1 - not yet defined: see method definition below.

2 - already defined in the class: nothing to do.

3 - already defined not in the class but in one of its superclass. The effective procedure depends on the lookup strategy: if the class of the method is more prior, then we override the definition (see method redefinition) else we do nothing.

Several removing exist:

4 - if the method is defined in one of the superclass: then the index does not change.

5 - else a new hole appears, this color becomes free.

Let's formally define the method **definition**. The function *findColor* provides an index (color) that is not used in a given set of classes (*colors(class)*). If there are no free color, it adds a new color in the system (*N + 1*) and delivers it:

$Colors = 1..N$

$colors(class) \cong \{c : Colors \mid cache(class, c) = procedure\}$

$freeColors(class) \cong Colors - colors(class)$

$findColor(classes) = c \Leftrightarrow$

$[(\forall\ cl{:}classes \bullet \neg\ (c \in colors(cl))) \wedge c \leq N] \vee$

$[\cap_{cl \in classes} freeColors(cl)) \wedge c = N + 1]$

Four distinct cases of method definition in a given class C occur: selector **s** is

6 - specific, $s \mapsto findColor(subclasses(C))$,

7 - redefined, no change,

8 - separate,

$IF\ currentColor(s) \in freeColors(C)$

$THEN\ s \mapsto currentColor(s)$

$ELSE\ s \mapsto freeColors(\{class \mid$

$currentColor(s) \in colors(class)\})$

9 - declared:

$IF\ (\forall\ cl{:}subclasses(C) \bullet (currentColor(s) \in freeColors(cl) \vee$

$(s \in selectors(C)))$

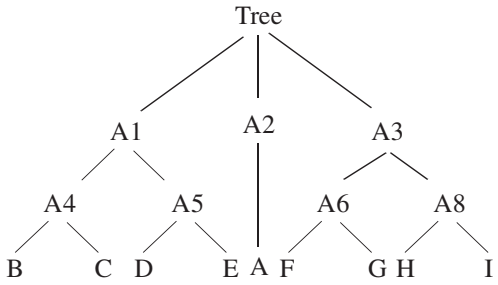
$THEN\ s \mapsto currentColor(s)] \vee$

$ELSE\ s \mapsto freeColors(\{class \mid$

$currentColor(s) \in colors(class)\})]$

When **removing** a method, an implementation alternative occurs. If a color becomes "never used", shall we reduce the range of index or no. The first case one implies to compress the cache. The second implies to let a empty line in the cache. Our opinion is that the second choice is greater because of dynamic modifications: if we define a new method after, it may require a new color.

It is easy and efficient to implement the function *findColor* in a cache based algorithm. It sum up in vectorial operations on the large cache (comparison, union, intersection) with lines given by selectors and columns provided by classes of the frontier.



The best squeezing rate is winning for the comb with empty intersection of complete behaviors.

Appendix B. Smalltalk-80 results

Next figure sum up some Smalltalk-80 staistics. Classes are ordered by their number of selectors:

i	Class	n	ns	M	nc	N
1	VariableDefinition	10	19	13	14	13
2	Change	10	28	25	26	25
3	StandardSystemView	8	73	65	69	65
4	OSErrorHolder	20	74	61	62	61
5	CompiledCode	5	88	84	86	84
6	ProgramNode	18	90	53	57	53
7	BitBlt	9	91	64	69	64
8	StringHolder	7	99	57	50	57
9	Integer	4	99	95	97	95
10	Stream	18	159	104	100	104
11	InstructionClient	7	170	152	150	152
12	SequenceableCollection	27	221	112	121	112
13	DisplayObject	17	238	111	117	111
14	MouseMenuController	29	242	126	102	126
15	Magnitude	17	248	140	110	140
16	View	31	249	152	176	152
17	Controller	36	258	136	117	136
18	Model	29	366	128	99	128
19	Collection	41	380	160	165	160
20	Behavior	371	995	271	372	271
	Object	766	4533	371	378	372

where *n* is the number of classes of the *Class* subsystem, *ns* is its number of selectors, *M* is the size of maximum complete behavior, *nc* is the average number of conflicts by selector and *N* the resulting number of colors. Note that N is close to M and M ≤ chromatic number ≤ N.

Real profit is equal to simple cache size minus conflict graph size minus colored cache size: ***(n * ns) - (ns * nc) - (n * N)***. We verified that only for Object subsystem, real profit is positive.

Appendix C. Incremental coloring algorithm

From an indexing point of view, all the modifications of the inheritance graph resume in adding or removing a method. Here we do not deal with the content of the cache, we only worry about the indexes. Of course, our implemented algorithm checks valid contents.

This abstract view does not really fit with an efficient implementation. We present it that way to be clearer.

a - class level:

We distinguish four dual primitive operations:

- add/remove one subclass with empty method dictionary to/from one class: no effect on coloring,

- add/remove a superclass to/from a class is the same as add[6]/remove all the visible selectors of this superclass in the complete behavior of the class.

Adding a class with methods and other superclasses, replacing a superclass, are a composition of these primitives.

b - method level:

We have the same classification: add/remove a method in a class. Renaming a method is a composition of the above primitives.

Three distinct cases of method **adding** occur:

6. We had better use the word addition than definition in this context, because added method come from an inherited class .

[KIC 90] KICZALES, G., RODRIGUEZ, L.
Efficient Method Dispatch in PCL,
ACM proceedings of POPL 1990 p. 99-105 (1990)

[MAS 90] MASINI, G., NAPOLI, A., COLNET D., LEONARD, D., TOMBRE, K.
Les langages à objets.
InterEditions (Octobre 1990)

[MEY 88] MEYER. B.
Object-oriented Software Construction.
Prentice-Hall (1988)

[MOO 86] MOON, D.A.
Object-oriented Programming with Flavors,
Proceedings of OOPSLA'86 (September 1987)

[ROY 90] ROYER, J.C.
Extensions Orientées Objets de SCHEME
Journées Francophones des Langages Applicatifs - Bigre + Globule n^o 69 pp. 148-158
La Rochelle, Janvier 1990

[ROY 91] ROYER, J.C.
Une étude de l'héritage dans le cadre d'une utilisation plus stricte en programmation,
Rapport LIST N^o 9/91, Université de Nantes (Juin 1991)

[SUZ 83] SUZUKI, N. & TERADA, M.
Creating efficient systems for object-oriented languages,
ACM proceedings of POPL 1983 p. 200-296 (1983)

[HOL 91] HOLZLE, U., CHAMBERS. C., UNGAR. D.
Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches
Proceedings of ECOOP'91 p. 21-38 (1991)

Appendix A. EROS

Even if Smalltalk-80 represents a good structuration, mesurements should be done in C++, CLOS or Eiffel to validate multiple inheritance. We decided to use our own multiple inheritance system EROS to test some particular graphs like apple, comb, tree, branch and massive redefinition rate.

