プログラミング－言語・基礎・実践－　1－5
（1991. 4. 26）

Cベースのオブジェクト指向言語における再コンパイルの短縮

安田 和　小野寺 民也　北山 文彦　久世 和資　上村 務
日本アイ・ビー・エム（株）東京基礎研究所

Abstract

現在、C＋＋をはじめ、静的な型をもつコンパイル方式のオブジェクト指向言語において、クラスの仕様変更にともなって引き起こされる長時間の再コンパイルが、プログラム開発効率上の大きな問題となっている。我々の研究・開発しているCベースのオブジェクト指向言語COBでは、高いモジュラリティをもつ言語仕様を late binding という手法によって実現し、さらに smart compilation と呼ばれる機構を導入することによって、再コンパイルされるファイルの数を大幅に減らすことに成功した。本論文は、それぞれの手法について、その機構およびコンパイル時・実行時の効率を紹介する。

Reducing Recompilation for C-based Objects

K. Yasuda, T. Onodera, F.Kitayama, K.Kuse and T.Kamimura
IBM Research, Tokyo Research Laboratory,
5-19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan

Abstract

Static binding is a major contributing factor for achieving good run time performance in languages such as C++. It becomes, however, a cause of large overhead for recompilation under program modifications. To overcome the problem of recompilation, we designed our own C-based object oriented language COB with high modularity, and implemented it by combining *late binding* and a mechanism called *smart compilation* to control the recompilation process. This paper describes these methods with performance evaluation.

1　はじめに

近年のオブジェクト指向プログラミングの一般への普及について、Cベースのオブジェクト指向言語、中でもC++の果たした役割は大きい。C++が成功した大きな要因に、実行効率の重視という点があげられる。C++においては、クラスは型として取り扱われ、そのデータ構造はプライベートなメンバーに関する点までコンパイル時点で静的に決まっている。このことがC++の実行効率のよいコード生成を可能にしている。

しかし一方で、この方式はプログラムの開発効率を低下させる。つまり、C++のような静的データ結合を実現するためには、クラスのインターフェースにプライベートなメンバーまで書かねばならない。その結果、その些細な変更が大量の再コンパイルを引き起こすということが、実際に大きな問題となっている。この問題は、多くのアプリケーションから利用されているクラス・ライブラリに修正があった場合など、特に影響が甚大である。

我々はこういった点を念頭において、完成したプログラムの実行効率を落とすことなくプログラムの開発効率を向上させることを目的とする独自のCベースのオブジェクト指向言語COBを開発した。

COBの言語仕様決定にあたっては、プログラム開発時における再コンパイルを少なくすることを重視し、ファイル間の依存関係をできるだけ少なくするため、クラスのモジュラリティを高めた。具体的には、クラスのインターフェースとインプリメンテーションを完全に分離し、プライベート・メンバーをインターフェースから隠蔽することによって、その変更による影響がクライアントに及ばないようにした。本論文の第2節では、この点を含めCOBの言語上の特徴を本論文の内容に関係する点に重点をおいて紹介する。さらに第3節で、COBの言語仕様を実現する際に使用した2通りのlate bindingの手法を説明し、それぞれのコンパイル効率および生成コードの実行性能を比較する。

また、実際のプログラム開発過程では、クラス・インターフェースのパブリックな部分に変更があった場合の再コンパイル時間も無視できないものであるため、我々はそれを短縮する目的でスマート・コンパイルsmart compilationと呼ばれる機構をソフトウェア・ツールとして実現した。これについては第4節で紹介し、最後にこれらの手法を実際のプログラム開発過程に適用した際の効果を紹介して論文のまとめとする。

2　COB

COBは、C言語の拡張であること、強力な型をもち、クラスが型として扱われることなど、C++と多くの類似点を持っている。インスタンス変数およびインスタンス関数へのアクセスやインスタンスの生成についても、C++と類似した方法をとっている。

