Clean Blocks at the Opal Compiler

Nahuel Palumbo, Marcus Denker

▶ **To cite this version:**

Nahuel Palumbo, Marcus Denker. Clean Blocks at the Opal Compiler. Proceedings of the International Workshop on Smalltalk Technologies 2025, 2025. ⟨hal-05299729⟩

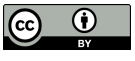
HAL Id: hal-05299729

https://hal.science/hal-05299729v1

Submitted on 6 Oct 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Clean Blocks at the Opal Compiler

Nahuel Palumbo[1], Marcus Denker[1]

[1]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Lille, France

Abstract

Higher-order languages encourage programmers to use lambda functions or closures. In Smalltalk, we see a lot of use of block closures, for example, in the collection API.

Closures are expensive as they have to be created at runtime, impacting code execution efficiency. However, there are closures with code agnostic to the context where they have been defined and, thus, can be created at compile time.

In this paper, we present Clean Blocks, an optimization implemented in the Pharo compiler for detecting and creating closures independent of the context at compile time. We also implemented a specialization for Constant Blocks, *i.e.,* closures that only return constant values.

We evaluate the impact of this optimization, statically in a new Pharo image, and dynamically by running several benchmarks and measuring closure activations and execution time. The results show that our optimization improves performance up to 1.35x, and leaves some open questions regarding the Garbage Collection overhead.

Keywords

Compiler, Optimization, Block Closure, Opal, Pharo

1. Motivation

Higher-order languages permit the definition of generic algorithms by receiving functions as parameters. While Smalltalk-80 used a limited model [1], modern Smalltalk implementations implement blocks as real closures. Closures have access to the variables of the context where they are created, even if activated after the creation context is already finished executing.

For example, the method `giveMeANumber` in the following code has a local variable `x := 10` that is accessed by the block closure `[ x + 7 ]`. On closure evaluation, even if it happens in another context, the variable x from the method where it was created is read, returning `17`.

```
SomeClass >> giveMeANumber
        | x |
        x := 10.
        ^ self evaluate: [ x + 7 ]

SomeClass >> evaluate: blockClosure
        ^ blockClosure eval
```

Thus, closures need to be created during method execution, putting pressure on the execution engine's performance.

However, some closures do not need to access values on the enclosing context because their code is agnostic to the creation context. For example, the closure `[ nil ]` in the following method will *always* return the constant `nil`, regardless of the method context.

```
Chart >> verticalTick
        ^ decorations detect: #isVerticalTick ifNone: [ nil ]
```

We call these closures independent of the method context *Clean Blocks*. Since they do not need to be bound to the current execution context, they can be **created at compile time**, before execution.

IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland

*Corresponding author.

†These authors contributed equally.

nahuel.palumbo@inria.fr (N. Palumbo); marcus.denker@inria.fr (M. Denker)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In this paper we describe

- **Implementation of *Clean Blocks*.** The implementation of compile-time closure creation optimization in Pharo.
- **Specialization for *Constant Blocks*.** A specialization for *Clean Blocks* that only returns constants, improving closures evaluation.
- **Impact on the runtime.** Analysis of the impact of the presented optimization on the runtime system over many benchmarks.

2. Implementation

We implemented *Clean Blocks* in Pharo, a fully dynamic object-oriented language [2]. The optimization has been completed and stable since December 2022 in Pharo 11[1], and it is used by users in newer distributions. At the moment of this work, only *Constant Blocks* are already enabled by default. *Clean Blocks* have to be enabled by the user; it will be enabled by default in Pharo 14[2].

