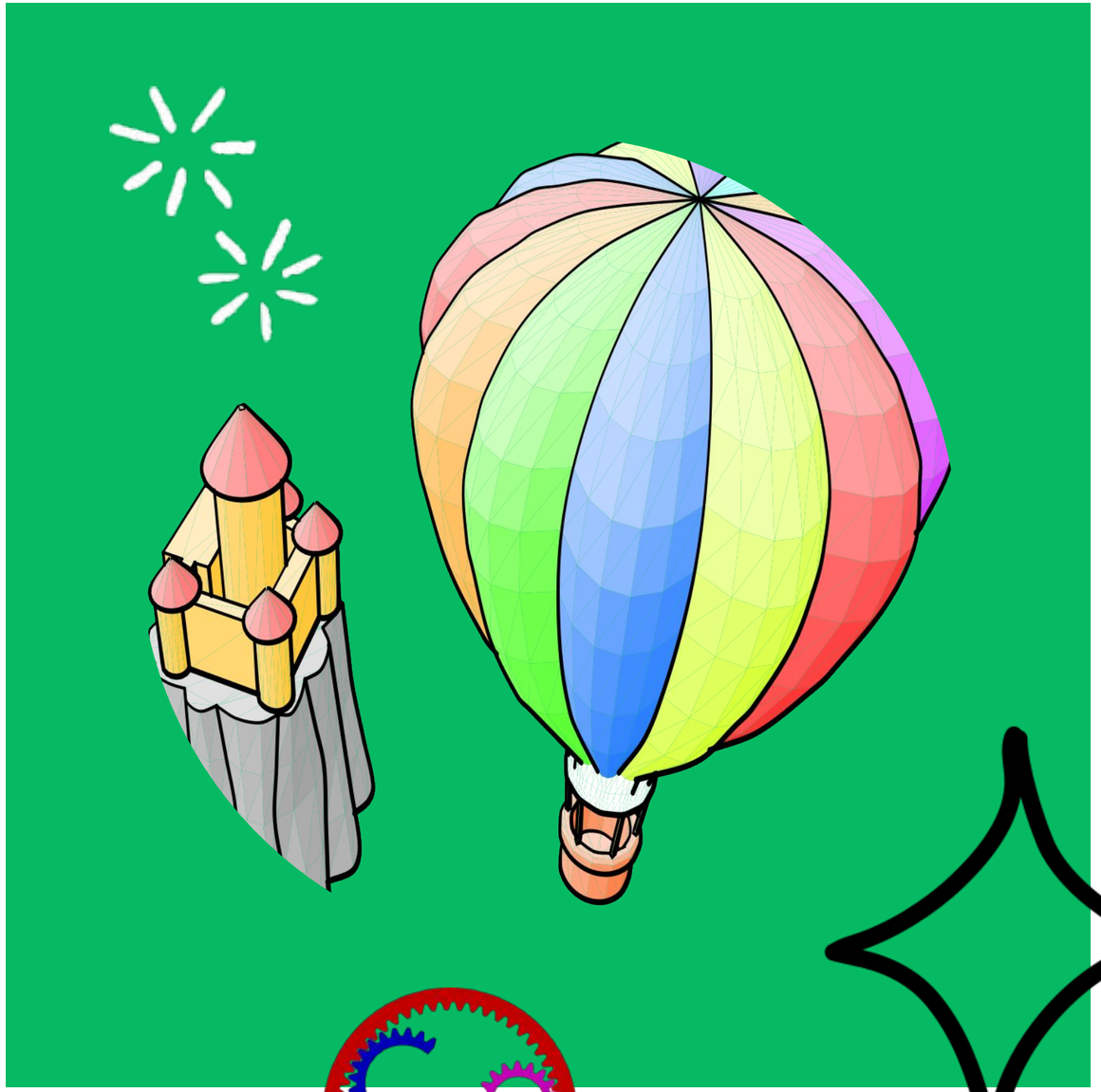
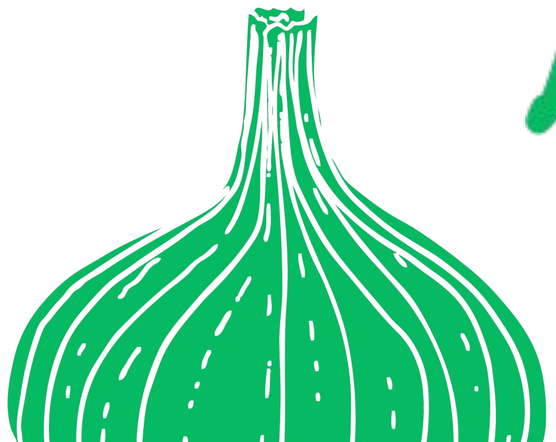
Compile-time PIC Generation using LiveTyping information



Main Goal

Generate PICs upfront based on available LiveTyping information

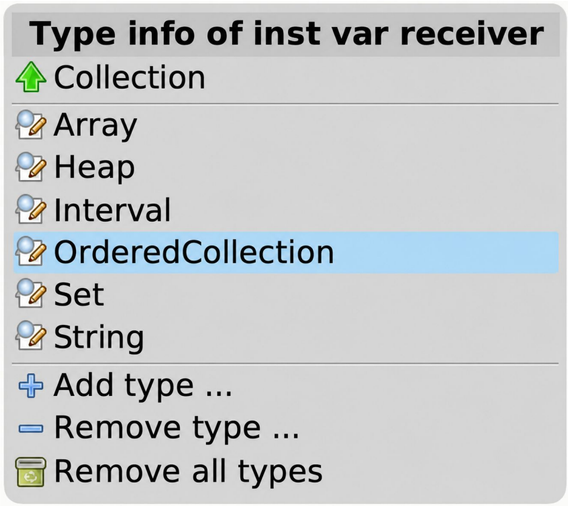
Minimal Background



LiveTyping

Before

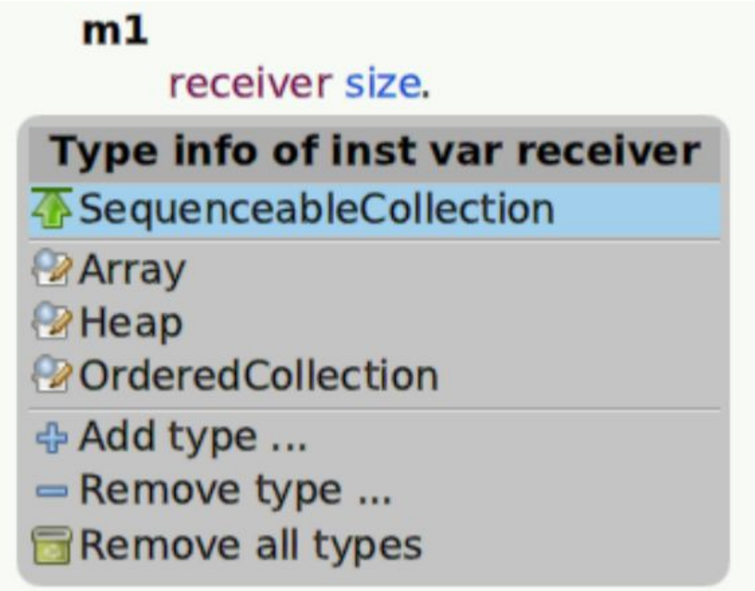
After



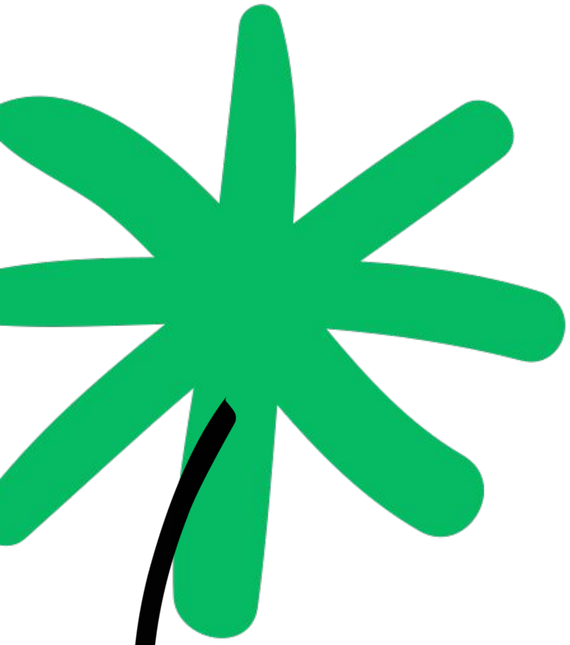
Code that assigns values to the receiver instance variable is executed

```
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
        ]
    ]
]
```



What we’ve done

(Summary)

