Optimizing Sideways Composition

Fast Context-oriented Programming in ContextPyPy

Tobias Pape[*] Tim Felgentreff[*†] Robert Hirschfeld[*†]

[*]Hasso Plattner Institute, University of Potsdam, Germany
[†]Communications Design Group (CDG), SAP Labs, USA
[†]Viewpoints Research Institute, USA
{firstname}.{lastname}@hpi.uni-potsdam.de

Abstract

The prevalent way of code sharing in many current object systems is static and/or single inheritance; both are limiting in situations that call for multi-dimensional decomposition. Sideways composition provides a technique to reduce their limitations. Context-oriented programming (COP) notably applies sideways composition to achieve better modularity. However, most COP implementations have a substantial performance overhead. This is partly because weaving and execution of layered methods violate assumptions that common language implementations hold about lookup. Metatracing just-in-time (JIT) compilers have unique characteristics that can alleviate the performance overhead, as they can treat lookup differently.

We show that meta-tracing JIT compilers are good at optimizing sideways composition and give initial, supporting results. Furthermore, we suggest that explicit communication with the JIT compiler in a COP implementation can improve performance further.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords context-oriented programming, meta-tracing JIT compilers, optimization, virtual machines, PyPy

1. Introduction

Sideways composition provides a technique to avoid some of the limitations of static, single inheritance object-oriented systems where multi-dimensional composition of behavior is desirable. In particular, context-oriented programming applies sideways composition to improve modularity. COP as a modularity mechanism to dynamically adapt behavior at runtime has been demonstrated to be useful in a variety of scenarios. Beyond its original motivation for dynamically adapting systems based on environmental factors such as battery level, geolocation, or time of day [9], COP has also been applied to provide safety in the development of live systems [12] or to let multiple conflicting versions of programming interfaces co-exist [7].

However, as with many abstraction mechanisms, COP comes with some overhead, a fact that has been repeatedly recognized. By weaving and executing layered methods, it violates assumptions of common language implementations and thus context layer aware method lookup requires additional operations at run-time. Most COP implementations in dynamic languages use the host language's metaprogramming facilities to redirect method dispatch, whereas statically compiled languages require additional compilation steps to construct data structures to track layer activation states at run-time. Both of these solutions come with considerable performance decrease from 75 % to 99 % [2].

We argue that virtual machines (VMs) with meta-tracing JIT compilers can alleviate the performance overhead, because their execution model allows them to optimize nonstandard lookup. This can be achieved by explicitly telling the JIT complier crucial information using small "hints" on the side of the COP implementation, avoiding a complex architecture to cache and optimize lookup.

In this work, we make the following contributions and show

- that the performance overhead of sideways composition is still present in most COP implementations;
- that meta-tracing–based JIT compilation can reduce the overhead of sideways composition; and
- that announcing the active layer composition *explicitly* to the JIT compiler can further reduce the overhead of sideways composition.

COP'16, July 17-22 2016, Rome, Italy
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4440-1/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2951965.2951967

2. Background: Meta-tracing JIT Compilers

Just-in-time (JIT) compilation is one of the most frequently used technique for speeding up the execution of programs at run-time. Many language implementations have yet benefitted from JIT compilers, including but not limited to today's popular languages such as Java [13] or JavaScript [10]. A particular implementation approach for JIT compilers is to use *tracing*, that is, recording the steps an interpreter takes to obtain its instruction sequence, a *trace*. This trace is subsequently used instead of the interpreter to execute the same part of that program [3] at higher speed.

With *meta-tracing* the execution of the interpreter is observed instead of the execution of the application program. The resulting traces are therefore not specific to the particular application but to the underlying interpreter [5]. This way, language implementers do not have to implement optimized language-specific JIT compilers but rather to provide a straightforward language-specific interpreter that is subject to the meta-tracing technique. "Hints" enable communication with and hence fine-tuning of JIT compiler [4].

3. Faster Sideways Composition with Meta-tracing

We propose that meta-tracing JIT compilers can reduce the overhead of sideways composition and COP, possibly more so when telling the active layer composition to the JIT compiler.

3.1 Employing a Meta-tracing JIT

Context-oriented programming employs sideways composition to inject context-dependent behavioral changes into an existing hierarchy of behavior. These hierarchies are typically defined by the static and/or single inheritance of object systems. These hierarchies are typically important for execution time performance, as they form the basis for *lookup*.

Most execution environments, such as VMs for dynamic object-oriented languages, assume that those hierarchies change rarely and hence lookup can be fast. However, using sideways composition to alter behavior invalidates this assumption. Especially, since COP explicitly redefines lookup based on currently active layers; the composition of currently active layers becomes *crucial* for calculating the lookup in the dynamic extent of executed code. If this composition stays the same, lookup stays the same, if it changes, lookup may change. Execution environments typically have to decide, whether to always re-exercise the lookup for every method under the active layers, or cache (and invalidate) lookup information when the composition of active layers changes. For example in the sequence

```
-- active layers: ∅
method1()
  activate(layer1)
    method2()
  deactivate(layer1)
```

a COP implementation typically uses one of the following two strategies:

1. lookup `method1()` under the composition ∅ and lookup `method2()` under the composition `[layer1]`, or
2. use cached lookup information for `method1()`, switch cached lookup information due to change in active layers, use (new) cached lookup information for `method2()`.

Both cases effect a performance impact either on every lookup or on every layer change.

With meta-tracing JIT compilers, however, this effect is much less severe. Although rarely-changing lookup still is helpful for its operation, a change in lookup — for example induced by a layer activation — can be anticipated and be accounted for.

