Guard the Cache: Dispatch Optimization in a Contextual Role-oriented Language

Lars Schütze
lars.schuetze@tu-dresden.de
Technische Universität Dresden,
Chair for Compiler Construction
Germany, Dresden

Cornelius Kummer
cornelius.kummer@mailbox.tu-dresden.de
Technische Universität Dresden
Germany, Dresden

Jeronimo Castrillon
jeronimo.castrillon@tu-dresden.de
Technische Universität Dresden,
Chair for Compiler Construction
Germany, Dresden

ABSTRACT

Adaptive programming models are increasingly important as context-dependent software conquers more domains. One such a model is role-oriented programming where behavioral changes are implemented by objects playing and renouncing roles. As with other adaptive models, the overhead introduced by source code adaptations is a major showstopper for role-oriented programs. This is in part because the optimizations of object-oriented virtual machines (VMs) do not provide the same performance gains when applied to role-oriented programs. Recently, dispatch plans have been shown to enable optimizations beyond those in VMs, thereby improving the performance of role programs with low variability. This paper introduces guarded dispatch plans, an extension of dispatch plans with a context-aware guarding mechanism that allows reuse in high-variability scenarios. Fine-grained guards use run-time feedback to partially reuse dispatch plans across call sites when contexts are changing. We present an algorithm to construct and compose guarded dispatch plans and provide a reference implementation of the approach. We show that our approach is able to gracefully degrade into a default dispatch approach when variability increases. The implementation is evaluated with synthetic benchmarks capturing different characteristics. Compared to the state-of-the-art implementation in ObjectTeams we achieved a mean speedup of 3.3× in static cases, 3.0× at low variability and the same performance in highly dynamic cases.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Compilers**; *Context specific languages*.

KEYWORDS

roles, context, dynamic dispatch, adaptive programming

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Lars Schütze, Cornelius Kummer, and Jeronimo Castrillon. 2022. Guard the Cache: Dispatch Optimization in a Contextual Role-oriented Language. In *Proceedings of the 14th ACM International Workshop on Context-Oriented Programming and Advanced Modularity (COP '22), June 06–10, 2022, Berlin, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Today, many applications must be designed for adaptability and extensibility in mind. In the domain of robotics and cyber-physical system (CPS), adaptability defines the ability to change the behavior of the system depending on the context the application is executed. This requires weaving complex logic into the applications to switch among behaviors. Context-dependent behavior is thus scattered across the application and decisions to switch between the behaviors is tangled with the application logic.

Separation of concerns is a guiding principle to conquer the complexity of such modern software systems. Prominent approaches are aspect-oriented programming (AOP) [24], context-oriented programming (COP) [18], and role-oriented programming (ROP) [4, 35, 42]. The latter has been proposed as an extension to object-oriented programming (OOP) to enable adaptive software by design. Classes represent the structural aspect of the domain while roles capture the behavioral aspects. To model context-dependency, compartments encapsulate roles and represent the context in which these roles can be active. Behavioral changes are implemented by objects playing and renouncing roles which in fact adds and removes behavior to and from the object. Hence, role-oriented programming can be seen as a combination of AOP and COP.

Virtual machines (VMs) use run-time profiles to optimize the application. However, heuristics and optimizations designed to work with object-oriented programs do not provide the same performance gains when used with role-oriented programs as a guest language. To apply the adaptations, the artifacts need to be combined properly. This may be done at compile-time whenever the conditions under which the artifacts are combined can be evaluated statically. For conditions that depend on run-time information this has to be done dynamically. Evaluating these residuals at run-time introduce a non-negligible overhead [37].

ObjectTeams [15] is a mature ROP implementation that delivers an overall good performance while supporting most of the features attributed to roles [27]. Recently, a new language runtime was designed for ObjectTeams that builds a plan to capture the dispatch to role functions at runtime to further improve execution performance [38, 39]. This is achieved by caching the results of lookups to foster reuse of dispatch code. The assumption is that applications become stable over time and the context in which the lookup has been performed does not change often. The current language runtime, however, leads to performance penalties in cases with high variability due to a higher cache contention.

In this paper, we propose a language runtime approach that improves support for situations with high variability and demonstrate it with an implementation on ObjectTeams. First, we define and implement improved guards that capture conditions when lookup results may be reused. This extends the use-cases under which reuse is possible, improving performance over existing approaches. Second, we also define a strategy that gracefully degrades the lookup procedure when variability increases.

2 BACKGROUND

This section introduces the role-oriented programming concept and dispatch plans as a technique to optimize role-oriented, context-dependent programs.

2.1 Role-Oriented Programming

Classes in the object-oriented paradigm are good at capturing structures of a domain but not at capturing varying behavior of objects or groups of objects. The idea of roles originated from the domain of databases, where it was observed that persisted objects tend to represent more than a single specific class over time [3]. A similar observation was made in the domain of conceptual modeling [34].

