情報処理学会論文誌　プログラミング　Vol.16　No.1　1–13 (Jan. 2023)

遺伝的アルゴリズムによるコンパイラ最適化の効果の見積り

千葉 雄司[1,a)]　中川 満[2]

受付日 2022年5月17日, 採録日 2022年9月10日

概要：コンパイラの開発が長期にわたるほど顕著になる問題に，最適化の効果の見積りにかかる手間がある．コンパイラの開発が長期にわたると，最適化の，費用対効果に優れた部分の開発は終ってしまい，残った問題は，解決する処理の試作が容易でなく，効果の見積りに手間のかかるものばかりになる．そうした状況下で，見積りを，試作なし，かつ，少ない手間で実現する手段の候補に，ランダムサーチや遺伝的アルゴリズムがある．ランダムサーチや遺伝的アルゴリズムでは，優れた結果を得るまでに膨大な計算量が必要になるが，昨今の計算機の能力によれば実用になりうる．そこで本論文では，商用コンパイラ CC-RL が提供する，スタックフレーム内のデータ配置の最適化に，どれだけ改善の余地があるかの見積りを題材として，ランダムサーチと遺伝的アルゴリズムの見積りにおける有用性を検証する．23 個の実用的組込アプリケーションを対象として，Ryzen 3900（3.1 GHz，12core/24thread）を搭載した PC で 100.2 時間をかけ，遺伝的アルゴリズムで見積った結果，改善により，コードサイズを相乗平均で 0.27%削減できることが分かった．

キーワード：コンパイラ，最適化，コードサイズ，組込機器

Estimation of Compiler Optimization Effect Using Genetic Algorithm

Yuji Chiba[1,a)]　Mitsuru Nakagawa[2]

Received: May 17, 2022, Accepted: September 10, 2022

Abstract: The cost for the estimation of compiler optimization is one of a problem in compiler development. Its effect gets worse as the development proceeds, because we usually develop cost-effective optimizations first. After the development of cost-effective ones, there remain only optimizations that are not easy to implement and whose effect we cannot estimate easily. Random search and genetic alorithm might help the estimation at low labor cost, but they may require huge computational cost instead, so their effectiveness may not be obvious. In order to verify the effectiveness, using random search and genetic algorithm, we estimated how much we can improve our stack frame data layout optimization implementation, which is an optimization our commercial C compiler, CC-RL, provides. Using 23 practical applications, we evaluated the genetic algorithm on a PC with Ryzen 3900 (3.1 GHz, 12 core/24 thread) for 100.2 hours to find that we can improve the optimization to reduce the code size by 0.27%.

Keywords: compiler, optimization, code size, embedded device

1. はじめに

近年，ソフトウェアへの要求の高度化，複雑化により，開発コストが増大する傾向にあり，開発コスト削減のため，開発を省力化する技術の重要度が増している．本論文では，ソフトウェアの 1 つであるコンパイラを対象に，主要な開発工程である，最適化の効果の見積りを省力化する手段を検討する．

コンパイラは，高水準な言語によって記述したソフトウェアを，低水準な言語で記述したソフトウェアに翻訳するものであり，たとえば，C 言語などのプログラミング言語で記述したソースコードを，機械語の命令列に変換する．コンパイラの開発に際しては，変換結果の改善のために，

[1] 株式会社日立製作所
Hitachi, Ltd., Kokubunji, Tokyo 185–8601, Japan
[2] ルネサスエレクトロニクス株式会社
Renesas Electronics Corporation, Kodaira, Tokyo 187–0022, Japan
[a)] yuji.chiba.pd@hitachi.com

最適化機能も開発するのが一般的である．最適化機能には数多くのバリエーションがあるが，それらの効果や開発コストは一定でない．そこで，コンパイラの開発にあたっては，まず，どういった最適化機能があり，それらの費用対効果がどれだけかの見積りが必要になるが，見積りには手間がかかる．かかる手間の大きさは，コンパイラの開発が長期にわたるほど問題になる．なぜなら，コンパイラの開発が長期にわたると，開発対象の候補として残っている最適化機能が，試作はもちろん，効果の見積りも困難なものばかりになるからである．

そこで本論文では，開発が長期にわたったコンパイラ向けに，最適化の効果の見積りを省力化する方法を検討する．まず，2 章で見積りの題材とする最適化，すなわち，スタックフレーム内のデータ配置の最適化について述べる．次に，3 章で代表的な見積りの手段を概観し，題材の見積りへの有用性について述べる．さらに，題材の見積りを省力化する手段の候補に，ランダムサーチと遺伝的アルゴリズムがあることを示す．続く 4 章では関連研究を示す．5 章では，ランダムサーチと遺伝的アルゴリズムで見積りを行うための我々の実装を示し，6 章ではランダムサーチや遺伝的アルゴリズムが見積りの有用な手段となりうるか検証した結果を示す．7 章は結論である．

2. スタックフレーム内のデータ配置の最適化

スタックフレーム内のデータ配置の最適化は，実行の高速化や，コードサイズの削減を目的とする最適化である．スタックフレームは，関数が実行中に確保し，実行終了までに解放する記憶領域であり，関数の実行に必要なデータのうち，レジスタに保持できないものを記憶する．記憶するデータが複数ある場合，スタックフレーム内のどこにどのデータを記憶するか決める必要が生じるが，データの配置は実行速度やコードサイズに影響しうる．

配置がどう影響するかは，目的機械や，コンパイラが提供する最適化に依存する．たとえば目的機械をルネサスエレクトロニクス製マイコン RL78 [1]，コンパイラを同社の RL78 マイコン向け C コンパイラ CC-RL [2] とすると，次の要因が影響する．

(A) スタックポインタをベースとするロード/ストア命令
(B) 命令 push/pop
(C) 関数の実引数
(D) cross jumping [3], [4], [5]
(E) procedural abstraction [5], [6], [7]

要因 (A) から (E) のうち，CC-RL V1.11 が提供するスタックフレーム内のデータ配置の最適化で考慮するのは要因 (A) から (C) である．そこで，要因 (D)，(E) の考慮の追加がどれだけの有効か見積ることを本論文の題材とする．本章では，まず，要因 (A) から要因 (E) と，スタックフレーム内のデータの配置がどう関係するのかを明らかにし，次に，CC-RL V1.11 におけるスタックフレーム内のデータ配置の最適化の詳細を示す．

```
1:   movw ax, [sp+0x12]
```

(a) オフセットが 8 bit に収まる場合

```
2:   movw ax, sp          ; sp から ax へ複写
3;   addw ax, 0x1234      ; 16bit の定数を加算
4:   movw hl, ax          ; ax から hl へ複写
5:   movw ax, [hl]        ; 加算後のアドレスからロード
```

(b) オフセットが 8 bit に収まらない場合

図 1　オフセットを加算した番地からのロード
Fig. 1　Reference to an address added a 16 bit value.

```
行番号:   命令                    ; 命令サイズ
   1:   movw ax, [sp+0x00]   ; 2 バイト
   2:   movw hl, ax          ; 1 バイト
```

(a) 命令 movw による実現

```
   3:   pop hl               ; 1 バイト
   4:   push hl              ; 1 バイト
```

(b) 命令 push/pop による実現

図 2　[sp+0x00] からのロードの実現
Fig. 2　An implementation of load from [sp+0x00].

