Lars Schütze, Jeronimo Castrillon

# Efficient dispatch of multi-object polymorphic call sites in contextual role-oriented programming languages

**Erstveröffentlichung in / First published in:**

*MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes.* Virtual, 2019. ACM Digital Library, S. 52–62. ISBN 978-1-4503-8853-5

DOI: https://doi.org/10.1145/3426182.3426186

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062

# Efficient Dispatch of Multi-object Polymorphic Call Sites in Contextual Role-Oriented Programming Languages

Lars Schütze
Technische Universität Dresden
Dresden, Germany
lars.schuetze@tu-dresden.de

Jeronimo Castrillon
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

## ABSTRACT

Adaptive software becomes more and more important as computing is increasingly context-dependent. Runtime adaptability can be achieved by dynamically selecting and applying context-specific code. Role-oriented programming has been proposed as a paradigm to enable runtime adaptive software by design. Roles change the objects' behavior at runtime, thus adapting the software to a given context. The cost of adaptivity is however a high runtime overhead stemming from executing compositions of behavior-modifying code. It has been shown that the overhead can be reduced by optimizing dispatch plans at runtime when contexts do not change, but no method exists to reduce the overhead in cases with high context variability. This paper presents a novel approach to implement polymorphic role dispatch, taking advantage of run-time information to effectively guard abstractions and enable reuse even in the presence of variable contexts. The concept of polymorphic inline caches is extended to role invocations. We evaluate the implementation with a benchmark for role-oriented programming languages achieving a geometric mean speedup of 4.0× (3.8× up to 4.5×) with static contexts, and close to no overhead in the case of varying contexts over the current implementation of contextual roles in Object Teams.

## CCS CONCEPTS

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

## KEYWORDS

roles, context, dispatch, dynamic languages

**ACM Reference Format:**
Lars Schütze and Jeronimo Castrillon. 2020. Efficient Dispatch of Multi-object Polymorphic Call Sites in Contextual Role-Oriented Programming Languages. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes (MPLR '20), November 4–6, 2020, Virtual, UK*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3426182.3426186



## 1 INTRODUCTION

Separation of concerns is the main technique to conquer complexity as it allows decomposing a system into different smaller components. Decomposition is typically dominant, e.g., by object or function, and is predefined by the underlying programming language. However, different concerns often overlap and interact with one another, requiring different decompositions if treated on its own. Multi-dimensional Separation of Concerns (MDSoC) allows decomposing a system into multiple eventually overlapping concerns [38], which may be composed in different ways to reduce development complexity.

Composing methods (i.e., adaptations) out of decompositions and their execution violates assumptions common language implementations hold about lookup resulting in inefficient code implementing the dispatch. Optimizations such as polymorphic inline caches (PIC) [27], caching results of lookups of potentially polymorphic sites, or analyses to inline code of call targets at a call site do not improve performance in the same way they do for regular object-oriented programs [44]. For language implementations, such as the Java Virtual Machine (JVM), it is important to understand where potential variability is relevant at run time. Thus, research on MDSoC approaches such as aspect-oriented programming (AOP) [29], context-oriented programming (COP) [25], and role-oriented programming (ROP) [32, 46] mainly concentrated on improving run-time performance especially by improving lookup or the code generated to implement dispatch. However, there is still no solution to cope with the dynamism inherent to these concepts. In all approaches, compositions can be switched on or off invalidating generated code. The resulting impact on run-time performance can be of several orders of magnitude, depending on the language implementation [44].

Adaptations that happen on a *per-object* basis require the language runtime to treat every object separately. Due to the nature of cross-cutting aspects a rare case in AOP, it is the common case in contextual role-oriented programming, where objects may adapt their behavior by assuming and discarding roles bound to contexts at run time. The roles implement the specific context-dependent behavior and state. Hence, late binding in ROP has to ensure that every object, depending on its assumed roles, may have a different implementation of a method when invoked at a polymorphic reference. An object that reappears at the same call site to be dispatched may also assume different roles then before. In a single dispatched language this requires multiple steps where the language runtime might not observe the connection between functions.

It has been shown that the overhead can be reduced by optimizing dispatch plans at run time for static cases, but no method exists to reduce the overhead in cases with high variability. A static case

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



is reached if there is no contextual adaptation for some period of time so the same code is repeatedly executed. Variability emerges when contextual adaptations are applied changing the executed code.

This paper presents a novel approach to implement polymorphic role dispatch. By taking advantage of run-time information and a new lookup mechanism to link to actual role function implementations the overhead can be eliminated. The run-time generated call graphs are guarded by requiring a structural equivalent runtime state to be executed again. To minimize the overhead of repeatedly recalculating call graphs, call sites can deoptimize into a generic state reusing the original calling convention. The approach is implemented into the language Object Teams [21] that implements contextual roles, providing most of the features that nearly four decades of research have ascribed to role-oriented implementations [32, 46].

We evaluate the implementation with a benchmark for role-oriented programming languages achieving a geometric mean speed-up of 4.0× (3.8× up to 4.5×) in the static case, and close to no overhead in the dynamic case over the current implementation of contextual roles in Object Teams.

## 2 BACKGROUND

During the last decades multiple concepts have been introduced that not only allow to decompose a system along different dimensions, but also to change components dynamically at runtime. This introduces runtime overhead especially at the points in the system where dimensions are merged, i.e., call sites. This section introduces concepts related to Multi-Dimensional Separation of Concerns and gives an overview of their applicability to implement context-dependent adaptations. recent research in optimizing the inherent runtime overhead of these approaches. Furthermore, it introduces techniques that can be applied by language implementers to resolve additional variability at call sites to enable optimizations by the runtime environments.

### 2.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) increases modularity by enabling decomposition of cross-cutting concerns, i.e., logging. These concerns are encapsulated into *aspects* that provide expressions to define interceptors, class extensions (inter-type declarations) and properties [10]. An aspect can adapt the behavior of parts of a program, called *base* classes or methods, by applying *advices* which define the additional behavior. These advices are applied at specific points in the base program called *join points*, e.g., function calls or property accesses. *Pointcuts* provide predicates that quantify over the set of join points and choose a set of join points where the execution of the advice is desired. Context-sensitive adaptations have to be implemented into the predicates of the pointcut. Aspects are compiled by special compilers called *weavers* as advice invocation code is woven into the application code. Locations in the code where advices might be woven are called *join point shadows* [24].

A compiler of an aspect-oriented language consists of a module for evaluating pointcuts and a weaver, beside traditional elements of a compiler. After evaluating a pointcut, the join point shadows are forwarded to the weaver. The weaver, however, may not always decide at compile-time whether a pointcut apply at a join point shadow or not. Thus, for some join points advice invocation logic and guards are compiled into the application and are called *residuals*.

```
1  public aspect BankAspect {
2    private final float FEE = 1.1;
3    pointcut callWithdraw(float amount,
4      Account acc) :
5      call(void Account.withdraw(float)) &&
6        args(amount) && target(acc) && within(
             Bank);
7    void around(float amount, Account acc) :
8      callWithdraw(amount, acc) {
9      proceed(amount * FEE, acc);
10   }
11 }
```

**Figure 1: An aspect definition in AspectJ, defining a pointcut that applies to `Account.withdraw` and an around advice that adds the fee before proceeding to the original method.**

In Figure 1 an aspect-oriented example implemented with AspectJ [28] is shown. The aspect encapsulates all behavioral variations. The pointcut enables different behavior for `Accounts` whose withdraw method is called inside the `Bank`. The advice is woven around the identified joinpoints and will proceed with calling the original method. While AOP excels at defining cross-cutting behavior it also allows to integrate context-dependent adaptations using the `within` statement. However, it is not designed to easily implement object-specific adaptations, i.e., that apply only to specific `Account` instances.

