Static Basic Block Reordering Heuristics for Implicit Control Flow in Baseline JITs

G. Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
Lille, France
guillermo.polito@univ-lille.fr

S. Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
Lille, France
stephane.ducasse@inria.fr

P. Tesone
Inria, Pharo Consortium
Lille, France
pablo.tesone@inria.fr

ABSTRACT

Baseline JIT compilers in dynamically-typed languages often use techniques such as *static type predictions* to optimize common execution paths using static heuristics. Such compilations exhibit *implicit slow paths*, defined by the language implementation and not by a developer, representing uncommon execution paths *e.g.,* automatic type coercions, type validations and operation reifications. At run time *implicit slow paths* need to be jumped over and penalize overall execution.

Removing *implicit slow paths* from the main execution path requires code reordering techniques. However, such heuristics are generally designed to work with profiling information. Based on the insight that *implicit slow paths* are known at compile-time, and thus do not require runtime-profiles, we experimented with two different code reordering algorithms: Pettis-Hansen Bottom-Up ($O(n^3)$) augmented with static code layout heuristics, and a slow-to-end heuristic ($O(n)$).

Our results show that many micro-benchmarks improve their run time by 1.2x. Benchmarks governed by more expensive computations such as message sends or garbage collections show in general no visible performance improvement nor degradations, while very few cases show degradations of up to 1.2x. We show that such static heuristics have low performance impact at compile-time and have great potential when static type predictions are present in the JIT compiler.

1 INTRODUCTION

Efficient implementations of dynamically-typed languages use just-in-time (JIT) compilation techniques to avoid interpretation overhead and benefit from runtime type-feedback. In a multi-tier compiler, a baseline JIT is often the first tier generating sub-optimal code but very quickly: its focus is to avoid long compilation pauses. One common implementation of a baseline JITs uses method-based compilation as the granularity of the compilation units, because methods have well-established boundaries. A common technique to implement method baseline JITs combines in a linear fashion code templates following the order of the source code, typically bytecode.

In dynamically-typed languages, baseline JIT compilers often use techniques such as *static type predictions* to optimize common execution paths using heuristics [DS84, Hol94]. For example, languages such as Smalltalk in which operators (*e.g.,*+) are messages, those special messages are compiled by first trying integer arithmetic, and falling back to a slower message-send in case it is not possible (*e.g.,* because of type checks). Such code patterns imply that the compilation introduces *implicit slow paths* (See Section 2): execution paths that are defined by the language implementation and not by a developer. These implicit execution paths represent uncommon execution paths such as automatic type coercions, type validations and operation reifications. Linear translations as the ones done by a Baseline JIT mix developer-defined execution paths with *implicit slow paths* that need to be jumped over and penalize overall execution.

Holze informally reported in his thesis [Hol94] that having long pieces of code in those uncommon paths had negative performance impact, and replacing those uncommon paths by shorter calls to stubs improved overall performance. Further, removing completely those *implicit slow paths* from the main execution path requires code reordering techniques [PH90, CT03], with the ultimate goal of maximizing fallthrough branches (See Section 3). However, such heuristics are generally designed to work with profiling information. Newel and Pupyrev showed recently that they may impose suboptimal performance on instruction and I-TLB caches [NP20].

Based on the insight that *implicit slow paths* are known at compile-time, and thus do not require runtime-profiles, we experimented with two different code reordering algorithms: Pettis-Hansen augmented with static code layout heuristics, and a slow-to-end heuristic (See Section 4). Our results show that many micro-benchmarks improve their run time by 1.2x (See Section 5). Benchmarks governed by more expensive computations such as message sends or garbage collections show in general no visible performance improvement nor degradations, while very few cases show degradations of up to 1.2x. We show that such static heuristics have low performance impact and have great potential when static type predictions are present in the JIT compiler.

The contributions of this paper are:

- The identification of implicit control flow paths within static type predictions for baseline JITs.
- A static weighting heuristic for bottom up Pettis-Hansen code reordering that does not require any previous profiling information.
- A slow-to-end heuristic that based on static code heuristics runs linearly to the number of basic blocks.
- Empirical evidence that such reordering can be done fast and obtain performance gains of up to 1.2x.

2 IMPLICIT CONTROL FLOW

Dynamically-typed object-oriented languages such as Ruby, Javascript or Pharo [BDN+09] exhibit *implicit slow paths* because they support

high-level of polymorphism and because elementary operations can be applied on a large set of classes. Implicit slow paths are execution paths that are defined by the language implementation and not by a developer, and generally represent uncommon execution paths *e.g.,* automatic type coercions, type validations and operation reifications.

A case to illustrate implicit slow paths appears when using *static type predictions* [Hol94]. Static type predictions use language implementor type heuristics to optimize common execution paths. For example, languages such as Smalltalk define operators as messages, making it possible for developers to provide their own operator implementations. However, implementing all such operators as message sends over-penalises common arithmethic operations such as +. Using static type prediction, the language implementor defines that it is more common that + will refer to integer arithmetic than to a redefined + operator, thus it introduces a type check to try integer arithmethic when possible and eventually fall back to a slower message-send in case it is not possible.

It is important to note that this type prediction is just an optimization, and is not visible to the developer by other than by the perceived performance.

2.1 Running Example: Addition in Pharo

Addition in Pharo is a message like any other messages in the language. Here are two examples: the first one shows that adding a small integer with a large positive integer produces a large positive integer. The second one, an example taken from an AST interpreter, illustrates how we send the message + to AST node object instance of ExConstant using another (complex) node as argument. These examples show that the message plus can be sent to any instance with any argument.

```
(1 + 1000 factorial) class
>>> LargePositiveInteger

(ExConstant value: 1) +
  (ExMultiplication
    right: (ExConstant value: 11)
    left: (ExtConstant value: 2))
>>> anExContant (23)
```

The current implementation uses static type prediction for integer arithmetic, which works as follows:

- The implementation of + should check the types of the receiver and argument.
- If both are integers, an integer addition is tried out, and an overflow check takes place.
- When either the receiver or arguments are not small integers or an overflow is encountered, a much slower path is taken: a message is sent.

The usefulness of the message-send is two-fold. First, it allows developers to define their own operators with their own domain semantics. Second, in case an overflow happens, it lets the standard library promote the numbers to large integers before retrying a slower large integer addition. While the bytecode interpreter is not the focus of this article, the following excerpt of the interpreter mimics the JIT compiler behaviour, clearly illustrating the situation.

```
bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (objectMemory
      integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  messageSelector := self specialSelector: 0.
  argumentCount := 1.
  self normalSend
```

Not only the case of arithmetic. In Pharo, similar situations occur with other arithmetic operations, but also control flow operations such as conditionals and loops. For example, in Pharo the message ifTrue: expects a boolean receiver. When the receiver is not of the expected type, the VM activates in-place a mustBeBoolean callback similarly as with doesNotUnderstand:. This gives developers the power to provide their own boolean coercions.