2.1 スタックポインタをベースとするロード/ストア命令

RL78 マイコンはスタックポインタ（sp と略記する）をベースとする 8/16 bit のロード/ストア命令を提供するが，ベースに加算できるオフセットは符号なしの 8 bit の整定数である．したがって，オフセットが符号なしの 8 bit に収まる場合には，たとえば**図 1** (a) に示すように，1 命令でロード/ストアを実現できるが，符号なしの 8 bit に収まらない場合，図 1 (b) に示すように複数の命令が必要になる．

したがって，スタックフレームのサイズが 8 bit に収まらない場合，参照頻度の高いデータを sp の近傍に配置し，より多くの参照が 1 命令で済むよう配慮すると，実行速度やコードサイズの改善につながる．

2.2 命令 push/pop

RL78 マイコンは，スタックにデータを積む命令 push と，スタックからデータを下す命令 pop を提供する．これらの命令のオペランドには，任意の 16 bit レジスタを指定できるが，そのことを利用すると，実行速度やコードサイズを改善できる．

たとえば sp の指示先 [sp+0x00] からレジスタ hl へのロードについて考える．ロードはロード命令で実現することもできるが，RL78 マイコンはアキュムレータマシンであり，ロード先に指定できるオペランドはアキュムレータ ax のみである．したがってロード命令による実現は，**図 2** (a) に示すように，アキュムレータ ax へのロードと，レジス

```
行番号:   命令                    ; 命令サイズ
    1:    subw sp, 12             ; 2バイト
    2:    movw [sp+0x00], ax      ; 2バイト
    3;    movw ax, bc             ; 1バイト
    4:    movw [sp+0x02], ax      ; 2バイト
```

(a) 命令 push を使わない実現

```
    5:    subw sp, 8              ; 2バイト
    6:    push bc                 ; 1バイト
    7:    push ax                 ; 1バイト
```

(b) 命令 push を使う実現

図 3　仮引数の退避をともなうスタックフレームの確保
Fig. 3　Stack frame allocation with saving parameters.

タ hl への複写の 2 命令になる．sp の指示先からレジスタ hl へのロードは，図 2 (b) に示すように，命令 pop と命令 push を使っても実現できる．図 2 (b) と図 2 (a) の命令列を比較すると，命令数はどちらも 2 命令だが，図 2 (b) の方が省サイズである．また，図 2 (b) は，アキュムレータ ax を上書きしないため，アキュムレータ ax を割付可能な範囲を拡大する効果も持つ．

ただし，図 2 (b) の命令列は，図 2 (a) の命令列と違い，sp の指示先からしかロードできない．[sp+0x00] へのストアについても同様であり，したがって，最適化に命令 push や命令 pop を活用するには，参照頻度の高いデータを sp の指示先に配置しなければならない．

2.3　関数の仮引数

関数がレジスタ経由で受け取る仮引数の中には，関数の冒頭でスタックフレームに退避すべきものもある．そうした仮引数の退避は，命令 push を使ってスタックフレームの確保と同時に実施するのが効率的である．退避に命令 push を使う場合と，使わない場合の比較を**図 3** に示す．

図 3 (a)，(b) はいずれも，スタックフレームの確保と，レジスタ ax と bc 経由で受け取った仮引数の退避を実施する命令列だが，命令 push を使う図 3 (b) の方が実行速度とコードサイズの双方で優れている．図 3 (a) では，2 行目で発生するストールも実行速度の劣化の要因になっている．RL78 マイコンはベースアドレスを更新した直後にロードやストアを実行するとストールするが，命令 pop や push にともなうロードやストアではストールせず，したがって図 3 (b) ではストールしない．

命令 push を使って退避を行う場合，退避先をスタックフレームのどちらかの端から連続した領域に確保する方がよい．さもないと，退避先以外の領域を確保する命令，図 3 (b) でいえば 5 行目の命令 subw が複数必要になる．

2.4　cross jumping

cross jumping は制御フローの合流元に共通する命令を，

```
 1:   if (...){
 2:     int s0 = ···;
 3:     ···;
 4:     ax += s0;
 5:   }
 6;   else{
 7:     int s1 = ···;
 8:     ···;
 9:     ax += s1;
10:   }
```

(a) ソースコード

```
11:     ···
12:     movw bc, ax
13:     movw ax, [sp+0x08]
14:     br BB1
15:     ···
16:     movw bc, ax
17:     movw ax, [sp+0x08]
18:   BB1:
```

(b) 適用前

```
19:     ···
20:     br BB1
21:     ···
22:   BB1:
23:     movw bc, ax
24:     movw ax, [sp+0x08]
```

(c) 適用後

図 4　cross jumping
Fig. 4　cross jumping.

合流先に集約する最適化で，たとえば**図 4** (b) の命令列の 12，13 行目と 16，17 行目に共通してある命令を，制御フローの合流先の 19 行目に集約し，図 4 (c) の命令列を得る．

ここで図 4 (b) から図 4 (c) への書き替えが可能である理由は，合流元の 13 行目と 17 行目で sp に加算するオフセットが同一のためである．cross jumping の適用箇所を増やしてコードサイズを削減するには，合流元の命令列が同じになるよう，スタックフレーム内のデータの配置を最適化する必要がある．

図 4 (b) の命令列が図 4 (a) のソースコードから生じたものとすると，図 4 (b) の 13 行目で [sp+0x08] は図 4 (a) の 2 行目の変数 s0 の値を，図 4 (b) の 17 行目では図 4 (a) の 7 行目の変数 s1 の値を保持する．ここで別々の変数を，スタックフレームの同一の場所に配置できるのは，変数の生存区間が重複していないからである．

2.5　procedural abstraction

procedural abstraction はコードサイズ削減のため，プログラム中に繰返し現れる命令列を関数として切り出し，集約する最適化である．

```
 1:    movw ax, [sp+0x04]
 2:    addw ax, 10
 3:    movw [sp+0x10], ax
 4:    ···
 5:    movw ax, [sp+0x04]
 6:    addw ax, 10
 7:    movw [sp+0x10], ax
 8:    ···
 9:    movw ax, [sp+0x08]
10:    addw ax, 10
11:    movw [sp+0x14], ax
12:    ···
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
```

(a) 適用前

```
 1:    call CommonCode
 2:
 3:
 4:    ···
 5:    call CommonCode
 6:
 7:
 8:    ···
 9:    movw ax, [sp+0x08]
10:    addw ax, 10
11:    movw [sp+0x14], ax
12:    ···
13:  CommonCode:
14:    movw ax, [sp+0x08]
15:    addw ax, 10
16:    movw [sp+0x14], ax
17:    ret
18:    ···
19:
20:
21:
22:
```

(b) 一回目

```
 1:    call CommonCode
 2:
 3:
 4:    ···
 5:    call CommonCode
 6:
 7:
 8:    ···
 9:    call CommonCode2
10:
11:
12:    ···
13:  CommonCode:
14:    call CommonCode2
15:
16:
17:    ret
18:  CommonCode2:
19:    movw ax, [sp+0x0c]
20:    addw ax, 10
21:    movw [sp+0x18], ax
22:    ret
```

(c) 二回目

図 5　procedural abstraction
Fig. 5　procedural abstraction.

たとえば**図 5** (a) では，1 から 3 行目と同じ命令列が 5 から 7 行目にも現れているので，procedural abstraction を適用すると図 5 (b) の命令列を得る．図 5 (a) の 1 から 3 行目と 5 から 7 行目にあった命令列は切出しの対象となり，関数 `CommonCode` として図 5 (b) の 13 から 17 行目にあり，切出元の 1 行目と 5 行目は関数 `CommonCode` を呼ぶ命令 `call` に変化している．ここで命令 `call` は実行時に 4 バイトに戻り番地をスタックに積むので，切出後の命令列中の `sp` 相対参照のオフセットは，切出前より 4 だけ大きくなるが，大きくなると，新たな procedural abstraction の機会が生れることもある．たとえば図 5 (b) では，9 から 11 行目と同じ命令列が，集約によってできた関数の 14 から 16 行目に現れており，これらに procedural abstraction を適用して図 5 (c) の命令列を得られる．

procedural abstraction の適用範囲を広げるためには，同じ命令列が現れるよう，スタックフレーム内のデータの配置を最適化する必要がある．

なお，procedural abstraction は実行速度が重要な局面では余り有用でない．なぜなら，procedural abstraction を適用すると，切り出した関数を呼ぶ命令 `call` や，切り出した関数から戻る命令 `ret` の挿入が発生し，結果として，実行速度が低下するからである．

