Analyzing State-of-the-Art Role-based Programming Languages

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

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

ABSTRACT

With ubiquitous computing, autonomous cars, and cyber-physical systems (CPS), adaptive software becomes more and more important as computing is increasingly context-dependent. Role-based programming has been proposed to enable adaptive software design without the problem of scattering the context-dependent code. Adaptation is achieved by having objects play roles during runtime. With every role, the object's behavior is modified to adapt to the given context. In recent years, many role-based programming languages have been developed. While they greatly differ in the set of supported features, they all incur in large runtime overheads, resulting in inferior performance. The increased variability and expressiveness of the programming languages have a direct impact on the run-time and memory consumption. In this paper we provide a detailed analysis of state-of-the-art role-based programming languages, with emphasis on performance bottlenecks. We also provide insight on how to overcome these problems.

CCS CONCEPTS

• **Software and its engineering → Software performance**; *Object oriented architectures*;

KEYWORDS

Benchmarking, Role-based Programming, Optimization

ACM Reference format:
Lars Schütze and Jeronimo Castrillon. 2017. Analyzing State-of-the-Art Role-based Programming Languages. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079386

1 INTRODUCTION

The object-oriented paradigm is good at capturing the structure of a domain but fails at describing the dynamics, i.e., the collaboration between objects at run-time. Roles are an enhancement of object-orientation that can describe this kind of dynamic behavior. Dynamic behavior is extracted from classes to roles that can be bound and unbound to objects at run-time. Roles can add or overwrite the behavior of objects, effectively allowing objects to adapt. In recent years, role-based programming languages have been developed closing the gap between modeling and programming. These languages are a promising alternative to model the adaptive behavior required for future applications, e.g., in the area of autonomous driving or cyber-physical systems in general.

Role-based programming languages are young and thus have little support from tools and runtimes. In contrast, object-oriented languages have mature compilers and runtimes (e.g., Java Virtual Machine (JVM)). These infrastructures have been engineered for many years so that good designs, i.e., with logic implemented across multiple objects, do not suffer considerable performance penalties. There is no such support yet for the additional functionality brought in by roles though. Today, role-based languages are implemented using metaprogramming capabilities (e.g., Java reflection) or are pre-compiled to another language. These implementations use lots of time and space to realize the flexibility offered by the role concept. As a result, using the role concept to separately define the structure of classes as well as object collaboration results in bad performance.

Due to the lack of established benchmarks for role-based programming, authors have not assessed the performance of their individual implementations. Additionally, there is a wealth of features that have been identified across multiple role-based approaches [7]. This further complicates the selection of an appropriate language based on its performance and memory overhead. Such a selection is already difficult for more standard programming languages [9].

This paper assesses state-of-the-art role-based programming languages with a role-specific benchmark to allow cross language comparison and to identify where performance problems arise. The results show problems of the different approaches and we suggest how to optimize for performance. The main contributions of this work are (1) an analysis of current state-of-the-art role-based programming languages and (2) suggestions on how to increase the performance of these languages.

The remainder of this paper is organized as follows. Section 2 introduces the role concept, the role model of the benchmark, and state-of-the-art role-based programming languages. Thereafter, the benchmark is characterized and the results are discussed in Section 3. Section 4 discusses related work. At last, conclusions are drawn in Section 5.

2 BACKGROUND

This section introduces the role concept, the role model of the benchmark, and an excerpt of programming languages that allow to write role-based programs.

2.1 Role Concept

As mentioned in Section 1, the role concept is an enhancement of the object-oriented design. Objects can start or stop playing a role at run-time. Playing a role changes the behavior of that object. The

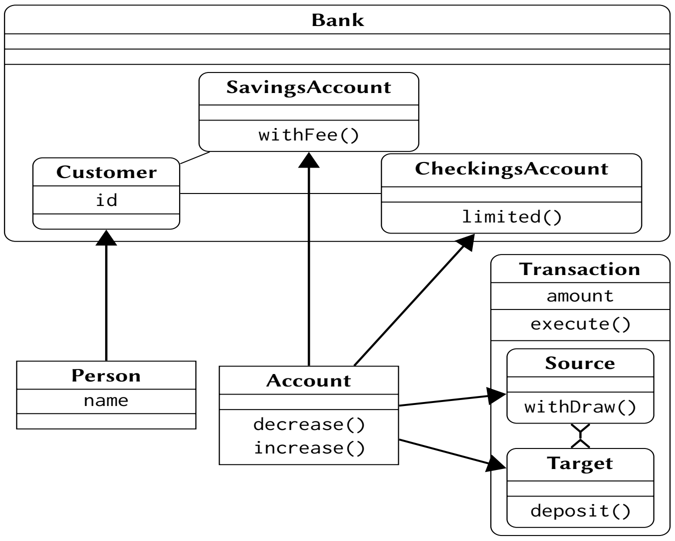

Figure 1: A simple role model of a bank. Accounts can play different roles across the compartments.

