Preexistence and concrete type analysis in the context of multiple inheritance

Roland Ducournau, Julien Pagès
LIRMM, CNRS - Montpellier Univ., France
firstname.name@lirmm.fr

Jean Privat
Université du Québec à Montréal, Canada
privat.jean@uqam.ca

ABSTRACT

In the framework of just-in-time compilation, *preexistence* is a property which asserts that the receiver of a given call site has been instantiated before the current invocation of the considered method [5]. Hence, preexistence is a mean to avoid such low-level repair techniques as *on-stack replacement* or *code patching*, when a method must be recompiled during its own activation, because a previous optimization such as *devirtualization* or *inlining* has been invalidated by a new class loading. In the original proposition, preexistence considers only values, its analysis is purely intra-procedural, and its result is immutable. In this paper, we reconsider all three aspects: we focus on types instead of values, especially on concrete types, and we accept a limited form of inter-procedural analysis, whose results can be, marginally, mutable. Moreover, while the original proposition considered only method invocation, we place ourselves in the context of full multiple-inheritance, where attribute accesses need to be optimized, too. Overall, we propose a static analysis at runtime, combining concrete types and preexistence. The former allow for more optimizations, while the latter provides the condition for applying these optimizations without any need for repair. We experimented the approach in the prototype of a virtual machine for the Nit language, and the results are really encouraging. In spite of the focus on multiple inheritance, and the experiment setting, this proposal could be applied to most languages with static typing and runtime systems like the Java virtual machine.

CCS Concepts

•**Software and its engineering → Object oriented languages; Inheritance; Just-in-time compilers;** *Classes and objects; Runtime environments;*

Keywords

Multiple inheritance, devirtualization, inlining, preexistence, concrete types

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland
© 2016 ACM. ISBN 978-1-4503-4135-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2972206.2972207

1. INTRODUCTION

Just-in-time recompilation and repair

Modern runtime systems like the Java virtual machine (JVM) rely on dynamic, just-in-time compilation, and their efficiency results from the aggressive optimizations performed by the compiler. These optimizations are aggressive in the sense that they rely on assumptions which are valid for the moment, without any guarantee for the future. It may thus occur that the dynamic loading of a class invalidates an assumption underlying the optimized compilation of a method which, thus, must be *recompiled.* When the invalidated method is currently active, for instance when its current invocation has provoked the aforementioned class loading, the current compiled code must be *repaired*, too.

In this framework, three optimizations are essential: (i) *devirtualization* involves substituting a static call to a more complex late-binding sequence; (ii) *inlining* can then be applied to a static call, in order to inline the callee in the caller; (iii) *code specialization* involves specializing a method (or a fragment of method) according to a specific calling context. In this paper, we focus on the first one, but our ultimate goal includes all of them.

Let us consider, for instance, a method `baz` containing the expression `x.foo().bar()`, where the static type of `x.foo()` is `A`. Then, suppose that, at the time of its compilation, `A` is loaded and has no loaded subclasses. Then, the compiler might decide to devirtualize the call to `bar`, by calling statically `A.bar`. However, it might happen that some indirect action of `foo` would be to load a subclass `B` of `A`, eventually returning an instance of `B`. Therefore, if `bar` is redefined in `B`, the static call to `A.bar` will be erroneous. This makes mandatory to repair the code of `x.foo().bar()` while it is active. This is, of course, a simplified example which can be complicated in several directions, for instance because `bar` has been inlined, or `B` has been loaded in another thread. Anyhow, the compiled code for calling `bar` must be repaired, and there are, classically, three ways to tackle this issue:

Guards: a first approach just avoids the need for repair, by guarding the optimized version and executing the non-optimized one when the guard fails. *Class guards* and *method guards* have been considered [5]. The cost of the former is low but they are too narrow, while the later are more accurate, but also more costly. Their drawbacks have been mostly fixed with *thin guards* [2].

On-stack replacement: in a second approach, there are no guards, and the whole method `baz` is recompiled when `B` is loaded, in order to deoptimize the call to `bar`;

then, the new version is assigned to the corresponding entries of the concerned method tables. Usually, it can be done lazily, with a *trampoline*, and this recompilation will account for future invocations of `baz`. However, it cannot solve the point of the current invocation. Then, *on-stack replacement* [10, 9, 19] involves patching the stack in order, for instance, to replace the return address of the call to `foo` by the corresponding address in the newly compiled code.

Code patching: this alternative [11] avoids the complete recompilation of a method by just patching the current compiled code of `baz`, replacing the static call to `A.bar` by a call to a stub function implementing the virtual call that dispatches to either `A.bar` or `B.bar`. This stub function can be generated on the fly, or may have been precompiled at the first compilation of `A.baz`. Thus, code patching avoids on-stack replacement, too.

Guards are robust and safe, but they suffer from two drawbacks: (i) critical parts of the code must be duplicated; (ii) whereas one may expect that the need for dynamic repair should decrease asymptotically during the execution of a program, guards will be checked perpetually. The two other guardless techniques are complementary and fairly delicate, since both involve patching, either the stack or the code, and patching is not that portable. Furthermore, patching is essentially not lazy. Overall, all three techniques work well, but it would be far better to be able to avoid any of them.

Preexistence

Preexistence is a property which asserts that the receiver of a given call site has been created before the current invocation of the including method [5]. Preexistence offers an important guarantee on compiled methods: when a method must be *recompiled*, the current compiled code of the call sites whose receiver is preexisting to the method invocation remains correct for the current activation and doesn't need to be *repaired*. The consequence of the guarantee is that method repair remains needed only for methods (i) that are currently active, and (ii) contain an invalidated call site with a non-preexisting receiver.

The authors used preexistence to decide on inlining for static calls with a preexisting receiver. In the context of the Java language, two kinds of receiver were considered, namely *invariant arguments*, i.e. the formal parameters of the method to which no assignments are made, and *immutable fields*, i.e. fields that are only assigned to in a constructor. Their experiments on a small set of Java benchmarks showed that the proportion of call sites with a preexisting receiver was highly significant. Overall, preexistence analysis is very often cited in the literature, and considered as a well-tried technique, complementary to the various repair techniques proposed, e.g. [11, 2].

Multiple inheritance and better preexistence

In this paper, we enlarge preexistence in both its usage and its analysis. We consider languages with full multiple inheritance, implemented with *perfect hashing* (PH) [6, 7]. With multiple inheritance, one has to consider three kinds of object-invocation site, for method calls, attribute (aka field) accesses and subtyping tests (e.g. casts). The PH implementation has been designed for use in a dynamic-loading setting, as a generalization of the usual single-inheritance—actually single-subtyping (SST)—implementation. Detailed experiments showed that the PH overhead over the single-subtyping implementation is not negligible [8], and similar to that of the subobject-based implementation used in C++virtual multiple-inheritance [13]. Therefore, the efficient way of using PH implementation involves optimizing it by using the single-subtyping implementation when both coincide. The details are irrelevant, here, and interested readers are referred to the cited articles. The only points to keep in mind is that (i) repairing a method could concern all of these three kinds of site; (ii) repair might simply involve switching from SST to PH implementation. In contrast, in languages with multiple subtyping, like Java, the considered optimizations don't concern at all attribute access. Moreover, the literature on these dynamic optimizations doesn't often consider subtyping tests, whereas they are eligible to optimizations similar to devirtualization. Indeed, when the test is always true or always false, it is similar to a static call for method invocation. Otherwise, when the target type is an interface, it can be optimized as a class test if the interface is directly implemented by a single class.

