Open and Efficient Type Switch for C++

Yuriy Solodkyy Gabriel Dos Reis Bjarne Stroustrup

Texas A&M University
Texas, USA
{yuriys,gdr,bs}@cse.tamu.edu

Abstract

Selecting operations based on the run-time type of an object is key to many object-oriented and functional programming techniques. We present a technique for implementing open and efficient type-switching for hierarchical extensible data types. The technique is general and copes well with C++ multiple inheritance.

To simplify experimentation and gain realistic preformance using production-quality compilers and tool chains, we implement our type swich constructs as an ISO C++11 library. Our library-only implementation provides concise notation and outperforms the visitor design pattern, commonly used for type-casing scenarios in object-oriented programs. For many uses, it equals or outperforms equivalent code in languages with built-in type-switching constructs, such as OCaml and Haskell. The type-switching code is easier to use and is more expressive than hand-coded visitors. The library is non-intrusive and circumvents most of extensibility restrictions typical of visitor design pattern. It was motivated by applications involving large, typed, abstract syntax trees.

Categories and Subject Descriptors D [*1*]: 5; D [*3*]: 3

General Terms Languages, Design

Keywords Type Switch, Typecase, Visitor Design Pattern, Memoization, C++

1. Introduction

Algebraic data types as seen in functional languages are closed and their variants are disjoint, which allows for efficient implementation of case analysis on such types. Data types in object-oriented languages are extensible and hierarchical (implying that variants are not necessarily disjoint). To be general, an type-switching construct must be *open* – allow for independent extensions, modular type-checking and dynamic linking. On the other, in order to be accepted for production code, the implementation of such a construct must equal or outperform all known workarounds. However, existing approaches to case analysis on hierarchical extensible data types are either efficient or open, but not both. Truly open approaches rely on expensive class-membership testing combined with decision trees []. Efficient approaches rely on sealing either the class hierarchy or the set of functions, which loses extensibility [9, 18, 44, 51]. Consider a simple expression language:

$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp/exp$$

In an object-oriented language without direct support for algebraic data types, the type representing an expression-tree in the language will typically be encoded as an abstract base class, listing the (sealed set of) allowed virtual functions, with derived classes representing variants:

```
struct Expr { virtual int eval() = 0; };
struct Value : Expr { ··· int eval(); int value; };
struct Plus  : Expr { ··· Expr& e1; Expr& e2; };
```

A simple evaluator for this language can be implemented with the aid of a virtual function eval() declared in the base class Expr. The approach is *intrusive* however, as we will have to modify the base class every time we would like to add a function. Instead we offer an external introspection of objects with case analysis:

```
int eval(const Expr& e)
{
  Match(e)
    Case(const Value& x) return x.value;
    Case(const  Plus& x) return eval(x.e1) + eval(x.e2);
    Case(const Minus& x) return eval(x.e1) - eval(x.e2);
    Case(const Times& x) return eval(x.e1) * eval(x.e2);
    Case(const Divide& x) return eval(x.e1) / eval(x.e2);
  EndMatch
}
```

The syntax is provided without any external tool support or additional definitions. Instead we rely on a few C++11 features [24], template meta-programming, and macros. It runs

about as fast as OCaml and Haskell equivalents (§4.3), and, depending on the usage scenario, compiler and underlying hardware, comes close or outperforms the handcrafted C++ code based on the *visitor design pattern* (§4).

The ideas and the library presented here were motivated by our unsatisfactory experiences working with various C++ front-ends and program analysis frameworks [1, 32, 39]. The problem was not in the frameworks per se, but in the fact that we had to use the *visitor design pattern* [18] to inspect, traverse, and elaborate abstract syntax trees to target languages. We found visitors unsuitable to express our application logic directly, surprisingly hard to teach students, and often slower than hand-crafted workaround techniques. We found users relying on dynamic casts in many places, often nested, to answer simple structural questions. That is, the users preferred shorter, cleaner, and more direct code to visitors. The consequential high performance cost was usually not discovered until later, when it was hard to remedy.

1.1 Summary

This paper makes the following contributions:

- A technique for implementing open and efficient type switching on extensible hierarchical data types as seen in object-oriented languages.
- The technique approaches the notational convenience of functional-language type-switch constructs and delivers equivalent performance to those for closed cases.
- The technique is simpler to use and outperforms the visitor pattern.
- A type-switch approach that combines subtype tests and type conversion; It delivers superior performance to approaches that combine (even constant-time) subtype tests with decision trees for even small class hierarchies.
- The type-switch construct handles multiple inheritance without workarounds.
- A constant-time function whose equivalence kernel partitions the set of objects with the same static type into equivalence classes based on inheritance path of the static type with the most-derived type.

In particular, our technique for efficient type switching:

- Comes close and often outperform various workaround techniques used in practice, e.g. visitor design pattern, without sacrificing extensibility (§4).
- Is open by construction (§3.2), non-intrusive, and avoids the control inversion typical for visitors.
- Works in the presence of multiple inheritance, both repeated and virtual, as well as in generic code (§3.7).
- Does not require any changes to the C++ object model or computations at link or load time.
- Can be used in object-oriented languages with object models similar to C++'s.
- For C++, can be implemented as a library written in ISO Standard C++11.

This is the first technique for efficiently handling type switching in the presence of general multiple inheritance. Being a library, our solution is that it can be used with any ISO C++11 compiler (e.g., Microsoft, GNU, or Clang) without requiring the installation of any additional tools or preprocessors. The solution sets a new threshold for acceptable performance, brevity, clarity and usefulness of open type-switching in C++.

2. Overview

A *heterarchy* is a partially ordered set $(H, <:)$ where H is a set of classes and $<:$ is reflexive, transitive and anti-symmetric *subtyping relation* on H. We are going to use terms *class* and *type* interchangibly as the exact distinction is not important for this discussion. Given two types in a subtyping relation $D <: B$, the type D is said to be a *subtype* or a *derived class* of B, which in turn is said to be a *supertype* or a *base class* of D. When the transitive reduction $<:_d$ of $<:$ is a function, H is usually referred to as *hierarchy* to indicate single inheritance.

Subtyping effectively makes objects belong to multiple types. We call by the *most-derived type* the type used to create an object (before any conversions). By *static type* we call the type of an object as known to the compiler based on its declaration as well as any supertype of it. By *dynamic type* of an object we call any base class of the most-derived type.

2.1 Type Switch

In general, *type switch* or *typecase* is a multiway branch statement that distinguishes values based on their type. In a multi-paradigm programming language like C++ that supports in various forms parametric, ad-hoc and subtyping polymorphisms, such a broad definition subsumes numerous different type casing constructs studied in the literature [21, 23, 45]. In this work we only look at type casing scenarios based on dynamic polymorphism of C++ (nominative subtyping polymorphism based on inheritance), similar to those studied by Glew [21]. It is possible to generalize type casing (the notion, not our implementation) to static polymorphism of C++ (ad-hoc and parametric polymorphisms enabled by overloading and templates) along the line of work introduced by Harper and Morrisett [23] and studied in the context of closed and extensible solutions by Vytiniotis et al [45], but we do not address such a generalization here. We use the term *type switch* instead of a broader *typecase* to stress the run-time nature of the type analysis similar to how regular **switch**-statement of C++ performs case analysis of values at run time.

Given an object descriptor, called *subject*, of static type S (pointer or reference) referred to as *subject type*, and a list of *target types* T_i associated with the branches, a type switch statement needs to identify a suitable clause m (or absence of such) based on the most-derived type D $<:$ S of

the subject as well as suitable conversion that csts the subject to the target type T_m. Due to multiple inheritance, types T_i in general may or may not be derived from S, however, because of the strong static type safety requirement, the type of applicable clause T_m will necessarily have to be one of subject's dynamic types: D <: T_m. A hypothetical type switch statement, not currently supported by C++, may look as following:

switch (subject) { **case** T_1: s_1; ... **case** T_n: s_n; }

There is no need for an explicit *default clause* in our setting because such a clause is semantically equivalent to a case clause guarded by the subject type: **case** S: s. The only semantic difference such a choice makes is in the treatment of null-pointers, which, one may argue, should be handled by the default clause. We disagree, because not distinguishing between invalid object and valid object of a known static but unknown dynamic type may lead to some nasty run-time errors.

Similar control structures exist in many programming languages, e.g. *match* in Scala [35], *case* in Haskell [25] and ML [33], *typecase* in Modula-3 [6] and CLOS [**?**] (as a macro), *tagcase* in CLU [30], *union case* in Algol 68 and date back to at least Simula's *Inspect* statement [11]. The statement in general can be given numerous plausible semantics:

- *First-fit* semantics will evaluate the first statement s_i such that T_i is a base class of D
- *Best-fit* semantics will evaluate the statement corresponding to the most-derived base class T_i of D if it is unique (subject to ambiguity)
- *Exact-fit* semantics will evaluate statement s_i if $T_i = D$.
- *All-fit* semantics will evaluate all statements s_i whose guard type T_i is a subtype of D (order of execution has to be defined)
- *Any-fit* semantics might choose non-deterministically one of the statements enabled by all-fit

The list is not exhaustive and depending on a language, any of these semantics can be a plausible choice. Functional languages, for example, often prefer first-fit semantics because it is similar to case analysis in mathematics. Object-oriented languages would typically be inclined to best-fit semantics due to its similarity to overload resolution and virtual dispatch, however, some do opt for first-fit semantics to mimic the functional style: e.g. Scala [35]. Exact-fit semantics can often be seen in languages supporting discriminated union types: e.g. variant records in Pascal, Ada and Modula-2, oneof and variant objects in CLU, unions in C and C++ etc. All-fit and any-fit semantics might be seen in languages based on predicate dispatching [17] or guarded commands [13], where a predicate can be seen as a characteristic function of a type, while logical implication can be seen as subtyping.

2.2 Expression Problem

Type switching is related to a more general problem manifesting the differences in functional and object-oriented programming styles.

Conventional algebraic datatypes, as found in most functional languages, allow for easy addition of new functions on existing data types. But they fall short in extending data types themselves (e.g. with new constructors), which requires modifying the source code. Object-oriented languages, on the other hand, make data type extension trivial through inheritance; but the addition of new functions operating on these classes typically requires changes to the class definition. This dilemma is known as the *expression problem* [10, 46].

Classes differ from algebraic data types in two important ways. Firstly, they are *extensible*, for new variants can be added later by inheriting from the base class. Secondly, they are *hierarchical* and thus typically *non-disjoint* since variants can be inherited from other variants and form a subtyping relation between themselves [21]. In contrast, variants in algebraic data types are *disjoint* and *closed*. Some functional languages e.g. ML2000 [2] and its predecessor, Moby, were experimenting with *hierarchical extensible sum types*, which are closer to object-oriented classes then algebraic data types are, but, interestingly, they provided neither traditional nor efficient facilities for performing case analysis on them.