Thus, with a properly instructed meta-tracing JIT compiler, a third option becomes available. At points in the execution where the composition of active layers becomes important, a *guard* ensures that this composition did not change. While counter-intuitive at first, this actually is a benefit. When a certain different composition has been encountered often enough at the guard, the meta-tracing JIT compiler will introduce a *bridge* into a new part of a trace, in which *this* different composition can be assumed not to change, and lookup can be optimized accordingly. Note that this resembles strategy 2 above, but is implicit and guided by the JIT compiler. Therefore, the COP implementation does not have to manage the caching information when altering the lookup information, saving both execution time and implementation complexity.

3.2 Promoting the Compositions of Active Layers

The compositions of active layers is crucially important for the lookup in COP, and all strategies above reflect this. However, only the language-level implementation of COP knows about the composition's importance, and even for meta-tracing JIT compiler's strategy, the JIT compiler first hast to become aware of the fact that the composition *is* important for its trace. Yet, the JIT compiler has to apply heuristics to identify the composition as trace-important.

This situation is commonly known when implementing VMs that use a meta-tracing JIT compiler. Based on the value of a certain object it may be desirable to *specialize* traces to these values (essentially what happened above with the guard and the bridge). Implementers can chose to *promote* [4, §3.1] such an object and the JIT compiler will ensure that traces are specialized to the object's values, regardless of wether the JIT compiler's *heuristics* would result in the same specialization or not. If applied carefully, this promotion can decrease execution time.

Up until recently, this *promotion* of objects had not been available to language-level implementers of COP. However, at the time of writing, one VM with meta-tracing JIT compiler (PyPy) exposes the *promote* functionality to the language-level and it is possible to use it for a COP implementation.

The composition of active layers can now be promoted and the meta-tracing JIT compiler now ensures that (a) a specialized traces exists for each encountered composition, and (b) within a given trace, the composition will not change and can be relied upon. This assumption now can be made when exercising lookup during execution, saving execution time.

4. ContextPyPy Implementation Outline

We briefly describe certain implementation details of ContextPy and how "hints" to the meta-tracing JIT compiler lead to ContextPyPy.

During development, programmers using ContextPy can annotate methods as being advices before/after/around their base methods in a certain layer (Layer-in-Class). Using Python annotations, these methods are registered to a *descriptor* holding onto all partial methods and possibly a base method. That way only methods that are layered at least once are affected. The descriptor, being a Python callable, takes the place of the named method in the class and, from this point on, dictates execution and effectively lookup of the partial methods. When code calls the method — now represented by the descriptor — the active layer composition is determined, considering globally activated (`globalActivateLayer` plus counterpart) but also dynamically active layers. For that, a thread local storage provides a dynamically scoped layer composition, which can be affected using Python's `with` syntax (`with activeLayers(myLayer): ...`). Based on that composition, a partial method ordering is determined and cached using the composition as key. This is a simple optimization most other COP implementations also use. Then, the first method of the ordering is executed. The global function `proceed()` inspects the descriptors ordering to determine the next method to be executed.

As described in section 3.2, it is desirable to have one trace per layer composition, which the heuristics typically provide. However, with recent PyPy, we can tell this directly to the JIT compiler. Moreover, we are interested in the method ordering *resulting* from the layer composition to know what method to execute next. Therefore, in ContextPyPy, the method "talking" to the JIT *promotes* this ordering, as presented in the following method:

```
def get_or_build_methods(self):
  if self.layer_composition_unchanged():
    return promote(self.last_ordering)
  layer_composition = _baselayers + _tls.activelayers
  last_ordering = self.method_ordering(layer_composition)
  return last_ordering
```

5. Performance Evaluation with COP

As described in the introduction, we evaluate the following:

1. Sideways composition *still* has a considerable impact on execution time.
2. Meta-tracing JIT compilers can alleviate the performance impact of sideways composition on execution time.
3. Explicit *promotion* of the active layer composition can further improve execution time.

For 1, we re-run in parts benchmarks presented in 2009 [2], which showed a performance impact of sideways composition as used with COP implementations. For 2, we additionally augment the well-known DeltaBlue benchmark [8] with layers and additional functionality and compare the impact of sideways composition on platforms with and without meta-tracing JIT compilers. For 3, we run both benchmarks with a *promote*-enhanced COP implementation, as well. We present and discuss the most important results for each benchmark and provide all results in the appendix.

5.1 Setup

System We executed the benchmarks on an Intel Core i7-4850HQ at 2.3 GHz with 6 MB cache and 16 GB of RAM. The machine ran Mac OS X 10.9.5. Certain benchmarks were run on an Intel Core i7-4650U at 1.7 GHz with 4 MB cache and 8 GB of RAM, this machine ran Windows 10.

Implementations We used ContextPy[1] with Python 2.7.5 and PyPy 5.1 on OS X and ContextPyPy[2] with PyPy 5.1 on OS X in both benchmarks. Additionally, for the re-run of the 2009 benchmark, we used ContextJS [11][3] with V8 (Chrome 50.0.2661.66 beta 64-bit) on OS X, Chakra (Edge 25.10586) on Windows 10 and V8 (Chrome 49.0.2623.112) on Windows 10; as well as LispWorks® 64-bit 7.0.0 on OS X.

Introducing layers with ContextPy increases the initial size of the traces beyond PyPy's standard maximum trace limit. Therefore, we use a slightly altered build of PyPy 5.1 that allows a much higher trace limit than the standard build.

Methodology For COP09a and COP09b (see section 5.2), every benchmark was run with increasing size until a measurement took at least 5 seconds; this matches the original methodology [2]. Warm-up is provided by the not measured runs. For DeltaBlue (see section 5.2), every benchmark was run 5 times uninterrupted in a new process with additional 3 times warm-up prior to measurement for PyPy[4]. The execution time was always measured *in-system* and, hence, does not include start-up. We show the arithmetic mean of all runs along with bootstrapped [6] confidence intervals for a 95 % confidence level.

