情報処理学会論文誌　デジタルプラクティス　Vol.3　No.1　1–10 (Jan. 2022)

レガシーシステム移行時の性能劣化を改善するリファクタリング支援手法の提案

岡田 譲二[1,2,a)]　パルヴァテ アブヘイ[3,b)]　石尾 隆[4,c)]　坂田 祐司[1,d)]　井上 克郎[2,e)]

受付日 2021年6月3日, 再受付日 2021年7月2日,
採録日 2021年8月24日

概要：長年保守し続けられてきたメインフレーム上の基幹システム（以降はレガシーシステムと呼ぶ）では，データベースから取得した結果に対して，ループや条件分岐などの制御構造（Loop Idiom）を用いて加工を行う手続き型のプログラムが数多く存在する．これらのプログラムを新しいプログラミング言語に移行（リライト）する際に実行時性能の劣化がしばしば起こる．こういった実行時性能の劣化を防ぐため，プログラムを分散処理などに書き換えるといったリファクタリングを行うことが多いが，このリファクタリングは単純ではなく工数がかかる．本稿では，新しいプログラミング言語にリライトしたプログラムを並列実行可能な形にリファクタリングする作業を支援することで，移行後のプログラムの実行時性能を改善する手法について提案する．評価実験として，2 つの実際のレガシーシステムのプログラム 7,565 本に対して提案手法を適用した．書き換え前後のプログラムに同一の入力データを与えることで振る舞いを保ったリファクタリングができていることを確認するとともに，リファクタリング前後のプログラムの実行時間の評価を行うことで，提案手法の有効性を確認した．本手法で実際のレガシーシステムのソースコード 3,529 本のリファクタリングができ，リファクタリング前と比較して実行時性能を 2 倍から 50 倍改善させることができた．

キーワード：リファクタリング，性能改善，レガシーシステム

Proposal of a Refactoring Support Method to Improve Performance Degradation after Legacy System Migration

Joji Okada[1,2,a)]　Abhay Parvate[3,b)]　Takashi Ishio[4,c)]　Yuji Sakata[1,d)]　Katsuro Inoue[2,e)]

Received: June 3, 2021, Revised: July 2, 2021,
Accepted: August 24, 2021

Abstract: In mainframe mission-critical systems that have been maintained for many years (hereafter referred to as legacy systems), there are many procedural programs that process results obtained from databases using control structures such as loops and conditional branches. When these programs are migrated to a new programming language, degradation of runtime performance often occurs. To prevent such degradation of runtime performance, programs are often rewritten for distributed processing, but this rewriting process is not simple and requires a lot of time and effort. In this research, we support the rewriting of programs that have been rewritten in a new programming language into a form that can be executed in parallel. As an evaluation experiment, we applied the proposed method to 7,565 programs of 2 real legacy systems. We confirmed that the behavior of the programs before and after the rewriting was maintained by providing the same input data to the programs. We also confirmed the effectiveness of the proposed method by evaluating the execution time of the programs before and after rewriting. By using this method, we were able to rewrite 3,529 source codes of an actual legacy system, and improved the runtime performance by 2 to 50 times compared with that before rewriting.

Keywords: refactoring, performance improvement, legacy system

1. はじめに

業務上は重要だが保守の継続が困難なメインフレーム上の基幹システム（以降はレガシーシステムと呼ぶ）がいまだに多数存在している [1], [2]．レガシーシステムもビジネス変化に対応するためには保守し続けることが必要であり，そのために新しいプログラミング言語や実行環境への移行といったシステム再構築のための活動が行われる [3]．

レガシーシステムを新しいプログラミング言語に移行する際に，メインフレームとサーバのハードウェアアーキテクチャの違いなどから，実行時性能の劣化がしばしば起こる [4], [5]．

実行時性能を改善する方法として，移行後のハードウェアスペックの増強（スケールアップ）やハードウェア台数の増強（スケールアウト）がある．スケールアップはハードウェアの調達コストやランニングコストの増大につながることと，増強できるスペックにも限界があるため，スケールアウトを選択することが多い．しかしスケールアウトはプログラム自体が並列実行に適している必要がある．

レガシーシステムの移行において単純なスケールアウトで解決できない性能劣化が起こることがある．レガシーシステムでは複数のプログラムを並列に実行することができるため，それをそのまま移行することで移行後システムでも複数プログラムの並列実行ができ，このような部分に関してはスケールアウトで性能改善を行うことができる．一方で，単一のプログラム内の処理の中身までを単純に並列実行することはできない．たとえば，データベースやファイルから取得したひとまとまりのデータを一括して処理するバッチ処理と呼ばれる処理方式では，1 つのプログラム内で大量のデータを処理するが，各データについての処理を単純に並列実行することができない．単一のプログラムにおいて実行時間が非常に長くかかる場合，そのプログラムがボトルネックとなり，複数プログラムの並列実行では全体の実行時間を短くできなくなる．

単一のプログラムの性能劣化を改善するためには，そのプログラム内の処理の中身を並列実行できるようにプログラム自体を書き換える必要があるが，この書き換えは単純ではなく工数がかかる．