EROS is an experimental platform designed to validate some efficiency ideas, especially static caches with colored indexes. The aim is to customize quickly some special features on a Smalltalk-80 layer. Indeed basic ideas are inspired from ObjVlisp of Pierre Cointe [COI 87] which is a simple but powerful language model.

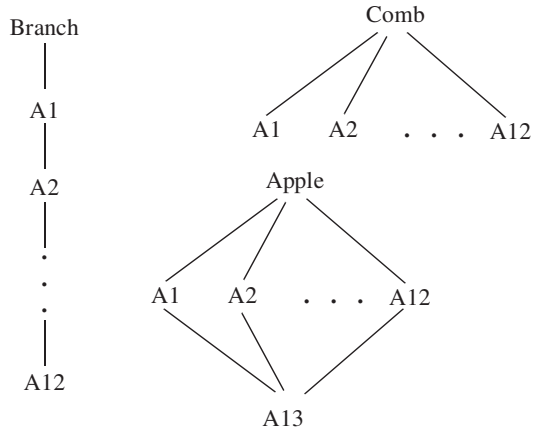
Following features were required for our environment and language:

- multiple inheritance,
- dynamic class modification,
- dynamic variables typing,
- dynamic binding (message - method),
- easy language access, simple concepts,
- metaclass protocolte standardize instance and class notation.

EROS system automatically generates the statistic charts in Latex.

Two releases exist, one with dynamical lookup method dispatch and another with static lookup caches. Two management policies are implemented: coloring by conflict graph and incremental coloring. This allows us to compare the benefits of each systems. We also implemented graph algorithms on EROS to validate our ideas on graph costs.

A real complex system is more relevant than random one. So we choose to copy the Smalltalk-80 class system into EROS our smooth toy language. To simulate multiple inheritance, we would like to copy Eiffel hierarchy into EROS but it stay a future idea. Just by now, we implements some typical inheritance graphs:



the constant cost at execution time and the implementation cost does not look as awfull as with a conflict graph.

We think that incremental coloring is the only solution applicable to big object systems.

6- Conclusions

This study is an investigation on the method dispatch problem. Static caches allow a constant method access cost during execution. Coloring techniques [DIX 89] drastically reduce the cache size. We disparage the attractive idea of colored lookup caches of Dixon and al. in δ-languages because of the conflict graph building and storing cost.

As we have seen in section 3 and 4, the size of the conflict graph and its building cost is expensive. Situations exist where coloring has no interest, examples are the branch, the apple (see appendix A). In these worse cases, time is proportional to the square of the number of selectors. In compiled languages, even if the conflict graph space is temporary, the building and coloring times are prohibitive.

Our incremental algorithm avoid this overhead because no conflict graph is computed. This technique is well-adapted to δ-languages where classes and methods are added step by step. Cache management cost is distributed on the complete life of the system. Incremental coloring is well-adapted to compiled object languages with significant librairies.

Further work is to better estimate these ideas in multiple inheritance where join operations generates more conflits and increases both the number of colors and the size of the conflict graph. Our incremental coloring avoid the second drawback.

7-Bibliography

[AND 91] ANDRE, P.
Application du coloriage à la recherche de méthodes dans un LOO,
Technical report - Equipe PPO.
LIST - Université de Nantes (Juin 1991)

[BER 70] BERGE C.
Graphes et hypergraphes,
DUNOD (1970)

[BLA 83] BLAU R.
Paging on an Object-oriented Personal Computer,
ACM Sigmetrics Conf. on Mesurements and Modelling of Computer Systems (1983)

[BOR 88] BORNING, A.H., INGALLS, D.H.H.
Multiple inheritance in Smalltalk-80,
AAAI National Conference on Artificial Intelligence Pittsburgh (1982)

[COH 81] COHEN, J.
Garbage Collection of Linked Data Structures,
ACM Computing Surveys, Vol 13, No. 3, September (1981)

[COI 89] COINTE, P. & BRIOT, J.P.
ClassTalk: transposition des métaclasses ObjVlisp a Smalltalk-80,
Reconnaissance des Formes et Intelligence Artificielle RFIA '89, Paris (1989)

[DEL 89] DELACOUR, V.
Picolo Expresso,
Putting Scheme to Work.
Bigre 89, p. 30-42 (Juil 89)

[DIX 89] DIXON, R, VAUGHAN, M., SCHWEIZER, P.
A fast method dispatcher for compiled languages with multiple inheritance,
OOPSLA'89 Proceedings p. 211-214 (October 1989)

[HAB 89] HABIB, M. & DUCOURNAU, R.
La multiplicité de l'héritage dans les langages à objets
T.S.I. Vol. 8 numéro 1, p. 41-61 (1989)

[DEU 83] DEUTSCH, P.
Efficient implementation of the Smalltalk-80 system,
The MIT Press, third printing (1986)

[GAB 86] GABRIEL, R.K.
Performance and evaluation of Lisp system,
ACM proceedings of POPL 1983 p. 297-302 (1983)

[JOH 87] JOHNSON, R.
Workshop on compiling and optimizing Object-oriented Programming Languages,
Addendum to the proceedings of OOPSLA'87 p. 57-65 (October 1987)

lectors heuristic of section 3.3 is implicitely used, in the way that we first add classes and methods near the root, and then the inheritance graph grows. Here, we only present some examples. We define the method:

- **g in A** is specific. We have to find a free color in the subclasses of A, that is not in {1, 3, 4, 6}. The color 2 is available, we take it.
- **f in A2** is separate, because it already exists in class F, but not in subclasses of A2. We test if its current color is valid: color 5 is not used in subclasses of A2.
- **f in D** is declared, because it already exists in class DEF. The current color of f is 5. That's ok.
- **e1 in A12** is separate, but color 4 is already used by selector a2 in that class. We try to find a free color in the terminal classes containing the selector e1 (A12, E11), that is not in the set {1, 2, 3, 4, 5, 6}. There are no free color, so we add a new one.
- **e in E11** is separate, color 4 is now available, we take it.
- **g in O** is declared because g is defined in one of the subclasses of O. But color 2 is also used by selector d in the class D. We try to find a free color in the terminal classes containing the selector d (A12, E11, DEF), that is not in the set {1, 2, 3, 4, 5, 6, 7}. There are no free color, so we add a new one.
- **h in E1 and i n A1** are specific and respectively color 4 and 2 are free.

Here is the final result:.

