Coloring for Shared Object-Oriented Libraries

Jean Privat[1] and Floréal Morandat[2]

[1] Département d'informatique, UQAM, Montreal, Canada — privat.jean@uqam.ca
[2] LIRMM, CNRS, Université de Montpellier II, France — morandat@lirmm.fr

Abstract. Coloring is an implementation technique for multiple inheritance which is as efficient as single inheritance but only works in a global compilation or global linking framework. In this short paper we propose a variation on coloring that makes it compatible with shared libraries. Our proposal is usable for *method invocation*, *attribute access*, and *subtype check*, and respects five functional and efficiency requirements: constant time, quadratic space, short code, compatible with multiple inheritance, and incremental. We validate our proposal with theoretical evaluations, simulations and real executions which all show that the overhead for shared libraries remains small.

Introduction

Coloring is an implementation technique with tables that avoids the overhead of multiple inheritance but requires knowledge of the whole class hierarchy [1, 2]. It can be applied to efficiently implement three main object-oriented mechanisms: method invocation, attribute access, and subtype check [2–7]. Coloring fulfills four functional and efficiency requirements: constant time, quadratic space in the size of the program, short code, and compatible with multiple inheritance. However, coloring is not an incremental technique since: (i) a global computation requiring knowledge of the whole hierarchy is needed before generating any code; (ii) generated code is specific to a program and cannot be shared with other programs.

In this paper we propose a variation on coloring that makes it compatible with dynamically linked shared libraries where classes can be compiled separately then shared by different programs (shared on disk and in memory). Our variation of coloring has an overhead, but we show by theoretical and experimental evaluations that this overhead is reduced by modern processor pipelines and efficient caching mechanisms.

In the following, Section 1 summarizes the coloring technique; Section 2 presents our proposal; Section 3 experimentally compares the two techniques; and Section 4 discusses related works on variations of coloring.

1 A Global Implementation Technique: Coloring

The underlying idea of coloring is to implement the three main object-oriented mechanisms by retrieving information stored in a table attached to objects (the

Method invocation
Send to `self` a message corresponding to the introducing method m.

```
1 load [self + #ctableOffset] → ctable                    B + 2L
2 load [ctable + #methodPos_m] → method
3 call method
```

Attribute access
`self` accesses the attribute a introduced in the class c.

```
1 load [self + #ctableOffset] → ctable                    3L + 1
2 load [ctable + #baseAttrPos_c] → baseOffset
3 add self + baseOffset → baseAddress
4 load [baseAddress + #attrOffset_a] → value
```

For write access, change the last line with:

```
4 store value → [baseAddress + #attrOffset_a]
```

Subtype check
Is `self` an instance of the class c?

```
1 load [self + #ctableOffset] → ctable                    2L + 2
2 load [ctable + #classPos_c] → class
3 comp class, #classId_c
4 bne #checkFailed
```

Fig. 1. Main Object-Oriented Mechanisms Implemented With the Original Coloring

class table): `target = self->ctable[targetPosition]`. The key point of this technique is to ensure that the positions in class tables are invariant by inheritance. The knowledge of the whole class hierarchy of a program is needed by coloring algorithms to compute *position values* that respect this invariance requirement. Therefore, coloring is an intrinsically global technique since all positions must be known before running the program and they must be computed at compile or link time. Therefore, coloring is not compatible with dynamic loading, nor even with shared libraries.

Figure 1 shows the implementation of the three mechanisms. The language used in an abstract assembly language borrowed from [9–11]. Names prefixed with `#`, like `#tableOffset`, stand for constant immediate values; other names stand for registers. The cycle count of each implementation is given as a function of L and B. L is the latency for memory loads; a typical value is 3 cycles. B is the latency for indirect branching; a typical value is 10 cycles. We do not consider cache misses in this theoretical evaluation (a typical cache miss costs 100 cycles or even more) but they will appears in our experimental evaluation in Section 3.

Method invocation is straightforward: each introduction of a method m is associated with a *position value* $\texttt{methPos}_m$. Each class that has the method m (or a redefinition m') has the address of m (or the address of m') in its table at $\texttt{methPos}_m$.