@OSVM + CUIS

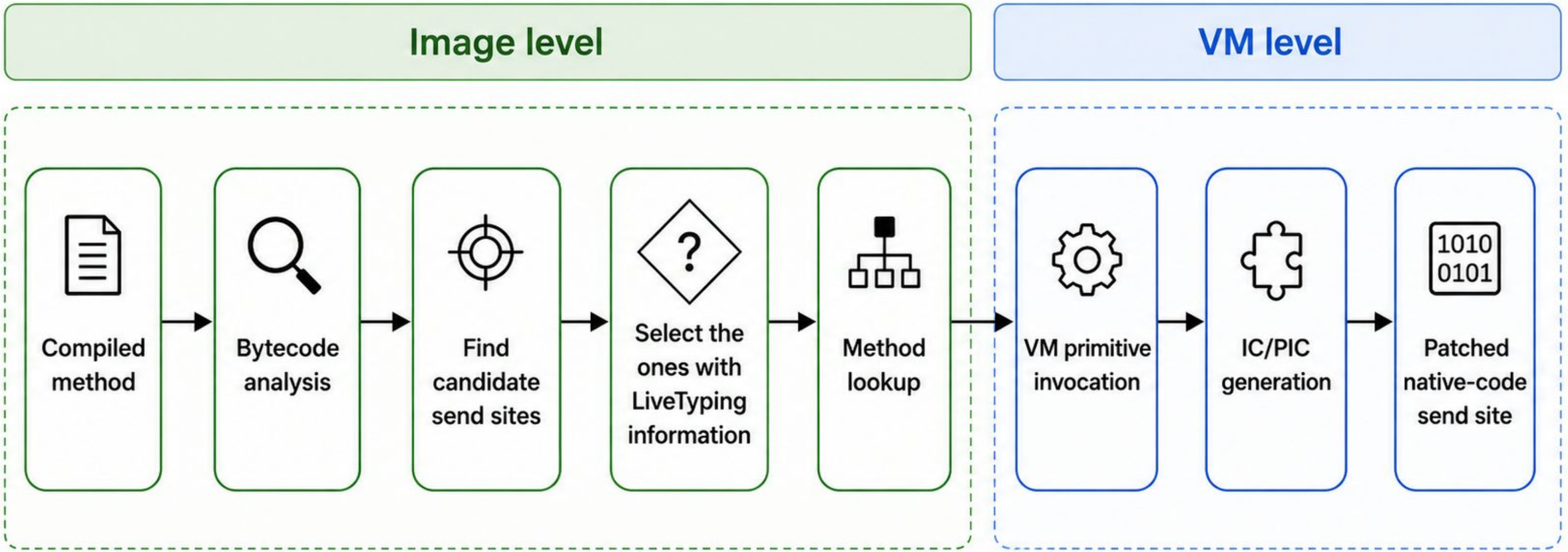
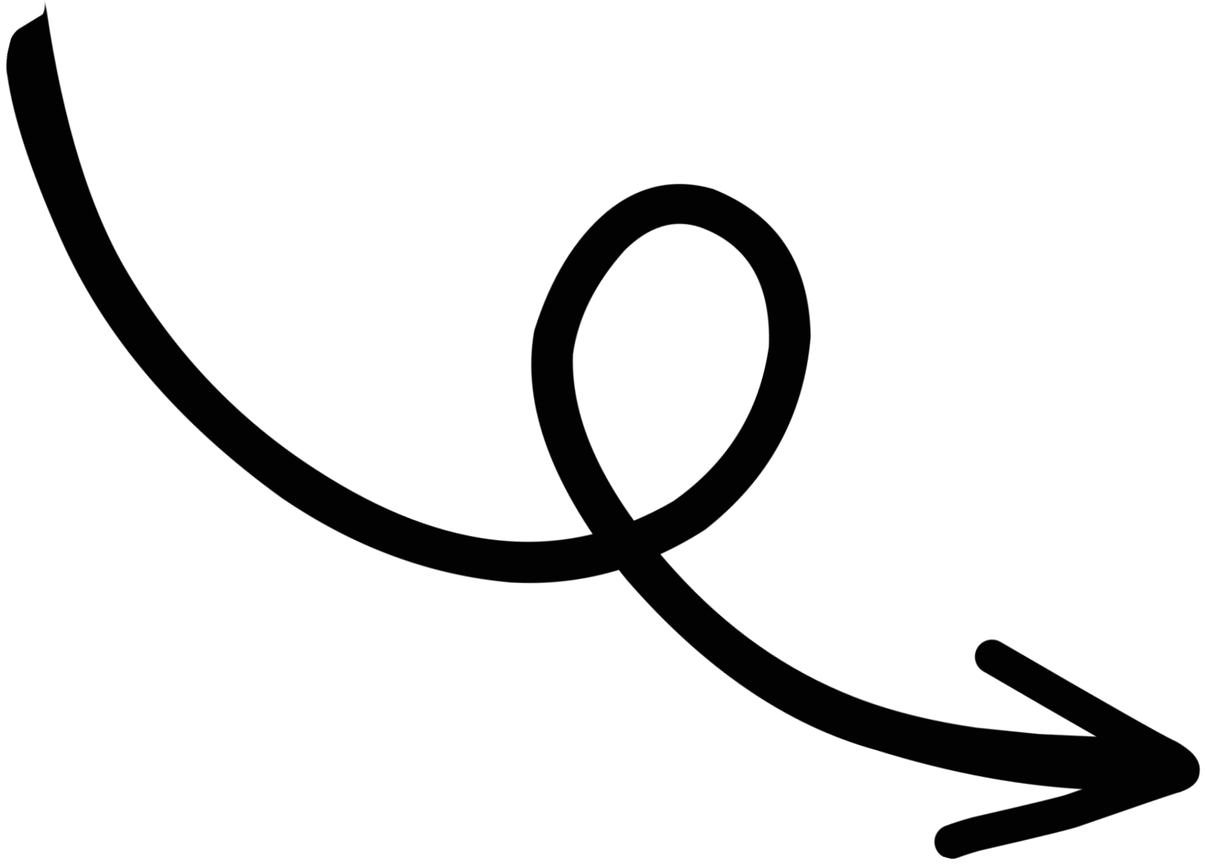


Image level
VM level
Compiled method
Bytecode analysis
Find candidate send sites
Select the ones with LiveTyping information
Method lookup
VM primitive invocation
IC/PIC generation
1010 0101
Patched native-code send site



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?

Experimental Setup

Workloads:

Controlled	Dynamic
● Benchmarks: **size** ; **copy** ; **asArray** ● Repeated sends of the same selector ● Six possible receiver classes: ○ OrderedCollection ; Array ; String ; Heap ; Interval ; Set ● Stable receiver pattern	● Benchmark: **depth** ● Dynamically changing receiver ● Six possible receiver classes: ○ DTLeaf1, DTLeaf2, … , DTLeaf6 ● Pseudo-random order

Variables:

- **Number of send sites: 1, 2, 3, 5, 8, 13, 21, 34**
- **Pre-generated PIC entries: 1 … 6**
- **Iterations: [1 - 10] ; [20] ; [50] ; [100]**

Experimental Setup

Example: size benchmark family

s1

```
receiver size.
```

s2

```
receiver size.
receiver size.
```

s3

```
receiver size.
receiver size.
receiver size.
```

...

s34

```
receiver size.
receiver size.
receiver size.
receiver size.
receiver size.
receiver size.
receiver size.
...
receiver size.
receiver size.
receiver size.
receiver size.
receiver size.
```

The suffix indicates the number of send sites

Experimental Setup

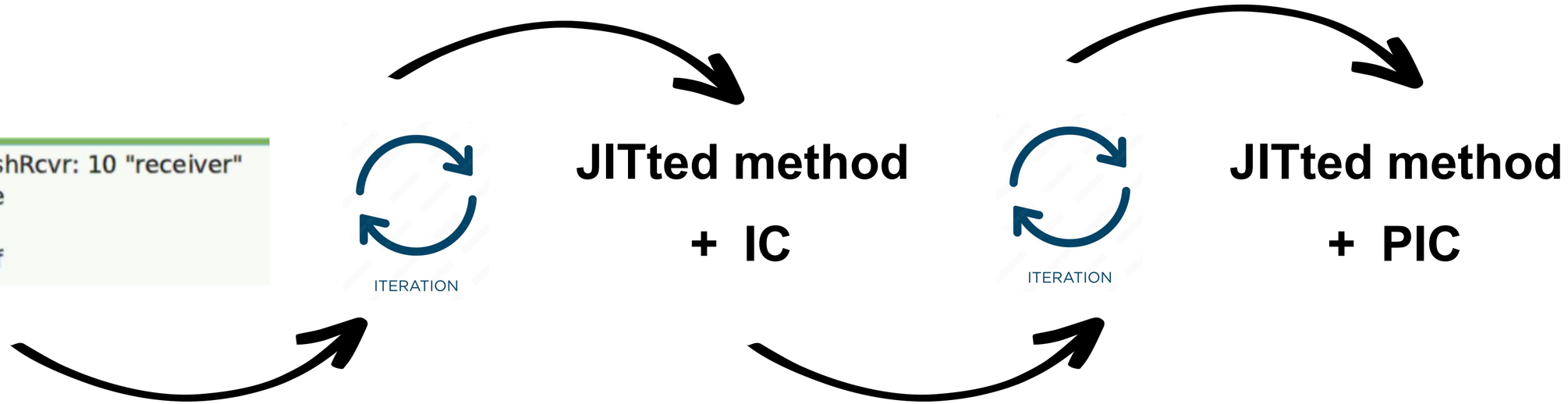
```
runAndMeasure: aMethod forReceivers: receivers

    | total took batchTotal |

    total := 0.
    Smalltalk garbageCollect.
    batchOk := true.
    gcTimes := Smalltalk vmParameterAt: 9.

    [ effectiveBatches < timesToIterate ] whileTrue: [
        batchTotal := 0.

        (1 to: 6) do: [:picEntry |
            receiver := receivers at: picEntry.
            took := self calculateTookTime: aMethod.
            (gcTimes = (Smalltalk vmParameterAt: 9)) ifTrue: [
                batchTotal := batchTotal + took.
            ] ifFalse: [
                batchOk := false.
                "Fail and discard run -- start over"
            ]
        ].

        total := total + batchTotal.
    ].

    ^total
```

Execution Scenarios

Scenario	What it represents?
CB	Current VM behavior
JIT	JIT-compiled method without a proactively generated PIC
PIC	Our proposed approach
WC	Worst-case execution without cache stabilization

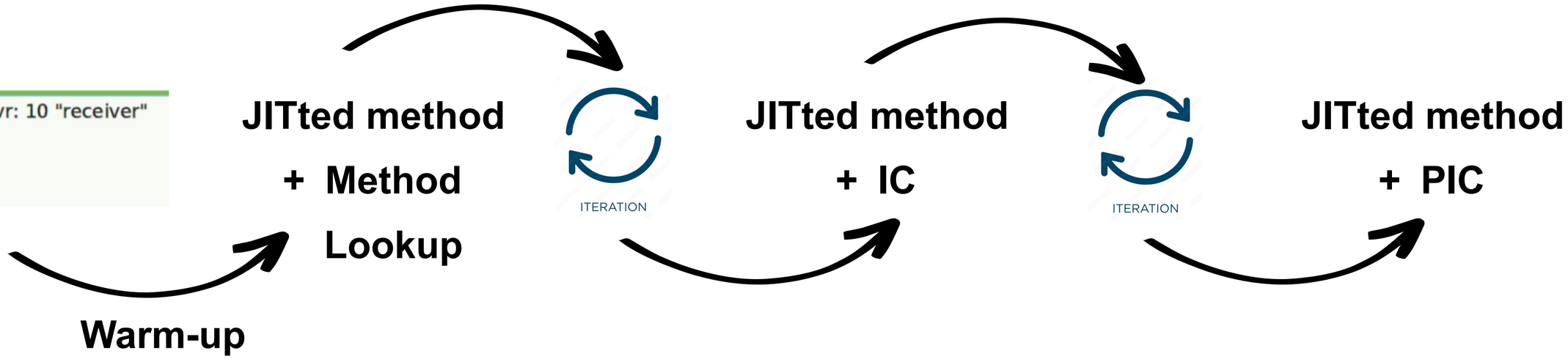
CB: Current VM Behaviour

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



(Only) JIT

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



PIC (This work)