The difference is to classify each entity in the domain to either be the *natural type* which is rigid and independent or the *role type* which is anti-rigid and dependent [26]. This dependency of role types is the foundation of the relation which defines which natural type *fills* a role type. On the level of instances, a natural that plays a role in a context is extended with the behavior and properties of the role. Thus, roles allow separating the structure and relations of entities in a domain and the (context-dependent) behavioral adaptations [27, 42]. This change in behavior enables adaptive software by design and in consequence unanticipated adaptation [44].

2.2 Contextual Roles and the Semantic Gap

Role-playing superimposes behavior onto the natural. The natural appears as an object with a compound type consisting of the natural type and role types [22]. This dynamic extension happens orthogonal to the inheritance hierarchy of the natural type. Method lookup on these compound objects may return different call targets on objects with the same natural type. Role-oriented semantics often must be emulated which in turn incurs a high runtime overhead [37]. The reason is the gap between the object model of these concepts and the object model of the underlying system or VM. In fact, heuristics often decide not to optimize these dynamic extensions resulting in inferior performance [32, 37]. Implementation techniques range from interfaces and design patterns [4, 9, 43] over embedded DSLs [8, 11, 20, 28, 33, 41, 45, 47] to standalone languages [1, 15, 21, 31]. The many implementations result from an inconsistent view on what features constitute a role-oriented programming language – forming a family of role languages [27].

```
1  class Account {
2   void withdraw(float amount) { ··· }
3  }
4  team class Bank {
5   class CheckingsAccount playedBy Account {
6    callin withFee(float amount) {
7     ···
8     result = base.withFee(amount * FEE);
9     ···
10    return result;
11   }
12   withFee(float) ← replace withdraw(float)
13  }
14 }
15 ···
16 Bank bigBank; Account acc;
17 ···
18 bigBank.activate();
19 acc.withdraw(100.00);
```

Figure 1: ObjectTeams/Java code to declare a role with additive behavior adaptations for accounts in a bank and its usage.

ObjectTeams [14] is a programming model which provides most of the features attributed to roles. It combines COP and AOP providing class-wide and instance-local adaptations. The reference implementation extends Java featuring (unanticipated) adaptation [15]. To our knowledge it is the fastest implementation for contextual roles [37]. Fig. 1 shows a snippet of ObjectTeams/Java code.[1] Contexts are represented as `team class` which encapsulate their roles. A role declaration uses the `playedBy` statement to declare the *base type* (i.e., natural type) which is eligible to play the role. A role may define additive behavior that is executed before, after or instead of a method of a base type. Adaptations are declared in *bindings* (Fig. 1 line 12) declaring the method to adapt, how the adaptation must be applied, and the role method (i.e., `callin`) going to be called. In terms used in AOP, `callin` intercepts a method call and `callNext` proceeds the intercepted call.

The compiler type checks declarations of bindings to have compatible type signatures. The compiler assumes closed-world on the types of teams and roles that it type checks. For these bindings code is generated to dispatch to the declared role functions (e.g., `callReplace`) which must be evaluated at runtime. The lookup code contains all possible dispatches to role functions defined inside a team class. For classes that are referenced by the bindings (i.e., base types) the assumption

[1] A detailed description of the language features is presented in [17].

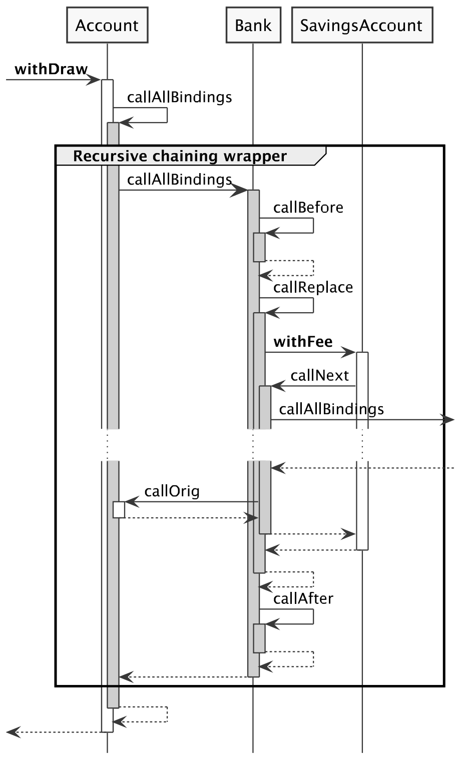


Figure 2: Control flow of a role method dispatch. Grey boxes represent execution of framework code while white boxes represents behavior implementations.

is open-world. At run-time there may be sub-classes loaded which are not known at compile time. To realize such a mixed setting the compiler deduces type information which is preserved in the class files for later consumption by the runtime. The runtime adapts loaded classes and generates entry points into role dispatch to preserve the semantics.