5.2 Benchmarks

Our first set of benchmarks is taken from one of our earlier papers [2]. These micro-benchmarks attempt to measure the pure performance overhead of dispatching to active layers and of layer activation. In the first part of these benchmarks (COP09a), we use an object with ten integer

[1] https://bitbucket.org/cfbolz/contextpy/ rev 5155cb7.

[2] ContextPy as above but with explicit *promote*.

[3] https://github.com/LivelyKernel/LivelyKernel/tree/master/core/cop rev 0bd117c.

[4] Python did not exhibit any warm-up–related differences

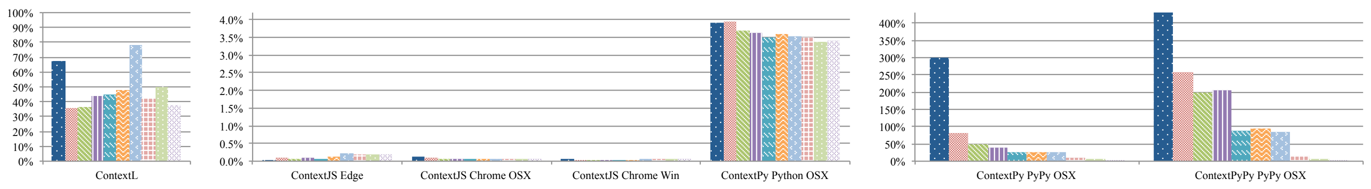

Figure 1. Results of COP09a. Relative throughput of method execution in COP implementations with (each left to right) 0 to 10 layers normalized to the respective non-layered workload. Higher is better. For raw numbers see Table A.2 and Table A.4. (Note the different scales, see Figure A.1 for an overview comparison.)

variables (`counter`$_1$ to `counter`$_{10}$) that provides ten methods (`method`$_1$ to `method`$_{10}$), where each `method`$_i$ increments all counters from `counter`$_1$ to `counter`$_i$ by one. The same behavior is provided by a method `layered`. The base method increments only `counter`$_1$, and nine layers (`Layer`$_1$ to `Layer`$_9$) provide a partial method to adapt the base method to each increment one of the other counters. Running just the `layered` method without any layers being active thus yields the same behavior as `method`$_1$.

In the second part (COP09b) of our first benchmark set, we measure the performance impact of layer activation. For most COP languages, layer activation means updating internal data structures with the current layer composition. To quantify this impact, we measured the execution time of running five methods (`method`$_1$ to `method`$_5$) in succession that each increment one counter. We compare this to the execution time of five partial methods from five layers that implement the same method body, where each layer is activated in succession.

Our second benchmark (DeltaBlue) measures the relative overhead of sideways composition on a more wide-spread benchmark, DeltaBlue. For that, we augmented DeltaBlue with additional functionality (reversed lists for storage, count of constraint executions and binary constraints), both statically added to the benchmark (Delta*Purple* △) and dynamically composed via layers (Delta*Red* ⊞). To measure the overhead of the mere presence of dynamic sideways composition functionality without actually using it, we also measured DeltaRed with all layers deactivated (Delta*Violet* ▣). We normalize the execution time of all four benchmarks to Delta*Blue* ◙ on each implementation and VM. That way, we show the overhead of the COP implementation rather than mere execution time.

5.3 Results

The results of the COP09a benchmark are shown in Figure 1. We evaluated the performance by comparing each ordinary method `method`$_i$ with the execution performance of activating all layers from `Layer`$_1$ to `Layer`$_i$ (which gives the same behavior) and normalizing to the ordinary method. As we reported in our previous work, ContextL shows a performance degradation ranging from 22 % to 65 %. Interestingly, the decrease in performance does not seem to correspond to the number of active layers, likely due to the variability of the optimizations of the underlying Lisp VM. On the V8 and Chakra JavaScript VMs, ContextJS incurs a massive performance hit of over 99.7 % in all cases, even where no layer is active. ContextPy on the Python VM is little better with overhead around 95 %.

Running ContextPy on PyPy, that is, using a meta-tracing JIT compiler, yields vastly different results. Against the trend of the other implementation/platform combinations, using COP layers can *increase* performance. When just layers are present but not activated, ContextPy on PyPy can gain a 3.5× speedup and ContextPyPy even over 4×. Compared with ContextPy on Python, the relative performance difference is up to two orders of magnitude improved.

Moreover, ContextPyPy manages to retain a speedup of at least 2× over the non-layered version up to three active layers, exhibiting only minor slowdown for four to six active layers. After that, the performance levels up with ContextPy on PyPy. We attribute the latter, rather steep decline in performance to how the meta-tracing JIT compiler handles rather long traces that can occur with an increasing number of active layers and which are yet to be investigated.

Discussion (COP09a) The meta-tracing JIT compiler of PyPy appears to be effective at eliminating the overhead of sideways composition. The results of ContextPyPy suggest that communicating layer information properly can vastly improve this effect, too.

The results of the COP09b benchmark are shown in Figure 2. The performance impact for layer activation for each of the tested systems is comparable and clearly increases as more layers are activated. Moreover, ContextPy on Python — the only implementation/platform combination without JIT or JIT-like optimizations — exhibits the least severe impact with increasing layers.

Discussion (COP09b) The meta-tracing JIT compiler of PyPy seems to have to invalidate assumptions on layer activations, possibly not being able to re-use certain traces. The results of ContextPyPy suggest that the current way of communicating layer information to the meta-tracing JIT compiler in fact can also hamper optimizations, that is, such *hints* have to be used with great care.