Zenger and Odersky later refined the expression problem in the context of independently extensible solutions [50] as a challenge to find an implementation technique that satisfies the following requirements:

- *Extensibility in both dimensions*: It should be possible to add new data variants, while adapting the existing operations accordingly. It should also be possible to introduce new functions.
- *Strong static type safety*: It should be impossible to apply a function to a data variant, which it cannot handle.
- *No modification or duplication*: Existing code should neither be modified nor duplicated.
- *Separate compilation*: Neither datatype extensions nor addition of new functions should require re-typechecking the original datatype or existing functions. No safety checks should be deferred until link or runtime.
- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

While these requirements were formulated for extensible data type with disjoint variants, object-oriented languages primarily deal with hierarchical data types. We thus found it important to explicitly state an additional requirement based on the Liskov substitution principle [29]:

- *Substitutability*: Operations expressed on more general data variants should be applicable to more specific ones that are in a subtyping relation with them.

We will refer to a solution that satisfies all of the above requirements as *open*. Numerous solutions have been proposed to dealing with the expression problem in both functional [19, 31] and object-oriented camps [22, 27, 37, 49], but very few has made its way into one of the mainstream languages. We refer the reader to Zenger and Odersky's original manuscript for a discussion of the approaches [50]. Interestingly, most of the discussed object-oriented solutions were focusing on the visitor design pattern and its extensions, which even today seems to be the most commonly used approach to dealing with the expression problem in object-oriented languages.

A lot has been written about the visitor design pattern [18, 36, 37, 49]. Its advantages include *extensibility of functions*, *speed*, and, *being a library solution*. Nevertheless, the solution is *intrusive*, *specific to hierarchy*, and requires a lot of *boilerplate code* to be written. It also introduces *control inversion*, and, most importantly, – *hinders extensibility* of classes.

2.3 Open Type Switch

Note that the presence of a type switch in an object-oriented language alone does not solve the expression problem because the existing code may have to be modified to take new variants into account. Relying on default clause is not considered to be an acceptable solution in this context, because often times the only reasonable default behavior is to raise an exception. Zenger and Odersky note that in such cases defaults will transform type errors that should manifest statically into runtime exceptions that are thrown dynamically [50].

While we generally agree with this observation, we would like to point out that in our experience newly added variants were more often extending an existing variant than creating an entirely disjoint one. In a hypotetical compiler, for example, a new kind of type expression will typically extend a TypeExpression variant, while a new form of annotation will extend an Annotation variant, thus not extending the root ASTNode directly. Due to substitutability requirement such a new variant will be treated as a variant it extends in all the existing code. The functions that will be affected by its addition and thus have to be modified will be limited to functions directly analyzing the variant it extends and not providing a default behavior.

To account for this subtlety of extensible hierarchical data types, we use a term *open type switch* to refer to a type switch that satisfies all the requirements of an *open solution to expression problem* stated above except for the *no modification or duplication* requirement. We loosen it to allow modification of functions for which the newly added variant becomes a disjoint (orthogonal) case not handled by default clause. We believe that the loosened requirement allows us to express pragmatically interesting restrictions that developers are willing to live with. Besides, open type switch overcomes all the major shortcomings of the visitor design pattern:

- Case analysis with an open type switch is *non-intrusive* as it inspects the hierarchy externally and can be applied retroactively.
- New variants can be accounted for in the newly written code and will be seen as a base class or default in the existing code.
- The affected functions are limited to those for which the newly added variant is a disjoint case.
- The code avoids the control inversion and the need for boilerplate code that visitors introduce, and is thus a more direct expression of the intent.

2.4 C++ Specifics: Subobjects

C++ supports two kinds of inheritance: *non-virtual* [15] (also known as *replicated* [40] or *repeated* [47]) inheritance and *virtual* [15] (or *shared* [47]) inheritance. The difference between the two only arises in situations where a class indirectly inherits from the same base class via more than one path in the hierarchy. Different kinds of inheritance give raise to the notion of *subobject* in C++, which are then used to define semantics of operations like casts, virtual function dispatch etc. We give an informal introduction to them here in order to show some subtleties of the C++ inheritance model, which must be taken into account when addressing type switching or subtype testing.

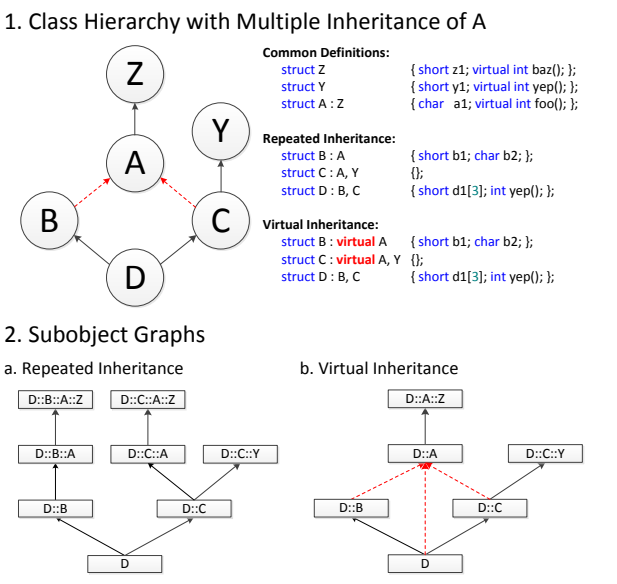

Figure 1. Multiple Inheritance in C++

Consider a simple class hierarchy in Figure 1(1). Class D indirectly inherits from class A through its B and C base classes. In this case, the user may opt to keep distinct subobjects of class A (repeated inheritance) or a shared one (virtual inheritance) by specifying how B and C are inherited from

A. The kind of inheritance is thus not a property of a given class, but a property of an inheritance relation between derived and base class and it is possible to mix the two in an object of the most-derived type.

A class hierarchy, i.e. an inheritance graph gives rise to a subobject graph, where a given class node may be replicated when inherited repeatedly or left shared when inherited virtually. The edges in such a graph represent subobject containtment and are marked with whether such containtment is shared or exclusive. Every class C in the class hierarchy will have its own subobject graph representing the subobject of an object of the most-derived type C. Figure 1(2) shows subobject graph for class D obtained for the class hierarchy in (1) under repeated (a) and virtual (b) inheritance of class A by classes B and C. The shared containment is indicated with the dashed arrows, while exclusive with the solid ones.

We will use term *object descriptor* to mean either pointer or reference to an object, which we will use interchangably when not explicitly specified. An object descriptor of static type A referencing an object of the most-derived type C can be understood as any $*{::}A$-node in the subobject graph of C. Rosie and Friedman call A an *effective type* of object, while the node in the subobject graph representing it – its *effective subobject*. Casts in such a model can be understood as a change from one effective subobject to another. We will use terms *source subobject* and *target subobject* to refer to effective subobjects before and after the cast. Their static types will be refered to as as *source type* and *target type* respectively. C++ distinguishes 3 kinds of casts: upcasts, downcasts and crosscasts.

An *upcast* is a cast from a derived class to one of its bases. When the base class is unambiguous, such casts are implicit and require no additional annotations. When the base class is ambiguous, cast failure is manifested statically in a form of a compile-time error. This is the case for example with casting D to A under repeated multiple inheritance of A, in which case the user needs to explicitly cast the object to B or C first in order to indicate the desired subobject and resolve ambiguity. In some cases, however, introduction of such an explicit cast is not possible: e.g. in implicit conversions generated by the compiler to implement covariant return types, cross casts or conversions in generic code. This does not mean that in such cases we violate the Liskov subtitution principle though – the classes are still in subtyping relation, but an implicit conversion is not available.

A *downcast* is a cast from a base class to one of its derived classes. The cast has to determine at run-time whether the source subobject is contained by a subobject of the target type in the most-derived type's subobject graph. Failure of such a cast is manifested dynamically at run-time.

A *crosscast* is a cast between classes that are not necessarily related by inheritance. Accordingly to the C++ semantics such cast is defined to be a composition of upcast to target type and downcast to the most-derived type. While the downcast to the most-derived type is always guaranteed to succeed regardless of the source subobject, the upcast to the target type may be ambiguous, in which case the cast will fail. A cast from Y to B inside an object of the most-derived type D in Figure 1(2a,2b) will be an example of a successful cross cast. A similar cast from Y to A inside D under repeated inheritance of (2a) will fail because of ambiguous upcast from D to A.

An interesting artefact of these distinctions can be seen on an example of casting a subobject of type Z to a subobject of type A in Figure 1(2a). The subobject D::B::A::Z will be successfully cast to D::B::A, while the D::C::A::Z will be successfully cast to D::C::A. These casts do not involve downcasting to D followed by an upcast to A, which would be ambiguous, but instead take the dynamic type of a larger subobject (D::B or D::C) the source subobject is contained in into account in order to resolve the ambiguity. A similar cast from Y to A will fail and should Y have also been non-virtually derived from Z, the cast from D::C::Y::Z to A would have failed. This shows that the distinction between crosscast and downcast is not based solely on the presence of a subtyping relation between the source and target types, but also on the actual position of the source subobject in the most-derived type's subobject graph.

C++ inheritance model, presented here informally, complicates the semantics and the implementation of a type switch further. On one side we have to define the semantics of a type switch when the cast between source and target types that are in subtyping relation is not possible. On the other – an implementation of the cast between source and target subobjects will have to take into account the location of the source subobject in the subobject graph into account in addition to the most-derived and target types on which a simple subtype test would solely depend.

2.5 Previous Work

The closed nature of algebraic data types allows for their efficient implementation. The traditional compilation scheme assigns unique (and often small and sequential) tags to every variant of the algebraic data type and type switching is then simply implemented with a multi-way branch [42] (usually a jump table) over all the tags [3]. Dealing with extensible hierarchical data types makes this extremely efficient approach infeasible:

- *Extensibility* implies that the compiler may not know the exact set of all the derived classes till link-time (due to *separate compilation*) or even run-time (due to *dynamic linking*).
- *Substitutability* implies that we should be able to match tags of derived classes against case labels representing tags of base classes.
- Presence of *multiple inheritance* might require pointer adjustments that are not known at compile time (e.g. due

to virtual base classes, ambiguous base classes or cross-casting).

There are two main approaches to implementing case analysis on extensible hierarchical data types, discussed in the literature.

