A Metaobject Protocol for Optimizing Application-Specific Run-Time Variability

Position Paper

Guido Chari
UBA, FCEN, Departamento de Computación. ICC-CONICET. Argentina
gchari@dc.uba.ar

Diego Garbervetsky
UBA, FCEN, Departamento de Computación. ICC-CONICET. Argentina
diegog@dc.uba.ar

Stefan Marr
Johannes Kepler University Linz, Austria
stefan.marr@jku.at

Abstract

Just-in-time compilers and their aggressive speculative optimizations reduced the performance gap between dynamic and static languages drastically. To successfully speculate, compilers rely on the program variability observed at run time to be low, and use heuristics to determine when optimization is beneficial. However, some variability patterns are hard to capture with heuristics. Specifically, ephemeral, warmup, rare, and highly indirect variability are challenges for today's compiler heuristics. As a consequence, they can lead to reduced application performance. However, these types of variability are identifiable at the application level and could be mitigated with information provided by developers. As a solution, we propose a metaobject protocol for dynamic compilation systems to enable application developers to provide such information at run time. As a proof of concept, we demonstrate performance improvements for a few scenarios in a dynamic language built on top of the Truffle and Graal system.

ACM Reference format:
Guido Chari, Diego Garbervetsky, and Stefan Marr. 2017. A Metaobject Protocol for Optimizing Application-Specific Run-Time Variability. In *Proceedings of ICOOOLPS'17, Barcelona , Spain, June 19, 2017,* 5 pages.
DOI: 10.1145/3098572.3098577

1 Introduction

In object-oriented languages, a program usually consists of objects (perhaps also classes) and methods which realize the desired behavior. In the presence of dynamic dispatching, the concrete types of objects are often known only at run time. Additionally, such languages typically provide reflective APIs allowing developers to observe or intercede with language elements programmatically while the application is running. All these aspects make the optimization of dynamic languages particularly challenging. However, just-in-time (JIT) compilation successfully faces many of these challenges by assuming that the run-time variability of applications is generally low. For instance, polymorphic inline caches (PICs) [5] for call sites work well under low variability but their performance decreases when new types are observed.

JIT compilers rely on heuristics that decide what, when, and how to optimize. In the absence of mechanisms to interact with compilers, application developers generally treat them as black boxes. This leads to unpredictable performance gains or losses depending on the heuristics hit rate [9]. For example, a call site with high variability is not always an issue for PICs. Based on its heuristics, an aggressive optimizer can clone (split) a method when it is called from different sites so that the variability is reduced by incorporating contextual information [1].

Unfortunately, we found that some use cases do not fit well with the common optimization heuristics. These scenarios present a special kind of variability exhibiting the following main characteristics: 1) They appear as high variability to JIT compilers resulting in run-time overhead. 2) Application developers could know the variability follows a fixed pattern, which makes it suitable for optimization. 3) The pattern instances are application-specific. This kind of variability is common in applications that use dynamic features extensively, for instance, when using proxies for complex initialization or when performing run-time adaptations such as object instance migrations.

To mitigate their overhead, compilation heuristics could be adapted whenever a new case is found. However, this requires changes to low-level and complex artifacts such as the compiler and possibly the virtual machine. This is costly, and gathering the necessary information to recognize complex variability patterns might also harm the overall application performance. Especially for variability that is highly application-specific, a general solution at the compiler level might be too complex and may introduce unacceptable run-time overhead for other applications.

We propose to investigate opening up JIT compilers and behavioral information about programs as part of a comprehensive API in the form of a metaobject protocol (MOP) [6]. Concretely, we propose to explore reflective compilers, *i.e.*, compilers exposing their capabilities to the language level. Exploiting these capabilities, developers can provide fine-tune optimizations and mitigate application-specific run-time overheads by providing additional information to the compiler about a program at run time. This has the advantage of enabling custom optimizations with a generic API, accessible at the language level, and applicable to different scenarios and applications dynamically.