The results of the DeltaBlue benchmark are shown in Figure 3. Expectedly, the additional workload of DeltaPurple

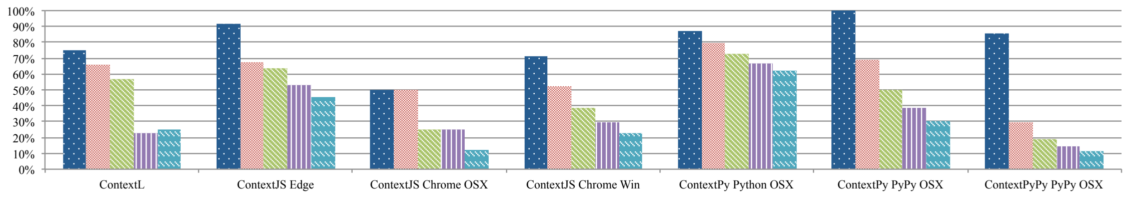

Figure 2. Results of COP 09 b. Relative throughput of layer activation in COP implementations with (each left to right) 1 to 5 layers normalized to a workload with no layers active whatsoever. Higher is better. For raw numbers see Table A.3 and Table A.4.

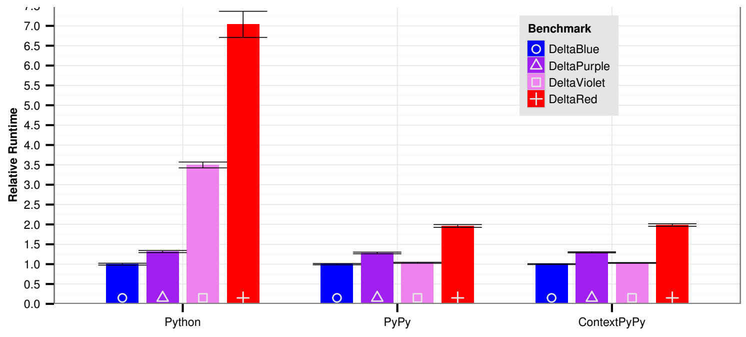

Figure 3. Results of DELTABLUE. Relative execution time of Delta. . . benchmarks on ContextPy on Python and PyPy; and ContextPyPy. Normalized to DeltaBlue. Lower is better. For raw numbers see Table A.1.

over DeltaBlue is comparable across all three implementation/platform combinations. However, the mere presence of layered methods (DeltaViolet) has a comparatively high impact on Python, with up to 3× slowdown. At the same time, virtually no overhead is present on PyPy for DeltaViolet. The overhead of activated layers (DeltaRed) is quite severe on Python, ranging about 5× of DeltaPurple, whereas on PyPy the slowdown is less than 50 % (≈ 1.4×). ContextPyPy performs virtually the same as ContextPy on PyPy.

Discussion (*DELTABLUE*) Being a less “micro” benchmark, DELTABLUE reinforces the impression of COP 09 a that the meta-tracing JIT compiler of PyPy is effective at eliminating the overhead of sideways composition. Especially the no-layers-activated case having no overhead seems important for wider adoption. Seeing COP 09 a and COP 09 b, the indifference between ContextPy on PyPy and ContextPyPy is unexpected, both the missing speedup from COP 09 a and the missing slowdown from COP 09 b. However, the slowdown of COP 09 b is neither present for ContextPy on PyPy, suggesting that ContextPyPy’s *promote* approach could be nevertheless viable to use.

6. Related Work

Other implementations of context-oriented programming have tried to optimize their implementations using traditional compiler and optimization techniques. However, while they can only reduce the performance overhead in some cases, they are sometimes difficult to apply and increase the complexity of the system. A common optimization approach is to shift the performance impact to the layer activation time under the assumption the changes in layer composition are comparatively rarer than execution of layered methods. This approach can drastically limit the performance impact of layers when the composition changes rarely, but at the cost of reduced performance for applications where the layer composition may change frequently.

Instances where this optimization is used are ContextAmber, Elektra, and cj. ContextAmber [17] optimistically flattens layered methods when the layer composition changes to achieve near native performance during execution. An extension to the C++ configuration management system Elektra [14] make use of extensive code generation and caching of the active layer composition to minimize the performance impact of running with active layers in a tight loop. The cj [15, 16] system that implements COP on top of the delMDSOC virtual machine model. This model is well suited towards multi-dimensional dispatch, and thus a COP implementation on top of it achieves good performance in this case. However, as with the other two approaches, switching layers becomes more expensive as a result.

Another approach at optimizing COP that is closer to the work presented here is to use facilities of general purpose VMs directly. One such facility is Java’s INVOKEDYNAMIC instruction that allows language extensions to implement new lookup semantics for the Java VM. However, prior work [1] indicates that this extension point may provide only minimally improved performance compared to an implementation using language-level caching facilities, albeit using less code and a simplified architecture.

7. Conclusion and Future Work

Our first results for using meta-tracing JIT optimizations to reduce the performance impact of COP are promising for micro benchmarks. The meta-tracing JIT compiler of PyPy appears to be effective at eliminating the overhead of sideways composition for method lookup. Our results also show that a few, carefully placed hints can help the runtime to improve this effect, too.

For future work we have to further investigate how careful one has to be in placing those hints and how the performance behavior changes with larger applications. Our results with a larger benchmark indicate that the approach is viable, but so far the performance is less than we had hoped for. Nonetheless the fact that we see no overhead with our approach for executing layered methods when no or only one layer is active and the layer composition does not change already shows that COP can be enabled in a language without paying a performance penalty. We think this result in itself is important for to argue for wider adoption of sideways composition mechanisms like COP.

Acknowledgments

We gratefully acknowledge the financial support of HPI's Research School and the Hasso Plattner Design Thinking Research Program (HPDTRP). LispWorks Ltd. kindly provided an evaluation license of LispWorks® 64-bit for Mac OS X for testing purposes. We greatly appreciate the support of Carl Friedrich Bolz with the ContextPy and ContextPyPy implementations.

We want to thank the anonymous reviewers for their feedback and suggestions.

References

[1] M. Appeltauer, M. Haupt, and R. Hirschfeld. "Layered method dispatch with INVOKEDYNAMIC: an implementation study". In: *Proc. Workshop on COP*. ACM. 2010, p. 4.