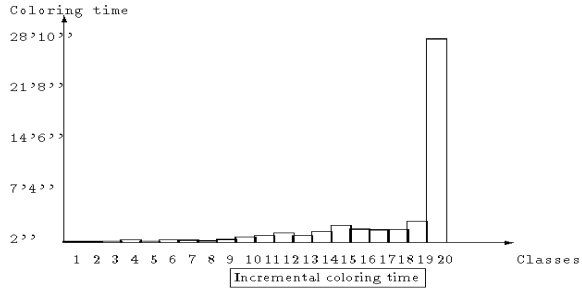
Coloring after

1	o
2	d, i
3	a1, e11, def
4	a2, e, h
5	f
6	a
7	e1

	O	A	A1	A2	A12	D	E1	E11	E	F	F1	DEF
1	o	o	o	o	o	o	o	o	o	o	o	o
2	-	-	i	-	i	d	d	d	-	-	-	d
3	-	-	a1	-	a1	-	-	e1	-	-	-	def
4	-	-	-	a2	a2	-	h	h	e	-	-	e
5	-	-	-	f	f	f	f	f		f	f	f
6	-	a	a	a	a	-	a	a	-	-	-	a
7	-	-	-	-	e1	-	-	e1	-	-	-	-
8	g	g	g	g	g	g	g	g	g	g	g	g

Colored cache

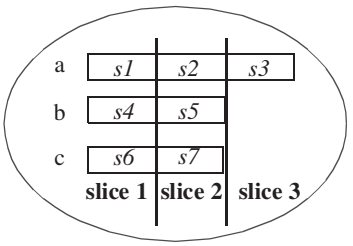
The result does not seem to be satisfying, due to the small frontier and the random method adding. But in practical cases, the coloring table is compressed, as we can see for smalltalk. The following chart presents time mesurements for incremental coloring. Coloring the whole system is now



possible and run about 9 hours. This time contains a Smalltalk-80 graph lookup, adding all classes and methods, and filling the lookup caches. The coloring cost is very good and the number of obtained colors too. In fact, we found an overhead less than 3% of the chromatic number.

A relevant reason of using such a coloring is that coloring cost is **distributed** during all the programming and debugging activities but **not** at execution time. So we preserve

in the next picture. For example, here we can color selectors



Equidomains of which color is 1

of the domains of color 1 in the same way. It gives: *{s1, s4, s6}* $\mapsto$ *1, {s2, s5, s7}* $\mapsto$ *2, {s3}* $\mapsto$ *3*. The number of added colors is then the addition of the slice number by equidomain color. Let nc_E the number of colors of equidomains and nc_S the number of colors of selectors, then $nc_S = \Sigma_{i=1..nbE}$ *(nb_slice(i))*, where *nb-slices(i)* is the size of the biggest equidomain of color i. In this example, a is the biggest equidomain of color 1.

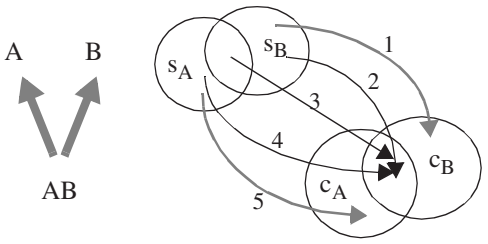
Statistical values proved that trick: for *Object* class, we obtained 829 domains and 1400 final colors with the trick instead of 3000. Remind that the best coloring is about 380. So equi-domains are not interesting enough, even if the decreasing cost was however attractive.

5- Incremental coloring

Previous results and statistics prove that coloring is an attractive idea, but because of the conflict graph cost the profit shut down. We propose here an incremental (step by step) coloring. Its basic idea is to find a correspondance between each inheritance graph modification and coloring disturbance, according to the ideas of section 3.4. This algorithm avoids computing or storing any conflict graph.

We suppose two main operations : add a superclass in the inheritance hierarchy, and define a method in a class.

In single inheritance, add a class does not change the conflicts. But in multiple inheritance, we have to check the validity of colors and recolor some selectors. In the simple case of two inherited classes A and B, the following algo-



rithm is applied:

- $\forall s \in ((s_A \cap s_B) \cup \{s' \in (s_A \Delta s_B) \wedge color(s') \in (c_A \Delta c_B)\})$,
  new_color(s) = color(s), *(arrows 1, 3, 5)*
- $\forall s \in (s_A \Delta s_B) \wedge color(s) \in (c_A \cap c_B)$,
  new_color(s) = find_color(s), *(arrows 2,4)*

where s_I is the set of selectors of class I and c_I is the set of colors used by s_I, and $A\Delta B = (A \cup B) - (A \cap B)$. The function *find_color* is described in the algorithm of appendix C.

Each part of the method definition partition (section 3.4) corresponds to a particular process. For example, if the selector is specific, we look for a free color in the class and its sub hierarchy. If there is no one, we add a new color in the systems. Following array shows the current coloring of the graph n^o 3 of section 3.4.

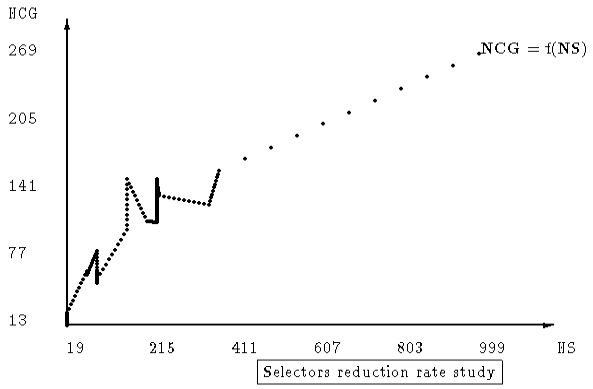
Coloring before

1	o
2	d
3	a1, e11, def
4	a2, e1, e
5	f

	O	A	A1	A2	A12	D	E1	E11	E	F	F1	DEF
1	o	o	o	o	o	o	o	o	o	o	o	o
2	-	-	-	-	-	d	d	d	-	-	-	d
3	-	-	a1	-	a1	-	-	e11	-	-	-	def
4	-	-	-	a2	a2	-	-	e1	e	-	-	e
5	-	-	-	-	-	-	-	-	-	f	f	f
6	-	a	a	a	a	-	a	a	-	-	a	a

Colored cache

The algorithm is described in appendix C. Critical se-



We verify that the minimum condition of section 3.5 is always true except for class *Behavior* and clas *Object* (see appendix B). We compute the real size of the conflict graph and we notice that a memory profit is gained only for class *Object*. For the *Behavior* class, the condition is false, but no benefit is found. Its subgraph contains 371 classes and 995 selectors and at least 271 colors. Statistics show that the simple cache need 738 KB, and the colored one 201 KB and the conflict graph requires1,5 MB.