The first approach is based on either explicit or implicit sealing of the class hierarchy, on which type switching can be performed. In Scala, for example, the user can forbid future extensions from a given class hierarchy through the use of a sealed keyword [16, §4.3.2]. The compiler then uses the above tag allocation over all variants to implement type analysis. In some cases the sealing may happen implicitly. For example, languages that allow names with internal and external linkage may employ the fact that classes with internal linkage will not be externally accessible and thus effectively sealed. While clearly efficient, the approach is not open as it avoids the question rather than solves.

The broader problem with this approach is that techniques that rely on unique or sequential compile or link-time constants violate independent extensibility since without a centralized authority there is no guarantee same constant will not be chosen in type unsafe manner by independent extensions. Updating such constants at load time may be too costly even when possible. More often than not however such updates may require code regeneration since decision trees, lookup tables etc. may have been generated by compiler for given values.

An important practical solution that follows this approach is the visitor design pattern [18]. The set of visit methods in visitor's interface essentially seals the class hierarchy. Extensions have been proposed in the literature [49], however they have problems of their own, discussed in §5.

The second approach employs type inclusion tests combined with decision trees [5] to avoid duplicate checks. The efficiency of the approach is then entirely focused on the efficiency of type inclusion tests [7, 9, 12, 14, 20, 26, 41, 44, 48, 51].

Type inclusion tests for single inheritance were initially implemented by traversing a linked list of types, as proposed by Wirth [48]. Such encoding requires little space, but runs in time proportional to the distance between the two types in the class hierarchy. A trivial constant-time type inclusion test can be achieved with a *binary matrix*, encoding the subtyping relation on the class hierarchy [12]. While efficient in time, it has quadratic space requirements, which makes it expensive for use on large class hierarchies. In response to Wirth' original publication, Cohen proposed the first space-efficient constant-time algorithm, which, howver, could only deal with single inheritance [9]. *Hierarchical encoding* is another constant-time test that maps subtype queries into subset queries on bit-vectors [7, 26]. The approach can handle multiple inheritance, but the space and time required for a subtype test in this encoding increases with the size of the class hierarchy, also Caseau's approach is limited to class hierarchies that are lattices. Schubert's *relative numbering* [41] encodes each type with an interval $[l, r]$, effectivelly making type inclusion tests isomorphic to a simple range checking. The encoding is optimal in space and time, however it is limited to single inheritance. *PQ-Encoding* of Zibin and Gil employs PQ-trees to improve further space and time efficiency of the constant-time inclusion testing [51]. While capable of handling type inclusion queries on heterarchies, the approach makes the closed world assumption and can be costly for use with dynamic linking because it is not incremental. The approach of Gibbs and Stroustrup [20] employs divisibility of numbers to obtain a constant-time type inclusion test. The approach can handle multiple inheritance and was the first constant-time technique to addresses the problem of casts between subobjects. Unfortunately the approach limits the size of the class hierarchies that can be encoded with this technique. Ducournau proposed constant-time inclusion test based on the fact that in an open solution a class has known amount of base classes and thus perfect hashes can be used to map them to this-pointer offsets typically used to implement subobject casts[14]. Unfortunately the approach addresses only virtual multiple inheritance and similarly to other approaches relies on load-time computations that may be costly. Detailed analysis and explanation of existing constant-time type inclusion tests can be found in [44] and [51].

With the exception of work by Gibbs and Stroustrup [20], all the approaches to efficient type-inclusion testing we found in the literature were based on the assumption that *the outcome of a subtyping test as well as the subsequent cast depend only on the target type and the most-derived type of the object*. While such assumption is sound for subtyping tests and subtype casts for shared inheritance (including single), it does not reflect the relationship between subobjects in the general case multiple inheritance present in C++.

2.6 The Source of Inefficiency

While constant-time type inclusion tests are invaluable in optimizing subtype tests in programing languages, their use in implementing a type switch is inferior to some workaround techniques. This may prevent wide adoption of a language implementation of such a feature due to its inferior performance. We implemented 3 constant-time type inclusion tests: binary matrix **??**, Cohen's algorithm [9] and fast dynamic cast [20] and combined them with a decision tree to implement a type switch on a class hierarchy ideally suited for such scenario. The class hierarchy used in this comparison was a perfect binary tree with classes number $2i$ and $2i + 1$ derived from a class number i. Our workaround techniques included visitor design pattern and a switch on the sealed sequential set of tags.

The chart in Figure 2 shows the time (Y-axis) each technique took to recognize an object of the most-derived type i (X-axis). It is easy to see that the logarithmic cost associated with the decision tree very quickly surpasses the constant

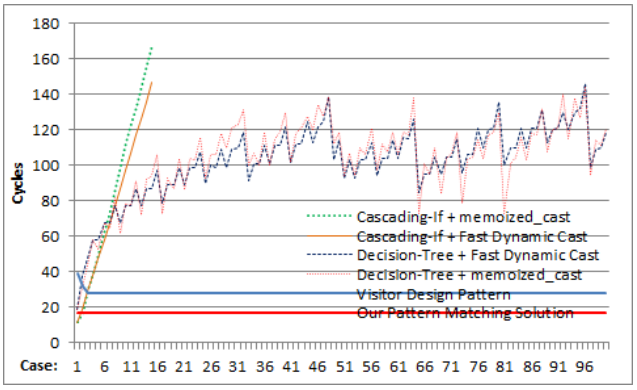

Figure 2. Type switch based on const-time type inclusion tests

overhead of double dispatch present in the visitor design pattern or the jump-table implementation of the switch on all tags. The edgy shape of timing results reflects the shape of the binary tree class hierarchy used for this experiment.

3. Type Switch

C++ does not have direct support of algebraic data types, but they can be encoded with classes in a number of ways. One common such encoding is to introduce an abstract base class representing an algebraic data type with several derived classes representing variants. The variants can then be discriminated with either run-time type information (referred to as *polymorphic encoding*) or a dedicated member of a base class (referred to as *tagged encoding*).

While our library supports both encodings, it handles them differently to let the user choose between openness and efficiency. The type switch for tagged encoding is simpler and more efficient for many typical use cases, however, making it open will eradicate its performance advantages. The difference in performance is the price we pay for keeping the solution open. We describe pros and cons of each approach in §4.2.

3.1 Attractive Non-Solution

While Wirth' linked list encoding was considered slow for subtype testing it can be adopted for very efficient type switching. The idea is to combine the fast switching on closed algebraic datatypes with a loop that tries the tags of base classes when switching on derived tags fails.

For simplicity of presentation we assume a pointer to array of tags be available directly from within object's taglist data member. The array is of variable size, its first element is always the tag corresponding to the most-derived type of the object, while its end is marked with a dedicated end_of_list marker distinct from all the tags. The tags in between are topologically sorted according to the subtyping relation with incomparable siblings listed in *local precedence order* – the order of the direct base classes used in the class definition. We call such a list a *Tag Precedence List* (TPL) as it resembles the *Class Precedence List* (CPL) of object-oriented descendants of Lisp (e.g. Dylan, Flavors, LOOPS, and CLOS) used there for *linearization* of class hierarchies. TPL is just an implementation detail and the only reason we distinguish TPL from CPL is that in C++ classes are often separated into interface and implementation classes and it might so happen that the same tag is associated by the user with an interface and several implementation classes. We also assume the tag-constant associated with a class D_i be accessible through a static constant D_i::class_tag. These simplifications are not essential and the library does not rely on any of these assumptions. Instead the user can retroactively narrate to the library the specific tag encoding used throught a trait-like classes.

A type switch below, built on top of a hierarchy of tagged classes, proceeds as a regular switch on the subject's tag. If the jump succeeds, we found an exact match; otherwise, we get into a default clause that obtains the next tag in the tag precedence list and jumps back to the beginning of the switch statement for a rematch:

```
    size_t attempt = 0;
    size_t tag = object→ taglist[attempt];
ReMatch:
    switch (tag) {
    default:
        tag = object→ taglist[++attempt];
        goto ReMatch;
    case end_of_list:
        break;
    case D1::class_tag:
        D1& match = static_cast⟨D1&⟩(*object);
        s1;break;
        ...
    case Dn::class_tag:
        Dn& match = static_cast⟨Dn&⟩(*object);
        sn;break;
    }
```

The above structure, which we call *Tag Switch*, lets us dispatch to case clauses of the most-derived class with an overhead of initializing two local variables, compared to efficient switch used on algebraic data types. Dispatching to a case clause of a base class will take time roughly proportional to the distance between the matched base class and the most-derived class in the inheritance graph, thus the technique is not constant. When none of the base class tags were matched, we will necessarily reach the end_of_list marker in the tag precedence list and exit the loop. As mentioned before, the default clause of the type switch can be implemented with a case clause on subject type's tag: **case** S::class_tag:

The efficiency of the above code crucially depends on the set of tags we match against be small and sequential to justify the use of jump table instead of decision tree to implement the switch. This is usually not a problem in closed hier-

archies based on tag encoding since the user of the hierarchy hand-picks himself the tags. The use of static cast to obtain proper reference once the most specialized derived class has been established, however, essentially limits the use of this mechanism to single inheritance only. This of course only refers to the way target classes inherit from the subject type – they can freely inherit other classes as long as they do not create repeated of virtual multiple inheritance of the subject type. Due to these assumptions, the technique is not open because it may violate independent extensibility. Moving away from these assumptions in order to make the technique more open (e.g. randomizing tags, using **dynamic_cast** etc.) will also eradicate its performance advantages.

3.2 Open but Inefficient Solution

Instead of starting with an efficient solution and trying to make it open, we start with an open solution and try to make it efficient. The following cascading-if statement implements the first-fit semantics for our type switch in a truly open fashion:

```
if (T1* match = dynamic_cast<T1*>(subject)) { s1;} else
if (T2* match = dynamic_cast<T2*>(subject)) { s2;} else
...
if (Tn* match = dynamic_cast<Tn*>(subject)) { sn;}
```

Its main drawback is performance: a typical implementation of **dynamic_cast** takes time proportional to the distance between base and derived classes in the inheritance tree. What is worse, is that the time to uncover the type in the i^{th} case clause is proportional to i, while failure to match will always take the longest. In a test involving a flat hierarchy of 100 variants it took 93 cycles to discover the first type and 22760 to discover the last (with linear combination of those times to discover the types in between). A visitor design pattern could uncover any type in about 55 cycles, regardless of its position among the case clauses, while a switch based on sequential tags could achieve the same in less than 20 cycles. The idea is thus to combine the openness of the above structure with the efficiency of a jump table on small sequential values.