All these cases share a common design: they check first if a fast path can be taken and do it if so, otherwise they take a slower path. This design is based on the static assumption that the slow case is uncommon, and it allows efficient bytecode interpreters and baseline JITs for common cases.

2.2 Baseline JIT Implicit Slow Paths

In a multi-tier compiler, a baseline JIT is often the first tier generating sub-optimal code but very quickly: its focus is to avoid long compilation pauses. One common implementation technique for such JITs is to combine machine code templates in a linear fashion following the order of the source code, typically bytecode. In other words, when such baseline JIT generates code for a sequence of bytecodes, the generated machine code follows the same order.

Consider for example a sequence of stack-based bytecodes for the statement return a + b + c: three values are pushed to the stack and the message + is sent twice, and finally the stack top is returned. From both the source-code and the bytecode point of view, the execution is linear and no control flow happens whatsoever (other than a message send that yields the control to another method).

```
b1: push a
b2: push b
b3: send #+
b4: push c
b5: send #+
b6: return top
```

Listing 1: Example of bytecode sequence without explicit control flow but with implicit control flow

On the other hand, when that bytecode is compiled to machine code, the baseline JIT compiler adds additional control flow to handle the cases explained above: type and overflow checks. This control flow is **implicit**: it is defined by the language implementation and not by a developer. Listing 2 below illustrates the intermediate representation (IR) used by our linear JIT compiler to compile the above bytecode sequence. It generates machine code in the same order as in the source bytecode, yielding a code layout similar as the one illustrated below.

We see in the listing that each bytecode generates zero or more IR instructions. We mark in green those instructions in the assumed common fast path, and in red those in the uncommon slow paths. Implicit slow paths in linear translations need to be jumped over, generating *slopes* and penalizing the overall execution.

```
1     ... # previous bytecode IR
2   b3        check a SmallInteger
3   b3        jumpzero notsmi
4   b3        check b SmallInteger
5   b3        jumpzero notsmi
6   b3        t1 := a + b
7   b3        jumpIfNotOverflow continue
8   b3  notsmi: #slow case first send
9   b3        t1 := send #+ a b
10  b3  continue:
11  b4        c := load [c−address]
12  b5        check t1 SmallInteger
13  b5        jumpzero notsmi2
14  b5        check c SmallInteger
15  b5        jumpzero notsmi2
16  b5        t1 := t1 + c
17  b5        jumpIfNotOverflow continue2
18  b5  notsmi2: #slow case second send
19  b5        t1 := send #+ t1 c
20  b5  continue2:
21  b6        return t1
```

Listing 2: Linear translation of the Pharo baseline JIT compiler of the example in Listing 1

2.3 Insights for Static Heuristics

Removing implicit slow paths from the main execution path requires code layout optimization techniques [CT03, PH90, CG99]. However, code layout optimizations are generally designed to work with profiling information, which are often not yet available to the first-tier baseline JIT compilers. Nevertheless, such techniques are applicable to *implicit slow paths* based on the following insight:

> *Implicit slow paths* are known at compile-time and thus do not require runtime-profiles.

In the rest of this paper we report on our experiments with two different code reordering algorithms: Pettis-Hansen augmented with static code layout heuristic, and a slow-to-end heuristic.

2.4 Static Heuristics Required Properties

Regardless of the algorithm used for code layout optimization, we would like to guarantee that its output satisfies the following properties:

Distinguish explicit from implicit control flow edges. The control-flow graph of a method contains both implicit edges defined by the language implementation, and explicit edges defined by the application developer. The code reordering algorithm should be able to discern between them to remove the slow cases from the middle of the fast paths.

Prioritize equally all explicit control flow edges. One explicit edge in the control flow should not be penalized (nor rewarded) over some other explicit edge, because we cannot statically decide without proper profiling information which one is going to be taken at run time. Because of this, an ideal code layout should reflect a reverse post-order of the control flow for all explicit edges. A reverse post-order guarantees that linear control flow stays linear, while conditionals have both their branches close to each other.

3 PETTIS-HANSEN CODE POSITIONING

Pettis-Hansen [PH90] (PH from now on) is a profile guided code positioning technique that can be used both for inter and intra procedural code reordering. Inter-procedural code positioning deals with the order of functions/methods in native code, and is outside of the scope of this paper. Intra-procedural code positioning, on the other hand, deals with the order of basic blocks within a function/method.

Code layout optimization is an NP problem, and PH implements two heuristics that have shown to produce very good results with profile information, and that can be implemented very efficiently using a fixed-point algorithm [CT03]: a greedy heuristic also known as top-down PH, and a bottom-up heuristic that builds hot chains of basic blocks before deciding the final code layout. In the rest of this paper we center our study of PH on the bottom-up heuristic.

3.1 Bottom-Up PH in a Nutshell

Intra-procedural bottom-up PH works as follows. Each edge in the control-flow graph is assigned a weight representing how likely that edge is going to be taken. This weight is based on profiling information either obtained offline during previous executions, or online during previous phases of the current execution.

The first step of the algorithm is to detect *hot chains i.e.,* chains of basic blocks that are very likely execute one after the other. For this, each basic block is assigned first a chain for itself and then all edges are iterated in likeliness order (from more likely to less likely taken). For each edge (x, y) connecting basic blocks x and y, we take the chains corresponding to its vertices (say A and B respectively) and merge those chains iff $x = tail(A)$ and $y = head(B)$. Each time two chains are merged, a new chain is created, both edge vertices are updated with that new chain, and that chain has its priority incremented. The priority of a chain shows the number of merges it has seen and represents to some extent the *hotness* of the chain.

Once all edges have been traversed and their chains merged, the algorithm proceeds to merge all hot chains by priority order and builds the final code layout. Figure 1 shows a control-flow

graph with edges weighted according to profiling information. This corresponds to an exact profiling, where the sum of the weights of the in-edges of each basic block is equals to the sum of the weights of the out-edges of each block. In the left of the figure there is the weighted control flow, and the dashed areas represent the final chains. In the right there is the final code layout and the jumps that remain.