Attribute access is a bit tricky. Coloring could be used to compute the positions of attributes in the instance itself, thus attribute access could be something like: `load [self + #attrPos`$_a$`] → value`. The drawback of this approach is that since coloring may generate holes in tables, there is a risk that the size of the instances will inflate. That could be problematic with a thousand instances or more. [2] has experimented coloring on real programs and showed that in the worst case the instance size of some classes can inflate up to 800%. A solution to avoid this inflation problem is to use an indirection and store the real position of attributes in the class table. This technique is called *accessor simulation* in [2] and *field dispatching* in [12]. Since all attributes introduced in a same class can be grouped together in the instance, only the position of the group needs to be stored in the class and the position of an attribute in the group is used to access this exact attribute. For each class, the location of each group of attributes (for the class and for inherited classes) in the instance is determined (it is the instance layout). Note that two proper instances of the same class have the same layout and two proper instances of directly related classes can have two different layouts. To summarize attribute access: each class c that introduces attributes is associated with a *position value* `#baseAttrPos`$_c$; for an attribute a introduced in a class c, `#attrOffset`$_a$ is the position of a in the group of attributes introduced in c; in each class table, the location of the group of attributes introduced in a superclass c is stored at `#baseAttrPos`$_c$.

Subtype check uses the technique presented in [6]. It's an adaptation of the Cohen's technique [5]. Each class c is associated with an unique identifier `classId`$_c$ and a *position value* `classPos`$_c$. Each class that is a subclass of c (or that is c) has in its table at `classPos`$_c$ the value `classId`$_c$. This technique is free of false positive iff two easy requirements are respected:

- there are no conflicting values among class identifiers and anything else we can find in the class table;
- class tables are large enough (or `classPos` small enough) to ensure that no `classPos` value can go beyond the limit of any class table.

2 Our Proposal

Our proposal is to not hard-code immediate values in the generated code but to get them, instead, using a supplementary indirection such that:

- the generated code is unrelated to any global knowledge: the generated code can be used, loaded or shared by any program;
- values and tables are specifically computed for each program: initial computation can be done at link time or at load time.

With an indirection, the three object-oriented mechanisms can be implemented as:

```
targetPos = localTable[targetPosPos]
target = self->ctable[targetPos]
```

Method Invocation
Send to `self` a message corresponding to the method m introduced in the class c.

```
1 load [self + #ctableOffset] → ctable
2 load [localTable_c + #methPosPos_m] → methPos
3 add ctable + methPos → methAddress
4 load [methAddress] → method
5 call method
```

$B + 2L + 2$

Attribute Access
`self` accesses the attribute a introduced in the class c.

```
1 load [self + #ctableOffset] → ctable
2 load [localTable_c + #baseAttrPosPos_c] → basePos
3 add ctable + basePos → baseOffsetAddress
4 load [baseOffsetAddress] → baseOffset
5 add self + baseOffset → baseAddress
6 load [baseAddress + #attrOffset_a] → value
```

$3L + 3$

For write access, change the last line with:

```
6 store value → [baseAddress + #attrOffset_a]
```

Subtype Check
Is `self` an instance of the class c?

```
1 load [self + #ctableOffset] → ctable
2 load [localTable_c + #classPosPos_c] → classPos
3 load [localTable_c + #classIdPos_c] → classId
4 add ctable + classPos → classAddress
5 load [classAddress] → class
6 comp class, classId
7 bne #checkFailed
```

$2L + 4$

Fig. 2. Main Object-Oriented Mechanisms Implemented with our Proposal

Our expectation is to rely on modern processor capabilities with pipelines and efficient caching mechanisms so that the supplementary cost is mainly avoided.

There is a local table $\texttt{localTable}_c$ for each class c and it is used to implement the object-oriented mechanisms introduced by class c. The structure of a local table (its length and the role of each specific position in the table) is determined when separately compiling the class. However, the content of local tables is specific to each program and is computed at link time or at load lime. During the separate compilation of any subclass or client of a class c, the compilation of object-oriented mechanisms introduced by class c requires the structure of the local table of c to be statically known.

The structure of a local table $\texttt{localTable}_c$ is the following: (i) two positions ($\texttt{\#classPosPos}_c$ and $\texttt{\#classIdPos}_c$) in the local table are reserved to store the $\texttt{classPos}_c$ and $\texttt{classId}_c$ values; (ii) if c introduces attributes, a position ($\texttt{\#baseAttrPosPos}_c$) in the local table is reserved to store the $\texttt{baseAttrPos}_c$ value; (iii) for each method m introduced in c (redefinitions are ignored), a position ($\texttt{\#methPosPos}_m$) in the local table is reserved to store the $\texttt{methPos}_m$ value.

	Method invocation		Attribute access		Subtype check	
	space	time	space	time	space	time
Original coloring	3	$B + 2L$	4	$3L + 1$	4	$2L + 2$
Our proposal	5	$B + 2L + 2$	6	$3L + 3$	7	$2L + 4$
Difference	+2	+2	+2	+2	+3	+2

Fig. 3. Theoretical Space and Time Cost.

