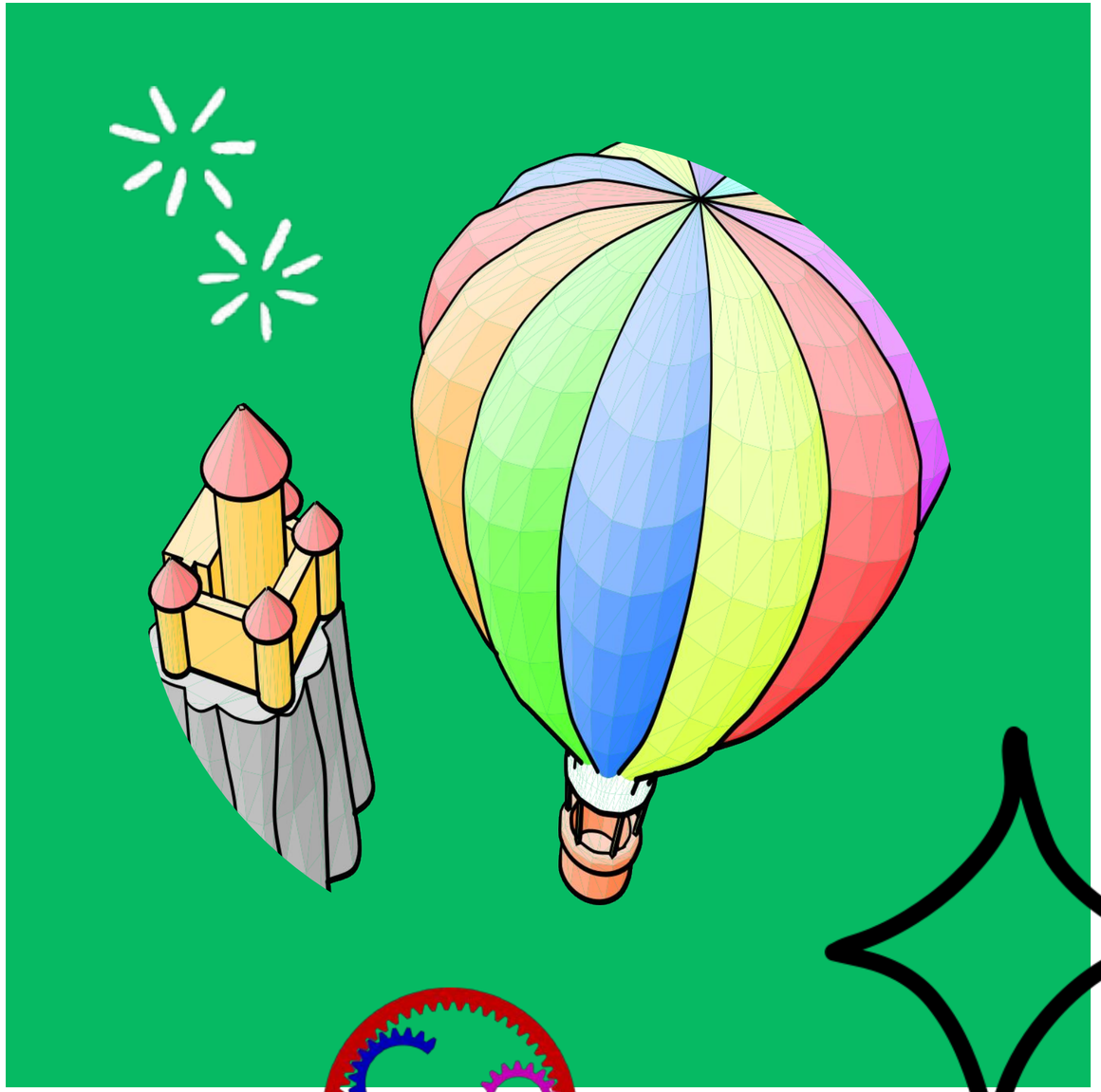
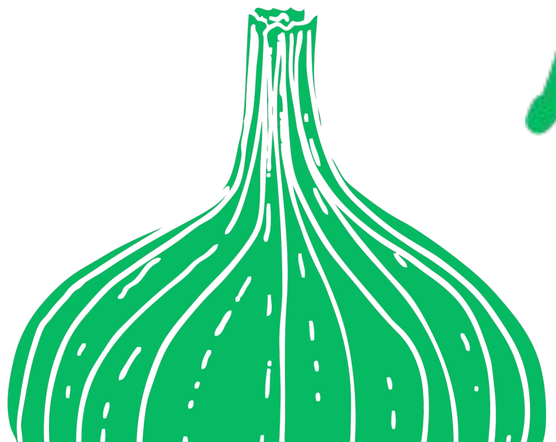
Compile-time PIC Generation using LiveTyping information



