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 3.3x in static cases, 3.0x at low variability and the same performance in highly dynamic cases.

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

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

```java
1 class Account {
2     void withdraw(float amount) { ... }
3 }
4
team class Bank {
5     class CheckingsAccount playedBy Account {
6         callin withFee(float amount) {
7             ... result = base.withFee(amount * FEE);
8             ... return result;
9         }
10     }
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.
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 $B$ (i.e., base types), context types $C$, and role types $R$ defined inside context types. Thus, a role-oriented program has types $T = C \cup B \cup R$. For example, consider a class $Account \in B$, a $Bank \in C$ which provides the role $CheckingsAccount \in R$. We use small letters to describe instances of classes at runtime, e.g., $acc$ is an instance of $Account$ written 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 \( C_i \in C \). Entries in \( \gamma \) 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 \( \gamma \) creates a runtime state \( \Gamma = (c_0, \ldots, C_{n-1}) \). Two states \( \Gamma \) and \( \Gamma' \) are structurally equal iff for each \( C_j \in \Gamma \) and \( C_j' \in \Gamma' \) the \( \Gamma' \) contains \( \Gamma \) iff \( C_j' = C_k \ldots \wedge C_j' = C_k \ldots \) for some \( k : 0 \leq k \leq n \).

For any type \( T \in T \), \( \gamma \) holds all methods declared in \( T \). Each method \( m \in M_T \) may be described by its name \( n \) and the signature \( \sigma = (\text{return}, (T_{\text{ret}}, T_0, \ldots, T_{n-1})) \), \( n \geq 0 \) consisting of the return type \( T_{\text{ret}} \) and a tuple describing the argument types. We define \( T_s = (T_0, \ldots, T_{n-1}) \) 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 calllee has to reside on the stack with the arguments. To give an example of a method signature consider the method \( \text{CheckingAccount.withFee} \) (see Figure 1) which has the concrete method signature \( \text{void, (CheckingAccount, float)} \).

### 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 \( \text{lift} \) [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 \( \sigma_R = (\text{return}, (R, T_{\text{ret}})) \) to replace \( m_B = (n_B, \sigma_B) \) with \( \sigma_B = (T_{\text{ret}}, (B, T_B)) \). While \( \sigma_B \) and \( \sigma_R \) 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:

\[
[[\sigma_B]]^\text{lift}_C = ([[T_{\text{ret}}]], ([[B]]^\text{lift}_{C}, [[T_B]])) = (T_{\text{ret}}, (R, T_R^*))
\]

### 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 \( \Gamma' \) is structurally equal to the active state \( \Gamma \). 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 \( \Gamma' \) is contained in \( \Gamma \) 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 \( \Gamma \) must be processed. Assuming there is no other context that declares a replace binding for method of \( B \) the guard will capture \( \Gamma'' \). Otherwise, the guard will capture the contained contexts \( (C_{j+1}, \ldots, C_i) \) for some \( C_i \) declaring a binding. The amount of context types captured in the guard is thereby minimized and only contains those contributing to the dispatch plan.

### 3.4 Guarded Dispatch Plans

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

---

**Algorithm 1: Construction of a Dispatch Plan**

**In:** Function \( m = (n, \sigma) \) declared in base type \( B \)

**Out:** Dispatch plan \( \mathcal{P} = (\text{guard, before, replace, after}) \)

1. `guard ← (\() \) after ← () before ← (\) replace ← (m)`
2. `\text{proceed} ← \text{true}`
3. `\text{foreach } c_i \in \gamma \wedge \text{proceed do}`
4. `C ← \text{typeof}(c_i)`
5. `\text{guard} ← (\text{guard, } C)`
6. `\text{// Resolve all bindings for } m \text{ in } C`
7. `M_c ← \text{lookupRoleBindings}(C, m)`
8. `\text{foreach } m_i = (n_i, \sigma_i) \in M_c \text{ do}`
9. `\text{// Lift signature } \sigma \text{ to } \sigma_i`
10. `m_r ← \text{liftBaseToRole}(m, m_i)`
11. `\text{switch } \text{kindof}(m_i) \text{ do}`
12. `\text{case before do}`
13. `\text{before} ← (\text{before}, m_r)`
14. `\text{end}`
15. `\text{case replace do}`
16. `\text{replace} ← m_r`
17. `\text{proceed} ← \text{false}`
18. `\text{end}`
19. `\text{case after do}`
20. `\text{after} ← (\text{after}, m_r)`
21. `\text{end}`
22. `\text{return } \text{appendAll}(\text{guard, before, replace, after})`
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.

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 \( \sigma_B \). 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 2.5x 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 3.3x 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 2.9x faster (max 3.5x) in the case in which all bindings are of type before. Role methods of type after are executed on average 3.3x faster (max 3.7x) than the original implementation. While not changing contexts the guards could be completely reused allowing to execute replace callins on average 2.9x faster (max 3.4x).

For polymorphic dispatch plans without degradation speed-ups between 3.8x to 4.5x 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 3.8x. 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].
Communicating the variability to the VM via annotations has also been proposed for COP, where sideways 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 uses 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 \texttt{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

on Pattern Languages of Programs (PLoP '97).


