Efficient Fail-Fast Dynamic Subtype Checking

Rohan Padhye
rohanpadhye@cs.berkeley.edu
University of California, Berkeley
USA

Koushik Sen
ksen@cs.berkeley.edu
University of California, Berkeley
USA

Abstract

We address the problem of dynamically checking if an instance of class S is also an instance of class T. Researchers have designed various strategies to perform constant-time subtype tests. Yet, well-known production implementations degrade to linear search in the worst case, in order to achieve other goals such as constant space and/or efficient dynamic class loading. The fast path is usually optimized for subtype tests that succeed. However, in workloads where dynamic type tests are common, such as Scala's pattern matching and LLVM compiler passes, we observe that 74%–93% of dynamic subtype tests return a negative result. We thus propose a scheme for *fail-fast* dynamic subtype checking. In the compiled version of each class, we store a fixed-width bloom filter, which combines randomly generated type identifiers for all its transitive supertypes. At run-time, the bloom filters enable fast refutation of dynamic subtype tests with high probability. If such a refutation cannot be made, the scheme falls back to conventional techniques. This scheme works with multiple inheritance, separate compilation, and dynamic class loading. A prototype implementation of fail-fasts in the JVM provides 1.44×–2.74× speedup over HotSpot's native `instanceof`, on micro-benchmarks where worst-case behavior is likely.

CCS Concepts • **Software and its engineering → Polymorphism**; *Inheritance*; *Compilers*.

Keywords object-oriented programming, dynamic casts, multiple inheritance, bloom filters

ACM Reference Format:
Rohan Padhye and Koushik Sen. 2019. Efficient Fail-Fast Dynamic Subtype Checking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3358504.3361229

VMIL '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361229

1 Introduction

In object-oriented programming languages, a common runtime operation is to check whether an object o, statically known to be an instance of class S, is also an instance of class T. Such a test enables guarded type-safe conversion of a value of static type S to a value of static type T; that is, dynamic type casting. This test is performed by `instanceof` in Java and `dynamic_cast` in C++.

Dynamic subtype checking is a well-studied problem; researchers have designed a number of efficient implementation strategies over the last four decades. Although several strategies proposed in the literature guarantee worst-case constant-time subtype tests, such strategies either: (1) impose restrictions, such as supporting only single inheritance or assuming a closed class hierarchy, or (2) require per-class storage that may be linear in the size of the class hierarchy. In practice, several production implementations optimize for other objectives such as constant space, fast dynamic class loading, and/or minimizing the number of instructions for subtype tests. In these implementations, a linear-time scan may be necessary in the worst case. These implementations assume that most dynamic subtype tests succeed—usually because the object o is exactly of the queried type T.

In this paper, we first show that in some domains where dynamic subtype tests are heavily used, most of the dynamic subtype tests fail; that is, o is not an instance of T. In particular, we consider Scala's implementation of pattern matching as well as the LLVM compiler infrastructure; preliminary experiments show that 74%–93% of dynamic subtype tests fail. In both these implementations, failed tests require linear search when multiple inheritance is involved.

In response to these observations, we propose a novel scheme for *fail-fast* dynamic subtype checking. Our scheme stores only one extra machine word per class and requires only a single load + bit-mask test to refute dynamic subtype checks with high probability. If such a refutation cannot be made, the scheme falls back to conventional techniques. Our scheme is simply an add-on for existing implementations. It attempts to prevent worst-case linear search when it is likely to occur. For example, a prototype implementation of fail-fasts for the JVM provides up to 2.74× speedup over HotSpot's native `instanceof`, on micro-benchmarks that exercise Scala's pattern matching on traits. There is no runtime overhead when the fail-fast test is not performed. Our scheme works with multiple inheritance, separate compilation, and dynamic class loading.

Table 1. Summary of dynamic subtype checking strategies.