Main Goal

Generate PICs upfront based on available LiveTyping information

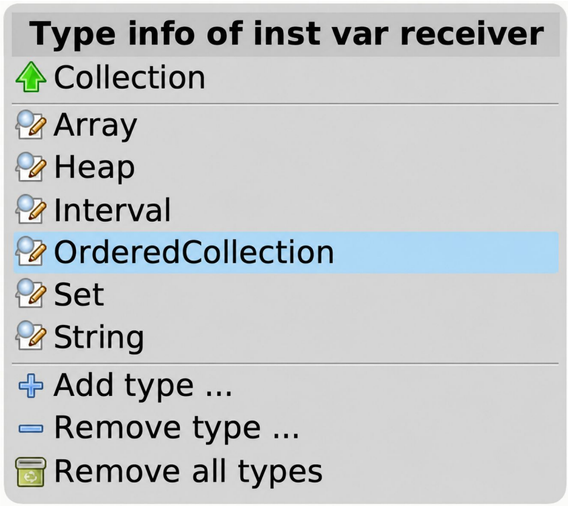
Minimal Background



LiveTyping

Before

After

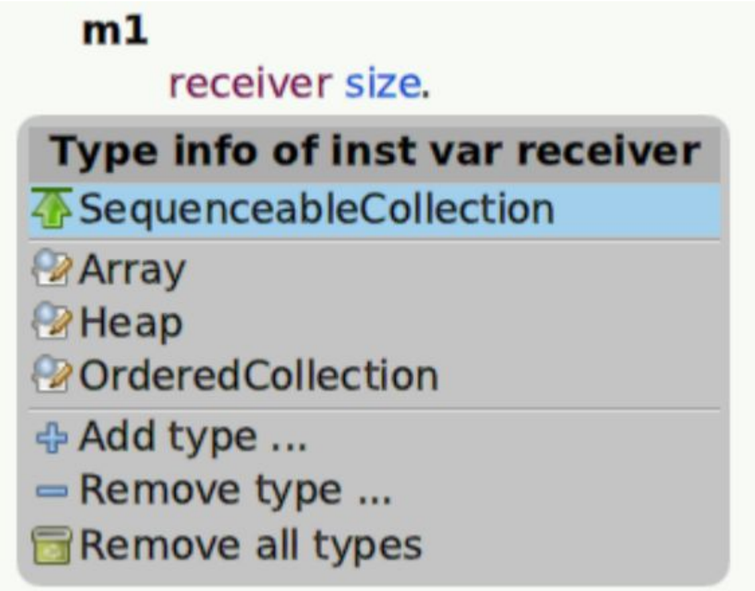


Code that assigns values to the receiver instance variable is executed

```
receivers := { OrderedCollection new. Array new: 100.
    ''. Heap new. (1 to: 100). Set new }.

(1 to: 6) do: [ :i |
    receiver := receivers at: i
].
```

PIC: Polymorphic Inline Cache




```
(receiver class = OrderedCollection) ifTrue: [
    receiver executeMethod: OrderedCollection>>#size
] ifFalse: [
    (receiver class = Heap) ifTrue: [
        receiver executeMethod: Heap>>#size
    ] ifFalse: [
        (receiver class = Array) ifTrue: [
            receiver executeMethod: ArrayedCollection>>#size
        ] ifFalse: [
            receiver methodLookup: #size
        ]
    ]
]
```

Traditional Approach vs. Our Proposal

Traditional Approach

PIC ➔ Type Information

Traditional Approach vs. Our Proposal

Our Proposal

LiveTyping Type Information

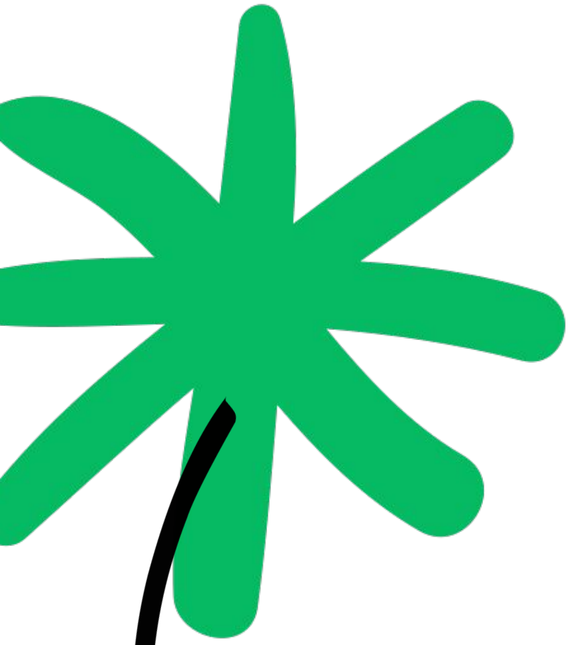
➔ PIC

Why focus on Mono/Bi/Poly PICs?

96.64% of observed selectors can be handled by ICs and closed PICs

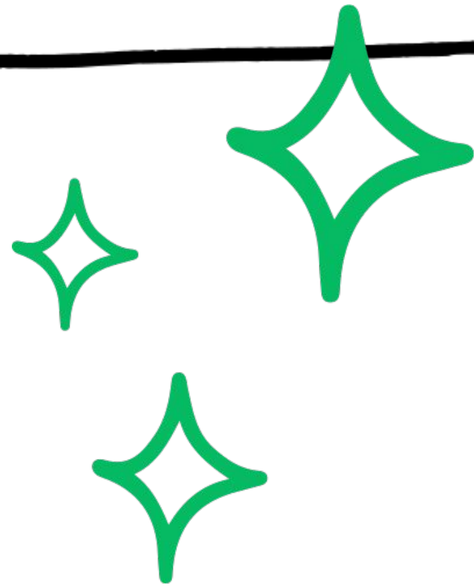
Monomorphics	77.5%
Bimorphics	8.49 %
Polymorphic (3–6)	10.65 %
Megamorphic (> 6)	3.36 %

