修　士　論　文

題　　目　　組み込み開発における
　　コンパイラ最適化に起因する
　　不具合の推定手法の提案

主任指導教員　　水野　修　教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　16622023

氏　　名　　田中　健太郎

平成30年2月9日提出

様式 2 号

学位論文の要旨（和文）

平成 30 年 2 月 9 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　　情報工学専攻
平成 28 年入学
学生番号　　16622023
氏　　名　　田中　健太郎　㊞

（主任指導教員　　水野　修　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

組み込み開発におけるコンパイラ最適化に起因する不具合の推定手法の提案

2.　論文内容の要旨（400 字程度）

コンパイラ最適化は実行速度の向上やコードサイズの削減の観点から非常に重要な機能である．しかし，割り込み処理やリアルタイム OS による並行処理を使用する組み込み開発では，命令の実行フローがコンパイラ最適化時に想定されたものと異なる場合があり，最適化によって不具合がしばしば発生する．

一方で，組み込み開発で広く使われているコンパイラ群 GNU Compiler Collection には最適化レポートを出力する機能が搭載されていないため，最適化によって発生する不具合を特定することが難しい．そこで，本研究では，GCC が出力する中間言語ファイルを静的解析することにより，コンパイラ最適化によって生成される不具合がどのように発生するかを調査し，それらの自動検出を行った．事例研究より，3 種類のコンパイラ最適化に起因する不具合を発見し，提案手法により不具合の候補となる箇所を出力できると分かった．

Detecting software failures generated by compiler optimization in embedded systems

2018 16622023 *TANAKA Kentaro*

Abstract

Compiler optimization assumes an important role in minimizing memory consumption and size of executable files, and maximizing speed especially for embedded systems. However, in embedded systems, it sometimes generates software failures since a compiler does not consider interrupts and concurrent processing.

Although the GNU Compiler Collection (GCC) is commonly used in embedded system development, it cannot generate optimization report. This makes detecting failures difficult. In this paper, we collected open-source embedded projects from GitHub using information of timelines which are listed in the GitHub Archive. Then, using these embedded projects, we investigated how optimization failures were generated, and proposed a method of detecting them automatically with interlanguage files which are generated by GCC. As results of experiments, we found three types of failures related to compiler optimization and we also found that the proposed approach was able to successfully detect candidates of these failures. Therefore, we can conclude that the proposed approach is helpful for developing embedded systems.

目　次

1. 緒言

コンパイラには実行する命令を変換することにより，実行速度の向上やコードサイズの削減を行う最適化という機能がある．メインメモリやフラッシュメモリの容量が限られる組み込み開発において，それらの効果は非常に重要である [1][2].

しかし，割り込み処理やリアルタイム OS による並行処理を使用する組み込み開発では，命令の実行フローがコンパイラ最適化時に想定されたものと異なる場合があり，最適化によって不具合がしばしば発生する．C 言語ではこのような可能性を想定し，特定の変数に関するコンパイラ最適化を抑制するために volatile 装飾子が用意されている [3].

コンパイラ最適化に関連する不具合はコンパイラの機能に依存するため，プログラマが記述したソースコードの情報のみで不具合を検出することは難しい．しかし，組み込み開発で広く使われているコンパイラ群である GNU コンパイラコレクション (GNU Compiler Collection; GCC) (注 1)には，最適化レポートを出力する機能が搭載されておらず，最適化に起因する不具合を開発者が発見しにくい問題が存在する [4]. 一方で，GCC には最適化後の各関数の処理内容を中間言語で出力する機能が存在する．この機能を用いて最適化前後の処理内容を比較することにより，最適化の内容を知ることができる．

本研究では，GitHub に公開されているオープンソースプロジェクトから組み込み機器をターゲットにしたプロジェクトを取得し，それらの特徴を探査的に研究する．さらに，GCC が出力する中間言語を静的解析することにより，volatile 装飾子を使用していないことに起因する不具合の検出手法を提案する．また，GitHub に公開されている組み込み向けオープンソースプロジェクトの開発履歴を用いることにより，提案手法によって不具合を検出可能であるかを検証する．

(注 1) : https://gcc.gnu.org/

2. 用語

2.1 コンパイラ最適化

2.1.1 静的単一代入形式（SSA 形式）

静的単一代入形式（SSA 形式）は，変数の定義が字面上唯一になるように添字をつけた中間表現形式である [5][6][7]．例えば，C 言語で書かれた図 2.1 のような処理の場合，SSA 形式を用いると図 2.2 のように表現できる．SSA 形式で表現することにより，変数 a の値が 0 である状態と 1 である状態をそれぞれ $a.0$ と $a.1$，変数 b の値が 1 である状態と 2 である状態をそれぞれ $b.0$ と $b.1$ で区別することができる．このように変数の各状態を区別して，コンパイラによる最適化は行われる．