一方でC++との大きな相違のひとつに、クラス・インターフェースとクラス・インプリメンテーションの完全な分離という点

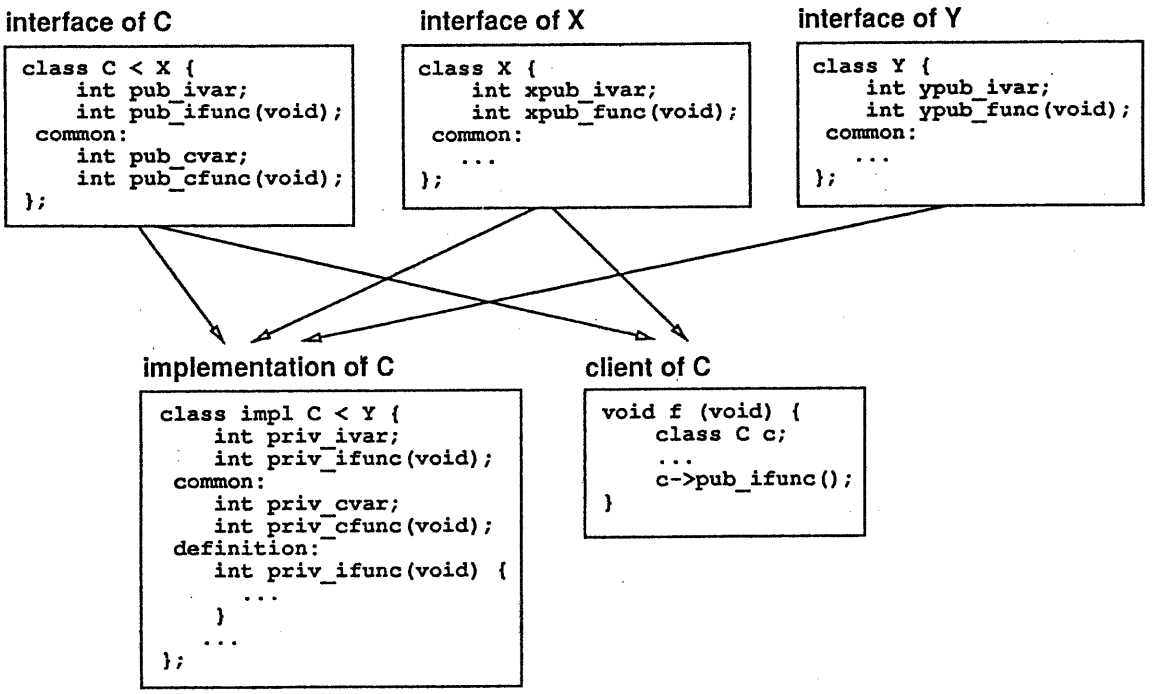

Figure 1: Example of COB Language

が挙げられる。クラスのインターフェースにはパブリックな情報だけが書かれ、プライベートに関する情報は一切現れない。また、インスタンス関数はすべて再定義可能(virtual)である。図1にＣＯＢのプログラムの例を示す。ＣＯＢでは < は継承を表し、common:はＣ＋＋の用語でいうstaticなメンバーを示す。

この例で、クラスＣはパブリック・スーパークラスとしてクラスＸをもちプライベート・スーパークラスとしてクラスＹをもつ。クラスＣのクライアントは、クラスＣおよびＸのインターフェースに依存しているが、これらのクラスのプライベートな変更からは何ら影響を受けない。したがってクラスＹのインターフェースにも依存しないことになる。

その他の言語の特徴として、ＣＯＢコンパイラは、未知のクラスに出会うと、そのクラスのインターフェースを自動的に探して読み込むようになっている。つまりクラスのインターフェースに関する限り#includeは不要である。このことは、不必要なクラスのインターフェースを#includeしてしまう心配をなくし、間接的ながらもファイル間の依存関係を減らすことに貢献している。

