Vol.2018-OS-143 No.9
2018/5/22

大規模ソフトウェアにおけるコンパイル時間の定量的分析と高速化手法の提案

窪田 貴文[1]　鈴木 勇介[1]　河野 健二[1]

概要：大規模なソフトウェアプロジェクトでは多くの開発者が修正・機能追加を行っており，膨大なファイルをコンパイルする機会が頻繁に生じている．例えば，オープンソースのブラウザエンジンである WebKit のビルドボットでは 31 日間のうち 26 日で 2000 秒超えるビルドが実行されており，その時のコンパイルしているファイル数は平均 1000 を超える．本研究では，まず，webkit を含むオープンソースの C/C++ プロジェクトのコンパイル時間を分析した結果を示す．その結果，コンパイラのフロントエンドにおいて冗長な処理が多く含まれていることがわかった．そこで本研究では，コンパイル結果を再利用することでコンパイラのフロントエンドの実行を高速化する手法を提案する．

キーワード：コンパイラ，インクリメンタルビルディング

1. はじめに

近年，システムソフトウェアのコードサイズが大きくなってきている．例えば表 1 に示す通り，オープンソースのブラウザエンジンの 1 つである Webkit [1] では 13,587,992 LOC のコードサイズである．また，オープンソースのコンパイラフレームワークである LLVM [2] は 2,677,161 LOC，そして，オープンソースなオペレーティングシステムの 1 つである Linux Kernel では 11,544,016 LOC と非常に規模が大きい．

このようにコードサイズが大きくになるとビルド時間，コンパイル時間も増大する．本論文では，「ビルド時間」をコンパイラ，リンカを含むプロジェクトのビルドに必要となった全体の時間であり，「コンパイル時間」はそのうちコンパイラが消費した時間である．表 1 は各プロジェクトをシングルスレッドでフルビルドしたときの経過時間とコンパイラが消費した時間を示している．ビルド環境は Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz のプロセッサ, メモリサイズ 128GB　である．使用したコンパイラは LLVM/Clang version 7.0.0@329912, リンカは GNU ld v2.29 である．Webkit では 3h 40m 消費し，そのうちコンパイラが占める割合は 97.7% である．LLVM では 3h 8m 消費し，そのうちコンパイラが占める割合は 98.0% である．Linux では 24m 消費し，コンパイラが 90.1% のビルド時間を専有している．

このようにビルド時間が増加すると開発者の時間を奪うことになり，その生産性が低下するという問題が生じる．開発者は日々，ソースコードを編集しては再ビルドを行なうというプロセスを繰り返して開発を行っている．このような開発サイクルの中でビルド時間が延びることは開発者の待機時間が増えるため望ましくない．

こうした開発サイクルの中で再ビルドの時間を短縮させる手法が幾つか存在する．大規模なシステムソフトウェアでは何らかのビルドシステムに依存してコンパイルを行っている．例えば，Linux kernel では GNU Makefile [3], LLVM/Clang, Webkit では CMake [4] と GNU Makefile もしくは Ninja [5] を採用している．このようなビルドシステムではソースファイルの依存関係を記述することで，タイムスタンプが更新されたファイルとそのファイルに依存するファイルのみを再コンパイルすることでビルド時間の短縮を行っている．

また，ccache [?] ではプリプロセスした結果をキーとしてコンパイル結果をキャッシュしておくことで，再コンパイル時間を短縮できるツールである．ccache を用いた再コンパイルではプリプロセス後に前回のコンパイル時のプリプロセス結果と同一かどうかチェックする．もし同一ならばそれ以降の処理を中断し，キャッシュされたコンパイル結果を再利用する．このようにすることでたとえビルドシステムが再コンパイルをする必要があると判断したとしても，プリプロセスしたときに変化がなければコンパイル時

1　慶應義塾大学
Keio University

ⓒ 2018 Information Processing Society of Japan

表 1 システムソフトウェアにおけるコードサイズとコンパイル時間

Projet	Code Size	Build Time	Compilir Time
Webkit	13,587,992	3h 40m	3h 35m (97.7%)
LLVM/Clang	2,677,161	3h 8m	3h 4m (98.0%)
Linux	11,544,016	24m	22m (90.1%)