### 2.2 Context-Oriented Programming

Context-Oriented Programming (COP) aims at adapting the behavior of an application to a known context by providing contextual variations. In contrast to AOP, contexts are first-class citizens with dedicated language support. Similar to AOP the base program can be altered at join points at method-level granularity. To achieve contextual variation, *layers* implement context-dependent behavior in *partial methods*. Variations can replace the original function, be executed before and after, or use the mechanism of *proceed* to delegate to the next active layer. The activation and deactivation of layers drive contextual adaptation. This process is called *side-way composition* [26] as the original inheritance mechanism is extended by a dynamic, orthogonal extension at run time.

In an object-oriented execution model, function invocation is understood as a two-dimensional message sent to the receiver object consisting of the name of the function to be executed and a list of parameters. In COP, however, this message is extended to four dimensions, adding the sender object and the context of the actual message [25]. Thus, COP resembles *multiple dispatch* which takes any argument into account for the dispatch. Modern object-oriented programming languages use *single dispatch*, only taking into account the first argument, i.e., the run-time type of the receiver. While there have been implementations of multiple dispatch

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



```
1  public layer SavingsAccount {
2    private final static float FEE = 0.1;
3    public void Account.withdraw(float amount){
4      proceed(FEE * amount);
5    }
6  }
7
8  public class Bank {
9    public void transaction(Account source,
10     Account target, float amount) {
11     with(SavingsAccount) {
12       source.withdraw(amount);
13       target.deposit(amount);
14     }
15   }
16 }
```

**Figure 2: JCOP uses layers to define different behavior when withdrawing money for a savings account. The changed behavior is scoped by the `with` statement.**

```
1  class Account {
2    void withDraw(float amount) {...}
3  }
4
5  team class Bank {
6    class CheckingsAccount playedBy Account {
7      ...
8    }
9
10   class SavingsAccount playedBy Account {
11     callin void withFee(float amount) {
12       base.withFee(FEE * amount);
13     }
14     withFee <- replace withDraw;
15   }
16 }
```

**Figure 3: Object Teams source code showing how Accounts can have different behavior when withdrawing money. The SavingsAccount adds a fee for each withdrawal.**

in single dispatched languages like Java [49], context-aware execution semantics is often implemented using imperative control flow and libraries [43].

Figure 2 is an example of such a context-oriented implementation using imperative control flow with the JCOP programming language [4]. Behavioral variations can be defined as classes with layers. A layer can define extensions to methods of other classes. The `with` statement defines the scope and order in which layers are active. However, there is no language concept to define the context and constraints whether to apply behavioral variations. Because there is also no cross-cutting semantics all possible control flow paths that must lead to behavioral variations have to be identified and guarded.

## 2.3 Role-Oriented Programming

The successful adoptions of roles in software analysis and conceptual modeling [2, 40, 41, 46] led to a demand for programming language support. First approaches hid the concept in the implementation with design patterns [7, 13].

Role-oriented programming distinguishes between the base entities themselves and the roles they play in a collaboration. This provides explicit support for object collaboration in a way not normally supported by language features. Base classes can be adapted by implementing the behavioral adaptations in *roles* which work at method-level granularity while also encapsulating role specific state.

In *contextual* role-oriented programming, objectified *compartments* (contexts) encapsulate roles to capture the context-dependent relationship of behavior. Objects may adapt their behavior by assuming and discarding roles bound to contexts. Hence, role-oriented programming can be seen as a combination of AOP and COP in a way, depending on the specific role-oriented programming language, it uses a mixture of instance-local and class-wide aspects to implement context-dependent adaptations.

In the past years different role-oriented approaches and implementations emerged which can be grouped into behavioral [6, 11, 14] and contextual role languages [5, 22, 33, 37, 47]. Different implementations support different sets of role features [32, 46].

## 2.4 Contextual Roles with Object Teams

While behavioral roles can be compared to dynamic programming languages such as Ruby or Python, where playing a role merely reduces to changing the implementation of a function, contextual role languages establish their own domain. Behavior is not only dependent on the instance of the role, but also on the instance of the context that role is played in. Thus, role playing becomes a triple of contexts, role and player [31].

Object Teams [19] is the most sophisticated role-oriented programming language supporting most of the features attributed to roles [32]. There currently exist a reference implementation in Java [21] which is the fastest implementation of roles despite the many features supported [44]. An example is shown in Figure 3 which is also part of the evaluation in Section 4. It shows how a bank is modeled as a team representing the context of different types of accounts. These types add additional constraints to the basic functionality of withdrawing and disposing money from those accounts. The reference implementation extends the syntax of Java and introduces a new class type named `team class`. Teams can be instantiated to represent objectified contexts and encapsulate roles. Roles are defined as inner classes of teams. With a slight extension to the Java syntax, the `playedBy` relation can connect roles to the role-playing base classes as shown in Figure 3 line 10. Roles in Object Teams define new or modified behavior and state of their base classes while their semantics is similar to crosscutting concerns in AOP. Thus, every instance of the base class which plays a role in a team is affected whenever an instance of that team is active.

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



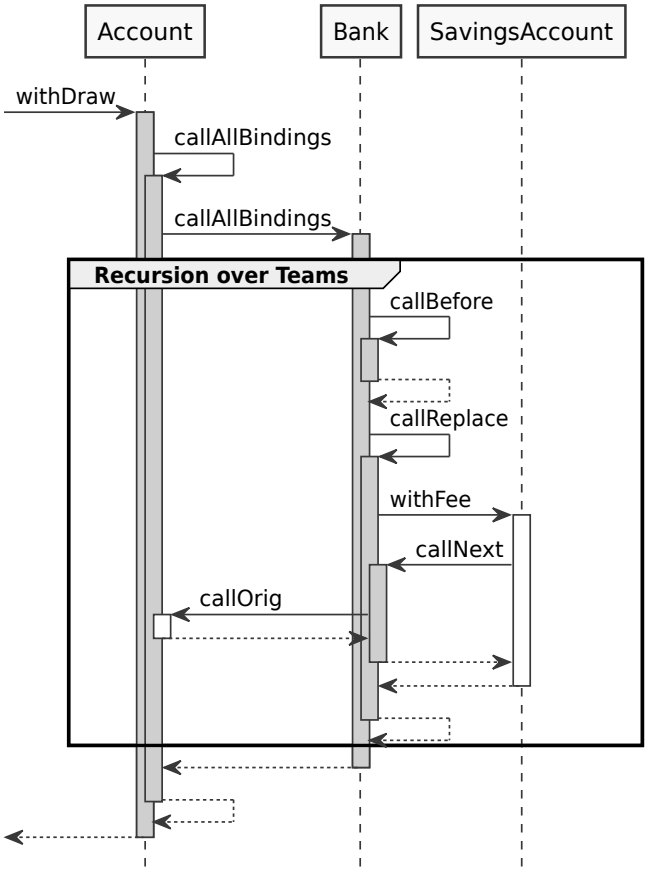


**Figure 4: A sequence diagram of the trace of a role dispatch in Object Teams implemented in a single dispatch language.**

*Role Bindings.* While most approaches resort to structural typing where the signature of role functions must be identical to the signature of the base function, Object Teams provides mappings to state the *binding* from role functions to base functions. Mappings are defined inside the role classes as shown in Figure 3 line 14. To accommodate reuse of legacy code bindings can define permutations on the arguments, or define arbitrary glue code to achieve signature compatibility.