Distribution of selectors by number of receiver classes

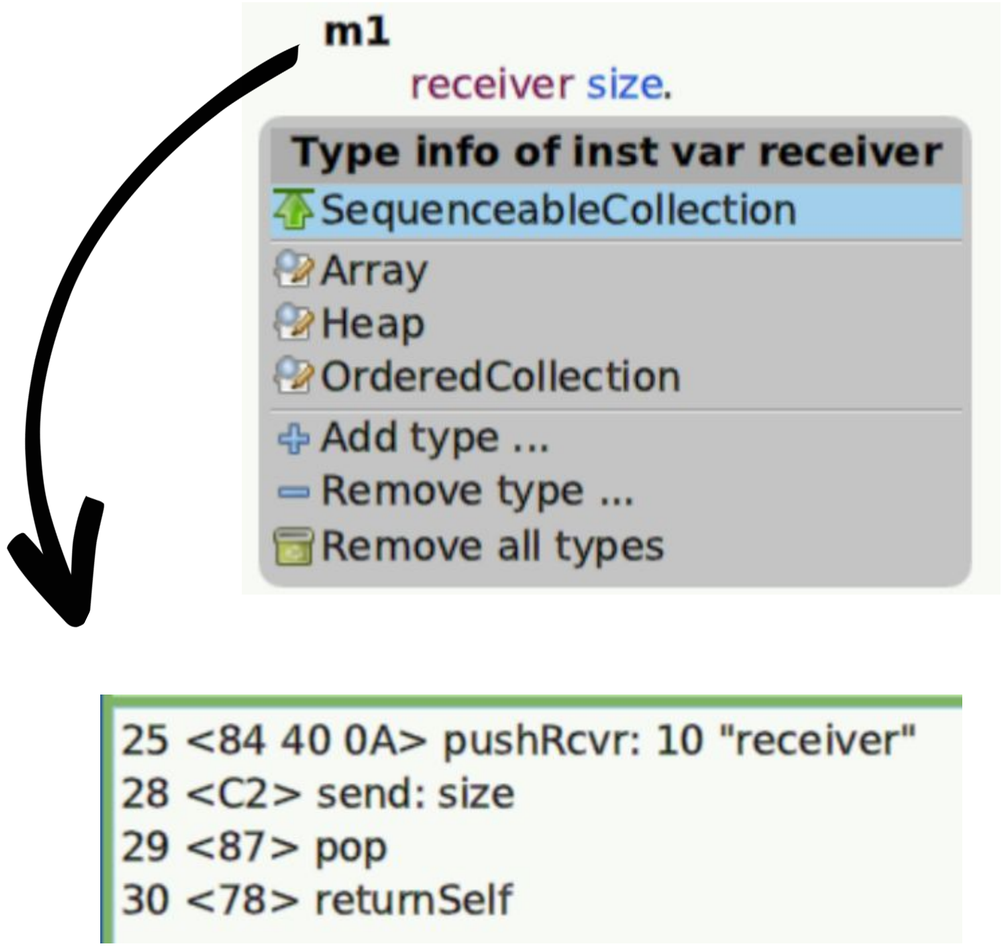


What we’ve done

@OSVM + CUIS



Define in the VM side a primitive which generates the PIC upfront




```
25 <84 40 0A> pushRcvr: 10 "receiver"
28 <C2> send: size
29 <87> pop
30 <78> returnSelf
```

```
ProactivePICBuilder
    handleCompiledMethods: {
        OrderedCollection>>#size .
        ArrayedCollection>>#size .
        Heap>>#size
    }
    forClasses: { OrderedCollection . Array . Heap }
    withSelector: #size
    patchMethod: PerformanceTest class>>#m1
    withSelector: #m1
    atSendReturnPCs: #(28)
```

RUNNING EXAMPLE

Example send site

Smalltalk source

```
m1
    receiver size
```

Bytecodes (method m1)

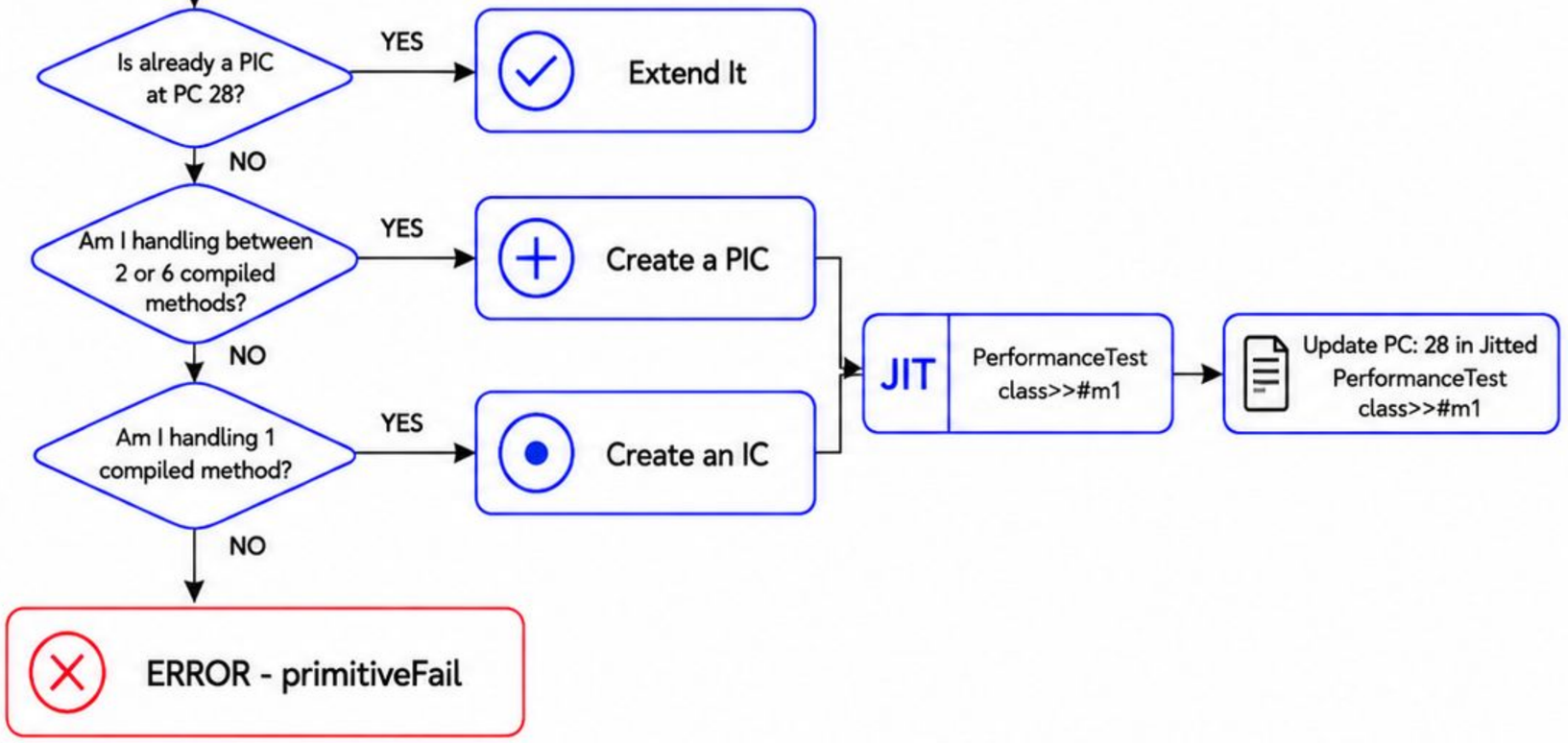
```
25  <84 40 0A>  pushRcvr: 10 "receiver"
28  <C2>  send: size
29  <87>  pop
30  <78>  returnSelf
```