interface of an object is defined by all the roles the object currently plays. Thus, two objects of the same class can exhibit different behavior at run-time.

A role model is a model of roles, their relations, and constraints. Roles are grouped in reified contexts, called *Compartment.* As classes in object-orientation encapsulate different behaviors it is not visible what behavior causes which relationship to other classes. With the role concept, these collaborations among objects are explicitly modelled by relationships between roles. Thus, the behavior of an object is determined by the active context and the roles it plays. Changing the context effectively leads to object adaptation since it changes the roles the object plays.

Figure 1 shows a simple role model of a bank. There are `Person` objects which play the role of a `Customer` in a `Bank`. `Accounts` can be of different type in the bank, namely `SavingsAccount` and `CheckingsAccount`. Besides, there is a `Transaction` compartment, where an account can either play the role of a `Source` or a `Target`, but not both at the same transaction. For example, when playing the role of the source of the transaction, it is just allowed to withdraw money from the account.

2.2 Role-based Programming Languages

During recent years, role-based programming languages have been developed that realize the role concept using different approaches. In the following, different implementations will be introduced that are later measured and compared.

2.2.1 Role Object Pattern. Dirk Bäumer et al. have introduced the Role Object Pattern (ROP) because of the need for a flexible design pattern that allowed for unanticipated changes without the need to recompile the whole application [2]. It allows to model different views of an object designed as role objects which are dynamically added and removed from the object (i.e., the core object). This results in a segmentation of the logical entity into multiple physical entities, where the identity of each entity is different. Thus, as many design patterns, ROP suffers from object schizophrenia [5], as common object-oriented programming languages do not support delegation, but a weaker form called *forwarding*. Using role-based programming languages can help to solve the object schizophrenia problem, as compilers can enforce rules [4]. Being a simple pattern, the segmentation of entities has to be done for every single entity that should be able to play roles. Additionaly, the pattern does not take the active context into account to activate or deactivate roles.

2.2.2 Object Teams/Java. Stephan Herrmann recognized that collaborations are a crosscutting concern as multiple classes are involved [3]. Object Teams (OT)[1] is an approach to allow to define crosscutting concerns for an existing application (i.e., a posteriori adaption) extending the idea of aspect-orientation and software composition. Objects are reified into compound objects called *Team* containing all participating roles. A role has to be played by a certain type using the `playedBy` keyword. In OT these players are called the *base class.* A role can add methods, define methods that either forward to the base class (*callout binding*), or like aspects are called before, after, or replace a method of the base class (*callin binding*). Whenever there is such a *callin binding* the base method is called *bound.* Therefore, the OT compiler generates Java code and for each of the callins the compiler adds an attribute to the `.class` file of the respective `Team` class. At load time (or whenever a binding to a base method becomes known at run-time), the weaver redefines the base class: the body of the bound methods of the base class are moved and replaced with the *initial wrapper* (the entry point to the OT runtime) and the *chaining wrapper* that is used to dynamically call all bindings registered on a base method. Furthermore, a *bound method id* is stored in the base class. There is the `TeamManager` to dynamically decide which teams are active as activating and deactivating teams are registering themselves.

2.2.3 LyRT. Variability can be achieved on different levels. LyRT[2] allows to define variability on the granularity of single objects (called *dynamic binding mechanism*) by providing a Java API [13]. However, to allow role-playing the participating classes have to implement the `IPlayer`, `IRole` and `ICompartment` interfaces, respectively. The runtime provides a registry to handle all state in a central lookup table. At this registry, new cores (i.e., player objects) can be requested. The registry stores relations between compartments, players, and roles. Furthermore, it stores the level of the relation (e.g., when roles play roles the level will increase by one) and the sequence of the relation (i.e., multiple roles bound at the same level). If there are multiple roles implementing the same method, the one with the highest level and sequence will be chosen. At a given point in time only one compartment can be active. Every binding of roles to a core will be stored w.r.t. the active compartment. Variability is achieved by generating sub-classes at run-time. Into these sub-classes the dispatch logic is implemented using proxy instances. These can be exchanged without touching the core objects allowing for unanticipated adaption without restarting the application.

