of changes to the AST. Our insight is that a commit with a smaller number of AST differences is more likely to be a bug fix than a commit containing large changes. Thus for the experiments, we use three different datasets: `OneDiff` with precisely one edit; `ZeroOneDiff` with zero and one edit and `ZeroOneTwoDiff` with zero, one or two edits. We additionally filter out data points with ASTs larger than 500 nodes as a parameter in our system. A detailed overview of our corpus crawler is available in Appendix B.

- **Looks at repo changesets – if small enough, deem a potential bug fix**
  - **Infers *bug fix* semantics on repository delta size**

- **How would Hoppity perform if the *<u>semantics</u>* of *bug fix* are known?**
  - **What about other environmental factors that could be inferred?**

WHY EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS?

WHY EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS?

- Evolving:
  - Code that is used, tends to be maintained
    - “Software that is used is never finished”
  - Evolving code == evolving semantics?

- Multi-dimensional:
  - A code snippet may have multiple semantic meanings
  - A bit more challenging to understanding …

WHY MULTI-DIMENSIONAL CODE SEMANTICS?

Software Language Comprehension using a Program-Derived Semantic Graph

Roshni G. Iyer
University of California, Los Angeles, USA
roshnigiyer@cs.ucla.edu

Yizhou Sun
University of California, Los Angeles, USA
yzsun@cs.ucla.edu

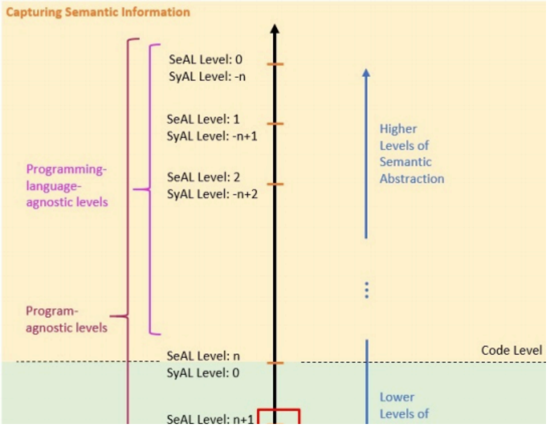
Wei Wang
University of California, Los Angeles, USA
weiwang@cs.ucla.edu

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

PREPRINT

ABSTRACT

Traditional code transformation structures, such as an abstract syntax tree, may have limitations in their ability to extract semantic meaning from code. Others have begun to work on this issue, such as the state-of-the-art Aroma system and its simplified parse tree (SPT). Continuing this research direction, we present a new graphical structure to capture semantics from code using what we refer to as a *program-derived semantic graph* (PSG). The principle behind the PSG is to provide a single structure that can capture program semantics at many levels of granularity. Thus, the PSG is hierarchical in nature. Moreover, because the PSG may have cycles due to dependencies in semantic layers, it is a graph, not a tree. In this paper, we describe the PSG and its fundamental structural differences to the Aroma's SPT. Although our work in the PSG is in its infancy, our early results indicate it is a promising new research direction to explore to automatically extract program semantics.



WHY MULTI-DIMENSIONAL CODE SEMANTICS?

Software Language Comprehension using a Program-Derived Semantic Graph

Preprint, April, 2020

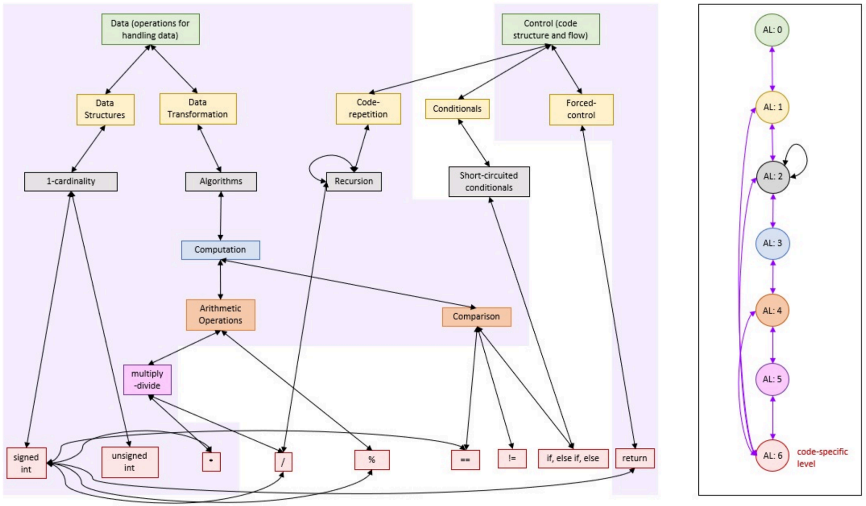


Figure 5: PSG of Recursive Power Function. The shaded region denotes overlap in the nodes of the PSG for the iterative power function shown in Figure 6. These total 17 of the 24 total nodes, a 70.83% overlap.

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);
}
```

WHY MULTI-DIMENSIONAL CODE SEMANTICS?

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

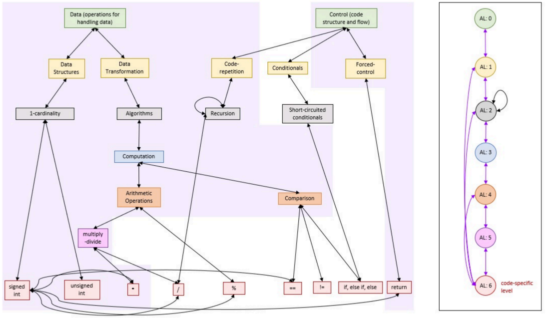


Figure 5: PSG of Recursive Power Function. The shaded region denotes overlap in the nodes of the PSG for the iterative power function shown in Figure 6. These total 17 of the 24 total nodes, a 70.83% overlap.

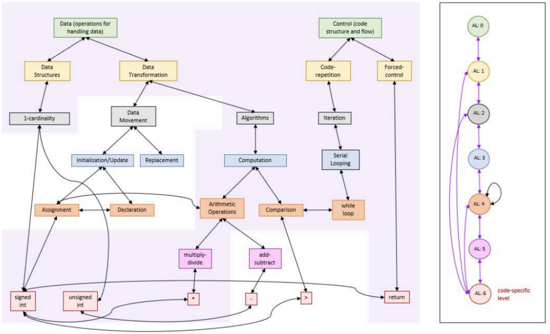


Figure 6: PSG of Iterative Power Function. The shaded region denotes overlap in the nodes of the PSG for the recursive power function shown in Figure 5. These total 19 of the 27 total nodes, a 70.37% overlap.

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;
}
```

WHY MULTI-DIMENSIONAL CODE SEMANTICS?

