Revisiting Dynamic Dispatch for Modern Architectures

Dave Mason
dmason@torontomu.ca
Toronto Metropolitan University
Toronto, Canada

Abstract

Since the 1980s, Deutsch-Schiffman dispatch has been the standard method dispatch mechanism for languages like Smalltalk, Ruby, and Python. While it is a huge improvement over the simple, semantic execution model, it has some significant drawbacks for modern hardware and applications.

This paper proposes an alternative dispatch mechanism that addresses these concerns, with only memory space as a trade-off, that should demonstrate dynamic performance only slightly worse than the best possible with full type information for the program.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Dynamic compilers**; *Just-in-time compilers.*

Keywords: method dispatch, dynamic types

ACM Reference Format:
Dave Mason. 2023. Revisiting Dynamic Dispatch for Modern Architectures. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3623507.3623551

1 Introduction

Since the 1980s, Deutsch-Schiffman dispatch[4, 8, 25] has been the standard method dispatch mechanism for dynamically-typed, object-oriented, languages like Smalltalk, Ruby, and Python. This dispatch is much more efficient than the simple algorithm derived from the semantics of dispatch, and was originally created in order to make Smalltalk useable on commodity hardware, rather than the custom machines upon which it had originally been implemented.

VMIL '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0401-7/23/10…$15.00
https://doi.org/10.1145/3623507.3623551

This paper describes dispatch for class-based, object-oriented systems like Smalltalk, Ruby, and Python. Prototype-based object-oriented languages like Javascript and Self should benefit from the same approach, but in the interests of simpler exposition the relevant details have been omitted. While this paper specifically uses Smalltalk as the language for discussion purposes, most of the same considerations apply to Ruby and Python.

Improving the performance of dynamic dispatch is important because there is considerable evidence of interest[15, 21, 22] in dynamic and OO languages. Most particularly, according to the TIOBE Index[20], the most popular language in the world as of this writing (and in the top 3 for most of the last 4 years), is Python. At least 4 other of the top 20 (Javascript, R, Lua, and Ruby) have similar dispatch characteristics, and 4 others (Visual Basic, Delphi, Swift, Objective-C) probably have some similarities.

If full type information is available at compile time, theoretically all dispatch could be done by directly indexing into a virtual dispatch table (a so-called vtable), but even in languages such as Java this is not fully exploited.

1.1 Problems

Smalltalk is an interesting performance challenge for several reasons.

1. it is dynamically-typed to a fault - everything is an object including methods, classes, and even stack frames;
2. there is no control-flow syntax - all control flow is implemented via message-sending with blocks (lambda functions) as parameters, and the blocks often contain non-local returns;
3. there are no primitive operators as found in most languages (arithmetic operators, comparisons, and indexing are all messages);
4. methods are typically very short (1-5 lines of code) and all expressions are literals, variable references, assignments, or message sends.

In order to obtain adequate performance for Smalltalk on commodity hardware, there are two main techniques:

1. compilers assume the meaning of some particular messages (such as `ifTrue:ifFalse:` and `whileTrue:`) and compile them as if they were syntax in a more conventional language;

2. Deutsch-Schiffman dispatch uses inline caching - i.e. self-modifying code - to obviate the need for the full cost of dynamic dispatch.

There are several problems with these approaches for modern computer systems.

Multiprocessing. In order to capitalize on modern computational power, the ability to use all of the available compute cores/threads is key. Many existing Smalltalk systems use cooperating/green threads to run in a single compute core. This allows the standard dispatch optimizations to function well. However, if multiple, preemptive threads of execution are being utilized, maintenance of inline caches would have to be either put behind a synchronization barrier or have per-thread caches, at considerably greater complexity and memory cost, and reduced effectiveness.