2.6　CC-RL における実装

CC-RL は，スタックフレーム内のデータ配置を定める際，要因 (A) から (C) を考慮して配置を最適化する．具体的には，次の規則に従って，配置を行う．

(1) まず，レジスタで受け取る仮引数の退避先を，スタックフレームの端から連続する位置に配置する
(2) 次に，仮引数以外のデータを，参照の密度[*1]の高いものから順に，スタックフレーム確保後の `sp` の指示先に可能な限り近い場所に配置する

なお，スタックフレーム内のデータ配置を定めるタイミングは，レジスタ割付の際である．定めるタイミングをレジスタ割付より前にしない理由は，どのデータをレジスタでなく，スタックフレームに記憶するかが，レジスタ割付の段階まで不明だからである．また，レジスタ割付より後にしない理由は，次の 2 つである．

(1) 2.2 節で述べた，命令 `push`/`pop` によるスタックの参照で，アキュムレータ `ax` の上書きが発生しないという情報を活用する最適化がレジスタ割付だから
(2) データの配置を決めるのに必要な，データの生存区間の情報がレジスタ割付の間なら存在するから

3.　見積りの手段

コンパイラの開発が長期にわたると，スタックフレーム内のデータ配置の最適化に要因 (D)，(E) への考慮を追加するといった，既存の最適化を改善する案件の効果を，精度良く見積る要求が高まる．既存の最適化の改善が注目の

[*1] データの静的な参照回数を，生存区間長とサイズの積で除した値．

```
1:     ...
2:     movw bc, ax
3:     movw ax, [sp+0x08]
4:     br BB1
5:     ...
6:     movw bc, ax
7:     movw ax, [sp+0x12]
8:   BB1:
```

図 6　cross jumping を適用できうる命令列
Fig. 6　An instruction sequence cross jumping might be applicable to.

対象になる理由は，開発済の効果の大きな最適化に，改善の余地がある場合，改善によって得られる効果も比較的，大きいからである．また，見積りの精度が重要になる理由は，追加で開発する最適化は，必須とは限らず，開発の費用対効果を精緻に裏付ける必要性が高いからである．こうした見積りに，次に示す，一般的な手段は必ずしも有用でない．

(a) 文献調査
(b) 他のコンパイラの調査
(c) 目的コードの調査
(d) 試作

手段 (a)，すなわち文献調査は，安価に実施できる点で有益だが，目的とする題材を扱った文献があるとは限らず，あったとしても，評価対象が調査者にとって適切とは限らない．最適化の評価結果は評価対象のアプリケーションによって大きく変化しうるので，調査者の想定しないアプリケーションによる評価結果は，あったとしても，どれだけ有益か判断し難い．

評価対象のアプリケーションは，手段 (b)，(c)，(d) なら調査者が決定できる．手段 (b) の，他のコンパイラの調査は，見積り対象の最適化を実装済の，他のコンパイラを使って評価する．ただし，該当するコンパイラがあるとは限らず，あったとしても，評価に必要な処理，たとえば評価対象の処理の実行を抑止するオプションを提供するとは限らない．

手段 (c) の，目的コードの調査は，見積り対象のコンパイラを使って生成した目的コードの中に，見積り対象の最適化を適用できる箇所がどれだけあるかの調査を通じて見積りを行う．目視での調査は，覗き穴最適化のように，特定の命令列を対象とする最適化の効果を見積る手段としては有用だが，一見しただけでは最適化の対象になるかどうか分からない命令列を対象にするのは難しい．たとえば**図 6** の命令列は 3 行目と 7 行目のロード命令で sp に加算するオフセットが違うので cross jumping の適用対象にならないが，オフセットを変更できれば適用対象になる．しかしながら変更できるかどうかの判断は，一見では難しく，ロード対象のデータの生存区間の情報を必要とする．

一見で判断できないものの調査には，判断を支援するプログラム，たとえば生存区間の情報を生成するプログラムの開発もあわせて必要になるが，プログラムを開発するなら，いっそ，手段 (d) の試作をするのも一案である．

試作は，およその見積りだけを目的としたものと，そのまま開発を進めれば正式な実装になるものの 2 種類に分類できる．たとえば前者は，要因 (D)，(E) のおよその影響を見積るだけが目的なら，生存区間の干渉を無視してデータを同じ位置に配置する実装にもできる．ここで，位置を同じにする理由は，同じであることが要因 (D)，(E) の最適化の適用条件だからである．また，生存区間の干渉を無視する理由は，実装を簡単にするためである．生存区間の干渉を無視する実装は，無視しない実装がすでにあるなら，無視しないための部分を無効にするだけで実現できる．生存区間の干渉を無視した見積りは，正確でないが，要因 (D)，(E) の影響のおよその上限を示すものとして参考になる．

より精度の高い見積りを行うには，たとえば後者の試作を行えばよいが，後者にかかるコストは小さいとは限らず，たとえば図 5 における 2 回目の集約の先読みなど，難易度の高いものでは大きくなる．試作にかけたコストは，見積りの結果，効果が小さいと分かり，試作を捨てると損失になる．損失を回避するためには，事前に，より安価な手段で効果を見積り，試作に価するか判断したい．

そこで本論文では，後者の試作の前に行う，低コストな見積りの手段として，ランダムサーチと遺伝的アルゴリズムを利用できるか検討する．スタックフレーム内のデータ配置の最適化向けに，要因 (D)，(E) への考慮を試作するのにコストがかかる理由は，中間表現からどういった特徴を，どうやって抽出し，どう配置に反映するかの設計や実装が難しいからであり，配置をランダムに定める簡単な処理に差し替えるだけなら，大きなコストをかけずに実現できる．ランダムサーチや遺伝的アルゴリズムでは，総あたりと異なり，効果の最大値は求められないが，少なくともこれだけ効果があるいうことは求められ，求めた結果を費用対効果を算定する根拠の 1 つにできる．ランダムサーチや遺伝的アルゴリズムが要求する計算量は，総あたりほどでないにしても，膨大だが，昨今の計算機の能力で実用にできる範囲に収まるか否かを，本論文で検討する．

4. 関連研究

試行を繰り返してコンパイル結果の改善を試みる研究は過去に数多くあり，たとえば，最適な命令列の探索 [8] や，適用する最適化や順序の改善 [9], [10], [11], [12]，最適なコンパイルオプションの探索 [13], [14], [15], [16], [17], [18], [19], [20] に向けた試みがあった．ただし，過去の研究は，本論文のように，ランダムサーチや遺伝的アルゴリズムの実装の容易さをコンパイラ開発に活用できるか検討したものでない．

5. 実装

ランダムサーチや遺伝的アルゴリズムは，試行を繰返して解を求めるものであり，実装にあたっては，どう試行を繰り返すかが問題になる．そこで我々は，評価対象を問わず利用可能な，繰返しの仕組みを実装し，その上に，個々の評価対象に依存する，ランダムサーチや遺伝的アルゴリズムを実装した．本章ではまず，繰返しの仕組みについて述べ，次に，評価対象である，スタックフレーム内のデータ配置の最適化向けに実装した，ランダムサーチと遺伝的アルゴリズムの詳細を示す．

5.1 繰返しの仕組み

試行を繰り返すために必要なことは，試行を開始する直前の状態，たとえば中間表現を記憶し，その状態を個々の試行に供することである．この仕組みを我々はシステムコール `fork()` によって実現する．`fork()` はプロセスを複製するものである．我々の仕組みでは，複製したプロセスに試行を命じ，結果を観測してコンパイル結果を改善する．