3.3 Memoization Device

Let us look at a slightly more general problem than type switching. Consider a generalization of the switch statement that takes predicates on a subject as its clauses and executes the first statement s_i whose predicate is enabled:

```
switch (x) { case P1(x): s1; ... case Pn(x): sn; }
```

Assuming that predicates depend only on x and nothing else as well as that they do not involve any side effects, we can be sure that the next time we come to such a switch with the same value, the same predicate will be enabled first. Thus, we would like to avoid evaluating predicates and jump straight to the statement it guards. In a way we would like the switch to memoize which case is enabled for a given value of x.

The idea is to generate a simple cascading-if statement interleaved with jump targets and instructions that associate the original value with enabled target. The code before the statement looks up whether the association for a given value has already been established, and, if so, jumps directly to the target; otherwise the sequential execution of the cascading-if is started. To ensure that the actual code associated with the predicates remains unaware of this optimization, the code preceeding it after the target must re-establish any invariant guaranteed by sequential execution (§3.7).

The above code can easily be produced in a compiler setting, but producing it in a library setting is a challenge. Inspired by Duff's Device [43], we devised a construct that we call *Memoization Device* that does just that in standard C++:

```
typedef decltype(x) T;
static std::unordered_map<T,size_t> jump_targets;

switch (size_t& jump_to = jump_targets[x]) {
default: // entered when we have not seen x yet
    if (P1(x)) { jump_to = 1; case 1: s1;} else
    if (P2(x)) { jump_to = 2; case 2: s2;} else
      ...
    if (Pn(x)) { jump_to = n; case n: sn;} else
                 jump_to = n+1;
case n+1: // none of the predicates is true on x
}
```

The static jump_targets hash table will be allocated upon first entry to the function. The map is initially empty and according to its logic, request for a key x not yet in the map will allocate a new entry with its associated data default initialized (to 0 for size_t). Since there is no case label 0 in the switch, the default case will be taken, which, in turn, will initiate sequential execution of the interleaved cascading-if statement. Assignments to jump_to effectively establish association between value x and corresponding predicate, since jump_to is just a reference to jump_targets[x]. The last assignment records absence of enabled predicates for the value.

To change the first-fit semantics of the above construct into *sequential all-fit*, we remove the **else**s and rely on fall-through behavior of the switch. We also make the assignments conditional to make sure only the first one gets recorded:

```
if (Pi(x)) { if (jump_to ==0) jump_to = i; case i: si;}
```

Note that the protocol that has to be maintained by this structure does not depend on the actual values of case labels. We only require them to be different and include a predefined default value. The default clause can be replaced with a case clause for the predefined value, however keeping the default clause results in a faster code. A more important perfor-

mance consideration is to keep the values close to each other. Not following this rule might result in a compiler choosing a decision tree over a jump table implementation of the switch, which in our experience significantly degrades the performance.

The first-fit semantics is not an inherent property of the memoization device. Assuming that the conditions are either mutually exclusive or imply one another, we can build a decision-tree-based memoization device that will effectively have *most-specific* semantics – an analog of best-fit semantics in predicate dispatching [17].

Imagine that the predicates with the numbers $2i$ and $2i+1$ are mutually exclusive and each imply the value of the predicate with number i i.e. $\forall x \in \mathsf{Domain}(P)$

$$P_{2i+1}(x) \to P_i(x) \wedge P_{2i}(x) \to P_i(x) \wedge \neg(P_{2i+1}(x) \wedge P_{2i}(x))$$

An example of predicates that satisfy this condition are class membership tests where the truth of testing membership in a derived class implies the truth of testing membership in its base class.

The following decision-tree based memoization device will execute the statement s_i associated with the *most-specific* predicate P_i (i.e. the predicate that implies all other predicates true on x) that evaluates to true or will skip the entire statement if none of the predicates is true on x.

```
switch (size_t& jump_to = jump_targets[x]) {
default:
    if (P1(x)) {
        if (P2(x)) {
            if (P4(x)) { jump_to = 4; case 4: s4;} else
            if (P5(x)) { jump_to = 5; case 5: s5;}
            jump_to = 2; case 2: s2;
        } else
        if (P3(x)) {
            if (P6(x)) { jump_to = 6; case 6: s6;} else
            if (P7(x)) { jump_to = 7; case 7: s7;}
            jump_to = 3; case 3: s3;
        }
        jump_to = 1; case 1: s1;
    } else { jump_to = 0; case 0: ; }
}
```

Our library solution prefers the simpler cascading-if approach only because the necessary structure of the code can be laid out directly with macros. A compiler solution will use the decision-tree approach whenever possible to lower the cost of the first match from linear in case's number to logarithmic as seen in Figure**??**.

The main advantage of the memoization device is that it can be built around almost any code, providing that we can re-establish the invariants, guaranteed by sequential execution. Its main disadvantage is the size of the hash table that grows proportionally to the number of different values seen. Fortunately, the values can often be grouped into equivalence classes that do not change the outcome of the predicate. The map can then associate the equivalence class of a value with a target instead of associating the value with it.

In application to type switching, the idea is to use the memoization device to learn the outcomes of type inclusion tests (with **dynamic_cast** used as a predicate), thus avoiding calls to it on subsequent runs. It is easy to see that objects can be grouped into equivalence classes based on their most-derived type without affecting the results of predicates – the outcome of each type inclusion test will be the same on all the objects from the same equivalence class. We can use the address of class' **type_info** object obtained in constant time with **typeid**() operator as a unique identifier of each most-derived type. Presence of multiple **type_info** objects for the same class, as is often the case when dynamic linking is involved, is not a problem as we would effectively split a single equivalence class into multiple ones. This in fact would have been a solution if we were only interested in class membership. More often than not, however, we will be interesting in obtaining a reference to the target type of the subject and we saw in §**??** that proper this-pointer adjustments depend not only on the most-derived type, but also on target type and most importantly – path to the subject's static type from the most-derived type in the inheritance graph. Ideally we would like to have different equivalence classes per different paths from object's most-derived type to its static types, but there seem to be no easy way of identifying them given just an object descriptor.

3.4 Virtual Table Pointers

A class that declares or inherits a virtual function is called a *polymorphic class*. The C++ standard [24] does not prescribe any specific implementation technique for virtual function dispatch. However, in practice, all C++ compilers use a strategy based on so-called virtual function tables (or vtables for short) for efficient disptach. The vtable is part of the reification of a polymorphic class type. C++ compilers embed a pointer to a vtable (vtbl-pointer for short) in every object of polymorphic class type. CFront, the first C++ compiler, puts the vtbl-pointer at the end of an object. The so-called "common vendor C++ ABI"[8], further referred to as *C++ ABI* when not indicated otherwise, requires the vtbl-pointer to be at offset 0 of an object. The following compilers comply with the C++ ABI: GCC (3.x and up); Clang and llvm-g++; Linux versions of Intel and HP compilers, and compilers from ARM. We do not have access to the unpublished Microsoft ABI, but we have experimental evidence that Microsoft's C++ compiler also puts the vtbl-pointer at the start of an object.

While the exact offset of the vtbl-pointer within the object is not important for our discussion, it is important to realize that every object of a static type S* or S& pointed to or referenced by a polymorphic class S will have a vtbl-pointer at a predefined offset. Such offset may be different for different static types S, in which case the compiler will

know at which offset in type S the vtbl-pointer is located. For a library implementation we assume presence of a function **template** ⟨**typename** S⟩intptr_t vtbl(**const** S* s); that returns the address of the virtual table corresponding to the subobject referenced to by s. Such a function can be trivially implemented for the common C++ ABI, where the vtbl-pointer is always at offset 0:

```
template ⟨typename S⟩ std::intptr_t vtbl(const S* s) {
    static_assert(std::is_polymorphic⟨S⟩::value, "error");
    return *reinterpret_cast⟨const std::intptr_t*⟩(s);
}
```

Consider a repeated multiple inheritance hierarchy from Figure **??**(1). Each of the vtbl fields shown in Figure **??**(2) will hold a vtbl-pointer referencing a group of virtual methods known in object's static type. Figure 3(1) shows a typical layout of virtual function tables together with objects it points to for classes B and D.

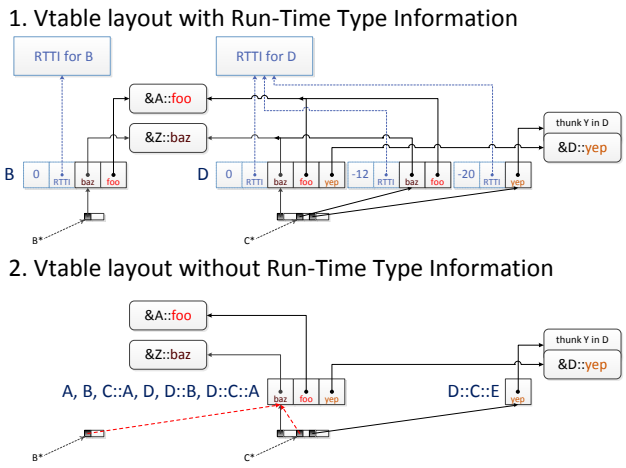

Figure 3. VTable layout with and without RTTI

Entries in the vtable to the right of the address pointed to by a vtbl-pointer represent pointers to functions, while entries to the left of it represent various additional fields like: pointer to class' type information, offset to top, offsets to virtual base classes etc. In many implementations, this-pointer adjustments required to properly dispatch the call were stored in the vtable along with function pointers. Today most of the implementations prefer to use *thunks* or *trampolines* – additional entry points to a function, that adjust this-pointer before transferring the control to the function, – which was shown to be more efficient []. Thunks in general may only be needed when its virtual function gets overriden. In such case the overriden function may be called via pointer to base class or a pointer to derived class, which may not be at the same offset in the actual object.

The intuition behind our proposal is to use the values of vtbl-pointers stored inside the object to uniquely identify the subobject in it. There are several problems with the approach however. First of all the same vtbl-pointer is usually shared by multiple types, for example, the first vtbl-pointer in Figure **??**(2) will be shared by objects of static type Z*, A*, B* and D*. This is not a problem for our purpose, because the subobjects of these types will be at the same offset in the most-derived object. Secondly, and more importantly, however, there are legitimate optimizations that let the compiler share the same vtable among multiple subobjects of often unrelated types.