2.1.2 無用命令除去

コンパイラの最適化の機能として，実行されることのない命令を削除する無用命令除去がある．例えば，図 2.3 のような C 言語のソースコードの場合，if 文の直前に $\mathrm{flag} = 0$ が存在するため，if 文の中の処理が行われることはない．したがって条件分岐は削除され，図 2.4 のように実行フローは最適化される．

2.1.3 ループ展開

ループ展開とは，ループ回数が決定している処理を独立した命令ブロックの連続に書き換え，プログラムのサイズを犠牲にすることで高速化する最適化である．この最適化により，ループの終了条件を確認する処理が削除され，実行速度が上昇する．

2.1.4 ループ不変式移動

ループ不変式移動とは，繰り返し処理中で結果が常に同じになるような式や代入を繰り返し処理の前に移動する最適化処理である [8]．これにより繰り返し内の実行命令数を削減し，実行速度の高速化を行うことができる．

```
  a = 0;
  a = a + 1;
  b = 1;
  b = b + a;
```

図 2.1　サンプルコード (C 言語)

```
  a.0 = 0;
  a.1 = a.0 + 1;
  b.0 = 1;
  b.1 = b.0 + a.1;
```

図 2.2　SSA 形式での表現

```
int flag;

void function(void)
{
  flag = 0;

  // ...

  if (flag)
  {
    do_something();
  }
}
```

図 2.3　無用命令除去の例 (sample.c)

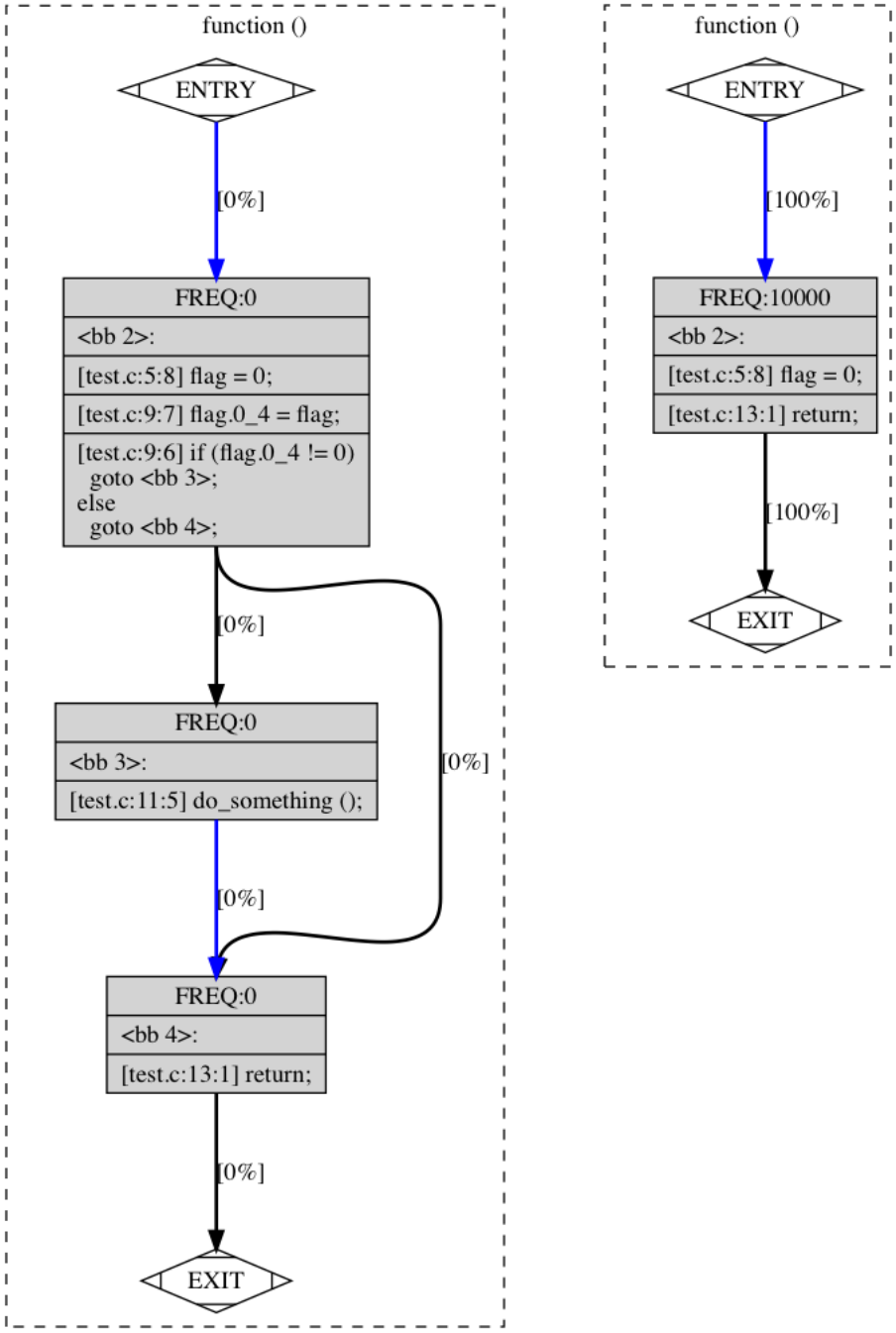

図 2.4　無用命令除去 (左：最適化前　右：最適化後)