2.2.4 SCROLL. Recent implemention of roles into object-oriented programming languages required a specific runtime environment.

[1]http://www.eclipse.org/objectteams/
[2]https://github.com/nguonly/role4j

SCROLL (SCala ROLes Language)[3] is a domain-specific language (DSL) written in Scala that allows role-based programming without a specific runtime [8]. Conceptually, SCROLL uses a single underlying model (SUM) and provides *Views* on that model. Roles are embedded in reified contexts, called *Compartments*. Activating a compartment activates all its related roles. Thus, a compartment mimics the behavior of a view. To each compartment the role-playing state of its roles is stored in a directed acyclic graph (DAG). Furthermore, the result of a role-playing object is a compound type, that is an intersection of the role types that object is playing (i.e., Scala `with` statement). The implementation pattern requires technical aspects of Scala, such as the *dynamic marker trait*. When a function is not available on the role-playing object the compiler rewrites the function call. These dynamic dispatches are influenced by the state of the object-player DAG. It is configurable how the dispatch is searching the DAG by providing a *dispatch query*. It describes the start type and the searched type. To reduce ambiguity the types in between can be stated, as well as omitted types. The involved compartments need to be merged so their underlying DAGs are merged.

3 EVALUATION

The goal of this paper is to analyze the core problems of state-of-the-art role-based programming languages. To this end, we use a synthetic benchmark that makes extensive use of the role concept. Based on this benchmark, we perform a cross-language comparison and highlight problems w.r.t. performance, scalability and memory management for role features. Most role-based programming languages have been implemented as a Java library (SCROLL, LyRT), or use Java as a host language (Object Teams). Therefore, this work restricts the analysis to state-of-the-art role-based programming languages running on the JVM.

```
bank.activate();
for (Account from : bank.getCheckingAccounts())
  for (Account to : bank.getSavingAccounts()) {
    Transaction transaction = new Transaction();
    transaction.activate();
    transaction.execute(from, to, 1.0f);
    transaction.deactivate();
  }
bank.deactivate();
```

Listing 1: The measured part of the Bank benchmark written in Object Teams/Java.

3.1 Benchmark Characterization

The benchmark[4] consists of an implementation of the bank example shown in the role model in Figure 1. There is no single unified implementation of the role concept, instead, existing role-based programming languages offer different features. For this reason, the benchmark implementation varies slightly from language to language. For example, Role Object Pattern does not support contexts, but plain roles. In LyRT, two compartments cannot be active at the same point in time. We tried however to keep implementations as similar as possible to ensure a fair comparison.

Listing 1 shows the implementation of the benchmark in Object Teams. Constructing the bank, its customers and their accounts are not measured. As such, the benchmark measures the combined execution time of creating a new transaction compartment, its activation, the binding of roles to this transaction and at last the deactivation. The benchmark is fully deterministic, which means that on every execution the same code paths are taken.

The Role Object Pattern is the most lightweight approach to realize roles in object-oriented programming. Hence, it serves as a baseline to compare the other approaches. Object Teams have been measured in both variants; using *callin* and *callout* to realize the source and target roles of the transaction. Both can express the same semantics of the benchmark but with different performance characteristics.

The experiments have been performed on a 4-core 2.2 GHz Intel Core i7 with 16GB RAM running Mac OS 10.11.6. As virtual machine we use Oracle Java HotSpot 1.8.0_111 with arguments `JVM_ARGS="-server -d64 -Xms1024m -Xmx4048m"`. To account for the just-in-time (JIT) compilation of the JVM, the garbage collector and the underlying operating system, the benchmark has been repeated several times. More precisely, 20 times for the slower implementations and 100 times for the faster ones.

We are interested in the overall execution time of the executed transactions. To find bottlenecks in the current implementations, we measure with different problem sizes to analyze scalability problems. For problem size N there are $N \cdot N$ transactions, using N persons having $2 \cdot N$ accounts (a `CheckingsAccount` and a `SavingsAccount`). To gather information about hot methods and memory consumption every benchmark has been repeated once with activated profiling. Applications running for a longer period of time with a higher footprint on memory will result in more pressure on the garbage collector. This could make the garbage collector a dominant factor for the execution time. For a given role-based language, the memory footprint provides an indication of how well the runtime manages resources.