Whereas code patching is a practical solution that deserves to be considered for non-inlined method invocation, because it just involves replacing a function call with another function call, it seems to be markedly less efficient for small inlined sequences of code, especially with attribute access and subtyping tests, because the patch generally requires more registers than the optimized code. Hence, some context must be saved and restored, and it would significantly degrade the efficiency of attribute accesses. There is thus a need for a more accurate preexistence analysis, which would increase the proportion of preexisting receivers. Therefore, in this paper, we extend preexistence analysis in several directions. Firstly, we consider preexistence of types, not only of values. Indeed, the considered optimizations depend only on the dynamic type of the receiver. This makes us consider, in a more general way, *concrete types*, i.e. the set of the possible dynamic types of an expression, at the time of its compilation. Secondly, we extend the analysis to other kinds of expression than parameters and attribute readings. We consider also casts and call sites, especially through a limited form of inter-procedural analysis.

Outline. Section 2 presents the context of our proposal, with an intermediate representation that can be viewed as an abstraction of the bytecode, along with a sketch of a just-in-time load-and-compile protocol. The next section states the preexistence principle, and its extension through concrete-type analysis. Both sections are essentially language-independent. Section 4 presents the results of our first experiments in the prototype VM of the Nit language [15]. A few related works are then discussed, and the paper ends with a conclusion and a plan for further studies.

2. CONTEXT

We place ourselves in the context of a virtual machine for an object-oriented language with static typing, which behaves roughly like the Java Virtual Machine: units of code are loaded and compiled, lazily, i.e. just in time.

2.1 Bytecode abstraction

We consider a bytecode abstraction, which can be viewed as a metamodel or an intermediate representation, and involves, for each method, the following items:

- a list of *input parameters*, including a *receiver* (`this`);
- a set of *literals* (mostly for `null`, singletons, and enumerated values);
- sets of *object-invocation sites*, split in four kinds: (i) *call sites*, for method invocation; (ii) *read* and *write sites*, for attribute accesses; and (iv) *cast sites*, for subtyping tests or casts (the site is assumed to return its receiver, typed with the cast target-type); each site has a *receiver*, and a call site has a list of *extra arguments*, and both receiver and arguments are *expressions*;
- a set of *instantiation sites*; they are not object-invocation sites, but they include a site calling the constructor;
- a set of *variables*, each variable depending on one or more *expressions*; if variable `x` depends on expressions `{e1,e2}`, it means that `e1` and `e2` have been assigned to `x`; dependence between variables is assumed to be acyclic; one may think of static-single assignment (SSA) variables, where multiple dependences stand for ϕ-functions, and variables in cyclic dependences are merged;
- a distinguished variable for the *value returned* by the method;
- write sites excepted, all these elements are *expressions*;
- finally, the *dispatched methods* of a call site are the methods that could be invoked according to the currently loaded classes. By default, they are computed with *class hierarchy analysis* (CHA [4]), on the subset of classes currently *instantiated*. We use the exact *concrete type* of the receiver when it is known.

Expressions with primitive types are not considered: such literals, parameters and variables are simply disregarded, and method invocations are considered as returning nothing. For the sake of factorization, object-invocation sites are grouped in *site patterns* which are determined by the receiver's static type (*rst*) and the invoked method, the accessed attribute or the target type. The sites of the same pattern share the same implementation, and the same dispatched methods, unless their *concrete type* is known.

In this intermediate representation, everything is static and immutable—although computed lazily—, apart from the dispatched methods which form ever-increasing sets.

It is worth noting that the two terms *invocation site* and *invocation expression* denote the same entity from, respectively, the caller and callee points of view: the former considers the receiver of the site, whereas the latter considers the value of the expression, which can be returned by the method or used as the receiver of another site. For instance, in the `x.foo().bar()` expression of our introductory example, the preexistence of the expression `x.foo()` is a condition of the preexistence of the site calling `bar()`.

2.2 Object implementation and optimizations

We focus on an implementation of objects based on *perfect hashing* [6, 7]. This implementation can be understood as an extension to multiple inheritance of the implementation of Java. Without loss of generality, an object is an array of attribute values, and it references a method table formed of an array containing, for each method known by the object class, the address of the implementing code. A key feature of this implementation is that the attributes (resp. methods) introduced by a class are grouped together in the layout. This allows classes in single inheritance to be implemented exactly like classes in Java. There are then several ways of implementing a given *object-invocation site*:

- *single-subtyping* (SST) implementation works only in situations akin to single inheritance, i.e. when the target method, attribute or type has always the same position in the *loaded* subclasses of the receiver's static type (*rst*); it is commonly used for Java classes;
- *perfect hashing* (PH) implementation works in every situation; although constant-time, it is less efficient than SST implementation; PH is a solution for Java interfaces;
- *static* implementation is unconditional, and involves *devirtualization* for call sites with a single dispatched method; it applies also to cast sites, when the test is guaranteed to always succeed, or to always fail;
- a last implementation, called *final*, is available for casts, when their target type has no loaded subclass, e.g. it is *final*. Then, the test can just involve comparing the method-table address of the target with that of the object. In the statistics presented hereafter, final and static cast implementations will be grouped together.

These four implementations are ordered according to both their increasing efficiency and their decreasing applicability:

$$\textit{static} < \textit{final} < \textit{SST} < \textit{PH}$$

When an object-invocation site is compiled, two implementations must be determined: (i) the *optimistic* implementation is the most optimized implementation applicable according to the currently loaded classes, as if the world was closed, regardless of the recompilation issue; (ii) the *conservative* implementation is the most optimized implementation that will never require recompiling the site. In general, the latter is PH. It can be SST, for a method introduced in `Object`, the root of the class hierarchy; static, when the concrete type of the receiver of a call is known; or final, when the target type of a cast is final.

When an object-invocation is compiled, the issue is thus to determine the optimal choice between the conservative and optimistic implementations, i.e. the best trade-off between the efficiency of the generated code and the cost of the possible future recompilations. A site is said to be *optimized* if the implementation selected for its compilation is strictly more optimized than its conservative implementation, hence

$$\textit{static} \leq \textit{optimistic} \leq \textit{optimized} < \textit{conservative} \leq \textit{PH}$$

.

2.3 Load and compile protocols

Overall, we consider runtime systems, akin to that of Java, which involve three main processes: (i) a *static compiler*, like `javac`, compiles units of code (e.g. classes) into a bytecode; we assume that this compiler does specific static analyses, and produces appropriate annotations in the bytecode; (ii) a *class loader* loads classes and creates relevant data structures; (iii) a *just-in-time compiler* compiles the bytecode of methods into machine code. We thus assume some abstract load and compile protocol based on the general idea that both class loading and method compilation are lazy and performed just-in-time, via trampolines. Furthermore, we assume that the preexistence analysis will take place just before a method is compiled.

Lazy compilation and trampolines. *Trampolines* involve filling the still uncompiled method entries in method tables with the address of a stub function which will compile the method, fill the method entry with the resulting address, then jump to it. Thus, lazy compilation of a method which is called via late-binding is both simple and efficient, and can be done at a reasonably high level. When the considered function is called statically, lazy compilation requires some low-level code patching, for instance by replacing, in the original compiled code, a jump to the trampoline address by a jump to the newly compiled address.