間が短縮できる．実際に Webkit, LLVM/Clang のビルドボットではビルドシステムと ccache の両方とも採用されている．

しかしながら，これらの開発ツールを使用してもソースコードの変更が大きく，プリプロセスした結果が一致しないときは再コンパイルされる．本論文ではまず，Webkit と LLVM のビルドボットを調査し日々の開発がビルド時間にどのような影響をあたえるか調査する．その結果，日々の開発による変更でもフルビルドに近いコンパイルが必要になる機会が頻繁に発生していることを示す．例えば，Webkit の GTK Linux 64-bit Release 用のビルドボットで 2018 年 3 月 17 日から 4 月 17 日まで合計 487 回ビルドを行っているが，２日に一回は平均 1119 ファイルのコンパイルを行なうビルドが実行され，この時のビルド時間は 8 スレッドで 25 分を超える．このように日々の開発においても 1000 を超えるファイルを再コンパイルする機会が多くビルド時間を短縮するにはコンパイラのスループットを上げる必要がある．

そこで本論文では Webkit, LLVM, Linux を対象にコンパイラのボトルネックとなっている部分を調査し，コンパイラのフロントエンド部分が実行時間の多くを占めていることを示す．例えば，WebKit ではフロントエンドの実行時間が全体の 63.7%以上を占めている．また，フロントエンドのヘッダファイルの処理において冗長な処理が多く含まれていることを示す．

本論文ではコンパイラのフロントエンドをインクリメンタルにビルドすることで，フロントエンドの実行時間を短縮する手法を提案する．本論文では，不必要なヘッダファイルの再コンパイルを省略し，コンパイル結果を再利用できるようにすることでコンパイラのフロントエンドの実行を高速化する．

コンパイラのフロントエンドをインクリメンタルに行なう研究は 1960 年代に活発に行われた [6–8]．しかし，これらの研究はまだマシンのリソースが潤沢ではないときに，１つマシンを複数ユーザーで共有することを目的としておりコンパイラの高速化を目的とはしていない．また，インタラクティブな開発環境を構築する研究の一環としてインクリメンタルにコンパイルを行なう研究が 1980 年代に行われた [9–11]．これらの研究では syntax-directed なエディタを使用して，変更が生じた部分のみを再コンパイルすることでインクリメンタルなビルドを実現している．一方，本論文ではフルビルド，もしくは変更量が大きい場合においてコンパイルを高速化するインクリメンタルなコンパイル手法を提案する．

本論文では，提案手法のプロトタイプを LLVM/Clang 上で実装し，予備実験として Hello World を出力するプログラムの 1st ビルドと 2nd ビルドの時間を比較した結果，最大 8 倍高速化できることを示す．

2. 背景

2.1 システムソフトウェアにおけるビルド時間の調査

表 1 で示されている通り，近年，システムソフトウェアのコードサイズが大規模になっている．オープンソースのブラウザエンジンの 1 つである Webkit [1] では 13,587,992 行もの C/C++ コードが存在している．また，オープンソースのコンパイラフレームワークである LLVM [2] は 2,677,161 LOC で，そして，オープンソースなオペレーティングシステムの 1 つである Linux Kernel でも 11,544,016 LOC におよぶ．

このようにコードサイズが膨大になるとソフトウェアのビルド時間の大半はコンパイルに割かれることになる．Webkit, LLVM, Linux kernel のいずれもビルド時間の 90% 以上がコンパイラが専有している．今回はコンパイルにかかる総時間を測るためシングルスレッドでビルドを行っている．

よってコンパイル時間が長くなると開発者の時間を奪うことになり，その生産性を低下させてしまうという問題がある．開発者の日々，ソースコードの変更を行っては再コンパイルし，テストを行なうことで変更が正しいか確かめている．このような開発サイクルの中でコンパイル時間が長くなると開発者が次の作業に取り掛かるまでの時間が長くなり生産性が低下してしまう．