2.1. Pharo Bytecode

Pharo code is compiled into bytecode and then executed by the Pharo Virtual Machine [3]. The bytecode set is based on a stack machine, and the compiled methods include references to objects in a *literals* section. For example, the bytecodes for the method `verticalTick` presented in Section 1 is:

```
1 <07> pushRcvr: 7                          "Push 'decorations' variable"
2 <20> pushConstant: #isVerticalTick        "Push symbol #isVerticalTick"
3 <F9 01 00> fullClosure: [ nil ]           "Create and Push the Closure"
4 <A2> send: detect:ifNone:                 "Send message #detect:ifNone:"
5 <5C> returnTop                            "Return"
```

We see, in the bytecode number 3 `fullClosure: [ nil ]`, the instruction for closure creation at runtime. This bytecode creates the closure, binds it to the method context, and pushes the block closure object to the stack.

With the *Clean Blocks* optimization activated, the bytecode for the same method looks:

```
1 <07> pushRcvr: 7                          "Push 'decorations' variable"
2 <20> pushConstant: #isVerticalTick        "Push symbol #isVerticalTick"
3 <21> pushConstant: [ nil ]                "Push the Block Closure (already on the literals)"
4 <A2> send: detect:ifNone:                 "Send message #detect:ifNone:"
5 <5C> returnTop                            "Return"
```

Here, the bytecode number 3 is replaced by `pushConstant: [ nil ]` because the block closure was already created at compile time and saved into the literals list.

For closures that are just returning a static value, like the `[ nil ]` in the example, we can even push specialized *Constant Blocks* that, on execution, just return a constant, speeding up closure evaluation.

2.2. Compiling Clean and Constant Blocks

We implemented the *Clean Blocks* optimization by extending the OpalCompiler [4]. The optimization is based on a closure AST analysis during method compilation. When the compiler has to compile a closure node, the following code is executed to determine if it is clean:

```
BlockNode >> isClean
        "It or any embedded blocks do not need self (message receiver object)"
        self isAccessingSelf ifTrue: [ ^ false ].
        "It or any embedded blocks do not need the outer context to return"
        self hasNonLocalReturn ifTrue: [ ^ false ].
        "There are no escaping vars accessed from the outer context"
        self scope hasEscapingVars ifTrue: [ ^ false ].
        ^ true
```

[1] https://github.com/pharo-project/pharo/issues/11195
[2] https://github.com/pharo-project/pharo/pull/18322

If the closure is clean, then the compiler instantiates a `CleanBlockClosure` object and generates a `pushConstant:` bytecode for it. Otherwise, it just compiles a `fullClosure:` instruction as always. Both examples were presented previously.

The compiler checks in addition if the block is returning just a constant (literal) for specialize *Constant Blocks*:

```
BlockNode >> isConstant
        "Is the block just returning a literal?"
        ^ body statements
                ifEmpty: [ true "empty block returns nil" ]
                ifNotEmpty: [:statements | statements size = 1 and: [statements first isLiteralNode] ]
```

In this case, the compiler creates an instance of `ConstantBlockClosure`, a subclass of `CleanBlockClosure` that implements a specialized code path to return the constant when the closure is evaluated.

2.3. Execution of Constant Blocks

The only difference between *Constant Blocks* and *Clean Blocks* is the evaluation of the block. Both are created at compile time, thus have the same performance characteristics for creation. However, the *Constant Blocks* overrides the `value` method used to evaluate the closure.

```
FullBlockClosure>>#value
        "Activate the receiver, creating a closure activation (MethodContext)
         whose closure is the receiver and whose caller is the sender of this
         message. Supply the copied values to the activation as its copied
         temps. Primitive. Essential."
        <primitive: 207>
        numArgs ~= 0 ifTrue:
                [self numArgsError: 0].
        ^self primitiveFailed

ConstantBlockClosure>>#value
        ^literal
```

For *Full Blocks*, as well as *Clean Blocks*, the method calls a VM primitive that executes the `CompiledBlock` of the closure. The failure code is only executed on primitive failure, leading to a fast code path for the common case with no argument error.

For *Constant Blocks*, the constant literal is stored in the object and just returned. To model the failure behavior related to the number of arguments efficiently, we have subclasses for zero, one, two, and three arguments. A constant closure with four or more arguments is always compiled as a `CleanBlockClosure`.

```
ConstantBlockClosure>>#value: anObject
        self numArgsError: 1
```

The specialized class with overrides for each case leads to code without any conditions, and thus faster execution.