[1] 株式会社 NTT データ
Koto, Tokyo 135–8671, Japan
[2] 大阪大学
Suita, Osaka 565–0871, Japan
[3] NTT データ先端技術株式会社
Chuo, Tokyo 104–0052, Japan
[4] 奈良先端科学技術大学院大学
Nara 630–0192, Japan
[a)] joji.okada@nttdata.com
[b)] abhay.parvate@intellilink.co.jp
[c)] ishio@is.naist.jp
[d)] yuji.sakata@nttdata.com
[e)] inoue@ist.osaka-u.ac.jp

本稿では，レガシーシステムを新しいプログラミング言語に移行した際の性能劣化（特に単一プログラムによる性能劣化）を解決するため，プログラム内の処理を並列実行可能な形に書き換える作業を支援する手法について提案する．本手法では，レガシーシステムのプログラムの中からパターンマッチによって並列実行可能な処理を抜粋し，書き換え方法を示す．書き換え作業者はその内容を基に抜粋された処理を並列実行可能なプログラムに書き換える．

レガシーシステムのプログラムの中から並列実行可能な処理を抜粋する基本的なアイディアについては，先行研究 [6] において発表しており，本稿ではその手法の詳細と評価実験を主に述べる．

評価実験として，2 つの実際のレガシーシステムのプログラム集合に対して提案手法を適用した．書き換え前後のプログラムに同一の入力データを与えることで振る舞いを保った書き換えができていることを確認するとともに，書き換え前後のプログラムの実行時間の評価を行うことで，提案手法の有効性を確認した．

以降，2 章では移行に伴う性能劣化問題の詳細とその解決のために行われる一般的な手法について紹介し，3 章で提案手法について述べる．4 章で評価実験の方法と結果を説明し，5 章で実験結果について考察する．6 章では関連研究につて紹介し，7 章でまとめと今後の課題を述べる．

2. 背景

2.1 レガシーシステムにおけるバッチ処理プログラム

基幹システムでは，データベースやファイルから取得したひとまとまりのデータを一括して処理するバッチ処理と呼ばれる処理方式が数多く存在する．レガシーシステムにおけるバッチ処理の入出力結果の例を**図 1** に，そのバッチ処理プログラムの例を**図 2** に示す．この処理は「入力の売上データの中から数量が 100 以上のレコードだけを抽出し，数量が 400 以上のレコードの場合は「大量 flg」に True を，そうでない場合は False を設定した結果を出力する」処理であり，このプログラムでは「売上データから 1 レコードずつ読み込み，読み込んだレコードの『数量』の値が 100 以上なら，その内容を抽出結果として出力する」ことによって実現している．

レガシーシステムのバッチ処理プログラムにおいて「ループ内でファイルを 1 レコード分読み込み，そのレコードに対する処理を行い，結果を出力する」といった実装は一般的である．このようなループを使った手続き的な実装では，レコード数だけ繰り返される各レコードに対する処理は直列に処理されることとなる．またすべてのレコードの処理を完了させるためには，各レコードの処理にかかった時間の総和分だけの時間が必要となる．

このような各レコードを直列に処理するプログラムには潜在的な性能問題がある．メインフレームはオープン系

売上データ

商品ID	売上日	数量
10001	2021/01/02	200
10002	2021/01/02	50
10002	2021/01/05	500
10004	2021/01/01	300

処理結果

商品ID	売上日	数量	大量flg
10001	2021/01/02	200	False
10002	2021/01/05	500	True
10004	2021/01/01	300	False

図 1　バッチ処理の入出力例（Filter）

Fig. 1　Example of input/output for batch processing (Filter).

数量が 100 以上の売上レコードを抽出し，
数量が 400 以上の場合「大量 flg」を True にする

```
1:  READ 売上データ
2:  PERFORM UNTIL 売上データEOF
3:    IF 売上データ.数量 >= 100 THEN
4:      MOVE 売上データ TO 抽出結果
5:      IF 売上データ.数量 >= 400 THEN
6:        MOVE True  TO 大量flg
7:      ELSE
8:        MOVE False TO 大量flg
9:      END-IF
10:     WRITE 抽出結果
11:   END-IF
12:   READ 売上データ
13: END-PERFORM.
```

図 2　バッチ処理プログラムの例（Filter）

Fig. 2　Example of a batch processing program (Filter).

サーバに比べて処理性能が高く，各レコードの処理が非常に高速に行われるため，各レコードの処理を直列処理してもその総和の時間も小さく，性能問題として顕在化しない．一方でこういったプログラムをメインフレームに比べて相対的に処理性能が劣るオープン系サーバで実行すると，各レコードの処理で時間がかかり，その総和の時間も大きくなるため性能問題として顕在化する．このような性能問題は，メインフレーム上で動くレガシーシステムのプログラムをそのままオープン系サーバで実行できるようにするリホストと呼ばれる再構築や，レガシーシステムのプログラムを機械的に別の言語に書き換えてオープン系サーバで実行するリライトと呼ばれる再構築を行った際に起こる [4], [5], [7].

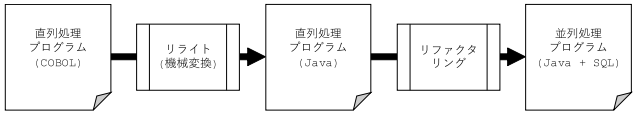

図 3　レガシーシステムプログラムの移行からバッチフレームワークの利用までの流れ

Fig. 3　Flow from migration of legacy system programs to use of a batch framework.