そこで，システムソフトウェアプロジェクトを効率的にビルドを行なうツールが幾つか確立されている．

ビルドシステム: 通常，大規模なシステムソフトウェアプロジェクトでは GNU Makefile [3], Ninja Build system [5], CMake [4] のようなビルドシステムを利用してビルドを行っている．ビルドシステムではプロジェクト内のソースファイルの依存関係を記述することで，ソースファイルのタイムスタンプが更新された際に更新されたファイルとそのファイルに依存するターゲットのみを再コンパイルすることで，プロジェクト全体のフルビルドを避け，再コンパイルの時間を短縮できる．

ccache: ビルドシステム単体ではタイムスタンプ更新されたファイルとそれに依存するファイルはソースコードの変更がなかったとしても必ず再コンパイルされてしまう．この問題を解決するために ccache [12] ではプリプロセス後のソースファイルの内容のハッシュ値をキーとしてコンパイル結果をキャッシュすることで解決している．つまり，たとえソースファイルのタイムスタンプが更新されても，プリプロセス後の内容が同一ならばキャッシュされた

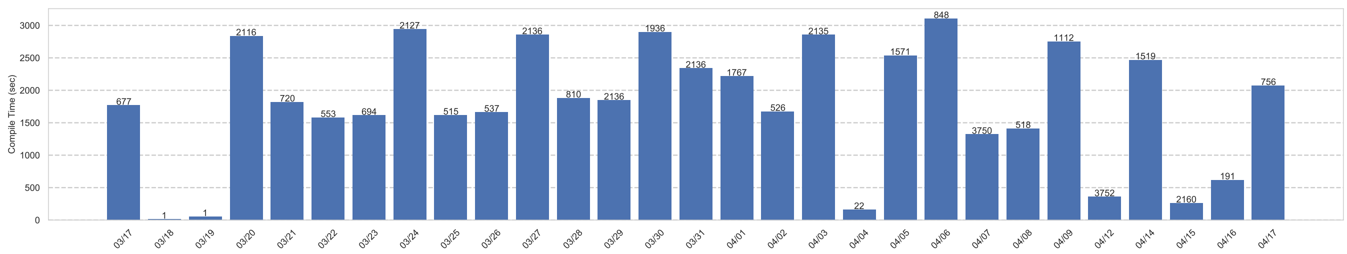

図 1 Webkit GTK Linux 64-bit Release 用ビルドボットにおける日にちごとの最大ビルド時間とその際の再コンパイルファイル数 (2018/03/17 ～ 2018/04/17)

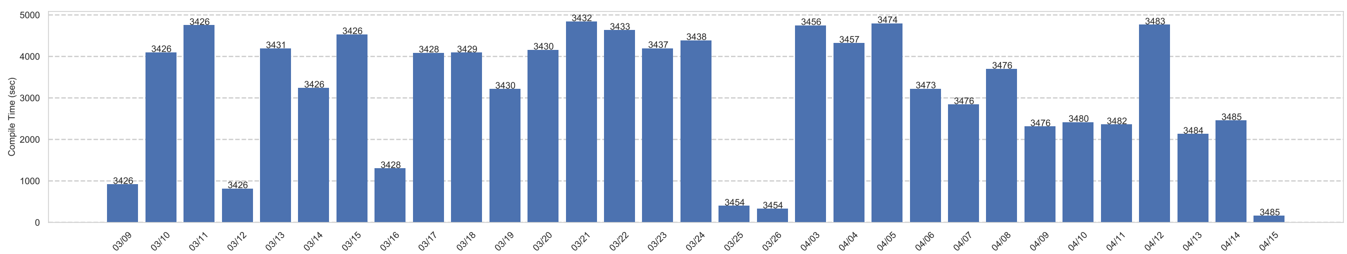

図 2 LLVM/Clang/LLD Debin 用ビルドボットにおける日にちごとの最大ビルド時間とその際の再コンパイルファイル数 (2018/03/9 ～ 2018/04/15)

コンパイル結果を再利用することでプリプロセス以降のコンパイルを処理を省ける．

しかしながら，これらの開発ツールを利用してもソースファイルの変更が大きい場合，再コンパイルが必要となる．本研究では，開発者の日々のソースコードへの変更がリビルド時間にどれだけ影響があるか調べるために，WebKit と LLVM のビルドボットのログを調査し図 1，2 に示している．各図では約 1 ヶ月間のビルドボットで実行されたビルドログを分析，各日付で一番長く時間を要したビルド時間とその際に再コンパイルを行おうとしたファイル数を表示している．横軸は各日付，縦軸はビルド時間，縦棒の上に表示されている数字が再コンパイルを行おうとしているファイル数である．

