Who You Gonna Call? A Study of the Call-Site Behaviour of Ruby-on-Rails Applications

Sophie Kaleba
University of Kent
Canterbury, United Kingdom
S.Kaleba@kent.ac.uk

Octave Larose
University of Kent
Canterbury, United Kingdom
O.Larose@kent.ac.uk

Stefan Marr
University of Kent
Canterbury, United Kingdom
S.Marr@kent.ac.uk

Richard Jones
University of Kent
Canterbury, United Kingdom
R.E.Jones@kent.ac.uk

TALK PROPOSAL

Web-applications are ubiquitous, from simple personal blogs to e-commerce platforms with millions of sales. Ruby-on-Rails [1] is a popular framework implemented in Ruby that provides tools to build such web-applications. Performance is often critical in the context of large-scale web-applications; especially in dynamic languages such as Ruby that feature reflection and the use of many small methods. Such languages therefore benefit from run-time optimisations, notably through the combined use of lookup caches, splitting and inlining.

To limit their overhead, such optimisations generally rely on assumptions that do not necessarily match with the actual run-time behaviour. With Phase-based splitting [4], we showed that splitting can benefit from using homogeneous patterns of behaviour, called "phases" [6, 7] to reach better performance. In an effort to identify such phases in real-world web-applications, we thoroughly analyse the run-time call-site behaviour of Ruby programs and Ruby-on-Rails applications, running on top of TruffleRuby [8]. This talk describes our findings and aims at guiding future research on call-site optimisation.

Findings about common call-site optimisations. Our analysis first focuses on the call-site behaviour at run time and more precisely how lookup caches are influenced by call-site optimisations. A **lookup cache** [3] holds the different call-targets that can be called at a particular call-site; this caching helps avoiding the otherwise expensive lookup cost. If the cache contains a single entry, it is qualified as monomorphic. It becomes polymorphic once it contains more than two entries; polymorphic lookup caches holding a large number of entries are often considered to be megamorphic. **Target polymorphism** is observed when several entries in the lookup cache point to the same target which may happen if i.e. two classes share the same implementation of a method. This issue is optimised in TruffleRuby by relying on an extra cache level to store the observed type of the receiver. **Splitting** [2] is an optimisation that clones methods so that their lookup cache is reset; splitting decisions are typically triggered based on the degree of polymorphism of the lookup cache of a call-site and its close callers. In TruffleRuby, splitting may in addition be explicitly triggered for core methods since they are more likely to be polymorphic and massively executed at start-up.

We analyse the effect of addressing target polymorphism and splitting in a benchmark set containing common micro- and macro-benchmarks [5] as well as micro- and macro-benchmarks used by the Ruby community. We first confirm that polymorphic caches tend to negatively affect performance, e.g. in TruffleRuby, a cache of 5 entries is 1.5x slower than a monomorphic cache. A megamorphic cache is up to 6x slower. However, polymorphic call-sites remain rare, and represent on average less than 1% of all method calls. When focusing on closure application sites, we see that 24% of all closures applications are polymorphic, which represent 2.6% of all calls (method calls and closure applications). We expect that addressing target polymorphism should lower the degree of polymorphism. Our analysis indeed shows that having an extra cache helps to monomorphise lookup caches, where some of our benchmarks saw more than 50% of their calls turning monomorphic.

We also expected that splitting should ideally lead to monomorphic lookup caches. It may also prevent call-sites from reaching the usually slower megamorphic state. In our analysis, we first assess the performance of the different splitting strategies available in TruffleRuby on Ruby-on-Rails web-applications and check how their combination performs in terms of execution time. At the same time, we examine the lookup caches of the most executed call-sites and check whether the splitting strategies in place perform as expected. We previously identified [4] that a call-site experiencing a polymorphic phase followed by a monomorphic one could be split on phase switch to improve performance. The goal of our analysis is to identify other kind of possible trigger points for high-level behaviour-driven, i.e., phase-based, splitting.

Identifying high-level behaviours impacting lookup caches. Monitoring the state of lookup caches at run time is time-consuming; a low-overhead alternative is to identify higher-level behaviours that could be used as proxy of the lookup caches' status. After examining lookup caches' behaviour at run time, we further analyse our set of web-applications to identify such high-level behaviours. We focus on request-based web-applications and investigate whether typical features of these applications produce execution phases. To do so, we observe different aspects of call-sites,

such as their type and their receiver(s) and monitor their evolution during execution; we then check how they relate to high-level behaviours.

We notably investigate how different web-routes impact call-site behaviour and whether phased-behaviour emerges. Such routes are generally composed of a type of HTTP method (e.g. GET), a path (i.e. URL) and a method responsible for handling that path; we explore different combinations of these components. Testing different variations of paths only do not seem to produce any phased-behaviour in a Ruby-on-Rails blog application; experimenting with other kinds of web-applications may however lead to different results, especially if different templates are rendered. We conducted further experiments by varying the type of HTTP methods: call-site behaviour seems to differ significantly between GET and POST requests, which creates phases. This use-case is currently under closer investigation.

REFERENCES

[1] Michael Bächle and Paul Kirchberg. 2007. Ruby on rails. *IEEE software* 24, 6 (2007), 105–108.

[2] Craig Chambers, David Ungar, and Elgin Lee. 1989. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM Sigplan Notices* 24, 10 (1989), 49–70.

[3] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*. Springer, 21–38.

[4] Sophie Kaleba, Stefan Marr, and Richard Jones. 2021. *Avoiding Monomorphization Bottlenecks with Phase-Based Splitting*. Association for Computing Machinery, New York, NY, USA, 16–18. https://doi.org/10.1145/3484271.3484976

[5] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language compiler benchmarking: are we fast yet? *ACM SIGPLAN Notices* 52, 2 (2016), 120–131.

[6] Priya Nagpurkar. 2007. Analysis, Detection, and Exploitation of Phase Behavior in Java Programs. (2007), 217.

[7] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, Rhodes Island, Greece, 10 pp. https://doi.org/10.1109/IPDPS.2006.1639325

[8] Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. 2017. TruffleRuby–A High Performance Implementation of the Ruby Programming Language.