性能問題が起こったときに一般的に行われる性能改善方法として，処理の並列化がある．各レコードの処理を別個に並列実行することで，並列数に応じて処理にかかる時間を短くすることができる．

2.2　バッチフレームワーク

バッチ処理プログラムの並列化を簡易に行える方法として，バッチフレームワークを利用することが挙げられる．バッチフレームワークが備えるべき仕様は JSR-352 [8] で提案されており，その実装として SpringBatch[*1] や JBeret[*2] などがある．バッチフレームワークはバッチ処理に必要な基本的な機能や並列実行のための仕組みを有しており，バッチフレームワークの利用者は実現したいバッチ処理プログラム固有の処理ロジックだけを実装することで並列実行可能なバッチ処理プログラムを実現することができる．

代表的なバッチフレームワークである SpringBatch では，以下の 3 つの処理について利用者が個別に実装することでバッチ処理プログラムを実現することができる．

- Reader：加工対象のデータを読み込む処理
- Processor：読み込まれたデータ 1 レコード分を加工する処理
- Writer：加工後のデータを書き出す処理

多くの場合，Reader や Writer は SQL で記述し，Processor は Java などの手続き型プログラミング言語で記述する．

リライトにおいて性能問題が顕在化した場合，このようなバッチフレームワークを利用し，既存の直列に処理するバッチ処理プログラムを Reader/Processor/Writer の各要素に分割して実装しなおすことで，並列化実装を実現し，性能問題を解決する．**図 3** にレガシーシステムプログラムの移行からバッチフレームワークの利用までの一連の流れを示す．

図 1 を実現するバッチフレームワークでの並列実装の例を**図 4** に示す．Reader では加工対象のデータを読み込む処理として「売上データから数量が 100 以上のデータだけ

[*1] https://spring.io/projects/spring-batch
[*2] https://github.com/jberet/jsr352

■Reader

```
SELECT 商品ID, 売上日, 数量 FROM 売上データ WHERE 数量 >= 100
```

■Processor

```
1: public Sale2 process(Sale item) throws Exception {
2:   Sale2 item2 = new Sale2(item);
3:   if(item.数量 => 400){
4:     item2.大量flg = True;
5:   else{
6:     item2.大量flg = False;
7:   }
8:   return item2;
9: }
```

■Writer

```
INSERT 抽出結果 SET 商品ID = #{売上データ2.商品ID}
     売上日 = #{売上データ2.売上日}
     数量 = #{売上データ2.数量}
     大量flg = #{売上データ2.大量flg}
```

図 4　バッチフレームワークによる実装例（Filter）

Fig. 4　Example of implementation using a batch framework (Filter).

を抽出する」ことを意味する SQL の SELECT 文を記述している．Processor では読み込まれたデータ 1 レコード分を加工する処理を記述する．今回の例では入力の「数量」が 400 以上のレコードの場合は「大量 flg」に「True」を，そうでない場合は「False」を設定する処理を記述している．Writer では加工後のデータを書き出す処理として「Processor で加工後の売上データを抽出結果テーブルに書き込む」ことを意味する SQL の INSERT 文を記述している．バッチフレームワークを用いてこのように実装すると，Reader でデータが読み込まれ，Processor 部分が並列実行され，Writer でその結果が出力されるバッチ処理プログラムを実現することができる．

2.3　バッチフレームワークへの変換

このように既存の直列処理のプログラムをバッチフレームワークを使った並列化実装に書き換えること（以降はバッチ処理プログラムのリファクタリングと呼ぶ）はバッチ処理プログラムの性能問題解決に有用なものの，実際にリファクタリングを行うことは容易ではない．容易ではない理由として処理の混在の問題が挙げられる．既存の直列処理のプログラムではバッチフレームワークの Reader/Processor/Writer に相当する処理が混在して書かれており，既存のプログラムのどこがそれぞれと対応する部分なのか，対応する部分をバッチフレームワークでどのように記述すべきかを判断することが難しいためである．

図 2 のような COBOL のバッチ処理プログラムでは，バッチフレームワークの Reader に記述する「数量が 100 以上」というロジックも，Processor に記述する「数量が 400 以上」というロジックも 1 つのプログラム内に混在して記述されており，それぞれがバッチフレームのどの要素と対応しているのかはプログラムの詳細を理解しないと判断することができない．実際のレガシーシステムのバッチ処理プログラムにおいては 1 つの COBOL プログラムが数千行から数万行に及ぶこともあり，その内容を理解しながらバッチ処理プログラムのリファクタリングを行うことは非常に困難である．

3.　提案手法

本研究では，性能改善を目的としたレガシーシステムのバッチ処理プログラムからバッチフレームワークの実装へのリファクタリングを支援する手法を提案する．提案手法ではレガシーシステムのバッチ処理プログラムのリファクタリングを以下のステップで実施する．

(1) Reader 部分の抽出
(2) Reader 部分の SQL への変換
(3) Processor の書き換え
(4) Writer 部分の SQL への変換

以降の節ではまずシンプルなバッチ処理プログラムに対する各ステップの詳細を紹介し，3.5 節では発展的な内容として実際のレガシーシステムに存在する複雑なバッチ処理プログラムに対する考え方を述べる．