Fig. 2 shows the evaluation from a function call of a program from Fig. 1. This scheme has been coined *recursive chaining wrapper*, as role method dispatch is implemented recursively over active team instances (i.e., `callNext`). The generated code and the recursive evaluation counters optimizations of the VM. For ObjectTeams, the implementation of the lookup in object-oriented VMs causes a performance penalty of 59.9× compared to a pure object-oriented design pattern implementation [37].

2.3 Dispatch Plans

Polymorphic dispatch plans [38, 39] have been proposed as a solution to overcome the inherent overhead of role dispatch. The approach supports the open-world assumption of lazily loaded types at run-time. Inspired by *partial evaluation* [10], the lookup uses runtime feedback from the application to construct a plan of role methods to execute. The resulting dispatch plan is used to generate a graph which can be optimized and re-executed by the Java Virtual Machine (JVM). This requires a mechanism to guard and invalidate the compiled code when the assumptions are not valid anymore.

Instead of implementing super instructions in the VM the approach uses the `invokedynamic` bytecode instruction to link and invoke user-provided, run-time generated code [36]. This bytecode instruction is implemented and optimized differently across JVMs [46, 48]. It is initialized and executed in a two-step process. First, the call site is initialized by invoking a user-defined bootstrap method which returns a call site object. Arguments to the bootstrap method must be statically compiled (e.g., the signature of the generated call site). The resulting call site object manages further calls at this call site. Second, the call site object links executable code like any other function in the JVM. The user-provided code is type-checked when linked making subsequent type-checks unnecessary.

To minimize the impact of role dispatch the graph must only include calls to role methods without any delegates (e.g., `callReplace`) as seen in Fig. 2. Inspired by polymorphic inline-caches (PICs) [19], the graph can be stored and reused in subsequent calls at the same call site. A *guard* ensures that invalid lookup results will not be executed. The guard captures types of active contexts on initial invocation and evaluates on re-execution whether the active context is structurally equal. That is, context instances might change, but their types must remain the same. Otherwise, the guard and the related dispatch plan will be invalidated and the graph must be recomputed. Caches speed up the approach up to 4× but lacks a mechanism to deal with call sites that are unstable (a.k.a., megamorphic). A megamorphic call site triggers cache contention, leading to slowdowns that can easily be of an order of magnitude.

3 GUARDED DISPATCH GRAPHS

As previously discussed, caching role invocation can lead to substantial speedups, but slowdowns can be prohibitively large in case of unstable call sites. On the other side the *recursive chaining wrapper* does not suffer from changing applications. This chapter proposes an approach to combine both worlds; caching and improved guards are used to speed up the application. Yet, it gracefully degrades to a default dispatch approach in which runtime performance is less insensitive to the stability of the call site.

3.1 Type Notations

In order to describe how to construct a dispatch plan and a dispatch graph we first need to describe the type notation used. A role-oriented program has (potential) role-playing classes $\mathcal{B}$ (i.e., base types), context types C, and role types $\mathcal{R}$ defined inside context types. Thus, a role-oriented program has types $\mathcal{T} = C \cup \mathcal{B} \cup \mathcal{R}$. For example, consider a class $Account \in \mathcal{B}$, a $Bank \in C$ which provides the role $CheckingsAccount \in \mathcal{R}$. We use small letters to describe instances of classes at runtime, e.g., acc is an instance of $Account$ written $\text{typeof}(acc) = Account$.

The runtime stores active context instances $\gamma = (c_0, \ldots, c_{n-1})$ of length $n \geq 0$ where $\text{typeof}(c_i) = C_i$ and $C_i \in \mathcal{C}$. Entries in γ are ordered so recently activated contexts come first. This reflects that recently activated contexts provide roles with a higher priority than older contexts.

Extracting the *types of the instances* in γ creates a runtime state $\Gamma = (C_0, \ldots, C_{n-1})$. Two states Γ and Γ' are *structurally equal* iff for each $C_i \in \Gamma$ and $C'_i \in \Gamma'$ is $C_i = C'_i$. Given $\Gamma = (C_0, \ldots, C_{n-1})$ and $\Gamma' = (C'_0, \ldots, C'_{j-1})$ we say Γ *contains* Γ' iff $C'_0 = C_k \wedge \ldots \wedge C'_{j-1} = C_{k+j-1}$ for some $k : 0 \leq k \leq n$.

For any type $T \in \mathcal{T}$, M_T holds all methods declared in T. Each method $m \in M_T$ may be described by its name n and the signature $\sigma = (T_{ret}, (T, T_0, \ldots, T_{n-1})), n \geq 0$ consisting of the return type T_{ret} and a tuple describing the argument types. We define $T^* = (T_0, \ldots, T_{n-1})$ as short-hand for the argument types. The element in the first position is the type T which the method is declared in. The reason is, that a call to m is transformed into a series of byte codes where the callee has to reside on the stack with the arguments. To give an example of a method signature consider the method `CheckingsAccount.withFee` (see Figure 1) which has the concrete method signature $(void, (CheckingsAccount, float))$.