In the remainder of the paper we first provide some background and discuss the different kind of application-specific variability we detected. Then we describe reifications of the compiler behavior and basic structural elements that could be affected by those variability conditions. Examples of such elements are dispatch chains, program specializations, application profiling, and code splitting. Finally, we present running time reductions in two scenarios exhibiting application-specific variability, provide some preliminary conclusions, and discuss our plans to continue this work.

2 Background

This section briefly discusses two basic dynamic language optimizations as background for the remainder of the paper.

ICOOOLPS'17, Barcelona , Spain

Dynamic Objects In dynamic languages such as JavaScript, PHP, or Python, fields can be added and removed dynamically. Furthermore, the type of values stored in fields is not known statically. To optimize the representation of such a dynamic structure of unknown size and shape, the notions of maps [1] and object *shapes* [10] have been proposed. Essentialy, they keep track and cache object structures and field types at run time. Exploiting this information enables a record-like representation in memory, where field accesses can be mapped to direct memory accesses.

Dispatch Chains Polymorphic inline caches (PICs) [5] record type information and cache methods to minimize their lookup overhead. Dispatch Chains generalize PICs to generic operations, such as object field accesses (for dynamic objects) and metaprogramming, caching arbitrary values [7, 10]. Dispatch Chains rely on the stability and low variability of run-time behavior. Each element caches a value (*a specialization*) and has a *guard* to test its validity in the current run-time context. If not valid, the following element in the chain is tested. Finally, dispatch chains are structured so that the last element implements the fallback behavior, i.e., the behavior for the most general case.

3 Handling Application-Specific Variability

As programming language implementers and application developers we have been on both sides of the table. On the one hand, as implementers we know that a compiler needs as much information as possible to *recognize* optimizable patterns. On the other hand, as developers we have dealt with code whose performance has degraded significantly without a clear reason.

3.1 Motivating Example: Ephemeral Variability

Let us consider an application that periodically walks through the elements of a list. To make it concrete and simple, suppose the elements are instances of a Point class and that the application requests their x field. The code snippet in Smalltalk syntax is:

```
Example>>#gatherX
  ↑points collect: [:point | point x]
```

Now, suppose that for at least one of the points in the list, there is a *proxy* protecting the actual point object that has not yet been initialized. When the field read is triggered, the proxy lazily initializes the object and updates the list at the corresponding position. This scenario contains what we call *ephemeral variability*. This variability is in the field access operation, which refers always to the same type (Point) with the exception of the first iterations where it also refers to Proxy.

Figure 1 illustrates the challenges this type of variability poses to a compiler by showing the evolution of the AST representation of *gatherX*.[1] On the left we see the output after parsing. In the middle is the AST after walking through the first point of the list: the uninitialized read was replaced by a specialization that caches the memory offset to look for the x field of Point instances. The AST remains unchanged until we reach the Proxy in the list and add another node for caching the offset for proxies (on the right). Afterwards, the AST no longer changes.

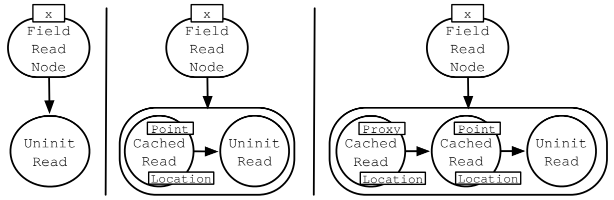

Figure 1. Run-time changes in the AST of the *gatherX* method.

As shown, ephemeral variability affects dispatch chains and thus, performance. In Section 5.2 we show preliminary empirical evidence of its performance effects. In our example, a node caching Proxy location is at the head of the chain although we know that proxies will never be observed after initialization. This phenomenon, which is application-specific, can be hard to detect in a general-purpose JIT compiler. In particular, the compiler cannot determine when Proxy instances will no longer be observed. However, this information can be determined by a developer based on the application behavior. Equipped with the right tools, she could inform the optimizer about the pattern.