3.1　Reader 部分の抽出

まず機械変換されたレガシーシステムのバッチ処理プログラムのソースコードから Reader に該当する部分を抽出する．

Reader は読み込むテーブルの全レコードの中から Processor や Writer が処理する対象レコードのみを抽出する処理である．レガシーシステムのバッチ処理プログラムの実装では，ループを使って 1 行ずつ読み込み，そのレコードを出力するかどうかを分岐文を使って手続き的に記述する．このため，各レコードを処理するループ中の READ/WRITE に相当する文と，その READ/WRITE に相当する文の実行有無を決定する制御構造に着目することで Reader に該当する部分を抽出できる．

たとえば図 2 の例では，10 行目の WRITE 文によって処理されたレコードが出力されるが，この WRITE 文は 3 行目の IF 文によって実行されるかどうかが決まる．このため，3 行目の IF 文はバッチフレームワークの実装の Reader に関係するロジックであることが分かる．一方で，5 行目の IF 文は WRITE 文の実行有無には関係しておらず，Reader に関係するロジックではない．

提案手法では以下の手順で Reader 部分に該当するロジックを抽出する．

- プログラム呼び出し文を呼び出し先の内容でインライン展開する
- 各レコードを処理するループに含まれない文を削除する
- READ/WRITE 文および，その READ/WRITE 文の実行有無を決定する制御構造以外の文を削除する
- 分岐文のネストは条件式を AND で繋げて 1 つの分岐

```
PERFORM UNTIL 売上データEOF
  IF 売上データ.数量 >= 100 THEN
    WRITE 抽出結果
  END-IF
  READ 売上データ
END-PERFORM.
```

図 5　図 2 の Reader 部分に該当するロジック
Fig. 5　Logic corresponding to the Reader portion of Fig. 2.

表 1　Loop Idiom の一覧
Table 1　List of Loop Idioms.

Loop Idiom	意味
Edit	全ての行を出力する.
Filter	抽出条件にマッチした行だけを出力する.
Grouping	行をグループごとに分類しそれぞれの値を計算する. 例: min/max
Join	2 つの入力ファイルでキーが同じ行を結合して出力する.
Difference	片方のファイルから他方のファイルに属する行を取り除いた行を出力する.
Split	条件に従って入力ファイルの行を複数の出力ファイルに分割する.
Union	2 つの入力ファイルの全ての行を 1 つのファイルとして出力する.

文にまとめる

図 5 に図 2 の Reader 部分に該当するロジックを示す.

3.2　Reader 部分の SQL への変換

次に抽出した Reader 部分に該当するロジックを SQL に変換する.

一般的には Reader 部分に該当するロジックは無数にあり，それに対応する SQL も無数にある．しかし我々の予備調査の結果，実システムに存在する多くのバッチ処理プログラムの Reader 部分に該当するロジックは少数のパターンでカバーできることが分かったため，Reader 部分に該当するロジックのパターンごとに対応する SQL をあらかじめ用意しておき，その SQL に変換する.

表 1 に我々が見つけた Reader 部分に該当するロジックのパターン（以降，Loop Idiom と呼ぶ）の一覧を示す.

図 5 は Filter の Loop Idiom の例だが，別の例として**図 6** に Join の Loop Idiom の例を示す．この例では売上データに対して，売上データの商品 ID と同じ ID の商品データの情報を結合して出力する.

また各 Loop Idiom と対応する SQL への変換方法は**表 2** のとおりである.

また実際のソースコードでは，条件式に中間変数が使われている場合があるが，その場合は中間変数についてデータフロー解析を行い，関連する文を抽出し，その内容を基に SQL に変換する.

```
PERFORM UNTIL 売上データEOF
  IF 売上データ.商品ID == 商品データ.ID THEN
    WRITE 売上詳細2
    READ 売上データ
  ELSE IF 売上データ.商品ID > 商品データ.ID THEN
    READ 商品データ
  ELSE
    READ 売上データ
  END-IF.
END-PERFORM.
```

図 6　Loop idiom (Join)
Fig. 6　Loop Idiom (Join).

表 2　Loop Idiom と対応する SQL
Table 2　Loop Idioms and corresponding SQL.

Loop Idiom	SQL
Edit	SELECT * FROM [入力ファイル]
Filter	SELECT * FROM [入力ファイル] WHERE [WRITE 文を内包する分岐文の条件式]
Grouping	SELECT * FROM [入力ファイル] GROUP BY [WRITE 文を内包する分岐文の条件式中の属性]
Join	SELECT * FROM [入力ファイル 1] JOIN [入力ファイル 2] ON [WRITE 文を内包する分岐文の条件式中の属性]
Difference	SELECT FROM [入力ファイル 1] EXCEPT SELECT FROM [入力ファイル 2]
Split	複数の SELECT * FROM [入力ファイル] WHERE [WRITE 文を内包する分岐文の条件式]
Union	SELECT * FROM [入力ファイル 1] UNION SELECT * FROM [入力ファイル 2]

3.3　Processor の書き換え

次に Processor 部分の書き換えを行う．本提案手法のステップ (1)「Reader 部分の抽出」では READ/WRITE 文の実行有無を決定する制御構造に着目したが，このステップでは逆にその際に無視した分岐文や代入文に着目する．たとえば図 2 の例では，4 行目の MOVE 文（代入文）や 5 行目から 9 行目までの IF ブロックは処理対象となった行の編集処理であり，Processor に記述する内容である.