Role functions may have two directions. A *callout* delegates a role function to a base function to reuse existing behavior of base classes. In the other direction, role functions intercept calls to functions of their base classes. They may be executed before, after, or replace base functions. Because they alter the behavior of base classes and result into calling into the role, these functions are called *callin*.

In a running application there can be multiple active teams that provide roles that have bindings for the same base function. The Object Teams runtime keeps a stack of active teams where the latest activated team has the highest priority. If a callin replaces a base function it can also call back into the original function performing a *base call*. Because a callin can have potentially multiple bindings to different base functions the syntax is defined to do a base call to the role function as shown in Figure 3 line 12. This behavior is comparable to the `proceed` keyword introduced by aspect-oriented and context-oriented programming. Whenever such a base call happens, the next callin from the stack of active teams has precedence over the original function. This results in a recursive application of replace callins until there is no more active callin or there is no more base call executed.

*Lookup.* The Object Teams reference implementation consists of two compilers. The application is first statically compiled and then dynamically *woven* at run time. The static compiler extends the Eclipse Java compiler. It provides semantic checks on the bindings and role functions to detect type errors and generates the dispatch logic. The weaver adapts base classes when they are loaded. It rewrites functions if there exist a binding and weaves code to redirect them into the Object Teams runtime for dispatching. For each function, a unique ID is selected and the body is moved into a generic function `callOrig`. The purpose of this function is to dispatch to the original code which is guarded by that ID.

To resolve lookups, Object Teams provides a sophisticated mechanism to implement role lookup into an object-oriented programming language. The static compiler associates each binding with an ID that is uniquely associated with that binding among the inheritance hierarchy of the team. For each team a generic dispatch method `callAllBindings` is generated. The purpose is to delegate to the respective methods for each type of callin. Those special dispatch methods dispatch based on the generated ID to the respective role functions. To realize this, the implementation needs to retrieve the role instance played by the base object in the team instance. This process is called *translation polymorphism* where players are lifted to their respective role in a given context [23]. Figure 4 shows a trace of how the reference implementation of Object Teams resolves lookups at call sites bound by roles. To visualize the overhead the sequence diagram uses grey bars to highlight framework code while white bars are the actual code of the application.

Deep nested function calls result in less optimizations by the Just-in-Time (JIT) compiler such as function inlining. In a single dispatched language this requires multiple steps where the language runtime might not observe the connection between functions. Generated functions such as `Bank.callReplace` which is responsible for dispatching to every replace callin inside the team `Bank` may become too large to be chosen for any optimization at all. Thus, the approach hinders optimization for variability at call sites, since the language runtime's heuristic treat framework code as application code. The impact is severe as the most sophisticated approach to context dependent roles, Object Teams, is still roughly 54 times slower than a comparable implementation using design patterns [44].

*Family Polymorphism.* Due to the advanced concepts of inheritance defined in Object Teams, where subclasses of base classes, subroles of roles, and subteams of teams co-exist, the very powerful concept of *family polymorphism* is introduced [12]. By inheriting at team-level, a corresponding family, where the types of roles are bound to the team instance containing them, is created. This is important when roles are *externalized*, i.e., stored and used outside of teams [20, 23]. In the reference implementation, teams define inner classes which are roles. Role classes are virtual classes. Figure 5 shows a diagram which uses a notation that is inspired by UML

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



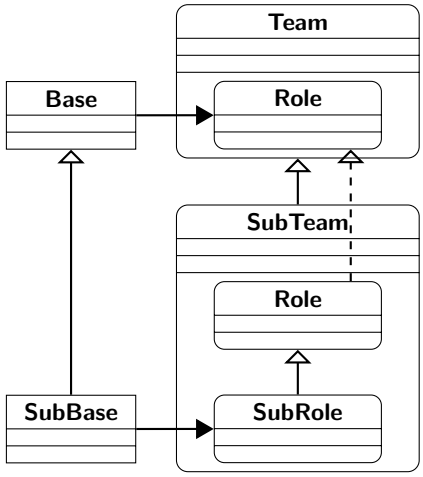


**Figure 5: Family polymorphism is introduced in Object Teams by inheritance between teams. The notion of the diagram is inspired by a UML class diagram.**

class diagrams. It represents the different inheritance relations between classes, teams and roles using the UML generalization arrows. Roles in subteams override (implicitly inherit) roles with the same name in superteams is shown as a UML realization arrow. The fills relation defines which role is played by which class is represented as a solid arrow.

## 2.5 Optimizing Meta-Object Protocols

In traditional object-orientation the variability at a polymorphic call site is determined by the dynamic types arriving at run time. Such variability can be accounted for with polymorphic inline caches (PIC) [27] which map the resolved lookup results from their instances. However, the aforementioned approaches do not behave like traditional object-oriented programs, i.e., polymorphism is defined differently. To some extent, a commonality of all these approaches is that they use a Meta-Object Protocol (MOP) to embed domain-specific elements into the host language. For call sites that are part of the API of the MOP, variability can be reduced by chaining PICs, effectively constructing dispatch chains for these dynamic languages [36].

# 3 ROLE POLYMORPHIC DISPATCH

Efficient execution of role-oriented programs requires to cope with the different kind of *variability* at call sites as well as to improve *stability* to leverage optimizations of the language runtime. When a function is invoked via a polymorphic reference, late binding ensures that we get the appropriate implementation of that method for the actual object. Due to family polymorphism, the role methods have to be searched on the possible families (see Figure 5) of active teams. This section will introduce how the variability can be captured in a data structure resembling the idea of a polymorphic inline cache at the call site.

## 3.1 Late Binding of Role Dispatch

Deep nested function calls result in possibly less optimizations applied by the Just-in-Time (JIT) compiler. Especially, optimizations such as inlining use heuristics that penalize nesting. As it is shown in Figure 4 the Object Teams reference implementation uses long chains of function calls and recursive function invocations. Thus, the language runtime might not observe the connection between the call site and the resulting function calls.

However, the Object Teams compiler has access to the *bindings* which encode all relevant information of classes, methods and the respective role classes and methods. This metadata, together with run-time state of the Object Teams runtime, can be used to construct a call graph that defines, given a particular call site in a base class, the role method to be called.