[2] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. "A Comparison of Context-oriented Programming Languages". In: *Proc. Workshop on COP*. Genova, Italy: ACM, 2009, pp. 1–6.

[3] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *ACM SIGPLAN Notices* 35.5 (2000), pp. 1–12.

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. "Runtime Feedback in a Meta-tracing JIT for Efficient Dynamic Languages". In: *Proc. ICOOOLPS*. Lancaster, United Kingdom: ACM, 2011, 9:1–9:8.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. "Tracing the Meta-level: PyPy's Tracing JIT Compiler". In: *Proc. ICOOOLPS*. Genova, Italy: ACM, 2009, pp. 18–25.

[6] A. C. Davison and D. V. Hinkley. "Confidence Intervals". In: *Bootstrap Methods and Their Application*. Cambridge, 1997. Chap. 5.

[7] T. Felgentreff. "A Tool Building Example: Layers as Source Code Packages". Demo given at 5th International Workshop on Context-Oriented Programming, Montpellier, France. July 2, 2013.

[8] B. N. Freeman-Benson and J. Maloney. "The DeltaBlue algorithm: An incremental constraint hierarchy solver". In: *Proc. 8th Intl. Conf. on Computers and Communications*. IEEE. 1989, pp. 538–542.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. "Context-oriented Programming". In: *Journal of Object Technology* 7.3 (2008), pp. 125–151.

[10] M. Hölttä. *Crankshafting from the ground up*. Tech. rep. Google, Aug. 2013.

[11] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. "An open implementation for context-oriented layer composition in ContextJS". In: *Sci. Comput. Program.* 76.12 (2011), pp. 1194–1209.

[12] J. Lincke and R. Hirschfeld. "Scoping changes in self-supporting development environments using context-oriented programming". In: *Proc. Workshop on COP*. ACM. June 2012, pp. 2–10.

[13] M. Paleczny, C. A. Vick, and C. Click. "The Java HotSpot™ Server Compiler". In: *Proc. Symp. Java™ Virtual Machine Research and Technology*. Vol. 1. JVM'01. Monterey, California: USENIX Association, Apr. 24, 2001.

[14] M. Raab and F. Puntigam. "Program Execution Environments As Contextual Values". In: *Proc. Workshop on COP*. Uppsala, Sweden: ACM, 2014, 8:1–8:6.

[15] H. Schippers, M. Haupt, and R. Hirschfeld. "An implementation substrate for languages composing modularized crosscutting concerns". In: *Proc. SAC*. ACM. 2009, pp. 1944–1951.

[16] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. "Delegation-based semantics for modularizing crosscutting concerns". In: *ACM Sigplan Notices*. Vol. 43. 10. ACM. 2008, pp. 525–542.

[17] M. Springer, J. Lincke, and R. Hirschfeld. "Efficient Layered Method Execution in ContextAmber". In: *Proc. Workshop on COP*. ACM. 2015, p. 5.

A. Comprehensive Results

Table A.1. DeltaBlue. Execution time for each benchmark in milliseconds with confidence intervals for a 95 % confidence level. Lower is better.

Benchmark	Python	PyPy	ContextPyPy
DeltaBlue	10 269 ± 185	2368 ± 30	2354 ± 21
DeltaPurple	13 529 ± 171	3036 ± 36	3054 ± 23
DeltaViolet	35 924 ± 541	2451 ± 14	2427 ± 9
DeltaRed	72 331 ± 3509	4646 ± 77	4675 ± 73

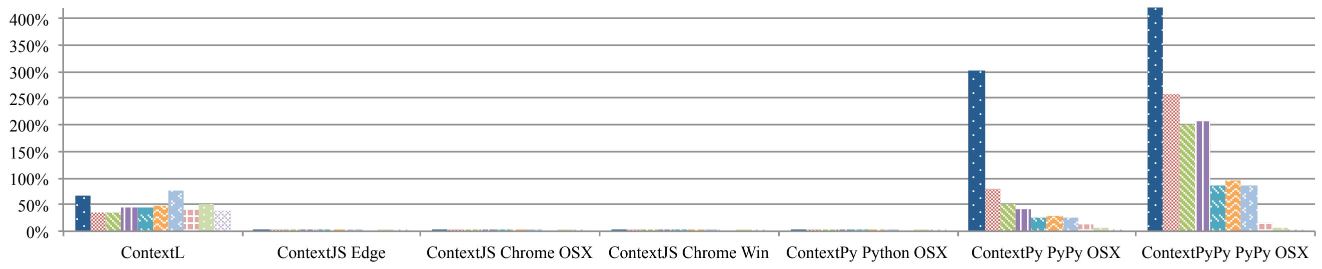

Figure A.1. Overview for COP09a. Relative throughput of method execution in COP implementations with (each left to right) 0 to 10 layers normalized to the respective non-layered workload. Higher is better.

Table A.2. COP09a. Relative throughput of method execution in COP implementations with 0 to 10 layers normalized to the respective non-layered workload. Higher is better.

	ContextL	ContextJS Edge	ContextJS Chrome OSX	ContextJS Chrome Win	ContextPy Python OSX	ContextPy PyPy OSX	ContextPyPy PyPy OSX
no activate layer	67.17 %	0.04 %	0.13 %	0.05 %	4.20 %	347.33 %	441.73 %
1 active layer	35.90 %	0.09 %	0.09 %	0.03 %	4.11 %	72.76 %	243.82 %
2 active layer	36.66 %	0.08 %	0.07 %	0.03 %	3.86 %	47.07 %	244.15 %
3 active layer	43.75 %	0.10 %	0.06 %	0.02 %	3.80 %	37.48 %	196.92 %
4 active layer	44.90 %	0.08 %	0.05 %	0.04 %	3.65 %	27.90 %	86.65 %
5 active layer	47.81 %	0.12 %	0.05 %	0.05 %	3.58 %	28.67 %	94.42 %
6 active layer	77.95 %	0.22 %	0.05 %	0.05 %	3.45 %	26.40 %	90.14 %
7 active layer	42.77 %	0.18 %	0.06 %	0.05 %	3.42 %	12.38 %	16.06 %
8 active layer	49.87 %	0.18 %	0.06 %	0.05 %	3.43 %	5.17 %	5.35 %
9 active layer	37.90 %	0.18 %	0.06 %	0.06 %	3.50 %	4.42 %	4.51 %

Table A.3. COP09b. Relative throughput of layer activation in COP implementations with 0 to 10 layers normalized to the respective non-layered workload. Higher is better.