基本的には，ステップ (1) で無視した分岐文や代入文をそのまま Processor に記述するだけだが，ステップ (2)「Reader 部分の SQL への変換」で抽出した Loop Idiom が Grouping だった場合は以下のような Reader 部への記述を追加する.

(I)　単純な代入文の場合，Reader でその変数の最終レコードの値を取得する SQL を書く
(II)　変数に 1 を加算している場合（例：$num = num + 1$），Reader の SQL で属性名に COUNT を用いる
(III) 変数に別の変数の値を加算している場合（例：*total*

$= total + num$），Reader の SQL で属性名に SUM を用いる

(IV) 前回の値よりも大きい（もしくは小さい）場合のみ値を更新している場合（例：$if(prev > now)\ prev = now$），Reader の SQL で属性名に MAX（もしくは MIN）を用いる

3.4 Writer 部分の SQL への変換

最後に Writer 部分を SQL に変換する．WRITE 文で出力されるファイルに相当するテーブルへの INSERT 文を記述する．

3.5 Loop Idiom の組み合わせ

本章では 1 つのバッチ処理プログラムに Loop Idiom が 1 つだけ含まれる場合について説明したが，実際のレガシーシステムでは 1 つのバッチ処理プログラムに複数の Loop Idiom が含まれることも多い．このような場合，表 1 で示した Loop Idiom の組み合わせとしてみなすことで，Reader 部分の SQL に変換することができる．

たとえば，1 つのバッチ処理プログラム中に Filter と Join の両方が含まれていた場合，Filter に由来する WHERE 句と，Join に由来する JOIN ON 句の両方を持つ SQL とする．ただし，本プロジェクトでは予算の関係で変換は行わなかった．

4. 評価実験

実際のレガシーシステムのシステム再構築プロジェクトの一環として，レガシーシステムのプログラムに対して，提案手法を用いて Loop Idiom の抽出と対応するリファクタリング方法の提示を行い，その結果に従って人手でリファクタリングを行った．また，それぞれの種類の Loop Idiom を含むプログラムを 1 つずつ選択し，手法の適用有無による工数および性能についての比較実験を行った．

4.1 実験方法

実験に用いたソースコードは，第一著者が所属する企業で実際に保守開発を行っているレガシーシステム 2 つのものである．各システムのプロフィールを**表 3** に示す．

レガシーシステムのプログラムはすべて COBOL で記述されていたが，実験を行う前に機械変換ツールを用いて Java に変換しており，実験には Java に変換後のプログラムを利用している．

まずこれらのレガシーシステムのバッチ処理プログラムにどのような Loop Idiom がどれだけあるのかを作成したツールを用いて調査する．

次に Loop Idiom を 1 つだけ含むプログラムについては，すべて人手でリファクタリングを行う．この際，リファクタリング実施者には，リファクタリング対象のプログラムにどの Loop Idiom が存在するのかという情報とともに，**図 7** のような SQL に書き直す際に必要な情報が，プログラムのどの文に対応するのかといった情報も与える．SQL に書き直す際に必要な情報としては，Edit であれば編集内容，Filter であれば Where 句となる WRITE 文を内包する分岐文の条件式などである．実プロジェクトの予算および期間的な制約のため，Loop Idiom の組み合わせとなるプログラムに関しては，Loop Idiom の調査までは行うもののリファクタリングを行わない．

表 3　対象のレガシーシステムのプロフィール
Table 3　Profile of the target legacy systems.

システム	業種	ファイル数	LOC
A	金融	6,550	14.8M
B	保険	1,015	2.3M

#	ソースコード	Edit
1	Statement st = connection.createStatement();	
2	String sql = "SELECT 商品ID, 売上日, 数量 FROM 売上データ";	○
3	ResultSet rs = st.executeQuery(sql);	○
4	rs.next();	○
5	while(!rs.isLast()){	○
6	Sale item = toSale(rs);	編集内容
7	Sale2 item2 = new Sale2(item);	編集内容
8	if(item.数量 => 400){	編集内容
9	item2.大量flg = True;	編集内容
10	} else{	編集内容
11	item2.大量flg = False;	編集内容
12	}	編集内容
13	sql = "INSERT 抽出結果 SET 商品ID = " + item2.商品ID	○
14	+ " 売上日 = " + item2.売上日	○
15	+ " 数量 = " + item2.数量	○
16	+ " 大量flg = " + item2.大量flg;	○
17	st.executeUpdate(sql);	○
18	rs.next();	○
19	}	○

図 7　リファクタリング実施者に提示する情報例
Fig. 7　Example of information to be presented to the refactoring implementers.