The bytecode of the `CompiledBlock` for *Constant Blocks* pushes the constant and then returns from the block:

```
17 <4F> pushConstant: nil
18 <5E> blockReturn'
```

Interestingly, the `ConstantBlockClosure` is compiled like *Clean Blocks* with a `CompiledBlock`, even though this bytecode is never executed. The reason for this approach is that the system does not need any special case for code relying on the bytecode of a block. An example for this is the *senders of* feature: it will find a #symbol in *Constant Blocks* without the need to treat them specially.

3. Evaluation

We divide the evaluation of the presented work into three parts: First, we measure the improvements of the specific optimized areas of the runtime, *i.e.,* closure creation for *Clean Blocks* and closure execution

for *Constant Blocks*, to see the impact in the best scenario possible. Later, we analyse the distribution of clean or constant closures in the base code, to approximate the scope of our optimizations on the Pharo programs. Finally, measure the impact on the overall runtime by running more than 20 benchmarks.

3.1. Optimization improvements

Clean Blocks is an optimization against closures created at runtime, while *Constant Blocks* optimizes the activation of closures. In this section, we use naive examples to measure the performance improvement achieved by the optimizations for the same trivial block. For them, we use the tools presented in Pharo 12, especially the method `BlockClosure » bench` that measures how many times a block is evaluated within 5 seconds.

3.1.1. Clean Blocks: Creation

Clean Blocks are created at compile time, avoiding the overhead of creating a closure at runtime. We measure the performance gain with a naive, tiny benchmark. We compile the closure `[1+2]` with and without the optimization for *Clean Blocks*:

```
"execute line by line"
OCCompilationContext optionCleanBlockClosure: true.
cleanEnabled := [ [1+2] ]  bench.
OCCompilationContext optionCleanBlockClosure: false.
cleanDisabled := [ [1+2] ]  bench.

cleanDisabled compareTo: cleanEnabled  "43351950.020/s * 9.073 = 393318782.487/s"
```

The result of this particular case shows a speedup of 9x for the creation of *Clean Blocks* over *Full Blocks*.

3.1.2. Constant Blocks: Execution

We compare the execution of *Constant Blocks* over normal activation of the `CompiledBlock` by the VM. We run a trivial, tiny benchmark of evaluating the block `[ nil ]` compiled as `ConstantBlockClosure` and as normal `BlockClosure`:

```
1  "execute line by line"
2  OCCompilationContext optionConstantBlockClosure: true.
3  block := [ nil ].
4  constEnabled := [  block value ]  bench.
5
6  OCCompilationContext optionConstantBlockClosure: false.
7  block := [ nil ].
8  constDisabled := [ block value ]  bench.
9
10 constDisabled compareTo: constEnabled  "186672246.301/s * 1.414 = 263952774.090/s"
```

The results show a speedup of 1.4x for the execution of *Constant Blocks* over `CompiledBlock` activations.

3.2. Block Distribution

Blocks	Quantity	Percentage
Full Blocks	26060	71%
Clean Blocks	8219	22%
Constant Blocks	2365	6%
All	**36644**	**100%**

Table 1
Distribution of closures on a fresh Pharo 12 image.

We measure the percentage of *Full Blocks*, *Clean Blocks*, and *Constant Blocks* in the base code of Pharo [2] (including kernel, standard libraries, and GUI) to see how many parts of the code are affected by our optimization.

Due to the live environment of Pharo, we get this information by evaluating how many instances of each kind are alive in a Pharo 12 image[3]. The results are presented in Table 1.

We see that 28% of the blocks presented in the base code of Pharo are optimized by our optimization, having 22% of *Clean Blocks* and 6% of *Constant Blocks*.

3.3. Impact on Runtime

We evaluate the impact of the optimization on runtime by running 22 different benchmarks described in Appendix A. We measure the number of activations for each kind of closure and the execution time, per benchmark.