Generation of the *Run-Time Type Information* (or RTTI for short) can typically be disabled with a compiler switch and the Figure 3(2) shows the same vtable layouts once the RTTI has been disabled. Since neither baz nor foo were overriden, the prefix of the vtable for the C subobject in D is exactly the same as the vtable for its B subobject, the A subobject of C or the entire vtable of A and B classes. Such layout, for example, is produced by Microsoft Visual C++ 11 when the command-line option /GR– is specified. Visual C++ compiler has been known to unify code identical on binary level, which in some cases may result in sharing of the same vtable between unrelated classes (e.g. when virtual functions are empty).

We now would like to show more formally that in the presence of RTTI, a C++ ABI compliant implementation will always have all the vtbl-pointers different. To do so, we need look closer at the notion of subobject, which has been formalized before [38, 40, 47]. We follow here the presentation of Ramamanandro et al [38].

3.5 Subobjects

In a given program P, a class B is a *direct repeated base class* of D if B is mentioned in the list of base classes of D without the **virtual** keyword ($D \prec_R B$). Similarly, a class B is a *direct shared base class* of D if B is mentioned in the list of base classes of D with the **virtual** keyword ($D \prec_S B$). A reflexive transitive closure of these relationships $\preceq^* = (\prec_R \cup \prec_S)^*$ defines the *subtyping* relation on types of program P. A base class *subobject* of a given *complete object* is represented by a pair $\sigma = (h, l)$ with $h \in \{\mathsf{Repeated}, \mathsf{Shared}\}$ representing the kind of inheritance (single inheritance is Repeated with one base class) and l representing the path in a non-virtual inheritance graph. A predicate $C \prec \sigma \succ A$ states that σ designates a subobject of static type A within the most-derived object of type C. More formally:

$$C \prec (\mathsf{Repeated}, C :: \epsilon) \succ C$$

$$\frac{C \prec_R B \qquad B \prec (\mathsf{Repeated}, l) \succ A}{C \prec (\mathsf{Repeated}, C :: l) \succ A}$$

$$\frac{C \prec_S B \qquad B \prec (h, l) \succ A}{C \prec (\mathsf{Shared}, l) \succ A}$$

ϵ indicates an empty path, but we will generally omit it in writing when understood from the context. In case of repeated inheritance in Figure 1(1), an object of the most-derived class D will have the following Repeated subobjects: D::C::Y, D::B::A::Z, D::C::A::Z, D::B::A, D::C::A, D::B, D::C, D. Similarly, in case of virtual inheritance in the same expample, an object of the most-derived class D will have the following Repeated subobjects: D::C::Y, D::B, D::C, D as well as the following Shared subobjects: D::A::Z, D::Z, D::A.

It is easy to show by structural induction on the above definition, that $C \prec \sigma \succ A \implies \sigma = (h, C :: l_1) \wedge \sigma = (h, l_2 :: A :: \epsilon)$, which simply means that any path to a subobject of static type A within the most-derived object of type C starts with C and ends with A. This objservation shows that $\sigma_\perp = (\mathsf{Shared}, \epsilon)$ does not represent a valid subobject. If Σ_P is the domain of all subobjects in the program P extended with $\sigma_\perp$, then a *cast* operation can be understood as a function $\delta : \Sigma_P \to \Sigma_P$. We use $\sigma_\perp$ to indicate an impossibility of a cast. The fact that δ is defined on subobjects as opposed to actual run-time values reflects the non-coercive nature of the operation – i.e. the underlain value remains the same. Any implementation of such a function must thus satisfy the following condition:

$$\delta(\sigma_1) = \sigma_2 \wedge C \prec \sigma_1 \succ A \implies C \prec \sigma_2 \succ B$$

i.e. the most-derived type of the value does not change during casting, only the way we reference it does. We refer to A as the *source type* and σ_1 as the *source subobject* of the cast, while to B as the *target type* and to σ_2 as the *target subobject* of it. The type C is the most-derived type of the value being casted. The C++ semantics states more requirement to the implementation of δ: e.g. $\delta(\sigma_\perp) = \sigma_\perp$ etc. but their precise modeling is out of the scope of this discussion. We would only like to point out here that since the result of the cast does not depend on the actual value and only on the source subobject and the target type, we can memoize the outcome of a cast on one instance in order to apply its results to another.

3.6 Uniqueness of vtbl-pointers under the C++ ABI

Given a reference a to polymorphic type A that points to a subobject σ of the most-derived type C (i.e. $C \prec \sigma \succ A$ is true), we will use the traditional field-access notion a.vtbl to refer to the virtual table of that subobject. The exact structure of the virtual table as mandated by the common vendor C++ ABI is immaterial for this discussion, but we mention a few fields that are important for the reasoning [8, §2.5.2]:

- rtti(a.vtbl): the *typeinfo pointer* points to the typeinfo object used for RTTI. It is always present and is shown as the first field to the left of any vtbl-pointer in Figure 3(1).
- off2top(a.vtbl): the *offset to top* holds the displacement to the top of the object from the location within the object of the vtbl-pointer that addresses this virtual table. It is always present and is shown as the second field to the left of any vtbl-pointer in Figure 3(1). The numeric value shown indicates the actual offset based on the object layout from Figure **??**(2).
- vbase(a.vtbl): *Virtual Base (vbase) offsets* are used to access the virtual bases of an object. Such an entry is required for each virtual base class. None are shown in our example in Figure 3(1) since it discussed repeated inheritance, but they will occupy further entries to the left of the vtbl-pointer, when present.

We also use the notation $\mathit{offset}(\sigma)$ to refer to the offset of the given subobject σ within C, known by the compiler.

Theorem 1. *In an object layout that adheres to the common vendor C++ ABI with enabled RTTI, equality of vtbl-pointers of two objects of the same static type implies that they both belong to subobjects with the same inheritance path in the same most-derived class.*
$\forall a_1, a_2 : A \mid a_1 \in C_1 \prec \sigma_1 \succ A \wedge a_2 \in C_2 \prec \sigma_2 \succ A$
$a_1.\mathit{vtbl} = a_2.\mathit{vtbl} \Rightarrow C_1 = C_2 \wedge \sigma_1 = \sigma_2$

Proof. Let us assume first $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$ but $C_1 \neq C_2$. In this case we have $\mathsf{rtti}(a_1.\mathit{vtbl}) = \mathsf{rtti}(a_2.\mathit{vtbl})$. By definition $\mathsf{rtti}(a_1.\mathit{vtbl}) = C_1$ while $\mathsf{rtti}(a_2.\mathit{vtbl}) = C_2$, which contradicts that $C_1 \neq C_2$. Thus $C_1 = C_2 = C$.

Let us assume now that $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$ but $\sigma_1 \neq \sigma_2$. Let $\sigma_1 = (h_1, l_1), \sigma_2 = (h_2, l_2)$

If $h_1 \neq h_2$ then one of them refers to a virtual base while the other to a repeated one. Assuming h_1 refers to a virtual base, $\mathsf{vbase}(a_1.\mathit{vtbl})$ has to be defined inside the vtable according to the ABI, while $\mathsf{vbase}(a_2.\mathit{vtbl})$ – should not. This would contradict again that both vtbl refer to the same virtual table.

We thus have $h_1 = h_2 = h$. If $h = \mathsf{Shared}$ then there is only one path to such A in C, which would contradict $\sigma_1 \neq \sigma_2$. If $h = \mathsf{Repeated}$ then we must have that $l_1 \neq l_2$. In this case let k be the first position in which they differ: $l_1^j = l_2^j \forall j < k \wedge l_1^k \neq l_2^k$. Since our class A is a base class for classes l_1^k and l_2^k, both of which are in turn base classes of C, the object identity requirement of C++ requires that the relevant subobjects of type A have different offsets within class C: $\mathit{offset}(\sigma_1) \neq \mathit{offset}(\sigma_2)$ However $\mathit{offset}(\sigma_1) = \mathsf{off2top}(a_1.\mathit{vtbl}) = \mathsf{off2top}(a_2.\mathit{vtbl}) = \mathit{offset}(\sigma_2)$ since $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$, which contradicts that the offsets are different. □

Conjecture in the other direction is not true in general as there may be duplicate vtables for the same type present at run-time. This happens in many C++ implementations in the presence of *Dynamically Linked Libraries* (or DLLs for short) as the same class compiled into executable and DLL it loads may have identical vtables inside the executable's and DLL's binaries.

Note also that we require both static types to be the same. Dropping this requirement and saying that equality of vtbl-

pointers also implies equality of the static types is not true in general because a derived class can share the vtbl-pointer with its primary base class. The theorem can be reformulated, however, stating that one static type will necessarily be a subtype of the other. The current formulation is sufficient for our purposes, while reformulation will require more elaborate discussion of the algebra of subobjects [38], which we touch only briefly.

During construction and deconstruction of an object, the value of a given vtbl-pointer may change. In particular, that value will reflect the fact that the most-derived type of the object is the type of its fully constructed part only. This, however, does not affect our reasoning, as during such transition we also treat the object to have the type of its fully constructed base only. Such interpretation is in line with the C++ semantics for virtual function calls and the use of RTTI during construction and destruction of an object. Once the complete object is fully constructed, the value of the vtbl-pointer will remain the same for the lifetime of the object.

3.7 Vtable Pointer Memoization

The C++ standard requires that information about types be available at run time for three distinct purposes:

- to support the **typeid** operator,
- to match an exception handler with a thrown object, and
- to implement the **dynamic_cast** operator.

and if any of these facilities are used in a program that was compiled with disabled RTTI, the compiler will emit an error or at least a warning. Some compilers (e.g. Visual C++) additionally let a library check presence of RTTI through a predefined macro, thus letting it report an error if its dependence on RTTI cannot be satisfied. Since our solution relies on **dynamic_cast** to perform casts at run-time, we implicitly rely on the presence of RTTI and thus fall into the setting that guarantees the preconditions of Theorem 1. Since all the objects that will be coming through a particular type switch will have the same static type, the theorem guarantees that different vtbl-pointers will correspond to different subobjects. The idea is thus to group them accordingly to the value of their vtbl-pointer and associate both jump target and the required offset with it through memoization device:

```
typedef pair⟨ptrdiff_t,size_t⟩ type_switch_info;
static unordered_map⟨intptr_t, type_switch_info⟩ jump_targets;
type_switch_info& info = jump_targets[vtbl(x)];
const void* tptr;
switch (info.second) ...
```

The code for the i^{th} case now evaluates the required offset on the first entry and associates it and the target with the vtbl-pointer of the subject. The call to adjust_ptr$\langle T_i \rangle$ reestablishes the invariant that match is a reference to type T_i of the subject x.

```
    if (tptr = dynamic_cast⟨const T_i*⟩(x)) {
        if (info.second ==0) { // supports fall−through
            info.first = intptr_t(tptr)−intptr_t(x); // offset
            info.second = i; // jump target
        }
case i: // i is a constant here − clause's position in switch
        auto match = adjust_ptr⟨T_i⟩(x,info.first);
        s_i;
    }
```

Class std::unordered_map provides amortized constant time access on average and linear in the amount of elements in the worst case. We show in the next section that most of the time we will be bypassing traditional access to its elements. We need this extra optimization because, as-is, the type switch is still about 50% slower than the visitor design pattern.