Optimization. In most existing Smalltalk systems, inlining is limited to recognizing where a `BlockClosure` can be inlined. This is critical because it substitutes a conditional branch instead of the creation of the closure (and its subsequent collection as garbage), 2 or more indirect method calls and a potential non-local return. However, because the inlining of particular message sends is key to performance, if those choices are not the correct ones, the performance will suffer. For example, there are many methods that enumerate/iterate over collections in Smalltalk, however only a few of them are inlined. This forces the creation and execution of BlockClosures that really need to be inlined for performance.

Beyond this, in order to perform significant optimizations, it is necessary to have a significant body of code. This is not what is available in most Smalltalk methods. Inlining beyond this level is a current research topic.[17]

Instruction-Level. No matter how efficient dispatch is, at the instruction level, it is loading a method address from memory and then calling that address. If this is unconditional, most modern hardware can schedule the fetching of the instructions at the destination – if it has a long enough pipeline. However, if it is conditional, there is no opportunity to do that scheduling, so it will suffer pipeline stalls.

1.2 Organization of Paper

§2 will discuss the semantics and opportunities for dispatch. §3 will discuss the approach this paper advocates. §4 will discuss the inlining opportunities facilitated by this dispatch model. §5 describes the small amount of related work. Finally, §6 will draw some conclusions and speculate on future work.

2 Dispatch

One of the defining aspects of object-oriented programming is that methods are customized to the class[1]. This requires dispatching to various code, dependent on the class of the object. Since this happens so frequently, optimizing the message dispatch is critical to performance. Furthermore, most OO languages support method inheritance – methods of superclasses are available in subclasses unless they are overridden.

In the rest of this section we will describe dispatch from several different angles: the semantic dispatch that must be emulated; the optimizations used by most Smalltalk implementation; the model that allows Java to have excellent performance; a theoretically optimal dispatch respecting Smalltalk semantics; and finally the flat dispatch model that we propose.

While our model was originally designed to improve the performance of dispatch, there turn out to be potentially more important knock-on effects resulting from the flat dispatch structure itself.

2.1 Semantic Method Dispatch

The semantics of Smalltalk dispatch are described in [7]. Smalltalk message dispatch follows these steps:

1. set Target to the class of the target (or the current class' superclass if it is a super send)
2. set Current to Target
3. look up the selector symbol in Current's method dictionary
4. if found, the value of the lookup is the method code - call it
5. if not found and Current has a superclass, set Current to the superclass and continue at 3
6. if not found, create a Message object and go back to 2 with the selector set to `doesNotUnderstand:` (DNU) - we are guaranteed to find something on this go-round because the class `Object` implements DNU.

Although semantically powerful, even the best possible implementation will still have a lot of overhead. Each lookup in a method dictionary is quite expensive – typically dozens of instructions – even though the hash value only need be calculated once. It is quite common to have class hierarchies 2 or 3 levels deep, requiring multiple lookups.

A further performance concern for Smalltalk is that all control structures are ultimately implemented as messages sent to Booleans passing 1 or more BlockClosures as parameters. This is demonstrated in figure 1 with a simple Smalltalk version of the fibonacci function. Here the test `self <= 2`

```
1 fibonacci
2     self <= 2 ifTrue: [ ^ 1 ].
3     ^ (self - 1) fibonacci +
4       (self - 2) fibonacci
```

Figure 1. Smalltalk implementation of fibonacci

[1] ... or object in prototype languages, or protocol/interface for Strongtalk

results in either `true` or `false` which are singleton instances of their respective classes. That value is then sent the message `ifTrue:` with the parameter `[^1]`. The `True` class has an implementation of the message `ifTrue:` which returns the result of evaluating the block. This particular block does not produce a result, but rather performs a non-local return, which is a bit of overhead. The non-local return passes the value 1 to the *caller* of `fibonacci`. The `False` class has an implementation of the message `ifTrue:` which ignores the block.