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

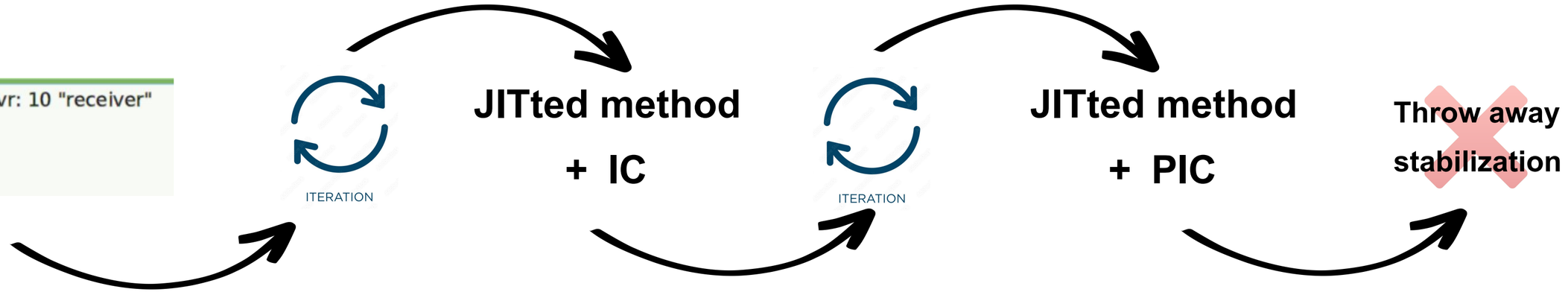
Warm-up

JITted method

+ PIC

WC

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

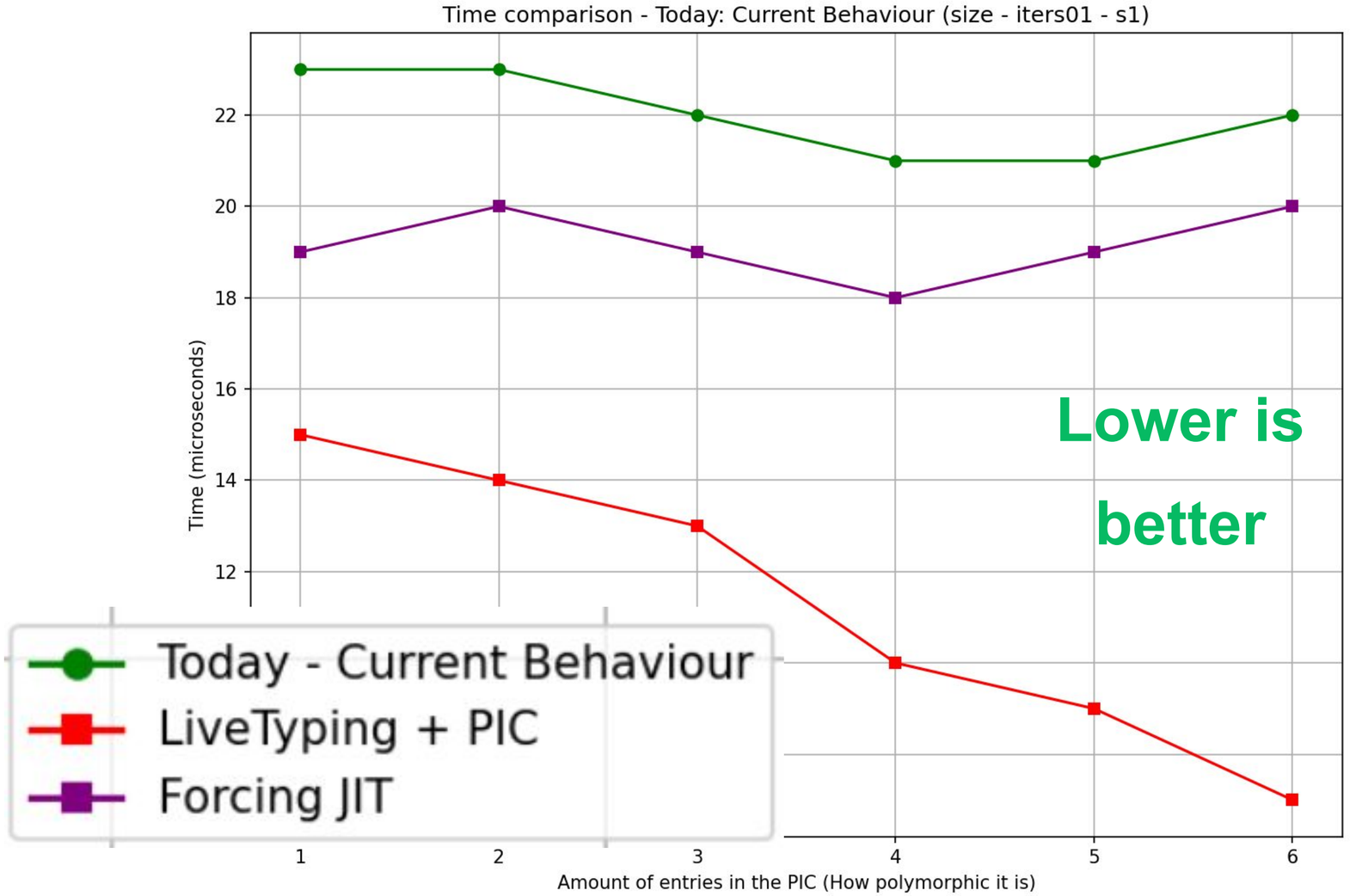


Measurement Environment

Controlled Execution

- **Cuis image** running in **reader/headless** mode
- **Automated** execution through scripts
- **Fresh image** for each run
- **100 repetitions** per configuration
- Measurements affected by **GC** were **discarded**
- Proxmox **containers**: 2 vCPUs, 1 GB RAM
- Execution time measured in **microseconds**

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?

Family: **copy**

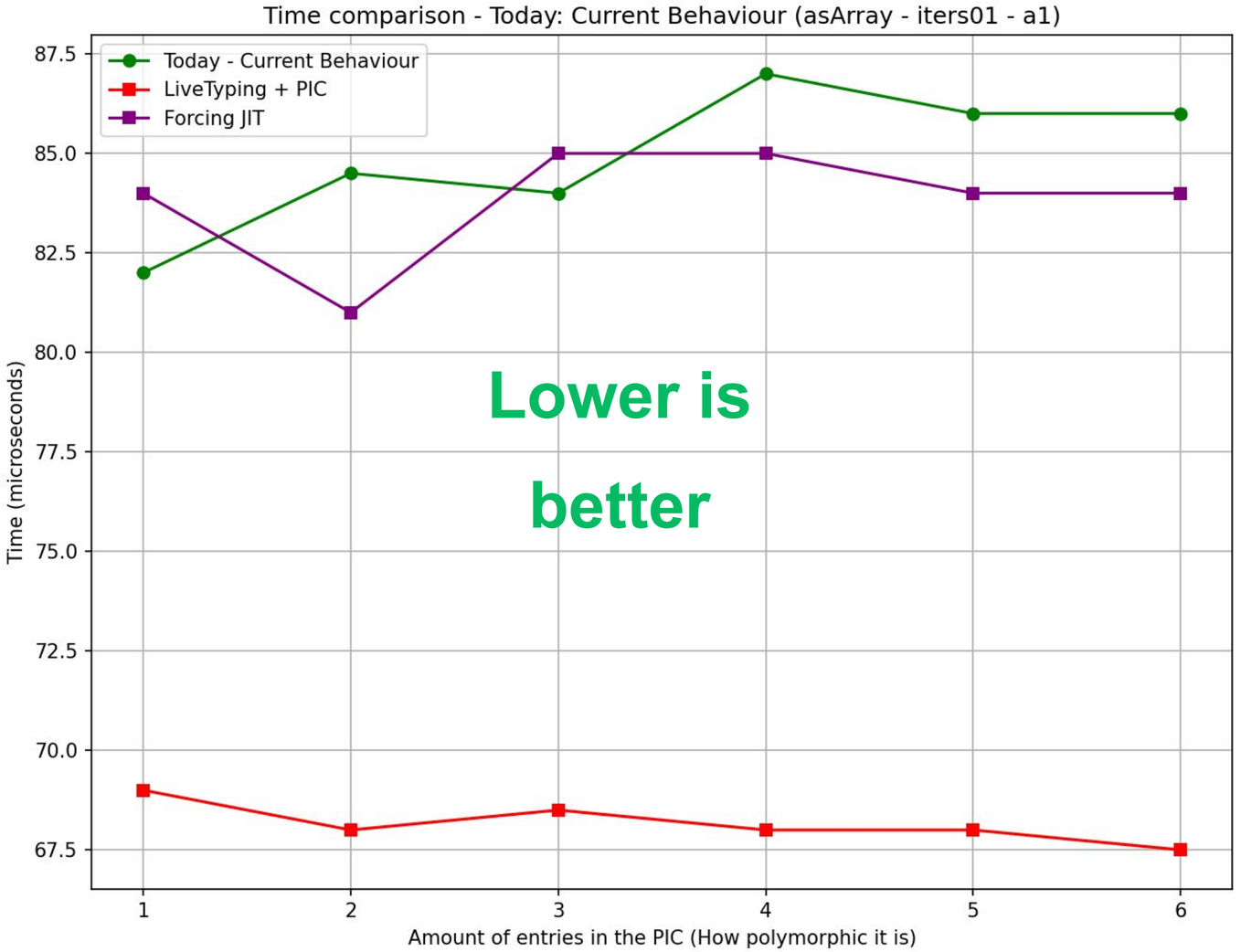
Iterations: **1**

Send sites: **1**

PIC vs CB

# PIC Entries	% Faster
1	46 %
2	55 %
3	59 %
4	47 %
5	47 %
6	61 %

RQ1. Does proactive PIC generation reduce execution time?