GTK Linux 64-bit Release 用のビルドボットでは CMake と ccache + gcc/g++ を用いて 8 スレッドでコンパイルが行われている．表 1 では 2018 年 3 月 17 日から 4 月 17 日までのビルドログの分析結果を示している．この期間の間に合計 484 回のビルドが実行された．表 1 によると従来のビルドシステムと ccache が有効に機能している日があることがわかる．例えば，3 月 18 日では 1 ファイルのみの再コンパイルで終えており 1 分以内ビルドを終了している，これはビルドシステムによる依存管理が有効に機能しているためである．また，4 月 12 日では 3752 ファイルの再コンパイルを実行しようとたが ccache が有効に機能して実際のビルド時間は比較的に短くすんでいる．これはチェンジセットがビルドシステムの設定変更によるものでプロジェクト全体への依存度は高いがソースコードのプリプロセス結果には影響がない変更のためだったからである．

しかしながら，29 日間のうち 21 日で 1500 秒，つまり 25 分を超えるビルドが実行されている．また，このときに再コンパイルを要したファイル数の平均は 1302 ファイルである．このように日々の開発によって大量のファイルを実際に再コンパイルする必要がある機会が容易に起こりうる．

表 2 によると LLVM/Clang/LLD Debin 用ビルドボットにおいても同様の傾向が見られる．このビルドボットでは CMake と ccache + clang を用いて 18 スレッドでコンパイルが行われている．LLVM のビルドでは前回のビルドディレクトリを消去して新しいビルドディレクトリでビルドを行っている．そのため，CMake + Ninja ビルドシステムは毎回フルビルドを実行しようとするので再コンパイルのファイル数は毎回ほぼ一定である．実際には ccache によって必要最低限の再コンパイルのみが実行される．

この表では 2018 年 3 月 9 日から 4 月 15 日までの合計 473 回のビルドログの分析結果を示している．WebKit のケースと同様にビルドシステムと ccache が機能している日（例，3 月 25 日）があるが，31 日間のうち 26 日で 2000 秒超えるビルドが実行されている．このように大規模なオープンソースのシステムソフトウェアプロジェクトではコードサイズが大きくソースコード依存関係の規模も大きいため，一部のソースコードの変更が大量のファイルを再コンパイルする必要がでてきてしまう．再コンパイル時間が長くなると開発者の時間を奪うことになり，日々の edit-compile-test という開発サイクルが滞り生産性が落ちてしまう．

ⓒ 2018 Information Processing Society of Japan

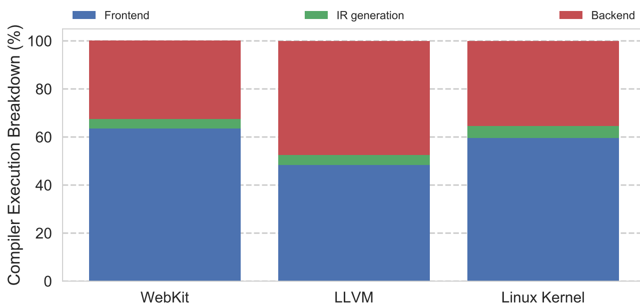

図 3 WebKit, LLVM, Linux におけるコンパイラ内の実行時間のブレイクダウン

2.2 コンパイラ内の実行時間のブレイクダウン

大量のファイルをコンパイルする時間を短縮させるためには，コンパイラのスループットを向上させる必要がある．そこで，本研究では WebKit，LLVM，Linux，においてフルビルドの際のコンパイラ内の実行時間のブレイクダウンを調査した．図 3 がその結果である．Frontend はコンパイラ内の字句解析，構文解析，意味解析に該当する．IR Generation が構文木 (AST) から中間表現 (IR) を生成するプロセスである．Backend は最適化処理とオブジェクトコードの生成部分の処理時間を示している．図 3 によると全プロジェクトにおいて，Frontend と Backend の処理時間の大半を占めていることがわかる．

よって，コンパイラのフロントエンドとバックエンドの処理を高速化すればコンパイラのスループットが向上できる．しかしながら，バックエンドの処理を高速化することは難しい．なぜなら，バックエンドの処理には様々な最適化処理が含まれているからである．この最適化によって生成されるコードの品質が決定される