Some semantics:

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

Each semantic may be useful.

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

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;
}
```

EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS

SOME THOUGHTS:

BIG AND DENSE CODE, FEW SEMANTIC CODE LABELS

DISCOVER NOVEL WAYS TO LIFT SEMANTICS

LIFT SEMANTICS WITHOUT COMPILATION (WORKS WITH BROKEN CODE)?

FIND SEMANTICS FROM SURROUNDINGS?

(NON-EXHAUSTIVE) TOPICS

- MACHINE PROGRAMMING USES STOCHASTIC AND DETERMINISTIC METHODS
- EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS
- NOVEL STRUCTURAL REPRESENTATIONS OF CODE
- AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY
- INTENTIONAL PROGRAMMING
- THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

NOVEL STRUCTURAL REPRESENTATIONS OF CODE

- **Why do we need new code structures?**

NOVEL STRUCTURAL REPRESENTATIONS OF CODE

- **Why do we need new code structures?**

Aroma: Code Recommendation via Structural Code Search

SIFEI LUAN, Facebook, USA
DI YANG*, University of California, Irvine, USA
CELESTE BARNABY, Facebook, USA
KOUSHIK SEN†, University of California, Berkeley, USA
SATISH CHANDRA, Facebook, USA

OOPSLA 2019

Programmers often write code that has similarity to existing code written somewhere. A tool that could help programmers to search such similar code would be immensely useful. Such a tool could help programmers to extend partially written code snippets to completely implement necessary functionality, help to discover extensions to the partial code which are commonly included by other programmers, help to cross-check against similar code written by other programmers, or help to add extra code which would fix common mistakes and errors. We propose Aroma, a tool and technique for code recommendation via structural code search. Aroma indexes a huge code corpus including thousands of open-source projects, takes a partial code snippet as input, searches the corpus for method bodies containing the partial code snippet, and clusters and intersects the results of the search to recommend a small set of succinct code snippets which both contain the query snippet and appear as part of several methods in the corpus. We evaluated Aroma on 2000 randomly selected queries created from the corpus, as well as 64 queries derived from code snippets obtained from Stack Overflow, a popular website for discussing code. We implemented Aroma for 4 different languages, and developed an IDE plugin for Aroma. Furthermore, we conducted a study where we asked 12 programmers to complete programming tasks using Aroma, and collected their feedback. Our results indicate that Aroma is capable of retrieving and recommending relevant code snippets efficiently.

MISIM: An End-to-End Neural Code Similarity System

Fangke Ye *
Intel Labs and Georgia Institute of Technology
yefangke@gatech.edu

Shengtian Zhou *
Intel Labs
shengtian.zhou@intel.com

Anand Venkat
Intel Labs
anand.venkat@intel.com

Ryan Marcus
Intel Labs and MIT
ryanmarcus@csail.mit.edu

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

Jesmin Jahan Tithi
Intel Labs
jesmin.jahan.tithi@intel.com

Paul Petersen
Intel
paul.petersen@intel.com

Timothy Mattson
Intel Labs
timothy.g.mattson@intel.com

Tim Kraska
MIT
kraska@mit.edu

Pradeep Dubey
Intel Labs
pradeep.dubey@intel.com

Vivek Sarkar
Georgia Institute of Technology
vsarkar@gatech.edu

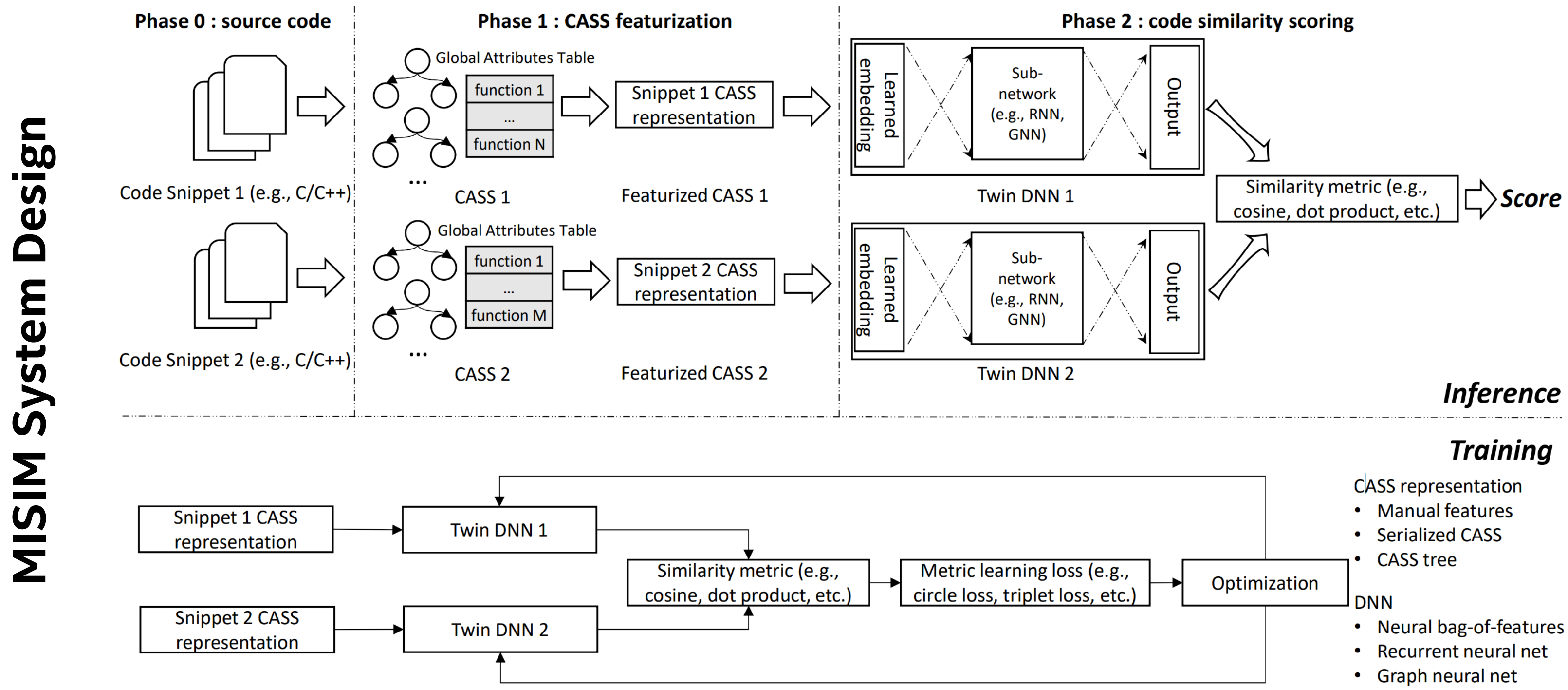
Justin Gottschlich
Intel Labs and University of Pennsylvania
justin.gottschlich@intel.com

PREPRINT

Abstract