We propose *Polymorphic Dispatch Plans*, a compilation strategy that supersedes the control-flow dependent dispatch, enabling optimizations by the JIT compiler. By directly linking callin functions, glue code and intermediate functions used for dispatching are avoided. The original calling convention is used as a fallback in case of unstable call sites to avoid overhead of repeatedly relinking the same call site [45]. A *polymorphic dispatch plan* is a composition of role functions and necessary type conversions at run time, conceptually representing a directed acyclic graph (DAG). Figure 6 shows a DAG of a resolved polymorphic dispatch plan of a before callin followed by the original function. The DAG consists of control flow edges represented by solid arrows and data flow dependencies as dashed arrows. Gray boxes represent data that is required when generating the DAG while white boxes are required when executing the DAG. The DAG is valid until there are too many cached DAGs at the call site. On invalidation it drops out of the role polymorphic inline cache.

To achieve this, both compilers need to be adapted. The run-time weaver needs to change the bytecode of base functions to call role functions of active teams. The static compiler needs to change how base calls are implemented. To dispatch those, the compiler needs to peek if there is a next active team or whether it has to call the base function. The implementation uses *invokedynamic* that is directly observable and walkable by the JIT compiler [42, 48]. The graph is also completely visible to the JIT compiler which allows to execute optimizations. Because the static compiler is an extension of the Eclipse Java Compiler, it must be extended to allow the definition and compilation of arbitrary invokedynamic call sites that has not been possible before.

## 3.2 Runtime Feedback in Object Teams

The Object Teams static compiler produces metadata, containing information about callins, that is compiled into Java bytecode as class attributes. Among these attributes are information about the bindings which are compiled into team classes. These attributes are read by the run-time weaver to identify and change base classes and function invocations as described in section 2. In the reference implementation, this information was only used to generate code that realizes the dispatch semantics of Object Teams. However, this metadata may also be used to identify and link callins directly from a call site.

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



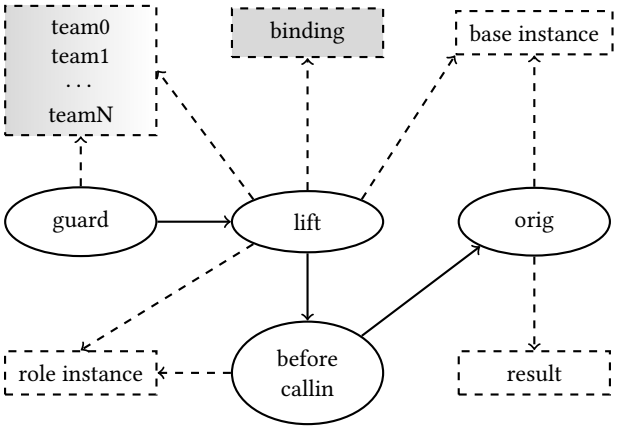


**Figure 6: A DAG of a guarded Polymorphic Dispatch Plan for a single before callin and the original function, highlighting the control flow (solid arrows) and the data flow (dashed arrows). Gray boxes are required for generation, white boxes for execution.**

The runtime of Object Teams keeps a stack of active team instances. This data structure can be used when linking call sites at run time. To include this information, the code compiled by the two compilers needs to be adapted. Prior, the static compiler enhanced the signature of callin functions to pass relevant stack frame data. This data was used to drive the dispatch, e.g., by recursively iterating over an array of teams representing the active run-time team stack.

At run time there is more information available that can be used to directly determine the call target. Given a dynamic call site, the called base function is known, as well as the team instances and classes. We use this information to query the MOP of Object Teams to retrieve the binding for the base class. The binding contains the name of the role class, the name and signature of the role function as well as other information. This is used to directly link to the implementation of the respective callin function.

Figure 7 explains the new lookup process we implemented in our approach. The left side shows a twofold initialization phase. In the beginning the call site must be bootstrapped first. During the bootstrapping phase, the uninitialized call site points to an initialization function that is associated with the call site at compile time. To be consistent with the reference implementation that function is also called `callAllBindings`. It returns a mutual call site object that is set to relink itself upon invocation and shifts the call site into the next phase.

In the initialization phase, the call site points to the relink function which in turn sets the call target depending on the run-time state to the actual role function or base function. To achieve this, we peek iteratively into the stack of active teams and retrieve the respective bindings. This is done until the first replace callin is reached effectively composing a chain of function invocations. This iterative approach replaces the recursive lookup strategy. Thus, the control-flow dependent dispatch can be removed and replaced by a call graph directly observable and walkable by the JIT compiler.

Finally, after initializing the call site it is guarded to accept *structurally* equivalent run-time states. This means, dispatching with a stack of active teams $(T_1, T_2, \cdots, T_N)$ is not dependent on the particular instances of those teams. Whenever the guard fails, the process of initializing the call site is repeated and the result is added to the known targets of the call site. Thus, it is only dependent on a specific order of team classes which resembles a polymorphic inline cache applied to contextual roles.

The right side of Figure 7 shows how the call site can optimized for specific cases, effectively eliminating the overhead of the staged role dispatch. Given our running example of an `Account` whose functions are extended by the `SavingsAccount` there is only one replace callin to be executed, namely `withFee`. Thus, the base call will directly execute the base function. In general, if a callin has a base call, it originally called deeper into the stack of active teams. This could either result in the execution of new callins until there is a replace callin or the base call will be dispatched to the base function. In our approach the base call is another invokedynamic call site that repeats the initialization phase starting at a deeper index of the stack. If the call site is called often enough it eventually is recognized by the JVM to be stabilized. The JIT compiler will emit much more efficient code for the call site resulting in a performance improvement.

## 3.3 Optimizing Role Polymorphic Call Sites

For the JVM the signatures of a base function and its role functions is incompatible. While the signature of a base function for some base class $B$ is $(C_B, Arg_B, \cdots)Ret_B$ the signature of a role $R$ defined in team $T$ is $(C_R^T, Arg_R^T, \cdots)Ret_R^T$, where there can be arbitrarily many arguments. If the elements of $Arg_B$ and $Arg_R^T$ are not directly compatible, the Object Teams compiler will generate a mapping function which maps each argument to the other. However, the classes $C_B$ and $C_R^T$ are *role polymorphic* which means on the *type level* there is a *lifting* function $lift_R^T$ such that $lift_R^T(C_B) = C_R^T$. In other words, base classes can be lifted to their respective role representation [23]. At the instance level, each *role instance* is dependent on the *team instance* it is the role playing in. For two different team instances $t_1, t_2 \in T$ and a base instance $b \in C_B$ it is $lift_R^{t_1}(b) \neq lift_R^{t_2}(b)$. Thus, a role polymorphic call site requires $(C_B, T)$ and the binding to retrieve the meta-information $C_R^T$ to find $lift_R^T$. To actually dispatch the call site the instances $(b, t)$ are required.