コンパイラにとって生成されるコードの品質はスループットと同様に重要な要素である．通常，生成されるコードの品質は最適化処理によって決定される．開発者はコンパイラに最適化オプション (-O0, -O1, -O2 など) を指定することによって最適化のレベルを変更できる．最適化を一切行わないことによってバックエンドの処理時間は大幅に短縮できるが，これは望ましくない．

システムソフトウェアプロジェクトではそのリリース時にソフトウェアのパフォーマンスを上げるために最適化を有効にしてコンパイルを行なうことが普通である．しかし，もしリリース時と違った最適化レベルで開発を行っているとリリース時とは違ったパフォーマンス特性のもと開発を行っているので，パフォーマンスチューニングなどが無意味になる危険がある．また，リリース前にしか最適化を行わない場合，最適化を行ったバイナリのテストが不十分になるという問題もある．したがって，コンパイル時間を短縮させるために最適化処理を行わないといういうことは現実的ではない．

一方，フロントエンドでは多くの冗長な処理が含まれている．ここで，簡単なソースファイル A.cpp (図 4)，B.cpp(図 5) を連続してコンパイルする時を考える．それぞれのファイルのコンパイルは別プロセスで行われ，フロントエンドのワークフローは図 6 で表される．各ファイルのコンパイルではまず，ソースファイルの内容をメモリバッファに保存することから始まる．そして，Lexer と Parser によってソースファイルの AST を構築する．この時, Lexer によって `#include` Directive が発見されたら現在の字句解析を中断し，インクルード先の解析を先に行なう．

```
1 #include <iostream>
2 int A() {
3   std::cout << "A()" << std::endl;
4   return 0;
5 }
```

図 4 A.cpp

```
1 #include <iostream>
2 #include <vector>
3 int B() {
4   std::vector<int> v;
5   std::cout << "B(): " << v.size() << std::endl;
6   return 0;
7 }
```

図 5 B.cpp

なので A.cpp では, A.cpp ファイルの読み込み後, 1 行目で iostream ヘッダをインクルードしているので，iostream の解析が先に行われる．そして iostream の AST の構築後, 2 行目以降の解析が再開される．B.cpp ではほぼ A.cpp と同じワークフローだが vector　ヘッダの処理が追加されている．

ここで問題なのは, A.cpp と B.cpp のコンパイルにおいて iostream ヘッダのコンパイルが冗長であるということである． 2 つのファイルのコンパイルにおいて，iostream ヘッダのコンパイル結果は同じになるので，B.cpp でのコンパイル時には A.cpp でのコンパイル結果を再利用できるはずである．同じヘッダファイルを別々のソースファイルがインクルードすることは大規模なシステムソフトウェアでは頻繁に生じる．例えば，大規模なシステムソフトウェアでは独自のデータ構造をヘッダファイルに定義していることは普通であり，そのデータ構造を使用するたびに同一のヘッダファイルをインクルードする．本研究では，このような不必要なヘッダファイルの再コンパイルを省略し，コンパイル結果を再利用できるようにすることでスループットの向上を図る．