Scheme	Constant Space	Constant Time	Multiple Inheritance	Open Hierarchy
Schubert et al. [14]	✓	✓	✗	✗
Cohen's display [5]	✗	✓	✗	✓
NHE [10]	✗†	✓	✓	✗
Packed encoding [15]	✗†	✓	✓	✓‡
PQ-Encoding [17]	✗†	✓	✓	✗
R&B [13]	✗†	✓	✓	✓‡
Gibbs and Stroustrup [8]	✓	✓	✓	✗
Perfect Hashing [6]	✗	✓	✓	✓
HotSpot JVM [4]	✓/✗	✗	✓	✓
LLVM [1]	✓	✗	✓	✗

† The per-class space requirement is very small in practice.
‡ Requires non-trivial recomputation when dynamically loaded classes change the hierarchy.

2 Related Work

The most general approach for dynamically checking whether an object o is a an instance of T involves walking the inheritance tree of o's class to check if it extends T [16].

Of course, researchers have developed more efficient strategies. These schemes differ in their storage requirement, their run-time complexity, whether they make a closed-world assumption (i.e., the type hierarchy is known at compile-time), and whether they support multiple inheritance.

Schubert et al.'s [14] scheme, originally developed for natural-language taxonomies, assigns each class in a single-inheritance hierarchy an integer range: $\langle min, max \rangle$. This range has the property that the range of each node in the tree is a sub-range of all its ancestors' ranges. Thus, dynamic subtype checks are a simple range-inclusion test. This scheme makes a closed-world assumption.

Cohen's display [5] associates a table of size D with each class that is at depth D from the root of the class hierarchy. The table is populated with unique type identifiers for each transitive superclass in order. The test for whether an object o is an instance of class T is performed by querying the table of o's class at index $depth(T)$ to test if it matches the unique identifier of T. This scheme works with open class hierarchies, but does not support multiple inheritance.

Vitek et al. [15] propose several encodings of the class hierarchy to support space- and time-efficient subtype tests that support multiple inheritance. Palacz and Vitek [13] propose a *range-and-bucket* scheme for Java: single-inheritance classes are handled by Schubert-style *range* queries, while multiply-inheritable interfaces are mapped to *buckets*. An invariant is that no two interfaces that are reachable from each other in the hierarchy may map to the same bucket. Although these schemes support open hierarchies, dynamic class loading requires non-trivial recomputation at run time.

Gibbs and Stroustrup [8] propose mapping every class to a prime number. Each compiled class stores the product of the primes associated with all its transitive superclasses. Dynamic subtype tests then reduce to simple integer divisibility. This scheme requires constant space per class and supports multiple inheritance. However, in order to ensure uniqueness of primes within a class hierarchy, a closed world assumption must be made.

Ducournau [6] proposes *perfect hashing* to perform guaranteed constant-time dynamic subtype checks for open hierarchies with multiple inheritance.

Table 1 summarizes the trade-offs for these schemes as well as some others.

3 Real-World Case Studies

Although much research has focused on guaranteeing constant-time dynamic subtype tests, production implementations have chosen to make other trade-offs.

We look at case studies from two domains where dynamic subtype tests are commonly used:

1. Scala supports pattern-matching of objects, using the `match` keyword [7]. When the Scala compiler translates the `case` clauses in a `match` to JVM bytecode, a series of `instanceof` and `checkcast` instructions are emitted. It is thus likely that when running any Scala program that makes use of pattern matching, a large number of dynamic subtype tests will be performed.
2. A core API of the LLVM compiler infrastructure [11] is the `dyn_cast<T>` function [2]. This function performs a safe dynamic type cast of LLVM IR nodes (e.g. casting from `Instruction` to `CallInst`). Unlike C++'s standard `dynamic_cast` operator, which relies on v-tables, LLVM uses it's own Run-Time Type Information (RTTI) that supports dynamic casts between instances of non-virtual classes. Internally the cast uses a function called `isa<T>`, which is similar to Java's `instanceof`. Dynamic casts are heavily used by LLVM analysis and optimization passes.