At global time (link time or load time), a coloring is computed, global values (`methPos`, `baseAttrPos`, `classPos` and `classId`) are stored in these tables, and class tables are built.

Figure 2 shows the implementation of the three object-oriented mechanisms using our proposal. The theoretical evaluation of time efficiency is based on an abstract processor specification (mostly the same as processor P95 in [11]): (i) the maximum number of running instruction per cycle is 2; (ii) 2 `load` instructions cannot be executed in strict parallel but need one cycle delay. To explicit the parallel execution of instructions, a possible schedule diagram is drawn on the figure. Note that the implementations in Figure 1 are not parallelizable.

Figure 3 summarizes the theoretical space and time cost of the original coloring technique (presented in Section 1) and our proposal (presented in the present section). The space cost of our proposal is due to: (i) longer code size for the implementation of object-oriented mechanisms, and (ii) local tables. The code size of our proposal is 2 or 3 assembly instructions longer than the original approach. Even if it is a small value, it means an increase between 50% and 75%. Note that this increase only affects the code segment of the process. The size of the local tables is linear since the maximal size is $3c + m$ where c is the number of classes and m the number of methods introduced. The total size of tables (including local tables and class tables) is still quadratic, thus, it fulfills our second functional and efficiency requirement.

Even if implementations are longer, the time cost of each mechanism is still constant, thus, it fulfills our first functional and efficiency requirement. Thanks to parallelism, the time cost of the supplementary indirection is strongly reduced for the three mechanisms as the time cost difference is only two cycles.

3 Experimental Evaluation

3.1 Experimentation Framework

We used the PRM framework [13, 14] to evaluate the time difference between the implementation using the original coloring and the one using our proposal. PRM is an object-oriented language that has features that will stress our proposal :

- fully object-oriented: each value is an object, each subroutine call is a potential polymorphic method invocation, each state access is an attribute access;
- multiple inheritance of classes: each method or attribute is subject to multiple inheritance;

	Stripped binary size (ko)
Original coloring	1 840
Our proposal	1 852
Difference	+0.65%

Fig. 4. Space Results.

	Instruction			Data		
	Refs	L1 miss	L2 miss	Refs	L1 miss	L2 miss
Original coloring	1 320 766 708	1 193 308	18 999	787 697 131	2 611 717	798 213
Our proposal	1 386 050 419	1 120 329	19 560	879 254 251	2 880 118	815 082
Difference	+4.94%	+6.11%	+2.95%	+11.62%	+10.27%	+2.11%

Fig. 5. Cache Simulation Results.

– unsafe covariant typing policy: many subtype checks must be inserted by the compiler to ensure the dynamic correctness of the programs.

The test program is the PRM compiler itself, written in PRM, that has 41 739 LOC, 382 classes, 2 590 methods and 1 905 attributes. The test machine is an Intel Core2 Duo T7500 2.20GHz. Only one core was used during experiments.

We compiled the PRM compiler with two flavors: one with the original coloring, one with our proposal. For each flavor, each module of the compiler is compiled separately: C code is generated then transformed to binary with gcc (version 4.2.3 with `-O2`). In one flavor, C contains static immediate values; in the other flavor, C contains access to local tables (declared as `extern`). Remark: separate compilation units in PRM are not classes but modules, thus there is a single local table per module instead of a local table per class. A last C file is generated and transformed to binary. In one flavor, it contains the definitions of the class tables, in the other flavor, it also contains the definitions of the local tables. Binary files are then linked accordingly to produce two final executables.

Figure 4 shows the static size of the final executable for both flavors. The small difference is due to the additional class table definition and the extended implementation size of the three object-oriented mechanisms.

3.2 Cache Simulation

We investigated the cache usage of the PRM compiler when it is compiled using the original coloring and our proposal. We used Cachegrind, a tool from the Valgrind tool suite [15], that simulates the processor to track cache misses.

Figure 5 shows that even if each method invocation, each attribute access, and each subtype check requires a supplementary indirection, the processor caches handle this cost efficiently. Remark: the simulation is deterministic and does not track cache misses related to other threads or kernel activities.

	minimal	average	maximal
Original coloring	5.88	6.05	6.38
Our proposal	6.10	6.26	6.46
Difference	+3.74%	+3.47%	+1.25%

Fig. 6. Time Results.