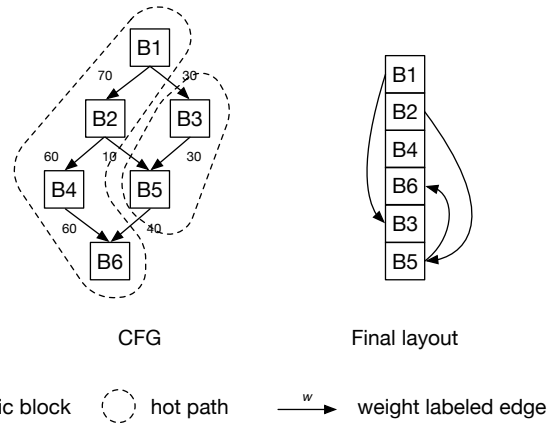

Figure 1: Example of Pettis-Hansen with Exact Profiling Information. The hot paths created by the algorithm are shown in a dashed line circle. At the right we depict the final linear layout. We observe that the hot-paths create traces connecting the highest weights. The edges in the hot-paths become fall through branches.

3.2 Challenges of Static Weighting

Using Pettis-Hansen without profiling information means to statically assign weights to the edges in the control-flow graph. However, naïvely assigning such weights could not only generate low benefits, but also be counter-productive and generate inefficient code layouts. For the sake of completeness, we state in this section the first failing weighting heuristic we tried: *unprioritized slow-paths*. This heuristic does not work properly because it satisfies only our first required property but not the second one. This means that slow paths are indeed taken out of the main execution path, but some explicit paths that are not statically decidable are penalized.

Unprioritized slow-paths weights edges to slow-paths statically with the lowest weight (say 0), and weights all other edges with a higher default weight (say 100). Figure 2 shows the same control-flow graph as before with edges weighted according to this first unrefined static heuristic. Bottom-up PH relies on how the edges are sorted, and since all edges have similar weights, the algorithm will not properly resolve ambiguities: as we can observe in the example, if we process the edge $(B2, B4)$ before $(B2, B5)$, then the blocks $B2$ and $B4$ will be merged in a single chain $\{B2, B4\}$, preventing $B5$ to be merged in that chain because $B2$ is not the tail of it's chain anymore.

Moreover, since $B5$'s chain is not merged with any other block (or merged with $B3$ very late), its chain will have low priority and be put at the end of the final layout, penalizing the execution of $B5$ which needs a far jump to the end of the method and a jump back to the *so-computed hot* control flow.

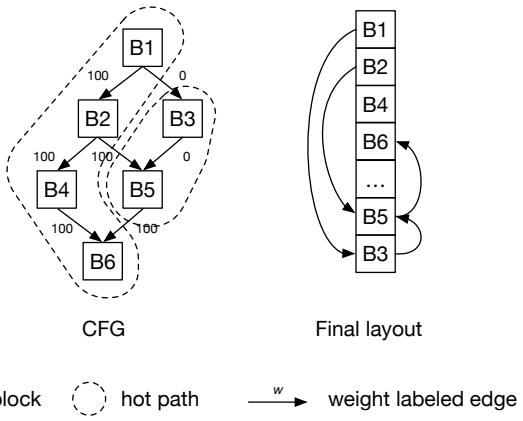

Figure 2: Example of Pettis-Hansen with Naive Edge Weighting when (B2, B4) is processed before (B2, B5). The hot paths created by the algorithm are shown in a dashed line circle. On the right we depict the final linear layout. This weighting does not differentiate between explicit paths. We observe that the hot-paths create traces connecting the highest weights. The edges in the hot-paths become fall through branches.

4 REORDERING TECHNIQUES ON A STATIC BASIS

In this section we present two other code reordering heuristics that satisfy our two stated properties but have different compile-time and implementation costs: a refined static weighting heuristic for Bottom-Up PH, and a slow-to-end heuristic that satisfies the previously stated properties for static reordering and yields good results. While Bottom-Up PH has a run time of $O(n^3)$, slow-to-end runs on $O(n)$ of the number of basic blocks, making it a low-overhead candidate for baseline JIT compilers.

4.1 Refined Static PH Weighting for Implicit Control Flow

We designed our weighting heuristic as follows:

- Edges to slow paths get the lowest weight (say 0).
- In conditional branches, the first branch has the highest weight (say 100), the second branch has the second highest weight (say 90).
- In merge points, the edge coming from the first branch has the second highest weight (say 90), the edge coming from the second branch has the highest weight (say 100).
- All other edges have the highest weight (say 100).

Although the associations of weights we chose may seem odd, they produce layouts that respect to some extent the properties stated above. First, edges to slow paths weighting 0 are not included in a hot chain unless they are the only successor of another basic

block. In our case that never happens because slow paths have always a single predecessor. This in turns means that they will be positioned at the end of the code layout. Second, the heuristic governing conditional branches and merge points ensures that branches are scheduled close to each other. The key objective of this heuristic is to avoid one of the branches to be isolated. This way one of the branches will be included in the hot path entering the conditional, while the other branch will be included in the hot path that exits the conditional. Figure 3 shows how hot chains are created using this heuristic and without it, and the final code layouts for both cases.

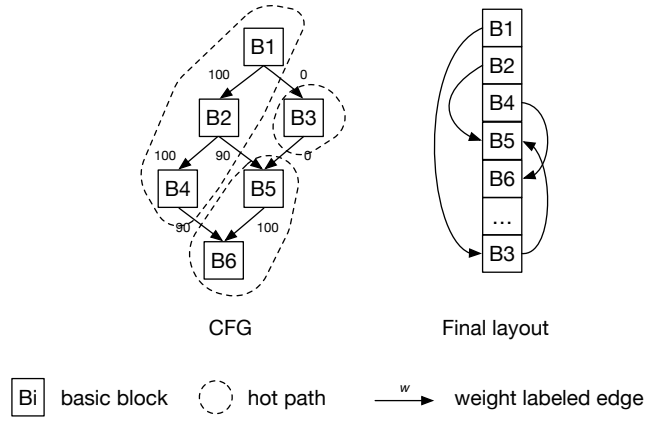

Figure 3: Refined Static Weighting Heuristics for Bottom-Up PH. The figure shows the same control flow graph as in Figure 2 but using different weight assignments. The hot paths created by the algorithm are shown in a dashed line circle. On the right, we depict the final linear layout. Our heuristic for conditional branches makes the final layout look like a reverse post-order excepting slow paths.

4.2 Slow-to-End Heuristic

As we stated before, an ideal code layout should reflect a reverse post-order of the control flow for all explicit edges. A reverse post-order guarantees that linear control flow stays linear, while conditionals have both their branches close to each other. In this section we propose another code layout optimization algorithm we call *slow-to-end*, that takes advantage of another insight: our source bytecode is already linearised in reverse post-order.

The key of the *Slow-to-End* code layout optimization is to identify basic blocks that belong to a slow path and push them to the end of the code layout. *Slow-to-End* takes as input a list of basic blocks in reverse post-order and does a single iteration on it. For each block, if the block is marked as a slow path it is removed from its current position and appended at the end of the list. Algorithm 1 illustrates the basics of Slow-to-End. Figure 4 illustrates how *Slow-to-End* works in a control flow graph already linearised using reverse post-order.

