2012 10th International Conference on Creating, Connecting and Collaborating through Computing

Efficient Layer Activation in ContextJS

Robert Krahn*, Jens Lincke*, and Robert Hirschfeld*
Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany
http://www.hpi.uni-potsdam.de/swa
*Email: {firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—Context-oriented programming (COP) describes language extensions for modularizing behavioral or structural variations that are to be composed at run-time. Different COP infrastructures and implementations offer several strategies for scoping, activation, and deactivation of such compositional units. Often, the mechanisms employed cause substantial execution overhead. In this paper we present an optimization technique for ContextJS—our COP extension to JavaScript—that can significantly reduce this overhead to run context-aware code efficiently.**

I. Introduction

ContextJS is a context-oriented programming (COP) extension with JavaScript as its host language. It is implemented as a library using the meta-programming facilities of JavaScript and does not require VM modifications. Similar to other COP extensions to dynamically typed host languages, ContextJS intercepts the runtime lookup mechanism for invoking behavior variations. So far, this approach comes with a significant impact on run-time performance. Micro-benchmarks [1] confirmed a performance loss of more than 99% compared to un-instrumented code.

Although this overhead does not affect the entire system but only methods made aware of behavioral variations, we experienced that extensive use of ContextJS can make an interactive system such as Lively Kernel [2], [3] noticeably slower.

In the Lively Kernel runtime and development environment we employ ContextJS for various applications. We can group those applications broadly in the following three categories:

1) Global modularization and deployment. ContextJS layers are used to both modularize and deploy/un-deploy structural and behavioral variations.
2) Object-specific behavior. ContextJS is used to selectively add experimental code on a per-object basis during development [4]. In self-sustaining systems like Lively Kernel, this allows to keep changes separate so that newly added erroneous code can be explored and removed easily. In such systems, this is especially important if programming tools like an editor might depend on the code that is being changed.
3) Tracing of system behavior. ContextJS provides infrastructure to selectively instrument and trace method activation and deactivation within the dynamic extent of a particular execution context [5]. It is often used to separate tracing target and tracer logic so that self-referentiality is avoided.

This work has been supported by the HPI-Stanford Design Thinking Research Program.

COP variations defined globally or on a per-object basis are rather static since they do not depend on propagation within dynamic extents at run-time but rather resemble partial class and method definitions.

In dynamic extend-based scoping the system's layer composition changes often, but for a single method the layer composition may be the same for consecutive calls, because the layers are (de-)activated in the same pattern. With that, layer compositions for specific objects will not change most of the time.

In this paper we present an optimization technique for layer composition and lookup in ContextJS that can significantly reduce method activation overhead. By using the meta-programming facilities of JavaScript and Lively Kernel, we reduce a composition of multiple related partial method definitions to a single method. For all compositions that will not change anymore, this compilation step removes all run-time composition machinery and with that all its run-time overhead.

This improvement does not restrict the expressibility or change the semantics of ContextJS. Especially dynamic-extent and object-based activations are optimized. To be able to react to layer composition changes, a validation check is added to the generated method. When the layer composition check indicates that the layer composition, which the method was compiled for, does not correspond to the current one, it will invalidate the current optimization leading to the generation of a new one.

The remainder of the paper is organized as follows. Section 2 gives a short overview of Context-oriented programming and the mechanisms used to define and activate behavior variations. Section 3 introduces our optimization mechanism. In Section 4 we present our implementation and discuss design decisions and implementation options. Section 5 evaluates our approach. Section 6 discusses the related work and section 7 summarizes the paper and mentions topics for future work.

978-0-7695-4672-8/12 $26.00 © 2012 IEEE
DOI 10.1109/C5.2012.20

IEEE computer society

```
Object.subclass('Person', {
  initialize: function(fullName) {
    this.fullName = fullName;
  },
  toString: function(beShort) {
    return beShort ?
      this.fullName : 'Person('+this.fullName +')';
  }
});

cop.create('EmployerLayer')
.refineClass(Person, {
  toString: function(beShort) {
    return cop.proceed(beShort) + ' (HPI)';
  }
});

cop.create('AddressLayer')
.refineClass(Person, {
  address: function() {
    return 'Prof.-Dr.-Helmert Str.';
  },
  toString: function(beShort) {
    return cop.proceed(beShort) +
      ' (' + this.address() + ')';
  }
});
```

Listing 1. Definition of Person class and two layers that refine the behavior of its *toString* method.

```
var obj = new Person('John Doo');
obj.toString(true);
// returns "Person(John Doo)"
AddressLayer.beGlobal()
obj.toString(true);
// returns
// "John Doo (Prof.-Dr.-Helmert Str.)"
cop.withLayers(
    [EmployerLayer],
    function() { return obj.toString() });
// returns
// "Person(John Doo) (Prof.-Dr.-Helmert Str.) (HPI)"
```

Listing 2. Output of *Person>>toString* for different layer compositions

II. Layer Composition and Method Activation in COP

In this section, we give a brief introduction to COP [6] and the method lookup and activation mechanism found in ContextJS [4].