3.2 Results

Figure 2 shows the relative execution time normalized to the execution time of the Role Object Pattern (ROP). The accumulated amount of data written to memory is shown in Figure 3.

While the implementation using ROP just needs 511 ms for executing the 2.25 million transactions, the implementation using Object Teams callin and the callout approaches is 59.9 and 42.7 times slower, respectively. On average, the callin version is 73.1% slower than the callout. Implementations using SCROLL and LyRT can only manage up to 2,500 and 10,000 transactions, respectively. Thus, SCROLL took 12min to execute 2,500 transactions which is 610,169.5 times slower. LyRT was 12,031.4 times slower for 2,500 transactions and 84,146.3 times slower for 10,000 transactions.

The plain amount of memory used to represent all types of accounts, the transactions and the bank itself, sums up to 250 MB for 2.25 million transactions for ROP and OT. For this case, ROP wrote a total amount of 365 MB data to the heap, while OT callin generated a total of 8.9 GB, three times as much as OT callout with

[3]https://github.com/max-leuthaeuser/SCROLL
[4]Available at https://github.com/lschuetze/benchmark-lassy-2017

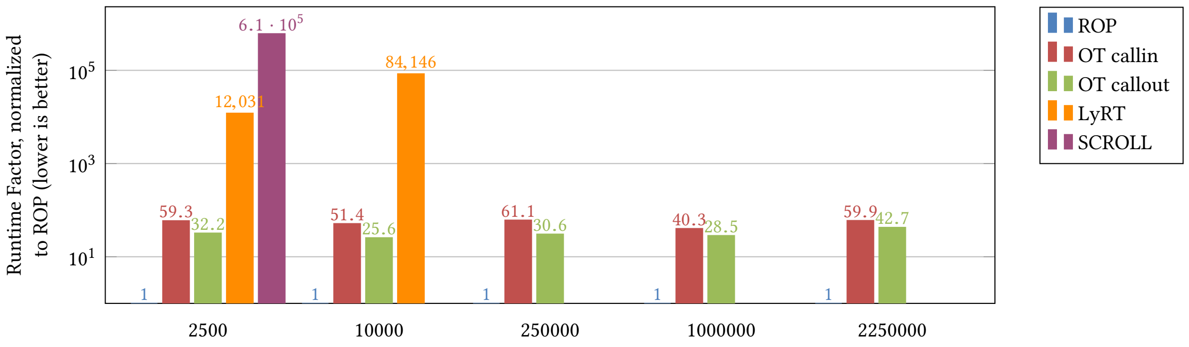

Figure 2: The relative runtime factor normalized to Role Object Pattern (ROP) in a logarithmic scale. A comparison of Object Teams (OT) using the callin and the callout approach, LyRT and SCROLL against ROP.

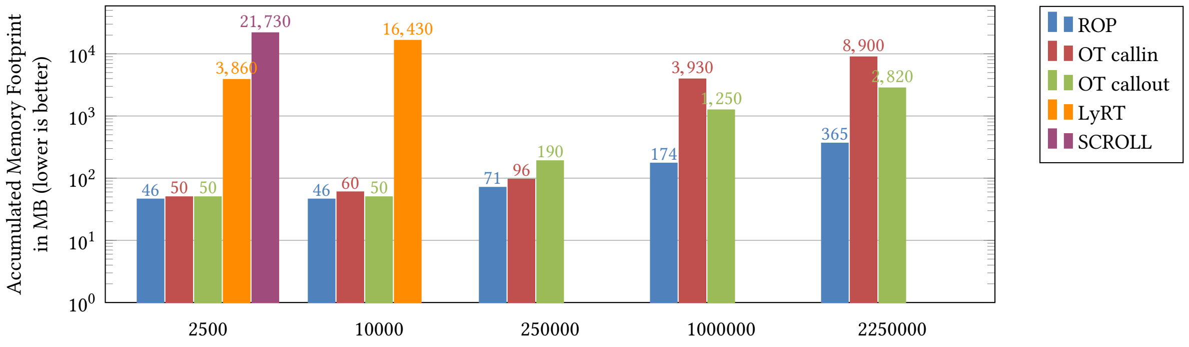

Figure 3: The accumulated amount of data written to memory during the benchmark in a logarithmic scale. A comparison of ROP, Object Teams (OT) using the callin and the callout approach, LyRT and SCROLL.

