Predicting Dynamic Properties of Heap Allocations Using Neural Networks Trained on Static Code

Christian Navasca  
UCLA

Martin Maas  
Google

Petros Maniatis  
Google

Hyeontaek Lim  
Google

Harry Xu  
UCLA
Profiling Guided Optimizations are Impactful

- Dynamic dispatch removal
- Branch Reordering
- ...

UCLA
Profiling Guided Optimizations are Impactful

- **Dynamic dispatch removal**
- Branch Reordering
- ...

```java
A a = getObj();
a.foo(); // B.foo? C.foo?
```

```java
A a = getObj();
B.foo(a)
```
Profiling Guided Optimizations are Impactful

- Dynamic dispatch removal
- **Branch Reordering**
- ...

```java
if (rareCondition)
    ...
else if (commonCondition)
    ...
```

```java
if (commonCondition)
    ...
else if (rareCondition)
    ...
```
Profiling Guided Optimizations Need Profiling

- Dynamic dispatch removal – Receiver-class Profiling
- Branch Reordering – Branch Profiling

```java
A a = getObj();
a.foo(); //B: 100/100 C:0/100
```
Profiling Guided Optimizations Need Profiling

- Dynamic dispatch removal – Receiver-class Profiling
- **Branch Reordering** – Branch Profiling

```java
A a = getObj();
a.foo(); //B: 100/100 C:0/100

if (rareCondition) // hit rate: 15%
else if (commonCondition) // hit rate: 85%
```
Heap Allocation Properties Enable Optimizations

- Object Lifetimes – Pre-tenuring
- Object Hotness – Object Placement
- …
Heap Allocation Properties Enable Optimizations

- **Object Lifetimes – Pre-tenuring**
- **Object Hotness – Object Placement**
- ...
Heap Allocation Properties Enable Optimizations

- Object Lifetimes – Pre-tenuring
- Object Hotness – Object Placement
- ...

```java
// Hot Object
A a = new A();
```
Profiling a Data Center is Expensive

- 1% performance degradation is substantial
  - Object lifetime profiling adds 6+% overhead
- Profiling is complex to collect
  - Engineering of representative benchmarks
  - Deployment challenges
- No “killer” application to optimize for
Can we predict object properties instead?
Goal: Profile only Some Applications

- Collect profiling data for representative workloads
- Train a model on this partial dataset
- Predict the profiling data for un-profiled applications
Contributions

This work is an Intellectual Abstract and does not fully realize cross-application predictions

Instead, we:

● Introduce a framework for reasoning about the design space
● Apply this framework to a specific prediction problem
● Highlight challenges and future directions
Conceptual Framework

1) Data: the input for our predictions
2) Model: how we make predictions
3) Application: how the predictions are used

In this work we predict Object Lifetimes
Part 1: Data
We Collect Object Lifetimes

- Objects represented by their *allocation context*
- Each context is associated with lifetimes
- We collect these lifetimes using a modified OpenJDK

1. AtmelInterpreter.<init>(()V
2. LegacyInterpreter.<init>(()V
3. Simulation.createSimulator()Z

Lifetimes: [0, 0, 1620, 2484, ...]
We Collect Object Lifetimes

- Objects represented by their *allocation context*
- Each context is associated with lifetimes
- We collect these lifetimes using a modified OpenJDK

1. `AtmelInterpreter.<init>()V`
2. `LegacyInterpreter.<init>()V`
3. `Simulation.createSimulator()Z`

Lifetimes:

```
[0, 0, 1620, 2484, ...]
```
We Collect Object Lifetimes

- Objects represented by their allocation context
- Each context is associated with lifetimes
- We collect these lifetimes using a modified OpenJDK

1. AtmelInterpreter.<init>()V
2. LegacyInterpreter.<init>()V
3. Simulation.createSimulator()Z

Lifetimes:
[0, 0, 1620, 2484, ...]
Part 2: Model
State of the Art

LLAMA

- Lifetime-aware Memory Allocator
- Predicted Object lifetimes based on allocation context

1. AtmelInterpreter.<init>()V
2. LegacyInterpreter.<init>()V
3. Simulation.createSimulator()Z
State of the Art: LLAMA

1. `AtmelInterpreter.<init>()`\(\text{V}\)
2. `LegacyInterpreter.<init>()`\(\text{V}\)
3. `Simulation.createSimulator()`\(\text{Z}\)

![Diagram of LSTM Cells and Embeddings]
State of the Art: LLAMA

1. AtmelInterpreter.<init>()V
2. LegacyInterpreter.<init>()V
3. Simulation.createSimulator()Z
State of the Art: LLAMA

1. `AtmelInterpreter.<init>()V`
2. `LegacyInterpreter.<init>()V`
3. `Simulation.createSimulator()Z`