This information can also be used to *guard* a call site. A guard requires the argument types of the base function to match the base type $C_B$. Because in Object Teams a call site can be changed by multiple roles the guard also checks the team stack $(T_1, \cdots, T_N)$. For the latter the order is important because different orders of active teams $T_i$ result in different call graphs. When executing a dispatch plan, the guard checks the run-time type of the arguments on the stack and, if succeeding, forwards it to the resolved dispatch plan. If the guard fails it checks the next guard at the call site. If there is no next guard a new dispatch plan has to be generated that is guarded with the current types of the arguments on the stack. This effectively constructs a polymorphic inline cache (PIC) for contextual roles. The benefit of this approach is that the variability at the call site is reduced as dispatch plans can be reused and the

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



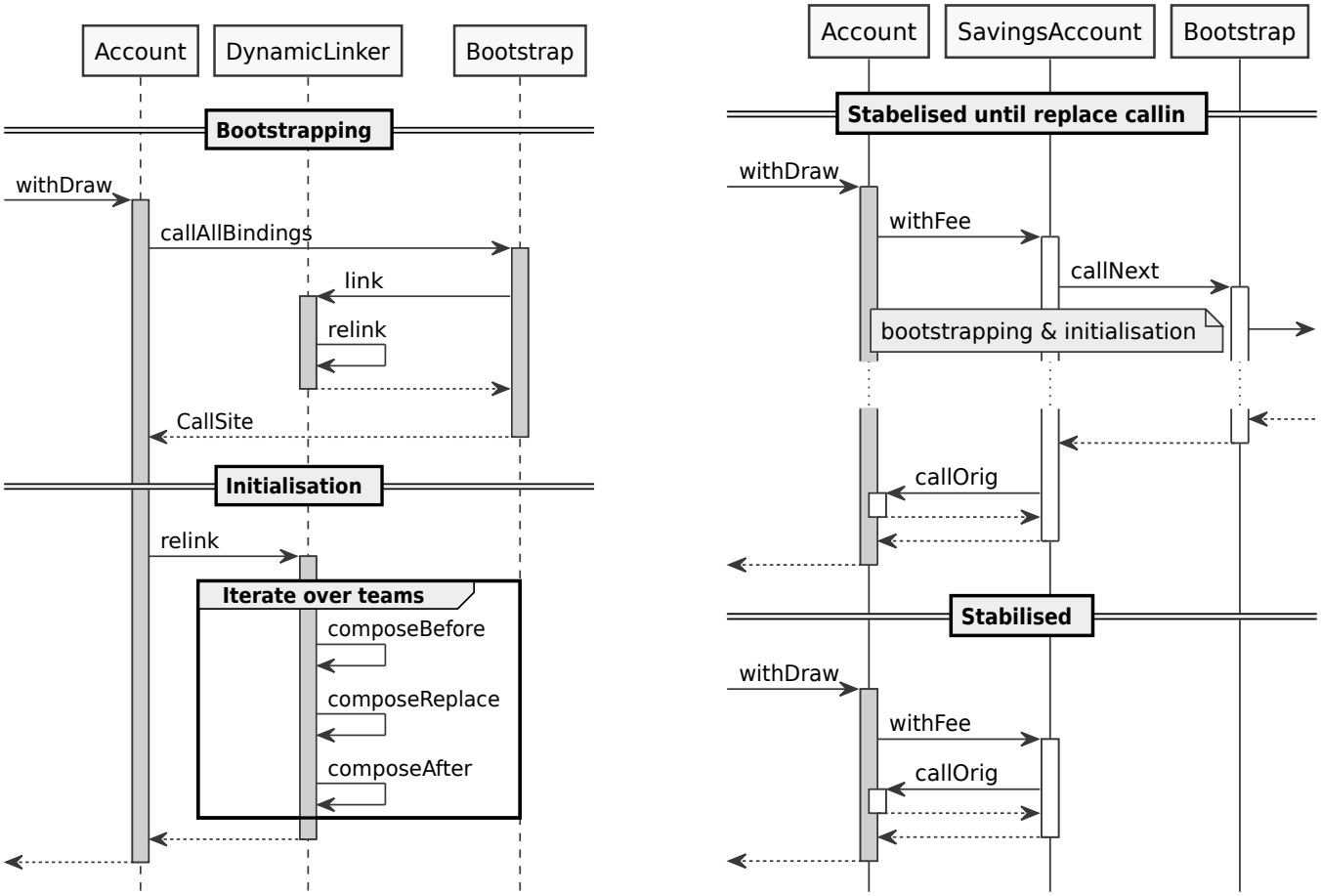


**Figure 7: Sequence diagrams of a dynamic call site where lookup results in direct calls to role methods. Left shows the initialization where the call graph is generated. The right side shows how the call site may stabilize and optimize.**

call site becomes stable enabling optimizations by the JIT compiler of the JVM. When a program is very dynamic, where role bindings change often, a call site can also be marked as unstable. A call site that is marked unstable will revert to the original lookup procedure.

## 4 EVALUATION

This section evaluates the run-time performance and characteristics of polymorphic dispatch plans and compares it with the original implementation of Object Teams. To support the usefulness of leveraging run-time type feedback to optimize the polymorphic case, the approach is also compared to dispatch plans (without polymorphism).

### 4.1 Benchmark Characterization

We used a typical synthetic benchmark we designed to compare different language implementations of the role-oriented concept [44]. The benchmark uses many demanding role-oriented programming features such as multiple active contexts, deep roles (i.e., roles playing roles), and multiple callins that are not easily built with object-oriented design patterns.

The benchmark describes a banking scenario. Persons and accounts are classes implementing basic behavior. For example, accounts can withdraw and deposit money. A bank is a compartment (i.e., context) where persons can play the role of customers. Accounts play roles that change the account's behavior such as different fees involved in withdrawing money from a checking account.

We evaluate on two variations of the benchmark, namely a *dynamic* case with variable context activations and a *static* case with a static context and no further activation, to explore different characteristics of context-dependent software. To achieve this, the dynamic case models the transaction as a team itself. This means, that in each iteration of transactions of money between two accounts the context switches twice. First when the transaction is activated and second when it is deactivated. In the static case the money is directly withdrawn and disposed from the accounts which means in the whole benchmark there is no change in the context. While software must be adaptable to changing contexts it also has to be performant enough to stay useful. We consider a case static if there is no adaptation happening for a period of time. Whenever there is a period of static behavior the language runtime should be able to

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



```
1  bank.activate();
2  for (Account from :
3    bank.getSavingAccounts()) {
4    for (Account to :
5      bank.getCheckingAccounts()) {
6      from.decrease(amount);
7      to.increase(amount);
8    }
9  }
10 bank.deactivate();
```

**Figure 8: The measured portion of the Bank benchmark static case written in Object Teams/Java. The interaction is directly with the accounts but their active roles of the bank change the executed behavior.**

```
1  bank.activate();
2  for (Account from :
3    bank.getSavingAccounts()) {
4    for (Account to :
5      bank.getCheckingAccounts()) {
6      Transaction transaction =
7        new Transaction();
8      transaction.activate();
9      transaction.execute(from,to,amount);
10     transaction.deactivate();
11   }
12 }
13 bank.deactivate();
```

**Figure 9: The measured portion of the Bank benchmark dynamic case written in Object Teams/Java. `Bank` and `Transaction` are teams whose roles influence the `Account` behavior.**

exploit it. This may be compared with the warm up of JIT compilers in dynamically compiled languages. However, if a call site registers eight adaptations it will not optimize acquiring the run-time stack data anymore but still uses dispatch plans for dispatching. Whenever the call site reached eight variations of structural different runtime stacks of teams, it will use the calling convention already present in the current implementation of Object Teams (`Classic 2020`).