2.2 割り込み

割り込みとは，コンピュータが周辺機器から受け取る要求の一種である．割り込み要求を受けたコンピュータは実行中の処理を中断して，優先度の高い別の処理を行う．具体的な処理の内容は割り込みサービスルーチン (Interrupt Service Routine; ISR) に記述する．

組み込み分野における割り込み処理の例を以下に記す．

- ボタンが押された場合
- データの送受信が完了した場合
- 一定時間が経過した場合

2.3 GNU Compiler Collection (GCC)

GNU Compiler Collection(GCC) とは，UNIX 互換ソフトウェア群開発プロジェクト GNU によって開発されるコンパイラ群であり，商用，非商用を問わず多くの環境で標準的な C コンパイラとして採用されている．GCC の特徴として，各マイクロコントローラのアーキテクチャに対応した様々なクロスコンパイラが開発されており，組み込み用途においても広く用いられている点が挙げられる．

2.3.1 最適化レベル

GCC には複数の最適化レベルが用意されており，レベルによって指定されたレベルに応じた最適化を行う．最適化レベルの指定は-O オプションにて行う．各最適化レベルの機能を以下に記す．

最適化レベル 0 (-O0)

非常に単純な最適化オプションを除き，最適化を行わない．

最適化レベル 1 (-O1)

出力バイナリのコードサイズ，及び実行速度の両方を低減できる最適化のみを行う．コンパイル時間を大幅に増大させる可能性のある最適化は実行しない．

最適化レベル 2 (-O2)

コンパイル時間を大幅に増大させる可能性のある最適化を含め，容量と速度のトレードオフが必要ない最適化を実行する．

最適化レベル s (-Os)

バイナリのサイズを削減する最適化を実行する．

2.3.2 -fdump-tree-optimized-lineno オプション

GCC のコンパイラオプションには，最適化した結果を中間言語を元のソースコードの行番号とともに出力する-fdump-tree-optimized-lineno という機能が存在する．例えば，図 2.3 の C 言語のソースコードを最適化なし (オプション-O0) でビルドした場合，表 2.5 のような中間言語が出力される．

本研究ではこの中間言語を用いることによって不具合の検出を行う．

2.4 GNU Make

GNU Make はプログラムのビルド作業を自動化するツールである．使用するコンパイラ，コンパイラオプション等をはじめとした，コンパイル，リンク，インストール等のルールを Makefile と呼ばれるテキストファイルに記述し，それに従ってビルド作業を自動的に行う．これにより，統合開発環境を用いることなくビルド作業を簡単に行うことが可能となる．

2.5 Git

Git はプログラムのソースコードなどの変更履歴を記録・追跡するための分散型バージョン管理システムの一つである．Git の特徴として，変更履歴の情報を含めたプロジェクトをローカルに保存することが可能であり，サーバに負荷をかけることなく変更履歴の解析が可能である．

2.6 GitHub

GitHub は Git レポジトリを扱っている有名な Forge サイトの一つである．個人のプロジェクトから有名なプロジェクトまで，数多くのプロジェクトを保有している，

```
;; Function function (function, funcdef_no=0, decl_uid=4114, cgraph_uid=0, symbol_order=1)

function ()
{
  int flag.0;
  int flag.0_4;

  <bb 2>:
  [sample.c:5:8] flag = 0;
  [sample.c:9:7] flag.0_4 = flag;
  [sample.c:9:6] if (flag.0_4 != 0)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 3>:
  [sample.c:11:5] do_something ();

  <bb 4>:
  [sample.c:13:1] return;

}
```

図 2.5　-fdump-tree-optimized-lineno オプションの例 (最適化レベル 0)

利用者の多い開発コミュニティである．

2.7 GitHub Archive

GitHubは，コミット，フォーク，チケットの発行，コメント等，20種類以上のイベントの情報を提供している．GitHub Archive [注2]では，GitHubのそれらのタイムラインを記録・アーカイブし，JSON形式で配布している．

（注2）：https://www.githubarchive.org/

3. 研究設問

ここでは，本実験で調査する研究設問 (RQ: Research Questions) について説明する．

本研究は Forge サイトである GitHub から，組み込み向けプロジェクトを取得する．さらに取得したプロジェクトの中から，GCC コンパイラを使用し，かつ C 言語で開発されているプロジェクトを取得する．そして，それらのプロジェクトを分析することにより，volatile 装飾子を使用していないことに起因する不具合がどのように発生するかを調査する．最後に，発見した不具合のパターンを検出する手法の提案と評価を行う．