3. 提案

本研究では以下の 3 つのデザインポリシーが存在する．

- 生成されるコードの品質を落とさずにコンパイル速度を向上させる: リリース時と同等の最適化処理を行な

ⓒ 2018 Information Processing Society of Japan

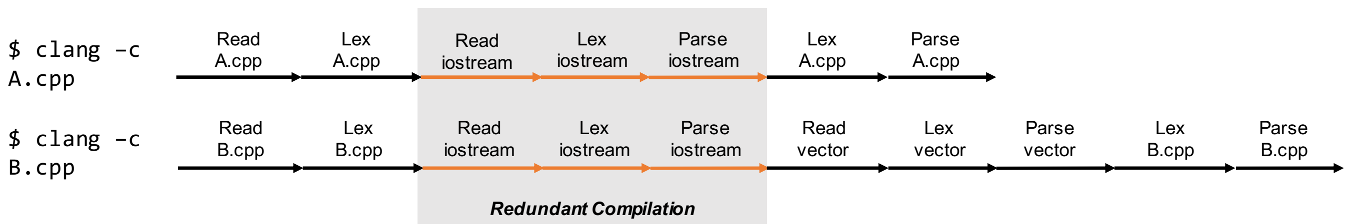

図 6　A.cpp と B.cpp をコンパイルした時のフロントエンドのワークフロー

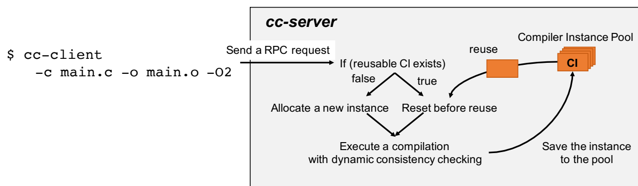

図 7　デザイン概要図

う前提のもと，フロントエンドの処理を高速化することで生成されるコードの品質を落とさずにコンパイル速度を向上させる．

- プロジェクトへの変更を最小限にする: 本研究の提案手法よるソフトウェアプロジェクトへの変更は最小限にするべきである．ソフトウェアプロジェクトへの変更が大きいと，結局個々のプロジェクトに対して ad-hoc な対応が必要になってしまう．本提案は drop-in replacement な手法である，開発者は通常，下のようなコマンドを実行する．

```
make CC=cc-cilent CXX=cc-cilent
```

  このようにすることで，複数の異なるプロジェクトで容易に同等の効果を得られることになる．
- マルチスレッド，分散コンパイル環境への対応: 通常，開発者はコンパイルを並列処理させてコンパイル時間を短縮させている．本研究の提案もマルチスレッド，もしくは分散環境でのコンパイルにおいても恩恵を受けられるように設計するべきである．

3.1　概要

図 7 は本提案の概要図である．本提案ではユーザーは通常のコンパイル時と同様にコンパイルコマンドを `cc-client` に対して行なう．`cc-client` は実際にはコンパイルを行わず，`cc-server` に対してコンパイルリクエストを送信する．`cc-server` はリクエストを受信するとスレッドを立ち上げ，コンパイルを実行する．`cc-server` では初めてコンパイルを行なう際は，通常のコンパイル時と同じように動作する．まず `cc-server` はコンパイル時のデータを管理するコンパイルインスタンスをアロケーションし，コンパイルを行なう．コンパイルインスタンスはコンパイル時に生成される様々なデータを管理している．例えば，ソースファイルのバッファキャッシュ，マクロ情報，AST，IR などである．通常のコンパイルが正常終了したらそのインスタンスを解放せず，インスタンスプールに保存しておく．この際に IR，バックエンドで管理されているデータは初期化しておく．

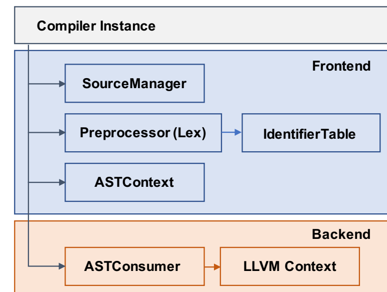

図 8　Clang におけるコンパイラインスタンスによって管理されるデータ群

そして，2 回目以降のコンパイル時には，まずインスタンスプールに存在するコンパイルインスタンスから再利用できるものがないかチェックを行なう．そして再利用できるインスタンスを発見したら，インスタンス内のフロントエンドのデータからソースファイル特有のデータをリセットして再利用する．こうすることで，ヘッダファイルに対する再コンパイルを行わないためフロントエンドの処理を高速化できる．

また，インスタンスを再利用したコンパイルでは動的にコンパイルの一貫性が壊れていないかチェックを行なう．もし，一貫性が壊れた際にはコンパイルを中断し，新しいインスタンスを利用して再コンパイルを行なう．一貫性が壊れたインスタンスは解放し，プールから取り除き再度利用されることは防ぐ．

3.2　コンパイラインスタンスとその再利用

コンパイラインスタンスとは，コンパイル時に生成される様々なデータを管理しているオブジェクトである．本研究は LLVM/Clang 用いて実装を行っている．Clang におけるコンパイルインスタンスが管理しているデータの簡略図を図 8 に示す．ソースマネージャは開いたファイル内容のメモリ管理をおこなっている．プリプロセッサは字句解析を行い，マクロ展開の情報などを Identifier Table に記録している．パーサによって構築される AST は AST コンテキストに保存される．これら 3 つのデータ構造をフロントエンドの処理で構築される．一方，構築された AST

```
1 #ifndef INCLUDE_GUARD
2 #define INCLUDE_GUARD
3
4 #if defined(DEP)
5   #include "A.h"
6 #else
7   #include "B.h"
8 #endif
9 #endif
```