4.4 - Conclusions on statistics.

Let's sum up the situation for the Smalltalk-80 system: the simple cache require memory space *(4500 indexes * 760 classes) = 6,8* mega-bytes. The colored cache needs *(375 indexes * 760 classes) = 570* kilo-bytes. The conflict graph need 3 mega-bytes. Remark that the Smalltalk-80 image takes out 2 mega-bytes. The winning is real but not significant. However that is surely lower for multiple inheritance (see appendix B).

Looking at these measures, it seems obvious that:

- builting time is prohibitive,
- conflict graph space is too expensive (we often loose more memory than we gain),
- coloring time is acceptable.

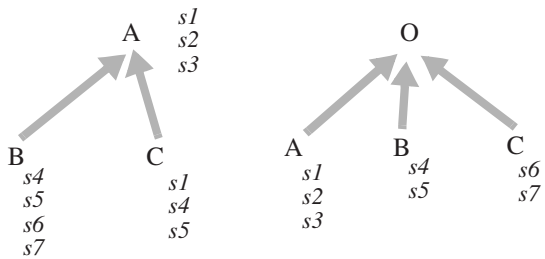
It is not reasonnable to keep such a conflict graph in memory or to compute it again. So we either reduce its size (section 4.5) or color without any conflict graph (section 5).

4.5 - Experimentation with equi-domains

Due to structural properties of inheritance graphs, we observe that methods are rather defined by group than single. A visibility domain of a selector is a set of classes where this selector is visible. If two selectors have the same visibility domain, they have exactly the same selector conflicts. Selectors may be assembled in **equi-domains**, equivalent classes of visibility domains. This leads to a conflict graph simplification.

For example, in Smalltalk-80 *Behavior* class and subclasses, 990 selectors catalog into 216 equi-domains. Equi-domain graph construction (217 nodes) and coloring time (798 colors) takes about 15 min. That is not so bad in comparison with 45 min. Note that about 370 conflicts by selector are counted (see appendix B), the maximum is 989 conflicts for the selector "*includesSelector:*".

Once the equi-domain conflict graph is colored, we color selectors. For example, let's study the inheritance graphs below. They have the same equi-domains. This illustrates the

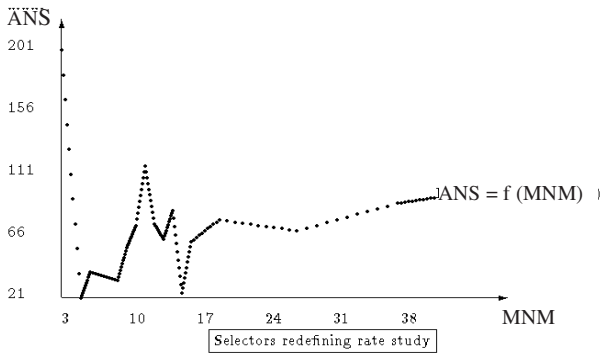


Graph ABC *Graph OABC*

fact that domains are not dependent from the graph structure. The second graph is more simple to understand from an equi-domain point of view. Three equi-domains are computed: *a = {s1, s2, s3}, b = {s4, s5}, c = {s6, s7}*. Equi-domains coloring is *{a, b, c}* $\mapsto$ *1*, because s1, s4, s6 are not dependent. Now we want to assign colors to the selectors. The first selector receive the color of the equi-domain, the others get a new color. Such a split provides: *{s1, s4, s6}* $\mapsto$ *1*; *{s2}* $\mapsto$ *2, {s3}* $\mapsto$ *3, {s5}* $\mapsto$ *4, {s7}* $\mapsto$ *5*. Five colors are needed.

We suggest a trick: split the same colored equi-domains in the same time, that means slicing those domains as shown

of methods are at least redefined once. In the whole system



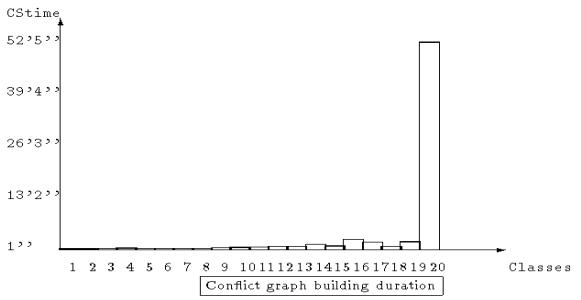
initialize selector is redefined 142 times.

4.2 - Coloring algorithms

We separate graph building and coloring to measure the weight of each process.

4.2.1 - Building the conflict graph

Building the conflict graph is very expensive in time. It takes more than one hour to build the conflict graph for the *Behavior* [5]class and its subclasses. The following chart indicates the time of building the conflict graph for some Smalltalk-80 subsystems (i.e. a class and its subclasses). Classes



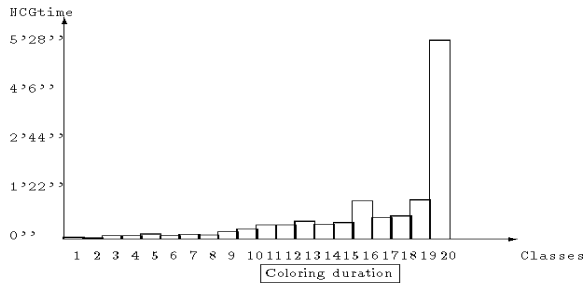
are ordered according to the number of selectors. Number 1 corresponds to *VariableDefinition* class and 20 is *Behavior* class (see appendix B). Terminal classes are examined and their behavior is computed by a dynamic lookup. This chart allows us to roughly approximate the weight of selector number in building time (proportional to ns^2).

Object whole system does not appear in statistics, because we had a lack of memory. Using the trick about the root Object (see section 3.3), we forget about 106 selectors, and graph building run nearly 1 hour 20 minutes.

From a memory point of view, we counted about 270 conflicts by selector for Smalltalk-80 system. We deduced a graph size about 2,4 mega-word (4,8 MBytes) with a successors list representation and 20 megaword (4500 * 4500) for a matrix one. Let's however remark that subtract the 106 Object methods destroy 435 800 conflicts. The remainder cost fall downto 3 MB. So we must find 3 mega-bytes during coloring step.

4.2.2 - Coloring the conflict graph

To color the whole Smalltalk-80 system, once the conflict graph is built, takes about 12 minutes. Few memory is engaged for that algorithm. The next figure shows the coloring time chart . Note that the scale is different (x 10) than



the previous one.