3.8 Minimization of Conflicts

Virtual table pointers are not constant values and are not even guaranteed to be the same between different runs of the application, because techniques like *address space layout randomization* or *rebasing* of the module are likely to change them. The relative distance between them will remain the same as long as they come from the same module.

Knowing that vtbl-pointers point into an array of function pointers, we should expect them to be aligned accordingly and thus have a few lowest bits as zero. Moreover, since many derived classes do not introduce new virtual functions, the size of their virtual tables remains the same. When allocated sequentially in memory, we can expect a certain number of lowest bits in the vtbl-pointers pointing to them to be the same. These assumptions, supported by actual observations, has made virtual table pointers of classes related by inheritance ideally suitable for hashing – the values obtained by throwing away the common bits on the right were compactly distributed in small disjoint ranges. We use them to address a cache built on top of the hash table in order to eliminate a hash table lookup in most of the cases.

Let Ξ be the domain of integral representations of pointers. Given a cache with 2^k entries, we use a family of hash functions $H_{kl} : \Xi \to [0..2^k - 1]$ defined as $H_{kl}(v) = v/2^l \mod 2^k$ to index the cache, where $l \in [0..32]$ (assuming 32 bit addresses) is a parameter modeling the number of common bits on the right. Division and modulo operations are implemented with bit-shifts since denominator in each case is a power of 2, which in turn explains the choice of the cache size.

Given a hash function H_{kl}, pointers v' and v'' are said to be in *conflict* when $H_{kl}(v') = H_{kl}(v'')$. For a given set of pointers $V \in 2^{\Xi}$ we can always find such k and l that H_{kl} will render no conclicts between its elements, however the required cache size 2^k can be too large to justify the use of memory. The value K such that $2^{K-1} < |V| \leq 2^K$ is the smallest value of k under which absence of conflicts is still possible. We thus allow k to only vary in range $[K, K+1]$ to

ensure that the cache size is never more than 4 times bigger than the minimum required cache size.

Given a set $V = \{v_1, ..., v_n\}$, we would like to find a pair of parameters (k, l) such that H_{kl} will render the least number of conflicts on the elements of V. Since for a fixed set V, parameters k and l vary in a finite range, we can always find the optimal (k, l) by trying all the combinations. Let $H^V_{kl} : V \to [0..2^k-1]$ be the hash function corresponding to such optimal (k, l) for the set V.

In our setting, set V represents the set of vtbl-pointers coming through a particular type switch. While the exact values of these pointers are not known till run-time, their offset from the module's base address is. This can often be sufficient to at least estimate optimal k and l in a compiler setting. In the library setting we estimate them by recomputing them after a given amount of actual collisions happened in cache.

When H^V_{kl} is injective (renders 0 conflicts on V), the frequency of any given vtbl-pointer v_i coming through the type switch does not affect the overal performance of the switch. However when H^V_{kl} is not injective, we would prefer the conflict to happen on less frequent vtbl-pointers. Given a probability p_i of each vtbl-pointer $v_i \in V$ we can compute the probability of conflict rendered by a given H_{kl}:

$$P_{kl}(V) = \sum_{j=0}^{2^k-1} \Big(\sum_{v_i \in V^j_{kl}} p_i \Big) \Big(1 - \frac{\sum\limits_{v_i \in V^j_{kl}} p_i^2}{\big(\sum\limits_{v_i \in V^j_{kl}} p_i \big)^2}\Big)$$

where $V^j_{kl} = \{v \in V | H_{kl}(v) = j\}$. In this case, the optimal hash function H^V_{kl} can similarly be defined as H_{kl} that minimizes the above probability of conflict on V.

Probabilities p_i can be estimated in a compiler settings through profiling, while in a library setting we let the user enable tracing of frequencies of each vtbl-pointer. This introduces an overhead of an increment into the critical path of execution, and according to our tests degrades the overal performance by 1-2%. By default, we do not enable frequency tracing, however, because the significant drop in the number of actual collisions was not reflected in a noticeable decrease in execution time. This was because the total number of actual collisions, even in non-frequency based caching, was much smaller than the number of successful cache hits.

Assuming the uniform distribution of v_i and substituting the probability $p_i = \frac{1}{n}$, where $n = |V|$, into the above formula we will get:

$$P_{kl}(V) = \sum_{j=0}^{2^k-1} [|V^j_{kl}| \neq 0] \frac{|V^j_{kl}| - 1}{n}$$

The value $|V^j_{kl}| - 1$ represents the amount of "extra" pointers mapped into the entry j in cache and thus H^V_{kl} obtained by minimization of probability of conflict is the same as our original H^V_{kl} minimizing the number of conflicts. An important observation here is that the exact location of these "extra" vtbl-pointers is not important, only the total number m of them is. The probability of conflict under uniform distribution of v_i is thus always going to have form $\frac{m}{n}$, where $0 \leq m < n$.

4. Evaluation

Our evaluation methodology consisted of several independent studies of the technique in order to achieve a better confidence in its validity. Our first study involved comparison of relative performance of our approach against the visitor design pattern. The second study did a similar comparison with built-in facilities of Haskell and OCaml. In the third study we looked at how well our caching mechanisms deal with some large real-world class hierarchies. In the last study we did rewrite an existing application that was based on visitors using our approach and compared the two.

4.1 Comparison with Visitor Design Pattern

Our evaluation methodology consists of several benchmarks representing various uses of objects inspected with either visitors or type switching.

The *repetitive* benchmark (REP) performs calls on different objects of the same most-derived type. This scenario happens in object-oriented setting when a group of polymorphic objects is created and passed around (e.g. numerous particles of a given kind in a particle simulation system). We include it because double dispatch becomes twice faster (27 vs. 53 cycles) in this scenario compared to others due to cache and call target prediction mechanisms.

The *sequential* benchmark (SEQ) effectively uses an object of each derived type only once and then moves on to an object of a different type. The cache is typically reused the least in this scenario, which is typical of lookup tables, where each entry is implemented with a different derived class.

The *random* benchmark (RND) is the most representative as it randomly makes calls on random objects, which will probably be the most common usage scenario in the real world.

Presence of *forwarding* in any of these benchmarks refers to the common technique used by visitors where, for class hierarchies with multiple levels of inheritance, the visit method of a derived class will provide a default implementation of forwarding to its immediate base class, which, in turn, may forward it to its base class, etc. The use of forwarding in visitors is a way to achieve substitutability, which in type switch corresponds to the use of base classes in the case clauses.

The class hierarchy for non-forwarding test was a flat hierarchy with 100 derived classes, encoding an algebraic data type. The class hierarchy for forwarding tests had two levels of inheritance with 5 intermediate base classes and 95 derived ones. While we do not advocate here for the

closed solution of §3.1, we included it in our tests to show the performance gains a closed solution might have over the open one. Our library supports both solutions with the same surface syntax, which is why we believe many users will try them both before settling on one.

The benchmarks were executed in the following configurations refered to as *Linux Desktop* and *Windows Laptop* respectively:

- *Lnx*: Dell Dimension® desktop with Intel® Pentium® D (Dual Core) CPU at 2.80 GHz; 1GB of RAM; Fedora Core 13
  - G++ 4.4.5 executed with -O2
- *Win*: Sony VAIO® laptop with Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Professional
  - G++ 4.5.2 and 4.6.1 / MinGW executed with -O2; x86 binaries
  - MS Visual C++ 2010 Professional x86/x64 binaries with profile-guided optimizations

The code on the critical path of our type switch implementation benefits significantly from branch hinting as some branches are much more likely than others. We use the branch hinting in GCC to guide the compiler, but, unfortunately, Visual C++ does not have similar facilities. Microsoft suggests to use *Profile-Guided Optimization* to achieve the same, which is why the results for Visual C++ reported here have been obtained with profile-guided optimizations enabled. The results without profile-guided optimizations can be found in the accompanying technical report [4, §10].

We compare the performance of our solution relative to the performance of visitors in Figure 4. The values are given as percentages of performance increase against the slower technique. Numbers in regular font represent cases where type switching was faster, while numbers in bold indicate cases where visitors were faster.

		Open				Closed			
		G++		MS VC++		G++		MS VC++	
		x86-32	x86-32	x86-32	x86-64	x86-32	x86-32	x86-32	x86-64
Forward	REP	16%	55%	4%	**0%**	124%	216%	124%	47%
	SEQ	56%	3%	3%	1%	640%	520%	34%	14%
	RND	56%	1%	**18%**	**27%**	603%	542%	43%	16%
	REP	33%	67%	10%	**6%**	53%	79%	31%	**9%**
	SEQ	55%	90%	153%	145%	86%	259%	185%	118%
	RND	78%	27%	18%	**6%**	88%	31%	24%	**10%**
		Lnx	Win			Lnx	Win		

Figure 4. Relative performance of type switching versus visitors. Numbers in regular font (e.g. 67%), indicate that our type switching is faster than visitors by corresponding percentage. Numbers in bold font (e.g. **18%**), indicate that visitors are faster by corresponding percentage.

We can see that type switching wins by a good margin when implemented with tag switch as well as in the presence of at least one level of forwarding. Note that the numbers are relative, and thus the ratio depends on both the performance of virtual function calls and the performance of switch statements. Visual C++ was generating faster virtual function calls, while GCC was generating faster switch statements, which is why their relative performance seem to be much more favorable for us in the case of GCC. Similarly the code for x64 is only slower relatively: the actual time spent for both visitors and type switching was smaller than that for x86, but it was much smaller for visitors than type switching, which resulted in worse relative performance.

4.2 Open vs. Closed Type Switch

With a few exceptions for x64, it can be seen from Figure 4 that the performance of the closed tag switch dominates the performance of the open type switch. We believe that the difference, often significant, is the price one pays for the true openness of the vtable pointer memoization solution.

As we mentioned in §3.1, the use of tags, even allocated by compiler, may require integration efforts to ensure that different DLLs have not reused the same tags. Randomization of tags, similar to a proposal of Garrigue [19], will not eliminate the problem and will surely replace jump tables in switches with decision trees. This will likely significantly degrade the numbers for the part of Figure 4 representing closed tag switch, since the tags in our experiments were all sequential.