2.8 GB. SCROLL and LyRT generated 21.7 GB and 3.8 GB for 2,500 transactions, respectively. For 10,000 transactions LyRT wrote 16.4 GB of data to memory.

For Object Teams, about 58% of heap pressure accounts to callin management (5.2 GB). When an object begins to play a role in a team, the resulting role object is stored in a `WeakHashMap`. Elements in that collection can be garbage collected when free memory depletes. In the overall benchmark 17% of heap pressure (1.5 GB) is dedicated to the initialization of that cache structure.

In SCROLL, searching the data structures accounts for 78% of GC pressure (16.9 GB). Furthermore, using compound objects results in high usage of boxing and unboxing where 64% of execution time is spent when searching the role-play graph.

Using runtime subclassing in LyRT results in code being generated and stored at run-time which is responsible for 26% of the GC pressure.

3.3 Discussion

The Role Object Pattern is the fastest of all the approaches in all measurements. Even for small problems it was least 25 times faster and up to 610,000 times faster than the slowest approach. What we can clearly see from Figure 2 is that current approaches do not scale well. Figure 4 shows for each implementation where the most time has been spent during the benchmark and what contributed the most data to the heap. Some of the language runtimes spent a lot of time in special methods, others are generating too much data that needs to be processed by the garbage collector resulting in performance degradation.

As shown in Section 2, LyRT stores all relations between compartment, player, and role in a central lookup table. When a new relation is added, its level and sequence has to be calculated. This accounts for 89% of the execution time of the benchmark. That is, because there is an $O(R^2)$ search over that structure, where R is the number of relations (e.g., customers, accounts, and transaction roles). Currently, the lookup table is an `ArrayDeque`[5] that is explored multiple times. There is a need for a dedicated structure that allows faster search and more compact storage of the data.

Second, generating a new subclass for every player object and role object accounts for 26% of the overall GC pressure during the benchmark. For each of the generated classes a variable responsible to manage dispatch logic is *captured*. To reduce the pressure the amount of generated subclasses has to be reduced. A generated subclass per player type or role type can be stored in a type cache. Instead of capturing, a new modified constructor can be provided that provides a dedicated parameter.

In SCROLL there is a directed acyclic graph (DAG) for every compartment where players and their roles are stored. SCROLL

[5] An `ArrayDeque` is a performant collection from the Java Collections Framework.

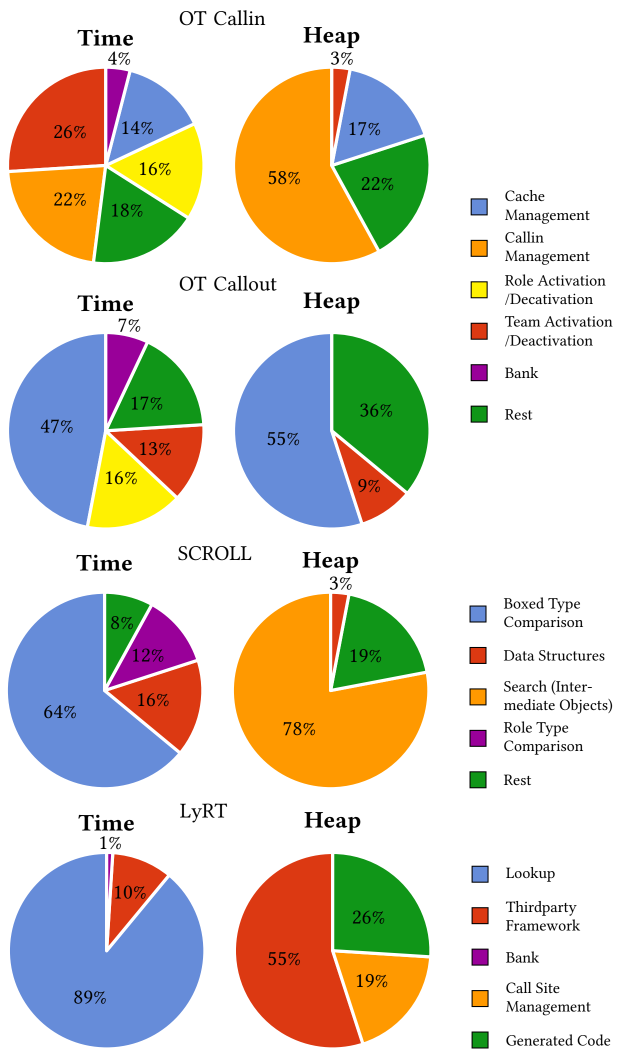

Figure 4: Shows where each implementation of the benchmark spent most time (left), and what contributed the most data to the heap (right).