4.3 - Coloring issues

We get 271 colors, which are added to the 106 one of Object. The complete system improvement may turn about 377 * 750 = 282 KB. As said in [DIX 89], we verify that the number of colors is close to ***M*** the size of the biggest complete behavior.

The following chart shows the squeezing rate for inheritance subgraphes. On some *Object* subgraphs, we find 40 to 80 % reduction rates. For the whole system we obtained, with our optimized conflict algorithm about 375 colors, that is a 92 % reduction rate (also called squeezing rate).

5. *Behavior* class is an *Object* direct subclass describing classes minimal behavior. It is the biggest subclass of object

are specific, selector *a* is redefined in class **A2**, separate in class **E1** and declared in class **A**. Sorting selector definition help us to understand their influence on coloring. For example, **redefined selectors do not disturb the coloring**.

3.5 - Computing the conflict graph

The conflict graph of any inheritance graph is the same as the simplified graph of section 3.3, where *ns* = size($\cup_{i=1..k}(f_i)$) is the number of selectors.

To build the conflict graph, we saw that it is sufficient to compute the behavior of the frontier classes, the conflict graph is the union of all these behaviors (clicks). Here is a complexity estimation of building time: $O(ns^2)$. Using caches to compute the complete behavior is an algorithm optimization. Caches avoid a costly lookup, because the behavior at the frontier is already computed.

Two ways are normally used to store a graph. If there are few edges, we use lists of sucessors otherwise, we handle an array (node * node) of booleans. It is difficult to precisely estimate the size of the conflict graph. We have the following relations:

- number of colors ≥ M = max(f_i)
- size of the simple cache= n * ns
- size of colored cache ≥ M * n
- size of conflict graph ≥ M^2

We can give a necessary condition which describes the best situation. If **$n < M^2/(ns - M)$** we can affirm that coloring do not spare any space. On the opposite side, a profit may exist. If ns ≈ M (like in branch or apple of appendix B) then n should be so greater that it is not possible to obtain profit. Differently, the comb graph is more attractive (see appendix B). The best benefit is gained for behaviors of the same size with empty intersections.

4- Experimental results

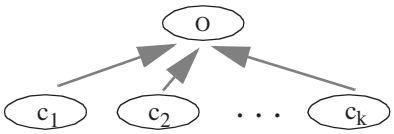
In order to evaluate coloring costs both in time units and in necessary memory space, some statistical mesurements have been computed. Representative examples were designed in EROS (see appendix A and B). Parts of a Smalltalk-80 system version 2.5 have also been tested on Sun Sparc Workstation 3/80 with 12 MByte RAM [AND 91]. The choice of Smalltalk-80 language is based on the system access facility and, the good inheritance graph structuration and the property of section 3.2. We already saw in section 3.2 that when you handle only the frontier, no difference exists between single or multiple inheritance. Carefully note that statistics are difficult to establish because:

- such systems always change (new classes and new methods), and
- the garbage collector [GAB 86], disturbs our measures.

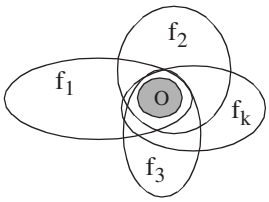
4.1 - Statistical facts

In the Smalltalk-80 system, we find 766 classes and 503 are terminal. Each class has approximatively four subclasses. Of course, Object owns the maximum number of subclasses. Ten percent of Object's direct subclasses are terminal. The average inheritance height is 5 and the highest one is 13 classes. In single inheritance (tree), the number of superclasses is equal to the inheritance height. The average number of defined methods by class is 9 and class *TextCompositor* has the maximum number: 112. About 230 methods are visible in each class, the biggest complete behavior belongs to class *MacTtyAccessor*, its size is 371. Some 7500 methods and 4500 selectors are counted. The graph below presents the variation of the average number of selectors *(ANS)* by the average number of methods *(MNM)*. These average numbers are computed on subsystems of Object inheritance graph (*Integer, Collection, View, Controller, Behavior...*). It can be seen that approximatively 30 to 40 %

inheritance graph with only the root and the frontier, where



simplified inheritance graph



behaviors relations

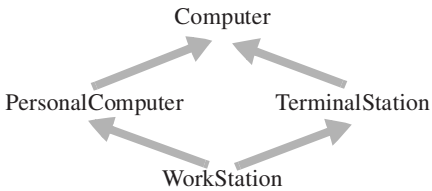
c_i are the frontier classes and f_i are their complete behavior. The more a selector belongs to behaviors, the more critical it is. The root selectors are good examples of very critical selectors. Intuitively, selectors defined in classes near the root are more critical than the one defined near the frontier. A coloring algorithm should use that heuristic.

3.4 - Graph structure and method repartition

Intuitively add a class or a method are the only things that disturb really the current coloring. We separate this two cases in first the **graph structure modification** and second, the **behavioral modification**.

3.4.1 - graph structure modification

We distinguish two structure modifications in the inheritance graph: split and join.



Graph n° 2: apple graph

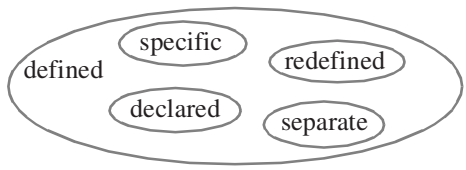
Split divides descending classes of a root class (*Computer*) in several subclasses (*PersonalComputer, TerminalStation*). Each subclass has a common part inherited from the root class and a private one defined in the subclass it self. The more you have split operations, the more the number of edges of the conflict graph is reduced.

Join groups all superclasses of a class in one group; the resulting conflict graph is a bigger click. The complete behavior of class *Workstation* is the union of the complete behaviors of *PersonalComputer* and *TerminalStation*. Hence the click build on *Workstation* is surely bigger than the ones of the superclasses.

Remark that, more the difference between the split number and the join number is great, more coloring is interesting. We can also note that every new class will belong to the frontier, and its superclasses may be removed from the frontier (if they were in there).

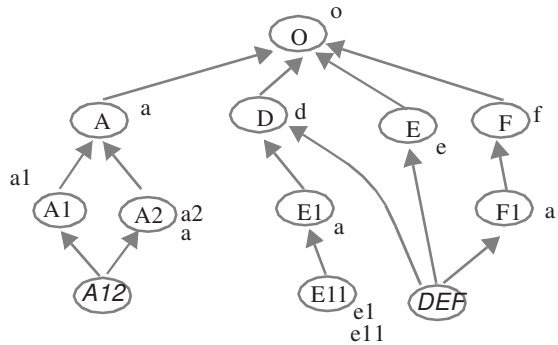
3.4.2 - behavioral modification

To define a selector is to associate a procedure to a selector . We classify definitions into a partition. A defined selector in a given class C_0 is:



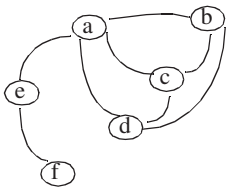
- **specific** if it is not yet defined in the system,
- **separate** if it does not belong to the behavior of one of the superclasses or one of the subclasses of C_0.
- **redefined** if it belongs to one of the superclasses behavior of C_0.
- **declared** if it does not belong to the behavior of one of the superclasses of C_0.