図 9 An example of the macro-dependent header

を処理する AST Consumer はバックエンド側で使用される．AST Consumer は AST から IR/Object code を生成し，LLVM Context によって管理される．

本提案では，コンパイラインスタンスがインスタンスプールで管理し再利用する．インスタンスがプールに追加されるときはフロントエンドの結果のみを再利用するのでバックエンドのデータは解放する．インスタンスが再利用できるかは，コンパイル時に渡される Definition のハッシュ値をキーとして判断される．インスタンスの再利用時には，前回のコンパイル時に生成されたデータからソースファイル特有のデータのみをリセットし，ヘッダファイルのコンパイル結果のみを再利用する．具体的には，ソースファイルによって定義されたマクロ情報のリセット，AST ノードの削除を行なう．そして，字句解析の開始位置を新しいソースファイルのエントリに設定することで，擬似的に前回のヘッダファイルをインクルード後の状態からコンパイルしているようにする．

ここで余分な AST ノードを処理することを避けるために，各ヘッダファイルをインクルードしたらどの AST ノードが処理されるかを記録しておく．そして，ヘッダファイルのインクルード時に字句解析を行なう代わりにすでに存在する AST ノードを記録どおりに処理する．このようにすることで過不足なく重複したヘッダファイルの AST を再利用できる．

3.3 一貫性のチェック

インスタンスの再利用できるかはコンパイル時に引数としてわたされる Definition の値をキーとして判断している．しかし，それだけではコンパイル時の一貫性が保たれない場合がある．

例えば，図 9 のようなマクロ依存があるヘッダファイルである．このコードでは DEP が定義されている場合，A.h がインクルードされそうでなければ B.h がインクルードされる．ここで，一回目のコンパイル時には DEP が定義されており，A.h がインクルードされているとする．この場合，コンパイラインスタンスには A.h の内容の AST が保存されている．そして，DEP が定義されていない別ファイルのコンパイルにこのインスタンスが再利用されたとすると，すでにこの macro-dependent なヘッダファイルはコンパイル済みなので A.h の AST を再利用する．しかしながら，今回のコンパイルでは DEP が定義されていないので正しくは B.h をインクルードして B.h の AST を生成しなければいけない．

この問題を解決するために，提案手法ではヘッダファイルで使用されるマクロ情報を記録し，再利用時には更新がないかチェックを行なう．もし更新があったときは再利用しているインスタンスでの処理を中断し，新たにコンパイラインスタンスをアロケーションして再度初めからコンパイルを実行する．このようにすることでコンパイル結果の一貫性が保たれる．

```
#include <stdio.h>
int main() {
  printf("Hello World.\n");
  return 0;
}
```

図 10 hello.c

```
#include <iostream>
int main() {
  std::cout << "Hello World." << std::endl;
  return 0;
}
```

図 11 hello.cpp

表 2 1st コンパイルと 2nd コンパイルの実行時間の比較

File	1st Compile Time (sec)	2nd Compile Time (sec)
hello.c	0.028	0.022
helloc.cpp	0.270	0.031

3.4 実装

本研究では提案手法の予備実験として，提案内容の一部をプロトタイプとして実装している．今回実装したのは，コンパイラインスタンスを再利用する機構までであり，一貫性のチェックはまだ未実装である．プロトタイプは LLVM/Clang 7.0.0@329912 上で使用している．RPC ライブラリは Apache Thrift 0.10.0 を使用している．

4. 実験結果

今回は予備実験として，実装したプロトタイプを用いて簡単プログラムの再コンパイル時間がどれだけ短縮されるか確かめている．使用したプログラムは図 10,11 である．プロトタイプのコンパイラを使用して 1st コンパイルと 2nd コンパイルの実行時間の比較を表 2 に示す．hello.c のコンパイルでは 1.27 倍，hello.cpp にコンパイルでは 8.7 倍高速になった．C++ のコンパイルのほうが高速になるのはテンプレートのインスタン化などフロントエンドの処理が C よりも重いためである．