Family: **asArray**

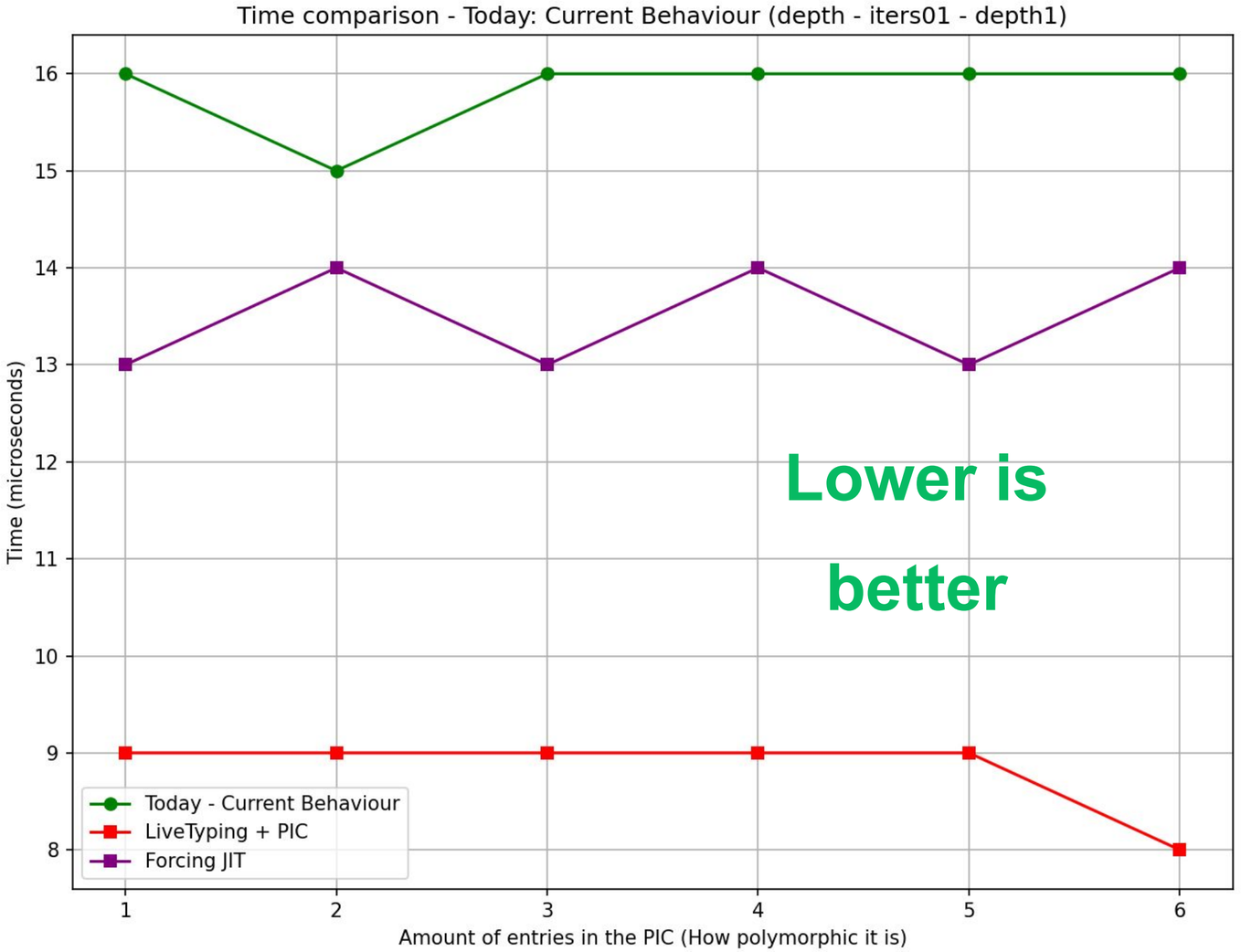
Iterations: **1**

Send sites: **1**

PIC vs CB

# PIC Entries	% Faster
1	18,8 %
2	24,3 %
3	22,6 %
4	27,9 %
5	26,5 %
6	27,4 %

RQ1. Does proactive PIC generation reduce execution time?



Family: **depth**

Iterations: **1**

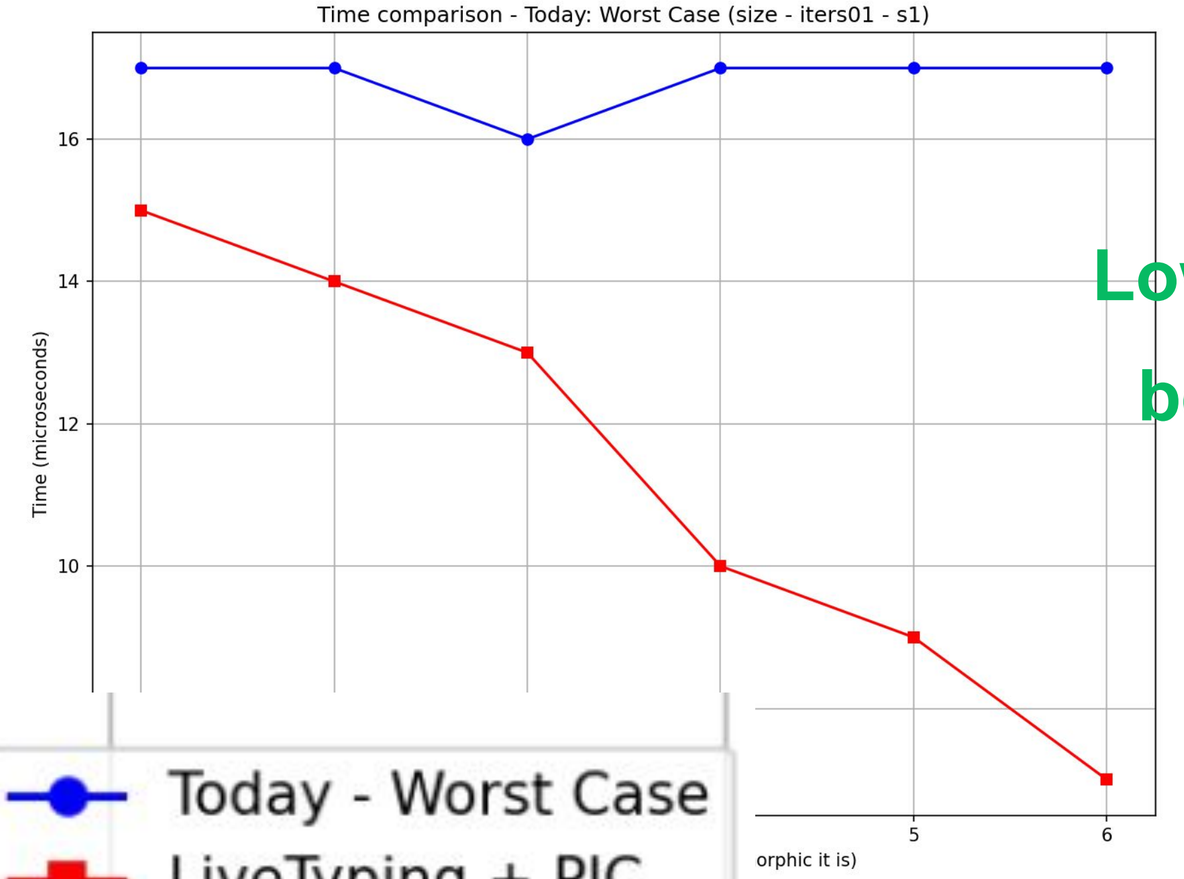
Send sites: **1**

PIC vs CB

# PIC Entries	% Faster
1	78 %
2	67 %
3	78 %
4	78 %
5	78 %
6	100 %

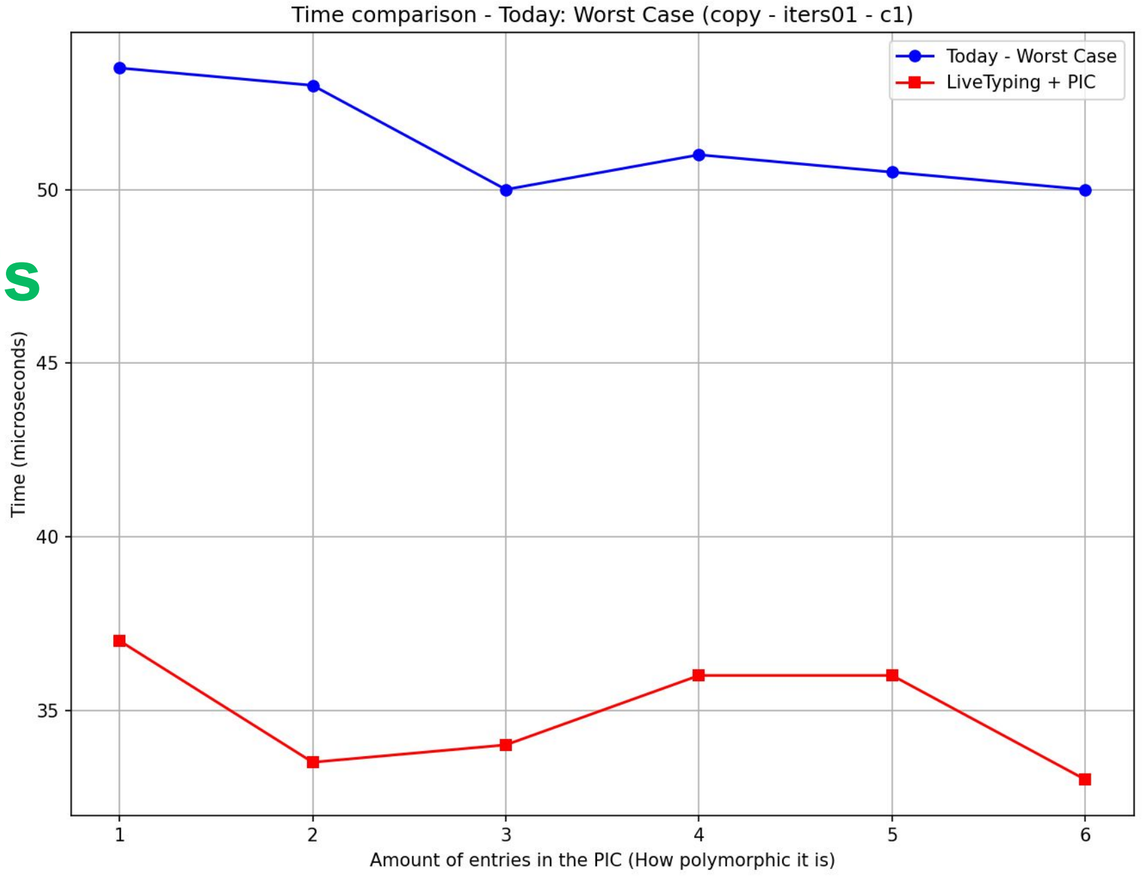
RQ1. Does proactive PIC generation reduce execution time?