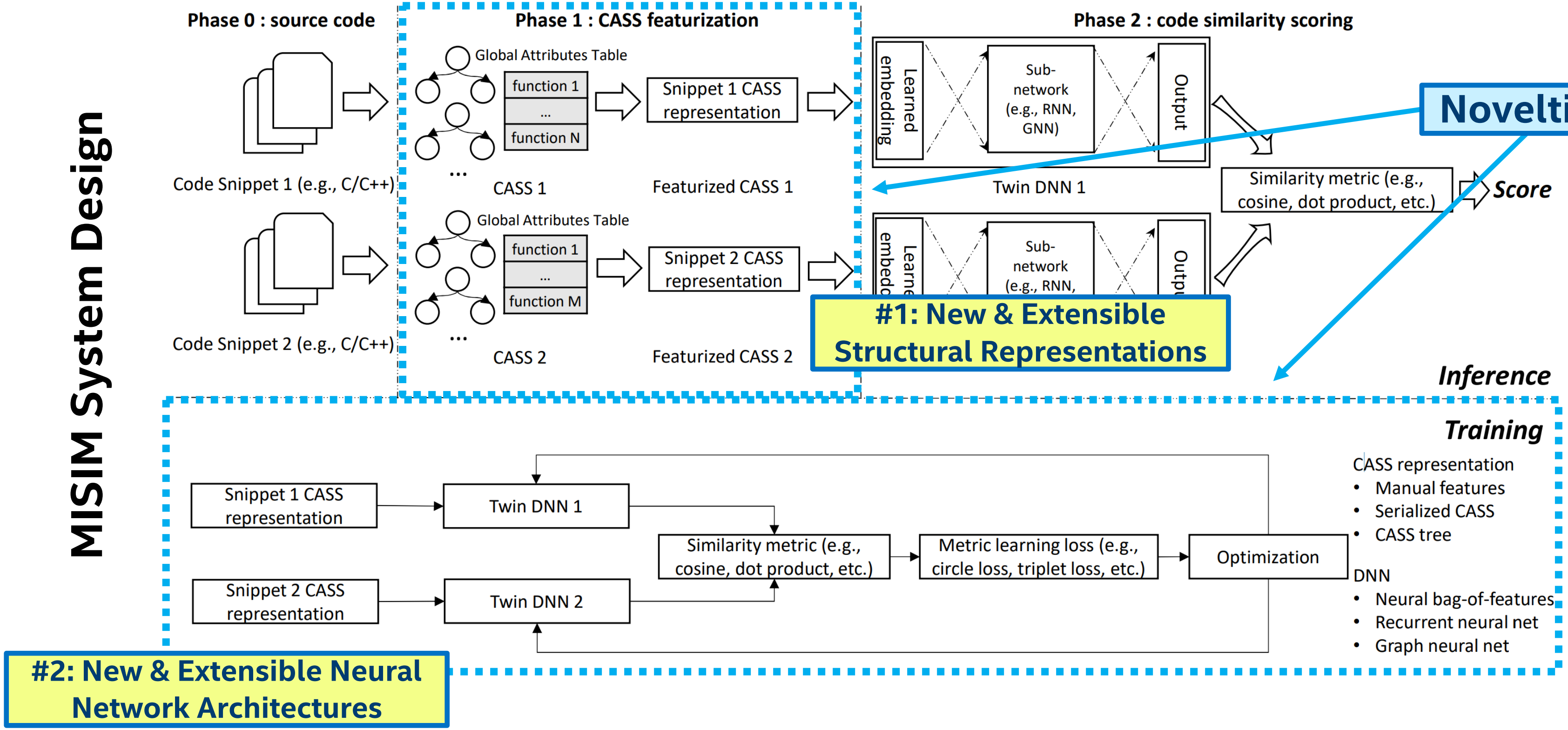
Code similarity systems are integral to a range of applications from code recommendation to automated construction of software tests and defect mitigation. In this paper, we present *Machine Inferred Code Similarity* (MISIM), a novel end-to-end code similarity system that consists of two core components. First, MISIM uses a novel *context-aware semantic structure*, which is designed to aid in lifting semantic meaning from code syntax. Second, MISIM provides a neural-based code similarity scoring algorithm, which can be implemented with various neural network architectures with learned parameters. We compare MISIM to three state-of-the-art code similarity systems: *(i)* code2vec, *(ii)* Neural Code Comprehension, and *(iii)* Aroma. In our experimental evaluation across 45,780 programs, MISIM consistently outperformed all three systems, often by a large factor (upwards of 40.6×).

MACHINE INFERRED CODE SIMILARITY (MISIM)



MISIM created by Intel, Georgia Tech, and MIT

MACHINE INFERRED CODE SIMILARITY (MISIM)



NOVEL STRUCTURAL REPRESENTATIONS OF CODE

- Aroma introduced the *simplified parse tree (SPT)*
- MISIM introduced the *context-aware semantics structure (CASS)*
- Both intentionally moved away from classical structures (like AST)

These structures have led to state-of-the-art accuracy

Take-away:

Historical code representations may restrict our thinking for pioneering research in MP. Let's not do that. ☺

MISIM: An End-to-End Neural Code Similarity System

Fangke Ye *
Intel Labs and Georgia Institute of Technology
yefangke@gatech.edu

Shengtian Zhou *
Intel Labs
shengtian.zhou@intel.com

Anand Venkat
Intel Labs
anand.venkat@intel.com

Ryan Marcus
Intel Labs and MIT
ryanmarcus@csail.mit.edu

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

Jesmin Jahan Tithi
Intel Labs
jesmin.jahan.tithi@intel.com

Paul Petersen
Intel
paul.petersen@intel.com

Timothy Mattson
Intel Labs
timothy.g.mattson@intel.com

Tim Kraska
MIT
kraska@mit.edu

Pradeep Dubey
Intel Labs
pradeep.dubey@intel.com

Vivek Sarkar
Georgia Institute of Technology
vsarkar@gatech.edu

Justin Gottschlich
Intel Labs and University of Pennsylvania
justin.gottschlich@intel.com

PREPRINT

Abstract

Code similarity systems are integral to a range of applications from code recommendation to automated construction of software tests and defect mitigation. In this paper, we present *Machine Inferred Code Similarity* (MISIM), a novel end-to-end code similarity system that consists of two core components. First, MISIM uses a novel *context-aware semantic structure*, which is designed to aid in lifting semantic meaning from code syntax. Second, MISIM provides a neural-based code similarity scoring algorithm, which can be implemented with various neural network architectures with learned parameters. We compare MISIM to three state-of-the-art code similarity systems: *(i)* code2vec, *(ii)* Neural Code Comprehension, and *(iii)* Aroma. In our experimental evaluation across 45,780 programs, MISIM consistently outperformed all three systems, often by a large factor (upwards of 40.6×).

Aroma: Code Recommendation via Structural Code Search

SIFEI LUAN, Facebook, USA
DI YANG*, University of California, Irvine, USA
CELESTE BARNABY, Facebook, USA
KOUSHIK SEN†, University of California, Berkeley, USA
SATISH CHANDRA, Facebook, USA