3.2 Lifting

Calls to methods of base types may not be directly dispatched to the respective role methods. In a single dispatched language, the receiver type is used to lookup the method to be called. Thus, the signature of both methods is not directly compatible. Due to the underlying role polymorphism the base type can be coerced to its role type, called *lift* [16].

Given active context instances $\gamma = (c_0, \ldots, c_{n-1})$. Lets assume there is a base type $B \in \mathcal{B}$, and some C_j provides a role type $R \in \mathcal{R}$. A binding in C_j declares a role method $m_R = (n_R, \sigma_R)$ with $\sigma_R = (T_{R_{ret}}, (R, T^*_R))$ to replace $m_B = (n_B, \sigma_B)$ with $\sigma_B = (T_{B_{ret}}, (B, T^*_B))$. While σ_B and σ_R are not directly compatible, a lifting can be defined that is sound due to the underlying *role polymorphism*. The lifting must ensure that for each declared binding there is a partial lifting function $\text{lift} : \mathcal{B} \times \mathcal{C} \rightarrow \mathcal{R}$ s.t. $\text{lift}(B, C_j) = R$. For brevity, we assume that the other argument types are equal:

$$[[\sigma_B]]^{lift}_C = ([[T_{B_{ret}}]], ([[B]]^{lift}_C, [[T^*_B]])) = (T_{R_{ret}}, (R, T^*_R))$$

3.3 Guards

A resolved plan only applies valid adaptations when the application state at the time of construction is structurally equal to the current state. When executed, the guard checks whether the stored state Γ' is structurally equal to the active state Γ. If the guard succeeds the adaptations it guards are executed. Otherwise, another guard may be executed. The chain of guards forms a polymorphic inline-cache of context types.

The semantics of role dispatch distinguishes two kinds of role invocations. The initial invocation at the call site of a base type B and the invocation of a base call (see Fig. 1 line 8). For guard of the latter stores the state which contains the following context types. This increases the chances that a dispatch plan is reused.

For an initial role invocation on base type B, given the state $\Gamma = (C_0, \ldots, C_{n-1})$ and some $C_j, 0 \leq j \leq n-1$ declares a replace binding for method of B. The guard captures the state $\Gamma' = (C_0, \ldots, C_j)$. Whenever Γ' is contained in Γ the guard can be reused.

When control reaches the base call, j contexts have been called. The remaining contexts $\Gamma'' = (C_{j+1}, \ldots, C_{n-1})$ contained in Γ must be processed. Assuming there is no other context that declares a replace binding for method of B the guard will capture Γ''. Otherwise, the guard will capture the contained contexts $(C_{j+1}, \ldots, C_l)$ for some C_l declaring a binding. The amount of context types captured in the guard is thereby minimized and only contains those contributing to the dispatch plan.

Algorithm 1: Construction of a Dispatch Plan

```
In: Function $m = (n, \sigma)$ declared in base type $B$
In: Active context instances $\gamma = (c_0, \ldots, c_{n-1})$
Out: Dispatch plan $\mathcal{P} = (guard, before, replace, after)$
$guard \leftarrow ()$ $after \leftarrow ()$ $before \leftarrow ()$ $replace \leftarrow (m)$
$proceed \leftarrow true$
foreach $c_i \in \gamma \wedge proceed$ do
    $C \leftarrow$ typeof($c_i$)
    $guard \leftarrow (guard, C)$
    // Resolve all bindings for $m$ in $C$
    $M_C \leftarrow$ lookupRoleBindings($C, m$)
    foreach $m_i = (n_i, \sigma_i) \in M_C$ do
        // Lift signature $\sigma$ to $\sigma_i$
        $m_r \leftarrow$ liftBaseToRole($m, m_i$)
        switch kindof($m_i$) do
            case before do
                $before \leftarrow (before, m_r)$
            end
            case replace do
                $replace \leftarrow m_r$
                $proceed \leftarrow false$
            end
            case after do
                $after \leftarrow (after, m_r)$
            end
        end
    end
end
return appendAll($guard, before, replace, after$)
```

3.4 Guarded Dispatch Plans

Whenever a context instance is activated or deactivated the runtime updates the store of active context instances γ. The construction of a dispatch plan takes these active context instances γ and the function m of the base type B that is invoked. It returns all role functions that must be called and