Family: **size –** Iterations: **1 –** Send sites: **1**



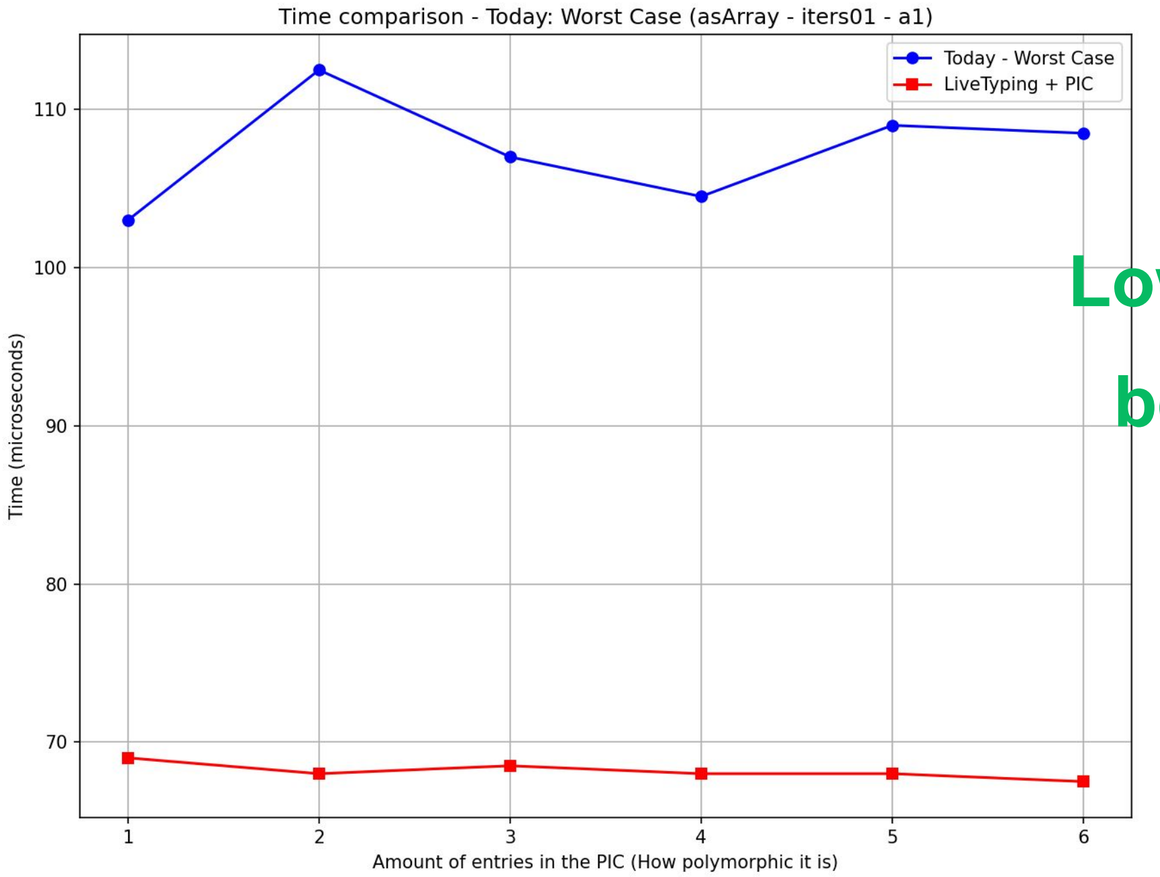
Lower is better

Family: **copy –** Iterations: **1 –** Send sites: **1**



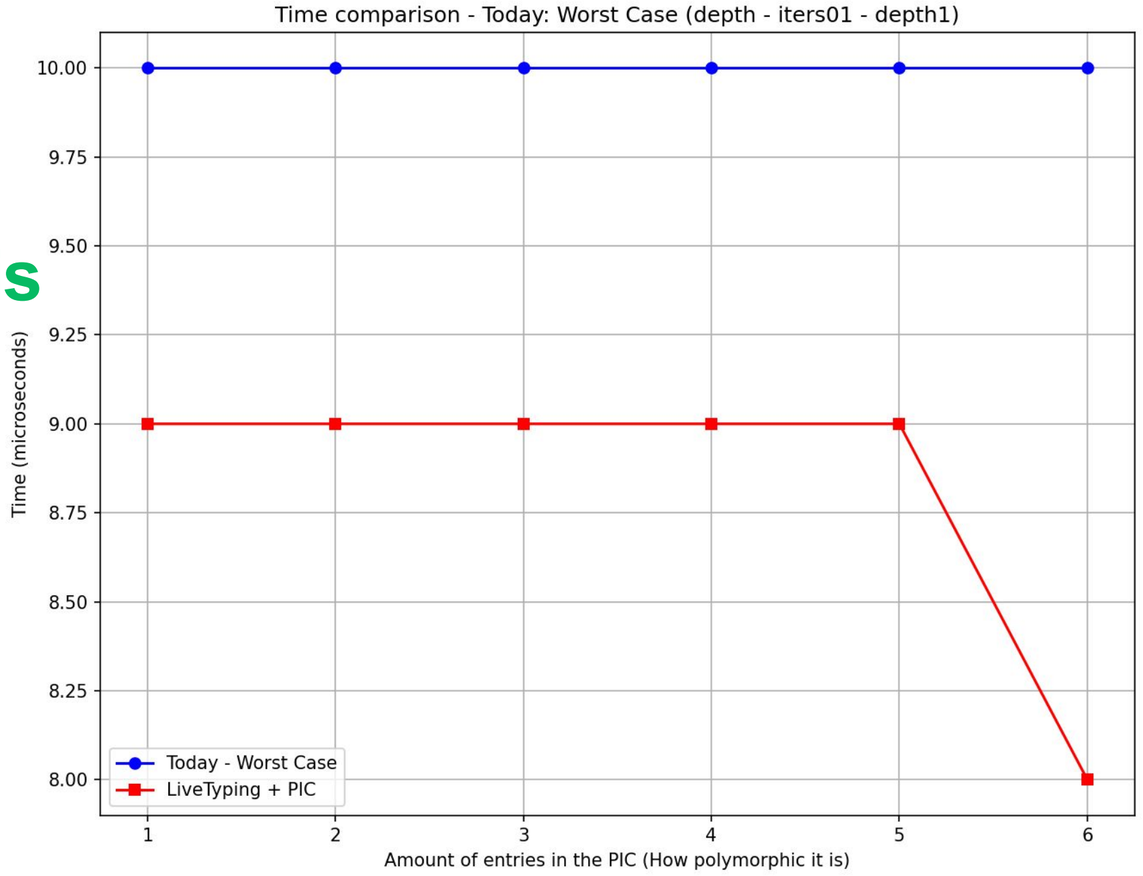
RQ1. Does proactive PIC generation reduce execution time?

Family: **asArray –** Iterations: **1 –** Send sites: **1**



Lower is better

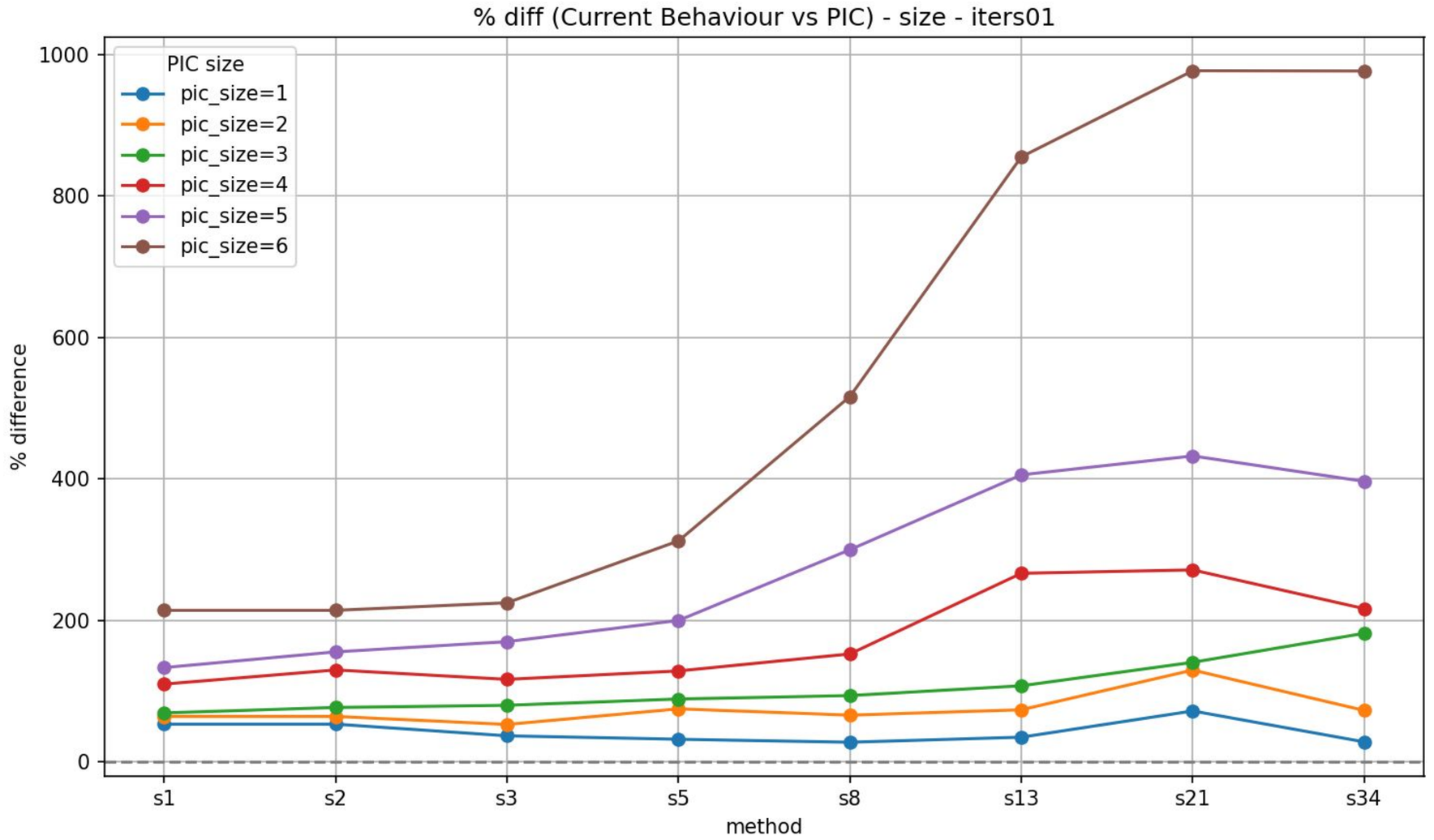
Family: **depth –** Iterations: **1 –** Send sites: **1**



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?