[ 3.35, -2.805, -1.54, 0.70, 2.03]
State of the Art: LLAMA

1. AtmelInterpreter.<init>()V
2. LegacyInterpreter.<init>()V
3. Simulation.createSimulator()Z
State of the Art: LLAMA

- Generalized to unseen stack traces in the same application
- Lifetime predictions were only 1% better than random across applications

1. AtmelInterpreter.<init>(LSimulator;)V
2. LegacyInterpreter.<init>(LSimulator;)V
3. LegacyInterpreter$Factory.newInterpreter()V
4. Simulator.<init>(LSimulation;)V
5. Simulation.createSimulator()Z

![Diagram of LSTM Cells and Embeddings]
State of the Art: LLAMA

- Generalized to unseen stack traces in the same application
- Lifetime predictions were only 1% better than random across applications
  - The representation is not generalizable across applications

1. `AtmelInterpreter.<init>(LSimulator;)V`
2. `LegacyInterpreter.<init>(LSimulator;)V`
3. `LegacyInterpreter$Factory.newInterpreter()V`
4. `Simulator.<init>(LSimulation;)V`
5. `Simulation.createSimulator()Z`
State of the Art: LLAMA

- Lifetime predictions were only 1% better than random across applications
  - The representation is not generalizable

Potential solution: Source Code
Why use a Code Model?

- There are correlations between source code and object behavior
Why use a Code Model?

- There are correlations between source code and object behavior

```java
for (int i = 0; i < 10; ++i) {
    // f never leaves the loop → short lived
    Foo f = new Foo();
    ...
}
```
Why use a Code Model?

- There are correlations between source code and object behavior

```java
Bar b = new Bar(); // returned → long lived
for (int i = 0; i < 10; ++i) {
    // f never leaves the loop → short lived
    Foo f = new Foo();
    ...
}

return b;
```
Why use a Code Model?

- There are correlations between source code and object behavior
- We could list them all and apply manual rules
Why use a Code Model?

- There are correlations between source code and object behavior
- We could list them all and apply manual rules
  - Brittle
  - Change over time

Machine learning gives us a chance to learn them automatically
Objects Represented by Allocation Contexts

AtmelInterpreter.<init>(..) V
: AtmelInterpreter.java@326

LegacyInterpreter.<init>(..) V :
LegacyInterpreter.java@74 ...

Harness.main([Ljava.lang.String;])V:@0

320 protected AtmelInterpreter(..) {
321 super(simulator);
322 // ...
323 // ...
324 Compiler.compileClass(...);
325 326 state = new StateImpl();
327 328 globalProbe = new ...
329 330 SREG = pr.getIOReg("SREG");
331 ...
378 }

Lifetimes: [ 0, 0, 1620, 2484, ...

33
Objects Represented by Allocation Contexts

AtmelInterpreter.<init>(..)V  
: AtmelInterpreter.java@326

LegacyInterpreter.<init>(..)V  
: LegacyInterpreter.java@74

Harness.main([LString;)V@0

Lifetimes: [0, 0, 1620, 2484, ...]
Objects Represented by Allocation Contexts

```
320 protected AtmelInterpreter(..) {
321    super(simulator);
322    // ...
323    // ...
324    Compiler.compileClass(...);
325
326    state = new StateImpl();
327
328    globalProbe = new ...
329
330    SREG = pr.getIOReg("SREG");
331
...}
```

Lifetimes: [0, 0, 1620, 2484, ...]
Objects Represented by Allocation Contexts

AtmelInterpreter.<init>(..)V  :  AtmelInterpreter.java@326

LegacyInterpreter.<init>(..)V  :  LegacyInterpreter.java@74

Harness.main([LString;)V:@0

320 protected AtmelInterpreter(..) {  
321 super(simulator);  
322 // ...  
323 // ...  
324 Compiler.compileClass(...);  
325  
326 state = new StateImpl();  
327  
328 globalProbe = new ...  
329  
330 SREG = pr.getIOReg("SREG");  
331  
332  ...  
333  
334  }  

Lifetimes: [0, 0, 1620, 2484, ...]
Objects Represented by Allocation Contexts

AtmelInterpreter.<init>(..) V : AtmelInterpreter.java@326

LegacyInterpreter.<init>(..) V : LegacyInterpreter.java@74

Harness.main([Ljava.lang.String;) V: @0

protected AtmelInterpreter(..) {
    super(simulator);
    // ...
    // ...
    Compiler.compileClass(....);
    state = new StateImpl();
    globalProbe = new ...
    SREG = pr.getIOReg("SREG");
    ...
}

Lifetimes: [0, 0, 1620, 2484, ...