To evaluate reuse, the static benchmark shown in Figure 8 does not model transactions but money is transferred directly from one account to the other. This mimics a monomorphic state at the call site where the roles of each account stays the same.

Figure 9 shows the measured portion of the dynamic case. The inner-most loop models transactions as teams which are activated and deactivated in every iteration. The accounts play the roles of the source and target of cash flow. The activation and deactivation of the `Transaction` team changes the active roles for each of its role-playing instances. While this may trigger an invalidation of the call site, the structure of activated teams can be reused.

The experiments were performed on an Intel Core i7-9700T CPU at 2.00GHz running Ubuntu 20.04 and 36GB of RAM. For the Polymorphic Dispatch Plans and the current reference implementation of Object Teams (Classic 2020) the Oracle JDK 14.0.2 was used, while the others used the Oracle JDK 9.0.4. The JVM had 4GB maximum heap space and only used the server compiler.

The benchmark used different problem sizes to evaluate the results on different inputs. In each benchmark, there are N persons having 2·N accounts (a CheckingAccount and a SavingsAccount). To capture run-time variation, the benchmark has been repeated 126 times per data point while the VM has been restarted every 42 iterations executed from a benchmarking framework [35]. We are interested in the overall execution time, normalized to the current *classic* implementation of Object Teams. To observe if there are scalability problems, we measured with different problem sizes. For the dynamic case we varied the amount of transactions from 1 to 2.5 million. The static case uses less objects and was scaled up to 6 million transactions.

## 4.2 Performance Analysis

The chart depicted in Figure 10 show the results of the reference implementation of Object Teams (`Classic 2020`) compared to our proposed `Polymorphic Dispatch Plans`. For comparison, an older version of the reference implementation of Object Teams (`Classic 2019`) as well as `Dispatch Plans` [45] were also measured. The plot shows the geometric mean of the run-time ratio, i.e., run-time factor. To compare to the current state of the art, the values are normalized to `Classic 2020`. The y-axis uses a log10 scale to highlight where each approach introduces a run-time overhead or improves over the current implementation. Lower values show better results. The error bar shows the standard derivation of the runtime ratio.

The left part of Figure 10 shows how the static case benefits both approaches, dispatch plans as well as polymorphic dispatch plans. While dispatch plans compile the active teams directly into the DAG, polymorphic dispatch plans still require them to be present on the argument stack. Because of the stable call site, the retrieval of this run-time data can be optimized. This allows implementing copy-on-write optimizations for the data structure representing the active team stack. Polymorphic dispatch plans show a geometric mean speedup of 3.8× up to 4.5× compared to `Classic 2020`. They also improve over dispatch plans by 1.4×. Executing Polymorphic Dispatch Plans on the GraalVM 20.2, does improve over `Classic 2020` by 2.8×, but is 2.6× slow than executed on JDK 14. Interesting is also the high standard derivations across the different benchmarks when running on GraalVM.

The right part of Figure 10 shows the results of the dynamic case. Dispatch Plans have a 14× run-time overhead compared to `Classic 2020`. The variability introduced by the dynamic case forces the dispatch plans to issue lots of deoptimizations. This is because they require the stack of team instances not to change to reuse generated dispatch plans and they were not designed to offer guards on run-time data. Polymorphic Dispatch Plans on the other side may reuse generated dispatch plans. Due to their guards, they can leverage from the fact that the run-time stack of teams is *structurally* equivalent. The approach of role-polymorphic inline caches achieves, for smaller inputs, a geometric mean speedup of

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



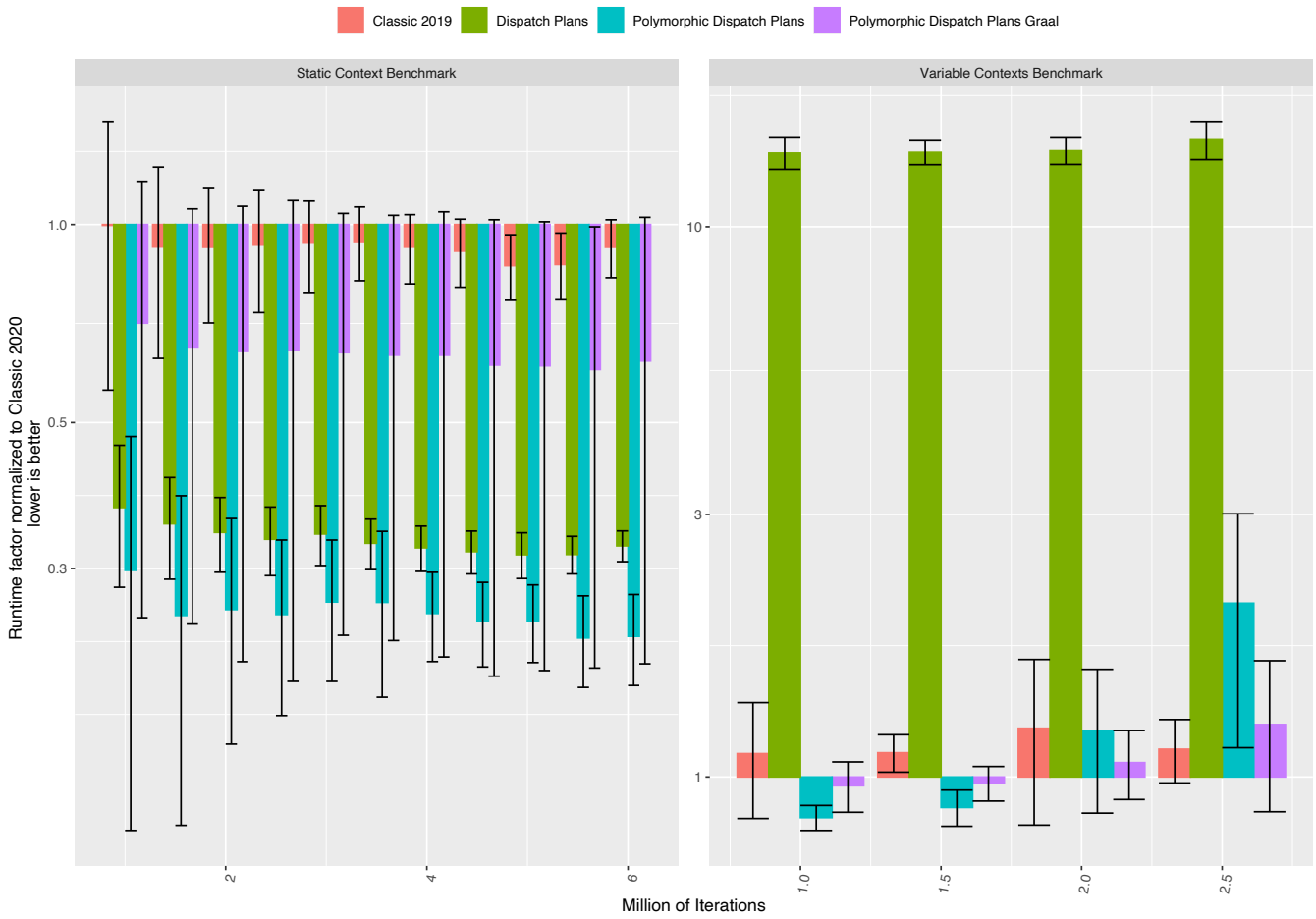


**Figure 10: The bar chart shows the geometric mean of the run-time ratio at logarithmic scale, normalized to the classic implementation of Object Teams for the respective suites. The error bar shows the standard derivation of the run-time ratio.**