したがって，本研究では以下に設定する研究設問について調査を行う．

RQ1 GitHub から組み込み向けプロジェクトは取得可能であるか．

RQ2 GitHub で公開されている組み込み向けプロジェクトの内，C 言語で記載されているプロジェクトの割合はどの程度であるか．

RQ3 GitHub で公開されている組み込み向けプロジェクトの内，GCC クロスコンパイラを使用しているプロジェクトはどの程度であるか．

RQ4 GitHub で公開されている組み込み向けプロジェクトの内，volatile 装飾子を使用していないことに起因する不具合に関係しうるプロジェクトはどの程度存在するか．

RQ5 volatile 装飾子を使用していないことに起因する不具合が発生するとき，どのようにコンパイラ最適化が行われているか．

RQ6 提案手法により，volatile 装飾子を使用していないことに起因する不具合を検出できるか．

4. データセット

本研究ではGitHubから組み込み向けのプロジェクトを取得し，分析を行う．組み込み向けプロジェクトであるかは，マイクロコントローラのベンダーから配布されているライブラリが各プロジェクトに含まれているかを調べることによって判別することができる．しかし，GitHubには特定の名前のファイルが含まれているプロジェクトを検索する機能があるものの最大1000件のプロジェクトしか取得することができない．そのため，本研究ではGitHub Archiveに保存されているタイムラインの情報を用いることで，組み込み向けプロジェクトの候補を取得する．

データセットの取得対象とするマイクロコントローラはSTMicroelectronics(ST)製マイクロコントローラの内STM32シリーズ，NXP Semiconductors(NXP)製マイクロコントローラの内LPC Zone，及びRenesas Electronics(Renesas)製マイクロコントローラの3種類である．Renesas製マイクロコントローラはシリーズが多岐に渡るため，一部のシリーズ名のみを後にキーワードとして用いる．

以下では取得方法の詳細を記述した上で，取得したプロジェクトに対して複数の調査を行う．また本研究では．GitHub Archiveのデータは2012年1月から2016年12月までのものを，GitHubに公開されているプロジェクトは2017年9月にcloneしたものを用いる．

4.1 取得方法

まず，GitHub Archiveから組み込み向けプロジェクトの候補を取得する．GitHub Archiveから取得したタイムラインから，以下の単語を含むプロジェクトを取得する．この時，大文字と小文字は区別しない．ただし，同一プロジェクトのcloneを抑制するために，GitHub上でforkされているプロジェクトは対象から除外する．

(a) ST製マイクロコントローラの場合

stm32

(b) NXP製マイクロコントローラの場合

nxp, lpc

(c) Renesas製マイクロコントローラの場合

r8c, r16c, r32r, rl78, rx62, rx63, rx64, rx65, rx71, sh701, sh702, sh703, sh704, sh705, sh706, sh708, sh713, sh714, sh720, sh721, sh723, sh724, sh712

次に，実際に組み込み向けのプロジェクトであるかを確認するために，キーワードに一致したプロジェクトを GitHub から clone し，各ベンダーが提供しているライブラリが含まれているかを正規表現を用いて確認する．

(a) ST 製マイクロコントローラの場合

```
".*stm32\S+_\S+\.[ch]$",
"core_cm\d\.[ch]$",
"stm32.+hal.*[ch]$",
"cmsis.*[ch]$"
```

(b) NXP 製マイクロコントローラの場合

```
"LPC\d+.+\.[ch]$",
"lpc\d+.+\.[ch]$",
"core_cm\d\.[ch]$"
```

(c) Renesas 製マイクロコントローラの場合

```
"iodefine.h"
```

4.2 結果

取得したプロジェクトのうち，各ベンダー製マイクロコントローラを使用しているプロジェクトの数を表 4.1 に記す．ST 製マイクロコントローラを使用しているプロジェクトは 4741 個と多数取得できた一方で，NXP, Renesas 製マイクロコントローラを使用しているプロジェクトはそれぞれ 256 個，45 個と限られた数しか得ることができなかった．この結果から，今回対象にした 3 社のマイクロコントローラ向けのプロジェクトにおいて，GitHub 上にある組み込み向けオープンソースプロジェクトでは ST 製マイクロコントローラのプロジェクトが最も多いと考えることができる．

表 **4.1**　取得した各ベンダー製マイコンに関するプロジェクト数

プロジェクト	プロジェクト数
ST 製マイクロコントローラに関するプロジェクト	4741
NXP 製マイクロコントローラに関するプロジェクト	256
Renesas 製マイクロコントローラに関するプロジェクト	45

4.3 分析

4.3.1 使用言語

2006 年の調査では，組み込みシステムプロジェクトの 51% が C で，30% が C++で記述されているという [9]．ここでは本データセットにおける，C++で開発されているプロジェクトの割合を調べる．

1. 手法

   各ターゲットのプロジェクトに対し，拡張子が.cpp であるファイルを含むプロジェクトの数を取得する．

2. 結果

   拡張子が.cpp であるファイルを含むプロジェクトの数を表 4.2 に記す．ただし，本データセットはベンダーが提供している C/C++のソースコードを含むプロジェクトのみを取得しているため，C 及び C++に関するプロジェクトしか取得できないことに注意しなければならない．