A. Activation of Behavior Variations

In the COP execution model, the program behavior depends upon the context in which it is run. For integrating behavior variations into the program COP introduces the layer construct. Layers are a modularization concept that encapsulates behaviors distributed over several objects or classes. The behavior variations are defined inside layers by means of partial method definitions. We use the term layered method for the composition of at least one partial method and a base method. Class or object slots are called layer-aware when there is at least one layer definition with a partial method for that slot, even if the layer is not activated.

The layered method compositions are based on the compositions of layers. Layer compositions are created at runtime by (de-)activating layers, e.g. in the dynamic extent of a control flow or globally. When layers are activated at runtime and a message send reaches a layer-aware slot, the object-oriented method lookup is extended by a sideways lookup. This lookup is called layer composition and gathers corresponding partial methods of active layers.

B. Layer Composition in ContextJS

ContextJS is a typical dynamic language COP implementation [1], because it is implemented only using the meta-programming facilities of its host language JavaScript.

Listing 1 shows a class definition of a *Person* class[1]. Its *toString* method returns a stringified representation and can be parameterized using a boolean parameter. Two layers are defined afterwards that extend the behavior of *Person>>toString*. The *EmploymentLayer* and *AddressLayer* define partial methods to append to the result of preceding partial method activations.

Listing 2 shows how the class is instantiated and how the *Person* instances returns different results depending on the activated layers.

In ContextJS the slots of a class or object are made layer-aware at layer definition time. This means that for all layered methods and properties a wrapper method is installed. In the example the *toString* slot of the *Person* is made layer-aware when the first layer is defined. The original implementation is removed from the slot and replaced by a generic wrapper method. Upon method activation this wrapper then computes the current layer composition and performs the lookup and activation of the partial methods belonging to the currently activated layers.

Figure 1 illustrates the layer composition process for the *toString* message call on line 10 in listing 2. While the address layer is globally and employment layer dynamically activated, *Person>>toString* is called (step 1). This activates the COP method wrapper. Upon activation the wrapper accesses the active layers and creates the corresponding *layer composition*, i.e. the wrapper gathers the partial methods defined in *EmploymentLayer* and *AddressLayer* as well as the base method (step 2). The partial methods are ordered accordingly to the activation order of their layers on a stack.

During the subsequent steps (3.1 - 5.1) the partial methods are activated. When a proceed call is encountered the control is handed back to the ContextJS meta level to fetch the next partial method from the stack that is activated in turn. This process continues until either a partial method does not proceed or the base method is reached. When that happens (step 6) the control is handed back to calling methods until the method wrapper is reached. The wrapper returns the result of the method activation to the call site.

[1]Using the Lively Kernel class system [2]