This send site (PC 28) performs a send with selector #size.

VM level

ProactivePICBuilder

```
handleCompiledMethods: {
  OrderedCollection>>#size .
  ArrayedCollection>>#size .
  Heap>>#size
}
forClasses: { OrderedCollection . Array . Heap }
withSelector: #size
patchMethod: PerformanceTest class>>#m1
withSelector: #m1
atSendReturnPCs: #(28)
```



Add in the image side an easy way to use the VM primitive

```
ProactivePICBuilder
    handleCompiledMethods: {
        OrderedCollection>>#size .
        ArrayedCollection>>#size .
        Heap>>#size
    }
    forClasses: { OrderedCollection . Array . Heap }
    withSelector: #size
    patchMethod: PerformanceTest class>>#m1
    withSelector: #m1
    atSendReturnPCs: #(28)
```

```
ProactivePICBuilder generatePICFor: PerformanceTest class>>#m1
```

Image-level analysis

RUNNING EXAMPLE

Example send site

Smalltalk source

```
m1
    receiver size
```

Bytecodes (method m1)

```
25  <84 40 0A>  pushRcvr: 10 "receiver"
28  <C2>  send: size
29  <87>  pop
30  <78>  returnSelf
```

This send site (PC 28) performs a send with selector **#size.**

Image level

1. ProactivePICBuilder
2. Instruction stream simulator
3. Is the instruction stream at the end? — YES → Done
4. Get next instruction
5. Is the instruction a message send? — NO → 3
6. Is the receiver an instance variable? — NO → 3
7. Is LiveTyping information available? — NO → 3
8. Send-site bytecode PC
9. Observed receiver classes
10. Selector and target methods
11. VM — Primitive invocation

Extracted information

selector
The message selector of the send

bytecode PC
The bytecode index where the send occurs

receiver information
Which variable or expression is used as the receiver (e.g., instance variable)