4.3.2 使用コンパイラ

本研究では GCC 独自の機能を用いることにより，意図しない最適化による不具合を検出する．そのため，組み込み分野において実際にどの程度 GCC が用いられているかを確認する．

1. 対象とするプロジェクト

   ビルド情報を必要とする実験のため，make コマンドによってビルド可能である，つまり Makefile を有するプロジェクトを対象とする．

2. 手法

   Makefile 内に”gcc”の文字列を含むプロジェクトの数を調べる．

3. 結果

   本データセットにおいて，Makefile を有するプロジェクト数と，Makefile 内で GCC が用いられているプロジェクト数を表 4.3 に記す．

表 4.2　拡張子が.cpp のファイルを含むプロジェクト数

ターゲット	C++プロジェクト数	割合
ST 製マイクロコントローラ	1086	0.229
NXP 製マイクロコントローラ	47	0.18
Renesas 製マイクロコントローラ	11	0.24

表 4.3　Makefile 内で GCC を用いているプロジェクト数

ターゲット	Makefile を有するプロジェクト数	GCC を用いたプロジェクト数	割合
ST	2190	1803	0.823
NXP	155	119	0.768
Renesas	29	26	0.90

4.3.3 volatile 装飾子を追加するコミットを含むプロジェクト数

ここでは，既に定義されている変数に対して volatile 装飾子を追加するようなコミットを検出する．このような変更を行うのは，以下のような場合が考えられる．

- 意図しないコンパイラ最適化による不具合を修正するため．
- 該当コミット以降の変更で，volatile 装飾子を付与した変数を，割り込み内や複数のスレッドで取り扱うように変更するため．

1. 手法

   プロジェクトの標準として指定されているブランチのコミットを一つずつ遡り，その差分を取得する．拡張子が.c または.cpp であるファイルの差分を行単位で比較し，文字列 ”volatile” および空白文字のみが追加されている行が存在するかを確認する．ただし，変数の定義部以外の場合を除外するために，該当の行でパーレンを用いているものは除外する．

2. 結果

   検出することのできた，既に定義されている変数に対して volatile 装飾子を追加するようなコミットを行ったプロジェクトの数，及びコミットの数を表 4.4 に示す．

4.4 考察

ここでは本データセットに関する考察を行う．

まず，今回取得した組み込み向けプロジェクトのデータセットには以下の制限が存在する．

- GitHub 上に存在するオープンソースプロジェクトのみを対象としている．
- GitHub Archive に記録されるタイムライン情報にマイクロコントローラの名前が記載されているプロジェクトしか取得できていない．
- 各ベンダーが配布しているライブラリを含むプロジェクトしか取得できていない．
- 公式で配布されているライブラリが C 言語で記述されているため，C 及び C++ で開発されているプロジェクトしか取得できていない．

表 4.4　volatile 装飾子を追加するコミット数及びそのプロジェクト数

ターゲット	プロジェクト数	volatile 装飾子を追加する対象コミット数
ST	380	3208
NXP	7	17
Renesas	6	139

使用されているプログラミング言語に関しては，どのベンター製マイコンのプロジェクトに関しても 2006 年の調査よりも C++で記述されたプロジェクトは少ない結果となった [9]．ただし，本実験ではフリーのオープンソースプロジェクトのみを対象とした実験を行なっていることに注意しなければならない，つまり，商用のプロジェクトと比較すると，オープンソースプロジェクトでは C++を用いた組み込み開発はまだ浸透しておらず，C による開発が主流である可能性を，本実験の結果は示唆している．

次に，コンパイラに GCC を用いているプロジェクトは，表 4.3 に示したように，全体で 82.1%，もっとも割合の低かった NXP に関するプロジェクトにおいても 76.8%と，非常に多くのプロジェクトが GCC を用いていることが分かった．ただし，対象プロジェクトがオープンソースプロジェクトの Makefile を含んだプロジェクトに限定している点に関しては注意しなければならない．商用のプロジェクトや Makefile を有さないプロジェクトでは Makefile を用いない IDE を使用していて，その IDE では GCC 以外のツールチェーンを使用している可能性が存在する．一方で，本データセット中 47.1%のプロジェクトが Makefile を有していることを考慮すると全体でも少なくとも 38.7%のプロジェクトが GCC を用いており，依然として多くのプロジェクトで GCC が用いられていると言うことができる．

最後に，volatile 装飾子を追加するコミットを行なったプロジェクトについて考察する．全体のプロジェクトのうち 7.8%が該当するコミットを行なっていることが分かった．全てのコミットが不具合を修正するために行われた変更であるとは限らないものの，組み込み向けプロジェクトにおいての volatile 装飾子そのもの，及び volatile 装飾子がないことに起因する不具合を検出することの重要性は十分に高いと考えられる．