In the following inheritance graph, selectors *e1* and *d*



Graph N° 3

flicts:

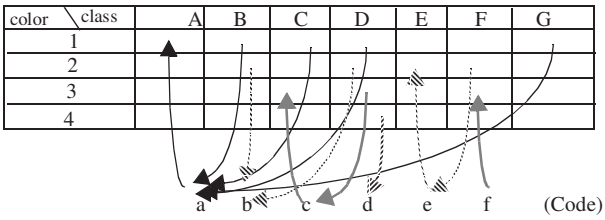


conflict graph

1	a
2	b e
3	c f
4	d

color table

Note that the subgraph *{a, b, c, d}* is a click (complete behavior of class D). Minimal coloring needs four colors. Instead of having six entries tables (one color per selector),



method cache

we get four indexes (some colors refer more than one selector), as illustrated in the previous figure. This array contains less holes than the previous one of the section 2.1.

3- Discussion

Method search is really a critical point due to the frequency of its use. Cache techniques avoid expensive dynamic lookup: we obtain a constant cost for method search. But a memory overhead is generated. The coloring step may reduce this overhead.

In this section, we analyse coloring in object systems. We propose some improvements and we point out some relevant problems in case of δ-languages.

Let ***G*** an inheritance graph with ***n*** classes. We note ***ns*** the number of selectors.

3.1 - Coloring algorithms

To find a good coloring, we adapt algorithms found in the graph theory [BER 70]. Some of them provided the best coloring for any kind of graphs. But their exponential complexity make them inoperable in voluminous graphs. Fortunately greedy algorithms exist and permit to extract a good coloring in a polynomial time *(ns^2)* in the worst case. A trick is given by Dixon and al, they suggest to order the node evaluation by their degree (number of edges). This corresponds to our notion of critical selectors (see section 3.3).

3.2 - Frontier and minimal number of colors

Let the **frontier** of the graph be the set of terminal classes that have no subclasses. [AND 91] shows that examining classes of the inheritance graph frontier is sufficient to compute all conflicts. In the example of section 2.1, the border classes are classes *D, F, G*. This reduces tremendously the conflict calculus time. Another advantage of study only the border is that the difference between single and multiple inheritance disappears.

The maximum complete behavior size [3]is an undervalue of the conflict graph maximum click (*Mclick*) [DIX 89] and then transitively underestimate the chromatic number [4][AND 91]. Let M be the size of the biggest complete behavior, then $M \leq Mclick \leq chromatic\ number \leq N \leq ns$. In our example, class *D* detains the biggest number of selectors *{a, b, c, d}*. Our experience shows that those three indicators have generally the same value (see appendix B).

3.3 - Critical selectors

Most inheritance graphs contain only one root class. For example, Smalltalk-80 root is *Object*. The root selectors are conflicting with every selectors of the system. This is the minimum behavior. There is a noticeable benefit both in time and in space if we subtract those selectors from the conflict calculus. For example, let a system with 3100 selectors and a minimum behavior of 100 selectors, the root conflicts produces 600000 useless edges!

Critical selectors are selectors that have a lot of adjacent nodes in the conflict graph. This picture presents a simplified

3. The size is the number of visible selectors.

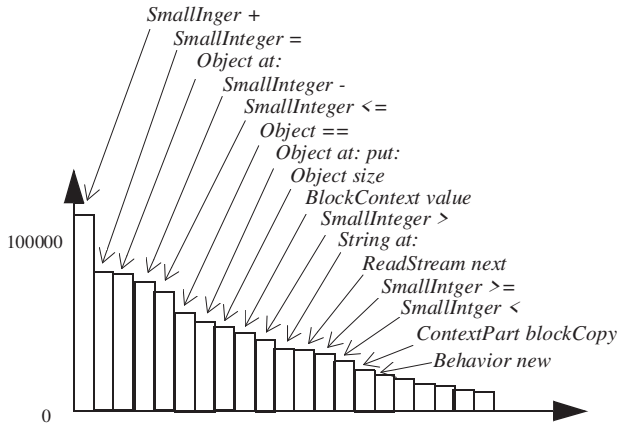
4. The chromatic number is the minimal number of colors needed to color each nodes of the graph. The coloring constraint is: all neighbours of a node must have a different color.

duce the entries number *N*. Then it reduces the hole rate. This technique is discussed in section 2.3.

Note that root selectors are in conflict with all defined selectors of each class. So it exists selectors knowing all classes. Then **the class dimension can not be compressed**.

2.2 - Other optimizations

Other solutions have been proposed to speed up method search including [SUZ 83, DEU 83, DEL 89, KIC 90, UNG 91]. Features of each langage interfer a lot with technical choices carried out. In statistical benchmarks, it was reported that some messages are sent more often than others. The chart below inspired from Smalltalk-80 roughly shows this idea. We put methods corresponding to these messages in a



special area, designed for it. At execution time, a simple class check ensures that we deal with the correct method. If it succeeds, the method body is executed. Otherwise, the classical method search is driven. Caches are more interesting than these optimizations because they do not depend on a special domain of application like software engineering, concurrency or artificial intelligence.

Other authors submit the idea of typing some expression to improve the lookup or bind statically message send and method apply. However, in polymorphic systems, this is not possible to statically solve the method call. Furthermore, this limits ease of use and the expressiveness power of the language. "*Typing expressions in polymorphic systems with overloading is a tricky problem*" ([HOL 91]). "*A polymorphism with no constraints would be incompatible with the notion of typing*" ([MEY 88]).

2.3 - Coloring selectors

The major drawback of caches is the required memory size. Dixon and al. [DIX 89] propose an algorithm to limit caches size by reducing the selectors index range.