3.2 Application-specific variabilities

What follows is an informal characterization of different kinds of variability we have observed in our experiments using TruffleMate as an adaptive platform [3]. While we assume they have been observed by many VM implementers before, to the best of our knowledge, they have not been widely documented yet.

Ephemeral Variability: As we illustrated in the motivating example, ephemeral variability refers to an increase in variability that disappears after some time, perhaps a program phase, after which the system continues the execution with lower variability. This is often observable in applications performing adaptations at run time such as instance migration. Another example is unexpected inputs that trigger exceptions and error handling. Ephemeral variability affects optimizations based on type profiling. Concretely, it may prevent inlining and other optimizations, because the dispatch chains will keep information about all previously observed types (especially in commonly used library functions) while the application only needs a subset of them after startup.

Warm up Variability: A particular kind of ephemeral variability that occurs whenever there is a clearly defined initial phase where applications feature highly dynamic behavior, but afterwards, stabilize and exhibit lower variability. For instance, during startup, an application might initialize a complex data structure involving the instantiation of heterogeneous objects and the execution of complex initialization methods. Afterwards, the application might merely use the data structure and never execute the initialization code again. Depending on the time it takes the instance to properly initialize, the motivating example could also be categorized as showing *warm up variability*.

Rare Variability: Some programs have rare but reoccurring variability in their behavior. Examples can be heterogeneous run-time values appearing occasionally or behavior triggered by a periodic task. This leads to short stretches of execution diverging from stable behavior, and thus, causing problems for existing heuristics. One

[1] We chose ASTs for illustrative reasons but the main idea is generalizable to other representations used by speculative compilers.

of the reasons is that the rarely occurring behavior can still block important optimizations from being performed for the frequently executed code. So, it might be better to avoid optimizing rarely executed code, and instead keep interpreting it. Moreover, in the cases where it is still worth optimizing, the dispatch chains would be contaminated with the last observed values, and a reordering of the specializations would mitigate this phenomenon. Unfortunately, PICs and other heuristics usually do not take rare variability into account.

Highly Indirect Variability: This phenomenon captures variability generated in different source locations, which are usually far from the points where they finally have an impact. It is usually observed in applications that make extensive use of frameworks, libraries, and/or high-order functions. An example is any standard library method that applies a lambda received as parameter. Different call sites will dispatch different lambdas. To mitigate this variability JIT compilers usually *split* (clone) the library method to enable context-sensitive profiling. This significantly reduces the length of the dispatch chain and promotes aggressive optimizations like the inlining of the lambdas. For highly indirect variability, splitting is not sufficient to distinguish different calling contexts preventing optimization. An example is detailed in Section 5.2.

Summarizing, general purpose JIT compilers are usually unable to properly recognize the aforementioned variability *patterns.* The reason is that to keep them tractable, these compilers profile properties that can be easily detected and monitored while these types of variability are usually opaque and application-specific. As a consequence, they lead to performance degradation.

4 Towards a Compilation Metaobject Protocol

Application-specific variability can potentially be handled by a compiler if developers provide the proper information at run time. To do so, we propose the use of reflective compilers, *i.e.*, compilers exposing an API to the language level that can be exploited at run time. In this section, we propose a (not necessarily comprehensive) set of operations for such an API. We focus on compilation aspects that may be affected by the already presented types of variability. Concrete examples, along with code illustrating how to use the API, can be found in Sections 5.1 ands 5.2.

Subsequently, we present the operations based on the entity starting the communication (*i.e.*, application or compiler). We assume an optimizing compiler that works on methods represented as ASTs. However, the ideas can also be applied to other representations, *e.g.*, bytecodes. We reify the essential concepts of ASTs for dynamic languages: nodes, dispatch chains, and profilers. To connect the base and meta levels, each method has access to its AST.