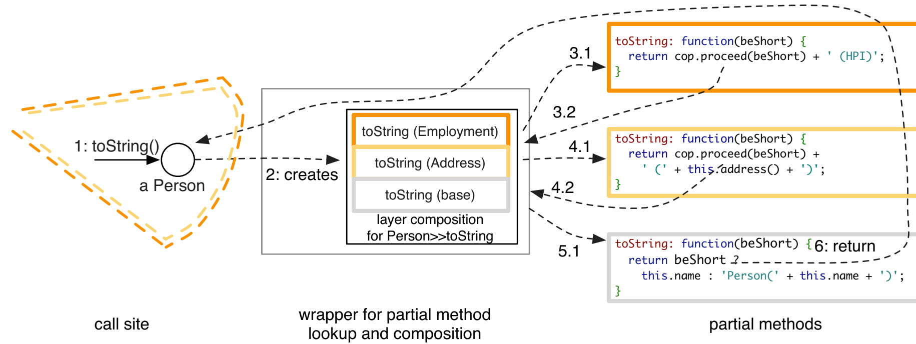

Fig. 1. Unoptimized method dispatch in ContextJS

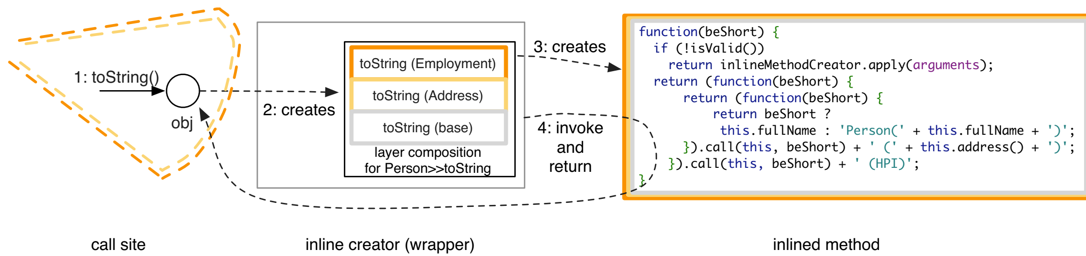

Fig. 2. Optimized ContextJS method dispatch on first call

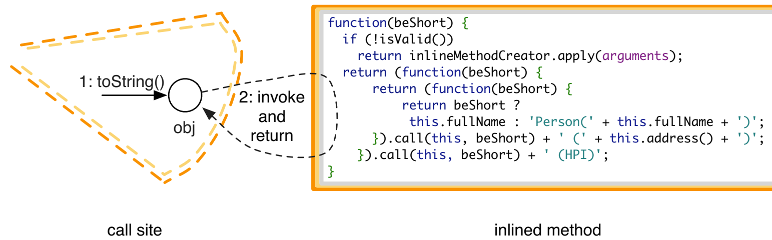

Fig. 3. Optimized ContextJS on subsequent calls

The lookup mechanism depicted here is used on every activation of a layered method. In the next section we will discuss an approach for optimizing this process.

III. Partial Method Inlining and Composition Caching

Since recorded runtime information for our COP applications showed that layer composition changes do not happen often for most objects, the default layer composition mechanism of ContextJS computes most of the time redundant compositions. In this section we will present an adapted layer composition and lookup mechanism that will only re-compute layer compositions when necessary. Furthermore, we denote what parts of the method composition and lookup mechanism can be optimized.

```
1  function toString(beShort) {
2    if (!currentLayerCompositionIsValid())
3      return inlineMethodCreator.apply(
4        arguments);
5    return (function(beShort) {
6      return (function(beShort) {
7        return beShort ?
8          this.fullName : 'Person(' +
9            this.fullName + ')';
10     }).call(this, beShort) + ' (' +
11       this.address() + ')';
12   }).call(this, beShort) + ' (HPI)';
13 }
```

Listing 3. The result of automatically inlining the *toString* method with our approach.

A. Inlining

For a given layer composition, the partial methods and the necessary COP infrastructure code are combined into a

single method. Each generated method corresponds to a layer composition. After its creation the generated method is used for all subsequent activations as long as the active layer composition is similar to the method's corresponding layer composition.

Listing 3 shows the inlining result of *Person>>toString* for the layer composition: EmploymentLayer, AddressLayer, base. According to the order of activated layers we inline each proceed call with the partial method that the proceed call would activate when executed. We directly embed the partial method and not just its statements so that the inlined source code has its own scope[2]. This also makes the source code transformation simpler since the actual call to the partial method will not be changed and parameters of the inlined partial method do not need to be renamed. The normal JavaScript function call semantic applies.

B. Composition Validation

A validation check is added as the first statement of the generated method. The validation check is the only COP meta-level code that is invoked. The validation check compares the active layer composition to the corresponding layer composition of the generated method and invalidates it if necessary. Invalidation means to remove the generated method from the slot it is installed in and either generate and install a new inlined method or install a common ContextJS method wrapper[3].

Since the validation check is invoked for every method activation its efficiency is critical. We propose the usage of fingerprints [7] for efficient comparison of layer compositions. The fingerprint of a layer composition regarding an object slot can be computed from unique layer identifiers and version information about the partial methods and base method. Given two layer compositions and an object slot their fingerprints should be equal if both layer compositions include the same layers in the same order and, regarding the slot, the partial methods and the base method for that slot are equal. We describe in the next section a possible implementation.

C. Composition Caching

The currently active layer composition must be accessed for both the validation check and for the partial method lookup while generating an inlined method. Since this must happen at least once when activating a layered method, also this operation should be efficient.