リファクタリング後には，リファクタリング前の振る舞いを保持できているかを確認するため，書き換え前後のプログラムに同一の入力データを与えて同じ出力がされるかを確認するテストを行う．実際のレガシーシステムの再構築プロジェクトでは，ここまで実施したものを運用する．

その後，実プロジェクトとは別に，手法の適用有無による工数および性能についての比較実験を行う．これらの比較実験では，各 Loop Idiom を含むプログラムを 1 つずつランダムに選択し，そのプログラムだけを対象とする．

工数についての比較実験では，提案手法を用いてリファクタリング方法を示した場合と提案手法無しでリファクタリングを行う場合の工数を比較し，提案手法の工数削減効果について確認する．工数についてはリファクタリングによる書き換えの時間だけでなく，上述の振る舞いを保持で

きているかを確認するテストの時間も含む．実施者による影響を小さくするため，Loop Idiom ごとに実施者への提案手法の有無の割り当てを入れ換えて実施した．たとえば，Loop Idiom が Edit の際にリファクタリング実施者 A が提案手法有りで実施し，B が提案手法無しで実施した場合，Loop Idiom が Filter の際には A が提案手法無しで実施し，B が提案手法有りで実施するといった割り当てを行った．

性能についての比較実験では，同一のデータを用いてリファクタリング前後のプログラムを実行し，リファクタリング前後で実行時性能が変化するか確認する．リファクタリング後の実行時性能については，並列度を 1 と 8 に変えて実行時間を計測することで並列度の違いによる実行時性能についても比較する．

使用したマシンのスペックとバッチフレームワークは以下のとおりである．

- CPU：Intel Core i5 1.4 GHz
- 仮想 CPU コア数：8
- Memory：16 GB
- OS：RedHat Enterprise Linux 7.5
- バッチフレームワーク：Spring Batch 4.3.3
- DBMS：MySQL 8.0.24

リファクタリング前後の両方で，DB に格納されている各テーブルの主キーとなるカラムにインデックスを付与した．それ以外の DB チューニングはリファクタリング前後ともに実施していない．

またプログラム実行時に必要な入力データは各プログラムが必要なデータ形式に合わせて作成したダミーデータを用いた．Edit/Filter/Grouping/Split はそれぞれ 124 万レコードのデータ，Join は 124 万レコードのデータと 400 レコードからなるデータ，Difference は 124 万レコードのデータと 123 万レコードからなるデータ，Union は 124 万レコードのデータと 124 万レコードからなるデータをそれぞれ用いた．

また今回は DBMS に MySQL を採用したため，EXCEPT を利用できなかった．このため，Difference に対応する SQL では EXCEPT の代わりに「LEFT JOIN」と「IS NULL」を用いて書き換えを行った．

4.2　Loop Idiom の調査結果

レガシーシステム内に含まれる Loop Idiom の調査結果は**表 4** のとおりである．「組み合わせ」は複数の Loop Idiom が 1 つのファイルに存在していたことを意味している．

実験に用いた実システムにおいては，組み合わせでない Loop Idiom が半数近くを占めている．また Loop Idiom の種類によって出現数にばらつきがあることが分かり，どちらのシステムも Edit/Filter/Join/Split の出現数が多く，

表 4　Loop Idiom の調査結果
Table 4　Loop Idiom survey results.

Loop Idiom	A システム	B システム
Edit	1,093 (16.7%)	123 (12.1%)
Filter	809 (12.4%)	144 (14.2%)
Grouping	226 (3.5%)	9 (0.9%)
Join	473 (7.2%)	56 (5.5%)
Difference	49 (0.7%)	2 (0.2%)
Split	369 (5.6%)	119 (11.7%)
Union	57 (0.9%)	0 (0%)
組み合せ	3,474 (53.0%)	562 (55.4%)

表 5　提案手法の有無によるリファクタリング工数の差
Table 5　Difference in refactoring man-days with and without the proposed method.

Loop Idiom	手法無し (人日)	手法有り (人日)
Edit	5	3 (60%)
Filter	10	5 (50%)
Grouping	15	5 (33%)
Join	15	8 (53%)
Difference	12	5 (42%)
Split	5	5 (100%)
Union	3	3 (100%)

Difference/Union はほとんど出現しないことも分かった．また，すべてのプログラムは 7 種類の Loop Idiom またはその組み合わせで表現できた．

実プロジェクトでは 1 つの Loop Idiom だけが含まれるプログラムのすべて 3,529 本について，リファクタリングおよびテストを行った．一方で，複数の Loop Idiom が含まれる 4,036 本のプログラムについてはリファクタリングを行っていない．

4.3　提案手法によるリファクタリング工数の差

提案手法の有無によるリファクタリング工数の違いは**表 5** のとおりである．

Split や Union では提案手法の有無で工数の差は無いものの，それ以外の Loop Idiom においてはいずれも本提案手法による手法有りのほうが工数が小さくなっている．特に Filter/Grouping/Difference については工数が 50% から 66% 程度まで削減されており，工数削減の効果が大きい．

4.4　リファクタリングによる性能改善結果

リファクタリングによる性能改善の結果は**表 6** のとおりである．「実施前」は Java への機械変換がなされただけのプログラムの実行結果であり，「1 並列」および「8 並列」は提案手法を用いてリファクタリングを行ったプログラムの実行結果である．

いずれの Loop Idiom においてもリファクタリング実施後のプログラムのほうがリファクタリング前よりも実行時

表 6　性能改善結果
Table 6　Performance improvement results.