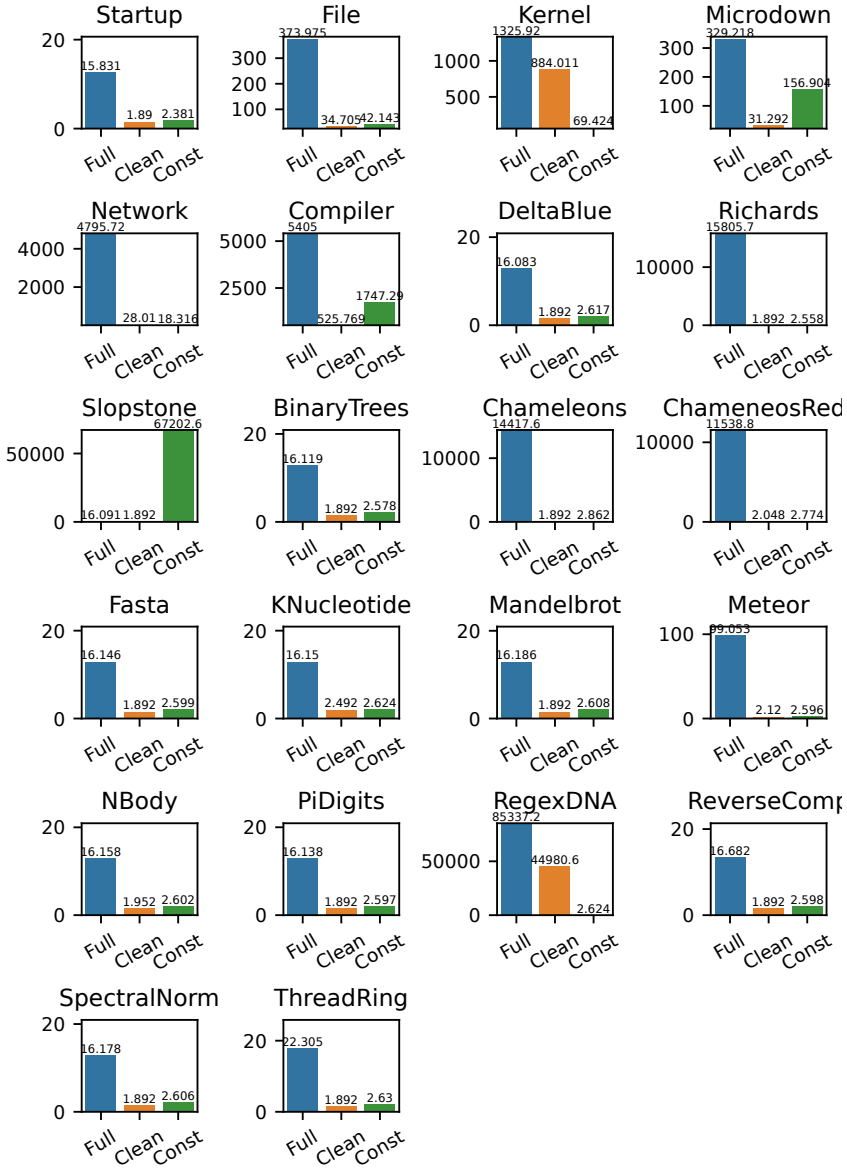


Figure 1: Number of activations for Full, Clean, and Constant Blocks per benchmark. Numbers are in K-activations (x1000).

[3]Build information: Pharo-12.0.0+SNAPSHOT.build.1571.sha.cf5fcd22e66957962c97dffc58b0393b7f368147 (64 Bit)

3.3.1. Setup

We run a set of benchmarks using the Rebench framework [5]. We run all benchmarks using 30 iterations. Micro benchmarks use 10 in-process with two (ignored) warmup iterations. Macro benchmarks, consisting of running the test suite of several packages, use a single in-process iteration.