3.3 Real Run

The last experiment is a run of the compiler in a real environment. We measured the user time given by the `time` command. 20 consecutive runs were done, the two worst were removed. Figure 6 summarizes the results we obtained. The average overhead is under +4%. That is a very good result. For comparison, on the same benchmark, the difference between attribute coloring and accessor simulation is near +20%.

4 Related Work

In [16], we adapted the coloring technique to a separate compilation framework. The idea was to substitute symbols with computed values in the binary file of separately compiled modules. We got the exact same efficiency than coloring and we also included other techniques like direct static call and binary tree dispatch. However, our proposal was only compatible with global linking and unusable with shared libraries.

In [7, 17], Ducournau proposes perfect hashing for subtype checks. This is a generalization of the coloring technique that is compatible with dynamic loading. Perfect hashing offers a trade-off between space and time efficiency (compact class table and slow implementation or large table full of holes and fast implementation).

5 Conclusion

We presented in this article a variation of the coloring implementation technique for the three main object-oriented mechanisms: method invocation, attribute access, and subtype check. Our proposal adds the compatibly with shared libraries while retaining time and space efficient and compatibility with multiple inheritance of the original coloring technique.

We evaluated the overhead of our proposal by comparing it with the original coloring: (i) theoretically on an abstract processor specification, (ii) experimentally with a simulation of cache misses, and (iii) experimentally with real executions of a large program. Results show that the overhead of our proposal is lower than we initially expected, thus it is a usable technique for shared library linked at link time or at load time.

Our proposal may also be usable within a virtual machine with strong requirements on dynamic loading. However, two issues remain to investigate to better

accommodate dynamic loading: (i) the efficient incremental recomputation of local tables and class tables, and (ii) the management of existing instances. For (ii), attributes do not need anything specific thanks to the accessor simulation but the class table pointer may need to be updated. We see here two options: either use another indirection to access the class table (causing an overhead, even when dynamic loading is not used), or walk trough all instances and update their class table reference (it should be easy to implement if the execution engine already has a precise garbage collector).

References

1. Lippman, S.B.: Inside the C++ Object Model. Addison-Wesley, New York (NY), USA (1996)
2. Ducournau, R.: Implementing statically typed object-oriented programming languages. Technical Report 02-174, LIRMM, Montpellier (2002)
3. Dixon, R., McKee, T., Schweitzer, P., Vaughan, M.: A fast method dispatcher for compiled languages with multiple inheritance. In: Proc. OOPSLA'89. (1989)
4. Pugh, W., Weddell, G.: Two-directional record layout for multiple inheritance. In: Proc. PLDI'90. (1990) 85–91
5. Cohen, N.H.: Type-extension type tests can be performed in constant time. Programming languages and systems **13**(4) (1991) 626–629
6. Vitek, J., Horspool, R.N., Krall, A.: Efficient type inclusion tests. In: Proc. OOPSLA'97. (1997) 142–157
7. Ducournau, R.: Coloring, a versatile technique for implementing object-oriented languages. Technical Report 06-001, LIRMM, Montpellier (2006)
8. Takhedmit, P.: Coloration de classes et de propriétés : étude algorithmique et heuristique. Mémoire de DEA, Université Montpellier II (2003)
9. Driesen, K., Hölzle, U.: Minimizing row displacement dispatch tables. In: Proc. OOPSLA'95. (1995) 141–155
10. Driesen, K., Hölzle, U., Vitek, J.: Message dispatch on pipelined processors. In: Proc. ECOOP'95. (1995) 253–282
11. Driesen, K.: Efficient Polymorphic Calls. Kluwer Academic Publisher (2001)
12. Zibin, Y., Gil, J.: Two-dimensional bi-directional object layout. In: Proc. ECOOP'2003. (2003) 329–350
13. Privat, J.: De l'expressivité à l'efficacité, une approche modulaire des langages à objets — Le langage PRM et le compilateur prmc. Thèse d'informatique, Université Montpellier II (2006)
14. Privat, J.: PRM—the language. 0.2. Technical Report 06-029, LIRMM, Montpellier (2006)
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6) (2007) 89–100
16. Privat, J., Ducournau, R.: Link-time static analysis for efficient separate compilation of object-oriented languages. In Ernst, M., Jensen, T., eds.: Workshop on Program Analysis for Software Tools and Engineering PASTE'05. (2005) 29–36
17. Ducournau, R.: Perfect hashing as an almost perfect subtype test. To appears in ACM Transactions on Programming Languages and Systems (2008)
18. Chambers, C., Ungar, D.: Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In: Proc. OOPSLA'89. (1989) 146–160