Current implementation: only instance-variable receivers

LiveTyping provides observed receiver classes

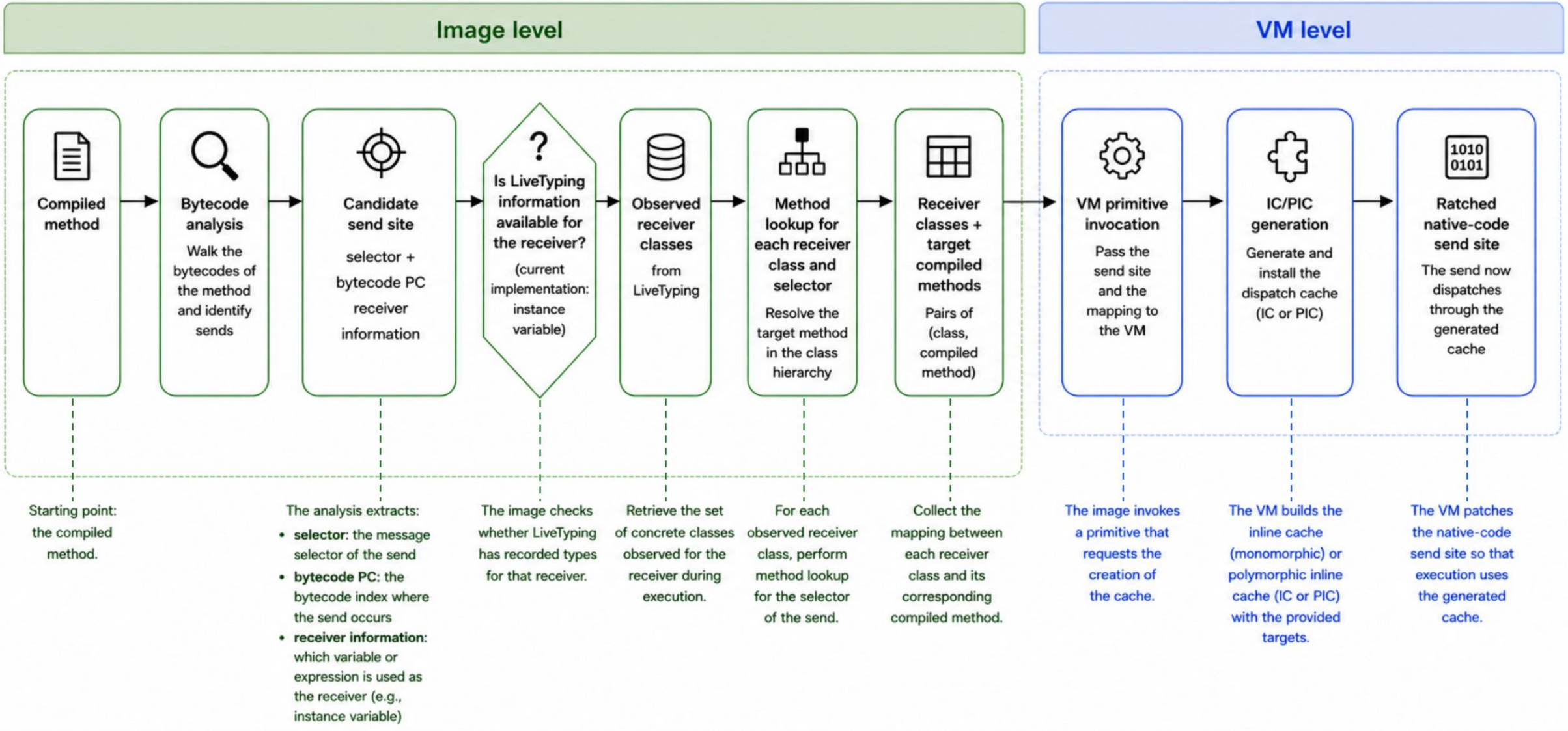
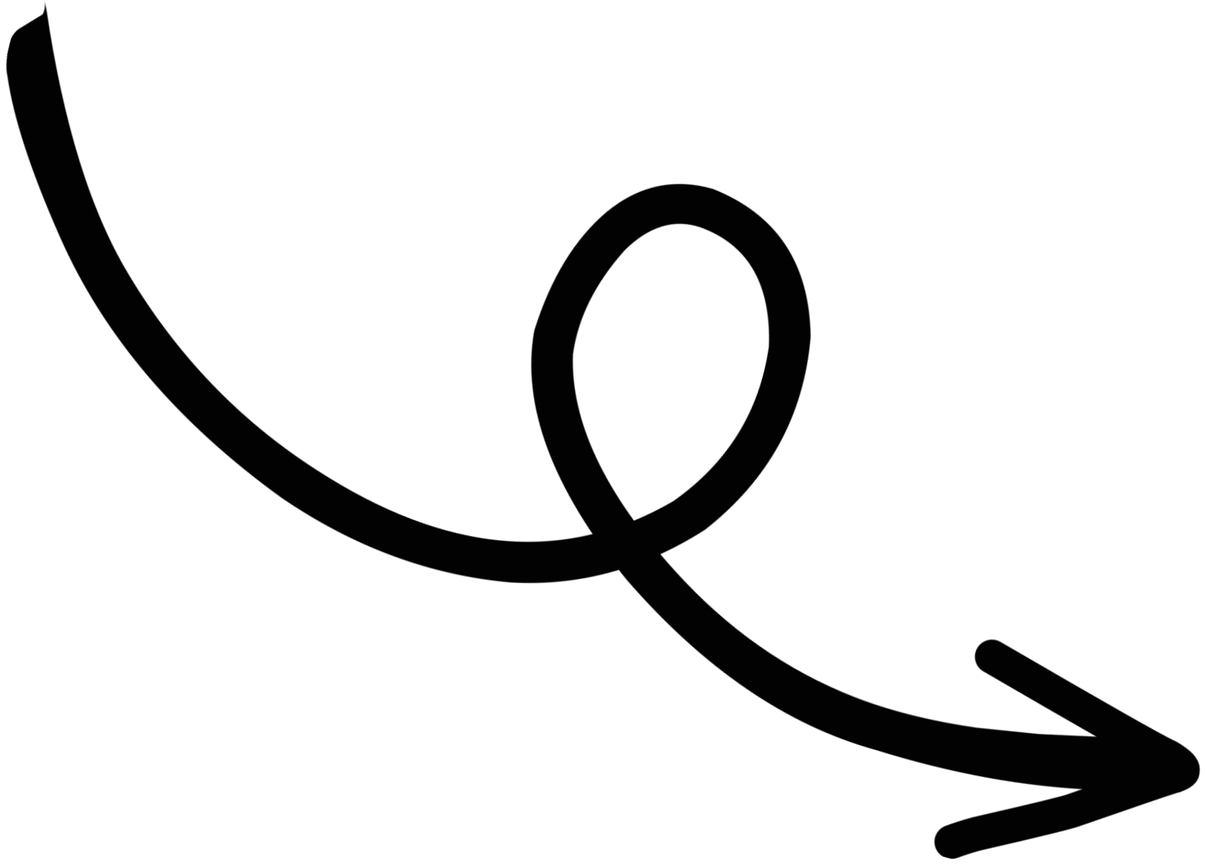