All the Pharo images[4] used in the evaluation were entirely recompiled with a given compiler configuration (enable/disable *Clean Blocks* and/or *Constant Blocks*) and saved. Thus, *Clean & Constant Blocks* are already allocated in the heap during benchmark executions.

We used a `Linux Debian 5.10.140-1` distribution on a `ARM64 Intel(R) Xeon(R) CPU E5620 @ 2.40GHz` processor with `16 GB DDR3` of RAM.

3.3.2. Closure activations

To understand the impact of *Clean & Constant Blocks* on our benchmark set, we measure the number of Closure activations for each kind of closure. The results are presented in Figure 1.

Except *Slopstones*, where almost all Closure activations are on *Constant Blocks*, all the benchmarks activate more *Full Blocks* than *Clean & Constant Blocks*.

On average, 76% of all the Closure activations are on *Full Blocks*. The rest 24% of the Closure activations made on *Clean & Constant Blocks* are divided into 40% *Clean Blocks* and 60% *Constant Blocks*.

After *Slopstones*, the benchmarks more impacted by our optimization are *Kernel*, *Microdown*, and *RegexDNA*, with Closure activations between ~40% and 35% on *Clean & Constant Blocks*.

3.3.3. Speed up

We evaluate the impact of our optimization on performance by measuring the execution time of each benchmark. Figure 2 presents the result per benchmark. We make the following configurations:

Unoptimized All the closures are compiled as *Full Blocks*. We use this configuration as the baseline.

Optimized Using our optimization, we create *Clean & Constant Blocks* at compile time when it is possible.

As we showed in Section 3.3.2, the benchmark *Slopstones* is the most beneficial from our optimization, so it has the highest speed improvement of 1.35x.

The rest of the benchmarks present a speed-up variation between 0.95x and 1.07x, being *BinaryTrees* and *ThreadRing* the worst cases. The average speed-up of all the benchmarks in our suite is 1.02x.

These unexpected slowdowns and tiny increments are related to the quantity of *Clean & Constant Blocks* activated during all benchmarks (24%) and due to the increasing pressure on the Garbage Collector, which we analyze in the next section.

3.3.4. Garbage Collection overhead

As our optimization creates *Clean & Constant Blocks* before execution, it allocates more objects in the heap during compilation, increasing the pressure on the Garbage Collector during compilation while reducing it at execution time. We measure the time that the Garbage Collector is running for each benchmark on each configuration. Results are presented in Figure 3.

We observe that the benchmarks with slowdown presented in Section 3.3.3, *BinaryTrees* and *ThreadRing*, show the highest increment of Garbage Collection overhead: 1.21x and 1.18x, respectively, relative to *Full Blocks*.

However, there are other benchmarks where the overhead decreased compared to *Full Blocks*. For example, *Compiler* and *Microdown* present a Garbage Collection overhead of 0.77x and 0.80x, respectively, relative to *Full Blocks*.

[4]Memory snapshot of a Pharo program.