Algorithm 1: Algorithm

```
Data: A list of basic blocks linearised in reverse post-order
Result: A list of basic blocks linearised in reverse post-order
        with slow basic blocks at the end
for each block in the original list do
    if current block is marked as slow then
        remove current block from the list;
        append current block at the end of the list;
    end
end
```

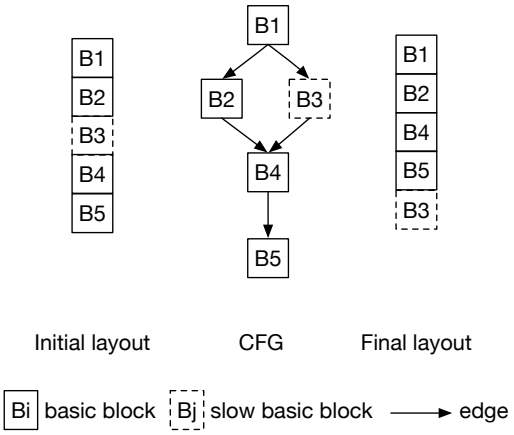

Figure 4: Slow-to-End example. The figure shows on the left the initial code layout with basic blocks on the slow path marked in dashed lines. On the center, the corresponding control flow graph. On the right, the final code layout after using *Slow-to-End*.

5 EVALUATION

We evaluate the code layout optimizations explained above using benchmarks on top of Pharo [BDN+09]. We chose four different sets of benchmarks to run, each category containing many benchmarks: (1) a set of micro benchmarks we designed to observe the impact of code layout, (2) a set of generic Pharo micro benchmarks, (3) an implementation of the Computer Language Benchmarks Game, and (4) a set of larger benchmark programs: a bytecode compiler, deltablue and richards. The first set of benchmarks were written by ourselves with this article in mind. The latter three sets of benchmarks are available as part of the SMark benchmarking library[1]. Since our prototype is not complete (e.g., we miss support for exceptions. See Section 6 for more details), some of the available benchmarks did not finish successfully on all our prototype implementations and were removed from the evaluation.

This section evaluates different characteristics of such benchmarks. First we present a profile of the benchmarks showing how often implicit control flow paths are reached during execution, and how often they are taken. Such profile gives an idea of the potential of these optimizations for each benchmark: the more implicit paths

[1] https://github.com/guillep/SMark

Category	Benchmark	# Method Activations	#Bytecodes	#Implicit Jumps Reached	#Implicit Jumps Taken
CodeLayout	False Conditionals	3	669	103 (16%)	0 (0% of total - 0% of reached)
CodeLayout	False Conditionals Compensated	3	669	103 (16%)	0 (0% of total - 0% of reached)
CodeLayout	False Fallthroughs	3	89	13 (15%)	0 (0% of total - 0% of reached)
CodeLayout	True Conditionals	3	129	13 (11%)	0 (0% of total - 0% of reached)
CodeLayout	True Conditionals Compensated	3	129	13 (11%)	0 (0% of total - 0% of reached)
CodeLayout	True Fallthroughs	3	269	103 (39%)	0 (0% of total - 0% of reached)
CodeLayout	SmallFactorial	4	263	37 (15%)	0 (0% of total - 0% of reached)
CodeLayout	LargeFactorial	128	2666	288 (11%)	0 (0% of total - 0% of reached)
CodeLayout	VeryLargeFactorial	1478	28766	2988 (11%)	0 (0% of total - 0% of reached)
Category	**Benchmark**	**#activations**	**#bytecodes**	**#Implicit Jumps Reached**	**#Implicit Jumps Taken**
Micro	ArrayAccess	3	87	3 (4%)	0 (0% of total - 0% of reached)
Micro	ClassVarBinding	2	63	3 (5%)	0 (0% of total - 0% of reached)
Micro	FloatLoop	2	51	1 (2%)	0 (0% of total - 0% of reached)
Micro	InstVarAccess	2	63	3 (5%)	0 (0% of total - 0% of reached)
Micro	IntLoop	2	60	4 (7%)	0 (0% of total - 0% of reached)
Micro	Send	2	63	3 (5%)	0 (0% of total - 0% of reached)
Micro	SendWithManyArguments	3	77	3 (4%)	0 (0% of total - 0% of reached)
Micro	Stone	41	1996	293 (15%)	0 (0% of total - 0% of reached)
Category	**Benchmark**	**#activations**	**#bytecodes**	**#Implicit Jumps Reached**	**#Implicit Jumps Taken**
GameSuite	Chameleons	473	6387	296 (5%)	4 (1% of total - 2% of reached)
GameSuite	FannkuchRedux	60	975	89 (10%)	0 (0% of total - 0% of reached)
GameSuite	Mandelbrot	108	1844	162 (9%)	3 (1% of total - 2% of reached)
GameSuite	Meteor	11359424	480056856	73380268 (16%)	4774 (1% of total - 1% of reached)
GameSuite	PiDigits	13	236	16 (7%)	0 (0% of total - 0% of reached)
GameSuite	RegexDNA	21920	355031	29100 (9%)	4286 (2% of total - 15% of reached)
GameSuite	ReverseComplement	693	12775	867 (7%)	11 (1% of total - 2% of reached)
GameSuite	SpectralNorm	108	4178	511 (13%)	0 (0% of total - 0% of reached)
GameSuite	ThreadRing	30904	322009	12735 (4%)	0 (0% of total - 0% of reached)
Category	**Benchmark**	**#activations**	**#bytecodes**	**#Implicit Jumps Reached**	**#Implicit Jumps Taken**
Program	Compiler	41921	657853	47278 (8%)	3403 (1% of total - 8% of reached)
Program	DeltaBlue	2923	49445	3023 (7%)	61 (1% of total - 3% of reached)
Program	Richards	1122372	14571071	878209 (7%)	0 (0% of total - 0% of reached)

Table 1: Dynamic characterisation of the benchmarks. Executed bytecode and implicit jump (found and taken).

present in the code, the more effect these optimizations will have. Second we report the run time of each benchmark, taking as comparison baseline the default Virtual Machine with a compiler doing no code layout optimization. Finally, we characterize how the full run time of the benchmarks is divided between the execution of actual work, compilation time, and garbage collection time.

5.1 Benchmark Characterisation

The Pharo baseline JIT compiler produces *implicit* edges in three cases: (1) integer comparison (<, >,...) (2) integer arithmetic (+, -, bitAnd, bitOr...) and (3) conditional jumps (mustBeBoolean check). To characterize the behavior of *implicit* edges at runtime, we estimated the number of times such implicit edges are reached and taken per execution, as shown in Table 1. Such profile gives an idea of the potential of these optimizations for each benchmark: the more implicit paths present in the code, the more effect these optimizations will have.