3.1 Production Implementations

3.1.1 HotSpot JVM

The dynamic subtype checking implementation of the HotSpot JVM [4] uses a variant of Cohen's display that requires constant space. This works because all Java classes (except `Object`) have exactly one superclass; multiply inheritable interfaces are handled out-of-band. Constant space is achieved by simply bounding the depth of the class hierarchy that the display supports. This implementation is used both by the `instanceof` instruction as well as implicit checks required by `aastore`.

At run-time, each class C stores a display table of up to 8 transitive superclasses, in order, starting from `Object`. These classes form C's primary supertypes. If C is very deep in the hierarchy; that is, its distance from `Object` is greater than 8, then all of C's supertypes that have depth larger than 8 are considered secondary supertypes. Similarly, all the interfaces implemented by C, taken transitively, also belong to its secondary supertypes. Dynamic subtype checks against primary supertypes are very fast: they require a constant-time access into the Cohen-style display table, which can require as few as 3 instructions on some architectures. Subtype checks against secondary supertypes require a linear scan over an array of secondary supertypes. Each class also has a single-element cache for the last secondary supertype against which a dynamic subtype check succeeded.

It is clear that this implementation is optimized for both dynamic subtype tests against primary supertypes and *successful* dynamic subtype tests against secondary supertypes. A dynamic subtype test against secondary supertypes that fails necessarily requires a linear scan. The original paper [4] reports experiments with one- and two-element *negative* caches; these caches were eventually dropped since failing tests against secondary supertypes were not found to be common on SpecJVM98.

Although this scheme requires only constant space in theory—the worst-case linear scan can simply traverse the inheritance graph—the HotSpot JVM currently pre-computes all secondary supertypes of a class and stores them in a variable-sized array associated with the class.

3.1.2 LLVM

In LLVM, the expression `isa<T>(o)` evaluates to `true` if object o is an instance of class T. This C++ template function simply expands to a static method invocation: `T::classof(o)`. Such a static method is defined by every class in LLVM's hierarchy. To implement the `classof` method, LLVM uses the following convention for propagating RTTI [1] with constant space per-class.

Every class S that forms the root of a class hierarchy defines: (1) an `enum`, say `SKind`, containing unique integer identifiers for all concrete classes that derive from S, and (2) a method, say `getSKind()`, that returns the `SKind` belonging to an instance of S. Further, every instance of a concrete class T that derives from S stores in its object layout an identifier of type `SKind` that identifies class T. The static method `T::classof(S* o)` returns `true` if `o->getSKind()` identifies T or any subclass of T. For single-inheritance hierarchies, the numeric kinds can be assigned using a preorder traversal of the class hierarchy so that `classof` requires only an integer range-inclusion test.

The scheme gets complicated in the presence of multiple inheritance. Consider `T::classof(M* o)`, where neither M nor T are subclasses of each other. This query will only return `true` for objects belonging to classes that inherit from both T and M. LLVM's implementation compares `o->getMKind()` with every `MKind` (i.e, identifier for subclasses of M) that belongs to a class that is also a subclass of T. When `o` is not an instance of T, this amounts to a full linear search over the common descendants of T and M.

3.2 Most Dynamic Subtype Tests Fail

It is clear that `instanceof` and `isa<T>` have fast paths for the case where dynamic subtype tests succeed. We performed some small experiments to measure how many dynamic type tests actually succeed in practice.

First, we used ASM [12] to instrument `instanceof` bytecodes in JVM `.class` files. Our instrumentation allowed us to profile the results of `instanceof` instructions. We started by profiling the Scala compiler (version 2.12), which itself is written in Scala. When compiling a trivial `HelloWorld.scala` input, the `instanceof` instruction was executed 47,597 times, and it returned `false` in *93% of the cases*! We then considered a larger workload: building the Scala compiler itself using `sbt`. In this workload, 3.1 billion `instanceof` instructions were executed, of which 2.35 billion (76%) returned `false`. More than 45 million such tests were against interfaces.