我々が実現した繰返しの仕組みは，コンパイルの処理の途中に挟み込んで使う．一般にコンパイルでは，**図 7** に示すように，最初にソースコードを構文解析して中間表現に変換し，レジスタ割付や cross jumping，procedural abstraction など最適化を適用し，最後にコード生成を適用して目的コードを得るが，この一連の処理の途中，試行する最適化の直前と，試行の評価が定まる地点が挟み込む先になる．スタックフレーム内のデータ配置の最適化の試行における挿入先は，当該最適化を含むレジスタ割付の直前と，試行の評価，すなわちコードサイズが定まる，コード生成の直前とした．

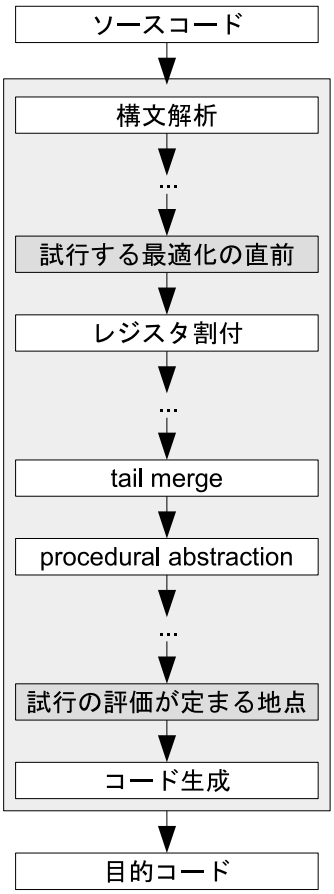

図 7　コンパイルの流れ
Fig. 7　Compilation flow.

試行する最適化の直前に挟み込む処理の詳細を**図 8** に示す．図 8 の処理は遺伝的アルゴリズム向けにもランダムサーチ向けにも使えるが，ここでは，まず，遺伝的アルゴリズム向けに使う場合について述べる．ランダムサーチ向けに使う場合との違いについては 5.2 節で述べる．

図 8 の処理は大きく 3 つの部分に分けられる．

1 つ目の部分は，1 から 10 行目で，遺伝的アルゴリズムの個体を生成する．具体的には，3 行目でプロセスの複製から情報を受け取るための領域 `sharedMemory` を確保したうえで，4 行目に進み，複製を行い，9 行目で複製の終了を待つ．複製はコンパイル処理を先にすすめて個体の生成に必要な情報を収集し，たとえばスタックフレーム内のデータ配置の最適化まで進んで，個々のスタックフレームにいくつのデータを配置するかの情報を収集し，結果を `sharedMemory` に記録して終了する．終了すると複製元が 9 行目から 10 行目に進み，`sharedMemory` に記録した情報を元に個体を生成する．

2 つ目の部分は，12 から 36 行目で，遺伝的アルゴリズムによって解を探索する．14 から 36 行目のループが遺伝的アルゴリズムの世代を進める役割を，その内側の 16 から 32 行目のループ群が，個々の個体の評価を定める役割を担う．個体の評価を定める処理が，18 行目にあるプロセスの複製である．ここで複製したプロセスは，スタックフレーム内のデータ配置の最適化について述べれば，20 行目で試行する配置を確認したうえで，コンパイルを先に進め，図 7 のレジスタ割付で配置を定めた後，cross jumping，procedural abstraction と最適化を適用し，評価の対象であるコードサイズが定まる地点，つまりコード生成の直前まで進んだら，当該箇所に挟み込んだ**図 9** の処理を行う．すなわち，到達したプロセスが複製なので，定まった評価を `sharedMemory` に記録して終了する．複製したプロセスの終了を待つ箇所は図 8 の 24 行目と 30 行目の 2 箇所にある．2 カ所にある理由は，16 から 32 行目のループ群で，同時に存在する複製の数に上限を設けるためである．

3 つ目の部分は，38 から 40 行目で，遺伝的アルゴリズムで探索した最良の解の利用に必要な処理を行う．

5.2 ランダムサーチと遺伝的アルゴリズム

スタックフレーム内のデータ配置の最適化向けに実装したランダムサーチと遺伝的アルゴリズムの詳細を示す．

```
 1:  _phase = INITIALIZE; // 解く問題を定めるフェーズに入ったことを記録
 2:  // 複製したプロセスから情報を受け取る領域を確保
 3:  void* sharedMemory = mmap(0, sharedMemorySize(), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0);
 4:  if (fork() == 0){ // プロセスを複製して
 5:    // 複製したプロセス側の処理:
 6:    // コンパイル処理を先に進め，解く問題を定めるのに必要な情報を収集し，sharedMemory に記録して exit()
 7:  }
 8:  int status;
 9:  wait(&status); // 複製したプロセスによる情報収集の終了を待つ
10:  initializePopulation(); // 初期の個体群を生成
11:
12:  _phase = TRIAL; // 試行のフェーズに移行したことを記録
13:  BitVector needsUpdate(POPULATION, true); // 個体の評価値の更新の要否．初期値はいずれも true
14:  for(unsigned epoch=0; epoch<MAX_EPOCH; ++epoch){
15:    unsigned numParallel = 0; // 並行に評価中のプロセス数
16:    for(unsigned i=0; i<POPULATION; ++i){
17:      if (needsUpdate[i]){ // 個体の評価値の更新が必要なら
18:        if (fork() == 0){ // プロセスを複製し
19:          // 複製したプロセス側の処理:
20:          setIndividual(i); // i 番目の個体を使って
21:          // コンパイル処理を先に進め，個体の評価値が確定したら sharedMemory に記録し exit()
22:        }
23:        if (++numParallel == MAX_PARALLEL){ // 複製したプロセスの数が上限に達したら
24:          wait(&statue); // 複製したプロセスの数が上限を超さないようにするため，いずれかの終了を待つ
25:          --numParallel;
26:        }
27:      }
28:    }
29:    while(numParallel){// 複製した全プロセスの終了を待つ
30:      wait(&statue);
31:      --numParallel;
32:    }
33:    if (!createNextGeneration(needsUpdate)){ // 次世代の個体を生成
34:      break; // これ以上，改善を見込めないと見切ったら試行のフェーズを終了
35:    }
36:  }
37:
38:  _phase = FINALIZE; // 本番のフェーズに移行したことを記録
39:  setBestGene(); // 評価値が最も良かった個体を使ってコンパイルを続行
40:  munmap(sharedMemory, sharedMemorySize());// 複製したプロセスから情報を受け取る領域を解放
```

図 8　試行する最適化の直前に挟み込む処理
Fig. 8　Procedure before the optimization to try.

```
1:  if (プロセスは複製){
2:    評価の結果を sharedMemory に記録する
3:    exit()
4:  }
```

図 9　試行の評価が定まる地点に挟み込む処理
Fig. 9　Procedure after evaluation.

ランダムサーチと遺伝的アルゴリズムの実装の違いは図 8 の 33 行目で呼び出す処理 `createNextGeneration()` の内容にある．具体的には，ランダムサーチでは，前の世代の個体，すなわちデータの配置順とは関係なく次の世代の個体をランダムに生成するが，遺伝的アルゴリズムでは，前の世代の個体を評価し，評価結果に基いて，次世代の個体を作ることを繰り返し，より良い個体を探索する．

我々の実装では，評価をコードサイズによって定め，ランダムサーチでも遺伝的アルゴリズムでも，評価値が最良の個体 1 つを次世代に残すこととした．

遺伝的アルゴリズムの実装では，次世代の個体の作成に際しては，個体ごとに，前世代の個体のいずれかを選択し，作成のベースとする．ここで，前世代の個体のベースとして選択する確率は，評価値によって定めるが，我々の実装

では，この確率を，前世代の最悪の評価値から個体の評価値を減じた値に正比例するものとした．ただし，前世代の評価値が全部同じ場合，全候補の確率を同一とした．

ベースとして選択した個体から，次世代の個体を作る操作は，交叉，突然変異，複製のいずれかである．ここでは，まず，個体，すなわちデータの配置順について述べ，次に，個々の操作について詳述する．