...]}
Problem: Way Too Much Code

- Pre-trained Transformer-based models are often trained to 512 tokens
- Even a single function can contain thousands of tokens
- 93% of traces we collect have 32+ frames
Problem: Way Too Much Code

- Select a small window of tokens per-frame
  - Miss far away tokens

```java
320 protected AtmelInterpreter(..) {
321   super(simulator);
322   // ...
323   // ...
324   Compiler.compileClass(...);
325
326   state = new StateImpl();
327
328   globalProbe = new ...
329
330   SREG = pr.getIOReg("SREG");
331   ...
378 }
```
ASTs Compactly Represent Structure of Code

```java
public class Foo {
    boolean otherMethod() {
        ...
    }
    public void bar() {
        if (otherMethod()) {
            ...
        }
        for (...) {
            Object foo = new Object();
        }
    }
}
```
Each Representation has a Trade-off

- **Function Names**
  - Captures behavior of a single trace
  - Not generalizable

- **Code**
  - Defines program behavior
  -Verbose

- **AST**
  - Captures structure of the code
  - Loses fine-grained information
Our Solution: A Multi-Modal Model

- Average
  - Frame Embedding

- Signature Transformer
  - Signature Embedding
    - Signature Tokens
  - Stack Frame

- Code Transformer
  - Code Embedding
    - Code Tokens

- AST Transformer
  - AST Embedding
    - AST Tokens

- Lifetime Prediction
  - LSTM
  - Frame Embedding
  - Input Stack Trace
Our Solution: A Multi-Modal Model

- **Signature Transformer**
- **Code Transformer**
- **AST Transformer**

  - **Signature Embedding**
  - **Code Embedding**
  - **AST Embedding**

  - **Signature Tokens**
  - **Code Tokens**
  - **AST Tokens**

  - **Stack Frame**

  - **Frame Embedding**

  - **Average**

  - **Frame Embedding**

  - **Lifetime Prediction**

  - **LSTM**

  - **Frame Embedding**

  - **Input Stack Trace**
Our Solution: A Multi-Modal Model

![Diagram of the multi-modal model]

- **Average**
  - **Frame Embedding**
    - **Signature Transformer**
      - **Signature Embedding**
        - **Signature Tokens**
    - **Code Transformer**
      - **Code Embedding**
        - **Code Tokens**
    - **AST Transformer**
      - **AST Embedding**
        - **AST Tokens**

- **Input Stack Trace**
  - **Frame Embedding**
  - **LSTM**
    - **Lifetime Prediction**

[Image of the diagram]
Our Solution: A Multi-Modal Model

- **Signature Transformer**
  - Signature Embedding
  - Signature Tokens

- **Code Transformer**
  - Code Embedding
  - Code Tokens

- **AST Transformer**
  - AST Embedding
  - AST Tokens

- **Stack Frame**
  - Input Stack Trace

- **Frame Embedding**
  - Frame Prediction
  - LSTM

- **Average**
  - Average Frame Embedding
Our Solution: A Multi-Modal Model

Signature Transformer
Signature Embedding
Signature Tokens

Code Transformer
Code Embedding
Code Tokens

AST Transformer
AST Embedding
AST Tokens

Stack Frame

Average
Frame Embedding

LSTM

Input Stack Trace

Lifetime Prediction

Transformer
AST Transformer
Code Transformer
Signature Transformer

Signature Tokens
Code Tokens
AST Tokens

Signature Embedding
Code Embedding
AST Embedding

Frame Embedding

Average
Frame Embedding
Our Solution: A Multi-Modal Model

- Signature Transformer
- Code Transformer
- AST Transformer

- Signature Embedding
- Code Embedding
- AST Embedding

- Signature Tokens
- Code Tokens
- AST Tokens

- Stack Frame

- Frame Embedding

- Average

- Lifetime Prediction

- LSTM

- Frame Embedding

- Input Stack Trace
Our Solution: A Multi-Modal Model

- **Signature Transformer**
- **Code Transformer**
- **AST Transformer**

- **Signature Embedding**
- **Code Embedding**
- **AST Embedding**

- **Signature Tokens**
- **Code Tokens**
- **AST Tokens**

- **Stack Frame**

- **Average**

- **Frame Embedding**

- **Lifetime Prediction**
  - **LSTM**
  - **Frame Embedding**
  - **Input Stack Trace**
Our Model Improved Accuracy

Improved Mean per-class Accuracy by 8 percentage points on un-profiled applications

Mean per-class Accuracy (%)

- Random: 50%
- LLAMA: 51%
- LLAMA-subword: 53%
- Code: 54%
- AST: 55%
- Transformer-Signature: 57%
- Multi-modal: 59%
Part 3: Application
Predictions Can Be Used In Multiple Ways

- **Offline**
  - Annotation predictions, Offline Optimizations
  - Source Code only
- **During JIT Compilation**
  - Source Code + Online Profiling
- **Online**
  - Run the model before a decision (e.g., at object allocation)
  - System state as input

Moving the predictions offline removes overhead, but reduces possible features
Goal: predict profiling information rather than collecting it
We present a conceptual framework for reasoning about this problem
Our multi-modal model out-performed single-modality models
  But not accurate enough to be practical
Such predictions can be used for numerous optimizations

We hope that this Intellectual Abstract helps to open a new research direction for the ISMM community