Image level
VM level
Compiled method
Bytecode analysis
Walk the bytecodes of the method and identify sends
Candidate send site
selector + bytecode PC receiver information
Is LiveTyping information available for the receiver?
(current implementation: instance variable)
Observed receiver classes
from LiveTyping
Method lookup for each receiver class and selector
Resolve the target method in the class hierarchy
Receiver classes + target compiled methods
Pairs of (class, compiled method)
VM primitive invocation
Pass the send site and the mapping to the VM
IC/PIC generation
Generate and install the dispatch cache (IC or PIC)
1010 0101
Ratched native-code send site
The send now dispatches through the generated cache
Starting point: the compiled method.
The analysis extracts:
• selector: the message selector of the send
• bytecode PC: the bytecode index where the send occurs
• receiver information: which variable or expression is used as the receiver (e.g., instance variable)
The image checks whether LiveTyping has recorded types for that receiver.
Retrieve the set of concrete classes observed for the receiver during execution.
For each observed receiver class, perform method lookup for the selector of the send.
Collect the mapping between each receiver class and its corresponding compiled method.
The image invokes a primitive that requests the creation of the cache.
The VM builds the inline cache (monomorphic) or polymorphic inline cache (IC or PIC) with the provided targets.
The VM patches the native-code send site so that execution uses the generated cache.



Experimental Evaluation

Research Questions

RQ1. Does proactive PIC generation reduce execution time?

RQ2. How does the benefit change as the number of send sites increases?

RQ3. How does the benefit change as the number of benchmark iterations increases?

RQ4. What is the cost at the VM level?

Schedule @IWST

Abstracts of talks and hands-on are available on **the asbtracts page**

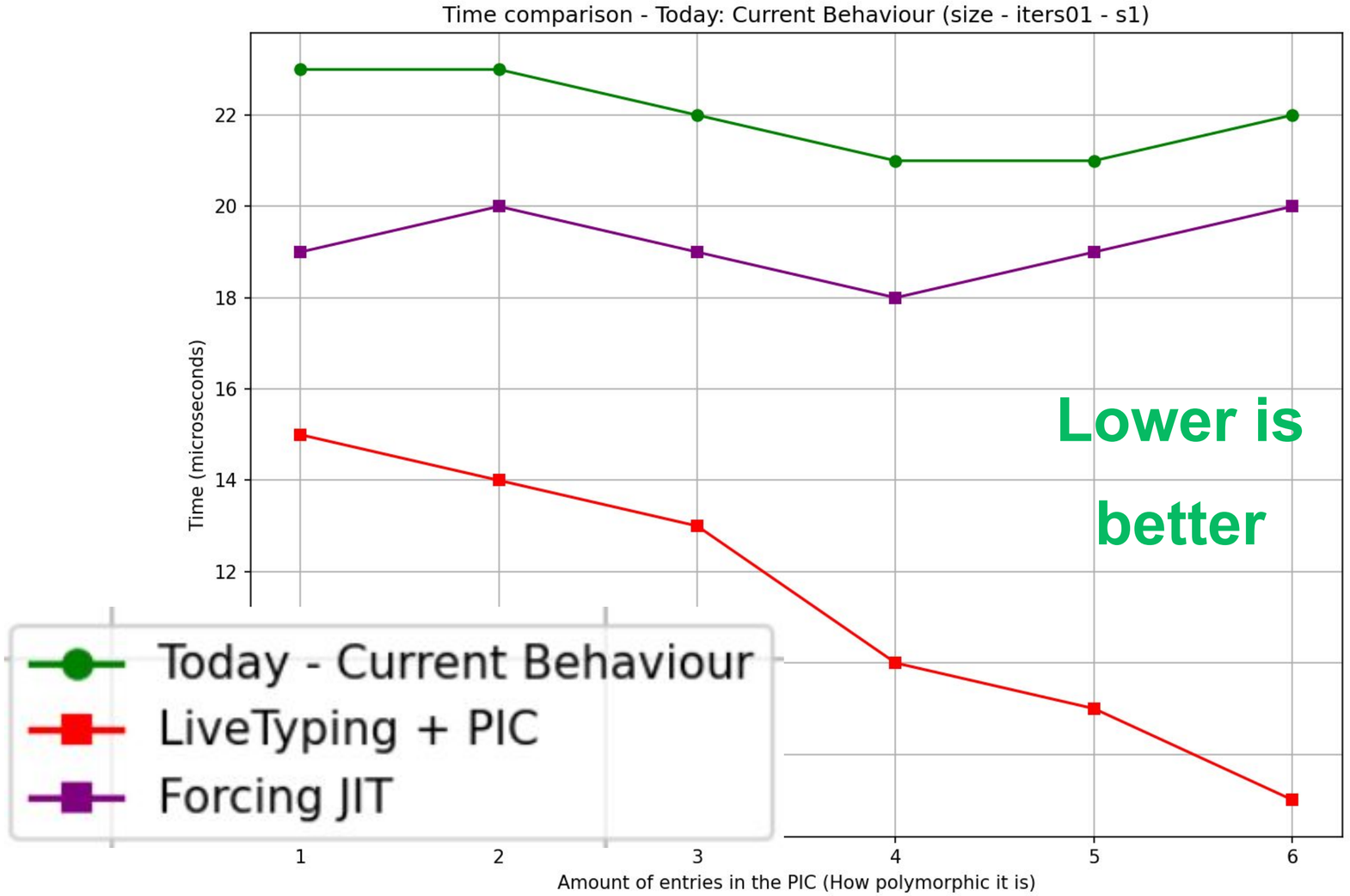
DAY-01
TUES. JULY 7TH

DAY-02
WED. JULY 8TH

DAY-03
THUR. JULY 9TH

TIME	SUBJECT	SPEAKER	VENUE
2:00 pm - 2:30 pm	Compile-time PIC Generation using LiveTyping Information	Nicolás Matías Sarfati and Hernán Wilkinson	Room B

RQ1. Does proactive PIC generation reduce execution time?