Variants. In order to make the design space more concrete, we present several variants. Variant 1 is defined as follows:

1. a class `A` is *instantiated* when a site `new A` is *executed* for the first time;
2. when `A` is *instantiated*, it is first *loaded*, if needed, and its method table is allocated and filled; then, the definitions of methods in `A` are propagated, as new dispatched methods, into all the sites concerned;
3. when a class is *loaded*, its direct superclasses are first recursively loaded, if needed, and the layouts of its method table and instances are computed;
4. a method is *compiled* when the *execution* of a call site dispatches to this method for the first time;

With Variant 2, Rule 1 is replaced with the following:

1'. a class `A` is *instantiated* when a site `new A` is *compiled* for the first time;

Rule 1' would avoid extra recompilations, at least when the instantiation is effective. The extra costs would occur when the instantiation is guarded, and loading this class might enlarge the call graph with never-compiled methods. A good trade-off involves applying Variant 2 when the instantiation site is unconditional, and Variant 1 otherwise, and it could be an annotation in the bytecode.

Finally, with Variant 3, all methods called statically would be compiled when their caller is compiled. Then rule 4 is complemented with

4'. a method is *compiled* when its static caller is compiled for the first time.

As for Variant 2, a good trade-off might be to apply Variant 3 for unconditional static calls. However, while there is no drawbacks to load a class when compiling a method which will unconditionally instantiate it, it is not the same, here. Indeed, if method `foo` calls `bar`, many things may happen between entering `foo` and calling `bar`, and the anticipated compilation of `bar` might be invalidated before its first use. Therefore, Variant 3 is likely not a good option, even for unconditional calls.

3. PREEXISTENCE ANALYSIS

3.1 Original preexistence

According to the original definition [5], an expression is *preexisting* in its enclosing method, if its *value* must necessarily have been *created* before the current invocation of the method, hence before the compilation of the method. Therefore, the compiler can guarantee that the optimistic implementation selected for a site will remain sound *during the current invocation of the method*, even if it is invalidated.

The proof of preexistence was based on the following set of rules, here adapted to our intermediate representation:

Parameter-P: all input parameters are preexisting;

ImmutableAttribute-P: a read expression is preexisting if its receiver is preexisting and the read attribute is guaranteed to be immutable; this guarantee can be offered by specific language features like `final` in Java or `val` in Scala; without loss of generality, it must be determined statically, at compile-time, and translated into a bytecode annotation;

Variable-P: a variable is preexisting if all the expressions which it depends on are preexisting.

In principle, as it is used only to select how an object-invocation site is compiled, preexistence should be a matter of dynamic types, not of values. However, the original definition was formulated only in terms of values, as if the only way to assert that the type of an expression is preexisting would be to assert that its value is preexisting. Consider the following example, supposing that `B` is a subclass of `A` and bothe have been already loaded:

```
foo (A x) {
    A y;
    if <some condition> {
       y=new B(); y.bar1(); }
    else  {
       y = x; y.bar2(); }
    y.bar3();     }
```

Then, according to original preexistence, `y` will be considered as preexisting in `y.bar2`. In contrast, it is not preexisting in `y.bar1`, but its concrete type is statically known and the site will be compiled as a static call. However, original preexistence fails to capture the preexistence of `y` in `y.bar3`, in spite of the fact that, here, the implementation of `y` can be considered as preexisting, since it is either that of `x` or that of `B`. This makes us distinguish between two kinds of preexistence:

- an expression is *type-preexisting* if its *dynamic type* must necessarily have been *instantiated* before the current invocation of the including method (`y.bar1`);
- an expression is *value-preexisting* if its *value* must necessarily have been *created* before the current invocation of the including method (`y.bar2`).

Of course, value-preexistence implies type-preexistence, and merging both forms through multiple dependences yields type-preexistence, too. Thus, `y` is type-preexisting in `y.bar3`. Preexistence is a property of expressions, not of types (they just may be loaded or instantiated).

3.2 Extended preexistence of expressions

We propose the following set of rules, which enlarge preexistence analysis in two directions: (i) with *concrete types* because they provide the basis for type-preexistence along with an opportunity to detect more optimized implementations; and (ii) with interprocedural analysis, because it could take into account frequent patterns like *factory* methods.

Literal-P, Parameter-P: literals and input parameters are value-preexisting;

Variable-P: a variable is value (resp. type) preexisting iff all the expressions which it depends on are value (resp. type) preexisting;

ImmutableAttribute-P: a read expression is value-preexisting if its receiver is value-preexisting and the read attribute is guaranteed to be immutable;

ConcreteType-P: the *concrete type* of an expression is the set of its possible dynamic types, if this set is known statically. Then, the expression is type-preexisting if all the classes in its concrete type are *loaded*[1]. Rules for concrete types will be described later.

Return-P: the return of a method has the preexistence of its distinguished return variable[2].

Call-P: a method-invocation expression is value (resp. type) preexisting if (i) its receiver and arguments are all value (resp. type) preexisting, and (ii) the returned values of all its dispatched methods are value (resp. type) preexisting; for the sake of simplification, recursive cases are considered as non-preexisting;

Cast-P: a cast expression is value (resp. type) preexisting if its receiver is value (resp. type) preexisting.

3.3 Preexistence of object-invocation sites

The aim of preexistence is for object-invocation sites.

Site-P: An object-invocation site is said to be *monomorphic* if the dynamic type of its receiver is trivially known at compile-time—because the receiver is a New expression or its static type is *final*—or the invoked method itself is *final*. Otherwise, the site is said to be *polymorphic*, and it is *preexisting* iff its receiver is type-preexisting.

We don't mind whether a preexisting site is value- or type-preexisting. Indeed, in both cases, the optimization would be the same. Monomorphic sites are special cases of immutable preexistence and concrete types. We exclude them from preexistence analysis, because their optimistic implementation is always conservative, and this well-known optimization doesn't resort to the preexistence notion. Moreover, because of their high frequency, they would impede a fair comparison between original and extended preexistence.

A conservative recompilation policy would restrict to preexisting sites the choice of an optimistic implementation. Then, if the recompilation of a method is triggered during its activation, only its preexisting sites may be invalidated, and the current execution can safely continue. The required recompilation can wait the next activation of the method.

In contrast, a more aggressive optimization policy, by optimizing non-preexisting sites, will require on-stack replacement or code patching. For instance, a middle-course policy might be to (i) use the *optimistic* implementation for all preexisting sites; inlining static call sites can be considered; (ii) use the *conservative* implementation for non-preexisting read, write and cast sites, because patching cannot be used efficiently with them; (iii) use the *optimistic* implementation, without inlining, for all non-preexisting call sites; code patching can be efficiently used for repairing those sites.

[1]Accordingly, the set of dispatched method is then computed on the set of *loaded* subclasses, not only on the *instantiated* ones like with CHA.

[2]This rule assumes that the code of the method is available to analysis and has been already compiled. If it is not the case, e.g. for precompiled methods, the returned value is assumed to be non-preexisting.

The recompilation of an *optimized* site must be considered in two situations: (i) when the optimistic implementation is invalidated; (ii) when the site becomes non-preexisting, even if its implementation is not invalidated. This recompilation concerns, for instance, the sites proven preexisting with the Call-P rule, and the transition may be caused by adding a new dispatched method. Conversely, when a non-preexisting site becomes preexisting, its recompilation might be considered in order to optimize it, but this recompilation is not mandatory.