以上より，本研究における研究設問 RQ1 から RQ4 への回答を行う．

RQ1 **GitHub から組み込み向けプロジェクトは取得可能であるか．**

取得可能である．ただし，本研究で用いた手法では，各ベンダーが配布しているライブラリを含むプロジェクトに限られる．

RQ2 **GitHub で公開されている組み込み向けプロジェクトの内，C 言語で記載されているプロジェクトの割合はどの程度であるか．**

78.5%のプロジェクトがC言語で記述されている．ただし，C及びC++のプロジェクトしか本データセットには含まれていないことに注意しなければならない．

RQ3 **GitHubで公開されている組み込み向けプロジェクトの内，GCCクロスコンパイラを使用しているプロジェクトはどの程度であるか．**

少なくとも38.7%のプロジェクトがGCCを用いている．Makefileを有するプロジェクトに限定すると，82.1%のプロジェクトがGCCを用いている．

RQ4 **GitHubで公開されている組み込み向けプロジェクトの内，volatile装飾子を使用していないことに起因する不具合に関係しうるプロジェクトはどの程度存在するか．**

少なくとも7.8%のプロジェクトがvolatile装飾子に関する不具合に関連している可能性が存在する．ただし，全てのプロジェクトに不具合が存在するとは限らない．

5. コンパイラ最適化に起因する不具合の推定

以下では volatile 装飾子を使用していないことに起因するコンパイラ最適化に関する不具合の内容を調査し，その検出手法を提案する．また，GitHub から取得したオープンソースプロジェクトをデータセットとし，提案手法の評価を行う．

5.1 組み込み開発における最適化不具合の例

volatile 装飾子が付いていないことによって発生する，組み込み開発における不具合の例を以下に挙げる．

- 割り込み完了を待つための遅延処理が無限ループになる．
- 割り込みが発生した際に変数でフラグを立て，メインループにおいてそのフラグを確認して割り込みが起きた後に処理を行おうとした．しかし，割り込みは発生するもののメインループに記載した処理が実行されない．
- レジスタには値を読み出すことで状態が変化するものが存在する．しかし，レジスタから値を読み出す処理が最適化によって削除され，状態が変化していないように見える．

このような不具合が発生するとき，実際にはどのように最適化が行われているかを実験 1 にて確認する．

5.2 データセット

以降の実験では，4 章で取得したデータセットを用いる．ただし，開発環境構築の簡略化のため，volatile 装飾子を追加するコミットを含むプロジェクトの内，取得できたプロジェクト数が 1803 個と最も多かった STMicroelectronics 製マイクロコントローラに関するプロジェクトを分析対象とする．また，その中でも Makefile を有し，GCC を使用しているプロジェクトは 213 個であり，これらのプロジェクトから無作為に選んだ合計 15 個のプロジェクトを対象に実験を行う．

実験 1 では不具合の内容の解析のために 10 個のプロジェクトを用いる．これら 10 個のプロジェクトを調査用データセットと呼ぶ．実験 2 及び実験 3 では提案手法の

評価のために，実験 1 で使用しなかった 5 個のプロジェクトを評価用データセットとして用いる．

5.3 実験 1

volatile 装飾子に関連する不具合をコンパイラが出力する中間言語で記載されたファイルから解析し，実際にどのような不具合が発生しているかを確認する．

5.3.1 手順

1. 不具合を含むソースコードの取得

   4.3.3 節で取得した，volatile 装飾子を追加する変更が適用される直前の状態のソースコードを取得する．

2. 中間言語の取得

   Makefile 内で GCC を用いているプロジェクトに対して，GCC のコンパイルオプションに-fdump-tree-optimized-lineno を追加する．さらに，最適化レベル 0(-O0) と最適化レベル 2(-O2) の，2 つの場合でコンパイルを行う．

   これによって，各レベルに応じた最適化後の処理内容を，中間言語で記載されたファイルによって確認することができる．

3. 処理内容の比較

   出力された 2 つのファイルを比較し，不具合の内容を手動で取得する．

5.3.2 結果

本実験によって 3 種類の不具合を確認することができた．各不具合の内容を以下に記す.

(不具合 A) 条件分岐の削除

条件分岐における条件式が，割り込みが起きなかった場合に真か偽に一意に定まることがある．このような場合コンパイラは無用命令除去機能により条件分岐を削除し，条件が真の場合は if ステートメント内に記述されている処理を必ず実行するように，偽の場合は削除するように最適化を行う．

(不具合 B) 遅延処理の削除

組み込み開発では通信処理完了のためなど，待ち時間を確保するために図 5.1 のように内容のないループ処理を行うことがある．しかし，実行時間を短縮しようとするコンパイラは，この処理を不要であると判断し削除する．

(不具合 C) ループ処理の条件式で使用される変数の値の変化が反映されない

ループ不変式移動により，条件式で参照している変数の値を取得する処理がループ外に移動され，割り込み処理等によってその変数の値が変化していたとしても無限ループに陥る．

図 5.2 に例を示す．本来であれば処理を繰り返すごとに ready 変数の値を pretmp_16 に代入するべきであるが，ループ処理内では ready 変数の値が変化しないとコンパイラは認識し，ループ処理の手前に代入処理を移動している．