3 Late Binding

ＣＯＢは現在Ｃコンパイラへのトランスレータとして実現されている。ＣＯＢは分割コンパイルをサポートしており、コンパイルの最小単位はクラスである。

クラスおよびそのインスタンスの実現は標準的な枠組みでなされている。まず、各クラスに対し１個のメソッド・テーブルが用意される。これは、このクラスのすべてのインスタンス関数へのポインタの配列である。また、それぞれのクラスのインスタンスは、自分自身およびそのすべてのスーパークラスで定義されたインスタンス変数を保有する１個のレコードであり、その先頭には当該クラスのメソッド・テーブルを指すポインタが置かれている。

こういった枠組みにおいて、あるクラスを使用するコードを生成するためには、そのクラスのインスタンスおよびメソッド・テーブルの構造、すなわちそのサイズ、各メンバーのオフセットについて十分な情報が必要である。しかしながら、ＣＯＢにおいては、クラスのプライベートな構成要素（メンバー、スーパークラス）はインターフェースから隠されているため、これらはいずれもクライアントのコンパイル時には決定できない。ちなみにＣ＋＋ではこれらの値はすべてコンパイル時に定数となっている。

この問題を解決するために、ＣＯＢコンパイラは、これらメソッド・テーブルやインスタンスのサイズ、メンバーのオフセットを参照するのに、一群の変数（アクセス変数）を用いてコードを生成するようにした。これらの変数は実際に参照されるまでに初期化しなければならないが、そのためには各クラスのインプリメンテーションに散らばっているクラスのプライベートな情報を集めてくる必要がある。

我々は、このアクセス変数の値を決定する時期について２種類の手法を実現し、その効率を比較した。ひとつは、リンクの前に特殊なフェーズをとりいれ、その段階で値を決定する方式で、プレリンキング方式 prelinking と呼んでいる。もうひとつは、余分なフェーズは用いず、実行時にすべての値を決定する方式で、実行時決定方式 run-time determination と呼ぶ。

3.1 プレリンキング方式

この方式では、ＣＯＢコンパイラはクラスのインプリメンテーションをコンパイルする際、同時にそのクラスに関する各種情報を、サマリー・ファイルと呼ばれるファイルに出力する。このファイルには、アクセス変数の値を決定するのに必要な情報、すなわち、そのクラスのプライベート・パブリック両方のスーパークラス名、そのクラスのインプリメンテーションの中で定義されているメンバーなどの情報が含まれている。

その後プレリンキング・フェーズにおいて、プレリンカーと呼ばれるツールが、すべてのクラスのサマリー・ファイルを読み込み、その情報をもとに全クラスの構造を決定し、その結果をもとに特殊な役割をもつＣプログラム群を出力する。このＣプロ

program	#line	#class	#inheritance	description
maze	843	7	2	create/solve a maze
lisp	1128	45	41	a lisp interpreter

Table 1: Program characteristics

グラム群には、各クラスのメソッド・テーブルの静的な宣言、アクセス変数の宣言、それらの初期化ルーチン、インスタンス生成ルーチンなどが含まれている。初期化ルーチンは、プログラム実行の最初に呼ばれて実行される。

プレリンキング・フェーズの次の段階でこのCプログラム群がコンパイルされ、リンク・フェーズで他のコードとリンクされる。

プレリンキングによるデータ決定方式では、あるクラスのインプリメンテーション・ファイルが修正された場合には、そのファイル自身の再コンパイル、およびプレリンキングとリンクが必要となる。その際、プレリンカーは、そのクラスとそのクラスの全サブクラスについてのみ、出力Cプログラムを更新する。

3.2 実行時決定方式

この手法では、情報の結合はリンカに任され、実際にアクセス変数の値などデータを決定するのはすべて実行時である。COBコンパイラはプレリンカーの出力に対応するコードを、クラス・インプリメンテーションのコンパイルの際に生成する。このコードは、値の決定を「インクリメンタル」に行なう。すなわち、あるクラスの初期化あるいはインスタンス生成用のコードは、その直接のスーパークラスの同様の仕事を行なうルーチンを呼びだし、その結果を利用して自分自身に関する初期化を行なう。