1.1×. For bigger inputs the garbage collection also takes up more time which introduces noise to the values resulting in a *median* slowdown of up to 2.3× and a standard derivation of 0.95. Executing Polymorphic Dispatch Plans on the GraalVM 20.2, which is built on OpenJDK 11, reduced the slowdown to up to 1.2×. To underpin the assumption that the noise is introduced by the garbage collector we increased the maximum heap size up to 8GB. As a result, the values level off at a geometric mean speedup of 1.1×. This conclusion also corresponds to the trace records of the running JVM we analyzed manually in separate runs.

# 5 RELATED WORK

This section introduces approaches that focused on improving lookup and dispatch in approaches related to Multi-Dimensional Separation of Concerns and recent research in optimizing the inherent runtime overhead of these approaches.

## 5.1 Aspect-Oriented Programming

Since object-oriented execution environments, i.e., virtual machines, do not understand aspect semantics, the aspect compiler produces a verbose description of aspects in an object-oriented paradigm which incurs high overhead [18]. The reason is that function invocations or property accesses are typical locations for join point shadows that will be decorated with residuals. Mechanisms for late binding of advices are not applied to such high-level concept, resulting in residuals being evaluated each time. This results in a severe performance penalty ranging from two orders of magnitude in AspectWerkz [9] to performance losses of less than one decimal power [15] in Steamloom [17].

Steamloom's approach was to optimize cross-cutting, class-wide aspects by enhancing the object model of the Jikes RVM [1], compiling advice invocation code into the function body, effectively eliminating residuals, which incurs minimal overhead [8, 16]. However, aspects that only apply to specific object instances, called *instance-local* aspects, block optimizations in Steamloom. For each instance-local aspect, Steamloom compiles different versions of

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019.
S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



a function where each version is local to the object instance and associated with the applied advices. If there are multiple versions of such a function, the compiler cannot inline calls to those functions anymore which introduces performance penalties [17].

### 5.2 Context-Oriented Programming

Weaving and execution of layered methods violate assumptions that common language implementations hold about lookup resulting in inefficient code implementing the dispatch. Thus, the compiler often does not optimize or fails to optimize a specific part of the code.

In a VM with a meta-tracing JIT compiler, the JIT compiler does not optimize the executed method itself but the trace of the interpreter by specializing towards the values of the objects. By employing compiler directives such as *promote*, the compiler can optimize layered dispatch even if the heuristics did not decide to optimize the part of the application [39]. The optimized parts must be guarded and invalidated when layers change. Thus, the optimization relies on a steady state w.r.t. layer activation. However, most COP languages use method-based JIT compilers such as the Java Virtual Machine (JVM) or Google's JavaScript Engine V8 and cannot profit from this approach.

In ContextJS [34] the layered dispatch is optimized by reducing the number of method calls and lookups by inlining partial methods into a generated wrapper function [30]. Active compositions can be cached for speedup on subsequent invocations, e.g., to check if there was no change. While the inlining improves the performance by 10× it still just reaches 3% of the performance of a comparative implementation using the host language only.

An implementation leveraging an early version of dynamic execution of methods in Java uses *invokedynamic* to improve the lookup and execution of layered dispatch in JCop [3]. Instead of delegation methods a list of all partial methods is generated by the compiler. The runtime provides a callable proxy that dispatches to all partial methods when executed. The authors report a performance improvement of 48 to 35 times while being 1.75 times slower than a pure Java method in a completely static benchmark. They also highlight the huge impact of the JIT compiler when layers change.

### 5.3 Role-Oriented Programming

By taking advantage of the run time generation of dispatch code we proposed the generation of a Dispatch Plan for each call site influenced by roles [45]. Such a plan encodes the direct steps required to lookup and execute role functions which can in the best case provide up to 2.7× speedup. The approach did not take into account how to speedup call sites when the behavioral adaptations change dynamically which results in an average slowdown of up to 9.76×.

## 6 CONCLUSION AND FUTURE WORK

Context-dependent software continues to become increasingly important. The role concept is a promising candidate to build context-dependent software as contexts and behavioral adaptations can be directly represented in the language. This allows for a flexible software development process to build context-dependent software. In general, however, role language implementations suffer from a high runtime overhead when dispatching compositions of adaptations. This paper analyzed Object Teams, a feature-rich programming language for contextual roles. To reduce the overhead of dispatching role functions in Object Teams, we propose *polymorphic dispatch plans* that use the notion of directed acyclic graphs to represent a call graph constructed at run time. The concept of polymorphic inline caches is extended to role polymorphic call sites. This enables the reduction of *variability* at a call site as dispatch plans can be reused. For a demanding role-based benchmark we achieved a mean speedup of 3.8× up to 4.5×. In the worst case, when adaptations change constantly, a median slowdown of 2.3× up to a mean speedup of up to 1.1× may be achieved.

This research explored how language implementations in the host language can exploit the knowledge about runtime data to improve the code generated by the virtual machine. In future work, we consider extending a VM to support role-oriented execution semantics. Research on JIT compilers, tuning compiler heuristics for role-oriented programs and efficient runtime data structures for object-level specializations look promising. The VM is able to manage important data structures such as the active team stack, while role-playing may be captured inside the object layout. This specialization will open further optimization possibilities.

## ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907) and the Center for Advancing Electronics Dresden (cfaed).

## REFERENCES

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal* 44, 2 (2005), 399–417. https://doi.org/10.1147/sj.442.0399

[2] Egil P. Andersen and Trygve Reenskaug. 1992. System Design by Composing Structures of Interacting Objects. In *ECOOP '92 European Conference on Object-Oriented Programming*. Vol. 615. Springer-Verlag, Berlin/Heidelberg, 133–152. https://doi.org/10.1007/BFb0053034

[3] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. 2010. Layered Method Dispatch with INVOKEDYNAMIC: An Implementation Study. ACM Press, 1–6. https://doi.org/10.1145/1930021.1930025

[4] Malte Appeltauer and Robert Hirschfeld. 2012. *The JCop Language Specification: Version 1.0.* Number 59 in Technische Berichte Des Hasso-Plattner-Instituts Für Softwaresystemtechnik an Der Universität Potsdam. Universitätsverlag Potsdam, Potsdam.

[5] Matteo Baldoni, Guido Boella, and Leendert van der Torre. 2006. powerJava: Ontologically Founded Roles in Object Oriented Programming Languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing - SAC '06*. ACM Press, Dijon, France, 1414. https://doi.org/10.1145/1141277.1141606

[6] Fernando Sérgio Barbosa and Ademar Aguiar. 2012. Modeling and Programming with Roles: Introducing JavaStage. *Frontiers in Artificial Intelligence and Applications* (2012), 124–145. https://doi.org/10.3233/978-1-61499-125-0-124

[7] 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)*.