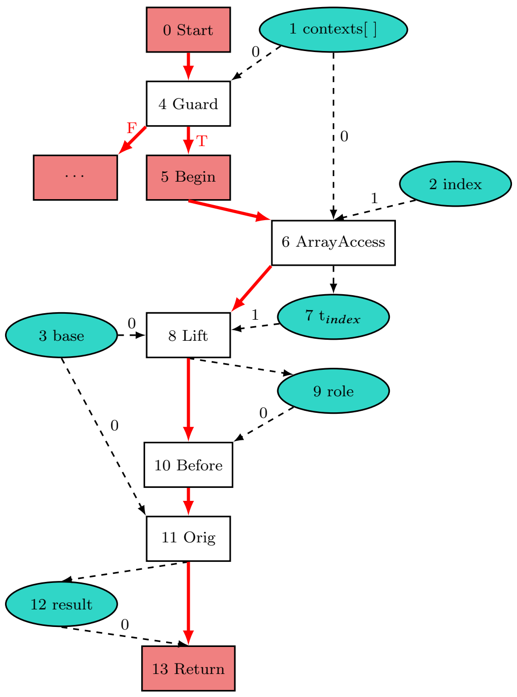


Figure 3: Dispatch graph representing the invocation of a before role function and the original function

the captured contexts in Γ'. The guarded approach makes each dispatch plan independent from the deactivation of a context. In case Γ and Γ' are not structurally equal, the guard forwards execution instead of discarding the whole plan as in previous approaches.

Algorithm 1 describes how the construction is performed. It iterates over context instances in γ as long as no replace binding is discovered. The guard captures each processed context type. For each context type the declared bindings are queried and signatures are lifted. According to the kind of binding declared the role function is appended to already captured role functions of the same kind. The returned dispatch plan $\mathcal{P}$ includes all discovered role functions separated by kind and the guard to be created. From such a plan the dispatch graph can be created which is executed by the JVM.

3.5 Guarded Dispatch Graph

The dispatch graph is the high-level intermediate representation (IR) generated from a dispatch plan. The IR is inspired by the sea-of-nodes notation [7] and is subsequently optimized by the JVM. Each white box represents a function, while red boxes represent special nodes such as the begin of a basic block or return instructions. Turquoise ellipses represent data. The red edge defines the control flow while dashed edges define data flow. The annotation on dashed edges defines the order of incoming arguments. A partial order among nodes connected by the control flow defines a possible execution order, one of which is highlighted by the annotation on the vertices. The root of the dispatch graph is the entry point to the execution at the call site with the arguments present on the call stack. The JVM is able to (abstractly) interpret the IR to yield whether the provided and required signature of the call site is fulfilled by the graph.

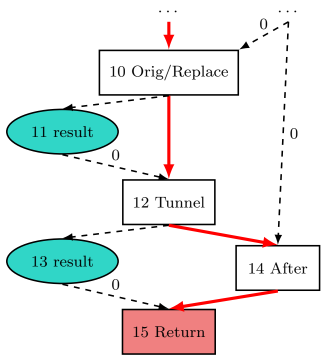


Figure 4: Result tunnelling in a dispatch graph

As shown in Algorithm 1 the creation of dispatch plans aggregates role functions that can be composed into a single execution. The dispatch graph must be composed out of the smaller graphs that represent each individual role dispatch. It accepts the signature according to the base method σ_B. Each role binding is type checked by the compiler and at run-time individual dispatches are type checked and lifted accordingly.

Role functions that have the *before* or *after* behavior modifier can be executed in conjunction before or after the original function (or the replacing role function). They may not change the control-flow by providing *base calls* and do not change the return value. In contrast, *replace* behavior may have a *base call* instruction inside the role function body. The return value of that instruction could be used in subsequent statements (see Figure 1) until the role method finally returns. Because the exact execution trace is not known a priori, the dispatch graph has to generate special instructions to cope with the result called *result tunneling*.

When a role function is executed after a replace the result of the replace (or original base function) must be returned to the call site. Figure 4 shows a dispatch graph that captures the result. There might be multiple functions executed afterwards. This also highlights how dispatch graphs capture the semantics of role dispatch and bridge the gap between language semantics and efficient execution.

3.6 Graceful Degradation

In call sites with high variance, cached dispatch plans are evicted and constructed anew. There is a break-even point after which the constant re-construction of the plan become prohibitively costly [39]. To cope with this, the *graceful degradation* mechanism is applied when a threshold is reached to reduce the impact of cache contention.

First, call sites track the length of the chain of guards and prunes the oldest entries whenever a threshold is reached.This impacts the number of comparisons with guards for structural equality. Second, there is a maximum number of evictions tolerated at a call site. If the threshold is reached the call site is tagged as unstable, gracefully degrades and ceases to construct dispatch plans at all. All calls will be forwarded to the recursive chaining wrapper without incurring extra overhead. In line with current approaches this decision is final to avoid optimizing and deoptimizing a call site.

We decided the initial values for the thresholds from numbers found in literature. An approach using PICs in JavaScript observed a maximum of 7 entries in their benchmarks [40] and an implementation for Squeak holds a maximum number of 8 slots [12]. We used these references to define the threshold for instability of 8 and half that value as an initial value for the length of the chain as each entry might consist of a bigger chunk of code.

4 EVALUATION

This section evaluates our approach compared to the original dispatch implemented in ObjectTeams and dispatch plans without graceful degradation [39]. We use microbenchmarks that help to stress different aspects of our approach.