この手法では、あるファイルが修正された場合に必要とされるのは、そのファイルの再コンパイルおよびリンクだけである。その代わり、プレリンキングを行なう場合にくらべ、クラスの初期化やインスタンスの生成にともなう関数呼び出しが増えるため、実行効率が低下する。

3.3 結果

ここで、以上2種類のlate binding方式について、コンパイル効率および生成コードの実行効率を測定した実験データを紹介する。比較のため、C++と同様にプライベートな情報をクラス・インターフェースに書き、定数によってコードを出力した場合(static binding)の効率も、COBコンパイラでシミュレーションを行なうことにより測定した。サンプルとして表1に示す2種類のプログラムを使用した。

program	modified class	Compilation time(sec.)			
		static binding	prelinking		run-time determination
			total(prelinking)		
maze	*all*	98.3	109.4	(12.8)	108.7
	class A	36.7	8.0	(2.0)	8.1
	class B	41.8	14.0	(5.3)	10.6
lisp	*all*	147.9	205.7	(57.6)	198.4
	class C	33.8	14.7	(4.4)	13.0
	class D	147.9	62.3	(51.7)	13.4

#modified class	Time ratio		
(all)	1	1.05 - 1.3	1.1 - 1.3
1	2.6 - 11	1.5 - 3.9	1

Table 2: Compilation time (under PS/2 AIX)

コンパイル効率　表2は、それぞれの方式によって実行可能コードを生成するのに必要なコンパイル所要時間を示すものである。*all* とある行はすべてのファイルをコンパイルした場合であり、その他の行は、特定のクラスのインプリメンテーション（プライベート・メンバー）が修正された場合の再コンパイル時間である。この中で、クラスAとクラスCはサブクラスをもたないクラスであり、クラスBとクラスDは1個以上のサブクラスをもつクラスである。

括弧の中にプレリンキング・フェーズのみの所要時間を示したが、これは修正されたクラスのサブクラスの数におおよそ比例している。全ファイルのコンパイルの場合はプレリンキング・フェーズの全コンパイル時間に占めるオーバーヘッドはさほど大きくないが、再コンパイルの場合には全コンパイル時間のかなりの部分を占めている。

Time ratio の欄はコンパイル時間の比を示している。表から、late binding を採用することによって再コンパイル時間を劇的に短縮できることがわかる。この効果はプログラムが大きくなるほど顕著となり、コード・サイズが約20K行のプログラムで測定したところでは、最大 1:50 という結果が出た。

実行効率　表3はプログラムの実行所要時間を示す。late binding による2方式の実行時間の増加は static binding の時と比べせいぜい10％である。前にも触れたように、

program	#object	Execution time(sec.)		
		static binding	prelinking	run-time determination
maze	7149	9.6	10.0 (+4.2%)	10.3 (+7.3%)
	9979	20.3	20.7 (+2.0%)	21.3 (+3.6%)
lisp	11222	7.5	8.2 (+9.3%)	8.3 (+10.7%)
	22246	15.1	16.2 (+7.3%)	16.4 (+8.6%)

Table 3: Execution time (under PS/2 AIX)

実行時決定方式はプレリンキング方式の場合より実行効率が落ちるがその差は1～2%である。

我々は同じサンプルをC++でもインプリメントした。COBには、実行時の型チェック、配列の境界チェックなどC++にはない機能があるため、この実行効率の結果をC++の場合と直に比較することはできない。ただしCOBにはstatic bindingを使った最適化を行なう機能が付け加えられているが、それによって生成されたコードからはC++に匹敵する効率が得られた。

現在、COBの処理系はデフォールトでは実行時決定方式を用いたコード生成を行なっている。実行効率を重視する場合には、プログラムの開発が終了した段階で、static bindingによる最適化を行なう。

4 スマート・コンパイル

通常のコンパイル機構（UNIXのmakeなど）では、あるクラスのインターフェースに修正があった場合には、そのクラスのクライアントすべてについて再コンパイルが行なわれる。しかしながら、その中で修正のあった特定のメンバーが、すべてのクライアントで使われているとは限らない。スマート・コンパイルは、プログラムの修正の影響の及ぶ範囲を計算によって求め、再コンパイルされるファイルを最小にするための機構である。我々は[3]の手法を参考にし、これをオブジェクト指向言語に向くよう改良した。