3.4 More accurate inter-procedural analysis

The Call-P rule is very restrictive, since it forces the arguments of the call-site to be preexisting, even when the value returned by the dispatched methods does not depend on their input parameters. Besides returning just a boolean value, the analysis might return, for each preexisting expression, a *dependence set* consisting of the subset of input parameter positions the expression depends on. The rules are modified as follows:

Parameter-P: an input parameter at position `p` has the dependence set `{p}`.

Variable-P, Return-P: the dependence set of a variable is the union of the dependence sets of the expressions it depends on;

Call-P: a method-invocation expression is value (resp. type) preexisting if, (i) its receiver is type-preexisting[3]; and (ii), for each dispatched method, the returned value is value (resp. type) preexisting, and all the arguments of the call site corresponding to the returned dependence set are value (resp. type) preexisting;

moreover, the dependence set of the expression is the union of the dependence sets of the receiver and of the call-site arguments whose corresponding input parameter belongs to the returned dependence set;

otherwise: the dependence set of an object-invocation expression is that of its receiver. All other dependence sets are empty.

The new Call-P rule can conclude to the preexistence of a call-site expression even when one of its arguments is not preexisting, because the corresponding parameter is not used in the returned values of the dispatched methods. The Call-P rule can be stated more formally as follows. Let $DSet_e(expr)$ denotes the dependence set of the *expr* expression, and $DSet_f(fun)$ denotes the dependence set of the return of the *fun* method. Let us assume that the call site $\texttt{arg}_0\texttt{.foo(arg}_1\texttt{,..,arg}_k\texttt{)}$ dispatches to the set *dispatch*. Then, the condition on dependence sets in the rule is:

$$\forall \texttt{X.foo} \in dispatch, \forall i \in DSet_f(\texttt{X.foo}), \texttt{arg}_i \text{ is preexisting}$$

and the resulting dependence set is

$$DSet_e(\texttt{arg}_0\texttt{.foo}(\texttt{arg}_1, .., \texttt{arg}_k)) = \bigcup_{\substack{\texttt{X.foo} \in dispatch \\ i \in DSet_f(\texttt{X.foo})}} DSet_e(\texttt{arg}_i)$$

[3]This is required for closing the set of dispatched methods.

Dependence sets must also be used for type-preexistence, because the combination of type-preexistence and value-preexistence gives type-preexistence, along with the dependence set of the latter. Let us consider the following function:

```
A foo (A x, B y) {
     if <some condition>
        return x;
     else return new A(y);
}

A a; B b;           //  supposed to be non-preexisting
foo(a,new B());     //  ==> non-preexisting
foo(new A(b),b);    //  ==> type-preexisting
```

Its return depends on `x` and `new A`, and `A` is supposed to be already instantiated. The former is value-preexisting, with `{1}` as dependence set, while the latter is only type-preexisting, with an empty dependence set. Therefore, the return of `foo`, is only type-preexisting, but its dependence set is `{1}`. An expression calling `foo` can be preexisting only if its first argument is preexisting, and even if its second argument is not preexisting.

3.5 Concrete types

The *concrete type* of an expression is the set of its possible dynamic types, when this set is known statically, and is guaranteed to remain immutable during the current activation of the including method. Therefore, a concrete type is both a condition and a set, and this set is valid only if the condition holds. Accordingly, *combining* several concrete types involves both the union of their sets and the conjunction of their conditions.

According to CHA, the concrete type of an expression would be the set of the instantiated subclasses of its static type. However, in the general case, there is no guarantee that a new subclass cannot be loaded before the expression is evaluated. Therefore, concrete types are interesting only when (i) they are stricter than the types deduced from CHA; (ii) there is a guarantee, akin to preexistence, that they will not change during the current method activation. We thus distinguish between *immutable* and *mutable concrete types*. When a rule yields an immutable concrete type—it can be computed statically, once for all—the ConcreteType-P rule applies without restriction. In contrast, when the concrete type is mutable, it must rely on some preexisting expression, generally the receiver of the object-invocation expression.

The concrete type of an expression is decided according to another set of rules:

FinalType-CT: when the static type[4] of an expression cannot be specialized (like with `final` in Java), then the immutable concrete type of the expression consists of this static type.

The other CT rules are considered only when the FinalType-CT rule doesn't apply.

New-CT: a New expression has the immutable concrete type of its instantiated class.

PrivateWrite-CT: the concrete type of an attribute is the combination of all the concrete types of the right sides of its compiled assignments, when each one has an immutable concrete type. Then, a read expression of this attribute has its immutable concrete type. However, extra conditions are required: (i) this concrete type must be computed during static compilation, and it must be translated into an annotation of the resulting bytecode; (ii) writing this attribute cannot occur outside the unit of code that has been statically compiled (e.g. it is declared `private`); (iii) for practical reasons, we consider only the New and FinalType CT-rules at the right side, but an actual static compiler could proceed to a more accurate analysis, by considering PrivateWrite and PrivateMethod CT-rules.

[4] This is the only need for static types in the preexistence analysis.

PrivateMethod-CT: this rule is the equivalent of the previous one for private methods, when their return variable has an immutable concrete type.

Variable-CT: the concrete type of a variable is the combination of the concrete types of all the expressions it depends on, and it is immutable if all the combined concrete types are immutable.

All the following rules yield a mutable concrete type.

SelfWrite-CT: in a variant of PrivateWrite-CT, attribute assignments are no longer reserved to the current unit of code, but instead reserved to the current receiver (as in Eiffel); in this case, the receiver of the read expression must be type-preexisting, too, since loading a subclass may augment or invalidate the concrete type.

Self-CT: the current receiver—hereafter called `self`, aka `this` or `Current` according to the different languages—has a very special concrete type because of overriding. In a method `foo` defined in class `A`, the concrete type of `self` is the subset of `A`'s subclasses (including `A`) that have been instantiated, and inherit `A.foo` without overriding it. If a subclass of `A`, say `B`, overrides `foo`, and `B.foo` calls `super`, then the concrete type of `self` in `B.foo` is added to that of `self` in `A.foo`. If it doesn't call `super`, then `B` and its subclasses don't belong to the concrete type of `self in A.foo`.

Return-CT: if the return type of the method is final, the FinalType-CT rule applies, even if the method has not been compiled; otherwise, if the method has been compiled, the Variable-CT rule applies;

Call-CT: the concrete type of a method-invocation expression is the combination of the concrete types of the return of all its dispatched methods, but the condition holds only if the receiver of the site is type-preexisting. Note that Call-CT is more general than PrivateMethod-CT, and it can be more accurate when both apply. However, the latter has the advantage of producing an immutable concrete type, independent of the receiver preexistence and the dispatched-method compilation.

Cast-CT: the concrete type of a CastSite is the subset of classes, in the receiver's concrete type, that are subclasses of the cast target-type. The receiver must be type-preexisting, since it is impossible to check this condition on an unloaded class. A variant of this rule could apply to boolean constructs similar to Java `instanceof`. Then, in an `if-instanceof-then-else` pattern, the concrete type would be split into two disjoint subsets, according to whether the test succeeds or fails.