Layers can be (de-)activated globally and within the dynamic extent of a control flow. ContextJS also allows object scoping [4], i.e. layers are activated and deactivated depending on the receiver of a message send. Since object layer activations can be completely customized, e.g. be made random, caching them is not possible without further constraints. However, when the receiver object does not implement its own layer computation, only globally and dynamically activated layers have to be considered when computing the active layer composition. When those layers are stored in order of their activation in a collection, e.g. a stack, it is possible to associate a composition cache together with that data structure. The cache can be invalidated whenever the collection changes.

[2]JavaScript only provides function not block scope so a function is required

[3]Both composition mechanisms are interchangeable and can be used at the same time in the running system

IV. Implementation

This section presents our implementation. We will show in detail how the optimization mechanism described previously was applied to ContextJS and discuss problems we encountered.

A. Method Generation

The process of inlining partial methods of a layer composition to create generated methods as described in section 3.3 is the following:

When a partial method is defined and the slot in the refined object or class was not yet made layer-aware, install a method wrapper in that slot. The method wrapper will not handle the partial method dispatch directly but is responsible for creating an inlined function. We designate that method wrapper as *inlinedMethodCreator*.

The first activation triggers the *inlinedMethodCreator* that will then lazily build the inlined method. This involves the following steps:

1) Access the current layer composition. If the object for which the inlined method is built does customize the active layer computation in a special way, the layer composition has to be computed. If not, a cached layer composition can be used. The global collection *cop.LayerStack* holds all dynamically and globally activated layers. When the currently active layers have not changed since a previous composition computation then the composition is cached and can be reused. Otherwise the *cop.LayerStack* has to be iterated and a new layer composition has to be computed from all with and without layer specifications.
2) Compute the fingerprint for the composition.
3) Gather partial methods from methods.
4) Inline partial methods by starting with the first partial method (the partial method that is activated when the layered method is executed) and inline partial methods of the next nesting level. Repeat that until all partial methods and the base method are transformed into source code block. Add the validation check and create a new function out of the transformed source code.
5) Call new inlined method.

For each subsequent method call do:

1) Access fingerprint of current layer composition. Either using *cop.LayerStack* that is also used to store fingerprints together with cached compositions or compute layer composition and compute hash.
2) If the corresponding layer composition is equal to the current layer composition continue running the inlined method. Otherwise remove the inlined method from its

```
var value = 5;
var f = function() { return value + 3 }
  .binds({value: value});
f(); // 8
f.getVarMapping(); // {value: 5}
```

Listing 4. First class closures and closure introspection.

```
var spyObj = {
  m: function() { alert(secret) }
}

cop.create('SecretLayer')
.refineObject(spyObj, {
  m: function() {
    var secret = ...
    cop.proceed();
  },
});

function inlined() {
  ...
  var secret = ...
  (function() { alert(secret) })
    .call(this);
}
```

Listing 5. First class closures and closure introspection.

slot and invoke *inlinedMethodCreator* again. This starts the composition process again.

Figure 2 shows the process of installing the inlined method using the Person example from listing 1 for the layer composition for the layer composition EmploymentLayer, AddressLayer, base. Upon the *toString* message send, the *inlinedMethodCreator* is activated. It inlines the partial methods found in the employment and address layers as well as the base method. When this is done the inlined method is activated and its return value passed back to the call site.

Following method calls with the same layer composition are processed as shown in figure 3. The inlined method gets directly activated and performs the validation check. Since the activated layers have not changed, the method is still valid and the control flow continues normally.

B. Dealing with Closures

Rewriting source code in JavaScript is an interesting challenge. Rewriting means that that existing functions are converted into a string representation that is then parsed and changed. Recompiling can be done by evaluating the function source code so that a function object is the result.

In JavaScript, functions bind values to free variables at definition time and are therefore closures. Once bound, JavaScript provides no means for introspection of the sub-method level[4]. When transforming source code this becomes a problem since bound variables cannot be transferred to rewritten code.

[4]Instrumenting a JavaScript VM would solve this problem but would make our optimization mechanism less general

To provide a solution that allows code rewriting and closures at the same time, we support first class closure objects. These objects can be created alongside functions as is shown in listing 4.

This mechanism allows to rewrite functions even with bound variables. However, this burdens users to explicitly define closures. The Lively Kernel is implemented using objects and classes; closures on a per method level are rarely used so that this was not a drawback up to this point.