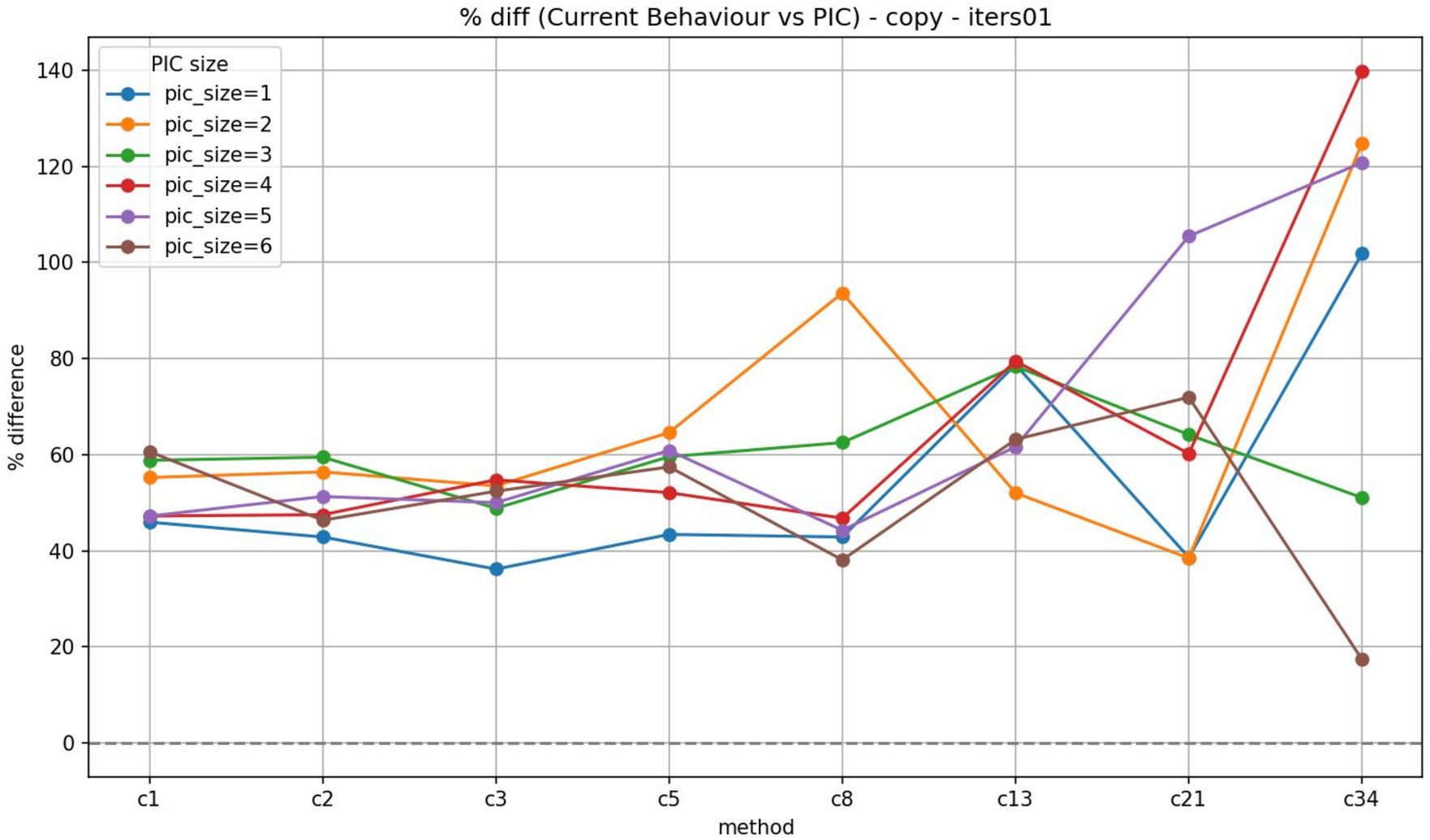
Family: **size**

Iterations: **1**

Send sites: **1 … 34**

Higher is better

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



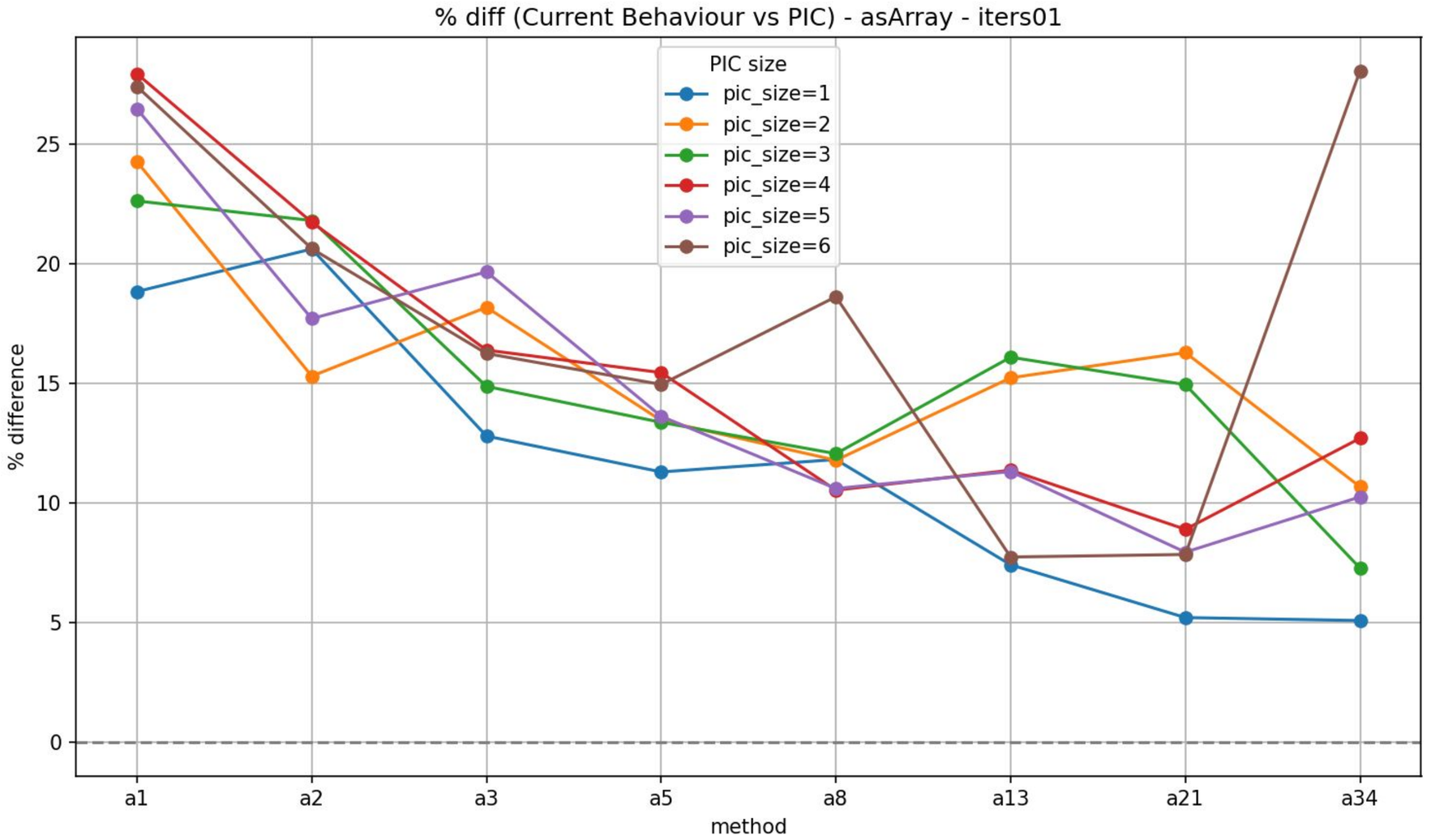
Family: **copy**

Iterations: **1**

Send sites: **1 … 34**

Higher is better

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