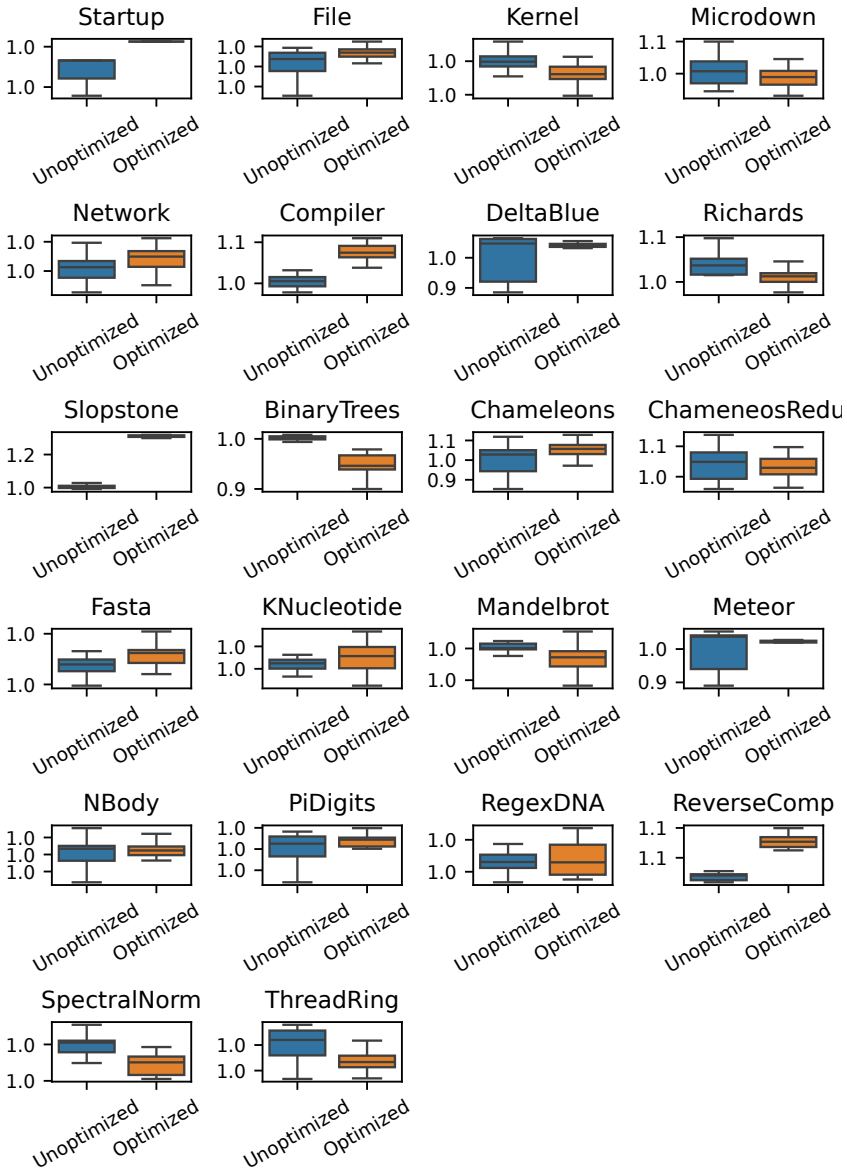


Figure 2: Speed-up using *Clean & Constant Blocks* relative to *Full Blocks*. Higher is better.

On average, the Garbage Collection overhead of *Clean & Constant Blocks* along all benchmarks is 1.01x the overhead of *Full Blocks*.

These are unexpected results. We thought that pre-allocating closures at compile time instead of runtime would reduce the pressure on the Garbage Collector. A better understanding of why our optimization impacts the Garbage Collection overhead in different ways is something to be analyzed in the future by the authors.

4. Related work

Sussman and Steele [6] introduced closures in Scheme as first-class entities that encapsulate both function code and the lexical environment in which they were defined.

Goldberg and Robson [1] describe the implementation of blocks in Smalltalk-80, including the representation of execution contexts and optimizations for method lookup and block invocation. It should be noted that blocks in Smalltalk 80 are not closures.

There are well-known transformations for optimizing closures based on outer context dependencies,