	ContextL	ContextJS Edge	ContextJS Chrome OSX	ContextJS Chrome Win	ContextPy Python OSX	ContextPy PyPy OSX	ContextPyPy PyPy OSX
no activate layer	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
1 active layer	74.97 %	91.56 %	50.00 %	71.13 %	89.22 %	89.59 %	80.16 %
2 active layer	66.25 %	67.17 %	50.00 %	52.01 %	80.41 %	61.37 %	21.88 %
3 active layer	57.18 %	63.70 %	25.00 %	38.59 %	72.23 %	43.84 %	13.81 %
4 active layer	22.56 %	53.07 %	25.00 %	29.40 %	66.51 %	34.79 %	10.32 %
5 active layer	24.72 %	45.31 %	12.50 %	22.74 %	61.59 %	30.19 %	8.08 %

Table A.4. Raw numbers for the comparison benchmarks (COP 09 a & COP 09 b). *ops* is number of operations (higher is better), *time (s)* is time in seconds (lower is better), *ops / s* is number of operations per second (higher is better).

		ContextL			ContextJS Edge			ContextJS Chrome OSX			ContextJS Chrome Win			ContextPy Python OSX			ContextPy PyPy OSX			ContextPyPy PyPy OSX		
		ops	time (s)	ops / s	ops	time (s)	ops / s	ops	time (s)	ops / s	ops	time (s)	ops / s	ops	time (s)	ops / s	ops	time (s)	ops / s	ops	time (s)	ops / s
NoLayer	1	$8.19 \cdot 10^{8}$	6.96	$1.18 \cdot 10^{8}$	$8.39 \cdot 10^{8}$	7.72	$1.09 \cdot 10^{8}$	$3.36 \cdot 10^{9}$	9.56	$3.51 \cdot 10^{8}$	$3.36 \cdot 10^{9}$	9.82	$3.42 \cdot 10^{8}$	$2.56 \cdot 10^{7}$	5.52	$4.64 \cdot 10^{6}$	$8.19 \cdot 10^{8}$	8.10	$1.01 \cdot 10^{8}$	$8.19 \cdot 10^{8}$	8.50	$9.64 \cdot 10^{7}$
	2	$4.10 \cdot 10^{8}$	5.21	$7.86 \cdot 10^{7}$	$8.39 \cdot 10^{8}$	8.26	$1.02 \cdot 10^{8}$	$3.36 \cdot 10^{9}$	9.60	$3.49 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	5.11	$3.28 \cdot 10^{8}$	$2.56 \cdot 10^{7}$	9.09	$2.82 \cdot 10^{6}$	$8.19 \cdot 10^{8}$	9.17	$8.94 \cdot 10^{7}$	$8.19 \cdot 10^{8}$	9.04	$9.07 \cdot 10^{7}$
	3	$4.10 \cdot 10^{8}$	6.41	$6.39 \cdot 10^{7}$	$8.39 \cdot 10^{8}$	8.90	$9.43 \cdot 10^{7}$	$3.36 \cdot 10^{9}$	9.59	$3.50 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	5.34	$3.14 \cdot 10^{8}$	$1.28 \cdot 10^{7}$	6.00	$2.13 \cdot 10^{6}$	$4.10 \cdot 10^{8}$	5.12	$8.00 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	5.28	$7.76 \cdot 10^{7}$
	4	$4.10 \cdot 10^{8}$	8.35	$4.90 \cdot 10^{7}$	$4.19 \cdot 10^{8}$	5.80	$7.24 \cdot 10^{7}$	$3.36 \cdot 10^{9}$	9.58	$3.50 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	5.19	$3.23 \cdot 10^{8}$	$1.28 \cdot 10^{7}$	7.68	$1.67 \cdot 10^{6}$	$4.10 \cdot 10^{8}$	5.81	$7.05 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	5.89	$6.96 \cdot 10^{7}$
	5	$4.10 \cdot 10^{8}$	9.98	$4.10 \cdot 10^{7}$	$4.19 \cdot 10^{8}$	5.33	$7.87 \cdot 10^{7}$	$3.36 \cdot 10^{9}$	9.62	$3.49 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	5.86	$2.86 \cdot 10^{8}$	$1.28 \cdot 10^{7}$	9.04	$1.42 \cdot 10^{6}$	$4.10 \cdot 10^{8}$	6.61	$6.19 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	6.54	$6.26 \cdot 10^{7}$
	6	$2.05 \cdot 10^{8}$	5.98	$3.43 \cdot 10^{7}$	$4.19 \cdot 10^{8}$	9.75	$4.30 \cdot 10^{7}$	$1.68 \cdot 10^{9}$	5.24	$3.20 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	6.69	$2.51 \cdot 10^{8}$	$6.40 \cdot 10^{6}$	5.25	$1.22 \cdot 10^{6}$	$4.10 \cdot 10^{8}$	7.56	$5.42 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	7.65	$5.35 \cdot 10^{7}$