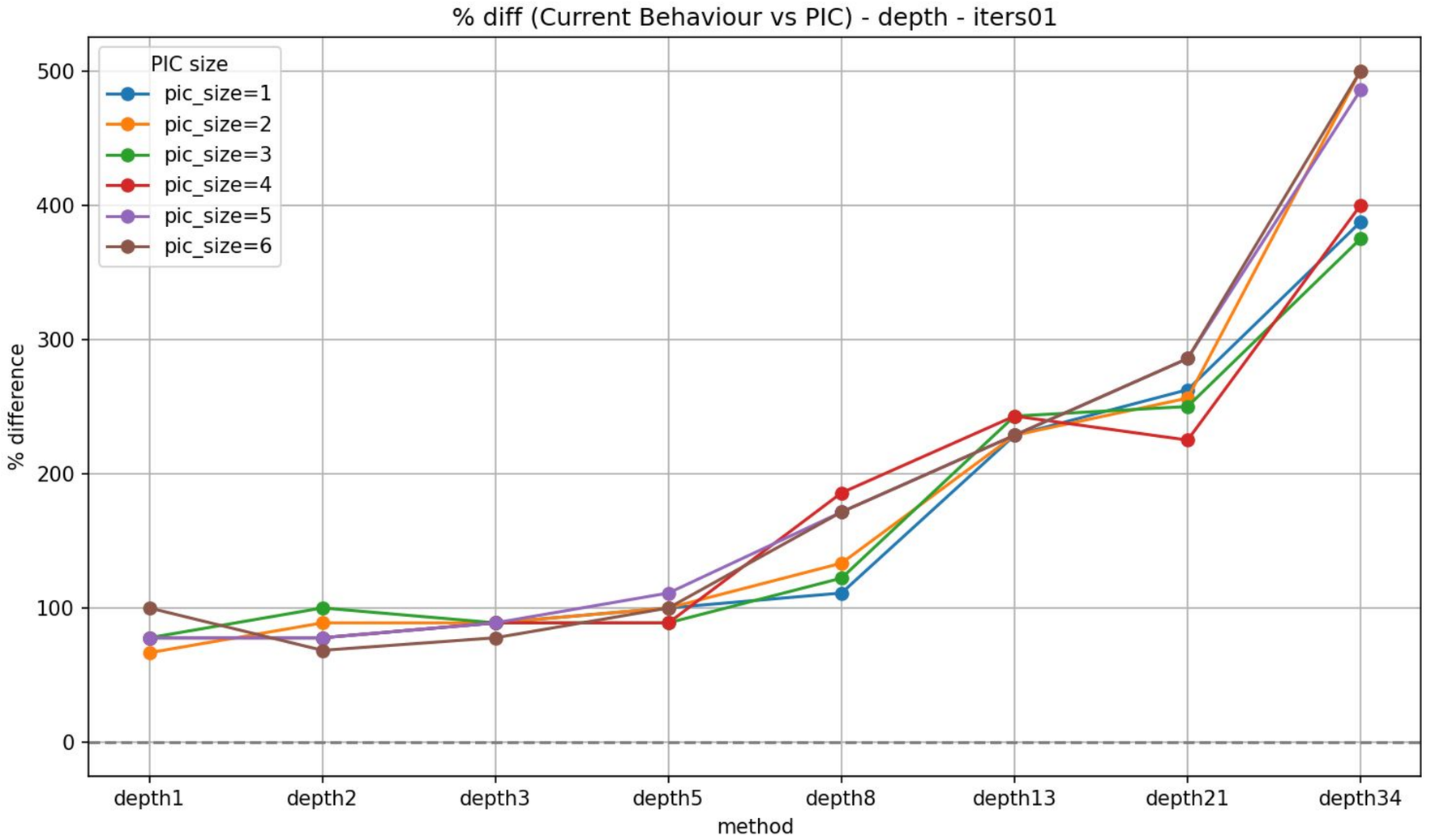
Family: **asArray**

Iterations: **1**

Send sites: **1 … 34**

Higher is better

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



Family: **depth**

Iterations: **1**

Send sites: **1 … 34**

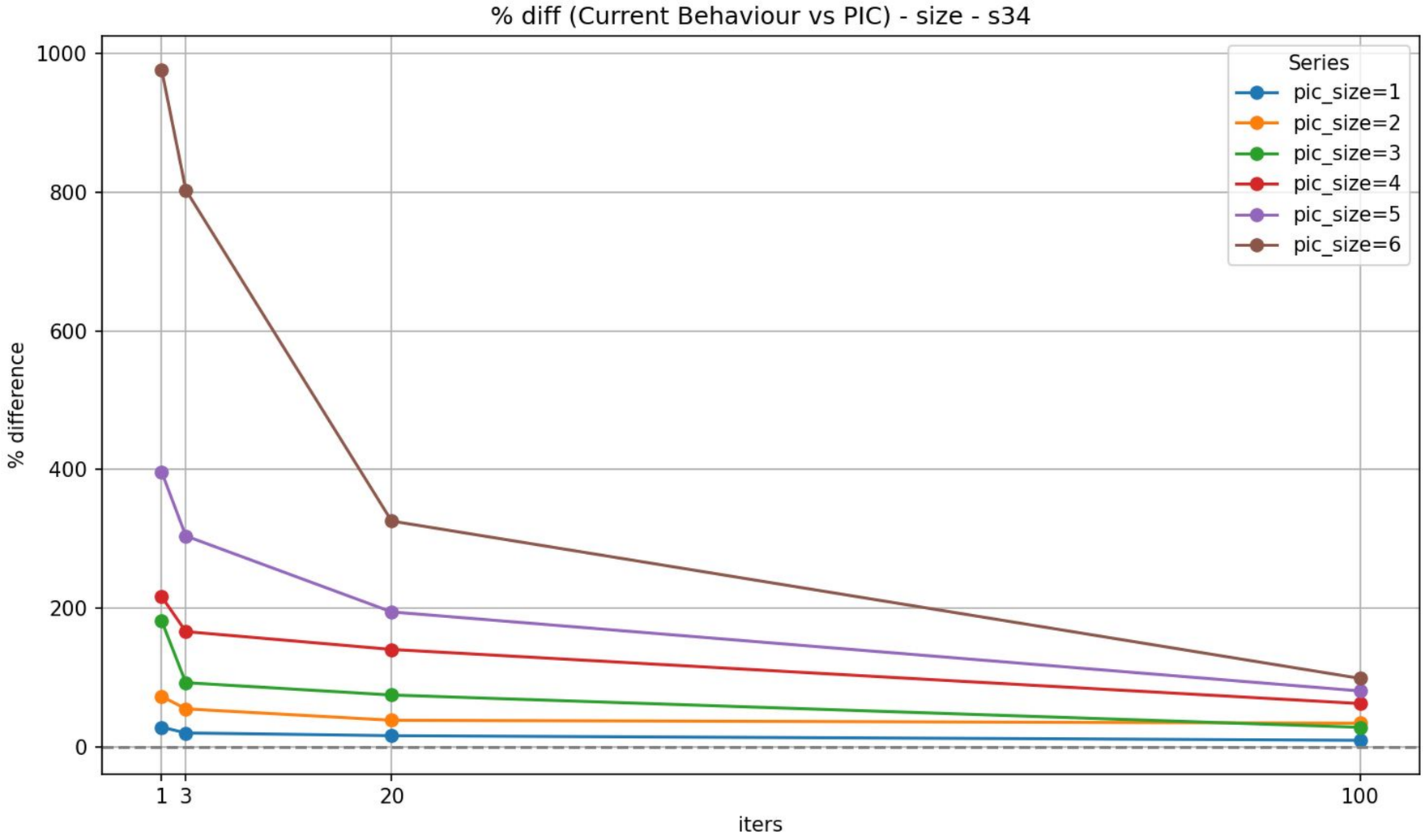
Higher is better

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?