スマート・コンパイルは大きく分けて、COBコンパイラ組み込みのルーチン、スマート・コンパイル・ドライバ、チェンジ・アナライザの3つの部分からなる、ソフトウェア・ツールである。

COBコンパイラ　COBコンパイラは、スマート・コンパイルを行なうために必要な情報を、それぞれのファイルのコンパイル時にサマリー・ファイルに書き込む。これらの情報は、直接にはチェンジ・アナライザで使用され、各クラスのインターフェースに関するdecl情報とインプリメンテーションに関するuse情報との2種類に分かれる。decl情報には、そのクラスのスーパークラスやメンバーに関する属性情報が含まれており、use情報にはそのファイルが実際に使っているクラス・メンバー名が含まれている。

一般にdecl情報を作る場合には、クラスのインターフェースをそのファイル単独でコンパイルできる必要がある。[3]においてはこのための専用のパーザーを用意しているが、COBでは第2節で触れたように、未知のクラスのインターフェースを自動的に読み込む機能があるため、コンパイラの変更はわずかですんだ。

チェンジ・アナライザ　チェンジ・アナライザchange_analysisは、あるクラスXのインターフェースX.hに変更があったときに、そのクライアントであるクラスYのインプリメンテーションY.cobを再コンパイルする必要があるかどうかを判定するルーチンであり、次の形で呼ばれる。

```
Bool change_analysis(X.decl,
                     X.ndecl, Y.use)
```

ここでX.declは変更前のX.hのdecl情報、X.ndeclは変更後のX.hのdecl情報、Y.useはY.cobのuse情報である。このルーチンは、次のような一連の条件判定を行なっている。細部はコンパイラのコード生成方式に依存するものであり、ここでは省略する。

1. X.decl と X.ndecl が同一であれば、false を返す。
2. X.decl と X.ndecl の間でスーパークラスに変更があった場合は、true を返す。
3. X のメンバーが削除されていて、かつそれが Y.use の中に現れていた場合は、true を返す。
4. X のメンバー変数の型に変更があって、かつそれが Y.use の中に現れていた場合は、true を返す。
5. X のメンバー関数のインターフェースに変更があって、かつそれが Y.use の中に現れていた場合は、true を返す。

⋮

ドライバ　ドライバはコマンドの形で用意され、実際にスマート・コンパイルを使用するには、これに必要なソース・ファイル名を引数として与えて起動する。ドライバは、まずファイルの新旧を判定する。クラスのインプリメンテーションに変更があった場合はまずそれをコンパイルする。次に、クラスのインターフェースに変更があった場合は、そのクラスのクライアントを引数としてチェンジ・アナライザを呼び、その

結果に応じて、再コンパイルのためCOBコンパイラを起動する。

5 効果

この節では、今までに紹介したCOBの言語仕様およびスマート・コンパイルの機構によって、どれだけ再コンパイルが短縮できたかを、実際のプログラム開発におけるデータで示す。

データの収集は、コンパイルが成功した時点で新旧のサマリー・ファイルを比較することによっておこない、前回のコンパイルからその時点までに変更されたファイルの行数の合計および、その変更によって再コンパイルが必要となるファイルの行数の合計を、static bindingの場合、late bindingの場合、さらにスマート・コンパイルを使用した場合の3通りについて計算して記録した。これらはすべて自動的に行なわれ、プログラマを煩わすことはない。

図2は、約6500行（クラス数50強）のプログラム開発において2日間データを収集した結果である。図の(a)は2日間全体の時系列データである。データはクラスの定義のあるファイルについてのみとったため、Cの関数のみを含むファイルだけが変更された場合はデータが0になっている。ほとんどの場合、変更されたファイルのみが再コンパイルされているが、これはメンバー関数内部の修正が多いためである。しかし、クラスのインターフェースの変更も少なくはない。図の(b)は特徴的なケースを見やすいように取り出したものである。