2.2 Classic Smalltalk Dispatch

To address these concerns, 3 mechanisms are applied in production Smalltalk runtimes:

1. Some messages such as `ifTrue:`, `ifTrue:ifFalse:` and `whileTrue:` and related messages are recognized by the compiler, and are turned into conditional byte code sequences with the closures inlined, rather than message sends. While this is an extremely important optimization, enabling fully correct semantics can require extremely complex work-arounds.[2]
2. A global method cache, indexed on class+selector provides a 20%-30% performance improvement[9].
3. After the lookup described above, the target method is cached in the calling code, so the next time we do the lookup we should be very fast. This caching gets complicated because there could be objects from another class in a subsequent execution, so somewhat complex mechanisms are used to save the multiple method targets. However the evidence[4, 13] is that about 90% of dynamic call-sites will have a single receiver class, another 9% will have 2–8 classes. Miranda claims[13] that most of the dispatch overhead (5%–10% of total execution time) is burned in the remaining 1% of sends with many classes.

2.3 Java Dispatch

Java has five opcodes to invoke methods, but the primary one we are interested in is `invokevirtual`[16] which does virtual dispatch with the same semantics as Smalltalk.

The difference is that the Java compiler statically knows the index into the dispatch table of each method call, so there is no need for a dictionary lookup. The dispatch table for each class has a prefix of a copy of the dispatch table from its superclass, followed by the methods defined in this class. Any methods that override superclass methods replace the corresponding method in the prefix. Since the names of all of the legal methods are known, finding the method for a particular name requires a simple index (which does not even have to be range checked).

The same thing could be done for Smalltalk, if we knew which class the object was an instance of. Failing that, for every class (there are over 20,000 classes in a recent Pharo image), we would have to have a dispatch table with an entry for all message names (there are over 62,000 method names in the same image).

Even if we somehow knew the class of the object, the tables would still be excessive because of the size of the Smalltalk Object class compared with the Java Object class. The Java Object class only has 11 methods, whereas the Smalltalk Object class has over 460 methods.

Because the dispatch table for every class has a prefix from all super-class methods, this means that the table for every class would start with 460 methods. While this is a significant improvement, it is still excessive (and would have horrible cache locality).

2.4 Optimal Dispatch

Even with the Object class having so many methods, if we had complete information of the types of the receiver for every Smalltalk message send, we could theoretically create unified mappings for all classes, so that all methods could be looked up by simple indexing into the vtable (i.e. creating a perfect hash). This would reduce the overhead of dispatch to a very low level, but for a complete system would have hundreds of megabytes of dispatch table. However, if generating a stand-alone executable from a given target class and its transitive closure of classes and methods, this is potentially a tenable option.

3 New Dispatch

With these considerations in mind, our goal must be to:

- inline everything possible so that code is mainly branches – which are handled well by modern hardware – and to minimize dynamic dispatches;
- make that inlining principle-based, rather than based on heuristics of appropriate selectors;
- where dispatch is necessary, minimize the complexity – a single hashed lookup should be the goal;
- interact with other threads as rarely as possible.

The principle is to have a **single flat dispatch table for each class**, which includes not just the methods of the class, but also all the inherited methods that have been invoked. The methods listed in the vtable are from anywhere in the hierarchy, but only methods that have actually been sent to any instance of this class, in any thread.

3.1 The Dispatch Table

Symbols have a 32-bit 'hash' value that is composed of a 24-bit unique id, and the arity of the symbol. Building perfect hash tables is not feasible (as they end up too large, take too

[2] In a very early version of Pharo/Squeak sending `ifTrue:` to a non-Boolean object required accessing `thisContext` and moving the program counter back. Pharo now de-compiles such a failed branch and does a send, but in a recent version this exposed a Heisenbug.

Table 1. Relative execution time of hash functions

Calculation	M1 factor	i7 factor
`key *% rand *size >> 32`	1.0	1.0
`(key^(key>>5))&size`	1.3	1.1
`key%size`	3.0	8.5

long to build, or take too long to calculate).[2, 5, 18] However it is important to not waste too much space with these tables, while retaining constant time lookup.

3.1.1 Building the Table. The most obvious aproach to hashing is to use a mod function to calculate the remainder of the symbol's hash value modulo the size of the table. Unfortunately, this is very slow on modern hardware. We can perform the same calculation with 2 multiplies and a shift.