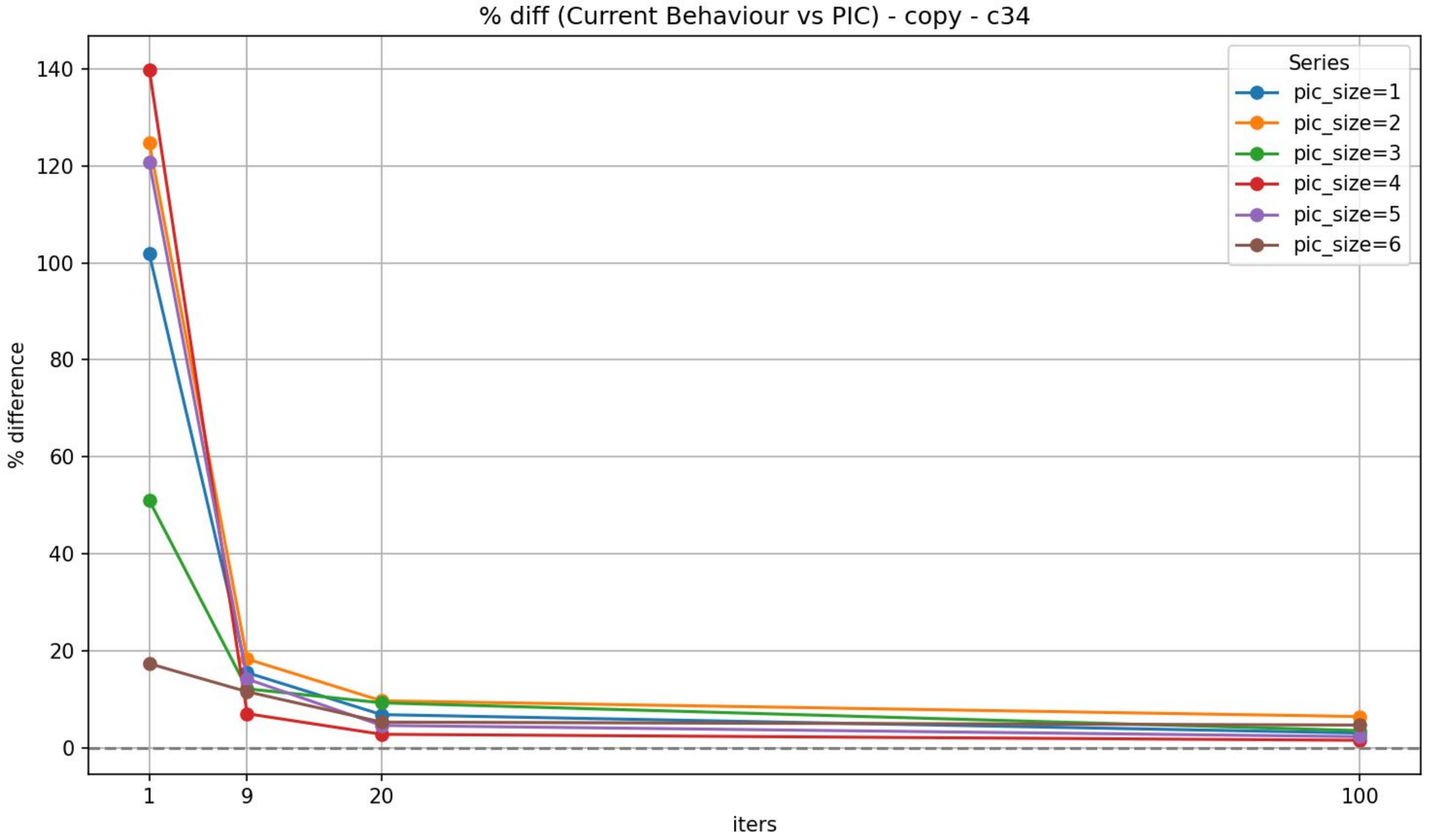
Family: **size**

Iterations: **1 … 100**

Send sites: **34**

Higher is better

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



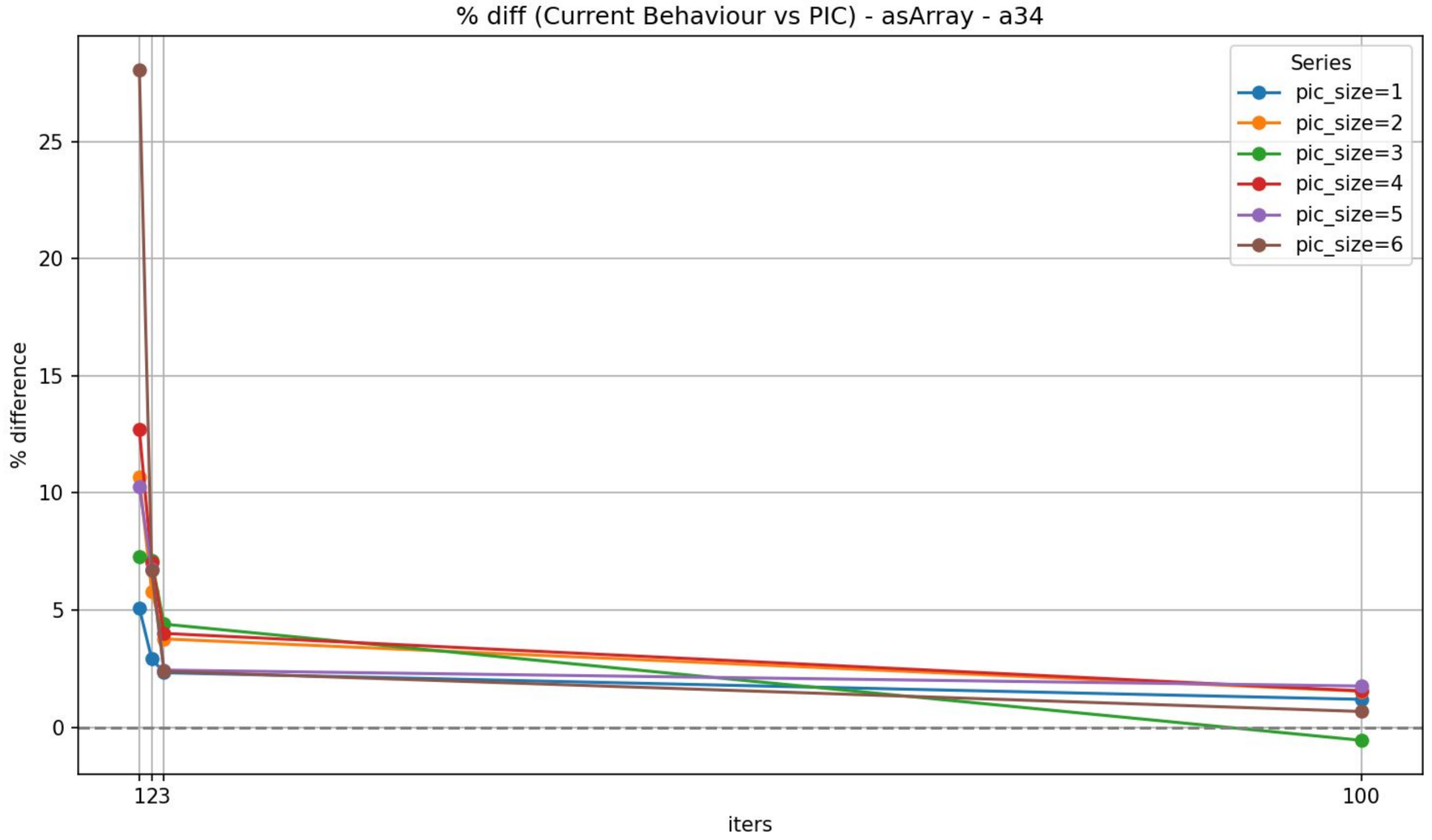
Family: **copy**

Iterations: **1 … 100**

Send sites: **34**

Higher is better

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



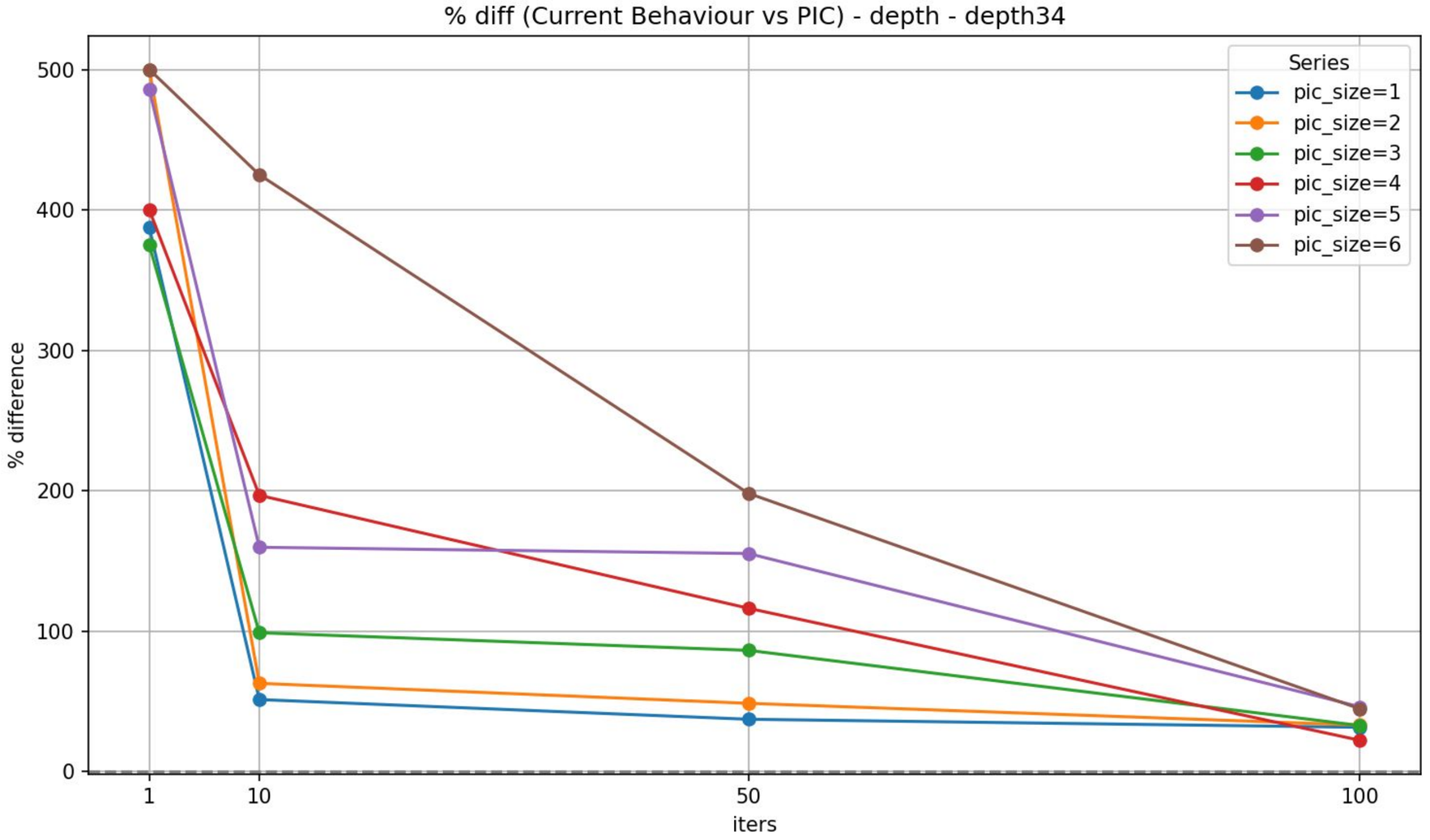
Family: **asArray**

Iterations: **1 … 100**

Send sites: **34**

Higher is better

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



Family: **depth**

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	#JIT	JIT bytes	#closed PICs	closed PIC bytes
1 iteration / 1 send site	5.8 ×	6.1 ×	7.5 ×	7.5 ×
1 iteration / 34 send sites	5.8 ×	5.5 ×	1.4 ×	1.4 ×
100 iterations / 1 send site	5.8 ×	6.1 ×	7.5 ×	7.5 ×
100 iterations / 34 send sites	5.8 ×	5.5 ×	1.4 ×	1.4 ×

VM Impact: **PIC / CB ratio**

Family: **size**

RQ4. What is the cost at the VM level?

Configuration	#JIT	JIT bytes	#closed PICs	closed PIC bytes
1 iteration / 1 send site	4.6 ×	5.5 ×	4.5 ×	4.5 ×
1 iteration / 34 send sites	4.6 ×	5 ×	1.4 ×	1.4 ×
100 iterations / 1 send site	4.6 ×	5.5 ×	4.5 ×	4.5 ×
100 iterations / 34 send sites	4.4 ×	4.7 ×	1.5 ×	1.5 ×

VM Impact: **PIC / CB ratio**

Family: **copy**

RQ4. What is the cost at the VM level?

Configuration	#JIT	JIT bytes	#closed PICs	closed PIC bytes
1 iteration / 1 send site	5.3 ×	5.7 ×	3 ×	3 ×
1 iteration / 34 send sites	5.3 ×	5.2 ×	1.4 ×	1.4 ×
100 iterations / 1 send site	5.3 ×	5.7 ×	3 ×	3 ×
100 iterations / 34 send sites	4.9 ×	4.9 ×	1.4 ×	1.4 ×

VM Impact: **PIC / CB ratio**

Family: **asArray**

RQ4. What is the cost at the VM level?

Configuration	#JIT	JIT bytes	#closed PICs	closed PIC bytes
1 iteration / 1 send site	2.7 ×	3.5 ×	0 → 12	0 → 2.53 KB
1 iteration / 34 send sites	2.7 ×	3.3 ×	0 → 45	0 → 9.49 KB
100 iterations / 1 send site	2.3 ×	3.3 ×	6.5 ×	6.5 ×
100 iterations / 34 send sites	2.3 ×	3.2 ×	1.3 ×	1.3 ×

VM Impact: **PIC / CB ratio**

Family: **depth**

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?