Next, we performed a similar experiment with LLVM (version 8.0). We modified the implementation of the `isa<T>` function in `llvm/Support/Casting.h` to profile its return value. We then built the Clang compiler, which is based on LLVM, with this modification. When using Clang to compile a simple `HelloWorld.cpp` program, LLVM performed 5,537,150 dynamic subtype tests. Of these, 74% failed. We then considered a larger workload: a 10KLOC single-file program written in C. When using Clang to compile this file, 93.7 million `isa<T>` tests were performed, of which 73 million (78%) failed.

Further, we observe that LLVM's class hierarchy is quite large and involves complex multiple inheritance. For example, the class `CallInst`, which represents call instructions in the LLVM IR, is 10 levels deep from the furthest root in its hierarchy, and has 18 transitive superclasses [9].

In summary, we observed scenarios where: (1) dynamic subtype tests are commonly used, (2) multiple inheritance is supported, and (3) *most dynamic subtype tests fail.*

4 Fail-Fast using Bloom Filters

We propose a scheme that augments existing implementations of dynamic subtype testing, such as those used by the HotSpot JVM and LLVM, in order to avoid worst-case linear search when it is likely to occur.

At compile-time, we assign each type T a randomly generated fixed-size bit vector $\alpha(T)$. Typically, we want this bit vector to be the size of a single machine word, say m bits. For example, if our target architecture uses 64-bit words, then m=64. An important constraint is that $\alpha(T)$ must have a fixed parity k; that is, exactly k bits of $\alpha(T)$ are set to one. $\alpha(T)$ is thus randomly chosen from one of ${}^{m}C_{k}$ choices, with replacement. We require that k be much smaller than m. Section 4.1 explores how to choose k optimally. For m=64, a good candidate is k=3.

For each type T, we compute another m-bit vector called $\beta(T)$ at compile-time. $\beta(T)$ combines the values of $\alpha(S)$ for every type S which is a supertype of T using a bitwise OR operation (denoted by the symbol $\vee$). Formally, if we use the notation $T <: S$ to denote that T is a subtype of S, then:

$$\beta(T) = \bigvee_{T <: S} \alpha(S)$$

The following property always holds: if S is a supertype of T, then the k set bits of $\alpha(S)$ must also be set in $\beta(T)$. If we use $\wedge$ to denote the bitwise AND operation, then we have the invariant:

$$T <: S \Rightarrow \beta(T) \wedge \alpha(S) = \alpha(S)$$

This implication is unidirectional. It is possible for $\alpha(S)$ to be a subset of $\beta(T)$ even if S is not a supertype of T. This can happen if the k set bits of $\alpha(S)$ are coincidentally set across the α values for T and its supertypes. However, we can reduce the probability of such collisions by picking an appropriate value of k (§4.1).

At run-time, $\beta(T)$ is stored along with the metadata of T. When performing dynamic subtype tests that are likely to fail and require linear scans, we can prefix the dynamic type test with a fail-fast in the following way:

```
// is object `o` an instance of type `T`?
boolean fail_fast_instanceof(S o, type T) {
  if (type(o).beta & T.alpha != T.alpha) {
    return false;
  } else {
    return slow_instanceof(o, T); // linear scan
  }
}
```

If $\alpha(T)$ is not a subset of $\beta(T)$, then `o` is *surely not* an instance of `T`. $\beta(T)$ is thus a *bloom filter* [3]: it enables fast refutations with high probability. When such a refutation cannot be made, we fall back to the slow linear search.

Note that the fail-fast test *only* needs to be performed for cases where linear search is otherwise necessary; for example, when performing `instanceof` with secondary supertypes in the JVM, or `isa` tests with multiple inheritance in LLVM. For cases where existing schemes return results quickly, such as `instanceof` on primary supertypes in the JVM, there is no extra cost. Also note that it is not necessary to store $\alpha(T)$ in the run-time metadata of T if the target type will always be known at compile time. In the above pseudocode, `T.alpha` is a compile-time constant. In such cases, our proposed scheme requires storing only one extra machine word per type T. However, $\alpha(T)$ would need to be stored at run-time if we would like to support dynamic target types in subtype tests.