Loop Idiom	実施前 (s)	1 並列 (s)	8 並列 (s)
Edit	51.9	42.6 (1.2 倍)	10.8 (4.8 倍)
Filter	57.6	41.2 (1.4 倍)	8.9 (6.5 倍)
Grouping	46.3	2.9 (16.0 倍)	0.7 (66.1 倍)
Join	54.1	40.4 (1.3 倍)	27.3 (2.0 倍)
Difference	97.7	4.6 (21.2 倍)	1.9 (51.4 倍)
Split	59.6	45.8 (1.3 倍)	10.1 (5.9 倍)
Union	102.1	86.7 (1.2 倍)	14.6 (7.0 倍)

性能が改善されることが確認された．またいずれも並列度を 1 から 8 に上げると実行時性能が改善されることから，並列実行が効果的に働いていることが分かる．

特に Grouping と Difference については，他の Loop Idiom と比較して並列度 1 においても顕著に性能が改善されている．

5.　考察

Loop Idiom の出現数のばらつきや提案手法による工数，作業品質，性能への影響について，それぞれ考察を行った．

5.1　Loop Idiom の出現数

Loop Idiom の種類ごとに出現数にばらつきがあることについて，それぞれのシステムの保守を担当する有識者にヒアリングを行ったところ，「連続するバッチ処理では，その前半に Join を行い，その後 Edit/Filter といった処理を多数行い，最後に必要に応じて Grouping を行うといった業務が多いため，Join/Edit/Filter/Grouping が多いのは感覚と一致する」という回答を得た．いくつかのバッチ処理についてそのプログラム間の実行順を調査したところ，有識者の回答どおりの傾向があることを確認した．

5.2　提案手法による工数への影響

提案手法の有無による工数の差について，その理由をリファクタリング実施者にヒアリングを行ったところ，「本手法の適用の有無で工数が変わらなかった Split/Union は元々のプログラムの構造が単純で，プログラムの規模自体が大きくても構造把握自体は難しくなかった．一方で適用無しで工数が相対的に大きかった Grouping/Join/Difference は，構造が難しく内容を把握するのに時間がかかった．また Edit/Filter は構造の把握自体は難しくないものの，条件分岐が多い場合に出力条件と編集条件を見分けることに工数がかかるため，機械的にその見分けをしてくれる本手法は工数削減に有効だった」という回答を得た．

5.3　提案手法による作業品質への影響

本手法の有無によるリファクタリングの作業品質についても確認したところ，提案手法の適用があった場合はリファクタリング時のバグの作り込みは 1 件だけであったが，提案手法の適用が無かった場合には 8 件のバグの作り込みがあったことが分かった．この差異についてリファクタリング実施者にヒアリングを行ったところ，「本手法の適用がある場合には抽出箇所などが明確になるため，作業内容が難しくなくバグを作り込みにくい．一方で適用無しの場合はどのような Reader としてどのような SQL を選ぶべきかを考える必要があり，その選択ミスや，出力条件と編集条件の見極めの誤りなどバグを作りこむ要素が多い」という回答を得た．

5.4　提案手法による性能への影響

本手法の適用によりどの Loop Idiom においても性能が改善されたが，Loop Idiom の種類によって異なる 3 種類のボトルネックがそれぞれ改善されたためと考えられる．

1 つ目は CPU がボトルネックとなるもので，複雑な編集や抽出条件を持つ場合の Edit や Filter が該当する．図 2 の例は非常に簡単な抽出条件だが，実際のソースコードでは数千行に及ぶ条件式や計算ロジックを持っていることが多く，CPU がボトルネックになっている．このような場合は，バッチフレームワークによる Processor の並列化が効果的となる．

2 つ目のボトルネックとしては I/O ネックが挙げられる．これは Split や Union，単純な Edit や Filter が該当する．これらの Loop Idiom は処理が単純であり，CPU 部分ではなくデータの入出力を行う I/O がネックになる．このような場合は，バッチフレームワークによるチャンク化（複数件をまとめて入出力する方式）が性能改善に貢献する．

3 つ目の改善ポイントしてはインデックスを用いた計算量の変更があり，Grouping や Difference が該当する．これらの Loop Idiom は，Reader の SQL においてインデックスを張ったカラムについて集約計算や差集合を求める内容であり，MySQL の最適化により通常であれば $O(n)$ 必要な計算も，インデックスを使用することで $O(logn)$ となったためと考えられる．

6.　関連研究

Allamanis ら [9] は，ソースコード内のループに着目してパターンマイニングを行う手法を提案している．この手法では変数の読み書きを対象にしているが提案手法はファイルの読み書きを対象にしている点で異なる．また，この手法では 11 プロジェクト，577 KLOC のソースコードに対して評価を行っているが，我々の提案手法は 17.1 MLOC とより大規模なソースコードに適用しており，

また実システムのプロジェクトとして実施した点も異なる．

提案手法と同様の問題を対象にしている研究として，Wiedermann ら [10], [11] の研究がある．彼らによるとORM（Object-relational Mapping）を用いたシステムでも性能問題は起きており，その原因としてデータベースの内部で処理したほうが良いとも思われる Filter や Join などの操作をプログラマーが手続き型プログラミング言語で記述していることを指摘している．提案手法と同じく，こういったプログラムを SQL クエリに書き換えることで性能問題を解決しようとしている．Wiedermann らの手法では，提案手法における Filter や Join に関しては書き換えることができるが，提案手法でカバーしている Grouping や Split，Union など他の Loop Idiom に関しては書き換えることができない．また，手法の評価は 900 LOC と小規模なソースコードに留まっている．