allows the user to define custom dispatch queries (e.g., searched types), which influence the method dispatch for that compartment. This flexibility for the method dispatch accounts for 64% of the execution time being spent in checking the equality of objects in the DAG. In the benchmark the dispatch accounts for 78% of all data written to the heap (16.9 GB) due to intermediate objects that are repeatedly generated.

Looking at Object Teams, we can see that using the callin approach is always slower than using callout. This is expected since both have different semantics (advice vs. forwarding) with different performance characteristics. To realize callins the runtime assembles string IDs out of smaller parts (e.g., a base method ID consisting of the class name, parameter names, and their types). Recalculating the values results in 58% GC pressure during the benchmark. Caching these results could improve the performance as the multitude of intermediate strings do not have to be created. Furthermore, a proper use of the String pool provided by the JVM could further reduce the amount of data used by these IDs. Using caches to store the result of a dispatch could also lead to a reduction in space and time for the callin approach. Invalidation has then to happen when a Team is being activated or deactivated and the call site is subject to change.

Using software caches is not always a solution to increase performance. Each binding of a base instance to a role instance is cached in the `Team` instance. When the same base object enters the team again played roles will be reused from the cache. Constructing and initializing this cache structure accounts for a total amount of 1.5 GB of data during the benchmark. This is 6 times as much as the data to be stored (i.e., player objects, role objects). Furthermore, as the overhead is neglectible for the callin approach, it is responsible for 47% of the whole execution time using the callout approach. A proposed solution is the introduction of annotations for roles and teams which tell the compiler and runtime not to use caches. This is useful for stateless roles and allows the application of the flyweight or prototype pattern.

More powerful realizations of the role-concept incur in a large overhead compared to the role-object pattern. As discussed above, the overhead stems from the way features are realized in the runtimes themselves. There are two problems related to the algorithms and data structures. First, most of the execution time is spent on searching the data structures. Second, searching the data structures results in many copies and high pressure on the garbage collector. Both degrade the performance of an application.

4 RELATED WORK

In this section, benchmarks from similar disciplines and other role-based programming languages are introduced.

JAWIRO (JAva WIth ROles) is a framework that enhances Java with roles [12]. Nowadays, the framework is not available anymore. The framework has been compared to other design patterns and has been measured using micro-benchmarks. The benchmarks measure how fast operations on roles behave in case of an increasing number of role instances in the stored hierarchy.

EpsilonJ is a role-based programming language that enhances Java with roles [10]. EpsilonJ programs are translated to standard Java. The effectiveness of the generated code is compared against hand-written Java code. The measurements include compilation and execution time. They conclude that compilation time is not a significant factor, but execution time was two times slower than hand-written Java.

ContextJS is a context-oriented extension to JavaScript [6]. They discuss different optimization techniques to improve context-oriented programming (COP) with JavaScript. To discuss these techniques they employ micro-benchmarks to measure the execution time of layer activations and dispatches to methods. They conclude that most time is spent in dispatching.

A common approach to speed up dispatching is using caches and to invalidate those caches when the cached result of the dispatch is not valid anymore. This can happen when for example a new layermposition changes the implementation of an already cached call site. In ContextPyPy [11] another approach is chosen. Instead of using a cache the capabilities of the meta-tracing JIT compiler is used. That is, the steps the interpreter takes are recorced. This instruction sequence is called a *trace*. Language implementers can use *hints* to allow fine tuning of the JIT compiler. As such, the JIT compiler can reuse recorded traces when layer compositions are stable enough. Therefore, a guarded switch is inserted into the code. As long as the composition stays the same, the lookup can be optimized accordingly. For the evaluation the execution time under different workloads has been compared across context-oriented programming languages.