Fail-fasts have the same space and time overhead as a negative cache [4], but are not limited to a single target type.

Our scheme appears to have some similarities to Krall et al.'s near-optimal hierarchical encoding (NHE) [10]. NHE associates each type T with an encoding $\gamma(T)$. This encoding provides a much stronger invariant than ours:

$$T <: S \Leftrightarrow \gamma(T) \wedge \gamma(S) = \gamma(S)$$

Dynamic subtype tests can therefore be performed in guaranteed constant time. However, computing the NHE is an NP-hard problem. Further, NHE requires knowing the entire type hierarchy ahead-of-time and does not support incremental recomputation if the hierarchy changes. Although our scheme is much weaker than NHE, the encodings $\alpha(T)$ and $\beta(T)$ are extremely fast to compute, support separate and parallel compilation, and do not require recomputation in the presence of dynamic class loading.

4.1 Choosing the Right Parity

An important design decision for implementing fail-fasts is picking a value for k, given a fixed value for m. For the purpose of discussion, let's assume that m=64, since 64-bit systems are widely used at the time of writing. Our goal is to pick a value for k that reduces the probability of *false positives*; that is, cases where the fail-fast refutation cannot be made even though the dynamic subtype test should return a negative result. A false positive results in a full linear search.

At a first glance, it is clear that very small and very large values of k are undesirable simply because they reduce the space of ${}^{m}C_{k}$ choices from which to generate $\alpha(T)$. For example, both k=1 and k=63 are bad candidates when m=64, because they permit only 64 unique values of $\alpha(T)$. This might suggest considering k=32, which maximizes ${}^{64}C_{k}$ with over 1.8×10^{18} unique values. However, this turns out to be a terrible choice: if a type T has 10 transitive supertypes, then the probability of a false positives when k=32 is over 80%!

The general formula for the false positive rate p with a bloom filter of m bits that contains n elements of k bits each is approximately [3]:

$$p = (1 - e^{-\frac{k \times n}{m}})^{k}$$

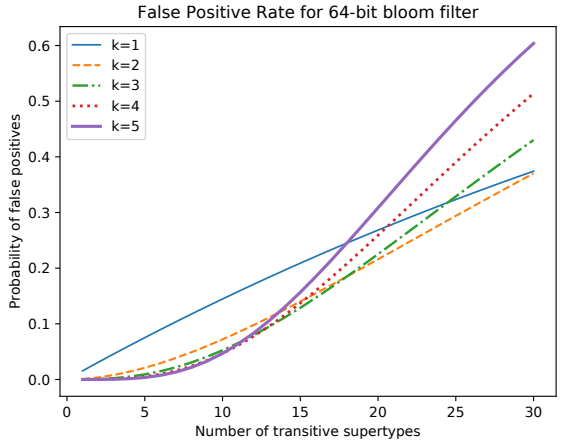

Figure 1. False positive rate when $m = 64$.

In our use of bloom filters for dynamic subtype checking, n is the number of transitive supertypes of T that together combine to form $\beta(T)$.

Figure 1 plots the false positive rate for some candidate values of k when $m = 64$, as a function of n. As the value of k becomes larger, the false positive rate quickly increases for large values of n. For example, when k=5, the false positive rate at n=30 is about 60%. On the other hand, k=1 performs poorly for small values of n; the false positive rate is 14% at n=10. Both k=2 and k=3 are good candidates: k=2 has lower false positive rate when $n > 18$, while k=3 has lower false positive rate when $n < 18$.

The choice of k would need to be made by an implementor based on the size of the machine word m and the typical size of type hierarchies encountered in real-world programs.

5 Preliminary Evaluation

We have implemented a prototype of our proposed scheme for the JVM. In this paper, our goal is to quickly evaluate if the proposed scheme speeds up `instanceof` checks with a secondary type as target, on micro-benchmarks.