同様の問題を解決する別のアプローチとして Cheung ら [12] は，基となるプログラムコードから後条件と不変条件を生成し，それらの条件を満たす SQL クエリを合成するという手法を提案している．これらの手法では，Filter や Join だけでなく，Grouping についても書き換えることができるが，基となるプログラムは ORM のようなテーブル構造を変換する操作だけである．一方で提案手法がターゲットにしているバッチ処理では，テーブル構造を変換する操作に加えて，データの値を編集する操作も 1 つのプログラムに混在して書かれている．データの値を編集する操作を後条件と不変条件だけで表現するのは不十分であり，Cheung らの手法でバッチ処理を合成することはできない．この手法は 123 KLOC のソースコードに対して評価しているが，提案手法のほうがより大規模に適用している点も異なる．

7. おわりに

本稿では，レガシーシステムのプログラムが新しいプログラミング言語に移行されたときに起こる性能劣化を改善するリファクタリング手法について提案した．また，2 つの企業のレガシーシステムのソースコードを人手で調査し，どのような Loop Idiom がどれくらい存在するかを調査し，各 Loop Idiom の種類について提案手法による支援を用いてリファクタリングを行った．

提案手法による支援を行うことでリファクタリングに必要な工数を最大 66% 削減でき，リファクタリング前後で実行時性能が最大 62 倍まで改善されることを示した．

本手法は実プロジェクトで採用されており，本手法による支援によってリファクタリングされたプログラムは実案件で運用されている．これらのことから本手法によるリファクタリング支援およびそのリファクタリングは有効であったと言える．

参考文献

[1] Bennett, K.: Legacy systems: Coping with success, *IEEE software*, Vol.12, No.1, pp.19–23 (1995).

[2] Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S. and Hage, J.: How do professionals perceive legacy systems and software modernization?, *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, pp.36–47 (2014).

[3] Bisbal, J., Lawless, D., Wu, B. and Grimson, J.: Legacy information systems: Issues and directions, *IEEE software*, Vol.16, No.5, pp.103–111 (1999).

[4] 独立行政法人情報処理推進機構：システム再構築を成功に導くユーザガイド第 2 版～ユーザとベンダで共有する再構築のリスクと対策～(2018).

[5] 独立行政法人情報処理推進機構：デジタル変革に向けた IT モダナイゼーション企画のポイント集～注意すべき 7 つの落とし穴とその対策～(2018).

[6] Okada, J., Ishio, T., Sakata, Y. and Inoue, K.: Towards Classification of Loop Idioms Automatically Extracted from Legacy Systems, *2019 IEEE 13th International Workshop on Software Clones* (*IWSC*), pp.34–35 (online), DOI: 0.1109/IWSC.2019.8665854 (2019).

[7] Suganuma, T., Yasue, T., Onodera, T. and Nakatani, T.: Performance pitfalls in large-scale java applications translated from COBOL, *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, Nashville, USA, pp.685–696 (2008).

[8] Kurz, S.: Batch Applications for the Java Platform Version 1.0.

[9] Allamanis, M., Barr, E. T., Bird, C., Devanbu, P., Marron, M. and Sutton, C.: Mining semantic loop idioms, *IEEE Transactions on Software Engineering*, Vol.44, No.7, pp.651–668 (2018).

[10] Wiedermann, B. and Cook, W. R.: Extracting queries by static analysis of transparent persistence, *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.199–210 (2007).

[11] Wiedermann, B., Ibrahim, A. and Cook, W. R.: Interprocedural query extraction for transparent persistence, *ACM Sigplan Notices*, Vol.43, No.10, pp.19–36 (2008).

[12] Cheung, A., Solar-Lezama, A. and Madden, S.: Optimizing database-backed applications with query synthesis, *ACM SIGPLAN Notices*, Vol.48, No.6, pp.3–14 (2013).

岡田 譲二（正会員）

2008 年名古屋大学大学院情報科学研究科博士前期課程修了．同年株式会社 NTT データ入社．プログラム解析技術の研究開発および現場適用に従事．

パルヴァテ アブヘイ

2012年インド・プネー大学物理学博士号を取得．2005年～2009年，プネー大学モデリングとシミュレーションセンター（CMS）で学科課程幹事．2009年～2011年インド・タタ基礎研究院（TIFR）と2011年～2013年インド・数学的科学院（IMSc）で博士研究員．2013年NTTデータ先端技術株式会社入社．入社後，主にレガシーシステム解析と改善の研究開発に従事．

石尾 隆（正会員）

2003年大阪大学大学院基礎工学研究科博士前期課程修了．2006同大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．2007大阪大学大学院情報科学研究科助教授．2017奈良先端科学技術大学院大学情報科学研究科准教授．2018同大学先端科学研究科准教授．博士（情報科学）．プログラム解析，プログラム理解に関する研究に従事．

坂田 祐司（正会員）

1996年東京大学大学院工学系研究科材料学科博士前期課程修了．同年NTTデータ通信株式会社（現，株式会社NTTデータ）入社．プログラム解析，システムモダナイゼーションに関する研究に従事．

井上 克郎（正会員）

1984年大阪大学大学院基礎工学研究科博士後期課程修了（工学博士）．同年同大学基礎工学部情報工学科助手．1984年～1986年，ハワイ大学マノア校コンピュータサイエンス学科助教授．1991年大阪大学基礎工学部助教授．1995年同学部教授．2002年より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索などのプログラム分析や再利用技術の研究に従事．本会フェロー．