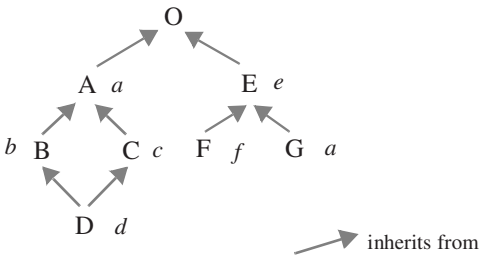
Let's define the **complete behavior** of an object as the set of all the **visible** selectors, i.e. selectors it can use (either defined or inherited). We assume that the search strategy fills correctly the caches, coping with conflict exceptions and so on. We use the natural hypothesis that there is no inheritance exception, i.e. all the selectors of the superclasses are inherited in the subclass[2]. Let's define the following many-to-one function *color*: each selector has a value in the interval of indexes (*Selector* $\rightarrow$ *[1..N]*). We want the smallest index range according to the constraint "**if two selectors belong to the same complete behavior,** they are said **conflicting** selectors, and **they must have a distinct index**". Let's illustrate this idea by a simple example. Let's have a look to the inheritance graph n^o 1 that we saw in section 2.1. Selectors *f* and *d* are independent, because it does not exist any object that may know both those two selectors. But *a* and *d* are conflicting because, instances of *D* class can receive messages with *a* or *d* selector.

To establish all conflicts between selectors, we build a **conflict graph**. Nodes are selectors, edges represents the conflicts between selectors. A **click** is a graph where all the nodes are linked **[deux a deux]** . For each class, the conflict graph associated to its complete behavior is a click. The conflict graph is the union of all the clicks. Choosing indexes is equivalent to coloring nodes in a graph. For this reason, indexes are called **colors** in the following discussion.

The following graph represents all the selector con-

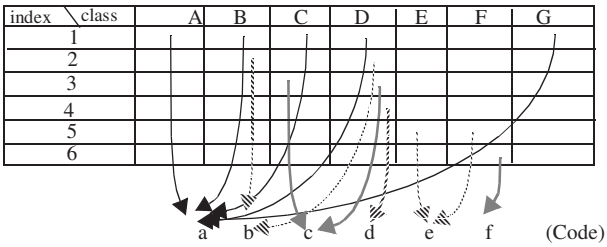
2. An exception may be viewed as a special method.

from [DIX 89]:



inheritance graph n° 1

Six selectors exist {a, b, c, d, e, f} in the graph. A simple indexing is {a ↦ 1, b ↦ 2, c ↦ 3, d ↦ 4, e ↦ 5, f ↦ 6}.

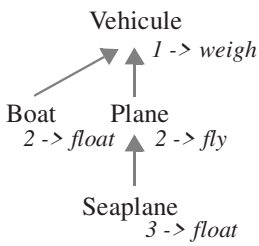


method cache

Classes are not indexed in that figure to facilitate the reading. Indexing is a mathematical function from the set of names into an interval *[1.. N]*. For example, selector *a* get the index *1*. The selector *f* is defined only in the class *F*, so line 6 has too many holes. Cache optimizations should reduce the hole rate.

2.1.4 - Optimizing cache size

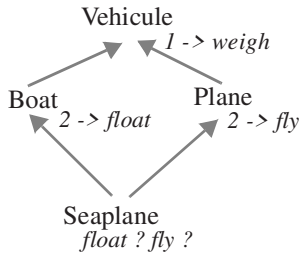
In single inheritance, attributes (variables and methods) dictionaries may be built by a concatenation of the class dictionary and the dictionaries of its ancestors. This is generally implemented for instance variables in most of single inheritance languages like Smalltalk-80 . It also exists for methods, see Picolo [DEL 89] for example. In the example below, the



simple inheritance

index *2* is given to *fly* and *fly* has the same index in *Seaplane* class. For one class and its subclasses, an index always corresponds to the same selector. This local technique is really simple and builds the smallest caches as possible (without hole). However the selector is not sufficient to find the index, you need the class. Hence type checking is required.

In multiple inheritance, the problem appears more difficult to solve. Simple concatenation does not act correctly. In our example, the index *2* is given to *fly* and *float*. There is a



multiple inheritance

problem in *Seaplane*, because the two selectors have the same index. This situation is named **conflict** of selectors. Multiple inheritance adds new conflicts between selectors.

Selector conflict notion is apart from method conflict one. The conflict concept discussed in [HAB 89] or [ROY 91] concerns methods conflicts. Two methods are conflicting in one class, if they have the same selector and they are inherited from distinct classes. Solving method conflicts depends on other criterion like the ancestors lookup strategy (depth-first, breadthfirst, topologic order...); for more explanation see [HAB 89, ROY 91].

A non optimized indexing function could be a one-to-one function and *N* would be the number of selectors. To reduce the table size, we want *N* as small as possible. The problem is then to compute a many-to-one function. With such a global indexing, the selector is sufficient to compute the index. Contrary to the previous technique for simple inheritance, the substitution of the selectors by the indexes in the code is easier.

A coloring algorithm inspired from graph theory can re-

2) **static** caches are entirely filled before execution. They are rather employed in compiled languages. They run faster than dynamical one: no lookup at all. The cost of a method call is constant in time. It rounds 2 or 3 functional call [DEL 89, DIX 89]. But they consume more memory space and need more time before the execution.

2.1.2 - Examples

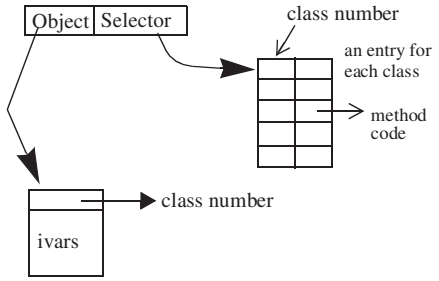
Most of Smalltalk-80 implementations use a lookup cache based on a locality principle. This cache is an array indexed by all the selectors of the system. During each message send, the handler accesses the selector entry, checks the receiver object class, and compares it it with the array's one. If it fails, the classical lookup is retained and fills the cache again. A significant profit (20-30%) is obtained for a small cost. According to Johnson [JOH 87], a cache hit occurs 95% of the time and takes only three or four instructions when the correct procedure is found. The inherent overhead takes nearly 10 % more memory. Deutsch [DEU 83] uses several distinct caches (dynamic, in-line). Then, in the best case, 90% of the time, procedure invocation requires only one comparison operation in addition to a conventional procedure linkage.

In-line caches and polymorphic in-line caches are discussed in [HOL 91]. The address of the effective method is cached in the code after the first call. But due to polymorphism, this is not always the right method, and dynamic search is required. Polymorphic in-line caches solve this problem but generate a overhead in space (not only one address, but a function) and in time (execute the function). Both techniques lead to a variable method access cost.