Family: **size**

Iterations: **1**

Send sites: **1**

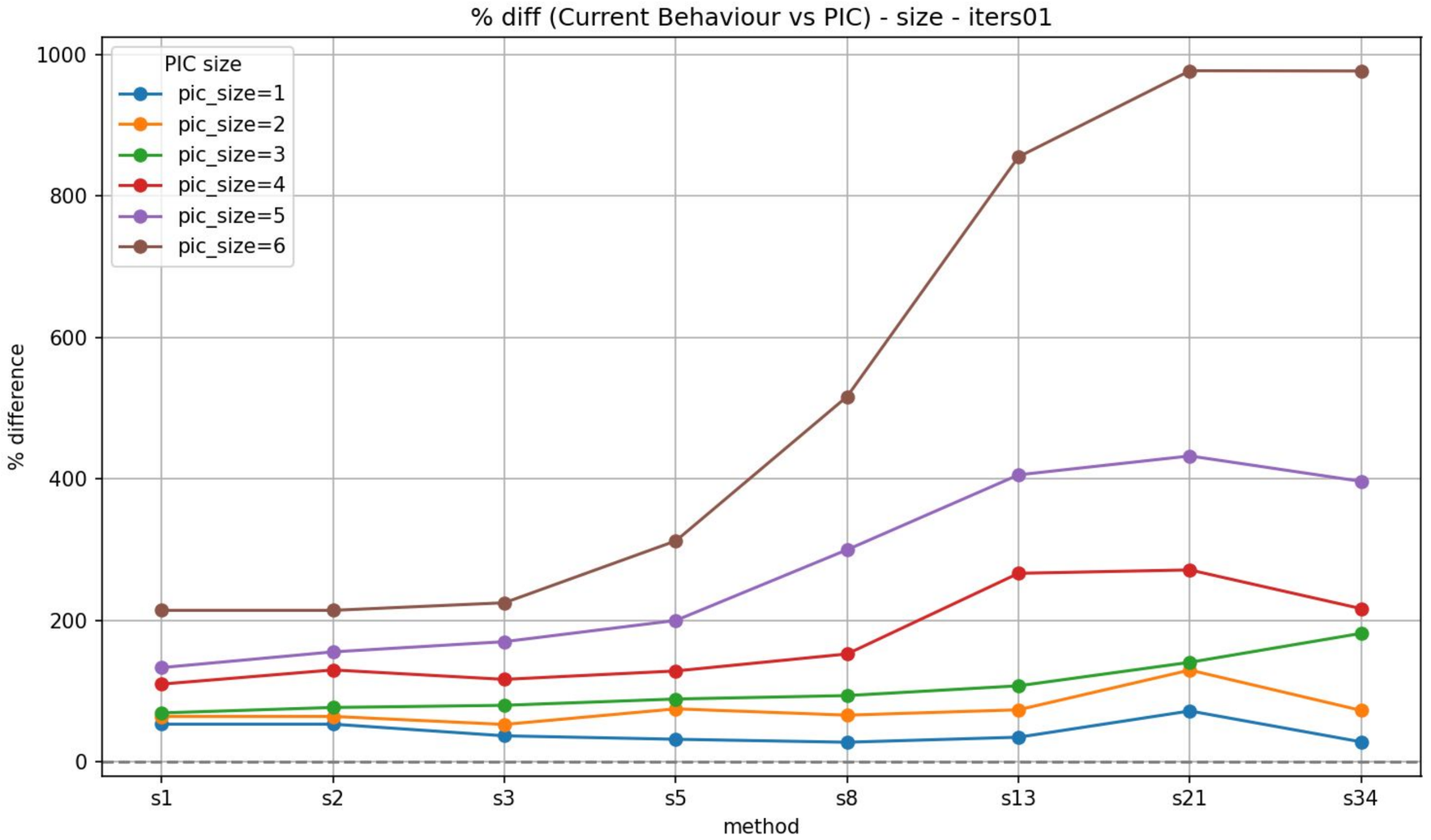
PIC vs CB

# PIC Entries	% Faster
1	53 %
2	64 %
3	69 %
4	110 %
5	134 %
6	214 %

RQ1. Does proactive PIC generation reduce execution time?

Proactive PIC generation reduces execution time ✔️

RQ2. How does the benefit change as the number of send sites increases?