For both read and call expressions, the concrete type is not attached globally to a given attribute or method, but to what we called a *site pattern*, i.e. a pair (c, p) of a class c and of the considered attribute or method p. Then, the concrete type of (c', p) is a subset of that of (c, p) when c' is a subclass of c.

About immutability. In the original preexistence analysis, preexistence and non-preexistence were immutable properties, which were computed only once. However, in the extended preexistence analysis, the preexistence of an entity is no longer immutable. Indeed, class loading can enlarge the call graph and add to a currently preexisting method-invocation expression a dispatched method whose return is not preexisting. In the opposite direction, a non-preexisting method-invocation expression may become preexisting when a still uncompiled dispatched method is compiled and has a preexisting return.

Similar immutability comes with concrete types. Therefore, in order to simplify the maintenance of both preexistence and concrete types, we decided to restrict the interprocedural analysis to the cases where the returned preexistence and concrete types are immutable.

This makes us reformulate the two rules about the return of a method, by adding the following restrictions:

Return-P, Return-CT: the preexistence and the concrete type of the return of a method is considered only when they are immutable.

3.6 Preexistence attribute analysis

The *immutable field* analysis of [5] is translated, here, into the ImmutableAttribute-P rule. So far, this particular case is the only reason for distinguishing value-preexistence from type-preexistence. Immutability can be deduced from specific languages features, with such keywords as `final`, `readonly` or `val`. Otherwise, as discussed by the authors, immutability could be deduced from the static analysis of a class, as the fact that the attribute satisfies the following conditions: (i) it is not assigned, apart from in a constructor; (ii) a constructor cannot be applied twice to the same object; (iii) assigning the attribute must be reserved to the considered class, for instance it is declared `private`. However, immutable fields don't fit well with our target language, Nit, because immutability cannot be expressed strictly and condition (ii) cannot be checked. Therefore, in our experiments, we don't consider the rule, and instead drop, in practice, the notion of value-preexistence.

The new PrivateWrite-CT rule is an interesting alternative which fits better to the language, since in Nit write accesses are private, by default. Furthermore, this rule provides more information than ImmutableAttribute-P, namely concrete types, and it would be of general interest for most languages (e.g. Java, C♯, Scala). In contrast, the SelfWrite-CT is a useful alternative for Eiffel. Both rules could be combined to deal with `protected` attributes, when this keyword has the same meaning as in C++, not as in Java.

4. EXPERIMENTS

4.1 Principle

Our testbed involves the Nit language and its prototype VM. This VM is currently based on pure interpretation, and our testbed simulates, at run-time, the behaviour of an actual JIT compiler.

We have implemented the proposed preexistence analysis in the Nit VM, and run a series of experiments in order to simulate lazy compilation, and assess the correctness and the benefits of the approach. We run this simulation in four different configurations (or steps):

1. as pure interpretation: executing an object-invocation site involves recomputing the optimistic implementation to use;
2. as pure code patching: each method is compiled lazily at its first call, and each site in the method memorizes its optimistic implementation; then, when a new class is loaded, the changes in the optimistic implementations are propagated to the sites impacted;
3. as pure preexistence: trampolines are simulated with a boolean flag in the methods; when a method is executed and this flag is set, it is first compiled, preexistence is computed, and each site is associated with its appropriate implementation, optimistic or conservative according to the protocol; when a site is impacted by a class loading, the trampoline flag is set on its including method.
4. as a mix of code patching, used for all method calls, and preexistence, used for attributes and casts.

The first two simulations are presumed to involve exactly the same implementations, computed, respectively, in backward (Step 1) and forward chaining (Step 2). This invariant is checked in the Step 2 simulation, and it provides some evidence (of course, not a proof) of correctness.

Efficiency measures. Without loss of generality, such protocols can be measured according to three main dimensions:

1. a static measure of the code at some time point during the execution, by counting object-invocation sites in the compiled code, according to their implementation and preexistence;
2. a dynamic measure of the code execution during some time interval, by counting the object-invocation sites that are executed, according to their implementation;
3. a dynamic measure of the protocol during the whole execution, by counting the transitions that maintain the system correctness: number of method recompilations; number of changes in the implementation or the preexistence of a site; number of patches required.

Various protocols (e.g. the variants described in Section 2.3, or variants defined by some subsets of rules) will be compared with respect to theses statistics. These experiments are currently in progress, and we present, here, our first results for a small family of protocols:

- Variant 1, complemented with Variant 2 for unconditional `new` sites;
- pure code-patching based on the optimistic implementation (Step 1-2), or pure preexistence (Step 3), with optimistic implementation for preexisting sites, and conservative implementation for non-preexisting sites;
- with two alternative rule sets for preexistence: (i) the original one; and (ii) the extended rule set proposed here, apart from the PrivateMethod, SelfWrite and Self CT-rules. The ImmutableAttribute-P rule is also excluded from both rule sets.

Table 1: Original preexistence

	method	attribute	cast	total
monomorph	5063	2711	0	7774
preexisting	5762	3537	360	9659
non preexisting	4422	1899	819	7140
total polymorph	10184	5436	1179	16799
preexistence rate	56%	65%	30%	57%

It is worth noting that the FinalType-CT rule has been interpreted in an ad hoc way. Indeed, there is no equivalent of the `final` keyword in Nit. Therefore, we decided to tag as final a few classes of the Nit kernel, like `NativeString` and `NativeArray`, which are intrinsically final. Moreover, as our benchmarks involve a parser which has been generated automatically, we decided that the leaves of the parser class-hierarchy could be considered as final, because they would have been generated as final in a language providing this feature. Overall, the preexistence extension relies on the rules PrivateWrite-CT, Call and Return, plus the fact that preexistence and concrete types can be combined in the Variable and Call rules, which is not the case for final types and new expressions in monomorphic sites.

Benchmarks. Our benchmarks are a series of large Nit programs, most of them being tools dedicated to Nit programs, such as interpreter, compiler or documentation system. Hence, these benchmarks resort to meta-programming. In the Nit VM, we run these metaprograms on a small Nit program, eg `fibonacci(5)`. It is worth noting that the benchmarks used were developped a long time before we use them in these experiments. In practice, we run these benchmarks once in each configuration, and we computed the complete set of statistics, during and at the end of each run. Overall, we compute: (i) Measure 1 at the end of each run; (ii) Measure 2 and 3 during each run.

In the rest of this section, we first present detailed statistics on a single benchmark, the Nit compiler. Then, we conclude with more synthetic statistics on several benchmarks.

4.2 Compile-time results (Measure 1)

4.2.1 Original preexistence

Table 1 presents the statistics of object-invocation sites according to whether they are monomorphic, preexisting or non-preexisting, after the original specification. Percentages are relative to the number of polymorphic sites. The analysis is performed at the end of the first run.