First we perform a 32-bit multiply of the 32-bit key value and $\frac{2^{32}}{\Phi}$ function to spread the keys out over the 32-bit space. This is then multiplied by the size of the table and shifted right 32 bits, resulting in a value $0 \leq x < size$. As shown in table 1, this is between 3-times faster than mod on an Apple-M1 processor, and 9-times faster on an Intel-i7 processor. The `xor` function is almost as fast (only 8% to 34% slower), but does not spread the hash values nearly as well as our approach and only works for power-of-two sizes.

One complication is the handling of `super` methods. A common pattern is to have a method handle part of a problem, but with a fallback to the same-named method in the superclass. With all methods for a class and its superclasses compiled into one dispatch table, we have to differentiate the superclass methods, which we do with some unused bits in the Symbol object. With inlining, described in §4, this problem is obviated by the super call usually being inlined.

3.1.2 Dispatching. Dispatching involves several steps:

1. Extract the class number from the target object (this is encoded in the object for immediate objects, or in the header for in-memory objects);
2. Index into a static table of pointers to class dispatch objects - each of which has an array of code pointers;
3. Extract the hash value from the message selector and multiply by the multiplier and the size;
4. Index into the array of code pointers, and make a threaded (indirect) call passing the selector (as well as other registers);
5. if this was a position that corresponds to an unassigned hash value, it will point to the NotFound code (see below).
6. if this is code of a regular method, it checks if the selector matches the selector for that method - if not, it calls the NotFound code.
7. if this was a position that corresponds to a collision, the invoked code is a disambiguator.

Because the dispatch does not need to compare the selector with keys in the method dispatch table, the table is half the size, which should improve cache locality. Rather, the method code compares the selector against the correct one for that method.

3.1.3 Not Found. If a dispatch described in the previous section branches to the NotFound code, it simply means that there is no compiled (threaded or native) method for the given selector. So we set about trying to find the correct code to put in the table.

First we look in the current class for the source (which we store as an AST) of the method. If not found, we try the superclass, and up. Where we find a method we compile it for the target class, and insert it in the dispatch table.

If we do not find it, we send a doesNotUnderstand, in the usual way and insert a specific DNU method for the particular selector.

4 Inlining

Because every method compilation is specific to a particular class there are many more inlining opportunities than in traditional systems where methods are compiled in their particular class and hence the methods reachable come from many levels in the hierarchy.

The primary optimization is inlining, for which there are many opportunities:

- any message to `self` or `super`: these methods could be in the current class, or the target subclass, or a superclass;
- any message to a block: this will inline the corresponding method from BlockClosure;
- any message to other literal values;
- any message to a value for which we have determined a type (such as we have assigned a literal, or we have already tried something that requires it to be the same as `self`);
- any message for which there are a limited number of implementations (e.g. less than 4): this will bring in the code from each of the classes and will generate conditional code that checks the class of the receiver branching to the designated code from the appropriate class, or fall back to a DNU if it is none of the classes;
- any message which has an implementation in particularly common classes (e.g. SmallInteger): this will bring in the code from the particular class and generate code that matches those classes, or fall back to normal dispatch if it is none of the classes.

The last two opportunities, in particular should support a great deal of useful inlining.

There are some interesting aspects of inlining:

- when a method is inlined, its return value will simply be passed to the rest of the method in which it is being inlined;
- when inlining, references to `self` and `super` must be considered relative to the class the method is in, and the receiver value itself can be inlined;
- when BlockClosures are inlined the block goes away and they become simply conditional code within their enclosing method;
- because BlockClosures will often be inlined, most non-local returns will become simple returns;
- methods that are overridden below the target class are irrelevant to this inlining.

In the full running environment the vtable would be generated lazily; if a method is not found, we would find the appropriate method, compile it, and add it to the dispatch table (resizing it as appropriate) – only the adding would have to be protected from other threads. If the source of a method is changed, we simply discard all the dispatch tables where it was inlined (which may not be in its heirarchy) – they would be regenerated on demand.