Family: **size**

Iterations: **1**

Send sites: **1 … 34**

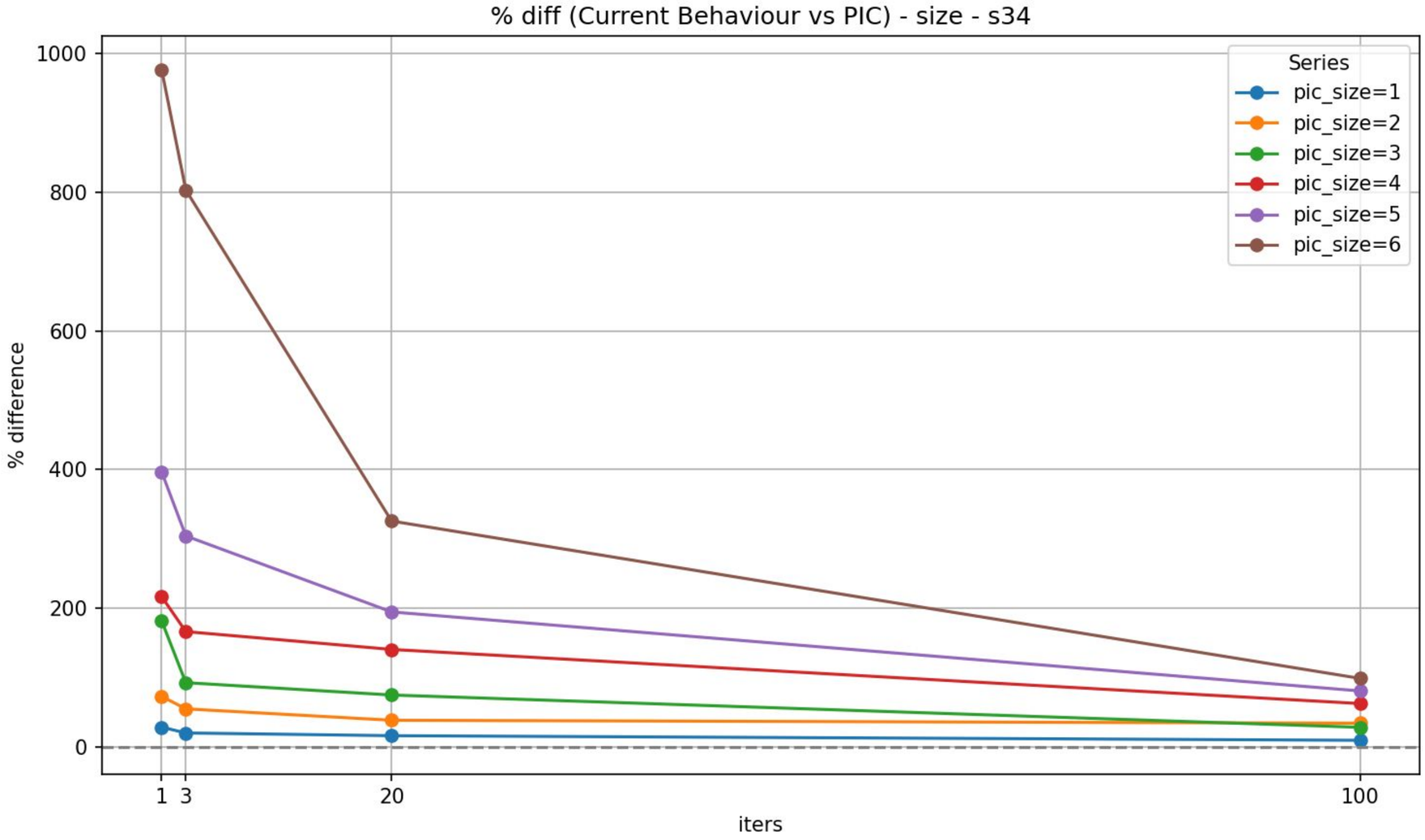
Higher is better

RQ2. How does the benefit change as the number of send sites increases?

The benefit increases with the number of dynamic send sites

More send sites ➔ more optimization opportunities

RQ3. How does the benefit change as the number of benchmark iterations increases?



Family: **size**

Iterations: **1 … 100**

Send sites: **34**

Higher is better

RQ3. How does the benefit change as the number of benchmark iterations increases?

The relative benefit decreases as the number of iterations increases.

Most of the benefit occurs during warm-up

RQ4. What is the cost at the VM level?

Configuration	Δ #JIT	Δ JIT bytes	Δ #closed PICs	Δ closed PIC bytes
1 iteration / 1 send site	+480 %	+513 %	+650 %	+652 %
1 iteration / 34 send sites	+480 %	+446 %	+37 %	+37 %
100 iterations / 1 send site	+480 %	+513 %	+650 %	+652 %
100 iterations / 34 send sites	+480 %	+446 %	+37 %	+37 %

VM Impact: **PIC vs CB**

Family: **size**

RQ4. What is the cost at the VM level?

There is a trade-off between lower initial execution time and increased VM resource usage

Conclusions

LiveTyping can drive VM-level optimizations

- We use **type information** already available in the image
- We **generate** ICs/PICs before the VM builds them **reactively**
- We reverse the traditional information flow:

  Type information ➔ PICs

The benefit is concentrated during warm-up

- **Proactive PIC** generation **reduces** the initial cost of dynamic dispatch
- The benefit is **greater** for **short-running** executions
- The benefit **increases** with the number of **send sites**
- In **longer executions**, the current VM behavior **amortizes** the cost of reactive PIC construction

Trade-off

- **Lower** initial execution time
- **More** VM structures are generated
- **More** JIT-compiled methods and closed PICs
- **Higher** code-space usage

Future work

👉 **More precise type information:** use Dragon TypeChecker, presented by Julian Ostrovsky at Smalltalks 2024, to generate PICs using context-specific types

👉 **Support** open PICs for **megamorphic** message sends

👉 **Refine** the interface exposed by the VM primitive

👉 **Extend** the optimization to additional receiver sources, **not only instance variables**

👉 **Develop heuristics** to identify send sites that would benefit from proactive PIC generation

👉 **Generate images** with **all methods precompiled** using JIT and PICs, which could be useful for web-server applications

👉 **Evaluate** the proposed approach **on real-world applications** or more **complex benchmarks**

👉 **Explore lazy PIC** generation driven by LiveTyping information

👉 **Explore sharing** equivalent PICs across send sites

Acknowledgments

FAST: Smalltalk Argentine Foundation

ESUG

Thank you! Questions?