We estimated such numbers by running each benchmark for a single iteration with a problem size of 1 using an instrumented bytecode interpreter. This makes an over-estimation because while the JIT compiler performs some straight-forward form of constant folding for the cases explained above, the bytecode interpreter does not do such optimizations. Still we are positive that such estimation is close to the JIT behavior because the compiler does not perform any kind of constant propagation, meaning that constant folding happens only in trivial cases.

Notice also that we only instrumented in the interpreter those bytecodes that present implicit control flow when compiled to machine code. Indeed, other bytecodes in the interpreter present implicit control flow cases (e.g., the multiplication message) but since those are not present in the compiler, we did not take them into account in our measurements.

Table 1 shows the dynamic characterization of the implicit jumps per benchmark, showing the total number of *executed* bytecodes, the total number of *reached* implicit jumps and the total number of *taken* implicit jumps. All these numbers are in absolute terms. In addition, we report total number of *reached* implicit jumps as a percentage of the total number of executed bytecodes, and the

total number of taken implicit jumps as a percentage of the total of bytecode and the total of reached implicit jumps.

From the table we observe:

- Having 0 in the last column (implicit Jumps Taken) is showing a good behavior of the benchmarks – The benchmarks do not execute a slow path (large integers or must be booleans for conditionals).
- By design our specific micro benchmarks shows higher number of implicit jumps. This is normal since this benchmark has been designed to stress the code reordering. Note that a high percentage of implicit jumps cannot be reached because of the bytecode distribution. Implicit jumps are a minority compared to pop, push and sends. The bytecode is also not optimized to support debugging and stepping at the level of expression and subexpressions (in contrast with line-based or statement-based).
- The game suite and the benchmark program largely vary in size. They show from 4% up to 16% of implicit jumps. This represents the potential target of the fall through maximization.

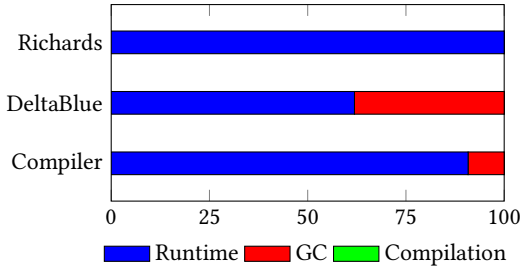

Figure 5: Runtime Profile of Program Benchmarks on Baseline VM. Time spent in GC and compilation vs actual runtime

5.2 Benchmark Methodology

We run all our benchmarks in three different setups; the stock baseline JIT compiler with no code reordering nor basic block support, a JIT compiler with a static PH heuristics, and a JIT compiler with a Slow-to-End heuristics. The stock baseline JIT was used as the baseline for comparison. We run our benchmarks on top of an Intel(R) Core(TM) i5-5287U CPU @ 2.90GHz OSX Mojave, with all applications closed, most services turned off and no internet connection plugged to minimise noises. Figures 6 through 9 show the results of our measurements. We based our performance methodology on [GBE07], with some modifications. We are aware of recent work on benchmarking methodologies [BBTK+17], but the engineering effort to put it in place did not make it in time at the moment of writing this article.

100 iterations per VM invocation. We performed 100 iterations per VM invocation to increment our confidence in the measurements.

Warmup takes two iterations only. We discard the two first iterations of all benchmarks. Since our virtual machine does not have an adaptive optimizing compiler, compilation is deterministic, and thus our *warmup phase* is fast: we need to wait until all methods are compiled. In the current implementation, the first execution of a method runs interpreted, the second one does generally set off compilation and executes compiled code. The main exception are hot interpreted loops, which are compiled when a certain threshold is reached and an on-stack replacement mechanism takes place. In our prototype VMs implementing the two said code reordering heuristics, we do not support yet on-stack replacement.

Single VM invocation. Again, since our virtual machine JIT compiler is deterministic and our code cache is big enough to hold the entire working set of methods in the benchmarks (1.4MB), we did not do several VM invocations.

Tailored Problem Sizes. We defined problem sizes for each benchmark that maximize the observability of the measurements. We did our best effort to calibrate those problem sizes so that each iteration is around 300 ms. This was not possible for some benchmarks that seem to have exponential behavior and incrementing slightly the problem size made the run time blow up.

5.3 Performance Analysis

Our benchmarks show that both heuristics present different behavior in different benchmarks. We measured the time spent in GC during the benchmarks. Only two of the benchmarks (deltaBlue and Compiler See Figure 5) did actually perform GCs.

On the one hand, the PH refined static weight heuristic shows gains of up to 1.2x in many benchmarks and losses of up to 1.2x in many others when compared to our baseline VM. After some analysis, we found that the degradations could be explained because our weighting heuristic applies to the bytecode-to-machine code translation but not to the native methods that are manually written in the intermediate representation (a.k.a. primitives). This means that handwritten IR routines are subject to the more naïve weighting explained in Section 3.2, and they need to be manually revised to mark potential slow paths in them.

On the other hand, Slow-to-End shows results that overlap with the baseline's in many benchmarks, meaning that those benchmarks are not sensitive to the code layout optimizations. Micro benchmarks where implicit slow paths are very present (e.g., tight loops with arithmetic) show speed-ups of up to 1.2x. Few benchmarks show degraded performance with Slow-to-End heuristic (e.g., SlopeStone, Richards). We want to characterize better such behavior in the future. Our hypothesis is that since our compiler does not do inlining, the cost of frame creation coupled with the general small code size of methods in Pharo dilute the gain of reordering. Zaitsev reports that methods in Pharo are 6.3 lines in average with a median of 3 lines [ZDA20]. We are at the moment analyzing why such degradation takes place, since our intuition is that Slow-to-End should not degrade performance unless implicit paths are taken very frequently.