4.1 Experimental Design

The experiments were conducted on a Linux server with Ubuntu 20.04, 32GB RAM and Core i7-9700T CPU. For execution we used Oracle JDK 14.02 with 8GB heap and compared against the latest version of ObjectTeams (2.8.1). All benchmarks were implemented with Java Microbenchmark Harness (JMH) and measured for 10 iterations after 10 warmup iterations. To coordinate the execution of the benchmarks we used ReBench [29].

4.2 Instrumentation Overhead

To implement the semantics of conditional interception often residuals have to be evaluated contributing to the overall execution time even if there is nothing to adapt. The time to call a single method that has a registered binding but no active context instance (i.e., *noop*) is measured. Second, a single context instance provides an adaptation to a single base class.

Figure 5a shows the geomean execution time where in the *noop case* our approach is 2.5× faster than the original implementation. The guard captures that there is no active context and the call site directly links to the original method. In the *single replace* case our approach is 3.3× faster. The guard captures the active context and links to the replace role method. The base call directly calls the original method. The original implementation always uses the stub methods to dispatch which explains the overhead.

4.3 Characteristics of Role Method Types

To evaluate the run time characteristics of each variant of adaptation in isolation (before, replace, after), we built a synthetic benchmark for each of them. In these benchmarks, a context only provides the type of adaptation that is subject to evaluation. The amount of active context instances is varied between 1, 10 to 100.

The results are shown in Figure 5b. Due to the fixed dispatch scheme, the execution time of the original implementation in every scenario is almost the same. Our approach composes a different plan depending on the types of behavioral adaptations. Dispatch Plans are on average 2.9× faster (max 3.5×) in the case in which all bindings are of type before. Role methods of type after are executed on average 3.3× faster (max 3.7×) than the original implementation. While not changing contexts the guards could be completely reused allowing to execute replace callins on average 2.9× faster (max 3.4×).

For polymorphic dispatch plans without degradation speedups between 3.8× to 4.5× have been reported for static cases [39]. This is comparable to our results as the guards introduce more computations and jumps in the resulting code.

4.4 Instability and Graceful Degradation

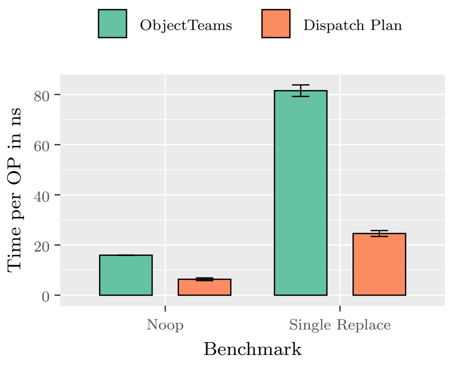
Our approach supports graceful degradation to diminish the overhead for unstable call sites. To evaluate the effectiveness of the degradation we designed a synthetic benchmark that creates a set of permutations of active contexts and measures the execution time across these permutations. Each context that participates contributes a before, replace and after role method. We compared the execution time of the original approach against dispatch plans and dispatch plans with degradation disabled.

Compared to the original implementation, our approach has on average the same execution time as the original approach since there is no possible reuse. However, without graceful degradation the execution time increases by up to 3.8×. This means that too much variability can be effectively countered by degrading to the original dispatch.

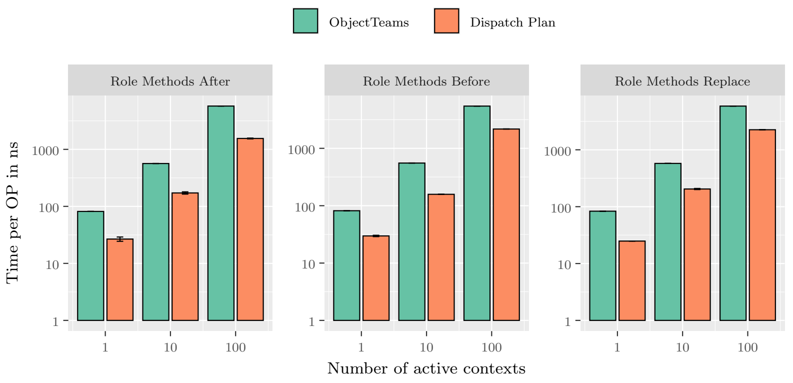
5 RELATED WORK

This section will introduce related work in the context of VM implementations application level. Our work is in between both approaches; lookup is defined at application level but optimized by the VM.

Optimizing dispatch is a recurring topic among approaches that use reflection or meta-object protocols (MOPs). MOPs enable the definition of extended semantics in the host language itself [23]. One can define dispatch from within the host language instead of relying on VM or compiler support. For these (reflective) call sites call target and argument types are not known until invocation. A single PIC may not be able capture these cases since same-named methods can be called on unrelated objects. A chain of PICs is able to store the result of the lookup for each intermediate level for further reuse [30]. VMs use heuristics for aggressive speculative optimizations. But in some cases heuristics fail to determine whether optimizations are beneficial. To communicate variabilities to the VM call sites in the application can be annotated. Thus, optimizations can be triggered where heuristics would fail [6].