4.1 Run-Time Directives from the Application to the Compiler

Applications can observe or modify the state of a compilation by using the following operations:

- Compile: force the compilation of a method. Handy to accelerate warm up times when the developer knows that the variability is low, especially, after a deoptimization. Also for compiling methods with customized specializations.
- Invalidate: discard the optimized code of a method. This is useful whenever compiled code contains specializations that no longer hold or are blocking optimizations.
- Manage compilation information, covering operations for: i) Inspecting the run-time AST nodes of a method to analyze its current state (specially its dispatch chains). ii) Altering the AST. Concretely: ii.a) Add new nodes, useful for customizing specializations ii.b) Reset, reorder, or remove nodes, in particular, specializations from dispatch chains. These can help to remove profile pollution caused by old phases and/or improve guard execution times (see one example in Section 5.1).
- Trigger optimizations: force method optimizations such as splitting, inlining, and loop unrolling. Useful when the compiler heuristics fail to provide the optimal performance. (See Section 5.2 for an explicit splitting example).
- Activate/Deactive profilers.

4.2 Run-Time Callbacks from the Compiler to the Application

These operations allow applications to customize how the compiler responds to compilation events.

- onSpecialization: when a specialization is going to be added because the current subject's type has not been observed before.
- onGuardSucess/onGuardFailure: to inform about the success or failure when executing a guard.
- onNodeReplacement: when a node is going to be replaced with a more specific behavior for the current subject's type.

5 Validation

As a preliminary validation[2] of our approach, we implemented almost all the compiler directives of the API presented in the previous section.

Implementation. We implemented the API in TruffleMate [2], a Smalltalk VM with comprehensive reflective capabilities for most of its components but not its compiler. In TruffleMate, metaobjects can be installed either on individual objects or method activation frames. When installed, they govern the semantics of the interceded entity. TruffleMate uses a self-optimizing AST interpreter featuring dynamic objects and dispatch chains for speculative optimizations. Preliminary evidence [3] suggests that, in combination with the Graal [11] JIT compiler, TruffleMate can run efficiently.

Experimental Setup. The benchmarking machine is a quad-core Intel Core i7-3770, 3.40 GHz with 16 GB RAM, running Ubuntu with Linux kernel 4.4, and Java 1.8.0_121 with HotSpot 25.121-b13. For both experiments we ran 400 iterations and report 100 measurements after steady state has been reached.

5.1 Ephemeral Variability

We used as baseline the `gatherX` method presented in Section 3 and execute three variations with subtle differences:

1. Instance Migration (IM): One of the points in the list is wrapped by a proxy as introduced in the motivation example.

[2] Instructions on how to reproduce the experiment can be found at: https://github.com/charig/TruffleMATE/tree/papers/ICOOOLPS17

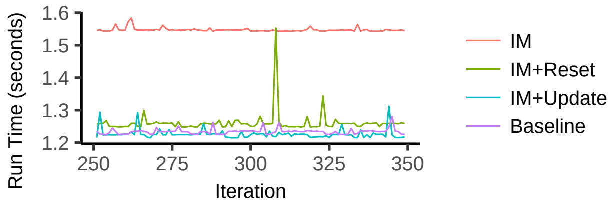

Figure 2. Steady state execution time for the instance migration micro benchmark with ephemeral variability.

2. IM+Reset: Same as IM, but after the proxy initialization, we use the compilation API to find the field reading node of x, gather its dispatch chain, and trigger a reset (See listing below).
3. IM+Update: Same to the previous case but instead of resetting the dispatch chain we just remove the specialization generated by the proxy instance.

```
Example>>#InstanceMigrationReset
  | ast node |
  ast ← (Baseline>>#gatherX) compilation.
  node ← (ast fieldReadsWithName: 'x') first.
  ↑node dispatchChain reset.
```

Results. Confirming our hypothesis, Figure 2 shows IM is slower than the baseline after stabilization: warm up + compilation. Furthermore, both, IM+Reset and IM+Update show performance boosts in comparison to IM producing running times similar to the baseline. The mean overall results for the iterations of each benchmark are: Baseline 1234 *ms*, IM 1547 *ms*, IM+Update 1227 *ms*, IM+Reset 1261 *ms*.