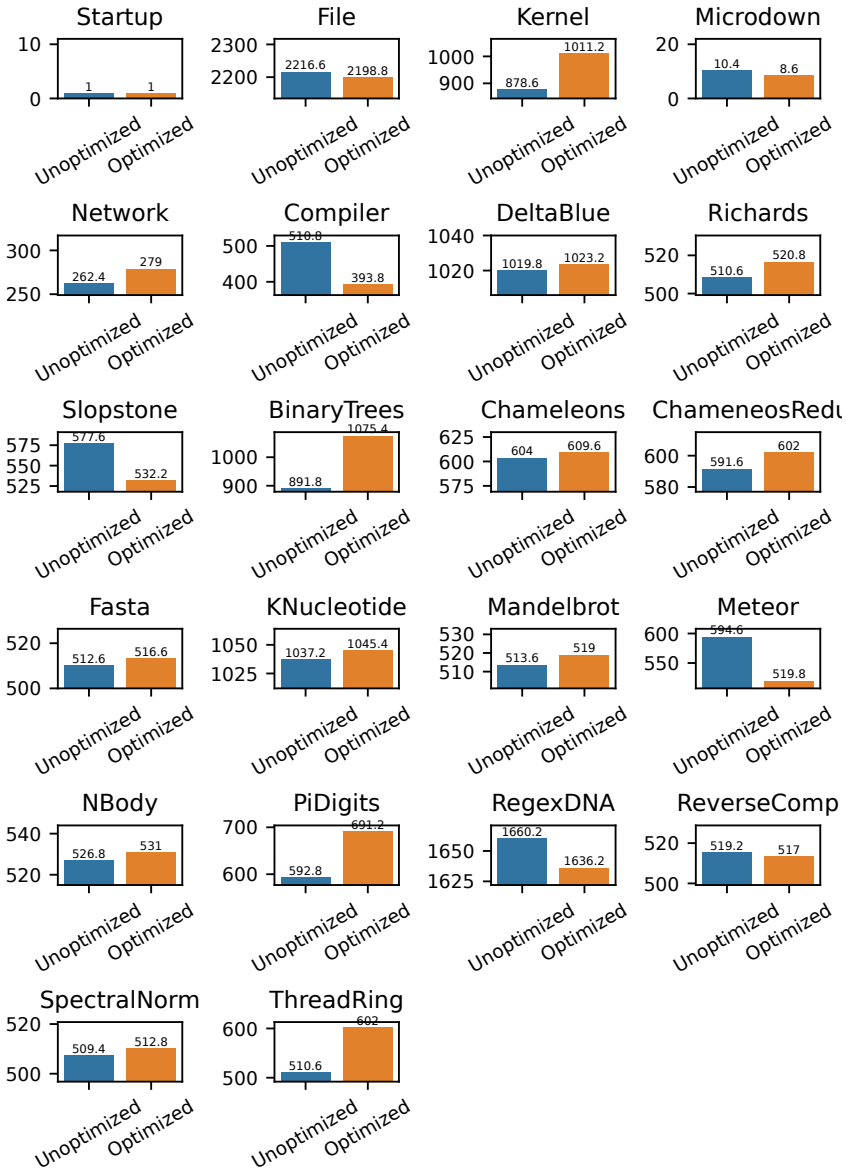


Figure 3: Garbage Collection overhead for each configuration per benchmark. Time is in milliseconds. Lower is better.

such as *lambda lifting*[5], transforming closures by turning free variables into explicit parameters, thereby reducing their dependence on outer contexts, and *flat-closure* [7], a technique used to optimize how variables from enclosing scopes are captured and accessed. Keep *et al.,* [8] presents a set of optimizations for closures creation and invocation, including free variables elimination and closure sharing. *Clean Blocks* are one of the optimizations presented (Case 2a).

Our optimization is not based on closure transformation, but on an escape analysis to determine if the block can be pre-allocated at compile time. However, we are interested to see how those optimizations work with *Clean Blocks*.

The Java Virtual Machine [9] implements *non-capturing (stateless) lambdas* for expressions that do not reference any variables from their enclosing scope. They are similar to *Clean Blocks*, but instead of creating the closures at compile time, they implemented a factory method that returns the same singleton object at runtime.

[5] https://en.wikipedia.org/wiki/Lambda_lifting

5. Future work

Future work is planned in two main directions. For one, we need to improve the analysis to find the reason for the GC impact. The second idea is to analyze more optimizations.

5.1. Improving the analysis

For now, we restricted the dynamic analysis to closure activation (that is, execution of a closure). But as clean blocks shift the creation from run-time to compile-time, we need to analyze how many closures are created during a benchmark run.