5.2.1　個体

データの配置順は，個々のデータに割り振る番号で，データの配置を何番目に行うかを示し，2.6 節に示した配置のアルゴリズムにおける優先度の代りに使う．具体的には，2.6 節に示した配置のアルゴリズムでは，データを優先度の高いものから順に配置していくが，遺伝的アルゴリズムによる実装では，個体の定めるデータの配置順に従って配置していく．ただし，要因 (B)，(C) の影響を抑えるため，命令 `push`/`pop` でアクセスできる領域に配置するデータは変更せず，仮引数の退避先をスタックフレームの端にする処理もそのままとした．

データの配置順は，スタックフレームごと，すなわち関数ごとに作成する．関数内のデータ数が n 個なら，個々のデータに 1 から n の数字を割り振り，データの配置順を，1 から n までの数字を並べ替えたものとする．個体は，コンパイル対象の関数一式のデータの配置順を保持する．

5.2.2　交叉

交叉では，ベースとする個体を 1 つ追加したうえで，関数ごとに次の処理を適用して次世代の個体を得る．

- ベースとする個体の一方から，データの配置順を，前から何番目まで引き継ぐか，ランダムに決め，引き継がない部分の配置順を，他方の個体の配置順にあわせて並べ替える

たとえば，ある関数について，ベースとする個体の一方におけるデータの配置順が 1，2，3，4，5 で，2 番目までを次世代に引き継ぐこととし，他方におけるデータの配置順が 5，4，3，2，1 なら，交叉の結果は，1，2，5，4，3 になる．

5.2.3　突然変異

突然変異では，関数を 1 つランダムに選び，選んだ関数に対応するデータを 2 つ，ランダムに選んで，配置順を交換することで，次世代の個体を得る．

5.2.4　複製

複製では，ベースとする個体をそのものを次世代の個体とする．複製の際には，個体の他に，評価値も次世代に引き継ぎ，図 8 の 33 行目で `needsUpdate` に `false` を記録し，17 行目で評価値を計算しなおさない．

6.　評価

ランダムサーチや遺伝的アルゴリズムを最適化の効果の見積りに使うことの有用性の評価を目的として，CC-RL V1.11 に，5 章で示した見積りのための実装を追加し，スタックフレーム内のデータ配置の最適化の効果を見積った．

表 1　評価環境
Table 1　Evaluation environment.

構成要素	詳細
CPU	Ryzen 3900 (3.1 GHz, 12 core/24 thread)
Memory	16 GByte (DDR4-3200)
Storage	XPG SX8100 256 GByte (NVMe PCIe Gen3)
OS	Ubuntu 21.10

表 2　遺伝的アルゴリズムのパラメタ
Table 2　Parameter for the genetic algorithm.

パラメタ	値
個体数	1,024
交叉/突然変異/複製を選択する比率	63 : 1 : 64
世代の上限	2,048
何世代連続で改善なしなら打ち切るか	256

見積る効果は，コードサイズへの影響とし，実行速度への影響とはしなかった．実行速度を除外した理由は，見積りの際に適用する最適化の 1 つである procedural abstraction が，実行速度に配慮する局面での利用に向かないことにある．なお，procedural abstraction のように，実行速度を犠牲にしてでもコードサイズを削減する最適化は，安価なマイコンにとって重要である．安価なマイコンの記憶容量は，価格に応じたものになっており，8 から 16 bit マイコン [21], [22], [23], [24], [25] で最小 0.5 から 2 KByte，32 bit マイコン [26], [27], [28], [29], [30], [31], [32] で最小 4 から 16 KByte にとどまる．そういったマイコン向けのプログラム開発では，コードサイズを小さくして容量に納めることが最も重要な課題になりうる．

見積りに使った機材は**表 1** に示すとおりである．見積りに際しては，コンパイラの最適化の目標をコードサイズの削減とするオプション `-Osize` を指定したが，中間表現の段階でマージするオプション `-merge_files` は指定しなかった．

見積りに先だち，5 章に示した実装を行い，さらに，実アプリケーションを使って遺伝的アルゴリズムのパラメタを調整した．調整の対象は**表 2** のパラメタと，交叉の実現とした．ランダムサーチのパラメタは，次世代を得る操作がランダムな個体の生成であるほかは，表 2 と同じとした．

見積りの対象は，調整に使ったものとは別に，車載/民生/Office Automation（OA と略記する）/産業の各分野からコードサイズの評価のために集めた 23 個の実アプリケーションとした．各実アプリケーションの規模を明らかにするため，コードサイズと，配置の最適化対象のデータの総数および，データの配置順のパタン数を**表 3** に示す．ここで，データの配置順のパタン数は，データの配置順を網羅するために必要になる試行の総数である．試行をファイル

表 3　スタックフレーム内のデータ配置の最適化の効果の見積り
Table 3　Estimated effect of stack frame data layout optimization.

ベンチマーク	コードサイズ（KByte）	データ総数	配置順のパタン数	干渉の無視	ランダムサーチ	遺伝的アルゴリズム				
						世代		必要世代数	計算時間	翻訳時間比（倍）
						最初	最後			
				見積った効果（%）						
産業	6	61	$5.0758 * 10^{87}$	0.00	0.00	0.00	0.00	1	53”26’	10,242
	2	0	10	0.00	0.00	0.00	0.00	1	0”05’	13
民生	58	321	$2.2149 * 10^{34}$	0.02	0.10	0.09	0.11	34	2°43”06’	5,867
	19	48	1901	0.00	0.02	0.02	0.02	1	22”08’	1,377
	30	398	$6.6443 * 10^{54}$	0.25	0.49	0.32	0.58	207	2°35”34’	13,665
	35	248	$7.7529 * 10^{32}$	0.29	0.48	0.27	0.50	371	2°04”27’	5,661
OA	31	1,159	$6.2770 * 10^{215}$	4.02	−0.10	−1.20	1.80	2,022	11°51”24’	43,467
	122	1,574	$1.3469 * 10^{484}$	0.00	−1.23	−1.56	−0.50	951	13°38”32’	9,544
	3	5	11	0.00	0.00	0.00	0.00	1	0”59’	143
	14	172	$1.1240 * 10^{21}$	0.84	0.18	0.18	0.18	1	29”46’	1,086
	23	316	$1.1013 * 10^{177}$	4.15	0.71	0.42	1.51	565	14°45”24’	1,465
車載	16	93	$1.3077 * 10^{12}$	0.00	0.00	0.00	0.00	1	15”12’	325
	16	42	$4.0642 * 10^{8}$	0.02	0.00	0.00	0.00	1	22”47’	1,789
	4	28	$1.0333 * 10^{4}$	0.02	0.14	0.14	0.14	1	7”56’	1,465
	3	0	1	0.00	0.00	0.00	0.00	1	0”04’	50
	21	104	$7.9921 * 10^{7}$	0.00	0.07	0.07	0.07	1	14”13’	185
	12	63	$1.9906 * 10^{8}$	0.05	0.03	0.03	0.03	1	27”31’	3,566
	13	143	$5.8932 * 10^{7}$	0.05	0.16	0.16	0.16	1	37”31’	1,536
	4	137	$1.2696 * 10^{73}$	0.00	0.75	−1.06	0.77	209	24”32’	4,500
	36	267	$6.2072 * 10^{23}$	0.01	0.16	0.14	0.16	4	42”30’	654
	31	182	$8.7674 * 10^{17}$	0.14	0.04	0.04	0.04	1	1°24”31’	3,586
	102	960	$3.7289 * 10^{149}$	0.16	0.26	0.13	0.45	888	33°07”42’	52,911
	152	688	$6.1181 * 10^{58}$	0.02	0.26	0.19	0.27	134	13°01”45	14,593
相乗平均				0.43	0.11	−0.72	0.27			2,374

単位で実施した都合，データの配置順のパタン数は，ファイルごとの配置順のパタン数の総和になる．ファイルごとの配置順のパタン数は，関数ごとのパタン数の総積によって，関数ごとのパタン数は，関数内にある配置の最適化対象のデータの数の階乗によって，それぞれ求めた．

表 3 には，見積った効果も示した．ここで効果は，スタックフレーム内のデータ配置の最適化の改善によって，コードサイズをどれだけ削減できるかを示す．見積りは，次の 3 つの手段で実施した．

(1) 3 章で述べた，おおよその見積りだけを目的とした試作．具体的には，遺伝的アルゴリズムで並べ替えるデータの配置先を，生存区間の干渉を無視し，関数ごとに一律にするもの
(2) 5 章で述べたランダムサーチ
(3) 5 章で述べた遺伝的アルゴリズム

遺伝的アルゴリズムで見積った効果については，最後の世代で得たものに加え，最初の世代で得たものも示した．最初の世代のものは，個体数分，すなわち 1,024 回のランダムサーチで得た見積りの結果と見なせる．

表 3 には，遺伝的アルゴリズムで最終的な見積り結果を得るのに，いくつの世代と，どれだけの時間を要したを示し，さらに，要した時間が翻訳時間の何倍かも示した．

表 4　準備と計算に要した時間
Table 4　Preparation and estimation time.