	7	$2.05 \cdot 10^{8}$	10.78	$1.90 \cdot 10^{7}$	$2.10 \cdot 10^{8}$	9.13	$2.30 \cdot 10^{7}$	$1.68 \cdot 10^{9}$	6.22	$2.70 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	8.96	$1.87 \cdot 10^{8}$	$6.40 \cdot 10^{6}$	5.81	$1.10 \cdot 10^{6}$	$4.10 \cdot 10^{8}$	8.33	$4.92 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	8.63	$4.75 \cdot 10^{7}$
	8	$1.02 \cdot 10^{8}$	5.02	$2.04 \cdot 10^{7}$	$2.10 \cdot 10^{8}$	8.11	$2.59 \cdot 10^{7}$	$1.68 \cdot 10^{9}$	6.98	$2.40 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	9.38	$1.79 \cdot 10^{8}$	$6.40 \cdot 10^{6}$	6.59	$9.72 \cdot 10^{5}$	$4.10 \cdot 10^{8}$	9.51	$4.31 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	9.49	$4.32 \cdot 10^{7}$
	9	$1.02 \cdot 10^{8}$	6.23	$1.64 \cdot 10^{7}$	$2.10 \cdot 10^{8}$	8.75	$2.40 \cdot 10^{7}$	$1.68 \cdot 10^{9}$	8.12	$2.07 \cdot 10^{8}$	$8.39 \cdot 10^{8}$	5.83	$1.44 \cdot 10^{8}$	$6.40 \cdot 10^{6}$	7.35	$8.71 \cdot 10^{5}$	$2.05 \cdot 10^{8}$	5.01	$4.09 \cdot 10^{7}$	$2.05 \cdot 10^{8}$	5.24	$3.91 \cdot 10^{7}$
	10	$1.02 \cdot 10^{8}$	6.19	$1.65 \cdot 10^{7}$	$2.10 \cdot 10^{8}$	9.54	$2.20 \cdot 10^{7}$	$1.68 \cdot 10^{9}$	8.89	$1.89 \cdot 10^{8}$	$8.39 \cdot 10^{8}$	6.81	$1.23 \cdot 10^{8}$	$6.40 \cdot 10^{6}$	8.26	$7.75 \cdot 10^{5}$	$2.05 \cdot 10^{8}$	5.33	$3.85 \cdot 10^{7}$	$2.05 \cdot 10^{8}$	5.47	$3.75 \cdot 10^{7}$
WithLayer	0	$4.10 \cdot 10^{8}$	5.18	$7.90 \cdot 10^{7}$	$4.10 \cdot 10^{5}$	9.14	$4.48 \cdot 10^{4}$	$3.28 \cdot 10^{6}$	7.00	$4.68 \cdot 10^{5}$	$3.28 \cdot 10^{6}$	17.51	$1.87 \cdot 10^{5}$	$3.20 \cdot 10^{6}$	16.43	$1.95 \cdot 10^{5}$	$3.28 \cdot 10^{9}$	9.33	$3.51 \cdot 10^{8}$	$3.28 \cdot 10^{9}$	7.70	$4.26 \cdot 10^{8}$
	1	$2.05 \cdot 10^{8}$	7.26	$2.82 \cdot 10^{7}$	$8.19 \cdot 10^{5}$	8.58	$9.55 \cdot 10^{4}$	$1.64 \cdot 10^{6}$	5.03	$3.26 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	8.15	$1.01 \cdot 10^{5}$	$3.20 \cdot 10^{6}$	27.65	$1.16 \cdot 10^{5}$	$4.10 \cdot 10^{8}$	6.30	$6.50 \cdot 10^{7}$	$1.64 \cdot 10^{9}$	7.41	$2.21 \cdot 10^{8}$
	2	$2.05 \cdot 10^{8}$	8.74	$2.34 \cdot 10^{7}$	$4.10 \cdot 10^{5}$	5.35	$7.66 \cdot 10^{4}$	$1.64 \cdot 10^{6}$	6.34	$2.59 \cdot 10^{5}$	$4.10 \cdot 10^{5}$	5.00	$8.19 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	38.87	$8.23 \cdot 10^{4}$	$2.05 \cdot 10^{8}$	5.44	$3.76 \cdot 10^{7}$	$1.64 \cdot 10^{9}$	8.65	$1.89 \cdot 10^{8}$
	3	$2.05 \cdot 10^{8}$	9.54	$2.15 \cdot 10^{7}$	$4.10 \cdot 10^{5}$	5.76	$7.11 \cdot 10^{4}$	$1.64 \cdot 10^{6}$	7.42	$2.21 \cdot 10^{5}$	$4.10 \cdot 10^{5}$	6.09	$6.73 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	50.55	$6.33 \cdot 10^{4}$	$2.05 \cdot 10^{8}$	7.75	$2.64 \cdot 10^{7}$	$8.19 \cdot 10^{8}$	5.98	$1.37 \cdot 10^{8}$
	4	$1.02 \cdot 10^{8}$	5.56	$1.84 \cdot 10^{7}$	$4.10 \cdot 10^{5}$	6.41	$6.39 \cdot 10^{4}$	$1.64 \cdot 10^{6}$	8.58	$1.91 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	6.49	$1.26 \cdot 10^{5}$	$3.20 \cdot 10^{6}$	61.86	$5.17 \cdot 10^{4}$	$1.02 \cdot 10^{8}$	5.93	$1.73 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	7.55	$5.43 \cdot 10^{7}$
	5	$1.02 \cdot 10^{8}$	6.25	$1.64 \cdot 10^{7}$	$4.10 \cdot 10^{5}$	7.63	$5.37 \cdot 10^{4}$	$1.64 \cdot 10^{6}$	9.93	$1.65 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	7.19	$1.14 \cdot 10^{5}$	$3.20 \cdot 10^{6}$	73.46	$4.36 \cdot 10^{4}$	$1.02 \cdot 10^{8}$	6.60	$1.55 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	8.10	$5.06 \cdot 10^{7}$