また 10 個のプロジェクトにおける各不具合の検出数は，不具合 A が 2 個，不具合 B が 2 個，不具合 C が 1 個であった．また，複数の不具合が発生しているプロジェクトは存在しなかった．

5.4 実験 2

この実験では，コンパイラ最適化によって削除された条件分岐を検出する手法を提案し，実験 1 で発見した不具合のうち，不具合 A(条件分岐の削除)・不具合 B(遅延処理の削除) の 2 種類を検出が可能であるかを検証する．条件分岐には，C 言語における if ステートメントだけでなく，while や for のループステートメントにおける条件分岐も含まれる．

5.4.1 提案手法

1. 構文木の生成

実験 1 と同等の手順で取得した 2 つの中間言語を解析し，それぞれにおいて構文木を作成する．出力される中間言語では C 言語における for 文や while 文は全て goto 文に変換されており，限られた以下の構文のみが用いられる．

(a) Function declaration (FunctionDecl)

```
void delay( uint32_t n ){
  for(; n != 0; n--);
}
```

図 **5.1**　遅延処理の例

```
wait_until_intterupt ()
{
  unsigned char pretmp_16;

  <bb 2>:
  ready = 0;
  interrupt_start ();
  pretmp_16 = ready;

  <bb 3>:
  if (pretmp_16 == 0)
    goto <bb 3>;
  else
    goto <bb 4>;

  <bb 4>:
  return 0;

}
```

図 **5.2**　ループ不変式移動に関する最適化不具合の例

関数の宣言を行う.

(b) Var declaration(VarDecl)

変数の宣言を行う.

(c) Assignment ステートメント (AssignmentStmt)

変数への値の代入を行う.

(d) If ステートメント (IfStmt)

条件分岐を行う. 条件にマッチした場合のステートメントには必ず goto 文が記されている.

(d) Else ステートメント (ElseStmt)

if 文の条件にマッチしなかった場合の処理がステートメント内に記される. ステートメントでは必ず goto 文が呼ばれる.

(e) Goto ステートメント (GotoStmt)

次の命令を指定したラベルへ移動する.

(f) Label ステートメント (LabelStmt)

goto 文における移動先を示す.

(g) Function call (FunctionCall)

関数の呼び出しを行う.

(h) Return ステートメント (ReturnStmt)

現在の関数を終了し，戻り値があれば値を返す.

また各文には，C 言語において対応する処理が存在する場合，該当する処理の位置情報 (ファイル名，行，及びトークン番号) が記されているため，それらの情報も合わせて取得する.

2. 削除された if 文の検出

2 つの構文木から If ステートメントをそれぞれ取得する．そして，最適化レベル 0 において存在する If ステートメントのうち，最適化ありの場合に削除されているものを検出する．If ステートメントは，C 言語の処理における位置情報を用いることによって簡単に比較することができる.

5.4.2 評価指標

以下では評価指標の説明を行う.

TP: 検出された不具合のうち，実際に不具合であるものの数.

FN: 検出されなかった不具合の数.

FP: 検出された不具合のうち，実際には不具合でないものの数.

5.4.3 結果

実験結果を表 5.1 及び表 5.2 に記す.

評価用データセットにおいて，実際には不具合ではない最適化が 1 個，不具合の候補として検出された．これは，ループ展開最適化によって削除された If ステートメントであった.

5.5 実験 3

この実験では，実験 1 で発見した不具合のうち不具合 C として挙げた，ループ処理の条件式で使用される変数の値の変化が反映されない問題の検出が可能であるかを検証する.

5.5.1 提案手法

1. 実行フローを示す有向グラフの作成

   実験 2 で取得した二つの構文木からそれぞれ，各処理の実行フローを示す有向グラフを作成する．各命令文をノードとし，次に実行されるノードに対してパスを生成する.

2. 条件付きループ処理の検出

   有向グラフのノードとして存在する各 If ステートメントについて，それを始点とした閉路が存在するものを取得する.

3. 条件文に使用されている変数の検出

   閉路が存在する If ステートメントの条件式をトークナイズし，条件に用いられている変数名を取得する．ここで取得した変数名は SSA 形式で示されている.