手段	実装行数	準備		計算
		実装	調整	
干渉の無視	2	3 分	-	35 秒
ランダムサーチ	328	1.6 日	-	6.9 日
遺伝的アルゴリズム	410	2 日	3 日	4.2 日

表 3 の結果をふまえ，表 3 で比較した 3 つの手段の有用性を比較する．比較の観点は次の 4 つとする．

- 準備に要する時間
- 計算に要する時間
- 見積り結果の正確さ
- 汎用性

3 つの手段の準備と計算に要した時間を**表 4** に示す．準備に要した時間については，実装に要した時間と，調整に要した時間に分けて示した．分ける理由は，かかる人手が同じでないからである．調整の対象は遺伝的アルゴリズムの表 2 のパラメタと交叉の実現だが，調整に要する時間の

大半は試行の待時間であり人手を要しない．

表 4 から，最短時間で見積る手段は，干渉の無視といえる．しかしながら干渉を無視して得た見積りはどれだけ正確か分からない．なぜなら干渉を無視したコンパイル結果は不正でありうるからである．干渉を無視した見積りを，要因 (D)，(E) の効果の上限ととらえることもできない．実際，表 3 から分かるように，ランダムサーチや遺伝的アルゴリズムの効果の方が高い場合もある．上限を示さない理由としては，たとえば図 5 の (b) から (c) への書換の考慮が十分でないことを指摘できる．

表 3 に示した見積りの結果をみれば，干渉の無視でもそれなりの精度が得られていたと分かるが，正確さが定かでない見積りのみでは，どこまで正確か確信できない．確信が必要な場合には，ランダムサーチや遺伝的アルゴリズムといった，コンパイル結果が不正でない手段の併用が有益になる．

ランダムサーチについて，表 4 に示した実装に要した時間は，推定値である．推定値である理由は，我々は遺伝的アルゴリズムの実装をベースにランダムサーチを実装したためである．実装時間の推定は，遺伝的アルゴリズムの実装に要した時間に，実装行数の比を乗じて行った．実装行数は表 4 に示してあり，コメントを含まない．

ランダムサーチは，遺伝的アルゴリズムに比べ，短い準備期間で見積りに着手可能な点で優れる．表 4 から分かるように，ランダムサーチにしたからといって，それほど実装行数は減らないが，遺伝的アルゴリズムの実現には調整を要する要因，たとえば，表 2 に示したパラメタや，交叉の実現があり，調整に時間がかかる．

ランダムサーチの問題点は，効果の見積りが正確でなかったことにある．表 3 から，見積りの相乗平均が，遺伝的アルゴリズムを使えば 0.27%になるのに対し，ランダムサーチでは 0.11%にとどまっていることが分かる．計算時間は表 4 に示したようにランダムサーチの方が長く[*2]，計算時間の違いでランダムサーチが劣るとはいえない．表 4 において準備と計算にかけた時間の総和を求めると，ランダムサーチが 8.5 日，遺伝的アルゴリズムが 9.2 日と大きな違いはなく，同じような時間をかけるなら，遺伝的アルゴリズムを使う方が正確な結果を得られた．

ただ，表 3 の遺伝的アルゴリズムの最初の世代，すなわちランダムサーチの 1,024 回の試行でコードサイズを削減できた実アプリケーションが，全 23 個の半数を超える 15 個に及んだことも指摘できる．削減幅は，最大で 0.42%だった．1,024 回の試行した際のコンパイル時間の総和は 27.8 分で，ランダムサーチや遺伝的アルゴリズムなしでのコンパイル時間の総和 0.590 分の $\frac{27.8}{0.590} = 47$ 倍にとどまった．27.8 分は，表 3 に示した遺伝的アルゴリズムの計算時間 4.2 日に比べ十分小さく，計算時間の限られた状況での実用性は，ランダムサーチが優れると考える．

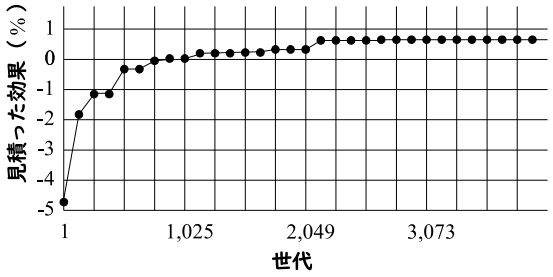

図 10　調整時の世代と見積った効果
Fig. 10　Generation and estimated effect in tuning.

遺伝的アルゴリズムは，準備や計算に時間はかかるものの，見積りの正確さ，ここでは効果の高い配置順をみつける能力に優れており，表 3 から，次のことがいえる．

- 遺伝的アルゴリズムによる見積りにより，スタックフレーム内のデータ配置の最適化の改善がもたらす効果は，最大で 1.8%，相乗平均で 0.27%に，少なくともなると分かった．
- 干渉を無視した見積りで，4%以上削減できると推定した 2 個の実アプリケーションについて，1.5%以上削減できる配置順をみつけられた．
- 遺伝的アルゴリズムが最初の世代から最後の世代までにもたらした解の改善は，最大で 3.0%であり，4 個のアプリケーションで改善が 1%を超えていた．

最も表 3 に示した遺伝的アルゴリズムによる見積りも必ずしも正確でない．その原因の 1 つは，探索を打ち切る基準を調整する際に，正確さだけでなく，計算にかかる時間も重視したことにある．

探索を打ち切る基準は，表 2 に示した世代の上限と，何世代連続で改善なしなら打ち切るかの 2 つである．何世代連続で改善なしなら打ち切るかを，本論文では連続の上限と記述する．これらの上限の調整は，次の考察に基づいて行った．

調整用の実アプリケーションを対象として，4,096 世代目まで見積るのに要した時間は 1.06 日であり，推移は**図 10** に示すとおりだった．図 10 では，見積りは 4,096 世代目までに収束するようにみえたが，同時に，23 の実アプリケーションの評価を 4,096 世代目まで行うのは時間がかかりすぎるとも判断した．判断の根拠は，アプリケーションの規模だった．調整用の実アプリケーションの規模は，コードサイズを規模とすれば，評価用の 23 の実アプリケーション全体の 1.03%にすぎない．23 の実アプリケーションの評価を 4,096 世代目まで行うのにかかる時間を，規模から推定すると，$\frac{1.06}{0.0103} = 103$ 日となる．時間がかかり過ぎるのを防ぐには，世代の上限を，たとえば図 10 において見積りを

[*2] かかる時間に違いをもたらす主な要因はコンパイルの省略の有無である．遺伝的アルゴリズムでは個体の約半数を複製によって作成し，複製した個体のコンパイルは省略するので，同じ世代数の計算なら，遺伝的アルゴリズムの方が早く終わる．