As already mentioned, we see Garbage Collection overhead that we cannot explain: clean blocks shift closure creation from runtime to compile time, yet we see GC overhead increase. We plan to analyse in detail the memory footprint of the benchmarks with the optimizations active to determine the causes of the overhead, *i.e.,* which part of the GC is spending more time, and understand the relations with our work.

5.2. More optimizations

The closure creation byte-code has a, right now not used, flag to create a full closure at runtime without an outer context. For execution, we only need an outer context if the closure has a non-local return. It will be interesting to analyze how many closures need to be compiled with an outer context and what the performance characteristics are.

A second idea is to experiment to see if we can avoid accessing outer variables directly by using the debugger infrastructure for specific patterns. This would allow us to turn non-clean blocks into clean blocks at the expense of the speed of variable access. This could be useful as we see one pattern often: a non-clean closure is handed over as a parameter for a failure case. The closure has to be created for every execution of the method, but then it is rarely executed. If it is executed, execution happens in all cases with the outer context still on the stack. This makes the variables thus accessible reflectively from the context, without the need for the block being compiled as a full closure.

References

[1] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983. URL: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[2] S. Ducasse, G. Rakic, S. Kaplar, Q. D. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, M. Denker, Pharo 9 by Example, Book on Demand – Keepers of the lighthouse, 2022. URL: http://books.pharo.org.

[3] C. Béra, E. Miranda, A bytecode set for adaptive optimizations, in: International Workshop on Smalltalk Technologies (IWST), 2014.

[4] C. Béra, M. Denker, Towards a flexible pharo compiler, in: International Workshop on Smalltalk Technologies 2013, 2013.

[5] S. Marr, Rebench: Execute and document benchmarks reproducibly, 2018. doi:10.5281/zenodo.1311762, version 1.0.

[6] G. J. Sussman, G. L. Steele, Scheme: An interpreter for extended lambda calculus, Higher-Order and Symbolic Computation 11 (1998) 405–439. doi:10.1023/A:1010035624696.

[7] M. A. Ertl, B. Paysan, Closures—the forth way, in: 34th EuroForth Conference, 2018, pp. 17–30.

[8] A. W. Keep, A. Hearn, R. K. Dybvig, Optimizing closures in o(0) time, in: Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 30–35. URL: https://doi.org/10.1145/2661103.2661106. doi:10.1145/2661103.2661106.

[9] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, The Java virtual machine specification, Pearson Education, 2014.

A. Benchmarks

Our set of classical **micro benchmarks**[6] is as follows:

BinaryTrees. An adaptation of *Hans Boehm's GCBench* for Garbage Collection.

Chameleons. Creates many threads to wait for mutex semaphores.

ChameneosRedux. Small problem size for *Chameleons* benchmark.

DeltaBlue. Classic object-oriented *constraint solver*.

Fasta. DNA sequence generation algorithm based on weighted random selection.

KNucleotide. Map the DNA letters into a hash table to accumulate count values.

Mandelbrot. Plot the Mandelbrot set on an N-by-N bitmap.

Meteor. Uses *ByteStrings* to solve a puzzle.

NBody. Classic n-body simulation of the solar system.

PiDigits. Generates and prints the first N digits of Pi.

RegexDNA. A simple regex pattern and actions to manipulate FASTA format data.

ReverseComplement. Computes the reverse complement of a DNA sequence.

Richards. Simulates a *task dispatcher* on a multitasking operating system.

Slopstones. *Smalltalk Low-level OPeration Stones*, a series of low-level operations.

SpectralNorm. Calculate the spectral norm of a N-by-N matrix.

ThreadRing. A token ring algorithm using threads and mutex semaphores.

In addition, we run some applications as **macro benchmarks**:

Compiler. Tests for source-to-bytecode Pharo compiler.

File Tests for file system access library.

Kernel Tests for basic language features.

Microdown. Tests for markup documentation library.

Startup. Just evaluate `1 + 1`, measure the time for starting up the system.

[6] Implementations are in https://github.com/guillep/SMark visited on 2024-02-04.