Citation for published version


DOI

Link to record in KAR

https://kar.kent.ac.uk/89179/

Document Version

Author's Accepted Manuscript

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
Avoiding Monomorphisation Bottlenecks with Phase-based Splitting

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

Programming languages, such as JavaScript, Ruby or Python, rely on a managed runtime to reach state-of-the-art performance (see respectively V8 [2], JRuby [4], PyPy [7]). Such runtime systems apply aggressive optimisations based on speculative assumptions: one common assumption is that the behaviour of a program remains mostly homogeneous at run time. However, literature [5, 6] on the run-time behaviour of programs shows that programs contain several distinct phases (i.e. intervals of time exhibiting a distinct and relatively homogeneous behaviour) and that these phases may also repeat. For instance, the backend of a scientific computation web server can be viewed as such a program: first it analyses the lookup cache containing several non-needed entries leads to a 14.3% slowdown. On NodeJS, we obtained respectively a 40% and a 19% slowdown. These results suggest that phase-based monomorphisation based on splitting can increase performance.

We argue that using data on the high-level behaviour of a program, especially phases, could guide splitting heuristics to improve performance. We introduce a proof-of-concept prototype in TruffleSOM that provides a minimal set of source code annotations to identify phase-sensitive call-sites candidates for splitting and to identify phase switches. Concretely, the first time a phase switch occurs, the phase-sensitive call-site is split so that two versions of the method(s) co-exist: one for the polymorphic phase and one for the monomorphic phase.

We apply this prototype to our two micro-benchmarks to respectively split a call-site and a closure application site that are both phase-sensitive. The results are promising: splitting the phase-sensitive call-site leads to an average 18.5% speedup compared to baseline, i.e. a version of the program without annotation. Furthermore, if we focus on the performance at the phase granularity, we reach speedups ranging from 37% to 47.6% during monomorphic phases where the call-site lookup cache used to be filled with non-needed targets. Likewise, splitting the phase-sensitive closure application site leads to a speedup: 10% on average compared to baseline, with speedups ranging from 21% to 23.2% when looked at phase granularity. In addition, this particular use-case highlights an interesting splitting issue: in this benchmark, a function parameter takes a closure, which is later activated. This code structure is common in modern frameworks, causing closure application sites to be polymorphic, without splitting heuristics as in GraalVM being able to resolve it.

These experiments are a starting point to show that phased behaviour is still overlooked, but suggest that a phase-aware lightweight system guiding dynamic optimisations could offer significant performance improvements. For instance, this approach could help reduce the overhead of frameworks and the inclination they introduce by monomorphising performance-critical code. In the last part of this talk, we will describe what could be the future of this project in terms of implementation choices. We will also discuss other dynamic optimisations that could benefit from phase awareness.

REFERENCES