Our observations suggest that structured code where loops and arithmetic are omni-present present high optimization potential by both heuristics. However, implementing Pettis-Hansen would require modifying all handwritten code to properly adjust edges, while in Slow-to-End the original order is respected. On the other

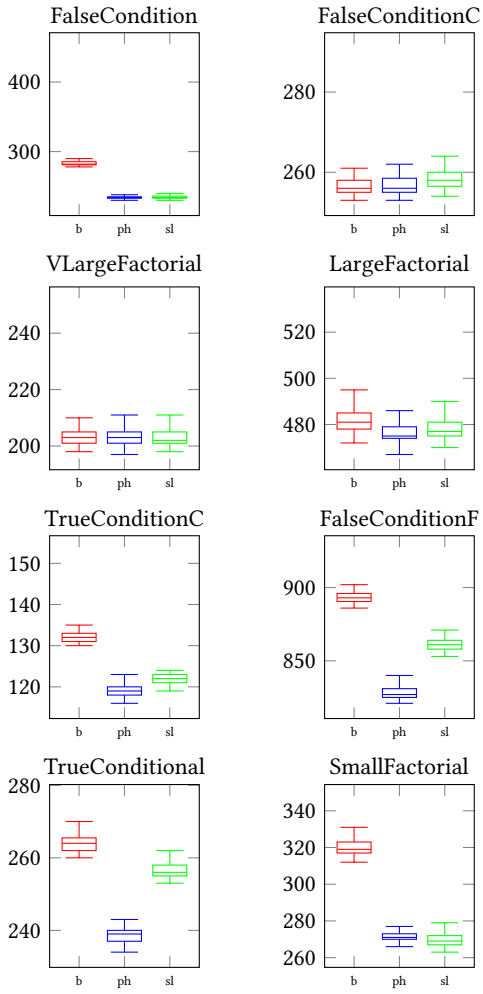

Figure 6: Micro Benchmarks in milliseconds (b=Baseline, p=Pettis-Hansen, sl=Slow-to-End). Lower is better.

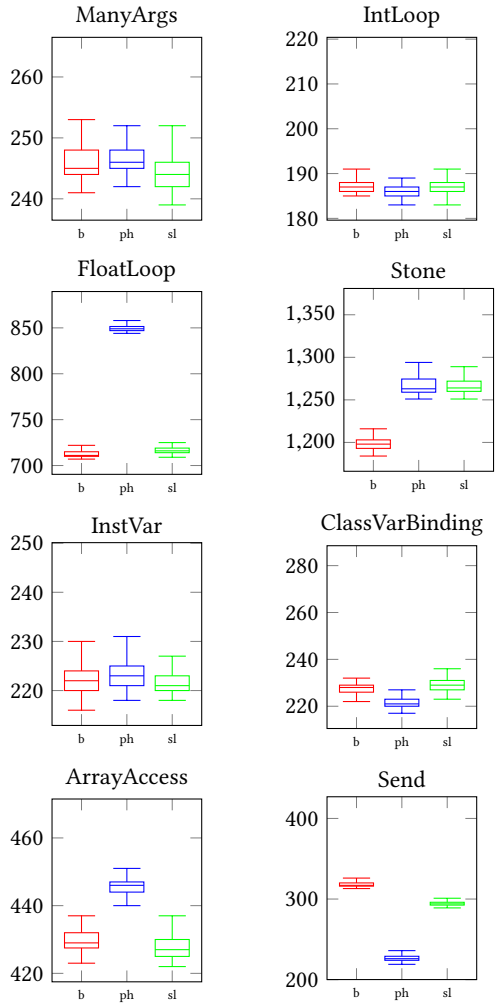

Figure 7: Generic Micro Benchmarks in milliseconds (b=Baseline, p=Pettis-Hansen, sl=Slow-to-End). Lower is better.

hand, highly object-oriented code is not very sensitive to such optimizations and their gain is diluted in the cost of message sends.

5.4 Instruction Cache Misses

For completeness, we report instruction cache misses for the SmallFactorial and Mandelbrot benchmarks in Table 2. We extracted this information using OSX's *Instruments* profiler. From the table we observe that the speed ups/downs observed in our benchmarks are correlated with corresponding decreases/increases in the CPU cache misses.

We chose to present in this section SmallFactorial and Mandelbrot because they exhibit different behaviour: the former shows performance improvements with our heuristics with respect to the baseline compiler; the latter shows that Slow-to-End has comparable performance than the baseline while Pettis-Hansen sees a noticeable degradation. Performance effects seem to be indeed related to the cache behaviour.

Benchmark	I-Cache Misses
SmallFactorial (b)	170.6M (1x)
SmallFactorial (ph)	151.3M (0.89x)
SmallFactorial (sl)	153.0M (0.9x)
Mandelbrot (b)	168.8M (1x)
Mandelbrot (ph)	218.8M (1.3x)
Mandelbrot (sl)	172.4M (1.02x)

Table 2: Instruction Cache Misses for the SmallFactorial and Mandelbrot benchmarks. Numbers represent millions of misses. Between parentheses the relative ration with respect to the baseline. (b=Baseline, p=Pettis-Hansen, sl=Slow-to-End)

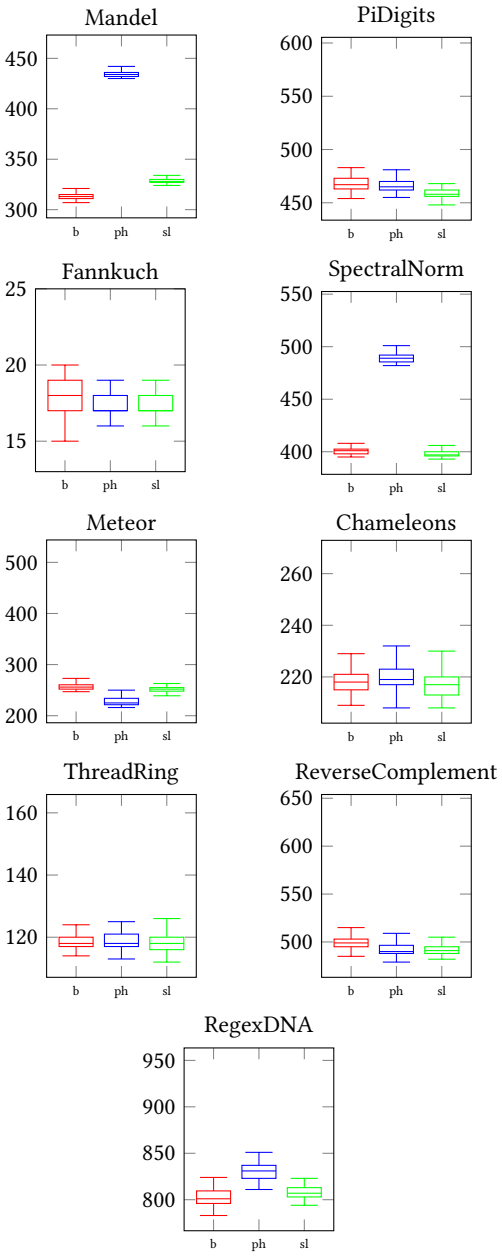

Figure 8: Game Suite Benchmarks in milliseconds (b=Baseline, p=Pettis-Hansen, sl=Slow-to-End). Lower is better.