We can make several observations:

- the set of sites is split into monomorphic, preexisting and non-preexisting subsets of similar sizes;
- the preexistence rate for method invocation (56%) is in the upper range of the benchmarks presented in [5] (between 20 and 60%); however, both statistics are not directly comparable, since the original numbers are relative to the inlined sites, while our numbers are relative to polymorphic sites; actually, when restricted to polymorphic sites with static implementation, our preexistence rate is similar;
- one of our motivations is multiple inheritance, and we can see that preexistence will be very useful for optimizing attribute accesses, with a markedly higher preexistence rate, about 65%; this is, however, not surprising, since it is common practice to access attributes only on the current receiver;
- in contrast, the preexistence rate is quite lower (30%) for subtyping tests: indeed, in our benchmark, casts are mostly applied to the value returned by a method;

Table 2: Inlinability of polymorphic sites (original)

	method	attribute	cast	total	%
preexisting	4912	3446	160	8518	50
non preexisting	3714	1892	443	6049	35
total inlinable	8626	5338	603	14567	86
non-inlinable	1562	98	576	2236	13
total	10188	5436	1179	16803	100

Table 3: Original non-preexistence

	method	attribute	cast	total	%
Read	2268	1420	97	3785	53
New	75	3	0	78	1
Call	1933	407	709	3049	43
Cast	11	13	0	24	0
other	135	56	13	204	3
total	4422	1899	819	7140	100

Compared to the original tests, we miss the Immutable-Attribute-P rule, but its effect seemed to be quite lower than that of parameters. Finally, these statistics show that there is room for improvement.

Inlinability. An objective of this study is to examine how many sites are *inlinable* from the point of view of their implementation. Here, inlinability means that the implementation is theoretically optimal: call and cast sites have a static implementation, and attribute sites don't need PH. Table 2 presents synthetic statistics of inlinability with original preexistence. Firstly, 13% of the polymorphic sites are not inlinable, most of them because they involve truly polymorphic method-dispatch (15%, i.e. 1562/10188, for call sites). In contrast, only 2% (98/5436) of polymorphic attribute sites require PH. All of the remaining sites (86%) could be inlined, but only 50% are preexisting and would not require any patching if inlined. Since all monomorphic sites can be inlined without any recompilation need, the overall inlinability rate amounts to 66% for all sites, and even 74% for attribute accesses.

We expect extended analysis to decrease the non-inlinable line, because of concrete types, and increase the preexisting line, because of preexistence.

Non-preexistence. The next table analyzes the source of non-preexistence, by depicting the statistics of non-preexisting receivers according to the kind of expression they depend on. The row associated with a given kind, namely Read, New, Call and Cast expressions, presents the number of receivers that depend only on this kind, and the 'other' row collects all the remaining receivers, ie those depending on more than one kind. The percentage are relative to the total number of non-preexisting sites. Most of the non-preexisting receivers are Read or Call expressions, and the 5% rest involves a few polymorphic New and Cast expressions, along with combinations of several kinds.

Table 4: Extended preexistence

	method	attribute	cast	total	%
Read	719	170	0	889	23
New	73	3	0	76	97
Call	413	100	9	522	17
Cast	1	4	0	5	21
other	49	14	2	65	32
total improved	1255	291	11	1557	21

Table 5: Inlinability of polymorphic sites (extended)

	method	attribute	cast	total	%
preexisting	6106	3737	162	10005	59
non preexisting	2591	1603	441	4635	27
total inlinable	8697	5340	603	14640	87
non-inlinable	1491	96	576	2163	12
total	10188	5436	1179	16803	100

4.2.2 Extended preexistence

We present now the corresponding statistics with extended preexistence. Table 4 presents the subset of originally non-preexisting sites that become preexisting, in the same form as in Table 3. Percentages are relative to the corresponding lines in the previous table.

At the end of the execution, almost all the New expressions used in polymorphic sites are preexisting. About 23% of all polymorphic read expressions used as a receiver involve attributes with a concrete type which is not final (Private-Write-CT). The result is slightly lower for call expressions, since only 17% of their use as a receiver have a preexisting return.

Overall, the improvement is significant, and 22% of the originally non-preexisting sites become preexisting. Note, however, that two rules have not been taken into account, yet. The PrivateMethod and Self CT-rules will increase the receivers with a concrete type, thus reducing the number of dispatched methods, and both rules should increase the number of preexisting receivers involving a call expression.

Inlinability. Table 5 presents the same statistics as Table 2 with extended preexistence. As expected, the number of non-inlinable sites decreases, very slightly, because of the slight increase in concrete types (lines Read and New in Table 4). The main improvement lies in the number of preexisting inlinable sites increasing from 50% to 59%.

Preexistence of method return. In order to analyze the effects of the Return and CallSite rules, we computed statistics on the preexistence of individual method returns. These statistics are done, successively, for methods, site patterns—ie pairs of methods and receiver static types—, call expressions, and the subset of call-expressions used as receivers (i.e. lines Call in Table 4). Moreover, these items are counted according to whether they return a preexisting or a non-preexisting value, or nothing (including a primitive value).

With extended preexistence (Table 6), half of the methods returning an object have a preexisting return. In contrast, the preexistence rate of call expressions is far lower (30%), while this rate is even lower for call-expression receivers (16%). The difference can be explained as follows: (i) receivers with a final type make the site monomorphic, thus excluding them from the 'receiver' column, while the other columns include those with a final type: (ii) call sites often dispatch onto several methods, and all of them must have a preexisting return for the expression being preexisting; Moreover, all the dispatched methods must be compiled, and this is not always the case.

Table 6: Extended preexistence for method return

	method	pattern	expression	receiver
preexisting	298	281	1442	515
non preexisting	304	452	3248	2534
total	602	733	4690	3049
preexistence rate	49	38	30	16
without return	2437	2095	10561	0

4.3 Runtime statistics (Measures 2 & 3)

Table 7 depicts the number of times a site is executed, according to its implementation, and with 2 configuration parameters: (i) original or extended preexistence analysis; (ii) with pure code-patching or pure preexistence-based protocol. These statistics are computed at the end of the execution, from the execution count and the resulting implementation and preexistence of each site. Hence, this is a faithfull representation of the next run, as if the benchmark was run again, after the virtual machine has been warmed-up.

These statistics deserve a few remarks:

- our benchmarks are not exactly deterministic, likely because object addresses are hashed, and memory allocation can differ between two runs. See, for instance, a discussion of this phenomenon on the test reproducibility in [8]. As each configuration of the test is tested in a separate run, their total runtime numbers differ. However, it doesn't affect the static numbers (Measure 1), and the variability is very low (below 1% for Measure 2) and does not affect the conclusions.
- while monomorphic sites represent roughly 32% of all sites in the code, they represent 15% of the site executions; this might be partly explained by the fact that (i) half of the monomorphic call sites are constructor calls, and (ii) the benchmark creates a mostly static object model (an AST of the tested program);
- with pure code-patching, there is no significant difference between original and extended preexistence; this is expected since the extended analysis doesn't improve the optimistic implementation, but marginally;
- in contrast, with pure preexistence, the differences are significant, with an increase of 50% for static call sites; regarding attribute access, the number of PH is divided by 2, and it was already very low.

Overall, at runtime, with pure extended preexistence, half of the polymorphic call sites are potentially inlined; the overhead of multiple inheritance for method calls is thus significant but low (about 18%), and it is almost negligible for attributes (5%). Actually, these results for attributes are markedly better than expected after the static numbers in Table 5. We are just lucky that sites with SST are markedly more executed than sites with PH.