Even though our lookup requires a hashed lookup, we will be almost as fast as `invokevirtual` in Java or C#.Selector symbols are immediate values, and dispatch tables are prime-sized arrays sized so that most lookups are hashed directly to the expected method.

4.1 Timing

Table 2 shows some preliminary timing for the fibonacci micro-benchmark. Other papers[11, 12] describes the two execution models that we propose in some detail. The models are 'CPS Object' and 'Threaded' - the other rows in the table are for comparison purposes. All are explained below.

The benchmarks were run 5 times each after 2 warmup runs and showed minimal variance (standard deviation less than 2% in all cases). The table records median timing values.

The Pharo data is from running on Pharo 10, on the Pharo version of the OpenSmalltalk VM. Pharo JIT is running on a JIT VM, and Pharo Stack is using the Stack Interpreter VM. While these are included for rough comparison, they are not directly comparable as the optimizations available are quite different.

Native is the straightforward implementation in Zig (which is the implementation language for the Zag runtime) using 64 bit integers. It is very fast, but interestingly this micro-benchmarks is also very small.

Object is the straightforward implementation in Zig using tagged Objects, with local Zig variables and recursion. This is as good as we could hope to be, using the well-supported hardware stack and call/return semantics and the maximum LLVM optimizations. It also demonstrates that using tagged integers does not add *too* much overhead.

CPS Object is the Continuation Passing Style conversion of the Object code using our Context objects and stack. This does not use any of the standard conventions - stack, hardware recursion, static types. The fact that this is only 20% slower, at least on AArch64 is surprisingly good. It is also 10-40% faster than the Pharo JIT'ed code. It is substantially larger though.

Threaded is the fully threaded version. This is extremely easy to generate as it is essentially an extensible bytecode, and it is rewarding that it is only 2-3 times slower than the Pharo JIT'ed code. There was almost no optimization available for such a simple method, so we are very optimistic about the potential for our system.

ByteCode is an unoptimized byte-code interpreter. It is about 1/2 the size of the threaded version and almost 4 times slower. Since it does not save much space, and is actually slightly more difficult to generate than the threaded version, it does not seem worth pursuing at this time.

All of the above handle recursive calls directly, i.e. with no dynamic dispatch.

CPS Send and Threaded Send are the same as the ones above, except messages to fibonacci are sent using full dynamic dispatch. Since there are about 200 million calls to fibonacci, this means that the dispatch costs between 1 and 4 nanosecond per send, depending on the architecture.

All of the above versions were hand-optimized as if the code generator were doing the best possible job.

Finally FullDispatch is a threaded version that does all the sends explicitly, as well as the non-local-return from the block. This is essentially a completely *un*-optimized version of Threaded Send.

As a micro-benchmark of hand-compiled code, this should be taken as simply a validation that this a viable approach to a Smalltalk runtime.

5 Related Work

Virtually all Smalltalk systems currently in use use Deutsch-Schiffman dispatch[4, 8, 25] with inline caching. It is very difficult to identify if others have implemented similar dispatch strategies.

Although it is prototype-based, and therefore not directly comparable, Self[23] may use a similar approach.

V8[1] is a Javascript engine that has very good performance. Potentially some of that performance may come from inlining supported by a flat vtable as advocated here. However from some of the discussion[14] that does not appear to be the case.