表 5.1　不具合 A の自動検出結果

データセット	TP	FN	FP
調査用	2	0	0
評価用	1	-	1

表 5.2　不具合 B の自動検出結果

データセット	TP	FN	FP
調査用	2	0	0
評価用	1	-	0

4. 変数の更新に対応しているかの確認

閉路内に存在する各ノードを巡回し，条件に用いられている変数への代入文が全ての場合に存在するかを確認する．最適化レベル 0 の場合に代入文を含まない経路が存在せず，最適化レベル 2 に存在する場合，ループ不変式移動によって不具合が発生した可能性がある．

5.5.2 結果

実験結果を表 5.3 に記す．調査用データセットに含まれている不具合を検出することができた．一方で，評価用データセットから不具合の候補は出力されなかった．

表 **5.3**　不具合 **C** の自動検出結果

データセット	TP	FN	FP
調査用	1	0	0
評価用	0	-	0

6. 考察

6.1 実験 1

本実験では，volatile 装飾子がないことが原因で発生する不具合を 3 種類発見することができた．プログラムを実行しただけではこれらの不具合を特定することは困難である．特に不具合 C のように，分岐処理における真と偽それぞれの処理がどちらも実行される可能性がある不具合では，開発者がプログラムの挙動を観察するだけで発見することは非常に困難である．

一方で，本実験では対象としたプロジェクトが 10 個しかなく，今回発見した不具合以外にも volatile 装飾子に関連する不具合は存在する可能性が存在する．特に，レジスタへのアクセスに volatile 装飾子がない場合に不具合が発生する可能性がある．

6.2 実験 2

不具合 A，不具合 B ともに検出することができた一方で，ループ展開による for ループの展開を不具合の候補として挙げてしまう問題が存在する．しかし，繰り返し回数が一定のループの内部に記載されている処理が最適化後にも記載されている場合は不具合の候補から除外するようにするように改善を行えば，このような不具合は発生しないと考えられる．

また，本実験では評価に合計 15 個のプロジェクトしか用いていない．このため，他にも実際には不具合ではないものを候補として出力してしまう可能性や，不具合を検出できない可能性を否定できない．しかし，必要に応じて出力された中間言語ファイルを確認することで実際に不具合であるかを確認できることからも，本手法は有用であると考えられる．

6.3 実験 3

本実験では 1 つの不具合にしか手法を適用できていないため，提案手法の有用性を十分に評価することはできなかった．ただし少なくとも 1 つの不具合は発見できていることから，不具合の候補を出力することは可能であると考えられ，提案手法

が有用である可能性を示している．

6.4 研究設問への回答

以上の実験より，研究設問のうち RQ5, RQ6 への回答を行う．

RQ5 **volatile 装飾子を使用していないことに起因する不具合が発生するとき，どのようにコンパイラ最適化が行われているか．**

条件分岐の削除，遅延処理の削除，ループ処理の条件式で使用される変数の値の変化が反映されない，これら 3 つの不具合を確認した．ただし，これ以外にも volatile 装飾子がないことによって発生する不具合は存在する可能性がある．

RQ6 **提案手法により，volatile 装飾子を使用していないことに起因する不具合を検出できるか．**

本研究のみでは十分に評価できておらず，より多くのテストデータで検証する必要がある．しかし少なくとも不具合の候補を出力することは可能であり，不具合候補を可視化できる観点から本手法は有用であると言える．

7. 結言

本研究では GitHub から取得した組み込み向けプロジェクトから，volatile 装飾子がないことに起因するコンパイラ最適化による不具合に関連するプロジェクトを取得し，それらの不具合の内容を調査し，検出手法の提案と評価を行なった．実験の結果，条件分岐が削除される不具合，遅延処理が削除される不具合，ループ処理の条件式で使用される変数の値の変化が反映されない不具合の 3 種類の不具合を発見し，それらの不具合の候補を検出する手法の有用性を示した．

7.1 今後の課題

本研究では 15 個のプロジェクトしか不具合の調査に用いていないため，より多くのプロジェクトを対象とし，異なる不具合のパターンの検出や提案手法の有用性を確かめる必要がある．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学・人間科学系水野修教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] P.R. Panda, F. Catthoor, N.D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandecappelle, and P.G. Kjeldsberg, “Data and memory optimization techniques for embedded systems,” ACM Trans. Design Autom. Electr. Syst., vol.6, pp.149–206, 2001.

[2] Y. Yao, Q. Yao, P. Liu, and Z. Xiao, “Embedded software optimization for mp3 decoder implemented on risc core,” IEEE Transactions on Consumer Electronics, vol.50, pp.1244–1249, 2004.

[3] 独立行政法人 情報処理推進機構ソフトウェア・エンジニアリング・センター（編），組込みソフトウェア開発向けコーディング作法ガイド [C 言語版]，翔泳社，2007.

[4] I. Free Software Foundation, “Using the gnu compiler collection (gcc),” 2018 (accessed February 8, 2018).

[5] B. Alpern, M.N. Wegman, and F.K. Zadeck, “Detecting equality of variables in programs,” POPL, pp.1–11, 1988.

[6] A.W. Appel and J. Palsberg, Modern Compiler Implementation in Java, 2nd edition, Cambridge University Press, 2002.

[7] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” ACM Trans. Program. Lang. Syst., vol.13, pp.451–490, 1991.

[8] E. Morel and C. Renvoise, “Global optimization by suppression of partial redundancies,” Commun. ACM, vol.22, pp.96–103, 1979.

[9] M.Nahas, Bridging the gap between scheduling algorithms and scheduler implementations in time-triggeredembedded systems, PhD thesis, University of Leicester, 2009.