2.1.3 - Static lookup cache organisation

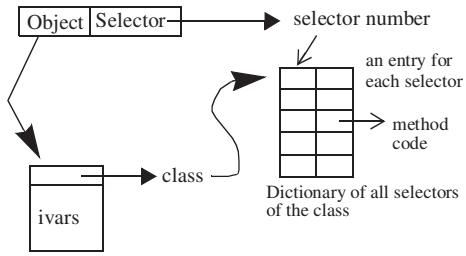
Caches contain method body (procedure) addresses. Three organisations are possible:

- each selector owns a hash table, describing for every class in the system, the procedure address. Classes may have distinct associated number, providing faster hash function (integer vs string). This organisation is

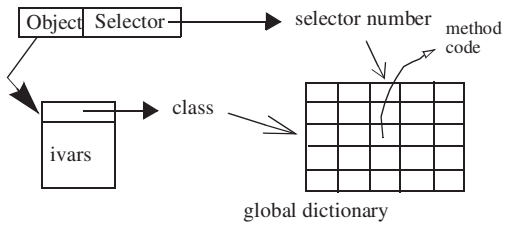


common in languages like CLOS or Flavors, handling complex method construction (method combination, generic functions...).

- the dual viewpoint consists in classifying selectors by class. Each class has a table of all its methods. Each selector gets an identification number, which is the index of the selector in the cache.



- the third organisation is a large table, which cumulates all those informations in a single area. This facilitates dynamical modifications of the caches. As quoted in



[BLA 83] this organisation decreases the default page rate. From now, caches will be a global table.

In order to speed up cache access, we want to replace names of classes or selectors by integer values. These values are indexes in the cache. Everywhere they appear (objects, code), a compiler changes the names by their corresponding indexes. Here is an example of inheritance graph, extracted

programmer wishes to observe quickly the effects of local little modifications, without lengthy compilations.

We show in this paper that Dixon and al's algorithm does not actually fit to δ-languages. We propose a simple incremental algorithm in order to color selectors with good performances in number of colors, and in time and without space overhead.

This paper is organized as follow: first, we have a look at method search optimization in object-oriented languages. The section 2 presents the major improvement techniques and studies the cache use. It describes a lookup cache method dispatch (to speed up the accesses) and a coloring algorithm (to reduce the cache size). Section 3 discusses the coloring interest in δ-languages. Section 4 establishes experimental results in the Smalltalk-80[1] system. These statistics verify and confirm the theoretical facts of section 3. Section 5 describes our incremental coloring and its results on the Smalltalk-80 system.

2- Background

Generally speaking, the object programming style does not lead to efficiency. It encourages to write numerous and short encapsulated modules. The number of method call is also very important, because message passing is the basic mechanism of communication between objects. Objects proliferation and high communication rate induce an expensive overhead in time and in space.

Benchmarks and performance mesurements [BLA 83, DEL 89, DEU 83, JOH 87, KIC 90, ROY 90, SUZ 83, UNG 91] show some efficiency bottlenecks:

- object management: object creation cost, object access cost, object memory recovery.
- attributes access: values of instance variables and methods of behavior.
- context allocation and deallocation.
- kind of code (compiled, virtual or native).

Note that bottlenecks depend on the features of each language (typing, binding, inheritance, compilation). From now we focus on the behavior access bottleneck.

During message passing, two informations are needed: the receiver class and the selector of the method to apply. A simple algorithm for method search is:

- check the selector in the method dictionary of the receiver class.
- if this check fails, continue the search in the superclasses.
- if all superclasses are visited without success, the search definitely fails.

Hence the dynamic binding cost is proportional to the number of visited classes. It is very expensive, because it occurs every communication between objects and often recursively. The most current optimization is caching the code addresses either in the code itself at the send point (in-line cache) or in a direct table (method cache also called lookup cache).

2.1 - Improving method dispatch by cache techniques

Caches are functions with two parameters : a class and a selector. They deliver method bodies (or addresses).

$$cache : Class \times Selector \rightarrow Procedure$$

2.1.1 - Cache filling

1) **dynamic** caches require memory only for called method. During a method call, the search begins through the cache. If the selector has no entry, a dynamic lookup provides the method body address. This address is then stored in the cache. Cache filling is then randomly distributed on execution. That overhead may not be reasonable for real-time systems. However, saved memory is voluminous in a large system where execution deals with few selectors.

1. Smalltalk-80 is a trademark of Parc Place Systems.

Optimizing Method Search with Lookup Caches and Incremental Coloring

André Pascal and Royer Jean-Claude
Equipe de programmation par objets
Faculté des Sciences et des Techniques
2, rue de la Houssinière
44072 Nantes Cedex 03
(andre, royer)@unantes.univ-nantes.fr
Telephone: (33) 40.37.30.22

Keywords

Object-oriented languages with classes, Smalltalk-80, statistics, efficiency, methods search, coloring, lookup caches.

Abstract

This paper describes an investigation on improving method search efficiency in object-oriented languages with classes. Static caches allow a constant cost dispatch mechanism, but they consume too much memory. A coloring technique can reduce the cache size. This technique can be used during the compilation of strongly typed languages, like C++ or Eiffel. However it is not immediately available to languages like CLOS or Smalltalk. This paper discusses the coloration of static caches in dynamically typed languages. We suggest an efficient incremental coloring technique to support it.

1- Introduction

Object-oriented programming is a step in the evolution of programming techniques. It possesses very attractive concepts like polymorphism, encapsulation, message passing, inheritance, instanciation, and dynamic binding. It provides some interesting solutions to software quality problems like reusability, extensibility, and modularity [MEY 88]. Execution efficiency is one of the basic criterion of software engineering. By definition, it consists in a good use of hardware resources like processor, memory, communications, etc. Moon [MOO 86] distinguishes four aspects: programmer's time, CPU time, page fault rate and virtual memory occupied. These features correspond to a weak point of object-oriented languages implementation, especially if sophisticated graphical interfaces are used. This is a classical reason for users to depreciate systems like Smalltalk-80 or CLOS. On the contrary Eiffel or C++, with static typing, compilation and C code translation provide faster executions.

To optimize drastically method search, we can use static lookup caches. Such a method dispatcher leads us to a constant access cost, independant from the graph structure and the method distribution. However this technique uses too much memory. To reduce this cost, a technique of selector coloring [DIX 89] minimizes caches size. This improvement is already implemented in compiled languages, but the coloring time and the size of the conflict graph are often prohibitive. Many situations lead to a great overhead (coloring time) without significant benefits (memory space).

In this paper, we are interested in lookup caches in languages allowing dynamical class modifications. By this, we mean object-oriented languages:

- with dynamic typing (like Smalltalk-80 or CLOS),
- with dynamic binding (realize method search at execution time),
- with partial compilation (optimizing method search...).

We call them **δ-languages**. Such languages are very attractive from the programmer's point of view. Indeed, the