[8] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. 2004. Virtual Machine Support for Dynamic Join Points. ACM Press, 83–92. https://doi.org/10.1145/976270.976282

[9] Jonas Bonér. 2004. AspectWerkz – Dynamic AOP for Java. In *International Conference on Aspect-Oriented Software Development*.

[10] Johan Brichau, Michael Haupt, Nicholas Leidenfrost, Awais Rashid, Lodewijk Bergmans, Tom Staijen, Istvan Nagy, Anis Charfi, Christoph Bockisch, Ivica Aracic, Vaidas Gasiunas, Klaus Ostermann, Lionel Seinturier, Renaud Pawlak, Mario Südholt, Davy Suvee, Theo D'Hondt, Peter Ebraert, Wim Vanderperren,

Final edited form was published in "MPLR 2020: 17th International Conference on Managed Programming Languages and Runtimes". Virtual, 2019. S. 52–62. ISBN 978-1-4503-8853-5
https://doi.org/10.1145/2802059.2802062



Shiu Lun Tsang, Monica Pinto, Lidia Fuentes, Eddy Truyen, Adriaan Moors, Maarten Bynes, Wouter Joosen, Shmuel Katz, Adrian Coyler, Helen Hawkins, Andy Clement, and Olaf Spinczyk. 2005. *Survey of Aspect-Oriented Languages and Execution Models.* AOSD-Europe-VUB-01 Deliverable D12. Vrije Universiteit Brussel, Brussels, Belgium.

[11] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. 2006. Rava: Designing a Java Extension with Dynamic Object Roles. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems.* IEEE, 7 pp.–459. https://doi.org/10.1109/ECBS.2006.57

[12] Erik Ernst. 2001. Family Polymorphism. In *ECOOP 2001 — Object-Oriented Programming.* Vol. 2072. Springer Berlin Heidelberg, Berlin, Heidelberg, 303–326. https://doi.org/10.1007/3-540-45337-7_17

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

[14] Kasper B. Graversen and Kasper Østerbye. 2003. Implementation of a Role Language for Object-Specific Dynamic Separation of Concerns. In *SPLAT: Software Engineering Properties of Languages for Aspect Technologies.*

[15] Michael Haupt and Mira Mezini. 2004. Micro-Measurements for Dynamic Aspect-Oriented Systems. In *Object-Oriented and Internet-Based Technologies.* Vol. 3263. Springer Berlin Heidelberg, Berlin, Heidelberg, 81–96. https://doi.org/10.1007/978-3-540-30196-7_7

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

[17] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. 2005. An Execution Layer for Aspect-Oriented Programming Languages. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments.* ACM Press, 142. https://doi.org/10.1145/1064979.1065000

[18] Michael Haupt and Hans Schippers. 2007. A Machine Model for Aspect-Oriented Programming. In *ECOOP 2007 – Object-Oriented Programming*, Vol. 4609. Springer Berlin Heidelberg, Berlin, Heidelberg, 501–524. https://doi.org/10.1007/978-3-540-73589-2_24

[19] 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. https://doi.org/10.1007/3-540-36557-5_19

[20] Stephan Herrmann. 2004. *Confinement and Representation Encapsulation in Object Teams.* Technical Report 2004/06. Technische Universität Berlin, Fakultät IV-Elektrotechnik und Informatik, Berlin, Germany.

[21] Stephan Herrmann. 2005. Programming with Roles in ObjectTeams/Java. In *AAAI Fall Symposium on Roles- an Interdisciplinary Perspective.*

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

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

[24] Erik Hilsdale and Jim Hugunin. 2004. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD '04.* ACM Press, Lancaster, UK, 26–35. https://doi.org/10.1145/976270.976276

[25] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *The Journal of Object Technology* 7, 3 (2008), 125. https://doi.org/10.5381/jot.2008.7.3.a4

[26] Robert Hirschfeld, Hidehiko Masuhara, and Atsushi Igarashi. 2013. L: Context-Oriented Programming with Only Layers. In *Proceedings of the 5th International Workshop on Context-Oriented Programming - COP'13.* ACM Press, Montpellier, France, 1–5. https://doi.org/10.1145/2489793.2489797

[27] 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. https://doi.org/10.1007/BFb0057013

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming.* Vol. 2072. Springer Berlin Heidelberg, Berlin, Heidelberg, 327–354. https://doi.org/10.1007/3-540-45337-7_18

[29] 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. https://doi.org/10.1007/BFb0053381

[30] Robert Krahn, Jens Lincke, and Robert Hirschfeld. 2012. Efficient Layer Activation in Context JS. IEEE, 76–83. https://doi.org/10.1109/C5.2012.20

[31] 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. https://doi.org/10.1145/2814251.2814255

[32] 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. https://doi.org/10.1007/978-3-319-11245-9_8

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

[34] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An Open Implementation for Context-Oriented Layer Composition in ContextJS. *Science of Computer Programming* 76, 12 (Dec. 2011), 1194–1209. https://doi.org/10.1016/j.scico.2010.11.013

[35] Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. (Aug. 2018). https://doi.org/10.5281/zenodo.1311762

[36] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. ACM Press, 545–554. https://doi.org/10.1145/2737924.2737963

[37] Supasit Monpratarnchai and Tamai Tetsuo. 2008. The Implementation and Execution Framework of a Role Model Based Language, EpsilonJ. IEEE, 269–276. https://doi.org/10.1109/SNPD.2008.103

[38] Harold Ossher and Peri Tarr. 1999. *Multi-Dimensional Separation of Concerns in Hyperspace.* Research Report RC 21452(96717)16APR99. IBM T.J. Watson Research Center, New York, NY, USA.

[39] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2016. Optimizing Sideways Composition: Fast Context-Oriented Programming in ContextPyPy. ACM Press, 13–20. https://doi.org/10.1145/2951965.2951967

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

[41] 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. https://doi.org/10.1145/286936.286951

[42] John R. Rose. 2009. Bytecodes Meet Combinators: Invokedynamic on the JVM. ACM Press, 1–11. https://doi.org/10.1145/1711506.1711508

[43] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-Oriented Programming: A Software Engineering Perspective. *Journal of Systems and Software* 85, 8 (Aug. 2012), 1801–1817. https://doi.org/10.1016/j.jss.2012.03.024

[44] 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. https://doi.org/10.1145/3079368.3079386

[45] 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. https://doi.org/10.1145/3357766.3359543

[46] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data & Knowledge Engineering* 35, 1 (Oct. 2000), 83–106. https://doi.org/10.1016/S0169-023X(00)00023-9

[47] 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. https://doi.org/10.1145/2892664.2892687

[48] 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. https://doi.org/10.1145/1852761.1852763

[49] David Ungar, Harold Ossher, and Doug Kimelman. 2014. Korz: Simple, Symmetric, Subjective, Context-Oriented Programming. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software.* ACM Press, 113–131. https://doi.org/10.1145/2661136.2661147