The reliance of a tag switch on static cast has severe limitations in the presence of multiple inheritance, and thus is not as versatile as open type switch. Overcoming this problem will either require the use of **dynamic_cast** or techniques similar to those used for vtable pointer memoization, which will likely degrade tag switch'es performance numbers even further.

Note also that the approach used to implement open type switch can be used to implement both first-fit and best-fit semantics, while the tag switch is only suitable for best-fit semantics. Their complexity guarantees also differ: open type switch is constant on average, but slow on the first call with given subobject. Tag switch is logarithmic in the size of the class hierarchy (assuming a balanced hierarchy), including the first call. This last point can very well be seen in Figure 4, where the performance of a closed degrades significantly in the presence of forwarding, while the performance of open solution improves.

4.3 Comparison with OCaml and Haskell

We now compare our solution to the built-in pattern-matching facility of OCaml [28] and Haskell [25]. In this test, we timed small OCaml and Haskell applications performing our sequential benchmark on an algebraic data type of 100 variants. Corresponding C++ applications were working with a flat class hierarchy of 100 derived classes. The difference be-

tween the C++ applications lies in the encoding used. Kind encoding is the same as Tag encoding, but it does not require substitutability, and thus can be implemented with a direct switch on tags. It is only supported through specialized syntax in our library as it differs from the Tag encoding only semantically.

We used optimizing OCaml compiler `ocamlopt.opt` version 3.11.0 working under the Visual C++ toolset as well as the Glasgow Haskell Compiler version 7.0.3 (with -O switch) working under the MinGW toolset. All the tests were performed on the Windows 7 laptop. The timing results presented in Figure 5 are averaged over 101 measurements and show the number of seconds it took to perform a million decompositions within our sequential benchmark.

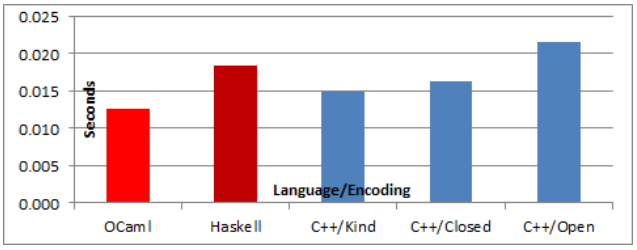

Figure 5. Performance comparison of various encodings and syntax against OCaml code

4.4 Dealing with real-world class hierarchies

We used a class hierarchy benchmark used before to study efficiency of type inclusion testing and dispatching techniques [14, 26, 44, 51]. We use the names of the benchmarks from Vitek et al [44, Table 2], since the set of benchmarks we were working with was closest to that work.

While not all class hierarchies originated from C++, for this experiment it was more important for us that the hierarchies were man-made. While converting the hierachies into C++, we had to prune inaccessible base classes (direct base class that is already an indirect base class) when used with repeated inheritance in order to satisfy semantic requirements of the C++. We maintained the same number of virtual functions present in each class as well as the number of data members. The benchmarks, however, did not preserve the types of those. The data in Figure 6 shows various parameters of the class hierarchies in each benchmark, after their adoption to C++.

The number of paths represents the number of distinct inheritance paths from the classes in the hierarchy to the roots of the hiearchy. As we showed in §**??** this number reflects the number of possible subobjects in the hierarchy. The roots listed in the table are classes with no base classes. We will subsequently use the term *non-leaf* to refer to the possible root of a subhierarchy. Leafs are classes with no children, while *both* refers to utility classes that are both roots and leafs and thus neiter have base nor derived classes. The average for the number of parents and the number of

Library	Language	Classes	Paths	Height	Roots	Leafs	Both	Parents		Children	
								avg	max	avg	max
DG2	Smalltalk	534	534	11	2	381	1	1	1	3.48	59
DG3	Smalltalk	1356	1356	13	2	923	1	1	1	3.13	142
ET+	C++	370	370	8	87	289	79	1	1	3.49	51
GEO	Eiffel	1318	13798	14	1	732	0	1.89	16	4.75	323
JAV	Java	604	792	10	1	445	0	1.08	3	4.64	210
LOV	Eiffel	436	1846	10	1	218	0	1.72	10	3.55	78
NXT	Objective-C	310	310	7	2	246	1	1	1	4.81	142
SLF	Self	1801	36420	17	51	1134	0	1.05	9	2.76	232
UNI	C++	613	633	9	147	481	117	1.02	2	3.61	39
VA2	??	3241	3241	14	1	2582	0	1	1	4.92	249
VA2	??	2320	2320	13	1	1868	0	1	1	5.13	240
VW1	Smalltalk	387	387	9	1	246	0	1	1	2.74	87
VW2	Smalltalk	1956	1956	15	1	1332	0	1	1	3.13	181
	Overalls	15246	63963	17	298	10877	199	1.11	16	3.89	323

Figure 6. Benchmarks Class Hierarchies

children were computed only among the classes having at least one parent or at least one children correspondingly.

With few useful exceptions, it generally makes sence to only apply type switch to non-leaf nodes of the class hierarchy. 71% of the classes in the entire benchmarks suite were leaf classes. Out of the 4369 non-leaf classes 36% were spawning a subhierarchy of only 2 classes (including the root), 15% – a subhierarchy of 3 classes, 10% of 4, 7% of 5 and so forth. Turning this into a cumulative distribution, $a\%$ of subhierarchies had more than b classes in them, where:

a	1%	3%	5%	10%	20%	25%	50%	64%	100%
b	700	110	50	20	10	7	3	2	1

These numbers reflect the percentage of use cases one may expect in the real word that have a given number of case clauses in them.

For each non-leaf class A we created a function performing a type switch on every possible derived class D_i of it as well as itself. The function was then executed with every possible subobject $D_i \prec \sigma_j \succ A$ it can possibly be applied to, given the static type A of the subject. It was executed multiple but the same number of times on each subobject to ensure uniformity on one side (since we do not have the data about the actual probabilities of each subobject the benchmark hierarchies) as well as let the type switch infer the optimal parameters k and l of its cache indexing function H_{kl}. We then plotted the result of each of the 4396 experiments as a point in chart of Figure 7 relating the optimal probability of conflict p achieved by the type switch and the number of subobjects n that came through that type switch. To account for the fact that multiple experiments could have resulted in the same pair (n, p), we use a shadow of each point to reflect somewhat the number of experiments yielding it.

The curves on which the results of experiments line up correspond to the fact that under uniform distribution of n subobjects, only a finite number of different values representing the probability of conflict p are possible. In particular, all such values $p = \frac{m}{n}$, where $0 \leq m < n$. The number m reflects the number of subobjects an optimal cache indexing function H_{kl} could not allocate their own entry for. We

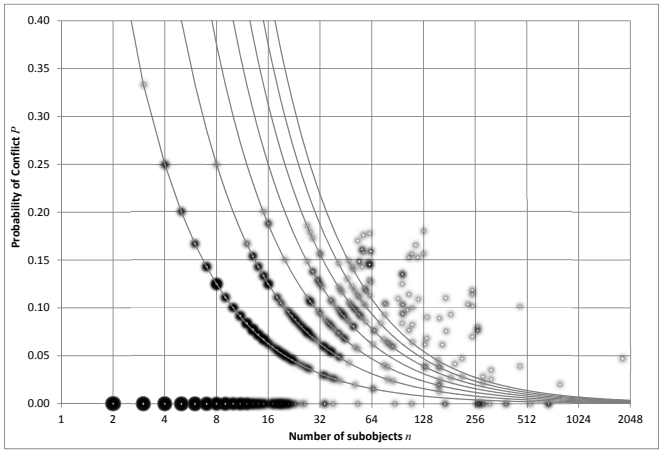

Figure 7. Probability of Conflict vs. Number of Subobjects in Hierarchy

showed in §3.8 that the probability of conflict under uniform distribution of n subobjects depends only on m. The points on the same curve (which becomes a line on a log-log plot) all share the same number m of "extra" vtbl-pointers that optimal cache indexing function could not allocate individual entries for.

While it is hard to see from the chart, 87.5% of all the points on the chart lay on the X-axis, which means that the optimal hash function for the corresponding type switches had no conflicts at all. In other words, only in 12.5% of cases the optimal H_{kl}^V for the set of vtbl-pointers V coming through a given type switch had non-zero probability of conflict. This is why the average probability of conflict for the entire set is only 1.17. Experiments laying on the first curve amount to 5.58% of subhierarchies and represent the cases in which optimal H_{kl}^V had only one "extra" vtbl-pointer. 2.63% experiments had H_{kl}^V with 2 conflicts, 0.87% with 3 etc. These numbers do not indicate that the hash function we used is better than other hash functions, they do indicate instead that the set of vtbl-pointers present in many applications is particularly suitable for such a hash function.

4.5 Refactoring an existing visitors based application

For this experiment we have reimplemented a visitor based C++ pretty printer for Pivot[39] using our pattern-matching library. Pivot's class hierarchy consists of 154 node kinds representing various entities in the C++ program. The original code had 8 visitor classes each handling 5, 7, 8, 10, 15, 17, 30 and 63 cases, which we turned into 8 match statements with corresponding numbers of case clauses. Most of the rewrite was performed by sed-like replaces that converted visit methods into respective case-clauses. In several cases we had to manually reorder case-clauses to avoid redundancy as visit-methods for base classes were typically coming before the same for derived, while for type switching we needed them to come after. Redundancy checking support provided by our library was invaluable in finding out all such cases.

Both pretty printers were executed on a set of header files from the C++ standard library and the produced output of both program was byte-to-byte the same. We timed execution of the pretty printing phase (not including loading and termination of the application or parsing of the input file) and observed that on small files (e.g. those from C run-time library and few small C++ files) visitors-based implementation was faster because the total number of nodes in AST and thus calls did not justify our set-up calls. In particular, visitor-based implementation of pretty printer was faster on files of 44–588 lines of code, with average 136 lines per those inputs, where visitors win. On these input files it is faster by 1.17%–21.42% with an average speed-up of 8.75%. Open type switch based implementation of pretty printer was faster on files of 144–9851 lines of code, with average 3497 lines per those input files, where open type switch wins. On these inputs it is faster by 0.18% – 32.99% with an average speed-up of 5.53%.

Figure 8 shows memory usage as well as cache hits and misses for the run of our pretty printer on 'queue' standard library header (it has the largest LOC after preprocessing in our test set).