5.5 Compile Time Analysis

Along with the run time of benchmarks, we measured also the number of methods compiled and the total time spent in compilation. Most of the benchmarks present working sets of ~200 methods, and total compilation time for all those methods fluctuates between 1 and 4 milliseconds, in all JIT compilers. Such compile times seems an acceptable in comparison with the associated run time, and make it difficult to assess associated overheads.

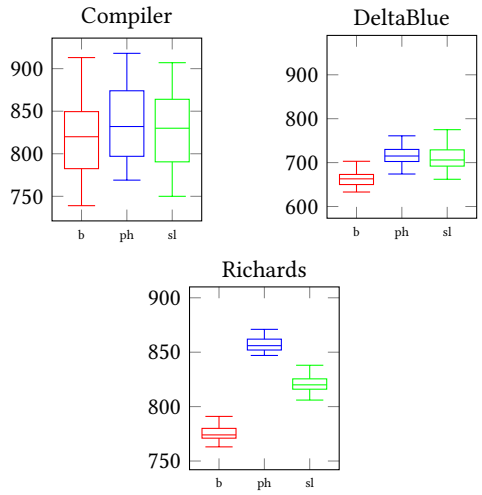

Figure 9: Program Benchmarks in milliseconds (b=Baseline, p=Pettis-Hansen, sl=Slow-to-End). Lower is better.

The more observable compile times in our measurements are shown by the compiler benchmarks which exhibits a working set of 1271 methods compiled. Our measurements report that the total compilation time of all those methods using our baseline compiler adds up to 8ms, while Slow-to-End is 17ms and Pettis-Hansen 24ms.

6 IMPLEMENTATION

We implemented our code reordering heuristics on top the Cogit JIT compiler of the Pharo VM. The Cogit JIT compiler is a linear JIT compiler that does not explicitly model a control flow graph (CFG). The JIT has a separate code zone [HBGM06] and a generational garbage collector. In this section we explain how we extended the compiler to build a CFG during the bytecode parsing phase, and how we extended the building of the intermediate representation (IR) to mark slow paths.

6.1 Extending the Cogit Compiler with a CFG

The Cogit JIT compiler is a linear JIT compiler that includes three main phases: a *bytecode scan phase* iterates the bytecodes of a method to extract meta-data from them, a *bytecode parsing* does an abstract interpretation of the bytecodes performing a stack to register IR transformation and a *code generation phase* computes IR instruction offsets and outputs the final machine code. Moreover, the Cogit JIT compiler does not explicitly model a control flow graph (CFG): conditional jump instructions of the IR have a single jump target and fallthrough edges are implicit.

We extended the compiler with a CFG by doing four main changes: (1) identify basic block leaders during the bytecode scan phase, (2) create basic blocks when bytecode parsing hits a leader instruction or when jump instructions are created, (3) turn fallthrough edges into explicit jumps on conditional jumps and (4) remove redundant fallthrough jumps after code reordering. With these changes, fallthrough edges are turned to explicit jumps in (3) and code reordering can be applied safely. Otherwise if a fallthrough edge moves it would fall through a different basic block. After code

reordering, jump instructions representing fallthrough edges are removed as a peephole optimization.

6.2 Marking Slow Paths

Pettis-Hansen and Slow-to-End have different requirements on how to mark slow paths in the CFG. On the one hand Pettis-Hansen requires annotating edges in the CFG, on the other hand Slow-to-End requires annotating uncommon basic blocks.

In both cases, we took advantage of the compiler's IR building schema. The Cogit compiler uses an internal DSL to express instructions in an at&t-like two-address-code fashion. The DSL works as an instruction builder creating IR instructions behind the scenes.

To mark slow paths we extended the IR building DSL:

Pettis-Hansen. We introduced new building methods to the DSL to create jump instructions with an explicit weight. We extended existing jump instructions to have a default weight. In the case of instructions that generally mark slow paths (*e.g.*, overflow checks), the default weight is the lowest weight.

Slow-to-End. We hooked into those places that mark the beginning of slow basic blocks and marked those blocks as slow.

7 RELATED WORK

Code Layout optimization Algorithms Pettis-Hansen [PH90] proposed more than three decades ago two heuristics for code layout optimization, now known as Pettis-Hansen top-down and Pettis-Hansen bottom-up, and usable for inter-procedural and intra-procedural optimization. The goal of these intra-procedural optimizations is to maximize fallthrough branches. Pettis-Hansen bottom-up has been presented in this paper in Section 3. These code layout optimization algorithms are profile-guided optimizations (PGO): they use profiling information, taken either offline or online, to assign weights to the call graph edges.

Newel and Pupyrev show that incorrect weighting may impose suboptimal performance on instruction and I-TLB caches [NP20]. Newell applies machine learning to discover the cache behavior and select optimal code layouts.

Huang et al. [HLM06] developed efficient algorithms for code layout that run up to 6000 times faster than the popular Pettis-Hansen algorithm. However, they require an expensive instrumentation to gather their profile data.

Offline PGO Intra-procedural code layout optimization. The HipHop Virtual Machine (HHVM) uses offline profiling to perform code layout optimization [OM17, OL21]. They have used offline profiling information to drive the reordering of the VM code that is compiled ahead of time. Recently they have taken advantage of their deployment stages to take representative profiles while testing and accelerate their warmup times. In [OL21], Ottoni et al. present JumpStart that improves VM warmup and steady-state performance by sharing VM profile data gathered during the early phase of new Facebook website deployment.

Ottoni et al. [OM17] study the impact of function placement. By using sample-based profiling, this methodology follows the same principle behind AutoFDO, i.e. using profiling data collected from unmodified binaries running in production, which makes it applicable to large-scale binaries. They first evaluate the impact of the traditional Pettis-Hansen function-placement algorithm on a set of widely deployed data-center applications. They show an average improvement of 2.6%. In addition they present new algorithm, called C3. C3 places a function as close as possible to its most common caller, and we do so following a priority from the hottest to the coldest functions in the program.

Online PGO code layout optimization. For more than two decades now, such techniques have been adapted to run-time optimizations [Arn02, ATDM03, SOT$^+$00].

Chen and Leupen applied dynamic code layout techniques to improve procedure placements based on their invocation order at run-time [CL97]. Scales proposes to use run-time information to dynamically reorder procedures [Sca98]. Scale's system instruments procedure calls and copies procedures to new locations dynamically, which incurs in a high run-time overhead.

Huang et al. [HBGM06] present a dynamic code reordering for the Jikes optimizing JIT compiler that takes online branching statistics to guide different code layout optimizations such as intra-procedural code splitting.

Missing and inaccurate profiles. Levin et al. present low overhead sampling techniques to handle the lack of profiling information. They propose an approach based on the minimal cost circulation problem [LNH08].