最後に改善できた2,582世代目にするのが一案だが，それではかかる時間を $\frac{2582}{4096} * 103 = 65$ 日までにしか減らせない．そこで，世代の上限を，2,582よりやや小さい2,048にするのと同時に，見積りが256世代連続して同じだったら探索を打ち切ると定めた．これらの基準を使うと，調整対象のアプリケーションでは1,340世代目で探索を打ち切ることになる．1,340世代目における見積りの結果は0.21%で，4,096世代目の0.65%より小さいが，最初の世代から最後の世代までの改善幅は，1,340世代目までで4.98%，4,096世代目までで5.37%なので，1,340世代目までの評価でも，4,096世代目までの改善幅の $\frac{4.98}{5.37} = 92.7\%$ を得られた．打ち切る世代を1,340にするだけで計算時間を十分短縮できるかは定かでなかったが，調整に使った実アプリケーションは，配置順のパタン数が $6.9868 * 10^{586}$ と，評価用の実アプリケーションのどれよりも多かったので，評価では，調整よりも早い世代で見積りが収束すると期待した．

実際の評価において，表3に示したように，世代の上限2,048に達して探索を打ち切ったアプリケーション，つまり，最終的な見積りを得るのに必要な世代数が $2{,}048 - 256 = 1{,}792$ を超えたものは1個だけだった．このことから，探索を打ち切る世代や，見積りの正確さ，計算時間に対して，より大きく影響したパラメタは，世代の上限よりも，連続の上限だったといえる．連続の上限と，見積りおよび計算時間の関係を調査した結果を**表5**に示す．調査にあたり，世代の上限は2,048のままとした．

表5から，遺伝的アルゴリズムが見積りの改善に貢献する全アプリケーションについては，連続の上限を緩和するほど，見積りが改善する傾向を観測でき，表5における緩和の上限1,024に至っても改善の余地が残っているようにみえ，さらには，見積った効果が負のままの項目も残っている．見積りの相乗平均は連続の上限が256のときは0.27%だが，1,024のときは0.32%であり，したがって256のときの見積りには，誤差が少なくとも $\frac{0.32}{0.27} - 1 = 19\%$ あるが，実際の誤差はもっと大きい．ただし連続の上限は計算時間の短縮にも貢献する．緩和の上限1,024における計算時間は319時間，およそ2週間と長い．長い時間をかければ，見積りの正確さは増すかもしれないが，長くかけすぎると，コンパイラの開発にかけられる時間が減ってしまう．時間の制約により，収束まで試行を繰り返しにくい状況では，単純に最後の世代の見積りに着目するのでなく，世代ごとの推移にも着目し，誤差の有無と大きさを推定するのが現実的と考える．

なお，計算時間を短縮する手段には，世代や連続の上限の他に，次の2つがある．

- 並列化
- 見積り対象の選定

並列化については，計算機の台数を増やすことでも実現でき，また，昨今のプロセッサのコア数の増大傾向に期待

表 5　打切りが見積りと計算時間に与える影響
Table 5　Effect of threshold on estimation and calculation time.

ベンチマーク	何世代連続で改善なしなら打ち切るか						
	16	32	64	128	256	512	1,024
	見積った効果（%）						
産業	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00
民生	0.11	0.11	0.11	0.11	0.11	0.11	0.11
	0.02	0.02	0.02	0.02	0.02	0.02	0.02
	0.47	0.50	0.50	0.50	0.53	0.53	0.61
	0.46	0.48	0.49	0.50	0.50	0.50	0.51
OA	0.97	1.20	1.50	1.66	1.80	1.90	2.33
	−0.87	−0.84	−0.59	−0.56	−0.50	−0.44	−0.24
	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.18	0.18	0.18	0.18	0.18	0.18	0.18
	1.14	1.14	1.22	1.22	1.51	1.63	1.66
車載	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.14	0.14	0.14	0.14	0.14	0.14	0.14
	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.07	0.07	0.07	0.07	0.07	0.07	0.07
	0.03	0.03	0.03	0.03	0.03	0.03	0.03
	0.16	0.16	0.16	0.16	0.16	0.16	0.16
	0.65	0.75	0.75	0.75	0.77	0.81	0.81
	0.16	0.16	0.16	0.16	0.16	0.16	0.16
	0.04	0.04	0.04	0.04	0.04	0.04	0.04
	0.36	0.38	0.40	0.44	0.45	0.47	0.47
	0.25	0.26	0.26	0.26	0.27	0.28	0.28
相乗平均	0.19	0.21	0.24	0.25	0.27	0.28	0.32
	計算時間（時間）						
総和	9.14	15.1	28.8	55.1	100.2	224	319

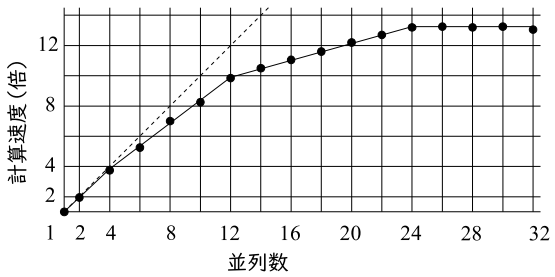

図 11　並列数と計算速度
Fig. 11　Degree of parallelism and performance.

することもできるが，実現にかかる人手の小ささは後者が優れる．我々の評価環境で，表3に示したアプリケーションの1つを対象に，並列に処理するプロセスの数と，計算速度，すなわち単一プロセスで処理する場合に比べ何倍速くなるかの関係を調べた結果を**図 11**に示す．図11から，評価にもちいたプロセッサのコア数，つまり12まで，並列数そのものとはいかないまでも速度を上げ，その後，並列に処理できるスレッドの数，つまり24まで，傾きをかえて，ほぼ線形に速度を上げる傾向を読み取れる．したがって，さらなるコア数の増加に，計算時間の短縮を期待する

余地があると推測できる．

見積り対象の選定では，たとえば，最適化が効かないアプリケーションを見積りの対象外とすることで，計算時間を短縮する．表 3 をみると，23 個のアプリケーション中，6 個については，見積りの結果が，最初と最後，どちらの世代も 0.0%であることが分かる．こうしたアプリケーションを見積りの対象外すれば，計算時間を短縮できる．ただ，6 個のアプリケーションの見積りの計算に要した時間の総和は 1.53 時間であり，全アプリケーションの見積りの計算に要した 100.2 時間の $\frac{1.53}{100.2} = 1.53\%$にすぎない．除外の対象を，最初と最後の世代の見積りの結果が同じ 13 のアプリケーションに広げたとしても，計算に要した時間全体の 5.26%にとどまる．除外にかかる人手を考えると，我々の評価では，見積り対象の選定がもたらす利得は限定的だった．

最後に，ランダムサーチや遺伝的アルゴリズムによる見積りの汎用性について述べる．これらの手法で，スタックフレーム内のデータ配置の最適化にどれだけ改善の余地があるかの見積りができた理由として，次の 2 つを指摘できる．

- 評価対象がコードサイズだった
- 実装が容易だった

評価対象がコードサイズだったことは，試行の高速化と並列化に貢献した．評価対象が実行速度の場合，目的機械がマイコンボードのような実機だと，たとえばシリアルポートの数や通信速度が制約となり，シミュレータだと，実行速度が制約となり，いずれにしても，コードサイズを評価する場合に比べ，試行回数を多くするのが難しくなる．

実装の容易さは，遺伝的アルゴリズムにとっての制約となりやすい．スタックフレーム内のデータ配置の最適化については，5 章で述べたように，データの配置順を調整するだけの問題ととらえることで，配置順を個体とする単純な実装にでき，実装規模を，表 4 に示したように，ランダムサーチと大差なくできた．しかしながら，実装が単純かは最適化対象による．たとえばレジスタ割付におけるグラフ彩色について考えると，彩色の過程で，彩色対象の生存区間の分割が発生するため，何を個体とし，どう交叉を実現するかが単純でない．単純でないものの実装に人手をかけることは，実装の目的が見積りのみである場合には難しく，他の見積り手段の方が有用になりうる．

7. 結論

スタックフレーム内のデータ配置の最適化を題材として，どれだけ改善の余地があるかの見積りを行う手段として，ランダムサーチ，遺伝的アルゴリズムをとりあげ，それぞれの有用性を検討した．23 個の実用的組込アプリケーションを対象として，遺伝的アルゴリズムで見積った結果，改善により，コードサイズを最大 1.8%，相乗平均で 0.27%削減できることが分かった．遺伝的アルゴリズムでの見積りに要した計算時間は，Ryzen 3900（3.1 GHz，12 core/24 thread）を搭載した PC で 100.2 時間だった．

参考文献

[1] Renesas Electronics Corporation: RL78 Low Power 8 & 16-bit MCUs (2021), available from ⟨https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rl78-low-power-8-16-bit-mcus⟩.