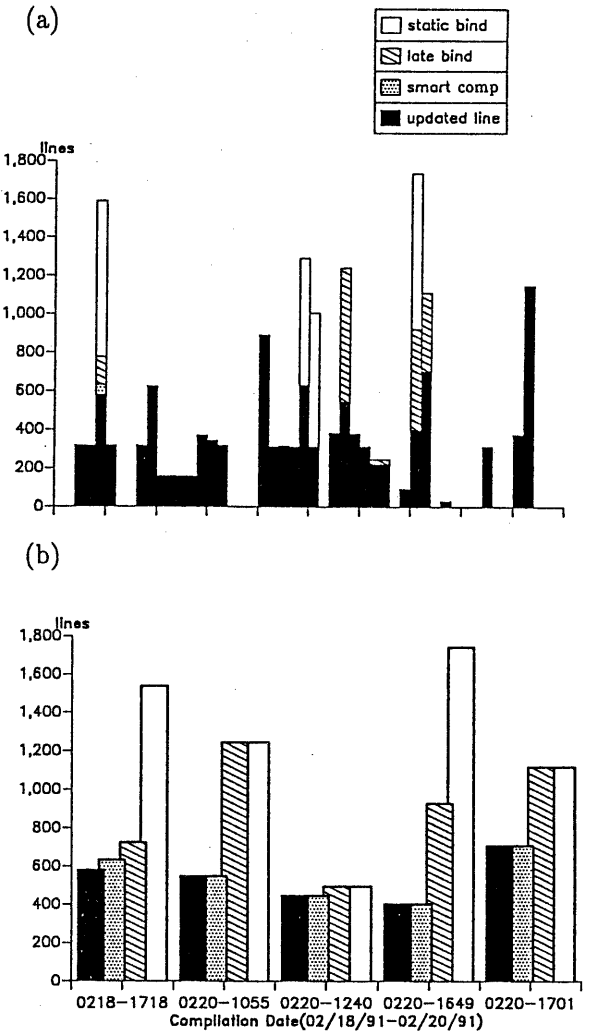

Figure 2: Estimated Lines of Compilation

測定期間中の総コンパイル量は、static binding の場合で約 22000 行、late binding の場合で約 19000 行であり、late binding による節約は 13 %ほどである。一方、スマート・コンパイルは効果が非常に大きく、late binding に比べても 50 %以上の節約となっている。これらは行数による比較であり、実際にはスマート・コンパイルにはその実行によるオーバーヘッドもあるわけだが、節約できるコンパイル時間のほうがはるかに大きい。

6 まとめと今後の課題

以上によって、クラスのインターフェースおよびインプリメンテーションの分離、ならびにスマート・コンパイル機構の、再コンパイル量の短縮に関する有効性を、Cベースのオブジェクト指向言語COBによる実践例によって示した。スマート・コンパイルについてはC++など他の言語についても有効であると考えられる。

スマート・コンパイルにはまだ改良の余地がある。現在の change_analysis は「疑わしきはコンパイル」という原則に基づくどちらかといえば大雑把なものである。これを精密なものにすることによって、さらにコンパイル量を減らすことができるであろう。しかし、精密な分析をすることは同時にスマート・コンパイルのオーバーヘッドを増加させる。したがって、プログラムの開発過程でどのような変更がソース・コードに加えられることが多いかといった点に関する実地のデータを、今後さらに収集することにより、効果的な手法を検討していきたい。

References

[1] Dausmann, M. Reducing Recompilation Costs for Software Systems in Ada. *System Implementation Languages: Experience and Assessment*(1984). Proceedings of the IFIP WG2.4 Conference.

[2] Ellis, M.A. and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[3] Tichy, W.F. and Baker, M.C. Smart Recompilation. *Proceedings of the Twelfth POPL*(1985), 236-244.

[4] T. Onodera and T. Kamimura, *COB Language Manual.* IBM Research, Tokyo Research Laboratory, 1990.