In `JCOP`, a Java implementation of a context-oriented programming language, the dispatch has been implemented using the new `invokedynamic` bytecode instruction [1]. The language implementer provides a `bootstrap` method that is being called the first time when the JVM wants to execute the invokedynamic bytecode. In this method, the dispatch is implemented. The result is a `method handle` that points to an actual method. On further invocations the method handle is invoked and there is no need to dispatch again. Their implementation replaced the traditional composition lookup with an invokedynamic instruction. When a composition is changed (i.e., a layer is added or removed) the method handle of the call site is updated. They measured a proof of concept implementation of JCOP with invokedynamic compared to the unmodified JCOP. They report a speedup of 160 times when there is no layer activated, as well as a speedup of 48 to 38 times for 1 - 5 layers activated. For language implementers invokedynamic seems to be a promising start as there is lots of speedup to gain when

In summary, none of the presented approaches measured memory usage beside execution time. There have been different approaches to improve the performance, e.g., the usage of caches to reduce the amount of recalculation or a similiar approach using a tracing JIT compiler to trace and replay stable parts of the program, i.e. layer compositions.

5 CONCLUSIONS

The role concept is a good candidate to model adaptive, context-aware systems. It allows modeling the static structure of a domain separately to the dynamic behavior of objects at run-time. This overcomes the limits introduced by the class-centric models of object-orientation. In recent years, many frameworks and languages have been proposed that map the concept to programming languages.

For use in production, performance is a critical aspect. However, the proposed approaches concentrated on implementing the many features the role concept has to offer. Thus, these frameworks and programming languages incur in large runtime overhead. To help to improve the situation we present an analysis of state-of-the-art role-based programming languages.

Our evaluation observed that state-of-the-art role-based programming languages do not just take a considerable amount of time in dispatch logic, but also generate lots of data during runtime.

Benchmarking programming languages often comes down to calculating execution times. With the higher complexity introduced by the role concept, it is considerably important to also look at the amount of data written to memory by the runtimes. With managed runtimes pressuring the memory results in more garbage collection runs which also degrades the performance of the application.

In future work, we will focus on increasing the single benchmark to a macro benchmark suite for role-based programming languages. This will be very helpful for guiding further optimization, as different usage patterns can be evaluated. Micro benchmarks on the other hand, will offer good opportunities to optimize performance of critical parts of the runtimes.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. 2010. Layered Method Dispatch with INVOKEDYNAMIC: An Implementation Study. In *Proceedings of the 2Nd International Workshop on Context-Oriented Programming (COP '10)*. ACM, New York, NY, USA, Article 4, 6 pages. DOI: http://dx.doi.org/10.1145/1930021.1930025

[2] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. 1998. The Role Object Pattern. In *Washington University Dept. of Computer Science*.

[3] Stephan Herrmann. 2003. *Object Teams: Improving Modularity for Crosscutting Collaborations*. Springer Berlin Heidelberg, Berlin, Heidelberg, 248–264.

[4] Stephan Herrmann. 2010. Demystifying Object Schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI '10)*. ACM, New York, NY, USA, Article 2, 5 pages.

[5] Elizabeth A Kendall. 1999. Aspect-oriented programming for role models. In *ECOOP Workshops*. 294–295.

[6] Robert Krahn, Jens Lincke, and Robert Hirschfeld. 2012. Efficient Layer Activation in Context JS. In *Creating, Connecting and Collaborating through Computing (C5), 2012*. IEEE, 76–83.

[7] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. *A Metamodel Family for Role-Based Modeling and Programming Languages*. Springer International Publishing, 141–160.

[8] Max Leuthäuser and Uwe Aßmann. 2015. Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering (MORSE/-VAO '15)*. ACM, New York, NY, USA, 25–33.

[9] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 120–131.

[10] Supasit Monpratarnchai and Tamai Tetsuo. 2008. The design and implementation of a role model based language, EpsilonJ. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008. ECTI-CON 2008.*, Vol. 1. IEEE, 37–40.

[11] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2016. Optimizing Sideways Composition: Fast Context-oriented Programming in ContextPyPy. In *Proceedings of the 8th International Workshop on Context-Oriented Programming (COP'16)*. ACM, New York, NY, USA, 13–20. DOI: http://dx.doi.org/10.1145/2951965.2951967

[12] Yunus Emre Selçuk and Nadia Erdoğan. 2004. *JAWIRO: Enhancing Java with Roles*. Springer Berlin Heidelberg, Berlin, Heidelberg, 927–934.

[13] 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 (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 137–142.