Strongtalk[3] would have the potential to have a flat vtable as advocated here, because it has type information that would allow collating the possible methods associated with a message send. From some of the discussion[14] it appears to have rather better performance than V8. However, we can find no reference to the actual dispatch method, and based on

Table 2. Comparative execution times for `40 fibonacci`

Execution	Apple M1	Apple M2 Pro	2.8GHz Intel i7	Size AArch
Pharo JIT	591ms	559ms	695ms	68b*
Pharo Stack	—	5033ms	3568ms	68b
Native	186ms	171ms	137ms	72b
Object	306ms	280ms	337ms	188b
CPS Object	393ms	340ms	655ms	528b
Threaded	1939ms	1153ms	1929ms	200b
ByteCode	4820ms	4230ms	4851ms	104b
CPS Send	749ms	628ms	1069ms	528b
Threaded Send	3503ms	1500ms	3063ms	176b
FullDispatch	7696ms	1433ms	7217ms	184b

* Note this is just the bytecode, the JIT'ed code is on top of this.
Note: unable to get the StackVM version of Pharo to build on the M1.

some of the discussion[13], it does not appear to be the case. It does benefit from similar levels of inlining of methods.

6 Conclusions and Future Work

Although this is preliminary work, we hope it opens up some opportunities for dynamically-typed Object-Oriented languages to get better performance so they will be more viable option in production environments.

We are building a Smalltalk compiler/runtime environment in Zig[6]. We have the managed environment to the point of running simple hand-translated programs, and are near to automatically generating Zig code from our Smalltalk classes and methods.

If the experimental results are encouraging, our goal is to use the LLVM-JIT[10] infrastructure to dynamically generate methods on the fly. Since the current version of Zig uses LLVM, we anticipate we should get comparable performance. We also will explore additional optimizations beyond inlining.

Aspects of this paper may imply that inlining is defining a new language. In fact, what we hope to show is that the flat dispatch model that we describe allows inlining from dead-stock Smalltalk into code that can be competitive with much less dynamic languages. In §1.1 we identified 3 areas where traditional Smalltalk systems fail to exploit modern hardware.

Multiprocessing. Existing Smalltalk systems only use a single core for computation. The same applies to most Python systems (cf. the Global Interpreter Lock [19]) and Javascript only allows additional computation streams via webworkers.

Zag Smalltalk is designed from scratch with multi-core processing in mind:

- there is no self-modifying code
- the only locking associated with dispatch is when a new selector needs to be added to a class dispatch table
- the memory manager has a per-core arena
- the shared memory uses a non-moving collector

Optimization. By convention, Smalltalk is oriented toward small methods. Semantically all control structures are implemented via message sends. Even within a method, the operations between message sends are minimal (the only syntactic primitives are return and assignment). This combination makes traditional optimization extremely difficult.

The flat dispatch described here enables very agressive inlining that replaces many message sends, creation of blocks, and non-local returns with primitive operations and control. The inlining is very synergistic as many inlinings provide additional information that allows yet more inlining.

Instruction-Level. As mentioned in the previous paragraph, after inlining a method will be substantially primitive operations and control. This enable a JIT to perform well-known optimizations, typically only available in statically-typed languages. Additionally, the inlining removes a significant number of method dispatches and therefore replaces indirect branches from the dispatch with conditional branches which re-enables the deep pipelines on modern hardware.

6.1 Future Work

The primary future work is to be able to generate code from "real" benchmarks like DeltaBlue and Richards to actually start to stress the system.

Currently the system uses double-indirection in the dispatch tables, but with a small increase in complexity it looks like we could switch to single-indirection, which would reduce pipeline stalls.

In writing this paper, we started looking at a simple cache similar to a part of Deutsch-Schiffman, which may provide a small but valuable performance improvement.

References

[1] Mads Ager. [n. d.]. *The V8 Javascript Engine: Design, Implementation, Testing and Benchmarking.* Retrieved 2022-05-27 from https://fdocuments.net/document/the-v8-javascript-engine-javascript-engine-built-from-scratch-in-aarhus-denmark.html

[2] Fabiano C. Botelho, D. Gomes, and Nivio Ziviani. 2004. A New Algorithm for Constructing Minimal Perfect Hash Functions.

[3] Gilad Bracha and David Griswold. 1993. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (Washington, D.C., USA) *(OOPSLA '93)*. Association for Computing Machinery, New York, NY, USA, 215–230. https://doi.org/10.1145/165854.165893

[4] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the* 11th *Annual ACM Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, 297–302. https://doi.org/10.1145/800017.800542

[5] Ahmed El-Kishky and Stephen Macke. 2012. A Simulated Annealing Algorithm for Generating Minimal Perfect Hash Functions.