The mixed protocol is not presented in Table 7 as its statistics can be reconstructed by taking the method column from the code-patching table, and attribute and cast columns from the preexistence table. Actually, in the mixed protocol, the number of potentially inlined static calls is the

Table 7: Statistics on executions (in thousands)

	Original preexistence					**Extended preexistence**				
		method	attribute	cast	total		method	attribute	cast	total
pure code-patching	monomorph	33990	1835	0	35825	monomorph	33998	1835	0	35833
	static	81434	0	663	82097	static	81394	0	663	82057
	SST	45668	66398	4054	116121	SST	45489	66322	4056	115868
	PH	578	548	322	1450	PH	574	548	322	1445
	total	161672	68782	5040	235495	total	161456	68706	5042	235204
		method	attribute	cast	total		method	attribute	cast	total
pure preexistence	monomorph	33998	1835	0	35833	monomorph	33994	1835	0	35829
	static	41565	0	663	42228	static	62637	0	663	63301
	SST	50224	59423	3564	113212	SST	42757	63477	3625	109860
	PH	36129	7602	814	44546	PH	21996	3372	752	26121
	total	161917	68860	5042	235820	total	161386	68685	5041	235112

Table 8: Statistics on site recompilations

	method	attribute	cast	total
pic-patterns	42	27	0	69
site-patterns	358	136	19	513
code-patching	1560	358	56	1974
preexistence sites	1332	183	27	1542
mixed	793	182	27	1002

same as with pure preexistence, but a high number of non-inlined static calls are added, and the use of PH is reduced to about 5% of all method calls.

Recompilations (Measure 3). Table 8 depicts the number of times the optimistic implementation of a site is modified according to the protocol. Similar numbers are displayed for site-patterns—i.e. for equivalence classes of sites calling the same method on the same receiver static type (*rst*)—and for *pic-patterns*, i.e. for equivalence classes of site-patterns whose methods/attributes have been introduced by the same class, called the property introduction class (*pic*). A pic-pattern can have two optimistic implementations, SST or PH, and it switches from SST to PH, when the *pic* gets a new position in a subclass of the *rst*. A site-pattern has a static implementation when there is a single dispatched method in the loaded subclasses of the *rst*. Otherwise, it takes the implementation of its pic-pattern. Finally, a site takes the implementation of its site-pattern, unless its receiver has a concrete type.

There are thus four kinds of propagation after a class loading: (i) a *pic*-pattern in the superclasses of the loaded class may switch from SST to PH, and this is then propagated to its site-patterns unless they are static; (ii) a site-pattern can then switch from STT to PH; (iii) a site-pattern can switch from static to SST/PH because a dispatched method has been added; (iv) finally, a change in a site-pattern can propagate to its sites without concrete type. This propagation is very efficient, because it focuses on the actual changes, through a cascade of equivalence classes which share implementations and changes. With code patching, the number of sites represents the number of patches, i.e. the number of changes in the optimistic implementation of a site. With preexistence or in the mixed protocol, the number of sites represents the number of changes in the optimistic implementation of a preexisting site, i.e. the number of times the trampoline flag is set (even if it was already set).

Finally, Table 9 presents the number of compilations and recompilations of methods, together with their cost. The compilation cost of a method is equal to the number of object-invocation sites in the method, plus 1. Statistics on recompilation are presented in the two protocols, with pure preexistence or with the mixed protocol. Both rely on the lazy recompilation of whole methods, but in the latter changing the implementation of a method call does not trigger the recompilation of the including method.

Table 9: Statistics on method recompilations

	first	recompilations	
	compilation	preexistence	mixed
number	3039	498	159
cost	27616	16129	8174

The number of method recompilations is rather low with pure preexistence, about 16% of the number of compiled methods. However, the recompilation cost is 60% of the first-compilation cost. This is not surprising, because the total recompilation cost of a method is roughly quadratic in its number of sites, since each optimized site has a chance of triggerring the recompilation of the whole method. Moreover, in the benchmark, about 1/3 of the total recompilation cost comes from a single, huge method which could be reasonably excluded by the optimization protocol. Therefore, it is not surprising that the recompilation cost of the mixed protocol is markedly lower, since non-inlinable call sites doesn't trigger any recompilation with this protocol.

Nevertheless, the total whole-method recompilation cost (16129) is far higher than the number of sites that effectively require a patch (1974, Table 8). It is worth noting, too, that the latter is far greater than the number of whole-method recompilations (498), so laziness is effective.

4.4 More bencharks

Besides the Nit compiler (`nitc`), whose statistics have been presented in detail, we run the testbed on the Nit interpreter (`niti`), the Nit documentation system (`nitdoc`), a wiki developped in Nit (`nitiwiki`) and a tool for interfacing Java and Nit (`jwrapper`). The detailed statistics on these benchmarks are available in an electronic appendix (www.lirmm.fr/~ducour/Publis/RD-JP-JP-PPPJ16.pdf).

Table 10 presents the number of preexisting static call sites in all benchmarks, respectively with original and extended preexistence, along with the percentage of improvement and the total number of polymorphic call sites. Table 11 presents similar statistics for preexisting attribute sites with an SST implementation. On average, the improvement is about 24%

Table 10: Number of preexisting static call sites

benchmark	original	extended	improvement	total
nitc	4911	6101	24%	10184
niti	3682	4381	19%	7918
nitdoc	1723	2397	39%	4525
jwrapper	2061	2356	14%	2950
nitiwiki	267	391	46%	680
total	12644	15626	24%	26228

Table 11: Number of preexisting SST attribute sites

benchmark	original	extended	improvement	total
nitc	3439	3726	8%	5436
niti	2274	2428	7%	3494
nitdoc	2293	2456	7%	3463
jwrapper	912	958	5%	1083
nitiwiki	363	379	4%	466
total	9281	9947	7%	13942

for call sites, and 7% for attribute accesses. The original preexistence rate is actually higher for attributes (63% vs 48%), which provides less chance of improvement. The ratios to the differences, ie (extended − original)/(total − original), are much more similar, 22 vs 14%.

5. RELATED AND FUTURE WORKS

5.1 Related works

The statistics presented in [5] are not directly comparable to ours. Indeed, they measure the rate of inlined sites with a preexisting receiver, and we may understand "inlined sites" as sites with a static implementation. However, it is unclear whether monomorphic sites are excluded or not. Another point would be to compare the respective sizes of the tested benchmarks. In their Table 7, their Java benchmarks involve between 211 and 1548 "inlined virtual call sites", among which between 54 and 723 are preexisting. In contrast, our Nit benchmarks involve up to 4917 preexisting sites with a static implementation, all of them being candidates to inlining. There is thus some evidence that our benchmarks are not smaller than theirs.

There have been few works on type analysis at runtime. [16] shows that CHA is generally close to the optimal, hence providing a good basis for inlining. Our experiments don't contradict this conclusion: the concrete type analysis proposed, here, increases the number of static calls by less that 1%. From these two arguments, one might draw the conclusion that analyses more accurate than CHA would be useless. However, coupled with preexistence analysis, our concrete-type analysis increases the number of preexisting static calls by 24% on average (Table 10).

Besides these two technical points, the methodology used in this study seems to differ markedly from the methodology commonly used in the literature on virtual machines and adaptive compilers, e.g. [1]. Indeed, we choose to assess the efficiency of the proposed approach by counting items in the program, or events in its execution, instead of measuring time durations. Obviously, the chronometer is the final arbiter, but there are so many biases in the experiments, and differences are often so small, that it is difficult to draw robust conclusions and explanations from time measurements. In contrast, counting seems to be a better way to explain the observed differences and to explore the state space of the designed system. Of course, besides these methodological considerations, our simulation approach forces us to count, since time measurements would have been absolutely meaningless. By the way, we feel ourselves frustrated to be unable to compare our numbers, e.g. recompilation numbers or costs, to similar numbers in the literature, because we couldn't find such numbers.