(a) **Comparison of no active contexts (noop) and a single context with a replace role function.**



(b) **The contribution of different role function types (before, after, replace) to the overall execution time. Each type is measured with 1, 10 and 100 active contexts.**

Figure 5: Evaluation results of the overhead (left) and comparison of the impact of role function types in both approaches (right)

Communicating the variability to the VM via annotations has also been proposed for COP, where sideway composition introduces an overhead [32].

Steamloom [5] uses super instructions in a custom VM to implement AOP semantics. A separate data structure manages aspect objects the just-in-time (JIT) compiler use to generate aspect code into method bodies [13]. The data structure contains meta-data for each class. Each entry of instance-local aspects must be cloned from the class adapted by the aspect. Inlining method bodies of these classes is not possible anymore as there exist multiple versions of these classes.

In ROP contexts introduce highly indirect variability. Our approach encodes opportunities to optimize specific call sites. Without using compiler directives, we rely on the fact that the abstraction used by dispatch graphs is optimized by the VM.

ContextJS [25] uses wrappers to dynamically rewrite the AST. The wrapper delegates to each partial method in the beginning but subsequently inlines the delegatee to reduce the overhead. When stabilized, the partial function boundaries are removed and the method bodies copied. In our approach the amount of optimizations is decided by the VM.

JCOP has been extended to use `invokedynamic` call sites [2]. Call targets to partial methods are stored in a map which is managed from each layer. Each call to proceed to the next active layer will link the next partial method stored in the map for the layer. They observed a speedup despite not embedding the call graph for partial methods.

6 CONCLUSION AND FUTURE WORK

We presented guarded dispatch plans that overcome the performance problems other role-oriented programming languages face. This can be achieved by regarding the dynamic run-time state, i.e., active contexts and their roles, as invariant during compilation. This requires a mechanism to guard and eradicate compiled code when the assumptions are not valid anymore.

We show that our approach is able to gracefully degrade into other dispatch approaches when variability increases. The implementation is evaluated with synthetic benchmarks capturing different characteristics. Compared to the state-of-the-art implementation in the ObjectTeams we achieved a mean speedup of 3.3× in static cases, 3.0× at low variability and the same performance in highly dynamic cases. Without guards, the approach would become 3.8× slower on very dynamic applications.

In the future we will evaluate our approach on real-world programs using ObjectTeams. A candidate application is the static analysis of the Eclipse Java Compiler to detect potential problems related to the null-ness of variables (null analysis), such as dereferencing a null value. Before the analysis has been officially released, a proof-of-concept implementation had been built using ObjectTeams. Due to its many extensions of the compiler it is a great use-case how our approach can improve real-world applications.

ACKNOWLEDGMENTS

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907)

REFERENCES

[1] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. 1995. Fibonacci: A Programming Language for Object Databases. *The VLDB Journal* 4, 3 (July 1995), 403–444.
[2] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. 2010. Layered Method Dispatch with INVOKEDYNAMIC: An Implementation Study. ACM Press, 1–6.
[3] Charles W. Bachman and Manilal Daya. 1977. The Role Concept in Data Models. In *Proceedings of the Third International Conference on Very Large Data Bases*, Vol. 3. Tokyo, Japan, 464–476.
[4] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. 1997. The Role Object Pattern. In *Proceedings of the 1997 Conference*

on Pattern Languages of Programs (PLoP 97).

[5] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. 2004. Virtual Machine Support for Dynamic Join Points. ACM Press, 83–92.

[6] Guido Chari, Diego Garbervetsky, and Stefan Marr. 2017. A Metaobject Protocol for Optimizing Application-Specific Run-Time Variability. ACM Press, 1–5.

[7] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 181–196.

[8] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. 2004. A Role Model and Its Metaclass Implementation. *Information Systems* 29, 3 (May 2004), 235–270.

[9] Martin Fowler. 1997. Dealing with Roles. In *Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP 97)*.

[10] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

[11] Georg Gottlob, Michael Schrefl, and Brigitte Röck. 1996. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems* 14, 3 (July 1996), 268–296.

[12] Michael Haupt, Robert Hirschfeld, and Markus Denker. 2007. Type Feedback for Bytecode Interpreters. In *Proceedings of the Second Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. Berlin, Germany.

[13] Michael Haupt and Mira Mezini. 2005. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet* 11, 3 (2005).

[14] Stephan Herrmann. 2003. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World*. Vol. 2591. Springer Berlin Heidelberg, Berlin, Heidelberg, 248–264.

[15] Stephan Herrmann. 2007. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology* 2, 2 (2007), 181–207.

[16] Stephan Herrmann, Christine Hundt, and Katharina Mehner. 2004. *Translation Polymorphism in Object Teams*. Technical Report Bericht-Nr. 2004/05. Technische Universität Berlin, Berlin.