[2] Renesas Electronics Corporation: C Compiler Package for RL78 Family (2021), available from ⟨https://www.renesas.com/us/en/software-tool/c-compiler-package-rl78-family⟩.

[3] Wulf, W., Johnsson, R.K., Weindstock, C.B., Hobbs, S.O. and Geschke, C.M.: *The Design of an Optimizing Compiler* (1975).

[4] Muchnick, S.S.: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998).

[5] Debray, S.K., Evans, W., Muth, R. and De Sutter, B.: Compiler Techniques for Code Compaction, *ACM Trans. Program. Lang. Syst.*, Vol.22, No.2, pp.378–415 (online), DOI: 10.1145/349214.349233 (2000).

[6] Standish, T.A., Harriman, D.C., Kibler, D.F. and Neighbors, J.M.: The Irvine Program Transformation Catalogue, Technical Report, University of California, Irvine (1976).

[7] Fraser, C.W., Myers, E.W. and Wendt, A.L.: Analyzing and Compressing Assembly Code, *SIGPLAN Not.*, Vol.19, No.6, pp.117–121 (online), DOI: 10.1145/502949.502886 (1984).

[8] Massalin, H.: Superoptimizer: A Look at the Smallest Program, *Proc. 2nd International Conference on Architectual Support for Programming Languages and Operating Systems*, *ASPLOS II*, pp.122–126, IEEE Computer Society Press (online), DOI: 10.1145/36206.36194 (1987).

[9] Nisbet, A.: GAPS: Iterative Feedback Directed Parallelisation Using Genetic Algorithms, *Proc. Workshop on Profile and Feedback-Directed Compilation* (1998).

[10] Cooper, K.D., Schielke, P.J. and Subramanian, D.: Optimizing for Reduced Code Space Using Genetic Algorithms, *SIGPLAN Not.*, Vol.34, No.7, pp.1–9 (online), DOI: 10.1145/315253.314414 (1999).

[11] Cooper, K.D., Subramanian, D. and Torczon, L.: Adaptive Optimizing Compilers for the 21st Century, *J. Supercomput.*, Vol.23, No.1, pp.7–22 (online), DOI: 10.1023/A:1015729001611 (2002).

[12] Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L. and Waterman, T.: Finding Effective Compilation Sequences, *SIGPLAN Not.*, Vol.39, No.7, pp.231–239 (online), DOI: 10.1145/998300.997196 (2004).

[13] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: Optimizing General Purpose Compiler Optimization, *Proc. 2nd Conference on Computing Frontiers*, *CF '05*, pp.180–188, Association for Computing Machinery (online), DOI: 10.1145/1062261.1062293 (2005).

[14] Haneda, M., Knijnenburg, P.M.W. and Wijshoff, H.A.G.: Generating New General Compiler Optimization Settings, *ICS '05*, pp.161–168, Association for Computing Machinery (online), DOI: 10.1145/1088149.

1088171 (2005).
[15] Lin, S.-C., Chang, C.-K. and Lin, N.-W.: Automatic selection of GCC optimization options using a gene weighted genetic algorithm, *2008 13th Asia-Pacific Computer Systems Architecture Conference*, pp.1–8 (online), DOI: 10.1109/APCSAC.2008.4625477 (2008).
[16] Ashouri, A.H., Mariani, G., Palermo, G., Park, E., Cavazos, J. and Silvano, C.: COBAYN: Compiler Autotuning Framework Using Bayesian Networks, *ACM Trans. Archit. Code Optim.*, Vol.13, No.2 (online), DOI: 10.1145/2928270 (2016).
[17] Garciarena, U. and Santana, R.: Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions, *Proc. 2016 on Genetic and Evolutionary Computation Conference Companion*, *GECCO '16* Companion, pp.1159–1166, Association for Computing Machinery (online), DOI: 10.1145/2908961.2931696 (2016).
[18] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C. and Krishnamurthy, A.: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, *13th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 18*), pp.578–594, USENIX Association (2018).
[19] Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G. and Silvano, C.: A Survey on Compiler Autotuning Using Machine Learning, *ACM Comput. Surv.*, Vol.51, No.5 (online), DOI: 10.1145/3197978 (2018).
[20] Chen, J., Xu, N., Chen, P. and Zhang, H.: Efficient Compiler Autotuning via Bayesian Optimization, *Proc. 43rd International Conference on Software Engineering*, *ICSE '21*, pp.1198–1209, IEEE Press (online), DOI: 10.1109/ICSE43902.2021.00110 (2021).
[21] Microchip Technology Inc.: 8-bit AVR MCUs (2022), available from ⟨https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/avr-mcus⟩.
[22] Microchip Technology Inc.: 8-bit PIC MCUs (2022), available from ⟨https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/pic-mcus⟩.
[23] Microchip Technology Inc.: 8051 Microcontrollers (2022), available from ⟨https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/8051-mcus⟩.
[24] NXP Semiconductors: 8-bit S08 MCUs (2019), available from ⟨https://www.nxp.com/products/processors-and-microcontrollers/additional-mpu-mcus-architectures/8-bit-s08-mcus:HCS08⟩.
[25] STMicroelectronics: STM8 8-bit MCUs (2019), available from ⟨https://www.st.com/en/microcontrollers-microprocessors/stm8-8-bit-mcus.html⟩.
[26] Texas Instruments Incorporated.: MSP430 microcontrollers (2022), available from ⟨https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/msp430-microcontrollers/overview.html⟩.
[27] Infineon Technologies AG: 32-bit XMC1000 Industrial Microcontroller ARM Cortex-M0 (2019), available from ⟨https://www.infineon.com/cms/en/product/microcontroller/32-bit-industrial-microcontroller-based-on-arm-cortex-m/32-bit-xmc1000-industrial-microcontroller-arm-cortex-m0/⟩.
[28] Microchip Technology Inc.: SAM D Microcontrollers (2022), available from ⟨https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus/sam-32-bit-mcus/sam-d/⟩.
[29] NXP Semiconductors: General Purpose Microcontrollers (2022), available from ⟨https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus:GENERAL-PURPOSE-MCUS⟩.
[30] Silicon Laboratories: 32-bit Microcontrollers (2022), available from ⟨https://www.silabs.com/mcu/32-bit-microcontrollers⟩.
[31] STMicroelectronics: STM32 Ultra Low Power MCUs (2019), available from ⟨https://www.st.com/en/microcontrollers-microprocessors/stm32-ultra-low-power-mcus.html⟩.
[32] Renesas Electronics Corporation: RX 32-bit Performance/Effiency MCUs (2022), available from ⟨https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rx-32-bit-performance-efficiency-mcus⟩.

千葉 雄司　（正会員）

1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．株式会社日立製作所においてコンパイラの開発に従事．中央大学非常勤講師，中央大学大学院客員教授を兼任．

中川 満

1980 年生．2005 年大分大学工学部大学院工学研究科知能情報システム工学専攻博士前期過程修了．ルネサスエレクトロニクス株式会社においてコンパイラの開発に従事．