C. Source Code Transformations

For the actual source code modifications we used OMeta-based [8] JavaScript parser and AST transformer. However, this approach introduced a lot of overhead for runtime layer inlining.

Since the code transformations are simple — finding and replacing the proceed statement, injecting a statement at the method start — our current implementation uses regular expressions for parsing and rewriting.

A special case that can appear while transforming and that is currently not covered by our implementation is that inlined partial methods with unbound variables can lexically capture variables defined in an outer function block. Listing 5 shows an example. The method *m* of *spyObj* and *SecretLayer* are inlined. In the generated method the inlined base method is able to access the variable *secret*.

Solutions to that problem include applying mechanisms that can be found in hygienic macros [9] or code generators [10]. Alternatively the inline mechanism could be adapted so that it would not directly inline the source code but would replace the *cop.proceed* with hard wired method calls. Thus, partial methods could no longer accidentally access the lexical scope of the generated method.

D. Composition Validation

The efficiency of the validation check is critical for the overall performance characteristics of method inlining since this check is invoked on every call.

Our current approach is the following: Each layer has a unique identifier that consists of its name and a timestamp of its last change, i.e. the point of time one of its partial method were changed, added, or removed. These identifiers are then concatenated for computing the fingerprint of the whole composition. On method inline creation the fingerprint layer composition is stored and later used for comparison when subsequent calls activate the inlined method again.

We use string comparison for the validation check. To improve its efficiency fingerprints should use better hashing or fingerprint mechanism like Rabin fingerprints [7], [11].

V. Evaluation

To evaluate the performance of the new layer inlining approach we run benchmarks used to compare several COP implementations [1]. The benchmarks measure the relative performance of a layered method lookup compared to a non layered method lookup of the host language. The results of

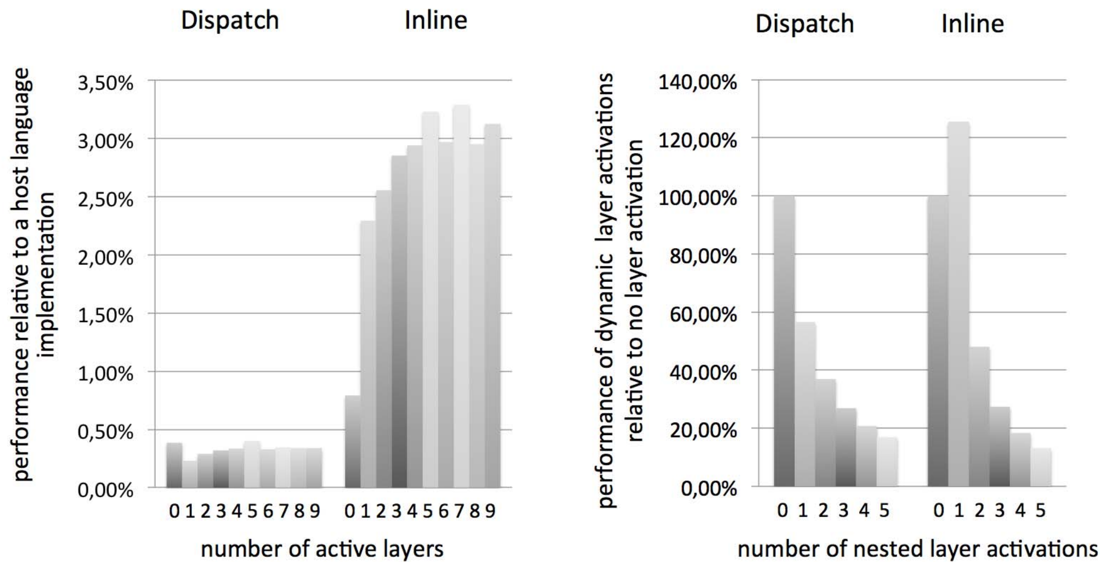

Fig. 4. Micro benchmark results of ContextJS with dispatch and with inline approach. The benchmark source is taken from the COP survey [1].