Instead of modifying a full-blown VM like HotSpot, we instead modify classes—either manually or via bytecode instrumentation—to support fail-fasts at the application level. This is done only for the purpose of preliminary evaluation; a production implementation of fail-fasts will require VM integration. We then replace uses of `instanceof` in the application with a fail-fast followed by a fallback to the native `instanceof`.

Classes are modified as follows. Assume that we have a class `Foo` that extends class `Bar` and implements interfaces `Baz` and `Qux`. In this case, `Foo`'s supertypes include `Foo`, `Bar`, `Baz`, `Qux`, as well as transitive supertypes of `Bar`, `Baz`, and `Qux`. The class `Foo` is modified by adding the following highlighted lines in its definition:

```
class Foo extends Bar implements Baz, Qux {
  public static final long __alpha__
                = FailFast.genAlpha();

  public static final long __beta__
           = Foo.__alpha__ | Bar.__beta__ |
             Baz.__beta__  | Qux.__beta__;

  @Override public long __getBeta__() {
    return Foo.__beta__;
  }

  /* members of Foo */
}
```

A static field `__alpha__` stores the randomly generated value of α(`Foo`) with k=3. Another static field `__beta__` stores the statically computed value of β(`Foo`). This value is computed by performing a bitwise OR with `Foo`'s own `__alpha__` as well as the `__beta__` of its immediate supertypes—the latter together contain the set bits from the `__alpha__` fields of all transitive supertypes (e.g. supertypes of `Bar`). The fields `__alpha__` and `__beta__` are similarly defined for interfaces.

For the purposes of our preliminary evaluation, we also inject a new virtual method `__getBeta__()` into every class: this method returns the `__beta__` value of the corresponding class. This method is declared at the root of a sub-hierarchy of classes or interfaces defined in an application. We can only add fail-fasts for `instanceof` operations where the static type of the left-hand side operand is a class or interface that declares the `__getBeta__()` method. This virtual method is a proxy that simulates a real-world implementation of our scheme in a VM, where the `__beta__` value would be retrieved from a class's metadata or descriptor.

Program fragments such as `if(o instanceof T){...}` can now be replaced with the following fail-fast:

```
if (o.__getBeta__() & T.__alpha__ == T.__alpha__
   && o instanceof T) { ... }
```

If the fail-fast succeeds, then the left-hand-side of the short-circuiting '&&' will be `false`, thereby preventing linear search. Otherwise, we fall back to the original implementation of `instanceof`. Note again that this replacement is only done when `T` is a secondary type (ref. §3.1.1).

We evaluate our proposed scheme on the following micro-benchmarks, each of which perform dynamic subtype tests on a single object: (1) a single `instanceof` operation against an interface, which always returns `false`. (2) k-random cases: a series of k if-`instanceof`-then-`checkcast` branches, emitted by the Scala compiler when performing pattern matching on k distinct traits (which in turn are compiled into interfaces). Exactly one of the cases match, and this case is chosen uniformly at random in each trial of benchmarking. We consider k=2,5,10. (3) k-random cases with n-noise: we make all classes implement n dummy interfaces that are not used in pattern matching. The idea here is that such noise increases

Table 2. Preliminary Experimental Evaluation

	Baseline [4]		With Fail-Fast		
Benchmark	**Time**	**Worst Case**	**Time**	**Speedup**	**Worst Case**
Single Negative	51.458 ± 0.126 ns	100%	35.663 ± 0.092 ns	1.44×	0%
2-Random Case Match	46.248 ± 0.127 ns	33%	39.314 ± 0.082 ns	1.18×	0%
5-Random Case Match	75.090 ± 0.226 ns	67%	47.550 ± 0.130 ns	1.58×	0%
10-Random Case Match	116.031 ± 0.582 ns	82%	50.722 ± 0.228 ns	2.29×	0%
10-Random Cases + 10-Type Noise	143.057 ± 0.424 ns	82%	52.286 ± 0.205 ns	2.74×	9%

both the cost of linear search for the baseline as well as the false positive rate for the bloom filters used for fail-fasts. We consider only $k = 10$ and $n = 10$.