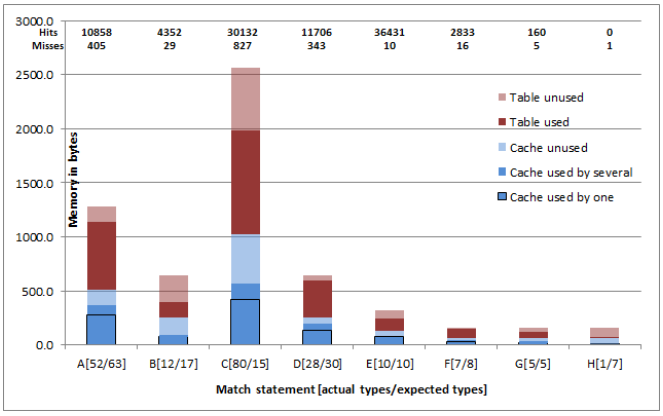

Figure 8. Memory usage in real application

The bars represent the total amount of memory in bytes each of the 8 match statements (marked A-H) used. The info $[N/M]$ next to the letter indicates the actual number of different subobjects (i.e. vtbl-pointers) N that came through that match statement, and the number of case clauses M the match statement had (the library uses it as an estimate of N). N is also the number of cases the corresponding match statement had to be executed sequentially (instead of a direct jump).

The blue part of each bar corresponds to the memory used by cache, while the maroon – to the memory used by the hash table. Transparent parts of both colors indicate the allocated extra memory that is not holding any data. The black box within the blue part also indicates the amount

of entries in the cache that are allocated for only one vtbl pointer and thus never result in a cache miss. The non-transparent part without black box represents the percentage of vtbl-pointers that have to share their cache entry with at least one other vtbl-pointer and thus may result in collisions during access.

The actual number of hits and misses for each of the match statements is indicated on top of the corresponding column. The sum of them is the total amount of calls made. Hits indicate situation when we found entry in cache and didn't have to make roundtrip to the hash-table to get it. Misses indicate the number of cases during actual run we had to pick the entry from the hash table and update the cache with it. The number of misses is always larger then or equal to N.

The library always preallocates memory for at least 8 sub-objects to avoid unnecessery recomputations of optimal parameters k and l – this is the case with the last 3 match statements. In all other cases it allocate the amount of memory proportional to the smallest power of 2 that is greater of equal than $\max(M, N)$. The table does not have to be hash table and can be implemented with any other container i.e. sorted vector, map etc. that let us find quickly by a given vtbl-pointer the data associated with it. If we implement it with sorted vector, the red part will shrink to only the non-transparent part.

5. Related Work

Extensible Visitors with Default Cases [49, §4.2] attempt to solve the extensibility problem of visitors; however, the solution has problems of its own. The visitation interface hierarchy can easily be grown linearly, but independent extensions by different authorities require developer's intervention. On top of the double dispatch the solution will incur two additional virtual calls and a dynamic cast for each level of visitor extension. The solution is simpler with virtual inheritance, which adds even more indirections.

Löh and Hinze proposed to extend Haskell's type system with open data types and open functions [31]. The solution allows top-level data types and functions to be marked as open with concrete variants and overloads defined anywhere in the program. The semantics of open extension is given by transformation into a single module, which assumes a whole-program view and thus is not an open solution unfortunately. Besides, open data types are extensible but not hierarchical, which avoids the problems discussed here.

Polymorphic variants in OCaml [19] allow the addition of new variants as well as define subtyping on them. The subtyping, however, is not defined between the variants, but between combinations of them. This maintains disjointness between values from different variants and makes an important distinction between *extensible sum types* like polymorphic variants and *extensible hierarchical sum types* like classes. Our memoization device can be used to implement pattern matching on polymorphic variants.

Tom is a pattern-matching compiler that can be used together with Java, C or Eiffel to bring a common pattern matching and term rewriting syntax into the languages [34]. In comparison to our approach, Tom has much bigger goals: the combination of pattern matching, term rewriting and strategies turns Tom into a fully fledged tree-transformation language. Its type patterns and %match statement can be used as a type switch; however, Tom's handling of type switching is based on decision trees and an instanceof-like predicate, which are inefficient.

Pattern matching in Scala [35] also supports type switching through type patterns. The language supports extensible and hierarchical data types, but their handling in a type switching constructs varies. Sealed classes are handled with an efficient switch over all tags, while extensible classes are similarly approached with a combination of an InstanceOf operator and a decision tree [16].

6. Conclusions and Future Work

Type switching is an open alternative to visitor design pattern that overcomes the restrictions, inconveniences, and difficulties in teaching and using, typically associated with it. Our implementation of it comes close or outperforms the visitor design pattern, which is true even in a library setting using a production-quality compiler, where the performance base-line is already very high.

In the future we would like to provide a compiler implementation of our technique, which will enable a better surface syntax, improved diagnostics, increased performance and many other benefits hard to achieve in a library implementation.

Acknowledgments

We would like to thank Xavier Leroy and Luc Maranget for valuable feedback and suggestions for improvements on the initial idea, Gregory Berkolaiko for ideas related to minimization of conflicts, Jaakko Jarvi for assistance in comparison to other languages, Andrew Sutton, Peter Pirkelbauer and Abe Skolnik for helpful discussions and comments to numerous rewrites of this paper. We also benefitted greatly from insightful comments by anonymous reviewers on earlier revisions of this work. We would also like to thank Karel Driesen for letting us use his class hierarchies benchmark for this work.

References

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`, 2007.

[2] A. Appel, L. Cardelli, K. Fisher, C. Gunter, R. Harper, X. Leroy, M. Lillibridge, D. B. MacQueen, J. Mitchell, G. Morrisett, J. H. Reppy, J. G. Riecke, Z. Shao, and C. A.

Stone. Principles and a preliminary design for ML2000. March 1999.

[3] L. Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. URL http://dl.acm.org/citation.cfm?id=5280.5303.

[4] Blind Review. Technical report submitted as supplementary material. Technical Report XXXX, University, Nov. 2011.

[5] L. Cardelli. Compiling a functional language. In *Proc. of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802037. URL http://doi.acm.org/10.1145/800055.802037.

[6] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The modula3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 202–212, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75295. URL http://doi.acm.org/10.1145/75277.75295.

[7] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 271–287, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: 10.1145/165854.165905. URL http://doi.acm.org/10.1145/165854.165905.

[8] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI, March 2001. http://www.codesourcery.com/public/cxx-abi/.

[9] N. H. Cohen. Type-extension type test can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115297. URL http://doi.acm.org/10.1145/115372.115297.

[10] W. R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag. ISBN 3-540-53931-X.

[11] O.-J. Dahl. *SIMULA 67 common base language, (Norwegian Computing Center. Publication).* 1968. ISBN B0007JZ9J6.

[12] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 83–100, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: 10.1145/236337.236344. URL http://doi.acm.org/10.1145/236337.236344.

[13] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. Jan. 1975.

[14] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):33:1–33:56, Oct. 2008. ISSN 0164-0925. doi: 10.1145/1391956.1391960. URL http://doi.acm.org/10.1145/1391956.1391960.

[15] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51459-1.

[16] B. Emir. *Object-oriented pattern matching*. PhD thesis, Lausanne, 2007.

[17] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.

[18] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5.

[19] J. Garrigue. Programming with polymorphic variants. In *ACM Workshop on ML*, Sept. 1998.

[20] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36:139–156, February 2006.

[21] N. Glew. Type dispatch for named hierarchical types. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 172–182, New York, NY, USA, 1999.

[22] C. Grothoff. Walkabout revisited: The runabout. In *ECOOP 2003 - Object-Oriented Programming*, pages 103–125. Springer-Verlag, 2003. URL http://grothoff.org/christian/runabout/runabout.pdf.

[23] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199475. URL http://doi.acm.org/10.1145/199448.199475.

[24] International Organization for Standardization. *ISO/IEC 14882:2011: Programming languages: C++*. Geneva, Switzerland, 3rd edition, 2011. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue

[25] S. P. Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[26] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In *In Proc. European Conference on Object-Oriented Programming, ECOOP'97, Lecture Notes in Computer Science*, pages 128–145. Springer-Verlag, 1997.

[27] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Berlin / Heidelberg, 1998.

[28] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0.

[29] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-266-7.

[30] B. Liskov, R. R. Atkinson, T. Bloom, E. B. Moss, R. Schaffert, and A. Snyder. Clu reference manual. Technical report, Cambridge, MA, USA, 1979.

[31] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM.

[32] Microsoft Research. Phoenix compiler and shared source common language infrastructure. `http://research.microsoft.com/phoenix/`, 2005.

[33] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262132559.

[34] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3.

[35] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the scala programming language (second edition. Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, 2006.

[36] B. C. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 439–456, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3.

[37] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8585-9. URL `http://dl.acm.org/citation.cfm?id=645980.674267`.

[38] T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for c++ multiple inheritance. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 67–80, New York, NY, USA, 2011. ACM.

[39] G. D. Reis and B. Stroustrup. A principled, complete, and efficient representation of C++. In *Proc. Joint Conference of ASCM 2009 and MACIS 2009*, volume 22 of *COE Lecture Notes*, pages 407–421, December 2009.

[40] J. G. Rossie, Jr. and D. P. Friedman. An algebraic semantics of subobjects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 187–199, New York, NY, USA, 1995. ACM.

[41] L. Schubert, M. Papalaskaris, and J. Taugher. Determining type, part, color, and time relationships. *Computer*, 16:53–60, 1983. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.1983.1654198.

[42] D. A. Spuler. Compiler Code Generation for Multiway Branch Statements as a Static Search Problem. Technical Report Technical Report 94/03, James Cook University, Jan. 1994.

[43] Tom Duff. Duff's Device, Aug 1988. http://www.lysator.liu.se/c/duffs-device.html.

[44] J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 142–157, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. doi: 10.1145/263698.263730. URL `http://doi.acm.org/10.1145/263698.263730`.

[45] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 13–24, New York, NY, USA, 2005. ACM. ISBN 1-58113-999-3. doi: 10.1145/1040294.1040296. URL `http://doi.acm.org/10.1145/1040294.1040296`.

[46] P. Wadler. The expression problem. Mail to the java-genericity mailing list, November 1998.

[47] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety prooffor multiple inheritance in c++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 345–362, New York, NY, USA, 2006. ACM.

[48] N. Wirth. Type extensions. *ACM Trans. Program. Lang. Syst.*, 10(2):204–214, Apr. 1988. ISSN 0164-0925. doi: 10.1145/42190.46167. URL `http://doi.acm.org/10.1145/42190.46167`.

[49] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. ACM.

[50] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005.

[51] Y. Zibin and J. Y. Gil. Efficient subtyping tests with pq-encoding. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 96–107, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504290. URL `http://doi.acm.org/10.1145/504282.504290`.