Programmers often write code that has similarity to existing code written somewhere. A tool that could help programmers to search such similar code would be immensely useful. Such a tool could help programmers to extend partially written code snippets to completely implement necessary functionality, help to discover extensions to the partial code which are commonly included by other programmers, help to cross-check against similar code written by other programmers, or help to add extra code which would fix common mistakes and errors. We propose Aroma, a tool and technique for code recommendation via structural code search. Aroma indexes a huge code corpus including thousands of open-source projects, takes a partial code snippet as input, searches the corpus for method bodies containing the partial code snippet, and clusters and intersects the results of the search to recommend a small set of succinct code snippets which both contain the query snippet and appear as part of several methods in the corpus. We evaluated Aroma on 2000 randomly selected queries created from the corpus, as well as 64 queries derived from code snippets obtained from Stack Overflow, a popular website for discussing code. We implemented Aroma for 4 different languages, and developed an IDE plugin for Aroma. Furthermore, we conducted a study where we asked 12 programmers to complete programming tasks using Aroma, and collected their feedback. Our results indicate that Aroma is capable of retrieving and recommending relevant code snippets efficiently.

OOPSLA 2019

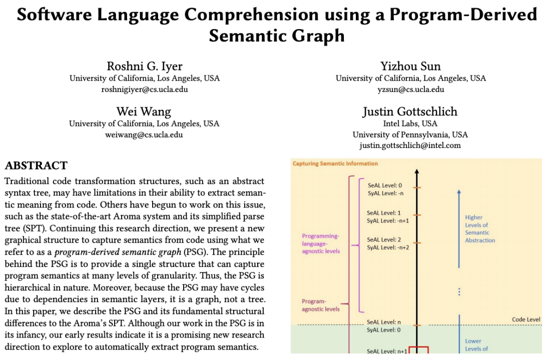
NOVEL STRUCTURAL REPRESENTATIONS OF CODE

SOME CHALLENGES:

GOOD EARLY PROGRESS

MORE STRUCTURES TO DISCOVER / PROBLEMS TO SOLVE
(E.G., HOW TO BUILD THE PROGRAM-DERIVED SEMANTICS GRAPH?)

I BELIEVE A NEW CLASS OF STRUCTURES ARE ABOUT TO EMERGE:
STRUCTURES THAT CAN ONLY BE LEARNED



Software Language Comprehension using a Program-Derived Semantic Graph

Roshni G. Iyer
University of California, Los Angeles, USA
roshnigiyer@cs.ucla.edu

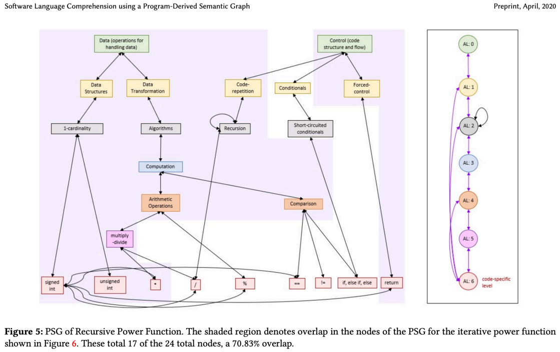
Yizhou Sun
University of California, Los Angeles, USA
yzsun@cs.ucla.edu

Wei Wang
University of California, Los Angeles, USA
weiwang@cs.ucla.edu

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

ABSTRACT

Traditional code transformation structures, such as an abstract syntax tree, may have limitations in their ability to extract semantic meaning from code. Others have begun to work on this issue, such as the state-of-the-art Aroma system and its simplified parse tree (SPT). Continuing this research direction, we present a new graphical structure to capture semantics from code using what we refer to as a *program-derived semantic graph* (PSG). The principle behind the PSG is to provide a single structure that can capture program semantics at many levels of granularity. Thus, the PSG is hierarchical in nature. Moreover, because the PSG may have cycles due to dependencies in semantic layers, it is a graph, not a tree. In this paper, we describe the PSG and its fundamental structural differences to the Aroma's SPT. Although our work in the PSG is in its infancy, our early results indicate it is a promising new research direction to explore to automatically extract program semantics.



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.

(NON-EXHAUSTIVE) TOPICS

- MACHINE PROGRAMMING USES STOCHASTIC AND DETERMINISTIC METHODS
- EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS
- NOVEL STRUCTURAL REPRESENTATIONS OF CODE
- AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY
- INTENTIONAL PROGRAMMING
- THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY

- **SW / HW heterogeneity is creating multiplicative complexity**

AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY

Verified Lifting:
Target multiple programming languages

Automatically Translating Image Processing Libraries to Halide

MAAZ BIN SAFEER AHMAD, University of Washington, Seattle
JONATHAN RAGAN-KELLEY, University of California, Berkeley
ALVIN CHEUNG, University of California, Berkeley
SHOAIB KAMIL, Adobe

SIGGRAPH ASIA 2019

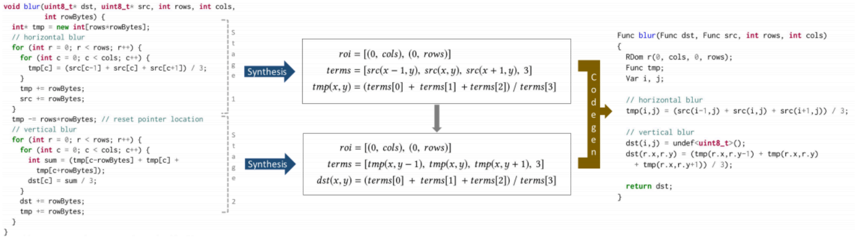


Fig. 1. DEXTER parses the input C++ function (shown on the left) into a DAG of smaller stages, then uses our 3-step synthesis algorithm to infer the semantics of each stage, expressed in a high-level IR (middle). Finally, code generation rules compile the IR specifications into executable Halide code (right).

Halide:
Target multiple hardware compute devices

Automatically Scheduling Halide Image Processing Pipelines

Ravi Teja Mullapudi* Andrew Adams‡ Dillon Sharlet‡ Jonathan Ragan-Kelley† Kayvon Fatahalian*
*Carnegie Mellon University ‡Google †Stanford University

Abstract

The Halide image processing language has proven to be an effective system for authoring high-performance image processing code. Halide programmers need only provide a high-level strategy for mapping an image processing pipeline to a parallel machine (a *schedule*), and the Halide compiler carries out the mechanical task of generating platform-specific code that implements the schedule. Unfortunately, designing high-performance schedules for complex image processing pipelines requires substantial knowledge of modern hardware architecture and code-optimization techniques. In this paper we provide an algorithm for automatically generating high-performance schedules for Halide programs. Our solution extends the function bounds analysis already present in the Halide compiler to automatically perform locality and parallelism-enhancing global program transformations typical of those employed by expert Halide developers. The algorithm does not require costly (and often impractical) auto-tuning, and, in seconds, generates schedules for a broad set of image processing benchmarks that are performance-competitive with, and often better than, schedules manually authored by expert Halide developers on server and mobile CPUs, as well as GPUs.

Keywords: image processing, optimizing compilers, Halide

Concepts: •**Computing methodologies** → *Graphics systems an interfaces;*

algorithm's execution on a machine (called a *schedule*). The Halide compiler then handles the tedious, mechanical task of generating platform-specific code that implements the schedule (e.g., spawning threads, managing buffers, generating SIMD instructions).

Although Halide provides high-level abstractions for expressing schedules, *designing* schedules that perform well on modern hardware is hard; it requires expertise in modern optimization techniques and hardware architectures. For example, around 70 software engineers at Google currently write image processing algorithms in Halide, but they rely on a much smaller cadre of Halide scheduling experts to produce the most efficient implementations. Further, production image processing pipelines are long and complex, and are difficult to schedule even for the best Halide programmers. Arriving at a good schedule remains a laborious, iterative process of schedule tweaking and performance measurement. Also, in large production pipelines, software engineering considerations (e.g., modularity, code reuse) may preclude experts from having the global program knowledge needed to create optimal schedules.

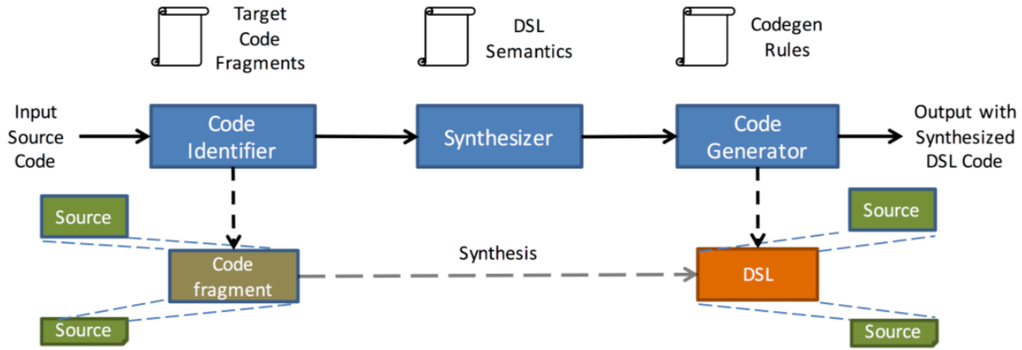
In this paper we address this problem by providing an algorithm