Table 2 lists the results of these experiments. For both the baseline as well as with our fail-fast instrumentation, we report (1) the time, as measured by OpenJDK's Java Microbenchmarking Harness (JMH), across 100 iterations of 500ms each (after 10 warmup iterations), and (2) the fraction of native `instanceof` operations that returned `false`—which in our case implies linear scan—measured using the same methodology as in §3.2. Note that any worst-case linear scans encountered by our fail-fast scheme are purely due to false positives (ref. §4.1). All experiments were run on a mid-2015 Macbook Pro running MacOS 10.13.6 and Java HotSpot Server VM version 12.0.2+10.

From Table 2, it can be seen that the fail-fast approach achieves significant speedup across all micro-benchmarks. Without noise, there are no false positives and all negative subtype tests can be returned in constant time. On the final benchmark which performs pattern matching on 10 cases, where the matched object also implements 10 dummy interfaces, the baseline requires performing linear scans about 82% of the time; even with false positives, our scheme performs linear scans only 9% of the time. The noise overhead due to false positives (1.56ns) is less than the noise overhead of the baseline (27.02ns), which is due to consistently longer linear scans. The overall speedup in this benchmark is 2.74×.

6 Conclusion

We presented a scheme for performing fail-fast dynamic subtype tests, which can be selectively applied when worst-case linear scan is likely. The proposed fail-fast can be implemented as an add-on to any existing subtype checking scheme, such as the HotSpot VM or the LLVM RTTI. On micro-benchmarks, the fail-fast helps provide more than 2× speedup over the HotSpot VM's native `instanceof`.

Acknowledgments

This work is supported in part by NSF grants CCF-1409872, CCF-1908870, CCF-1900968, and CNS-1817122.

References

[1] 2019. How to set up LLVM-style RTTI. https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html Accessed June 21, 2019.

[2] 2019. LLVM Programmer's Manual. https://llvm.org/docs/ProgrammersManual.html Accessed June 14, 2019.

[3] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[4] Cliff Click and John Rose. 2002. Fast Subtype Checking in the HotSpot JVM. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande (JGI '02)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/583810.583821

[5] Norman H Cohen. 1991. Type-extension type test can be performed in constant time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 626–629.

[6] Roland Ducournau. 2008. Perfect Hashing As an Almost Perfect Subtype Test. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 33 (Oct. 2008), 56 pages. https://doi.org/10.1145/1391956.1391960

[7] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, 273–298.

[8] Michael Gibbs and Bjarne Stroustrup. 2006. Fast dynamic casting. *Software: Practice and Experience* 36, 2 (2006), 139–156.

[9] LLVM Developers Group. 2019. llvm::CallInst Class Reference. https://llvm.org/doxygen/classllvm_1_1CallInst.html Accessed June 21, 2019.

[10] Andreas Krall, Jan Vitek, and R Nigel Horspool. 1997. Near optimal hierarchical encoding of types. In *European Conference on Object-Oriented Programming*. Springer, 128–145.

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[12] OW2 Consortium. 2018. ObjectWeb ASM. https://asm.ow2.io.

[13] Krzysztof Palacz and Jan Vitek. 2003. Java Subtype Tests in Real-Time. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 378–404.

[14] Lenhart K. Schubert, Mary Angela Papalaskaris, and Jay Taugher. 1983. Determining type, part, color and time relationships. *IEEE Computer* 16, 10 (1983), 53–60.

[15] Jan Vitek, R. Nigel Horspool, and Andreas Krall. 1997. Efficient Type Inclusion Tests. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, New York, NY, USA, 142–157. https://doi.org/10.1145/263698.263730

[16] Niklaus Wirth. 1988. Type extensions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 2 (1988), 204–214.

[17] Yoav Zibin and Joseph Yossi Gil. 2001. Efficient Subtyping Tests with PQ-encoding. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/504282.504290