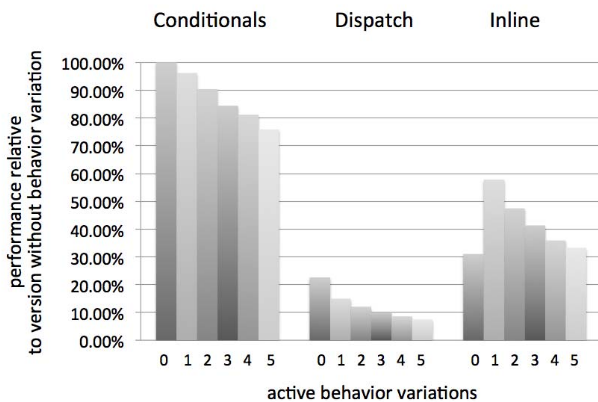

Fig. 5. A more realistic micro benchmark, comparing an Object-oriented version that uses a context object passed in as parameter and if conditional statements to express behavior variations, with two ContextJS versions that use layers to express these behavioral variations.

the micro-benchmarks[5] in Figure 4 show that the inlining approach is up to 10 times faster when compared with dynamic dispatch. However, relative to the not instrumented code the overhead is still very significant since only 1–3% of its performance is reached. Profiling shows that most of the time is spend accessing layer compositions even though compositions are cached. Further optimizations are needed here to get better results.

The relative performance characteristics of running an layer-aware method without any layer activations, compared to 1–5 nested layer activations, is for both approaches similar. Interesting is that we consistently observed that a method with one active layer is executed faster with inlining, than a method without any active layers. This might be the result of using a different path for executing a method without any layers, but further investigation is needed here.

[5]All benchmarks were executed with Google Chrome 12.0.742.91 on a MacBook Pro, Intel Core i7 2 GHz 4 Cores, 4 GB RAM.

```
1 // 1. run 1000 times
2 on=!on;
3 if (on) BenchLayer1.beGlobal()
4 else BenchLayer1.beNotGlobal()
5 benchmarkObj.m('')
6
7 // 2. results
8 // 3 ms with ContextJS dispatch
9 // 557.3 ms with ContextJS inlining
```

Listing 6. Benchmark that alternates the layer composition on every method call.

To measure the performance of our approach we developed a new micro benchmark scenario: Our own benchmarks in Figure 5 try to capture the performance in a more realistic manner. We compare our implementation with a basic implementation that uses additional parameters and if statements to express context-specific behavioral variations. Listing 7 shows the benchmark code. Compared to the first benchmark scenario that used counters, string concatenation is generally slower. Therefore the run-time ratio of the COP and OO version is smaller. The benchmark also reduces the difference between the *inlining* and the *dispatching* approach. Using inlining, ContextJS can achieve 50% of the performance of the hand coded version using *if* constructs. The dispatch approach achieves only 10% to 20% compared to that version.

Since the layer composition is not changed in our bench-

```
1 // 1. Behavioral variations expressed with
      conditionals
2 benchmarkObj = {
3   m: function(result, context) {
4     if (context.l5) result += 'l5'
5     if (context.l4) result += 'l4'
6     if (context.l3) result += 'l3'
7     if (context.l2) result += 'l2'
8     if (context.l1) result += 'l1'
9     result += 'base';
10    return result;
11  },
12 }
13
14 // 2. Behavioral variations expressed with layers
15 benchmarkObj = {
16   m: function(result) {
17     result += 'base'
18     return result
19   },
20 }
21
22 cop.create('BenchLayer1').refineObject(benchmarkObj,
       {
23   m: function(result) {
24     result += 'l1';
25     return cop.proceed(result)
26   }
27 })
28 cop.create('BenchLayer2').refineObject(benchmarkObj,
       {
29   m: function(result) {
30     result += 'l2';
31     return cop.proceed(result)
32   }
33 })
34 // etc ... to BenchLayer5
```

Listing 7. Object and layer definitions that are used in the benchmark of Figure 5. The first version of the benchmark object expresses the behavioral variation with if constructs and a context object that is passed in as a parameter. The second version uses layers to do the same.

mark in Figure 5, we measured how the performance can degenerate if the layer composition changes on every method call. In this extreme case the standard dynamic lookup outperforms the inlining as shown in Listing 6. Because of source code transformations and function generation computation, costs of method inlining are very high. When the layer composition changes every time the dynamic lookup is around 200 times faster. One approach to solve this problem is the introduction of an inlined method cache. The cache stores generated inlined methods that were installed before but invalidated later. When the object is activated with the same layer composition the old methods are fetched from the cache and re-installed so that source code transformations and function creation becomes unnecessary. We implemented that optimization prototypically but have not yet adopted it since layer composition changes do not happen often and had not yet a noticeable performance impact.

VI. Related Work

A common implementation approach to extending object-oriented programming languages with COP concepts is the extension of method dispatch by yet another dimension [6]. Depending on the particular implementation strategy, the performance overhead can be significant. In a benchmark using the popular "figure editor" from the AOP community as an example [12], ContextL[13] has shown that COP extensions can perform competitively [14]. But due to other implementation constraints not all COP extensions can take advantage of this particular optimization.

