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

島崎　圭介

富山県立大学　情報システム工学科

2023 年 12 月 15 日

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.

今回, コンパイラについて最適化の効果の見積りを省力化する手段を検討する.

コンパイラの開発にあたって最適化機能には数多くのバリエーションがある. どのような最適化機能があり, 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.

また, コンパイラの開発が長期にわたるほど問題になってくる.

本研究の目的

本研究では, 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をする方法について検討する.

データ配置の最適化

背景知識

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ものである.

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. 次にどのような要因が影響するかを示す.

今回は, 目的機械をルネサンスエレクトロニクス製マイコン RL78, コンパイラを同社の RL78 マイコン向け C コンパイラ CC-RL とする.

(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
(E)procedural abstraction

(A) から (E) で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.

データ配置の最適化

ロード/ストア命令

```
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

図 1 は,(a) のように 8bit に収まると 1 命令でロード/ストアを実現できるが,(b) のように 8bit に収まらないと複数の命令が必要となる.

データ配置の最適化

ロード/ストア命令

```
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

図 2 は,8bit に収まらない場合, 参照頻度の高いデータを sp(スタックポイント) の近傍に配置すると, より多くの参照が 1 命令で済むようになると実行速度やコードサイズの改善につながる.

データ配置の最適化

命令 push/pop

push 命令とは, レジスタの値をメモリに一時的に保存するための命令でありレジスタによって様々な命令がある.

スタックにレジスタの値を保存しながら sp の値をカウントダウンしていく push down に対して, スタックから値をレジスタに読み込みながら,sp の値をカウンとアップしていく,pop up があります.

図 2 のように (a)(b) の命令はどちらも 2 命令だが,(b) のほうが省サイズである.

提案手法

関数の仮引数

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.

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

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.(※)CPU においてパイプライン処理が停止すること.

見積り

見積りの手段

一般的な手段はいくつかある.
(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

評価

(a) は, 目的とする題材を扱った文献があると限らず, 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. (b),(c) も同様にある一定のものに評価が当てはまるが, 見積りが高い精度で行われるとは限らない. 今回,(d) では費用がかかってしまう場合があるので,(d) を行う前に行う, 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.

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る. 実装にあたって, どう試行を繰り返すかが問題になる. 今回は, 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.

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, コードサイズが定まる, コード生成の直前とした. 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.

遺伝的アルゴリズムでは, 前の世代の個体を評価し, 評価結果に基づいて, 次世代の個体を作ることを繰返し, より良い個体を探索する.

今回の実装では, 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.

実装

遺伝的アルゴリズム・ランダムサーチ

　ベースと選択した個体から, 次世代を作る操作は, 交叉, 突然変異, 複製のいずれかである.
交叉ではベースとする個体を 1 つ追加したうえ, 関数ごとに処理を適用して次世代の個体を得る.
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.
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ぐ.

評価

表 1: 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

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

評価

今回, 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もった. 見積もった効果は, 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.

- 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で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.

まとめ

まとめ

- スタックフレーム内のデータ配置の最適化を行いどれだけ改善の余地があるかの見積りを行った.
- ランダムサーチ, 遺伝的アルゴリズムをとりあげてそれぞれの有用性を検討した.
- コードサイズを削減できききることが分かった.