5. 参考文献

5.1 インクリメンタルビルディング

コンパイラのフロントエンドをインクリメンタルに行なう研究は 1960 年代に活発に行われた [6–8]．しかし，これらの研究はまだマシンのリソースが潤沢ではないときに，1 つマシンを複数ユーザーで共有することを目的としており コンパイラの高速化を目的とはしていない．また，インタラクティブな開発環境を構築する研究の一環としてインクリメンタルにコンパイルを行なう研究が 1980 年代に行われた [9–11]．これらの研究では syntax-directed なエディタを使用して，変更が生じた部分のみを再コンパイルすることでインクリメンタルなビルドを実現している．一方，本論文ではフルビルド，もしくは変更量が大きい場合においてコンパイルを高速化するインクリメンタルなコンパイル手法を提案している．

5.2 Zapcc

Zapcc [13] は商用コンパイラの 1 つである．Zapcc ではコンパイラをクライアント・サーバモデルにし，コンパイル結果をサーバーにキャッシュし，再利用することで C++ のコンパイル時間を短縮している．しかし，C のプロジェクトではキャッシュを無効にしている．

本提案では，C++ だけはなく C のプロジェクトにおいてもコンパイル時間を短縮させている．

5.3 プリコンパイル済みヘッダ，C++ モジュール

プリコンパイル済みヘッダは，ヘッダを予め中間表現に変換しておきコンパイル時にはその中間表現を利用することでコンパイル時間を大幅に短縮する手法である．しかしながら，この手法は複数のヘッダ間の依存関係を把握しなければならず，プロジェクト毎に ad-hoc な対応が必要となる．

C++ モジュールは従来の `#include` に代わり C++ においてモジュールを可能にする機能である．従来の `#include` とは違い，ライブラリのバイナリを直接ロードすることによってコンパイル時間を短縮させる手法である．しかし，C++ モジュールは新しい機能であり，レガシなプロジェクトでは対応していない．

本提案ではコンパイラが自動でコンパイラ結果を再利用することによって対象となるプロジェクトの変更を最小限にしながらコンパイラのスループットを向上させている．

6. 終わりに

大規模なソフトウェアプロジェクトでは多くの開発者が修正・機能追加を行っている．その結果，日々のソースコードの変更によって大量のファイルを再コンパイルする機会が多く生じてしまう．そこで本論文では，まず，Webkit, LLVM，Linux Kernel においてのコンパイル時間のブレイクダウンを調査し，コンパイラのどの部分がボトルネックになっているか調査している．その結果，フロントエンドの部分に冗長な処理が含まれていることを示している．そこで，本論文ではコンパイラのフロントエンド部分をインクリメンタルに行なう手法を提案している．提案手法によって C++ の Hello World のプログラムの再コンパイルが 8.7 倍に高速になる．

謝辞

本研究は，JST, CREST, JPMJCR1683 の支援を受けたものである

参考文献

[1] : Webkit. `https://webkit.org/`.
[2] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)* (2004).
[3] : GNU Make. `https://www.gnu.org/software/make/`.
[4] : CMake. `https://cmake.org/`.
[5] : Ninja. `https://ninja-build.org/`.
[6] Harry Katzan, J.: Batch, conversational, and incremental compilers, *Proceedings of 1969 Spring Joint Computer Conference, AFIPS Conference* (1969).
[7] Braden, H. V. and Wulf, W. A.: The Implementation Of A Basic System In A Multiprogramming Environment, *Communications of the ACM* (1968).
[8] Peccoud, M., Griffiths, M. and Peltier, M.: Incremental interactive compilation, *IFIP Congress (1)*, pp. 384–387 (1968).
[9] Medina-Mora, R. and Feiler, P. H.: An Incremental Programming Environment, *IEEE Trans. Softw. Eng* (1981).
[10] Tim Teitelbaum and Thomas Reps: The Cornell program synthesizer: a syntax-directed programming environment, *Communications of the ACM* (1981).
[11] Schwartz, M. D., Delisle, N. M. and Begwani, V. S.: Incremental Compilation in Magpie, *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Cowtruction (CC '84)* (1984).
[12] : ccache — a fast C/C++ compiler cache. `https://ccache.samba.org/`.
[13] : Zapcc; A (Much) Faster C++ Compiler. `https://www.zapcc.com/`.

ⓒ 2018 Information Processing Society of Japan