SIGGRAPH 2016

VERIFIED LIFTING EVOLUTION: METALIFT

MetaLift

Leveraging DSLs made easy



People

MetaLift is jointly developed by the folks at the University of Washington Programming Languages and Software Engineering Research Group, Adobe Research, and Intel Labs. The following are the main developers of MetaLift:

Verified lifting has been the underlying technology used to build the following compilers:

Dexter is a compiler that translates image processing kernels from C to Halide.

Casper is a compiler that translates sequential Java to Spark and Hadoop.

STNG

STNG is a compiler that enables Fortran kernels to leverage GPUs by compiling them into the Halide DSL.

AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY

AN OPEN QUESTION:

WHAT ARE THE QUALITY METRICS FOR HETEROGENEOUS TRANSLATION?

CORRECT & PERFORMANCE (OF COURSE)

WHAT ABOUT SECURITY, MAINTAINABILITY, POWER FOOTPRINT, ETC.?

(NON-EXHAUSTIVE) TOPICS

MACHINE PROGRAMMING USES STOCHASTIC AND DETERMINISTIC METHODS

EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS

NOVEL STRUCTURAL REPRESENTATIONS OF CODE

AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY

INTENTIONAL PROGRAMMING

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

INTENTIONAL PROGRAMMING

- Focus on *what* the intention is, not *how* that intention may manifest

INTENTIONAL PROGRAMMING

- **Focus on *what* the intention is, not *how* that intention may manifest**

Learning to Represent Programs with Property Signatures

Augustus Odena, Charles Sutton
Google Research
{augustusodena,charlessutton}@google.com

Abstract

We introduce the notion of property signatures, a representation for programs and program specifications meant for consumption by machine learning algorithms. Given a function with input type τ_{in} and output type τ_{out}, a property is a function of type: $(\tau_{in}, \tau_{out}) \rightarrow$ `Bool` that (informally) describes some simple property of the function under consideration. For instance, if τ_{in} and τ_{out} are both lists of the same type, one property might ask 'is the input list the same length as the output list?'. If we have a list of such properties, we can evaluate them all for our function to get a list of outputs that we will call the property signature. Crucially, we can 'guess' the property signature for a function given only a set of input/output pairs meant to specify that function. We discuss several potential applications of property signatures and show experimentally that they can be used to improve over a baseline synthesizer so that it emits twice as many programs in less than one-tenth of the time.

ICLR 2020

INTENTIONAL PROGRAMMING

- **Focus on *what* the intention is, not *how* that intention may manifest**

LEARNING TO REPRESENT PROGRAMS WITH PROPERTY SIGNATURES

Augustus Odena, Charles Sutton
Google Research
{augustusodena,charlessutton}@google.com

ABSTRACT

We introduce the notion of property signatures, a representation for programs and program specifications meant for consumption by machine learning algorithms. Given a function with input type τ_{in} and output type τ_{out}, a property is a function of type: $(\tau_{in}, \tau_{out}) \rightarrow \texttt{Bool}$ that (informally) describes some simple property of the function under consideration. For instance, if τ_{in} and τ_{out} are both lists of the same type, one property might ask 'is the input list the same length as the output list?'. If we have a list of such properties, we can evaluate them all for our function to get a list of outputs that we will call the property signature. Crucially, we can 'guess' the property signature for a function given only a set of input/output pairs meant to specify that function. We discuss several potential applications of property signatures and show experimentally that they can be used to improve over a baseline synthesizer so that it emits twice as many programs in less than one-tenth of the time.