5.2 Future works

In this paper, we consider only the optimizations related to the selection of the best implementation, thus generalizing *devirtualization*. A few other optimizations are commonly considered in similar contexts.

The combination of preexistence and inlining is worth an in-depth analysis. First of all, as already noted in [2, 16], preexistence is not preserved by inlining, since the formal parameter of a method is always preexisting in the callee, while the corresponding actual parameter is not necessarily preexisting in the caller. Preexistence of actual parameters passed to a method call has been used in order to predict the effects of inlining [18]. The extended preexistence analysis proposed here would improve the prediction. Besides, inlining resorts to combinatorial optimization. Indeed, in a 3-method static call-chain, say `foo` calling `bar`, and `bar` calling `baz`, it may happen that `baz` is no longer inlinable when `bar` is effectively inlined in `foo`. Moreover, once `bar` has been inlined in `foo`, the recompilation of `foo` may be triggerred by a site in `bar`. Therefore, inlining increases both the chance of recompiling a method and the cost of its recompilation. Our simulation-based methodology seems to be appropriate to study heuristics for optimizing the inlining decision.

`Self`-invocations are special because of the Self-CT rule. A well-known optimization involves *customization*, i.e. copying and specializing the code of all inherited methods [3]. Customization has contradictory effects wrt devirtualization: it makes `self` monomorphic but it increases the number of dispatched methods, at the risk of *revirtualizing* some sites. Therefore, like inlining, customization resorts to combinatorial optimization, and we will study simple heuristics to decide on its use.

Generics are not considered yet in the proposed analysis. When a generic instantiation site `new A<T>` is analyzed, the concrete type considers only `A` and disregards `T`. This will account for generics in a *homogeneous* implementation [14], as in Java. It will not work, however, in C♯ [12], because of its *heterogeneous* implementation which may yield different layouts for `A<int>` and `A<Person>`. Different optimizations of generics have been proposed in the literature [17, 20], and the preexistence and concrete type analysis could be extended in order to take generics into account. It would allow the analyzer to handle the concrete types of formal type parameters, and provide an efficient solution to the heterogeneous-implementation issue.

6. CONCLUSIONS

This paper has presented a preexistence analysis which extends previous work by distinguishing between type- and value-preexistence. The introduction of type-preexistence opens the door to a rather powerful concrete-type analysis, allowing us to take advantage of instantiation sites and final types, through a mix of simple static analysis (for read sites) and dynamic, inter-procedural control-flow analysis.

Our work has been motivated by multiple inheritance, for which a more accurate preexistence analysis was desired. Besides multiple inheritance, a secondary objective was to measure the efficiency and the cost of a recompilation protocol based on preexistence, without guards or patches. Such a protocol was mainly required for limiting the overhead caused by multiple inheritance, especially for attribute accesses. The proposed protocol provides a satisfying solution to this multiple-inheritance issue. For method calls, code-patching could be used, in complement, for non-preexisting sites and it should reduce significantly the recompilation cost, without significantly degrading the runtime efficiency. Overall, there is some evidence that full multiple inheritance could be used in such runtime systems, without yielding an overhead much higher than that of interfaces in Java, or traits in Scala.

In spite of the multiple inheritance motivation, the extended preexistence and concrete-type analysis could be used as well with languages like Java or C♯. The gain provided by the extended analysis is significant, although not revolutionary, and it might be even higher in languages like Java which provides a versatile `final` keyword.

Of course, the analysis is more intricate than simply testing whether a receiver is an input parameter of the method. However, most compilers and virtual machines proceed to analyses that are much more intricate. Furthermore, a large part of the analysis could be done at compile time, and its result could be translated into bytecode annotations, without impeding the runtime efficiency.

The proposed analysis has been tested in a prototype of VM for the Nit language. The language was mostly imposed by multiple inheritance. Indeed, languages with static typing and multiple inheritance are not so many. Once the language had been chosen, we found ourselves lucky that it was equipped with an interpreter which could serve to simulate just-in-time compilation. In counterpart, an obvious limitation of our experiments is that we used a family of closely connected benchmarks. This is however an inescapable limitation, as soon as the target language is a research language without a large application basis, apart from a family of tools mainly dedicated to code manipulation.

Anyway, besides this limitation, the experiment shows that the approach is effective on the tested benchmarks. We thus plan to continue this work by (i) computing more in-depth statistics for assessing the cost of the analysis; (ii) designing protocols for method inlining and code specialization, or for optimizing generics; (ii) developping a just-in-time compiler for the Nit VM.

7. REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Technical Report RC23429, IBM T.J. Watson Research Center, 2004.

[2] M. Arnold and B. Ryder. Thin guards: a simple and effective technique for reducing the penalty of dynamic class loading. In B. Magnusson, editor, *Proc. ECOOP'2002*, LNCS 2374, pages 498–524. Springer, 2002.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proc. OOPSLA'89*, SIGPLAN Not. 24(10), pages 146–160. ACM, 1989.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proc. ECOOP'95*, LNCS 952, pages 77–101. Springer, 1995.

[5] D. Detlefs and O. Agesen. Inlining of virtual methods. In R. Guerraoui, editor, *Proc. ECOOP'99*, LNCS 1628, pages 258–277. Springer, 1999.

[6] R. Ducournau and F. Morandat. Perfect class hashing and numbering for object-oriented implementation. *Softw. Pract. Exper.*, 41(6):661–694, 2011.

[7] R. Ducournau and F. Morandat. Towards a full multiple-inheritance virtual machine. *Journal of Object Technology*, 12:29, 2012.

[8] R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In G. T. Leavens, editor, *Proc. OOPSLA'09*, SIGPLAN Not. 44(10), pages 41–60. ACM, 2009.

[9] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proc. CGO'03*, pages 241–252. IEEE Computer Society, 2003.

[10] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. PLDI '92*, pages 32–43. ACM, 1992.

[11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proc. ACM OOPSLA '00*, pages 294–310, 2000.

[12] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proc. PLDI'01*, SIGPLAN Not. 36(5), pages 1–12. ACM, 2001.

[13] F. Morandat and R. Ducournau. Empirical assessment of C++-like implementations for multiple inheritance. In *Proc. ICOOOLPS Workshop*, pages 7–11. ACM, 2010.

[14] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*, pages 146–159. ACM, 1997.

[15] J. Privat. Nit language. http://nitlanguage.org/, 2008.

[16] F. Qian and L. Hendren. A study of type analysis for speculative method inlining in a jit environment. In *Proc. of CC'05*, pages 255–270. Springer-Verlag, 2005.

[17] O. Sallenave and R. Ducournau. Lightweight generics in embedded systems through static analysis. In *Proc. LCTES'12*, pages 11–20. ACM, 2012.

[18] A. Sewe, J. Jochem, and M. Mezini. Next in line, please! exploiting the indirect benefits of inlining by accurately predicting further inlining. In *SPLASH'11 Workshops*, pages 317–328, 2011.

[19] E. Steiner, A. Krall, and C. Thalinger. Adaptive inlining and on-stack replacement in the Cacao virtual machine. In *Proc. PPPJ '07*, pages 221–226. ACM, 2007.

[20] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the speed to code size tradeoff in parametric polymorphism translations. In *Proc. ACM OOPSLA'13*, 2013.