There are several COP implementations for Java that are each implemented with different approaches. ContextJ [15] uses compiler and preprocessor techniques. This allows for example to directly compile the lookup code into the base method. Other implementations like ContextLogicAJ [16], and ContextJ* [6], an Java5 library based implementation of COP concepts, are more restricted in their implementation possibilities.

The *cj* prototype [17], [18] is an atypical COP implementation. It implements a minimal subset of ContexJ [14], [19] to demonstrate a machine-model of multi-dimensional separation of concerns [20]. In *cj* a layer activation directly changes the method dictionaries of all affected classes. This is a very expensive operation, but it eliminates layer composition at message dispatch time. This approach is slow when layers get often (de-)activated and when a layer refines methods in many places, but it is fast if this does not happen often. In our approach a layer composition change also invalidates all inlined partial method compositions, but this invalidation is only detected and handled lazily, making the actual layer activation fast, but the next layered method executions slow.

Aspect-oriented programming (AOP) languages like AspectJ[21] also change the method dispatch in object-oriented programming. They are implemented on a source code, byte code or the virtual machine level [20]. AspectScript [22] is an AOP implementation for JavaScript. Since it allows for very expressive aspects, the implementation has to instrument every method in the system. This is done by parsing and manipulating the JavaScript source code of the base system. This slows down the whole system by 5-6 times even when no aspects are active. Contrastingly, in COP it can be statically determined which methods are refined by layers. Therefore ContextJS only needs to change those methods. Methods that are not refined run unaffected and with their normal performance.

Inlining of methods is a well-known compiler optimization for statically and dynamically compiled Object-oriented languages. For dynamic inlining in Self [23] it is argued that with the help of type feedback the inlining of messages leads to significant performance improvements because in pure OO languages methods are typically very small and are called very frequently [24]. In Self the inlining of methods is also done at run-time but on the level of the virtual machine (VM). Since our COP message dispatch is handled on the host language level the approach cannot directly used in our implementation. But since inlining of layer compositions in ContextJS makes the execution of layered methods less dynamic, we expect that optimizations in JavaScript VMs can be more often applied as could be with a dynamic dispatch mechanism.

VII. Summary and Outlook

COP language extensions enable behavior variations to be modularized, dynamically activated, and scoped. Our language extension ContextJS implements the COP paradigm in JavaScript and is used in the Lively Kernel environment for various applications. Regular ContextJS dispatch and composition mechanisms introduced a significant overhead for layer-aware code at run-time.

To make ContextJS more efficient, we present an optimization technique for ContextJS to improve run-time layer lookup and composition performance. Based on our observation that layer composition changes per object happen rarely, the optimizations we implemented are targeted at making execution traces fast that rarely include layer composition changes.

We propose the generation of a single method that has all contributing partial and base methods inlined. This removes all meta-level composition code apart from the validation checks in layered methods. Since a validation check still has to be included, we also presented techniques to optimize layer composition access and checks for composition change.

Our evaluation shows noticeable performance improvements. However, there is still a big performance gap to code not instrumented that comes from the validation check that is necessary to recompose layered methods on layer composition changes.

A macro benchmark suite for COP implementations would be of great help to guide further optimization efforts.

References

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A Comparison of Context-oriented Programming Languages," in *International Workshop on Context-Oriented Programming*, ser. COP '09. New York, NY, USA: ACM, 2009, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/1562112.1562118

[2] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen, "The Lively Kernel A Self-supporting System on a Web Page," in *Self-Sustaining Systems*, ser. Lecture Notes in Computer Science, R. Hirschfeld and K. Rose, Eds. Springer Berlin / Heidelberg, 2008, vol. 5146, pp. 31–50, 10.1007/978-3-540-89275-5_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89275-5_2

[3] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz, "Lively Wiki a Development Environment for Creating and Sharing Active Web Content," in *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, ser. WikiSym '09. New York, NY, USA: ACM, 2009, pp. 9:1–9:10. [Online]. Available: http://doi.acm.org/10.1145/1641309.1641324

[4] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld, "An Open Implementation for Context-oriented Layer Composition in ContextJS," *Science of Computer Programming*, vol. 76, no. 12, pp. 1194 – 1209, 2011, special Issue on Software Evolution, Adaptability and Variability. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642310002121

[5] J. Lincke, R. Krahn, and R. Hirschfeld, "Implementing Scoped Method Tracing with ContextJS," in *International Workshop on Context-Oriented Programming*, ser. COP '11. ACM, 2011, to appear.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented Programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, March - April 2008.