ICLR 2020

If you haven't read this paper, please read it!

- **Identify semantics of code**
- **Provide function semantics signatures**
- **Powerful & elegant**

INTENTIONAL PROGRAMMING WITH HALIDE

- Focus on *what* the intention is, not *how* that intention may manifest

- **Halide is a domain-specific language (DSL)**
- **Separation of concerns**
  - **Splits programming intention from programming adaptation**

Learning to Optimize Halide with Tree Search and Random Programs

ANDREW ADAMS, Facebook AI Research
KARIMA MA, UC Berkeley
LUKE ANDERSON, MIT CSAIL
RIYADH BAGHDADI, MIT CSAIL
TZU-MAO LI, MIT CSAIL
MICHAËL GHARBI, Adobe
BENOIT STEINER, Facebook AI Research
STEVEN JOHNSON, Google
KAYVON FATAHALIAN, Stanford University
FRÉDO DURAND, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley

We present a new algorithm to automatically schedule Halide programs for high-performance image processing and deep learning. We significantly improve upon the performance of previous methods, which considered a limited subset of schedules. We define a parameterization of possible schedules much larger than prior methods and use a variant of beam search to search over it. The search optimizes runtime predicted by a cost model based on a combination of new derived features and machine learning. We train the cost model by generating and featurizing hundreds of thousands of random programs and schedules. We show that this approach operates effectively with or without autotuning. It produces schedules which are on average almost twice as fast as the existing Halide autoscheduler without autotuning, or more than twice as fast with, and is the first automatic scheduling algorithm to significantly outperform human experts on average.

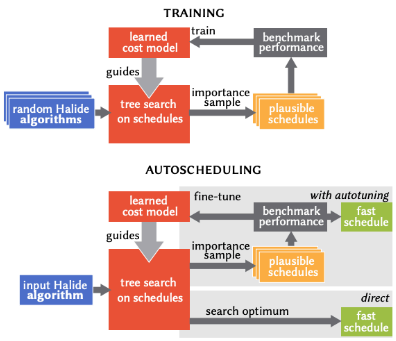


Fig. 1. We generate schedules for Halide programs using tree search over the space of schedules (Sec. 3) guided by a learned cost model and optional autotuning (Sec. 4). The cost model is trained by benchmarking thousands of randomly-generated Halide programs and schedules (Sec. 5). The resulting code significantly outperforms prior work and human experts (Sec. 6).

SIGGRAPH 2019

INTENTIONAL PROGRAMMING WITH HALIDE

Learning to Optimize Halide with Tree Search and Random Programs

ANDREW ADAMS, Facebook AI Research
KARIMA MA, UC Berkeley
LUKE ANDERSON, MIT CSAIL
RIYADH BAGHDADI, MIT CSAIL
TZU-MAO LI, MIT CSAIL
MICHAËL GHARBI, Adobe
BENOIT STEINER, Facebook AI Research
STEVEN JOHNSON, Google
KAYVON FATAHALIAN, Stanford University
FRÉDO DURAND, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley

SIGGRAPH 2019

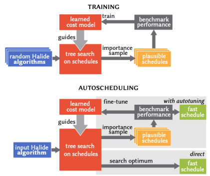
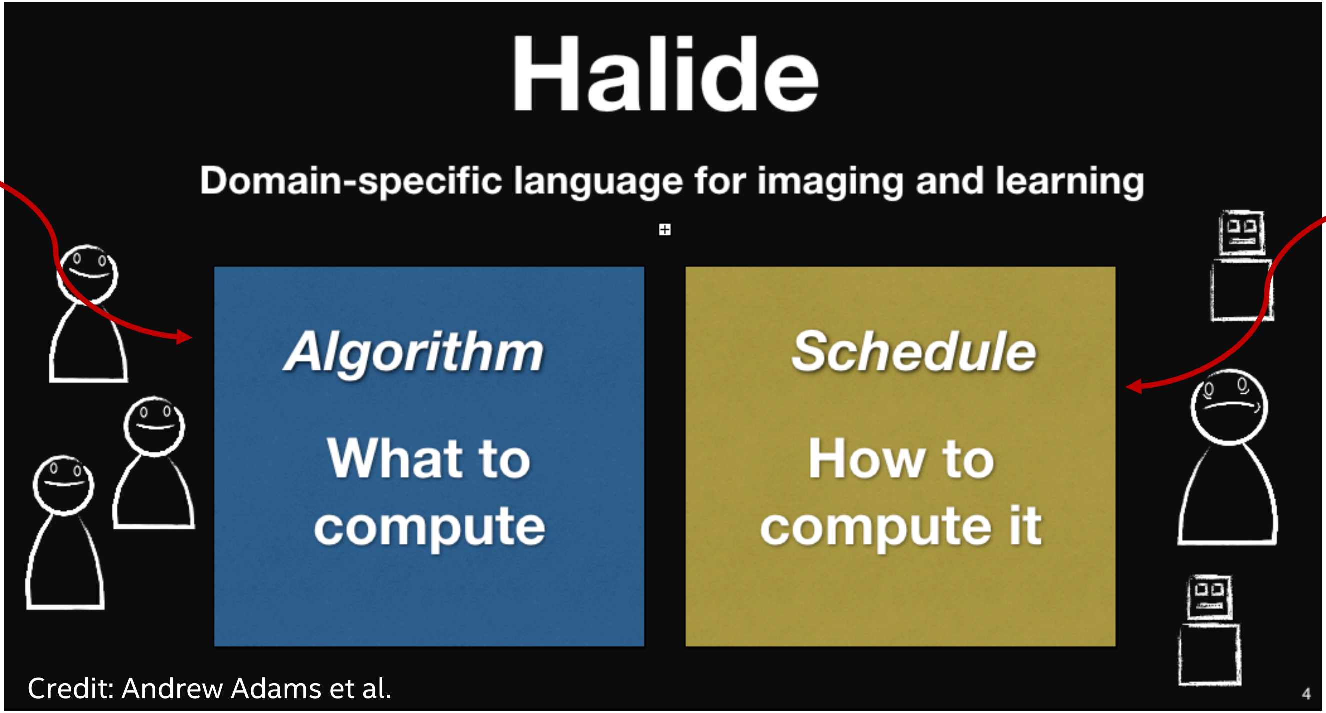


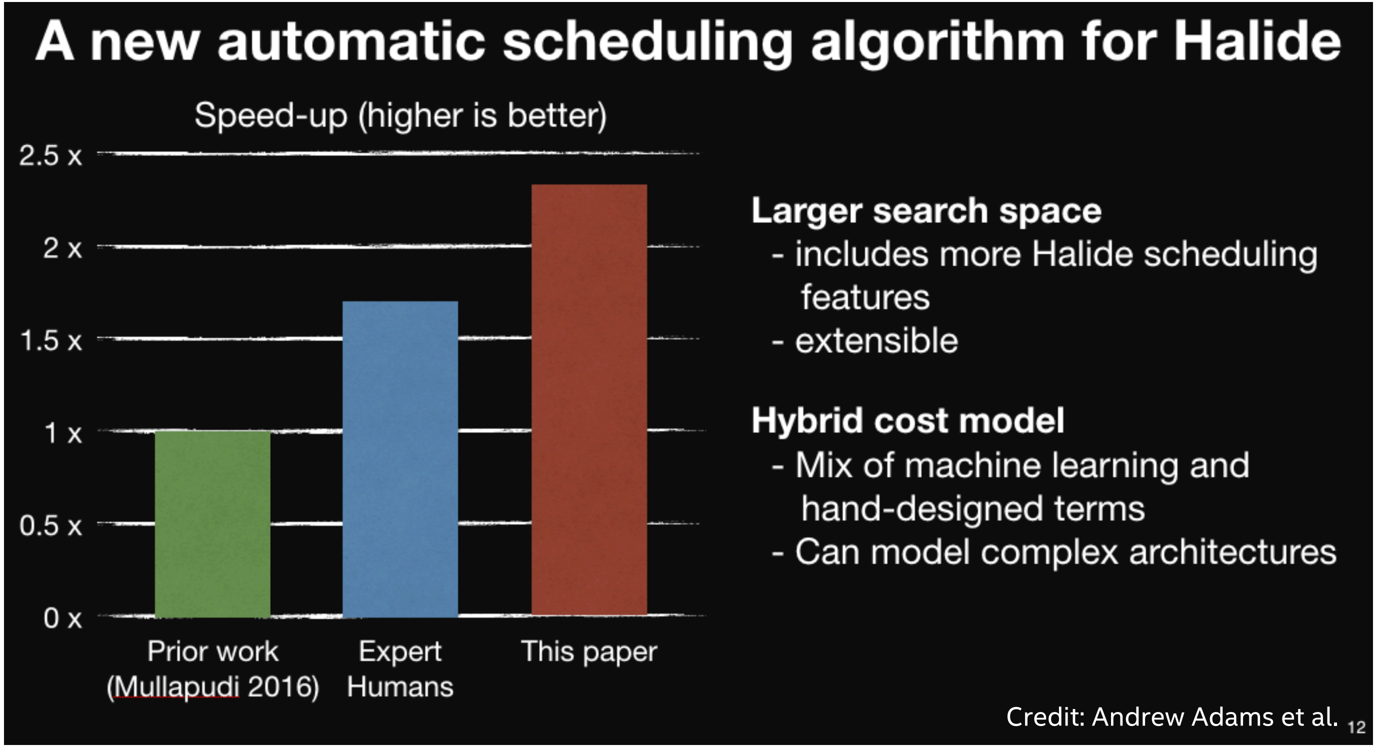
Fig. 1. We generate schedules for Halide programs using tree search over the space of schedules (Sec. 3) guided by a learned cost model and optional autotuning (Sec. 4). The cost model is trained by benchmarking thousands of randomly-generated Halide programs and schedules (Sec. 5). The resulting code significantly outperforms prior work and human experts (Sec. 6).

Intention

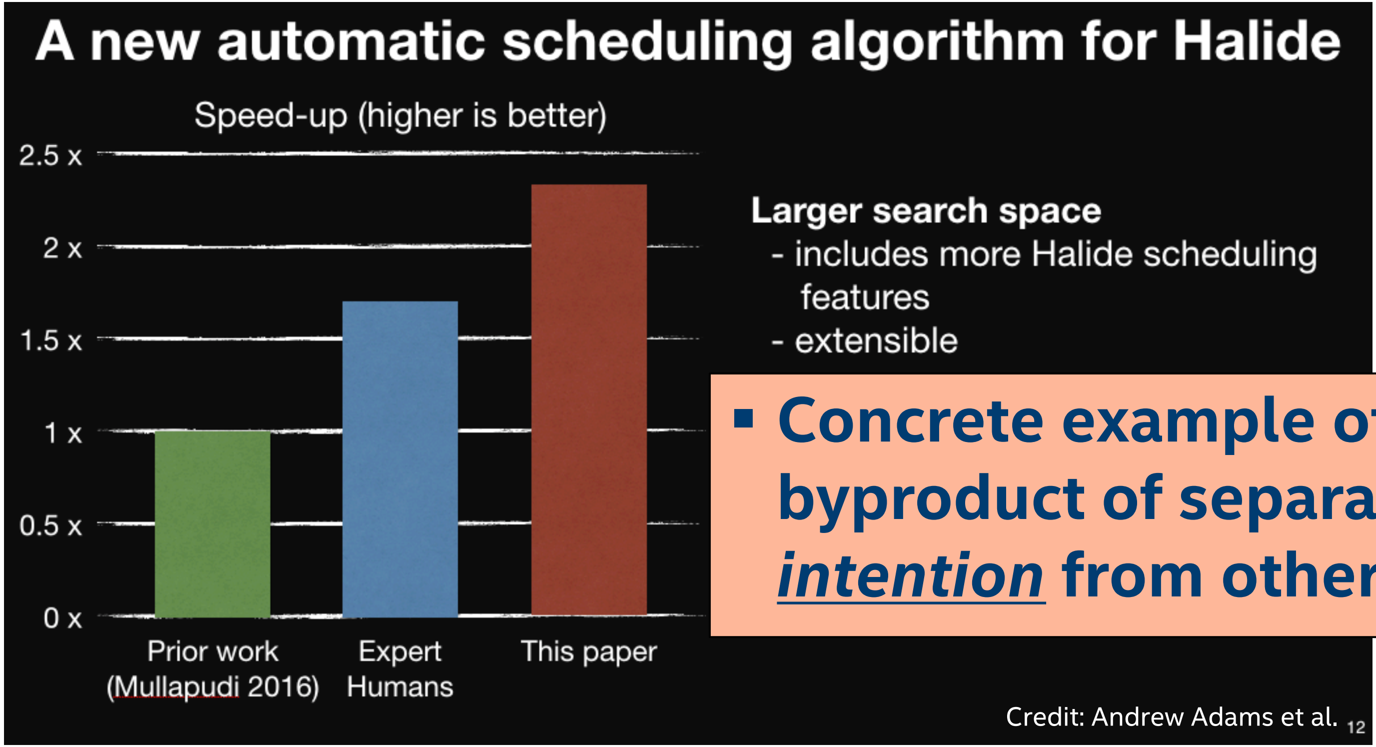


Adaptation