5.2 Highly Indirect Variability

Let us now consider the call to `Vector>>collect:` in the `gatherX` method from the previous example. The callee is part of the TruffleMate standard library. TruffleMate speculates on the block (closure) received as parameter for optimizing its dispatching and enabling the JIT compiler to inline it. To illustrate a highly indirect scenario, suppose now a subtle difference with the previous example: we need to collect both the x and y values from the vector:

```
Example>>#gatherAndProcessXandY
  | xValues yValues |
  xValues ← points collect: [:point | point x].
  yValues ← points collect: [:point | point y].
  self process: xValues and: yValues.
```

This example presents highly indirect variability because it calls `collect:` twice within the same method (context) but with two different blocks. Since the context is the same, the (Graal) compiler heuristics avoid splitting `collect:`.

Benchmarks. We ran the previous example in two different flavors: *Indirect* runs exactly the example while *Indirect+Split* forces the splitting of the second call to collect using the compilation API:

```
Example>>#splitCollect
  | ast send callSite |
  ast ← (Example>>#gatherAndProcessXandY) compilation.
  send ← (ast messageWithSelector: 'collect:') second.
  callSite ← send dispatchChain firstSpecialization.
  ↑callSite split.
```

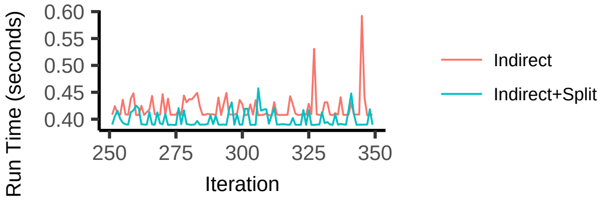

Figure 3. Steady state execution time for the splitCollect micro benchmark with highly indirect variability.

Results. Figure 3 shows a reduction in the execution time of Indirect+Split. The mean overall results for the iterations of each benchmark are: Indirect 420 *ms* and Indirect+Split 399 *ms*, resulting in a performance gain of about 5%. In the case of critical methods, this simple fine-tuning at run time appears worth trying.

5.3 Conclusions

These preliminary results indicate that a reflective compiler could improve the obtained performance of a general-purpose JIT compiler by leveraging developers knowledge of the application to fine-tune the optimizations. A more comprehensive validation is still needed to better understand several aspects such as the best applicable scenarios and the concrete limits and drawbacks of the approach.

6 Related Work

For brevity, this section discusses only decaying invocation counters [4] and Lancet [9], a recent compilation framework closely related to our work.

Decaying Invocation Counters. Hölzle proposed that method activation counters represent invocation rates instead of counts. He suggested counters that decay exponentially to avoid the compilation overhead for methods that are not performance critical. Furthermore, Hölzle proposed to dynamically adapt the decay rate depending on the stability of the system. This would mitigate problems with transient variability such as warm up, rare, and ephemeral variability. However, this heuristic does not apply to all issues we identified. Concretely, it would fail whenever an application phase containing transient behavior executes the corresponding method frequently enough. Our approach complements compilation heuristics for the cases when they are not enough, by giving developers the opportunity to amend those scenarios using a comprehensive language-level API.