	6	$1.02 \cdot 10^{8}$	6.92	$1.48 \cdot 10^{7}$	$4.10 \cdot 10^{5}$	8.14	$5.03 \cdot 10^{4}$	$8.19 \cdot 10^{5}$	5.53	$1.48 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	8.12	$1.01 \cdot 10^{5}$	$3.20 \cdot 10^{6}$	84.18	$3.80 \cdot 10^{4}$	$1.02 \cdot 10^{8}$	7.88	$1.30 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	9.57	$4.28 \cdot 10^{7}$
	7	$1.02 \cdot 10^{8}$	11.73	$8.73 \cdot 10^{6}$	$4.10 \cdot 10^{5}$	8.64	$4.74 \cdot 10^{4}$	$8.19 \cdot 10^{5}$	6.12	$1.34 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	9.04	$9.06 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	96.26	$3.32 \cdot 10^{4}$	$5.12 \cdot 10^{7}$	9.61	$5.33 \cdot 10^{6}$	$5.12 \cdot 10^{7}$	7.38	$6.94 \cdot 10^{6}$
	8	$5.12 \cdot 10^{7}$	6.25	$8.20 \cdot 10^{6}$	$4.10 \cdot 10^{5}$	9.55	$4.29 \cdot 10^{4}$	$8.19 \cdot 10^{5}$	6.61	$1.24 \cdot 10^{5}$	$4.10 \cdot 10^{5}$	5.21	$7.86 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	107.12	$2.99 \cdot 10^{4}$	$1.28 \cdot 10^{7}$	6.06	$2.11 \cdot 10^{6}$	$1.28 \cdot 10^{7}$	6.12	$2.09 \cdot 10^{6}$
	9	$5.12 \cdot 10^{7}$	8.17	$6.27 \cdot 10^{6}$	$2.05 \cdot 10^{5}$	5.25	$3.90 \cdot 10^{4}$	$8.19 \cdot 10^{5}$	7.29	$1.12 \cdot 10^{5}$	$4.10 \cdot 10^{5}$	5.38	$7.62 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	118.06	$2.71 \cdot 10^{4}$	$1.28 \cdot 10^{7}$	7.54	$1.70 \cdot 10^{6}$	$1.28 \cdot 10^{7}$	7.58	$1.69 \cdot 10^{6}$
Activation	0	$1.02 \cdot 10^{8}$	5.67	$1.81 \cdot 10^{7}$	$2.05 \cdot 10^{5}$	9.82	$2.09 \cdot 10^{4}$	$8.19 \cdot 10^{5}$	9.05	$9.05 \cdot 10^{4}$	$4.10 \cdot 10^{5}$	8.77	$4.67 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	85.01	$3.76 \cdot 10^{4}$	$1.02 \cdot 10^{8}$	6.22	$1.65 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	5.66	$7.24 \cdot 10^{7}$
	1	$1.02 \cdot 10^{8}$	7.56	$1.35 \cdot 10^{7}$	$1.02 \cdot 10^{5}$	5.36	$1.91 \cdot 10^{4}$	$4.10 \cdot 10^{5}$	6.66	$6.15 \cdot 10^{4}$	$2.05 \cdot 10^{5}$	6.17	$3.32 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	95.28	$3.36 \cdot 10^{4}$	$1.02 \cdot 10^{8}$	6.95	$1.47 \cdot 10^{7}$	$4.10 \cdot 10^{8}$	7.06	$5.80 \cdot 10^{7}$
	2	$1.02 \cdot 10^{8}$	8.56	$1.20 \cdot 10^{7}$	$1.02 \cdot 10^{5}$	7.31	$1.40 \cdot 10^{4}$	$4.10 \cdot 10^{5}$	9.30	$4.41 \cdot 10^{4}$	$2.05 \cdot 10^{5}$	8.43	$2.43 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	105.72	$3.03 \cdot 10^{4}$	$5.12 \cdot 10^{7}$	5.07	$1.01 \cdot 10^{7}$	$1.02 \cdot 10^{8}$	6.47	$1.58 \cdot 10^{7}$
	3	$1.02 \cdot 10^{8}$	9.91	$1.03 \cdot 10^{7}$	$1.02 \cdot 10^{5}$	7.70	$1.33 \cdot 10^{4}$	$2.05 \cdot 10^{5}$	6.36	$3.22 \cdot 10^{4}$	$1.02 \cdot 10^{5}$	5.68	$1.80 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	117.70	$2.72 \cdot 10^{4}$	$5.12 \cdot 10^{7}$	7.10	$7.21 \cdot 10^{6}$	$1.02 \cdot 10^{8}$	10.24	$10.00 \cdot 10^{6}$
	4	$2.56 \cdot 10^{7}$	6.28	$4.08 \cdot 10^{6}$	$1.02 \cdot 10^{5}$	9.25	$1.11 \cdot 10^{4}$	$2.05 \cdot 10^{5}$	8.59	$2.39 \cdot 10^{4}$	$1.02 \cdot 10^{5}$	7.46	$1.37 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	127.82	$2.50 \cdot 10^{4}$	$5.12 \cdot 10^{7}$	8.94	$5.72 \cdot 10^{6}$	$5.12 \cdot 10^{7}$	6.86	$7.47 \cdot 10^{6}$
	5	$5.12 \cdot 10^{7}$	11.47	$4.47 \cdot 10^{6}$	$5.12 \cdot 10^{4}$	5.42	$9.45 \cdot 10^{3}$	$1.02 \cdot 10^{5}$	5.63	$1.82 \cdot 10^{4}$	$1.02 \cdot 10^{5}$	9.64	$1.06 \cdot 10^{4}$	$3.20 \cdot 10^{6}$	138.02	$2.32 \cdot 10^{4}$	$2.56 \cdot 10^{7}$	5.15	$4.97 \cdot 10^{6}$	$5.12 \cdot 10^{7}$	8.75	$5.85 \cdot 10^{6}$