This work deals with our same challenge: doing code layout when profiling information is not present.

Code cache optimization. At the level of a virtual machine and its JIT, Huang et al. [HBGM06] explore the impact on separating generated code from heap objects. They develop a dynamic code reordering system using online information to improve instruction locality. DCR has three optimizations: (1) interprocedural method separation; (2) intraprocedural code splitting; and (3) code padding. DCR uses the dynamic call graph and an edge profile that most VMs already collect to separate hot/cold methods and hot/cold code within a method. It also puts padding between methods to minimize conflict misses between frequent caller/callee pairs. Extensive simulation and run-time experiments show that a simple code space improves average performance on a Pentium 4 by around 6% on SPEC and DaCapo Java benchmarks. These programs however have very small instruction cache footprints that limit opportunities for DCR to improve performance. Consequently, DCR optimizations on average show little effect, sometimes degrading performance and occasionally improving performance by up to 5%. Our JIT compiler already separated code space from other, therefore such element does not influence our reordering analysis. In addition, our JIT compiler does not split hot/cold methods and hot/cold code within a method (such optimizations does not show real impact).

Branch alignment. Several works extended the basic block reordering algorithms of Pettis Hansen to add information about cache padding. Such algorithms are often called branch alignment optimizations: Calder and Grunwald [CG99] extend Pettis Hansen's approach to basic block reordering and propose an improved branch alignment algorithm that takes into consideration the architectural cost model and the branch prediction architecture when performing

the basic block reordering. Torrellas et al. [TXD98] characterize the locality patterns of the operating system code and shows that there is substantial locality. They propose an algorithm to present these localities and reduce interference in the cache. Ramirez et al. [RLPN+99] design a profile-based code reordering technique which targets a maximization of the sequentiality of instructions, while still trying to minimize instruction cache misses [RLpN+98]: their approach, Software Trace Cache (STC), reorders basic blocks to change taken branches to non-taken ones, moves unused basic block out of the execution path and inline basic blocks from the most popular functions. To reduce instruction cache miss rate, it map the most popular traces in a reserved area of the i-cache. Their approach is based on program profiles and cross functions. In addition, it is unclear how basic blocks with low execution frequency are placed.

Lazy Basic Block Versioning. Chevalier et al. [CBF15, CBF16] optimize type tests by proposing the lazy generation of basic block versioning and generated code patching. This approach uses basic block versioning as a way to propagate type information. At run time basic blocks are compiled in a lazy fashion, making code layouts to reflect the dynamic execution of the code.

8 CONCLUSION

In dynamically-typed languages, baseline JIT compilers often use techniques such as *static type predictions* to optimize common execution paths using heuristics [DS84, Hol94]. Such code patterns imply that the compilation introduces *implicit slow paths*: execution paths that are defined by the language implementation and not by a developer. Further removing completely those *implicit slow paths* from the main execution path requires code reordering techniques [PH90, CT03], with the ultimate goal of maximizing fallthrough branches. However, such heuristics are generally designed to work with profiling information which is not available when the baseline JIT sets off. Newel and Pupyrev showed recently that they may impose suboptimal performance on instruction and I-TLB caches [NP20].

Based on the insight that *implicit slow paths* are known at compile-time, and thus do not require run-time profiles, we experimented with two different code reordering algorithms: Pettis-Hansen augmented with static code layout heuristic, and a slow-to-end heuristic. Our results show that many micro-benchmarks improve their run time by 1.2x. Benchmarks governed by more expensive computations such as message sends show in general no visible performance improvement nor degradations, while very few cases show degradations of up to 1.2x. We show that such static heuristics have low performance impact and could have potential when static type predictions are present in the JIT compiler.

REFERENCES

[Arn02] Matthew Arnold. *Online Profiling and Feedback-Directed Optimization of Java.* Ph.D. thesis, Rutgers University, October 2002.

[ATDM03] Ali-Reza Adl-Tabatabai, A Dynamic, and Compiler Managed. The starjit compiler: A dynamic compiler for managed runtime environments, 2003.

[BBTK+17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Vincent Knight, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. In *OOPSLA*. ACM, October 2017.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[CBF15] Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of javascript programs without type analysis. In *European Conference on Object-Oriented Programming (ECOOP'15)*, pages 101–123, 2015. volume 37 of Leibniz International Proceedings in Informatics (LIPIcs).

[CBF16] Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of javascript programs without type analysis. In *European Conference on Object-Oriented Programming (ECOOP'16)*, pages 1–24, 2016.

[CG99] B Calder and D Grunwald. Reducing branch costs via branch alignment. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1999.

[CL97] J. Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with just-in-time code layout. In *Large-Scale System Administration of Windows NT Workshop (Large-Scale System Administration of Windows NT Workshop)*, Seattle, WA, August 1997. USENIX Association.

[CT03] Keith Cooper and Linda Torczon. *Engineering a Compiler: International Student Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, January 1984.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[HBGM06] Xianglong Huang, Stephen M Blackburn, David Grove, and Kathryn S McKinley. Fast and efficient partial code reordering: taking advantage of dynamic recompilatior. In *Proceedings of the 5th international symposium on Memory management*, pages 184–192, 2006.

[HLM06] Xianglong Huang, Brian T Lewis, and Kathryn S McKinley. Dynamic code management: Improving whole program code locality in managed runtimes. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, page 133–143, New York, NY, USA, 2006. Association for Computing Machinery.

[Hol94] Urs Holzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, Stanford, CA, USA, 1994. AAI9508373.

[LNH08] R. Levin, I. Newman, and G. Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 291–304, 2008.

[NP20] Andy Newell and Sergey Pupyrev. Improved basic block reordering. *IEEE Transactions on Computers*, 69(12):1784–1794, 2020.

[OL21] Guilherme Ottoni and Bin Liu. Hhvm jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 340–350, February 2021.

[OM17] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 233–244. IEEE Press, 2017.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI'90*, 1990.

[RLpN+98] Alex Ramirez, Josep Larriba-pey, Carlos Navarro, Xavi Serrano, Josep Torrellas, and Mateo Valero. Code reordering of decision support systems for optimized instruction fetch. Technical report, Universitat Politechnica de Catalunya, 1998.

[RLPN+99] Alex Ramirez, Josep L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *International Conference on Supercomputing*, 1999.

[Sca98] Daniel J. Scales. Efficient dynamic procedure placement. Technical report, -, 1998.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, January 2000.

[TXD98] J. Torrellas, Chun Xia, and R.L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.

[ZDA20] Oleksandr Zaitsev, Stéphane Ducasse, and Nicolas Anquetil. Characterizing pharo code: A technical report. Technical report, Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla, jan 2020.