Lancet. Recently, Rompf et al. presented the Lancet JIT compiler framework for Java bytecode that enables programs to control several aspects of the JIT compilation process. Lancet features hooks, used to trigger a predefined set of macros whenever a method is going to be compiled. On top of these macros, Lancet features an API allowing developers to *annotate* methods with compilation directives such as compile, unroll, or freeze (partial evaluate at compile time). We share the same vision and goals noticing a potential in connecting a JIT compiler with the application it optimizes. However there are differences in the approaches. While Lancet's API is mainly used at compile time, our API was designed to enable the inspection and modification of the compilation aspects of a method at run time. In addition, we focused on reifying aspects regarding speculative optimizations such as dispatch chains and splitting, while Lancet focused on other aspects such as partial evaluation and inlining. For instance, to the best of our knowledge Lancet is not able to deal with our case studies. Lastly, Lancet advocates for an integration of the compilation directives within the source code, while we promote the modularization of the optimization aspects by the use of a MOP.

7 Conclusion

We presented a series of application-specific variabilities, namely: ephemeral, warm up, rare, and highly indirect. We showed how they may challenge state-of-the-art JIT compilers, leading to performance overheads. These overheads could be mitigated if application developers had the means to supply the compiler with additional information and improve over heuristics. Therefore, we proposed to make compilers reflective, enabling the introspection and intercession of optimization-related aspects, for instance, the ability to manage dispatch chains or force a deoptimization. As a roadmap to follow, we presented an API covering several aspects of the compilation process, implemented a subset, and obtained significant overhead reductions in a couple of preliminary scenarios including application-specific variability.

It is worth noting that our approach relies on developers with advanced knowledge of both, the application specifics and compiler optimizations for dynamic languages. We expect the API to be used in cases where the application-specific variability is known *a priori* to block optimizations. Furthermore, because of being reflective, we expect also a posteriori usages of the API, *i.e.*, run-time tweaks for boosting the performance of running systems. On the negative side, developers might not use the compiler API properly. Wrong hints and directives given to the optimizer may lead to potential performance penalties. More comprehensive use cases stressing the usage of the MOP are needed to analyze its eventual negative impact.

Concluding, we think reflective compilers open new perspectives for run-time adaptive scenarios. This was already suggested by the context-oriented programming community [8]. For the near future, we plan to refine, extend, and implement the API completely. Furthermore, we also plan to evaluate its performance results in the context of more comprehensive applications to better understand the prerequisites for, and overall impact of, using the compilation API.

Acknowledgments

We would like to thank the anonymous reviewers of this paper for their constructive feedback which contributed to improve it. This work was partially supported by the projects ANPCYT PICT 2013-2341, ANPCYT PICT 2014-1656, ANPCYT PICT 2015-1718, UBACYT 384, CONICET PIP 2015/17 №11220150100931CO. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70. ACM, October 1989. doi: 10.1145/74878.74884.

[2] G. Chari, D. Garbervetsky, S. Marr, and S. Ducasse. Towards fully reflective environments. In *Onward!* ACM, 2015. doi: 10.1145/2814228.2814241.

[3] G. Chari, D. Garbervetsky, and S. Marr. Building efficient and highly run-time adaptable virtual machines. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 60–71, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4445-6. doi: 10.1145/2989225.2989234. URL http://doi.acm.org/10.1145/2989225.2989234.

[4] U. Holzle and D. M. Ungar. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. Number 1520. Department of Computer Science, Stanford University, 1994.

[5] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, pages 21–38. Springer, 1991. ISBN 3-540-54262-0.

[6] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN 0262111586.

[7] S. Marr, C. Seaton, and S. Ducasse. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and Without Compromises. In *PLDI*, pages 545–554. ACM, 2015. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737963.

[8] T. Pape, T. Felgentreff, and R. Hirschfeld. Optimizing sideways composition: Fast context-oriented programming in contextpypy. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, COP'16, pages 13–20, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4440-1. doi: 10.1145/2951965.2951967. URL http://doi.acm.org/10.1145/2951965.2951967.

[9] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 41–52. ACM, 2014. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594316. URL http://doi.acm.org/10.1145/2594291.2594316.

[10] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the truffle language implementation framework. In *PPPJ*, pages 133–144. ACM, 2014. doi: 10.1145/2647508.2647517.

[11] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Onward!*, pages 187–204. ACM, 2013. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581.