[7] A. Z. Broder, *Some applications of Rabin's fingerprinting method*. Springer, 1993, pp. 143–152.

[8] A. Warth and I. Piumarta, "Ometa: an object-oriented language for pattern matching," in *Proceedings of the 2007 symposium on Dynamic languages*, ser. DLS '07. New York, NY, USA: ACM, 2007, pp. 11–19. [Online]. Available: http://doi.acm.org/10.1145/1297081.1297086

[9] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *Proceedings of the 1986 ACM conference on LISP and functional programming*, ser. LFP '86. New York, NY, USA: ACM, 1986, pp. 151–161. [Online]. Available: http://doi.acm.org/10.1145/319838.319859

[10] Y. Smaragdakis and D. Batory, "Scoping Constructs for Software Generators," in *Generative and Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, K. Czarnecki and U. Eisenecker, Eds. Springer Berlin / Heidelberg, 2000, vol. 1799, pp. 65–78, 10.1007/3-540-40048-6_6. [Online]. Available: http://dx.doi.org/10.1007/3-540-40048-6_6

[11] M. O. Rabin, "Fingerprinting by Random Polynomials." TR-CSE-03-01, Center for Research in Computing Technology, Harvard University., Tech. Rep., 1981.

[12] P. Tarr, M. D'Hondt, L. Bergmans, and C. Videira Lopes, "Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems for, Advanced Separation of Concerns," in *Object-Oriented Technology*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, J. Malenfant, S. Moisan, and A. Moreira, Eds. Springer Berlin / Heidelberg, 2000, vol. 1964, pp. 203–240, 10.1007/3-540-44555-2_16. [Online]. Available: http://dx.doi.org/10.1007/3-540-44555-2_16

[13] P. Costanza and R. Hirschfeld, "Language Constructs for Context-oriented Programming: An Overview of ContextL," in *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*. New York, NY, USA: ACM, 2005, pp. 1–10.

[14] P. Costanza, R. Hirschfeld, and W. De Meuter, "Efficient Layer Activation for Switching Context-Dependent Behavior," in *Proceedings of the Joint Modular Languages Conference 2006*, ser. LNCS, D. Lightfoot and C. Szyperski, Eds., no. 4228, Springer. Springer, 2006, pp. 84–103.

[15] M. Appeltauer, R. Hirschfeld, M. Haupt, , and H. Masuhara, "ContextJ: Context-oriented Programming with Java," *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*, vol. 28, no. 1, pp. 272–292, 2011.

[16] M. Appeltauer, R. Hirschfeld, and T. Rho, "Dedicated Programming Support for Context-aware Ubiquitous Applications," in *UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. Washington, DC, USA: IEEE Computer Society Press, 2008, pp. 38–43.

[17] H. Schippers, M. Haupt, and R. Hirschfeld, "An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns," in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1944–1951. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529716

[18] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld, "Delegation-based Semantics for Modularizing Crosscutting Concerns," in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ser. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 525–542. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449806

[19] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara, "ContextJ: Context-oriented Programming with Java," in *Proceedings of the JSSST Annual Conference 2009*, 2009.

[20] M. Haupt and H. Schippers, "A Machine Model for Aspect-Oriented Programming," in *21st European Conference on Object-Oriented Programming, ECOOP 2007*, ser. Lecture Notes in Computer Science, E. Ernst, Ed., vol. 4609. Berlin, Heidelberg, Germany: Springer-Verlag, August 2007, pp. 501–524.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *15th European Conference on Object-Oriented Programming, ECOOP 2001*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., vol. 2072. Berlin, Heidelberg, Germany: Spinger-Verlag, January 2001, pp. 327–354. [Online]. Available: citeseer.ist.psu.edu/kiczales01overview.html

[22] R. Toledo, P. Leger, and É. Tanter, "AspectScript: Expressive aspects for the Web," University of Chile, Tech. Rep. TR/DCC-2009-10, Oct. 2009.

[23] D. Ungar and R. B. Smith, "Self: The Power of Simplicity," *Lisp and symbolic computation*, vol. 4, no. 3, pp. 187–205, 1991.

[24] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 326–336. [Online]. Available: http://doi.acm.org/10.1145/178243.178478