INTENTIONAL PROGRAMMING CAN LEAD TO SUPER-HUMAN PERFORMANCE



INTENTIONAL PROGRAMMING CAN LEAD TO SUPER-HUMAN PERFORMANCE



- Concrete example of byproduct of separating ***intention*** from other pillars

INTENTIONAL PROGRAMMING CAN LEAD TO SUPER-HUMAN PERFORMANCE

MY BURNING QUESTIONS:

CAN WE BUILD GENERAL-PURPOSE INTENTIONAL PROGRAMMING SYSTEMS?

PERHAPS WE PROVIDE INTENTION-BASED INTERFACES TO EXISTING WIDELY USED LANGUAGES (C++, PYTHON, JAVASCRIPT)?

(NON-EXHAUSTIVE) TOPICS

- MACHINE PROGRAMMING USES STOCHASTIC AND DETERMINISTIC METHODS
- EXTRACTION OF EVOLVING AND MULTI-DIMENSIONAL CODE SEMANTICS
- NOVEL STRUCTURAL REPRESENTATIONS OF CODE
- AUTOMATION FOR SOFTWARE AND HARDWARE HETEROGENEITY
- INTENTIONAL PROGRAMMING
- THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

Challenges:

- Computational workload via FM and ML may be large
- MP data is large, can be dense, and is mostly unlabeled
- Given this, what does the future MP hardware look like?

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

Challenges:

- Computational workload via FM and ML may be large
- MP data is large, can be dense, and is mostly unlabeled
- Given this, what does the future MP hardware look like?

I have no idea.

But I do have ideas about things we can think about.

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

Some open questions:

- What interfaces do we expect for expression of intention?
  - What ramifications are associated with those?

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

Some open questions:

- What interfaces do we expect for expression of intention?
  - What ramifications are associated with those?

- **What are the core techniques used for MP?**
  - **What are the data, communication, and compute implications?**

THE FUTURE OF DATA, COMMUNICATION, AND COMPUTATION FOR MP

Some open questions:

- What interfaces do we expect for expression of intention?
  - What ramifications are associated with those?

- What are the core techniques used for MP?
  - What are the data, communication, and compute implications?

- **We have a massive big and dense data problem in front of us**
  - **As of summer 2020, there were over 200M+ github repos**
  - **Code is multi-dimensional by nature**
  - **This data implies *new frontiers* of compute, communication, and data hardware**

THE ERA OF MACHINE PROGRAMMING IS NOW

We are on the verge of a *revolutionary* shift

Machine Programming is a Pioneering Research Initiative at Intel

THE ERA OF MACHINE PROGRAMMING IS NOW

We are on the verge of a *revolutionary* shift

Machine Programming is a Pioneering Research Initiative at Intel

Many institutions are heavily investing in MP

- Many large tech companies (Amazon, Google, IBM, Intel, Microsoft, etc.)
  - Both research and engineering
- Dozens of startups to solve a single MP problem
- Several leading academic institutions (like UWisc)

MP has the potential to change the rules for (almost) everything

LET'S BECOME THE 100%

LET'S BECOME THE 100%

We can democratize the creation of software with MP

- Imagine a global population, where everyone can express their creativeness
- Imagine a world where coders only spent time expressing our intentions, not fixing code
- What kind of scientific, artistic, innovative things might we discover?

A great way forward is to build a community (like Remzi et al. are)!

We need more of this; please help us spread the word

Please reach out to me if you are interested in collaborating!

THANK YOU!

Machine Programming Inside

QUESTIONS / COMMENTS: JUSTIN.GOTTSCHLICH@INTEL.COM