[17] Stephan Herrmann, Christine Hundt, and Marco Mosconi. 2011. OT/J Language Definition v1.3.

[18] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *The Journal of Object Technology* 7, 3 (2008), 125.

[19] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 European Conference on Object-Oriented Programming*. Vol. 512. Springer-Verlag, Berlin/Heidelberg, 21–38.

[20] Bo Nørregaard Jørgensen and Eddy Truyen. 2003. Evolution of Collective Object Behavior in Presence of Simultaneous Client-Specific Views. In *Object-Oriented Information Systems*. Vol. 2817. Springer Berlin Heidelberg, Berlin, Heidelberg, 18–32.

[21] Tetsuo Kamina and Tetsuo Tamai. 2009. Towards Safe and Flexible Object Adaptation. In *International Workshop on Context-Oriented Programming*. ACM Press, 1–6.

[22] Tetsuo Kamina and Tetsuo Tamai. 2010. A Smooth Combination of Role-based Language and Context Activation. In *FOAL 2010 Proceedings*.

[23] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Mass.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *ECOOP'97 — Object-Oriented Programming*. Vol. 1241. Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[25] Robert Krahn, Jens Lincke, and Robert Hirschfeld. 2012. Efficient Layer Activation in Context JS. IEEE, 76–83.

[26] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. 2015. A Combined Formal Model for Relational Context-Dependent Roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. Pittsburgh, PA, USA, 113–124.

[27] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*. Vol. 8706. Springer International Publishing, Cham, 141–160.

[28] Max Leuthäuser. 2017. Pure Embedding of Evolving Objects. In *The Ninth International Conference on Advanced Cognitive Technologies and Applications*. 22–30.

[29] Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. (Aug. 2018).

[30] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. ACM Press, 545–554.

[31] Supasit Monpratarnchai and Tamai Tetsuo. 2008. The Implementation and Execution Framework of a Role Model Based Language, EpsilonJ. IEEE, 269–276.

[32] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2016. Optimizing Sideways Composition: Fast Context-oriented Programming in ContextPyPy. ACM Press, 13–20.

[33] Michael Pradel and Martin Odersky. 2008. SCALA ROLES A Lightweight Approach towards Reusable Collaborations. In *ICSOFT 2008 - Proceedings of the 3rd International Conference on Software and Data Technologies*. 13–20.

[34] Trygve Reenskaug, Per Wold, and Odd Arilc Lehne. 1996. *Working with Objects: The OOram Software Engineering Method*. Manning, Greenwich.

[35] Dirk Riehle and Thomas Gross. 1998. Role Model Based Framework Design and Integration. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press, 117–133.

[36] John R. Rose. 2009. Bytecodes Meet Combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. ACM Press, Orlando, Florida, 1–11.

[37] Lars Schütze and Jeronimo Castrillon. 2017. Analyzing State-of-the-Art Role-based Programming Languages. In *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17*. ACM Press, Brussels, Belgium, 1–6.

[38] Lars Schütze and Jeronimo Castrillon. 2019. Efficient Late Binding of Dynamic Function Compositions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2019*. ACM Press, Athens, Greece, 141–151.

[39] Lars Schütze and Jeronimo Castrillon. 2020. Efficient Dispatch of Multi-object Polymorphic Call Sites in Contextual Role-Oriented Programming Languages. In *17th International Conference on Managed Programming Languages and Runtimes*. ACM, Virtual UK, 52–62.

[40] Manuel Serrano and Marc Feeley. 2019. Property Caches Revisited. In *Proceedings of the 28th International Conference on Compiler Construction - CC 2019*. ACM Press, Washington, DC, USA, 99–110.

[41] Yannis Smaragdakis and Don Batory. 2002. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology* 11, 2 (April 2002), 215–255.

[42] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data & Knowledge Engineering* 35, 1 (Oct. 2000), 83–106.

[43] Friedrich Steimann. 2001. Role= Interface: A Merger of Concepts. *Journal of Object-Oriented Programming* (2001).

[44] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. 2016. A Dynamic Instance Binding Mechanism Supporting Run-Time Variability of Role-Based Software Systems. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM Press, 137–142.

[45] Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. 2016. Consistent Unanticipated Adaptation for Context-Dependent Applications. ACM Press, 33–38.

[46] Christian Thalinger and John Rose. 2010. Optimizing Invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java - PPPJ '10*. ACM Press, Vienna, Austria, 1.

[47] R.K. Wong and H.L. Chau. 1998. Method Dispatching and Type Safety for Objects with Multiple Roles. In *Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS 25 (Cat. No.97TB100239)*. IEEE Comput. Soc, Melbourne, Vic., Australia, 286–296.

[48] Shijie Xu, David Bremner, and Daniel Heidinga. 2016. MHDeS: Deduplicating Method Handle Graphs for Efficient Dynamic JVM Language Implementations. ACM Press, 1–10.