[6] Zig Foundation. 2022. *Zig is a general-purpose programming language and toolchain for maintaining robust, optimal, and reusable software.* Retrieved 2022-06-01 from https://ziglang.org

[7] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Don Mills, Ontario. https://rmod-files.lille.inria.fr/FreeBooks/BlueBook/Bluebook.pdf

[8] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, Pierre America (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–38.

[9] Glenn Krasner. 1983. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley Longman Publishing Co., Inc., Don Mills, Ontario. https://rmod-files.lille.inria.fr/FreeBooks/BitsOfHistory/BitsOfHistory.pdf

[10] LLVM. [n. d.]. *Building a JIT.* Retrieved 2022-05-27 from https://llvm.org/docs/tutorial/BuildingAJIT1.html

[11] Dave Mason. 2023. Threaded-Execution and CPS Provide Smooth Switching Between Execution Modes. In *Proceedings of IWST 2023: International Workshop on Smalltalk Technologies* (Lyon, France).

[12] Dave Mason. 2023. Threaded Execution as a Dual to Native Code. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) *(Programming '23)*. Association for Computing Machinery, New York, NY, USA, 7–11. https://doi.org/10.1145/3594671.3594673

[13] Miranda and Griswold. [n. d.]. *PICs and dispatch.* Retrieved 2022-05-27 from https://groups.google.com/g/strongtalk-general/c/FaDQ2i2Op3M/m/JDNukSaFaNEJ

[14] Miranda, Griswold, et al. [n. d.]. *V8 first impressions.* Retrieved 2022-05-27 from https://groups.google.com/g/strongtalk-general/c/d_R7nsYTVTQ/m/5UGxybWnbToJ

[15] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. 2005. On the Revival of Dynamic Languages. In *Software Composition*, Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13.

[16] Oracle. [n. d.]. `invokevirtual` *JVM instruction.* https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokevirtual

[17] Nahuel Palumbo. 2023. demonstration of inlining methods from a single class. personal communication.

[18] Derek Chi-Wai Pao, Xing Wang, and Ziyan Lu. 2013. Design of a near-minimal dynamic perfect hash function on embedded device. *2013 15th International Conference on Advanced Communications Technology (ICACT)* (2013), 457–462.

[19] Python 2009. *What Is the Python Global Interpreter Lock (GIL)?* Retrieved 2023-09-20 from https://realpython.com/python-gil/

[20] TIOBE. [n. d.]. *TIOBE Index.* Retrieved 2022-05-27 from https://www.tiobe.com/tiobe-index/

[21] Laurence Tratt. 2009. Chapter 5 Dynamically Typed Languages. Advances in Computers, Vol. 77. Elsevier, 149–184. https://doi.org/10.1016/S0065-2458(09)01205-4

[22] Laurence Tratt and Roel Wuyts. 2007. Introduction [Dynamically typed languages: special ed.]. *Software* 24, 5 (Oct. 2007), 28–30. https://doi.org/10.1109/MS.2007.140

[23] David M. Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. 1992. Object, message, and performance: how they coexist in Self. *Computer* 25 (1992), 53–64.

[24] van Jd Diederik Houten. 2011. *improving dynamic language performance using just in time compilation.* Master's thesis. Eindhoven University of Technology.

[25] Wikipedia. [n. d.]. *Inline caching.* Retrieved 2022-05-26 from https://en.wikipedia.org/wiki/Inline_caching

[26] Salikh S. Zakirov, Shigeru Chiba, and Etsuya Shibayama. 2010. Optimizing Dynamic Dispatch with Fine-Grained State Tracking. In *Proceedings of the 6th Symposium on Dynamic Languages* (Reno/Tahoe, Nevada, USA) *(DLS '10)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1869631.1869634

Received 2023-07-23; accepted 2023-08-28