Vol.2019-HPC-168 No.17
2019/3/6

LLVMにおけるレジスタ割付け後の改善

一場 利幸[1,a)]　津金 佳祐[1]　新井 正樹[1]　田原 司睦[1]

概要：近年，HPC 用途の ARM プロセッサが開発されており，注目が集まっている．そのため，AArch64 をターゲットとしたコンパイラの最適化機能の重要性が増している．しかし，コンパイラ基盤 LLVM の AArch64 向け最適化は，GCC に比べて不十分であることが報告されている．具体的な例の 1 つとして，LLVM は，GCC に比べて多くのスピルコードを挿入する．LLVM が生成したコードを分析すると，空いているレジスタがあるにも関わらず，スピルコードが挿入される場合があった．本研究では，LLVM で挿入される不要なスピルコードについて述べ，それらを削減する方法を提案する．これは，従来通りに LLVM のレジスタ割付けを行った後に，その結果を変更してスピルコードを削減する方法である．2 パターンの不要なスピルコードに対して，提案方法を NPB に適用した結果，パターン 1 については平均 1.25%，パターン 2 については平均 2.87%のスピルコードを削減できた．

1. はじめに

ARM プロセッサは，組込み機器を中心に広く使われているが，近年は HPC (High Performance Computing) 用途においても ARM 64 ビットアーキテクチャ（以降，AArch64 と呼ぶ）を採用したプロセッサが開発されており，注目が集まっている．HPC では，コンパイラの性能によってプログラムの実行性能が大きく変動するため，AArch64 をターゲットとしたときのコンパイラの最適化機能の重要性が増している．しかし，コンパイラ基盤 LLVM[3] の AArch64 向けの最適化は，GCC と比較した結果，不十分であることが報告されている [1]．不十分な最適化の 1 つとしてレジスタ割付けが挙げられており，LLVM で挿入されたスピルコードの数は，GCC に比べて多いことが述べられている．

レジスタ割付けとは，任意個の仮想レジスタを，プロセッサ上にある有限個の物理レジスタに割付ける処理である．レジスタ割付けの際に，物理レジスタが不足すると，レジスタとメモリとの転送命令であるスピルコードが挿入される．スピルコードが多いと実行時にメモリアクセスが発生するため，実行性能が低下する．そのため，スピルコードを減らすことがレジスタ割付けの重要な性能指標となる．

LLVM 7.0.0 で挿入されたスピルコードを詳細に分析すると，不要なスピルコードを生成している場合があることがわかった．本研究では，一度レジスタ割付けを行った結果から，不要なスピルコードを探索し削減する方法を提案する．評価のため，提案方法を NPB (NAS Parallel Benchmark) [2] に適用し，スピルコードが削減されることを示す．

本稿の構成は以下の通りである．2 章では，LLVM によるレジスタ割付け結果の解析と，不要なスピルコードが挿入されていることを述べる．3 章で，不要なスピルコードを削減する方法を提案する．4 章で，提案方法を NPB に適用した結果を考察する．5 章で，まとめと今後の課題を述べる．

2. LLVMにおけるレジスタ割付け

2.1 LLVM

LLVM[3] は，イリノイ大学により開発が始められた，オープンソースのコンパイラ基盤である．プログラムを LLVM IR と呼ばれる中間表現に変換するフロントエンド，LLVM IR に対して汎用的な解析・最適化を行うオプティマイザ，LLVM IR から特定のプロセッサ向けの最適化やコード生成を行うバックエンドからなる．C/C++に対応するフロントエンドとしては，Clang[4] が広く用いられている．オプティマイザは，LLVM IR を入出力とした解析・最適化を繰り返し適用する仕組みを持っており，新たな最適化機能を比較的実装しやすい．バックエンドで行う最適化としては，命令選択，レジスタ割付け，命令スケジューリングなどがある．LLVM のレジスタ割付けでは，fast, basic, greedy, PBQP の 4 つの実装が存在している．HPC では，実行時間と挿入されるスピルコード数のバランスの良い greedy が使われる [5]．そのため，本稿では greedy のみを対象とする．

[1] （株）富士通研究所
FUJITSU LABORATORIES, LTD.
[a)] t.ichiba@fujitsu.com

```
1  124B bb.2.for.body:
2        ; predecessors: %bb.1, %bb.2
3        successors: %bb.2, ...
4        liveins: $x1, $x3
5  140B renamable $x0 = LDRXui %stack.10, 0 :: (load 8
       from %stack.10)
6  156B renamable $x2 = ADDXri killed $x0, 1, 0
```

図 1　MIR の例

2.2　MIR とレジスタ割付け情報

LLVM のバックエンドでは，特定プロセッサ向けに最適化を繰り返し，変換を行っていく際に，Machine IR (MIR) と呼ばれる表現を用いる．レジスタ割付け後の MIR からは，各レジスタで保持するデータの生存区間を知ることができる．生存区間を知るためには，処理の初めに生存しているデータ，処理内で新たなデータが作られて生存区間が始まるデータとそのタイミング，そして，生存区間が終わるタイミングといった情報が必要である．MIR では，基本ブロックの実行開始時に生存するデータを割付けられているレジスタは liveins という項目に記述され，生存区間の終わりには killed フラグがレジスタに付与されている．また，MIR には命令ごとにアクセスするレジスタが記述されているため，処理内の新たな生存区間の始まりを知ることができる．図 1 に MIR の例を示す．この例では，基本ブロックの実行開始時に，レジスタ `x1`，`x3` には既にデータが割付けられていることが分かる．そして，最初の実行で，`x0` の生存区間が終わり，新たに `x2` が割付けられ生存区間が始まることが分かる．

MIR の情報に加え，レジスタ割付けの動作過程のログから，挿入されたスピルコードとそれらがアクセスするスタック領域の番号を知ることができる．

各レジスタの割付け状況やスピルコードがアクセスするスタック領域の番号などをまとめると，図 2 が得られる．横軸は命令の実行順であり，縦軸はレジスタ番号である．1 つの命令は，1 列に対応しており，その命令でアクセスするレジスタには READ や WRITE の頭文字が表記されている．'RW' と表記されている場合は，そのレジスタを読出して命令を実行した結果の書き込みを行う．着色された区間はデータを保持していることを表す．スピルコードであった場合にアクセスするスタックの番号をスピルインもしくはスピルアウトの種類ごとに記述している．スピルインとはデータをメモリからレジスタにロードする操作であり，スピルアウトとはデータをレジスタからメモリにストアする操作である．図 2 は，レジスタが 4 つで，12 の命令を持つ基本ブロックのレジスタ割付け結果を表示した例である．実際には，AArch64 では汎用レジスタ（GPR）を 31 個, 浮動小数点レジスタ（FPR）を 32 個持っているため，より大きな図となる．なお，図 2 の命令 0,1 のレジスタ割付け結果は，図 1 の 5,6 行目の命令に対応している．

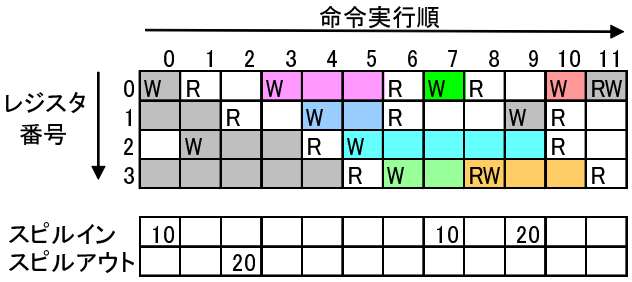

図 2　レジスタ割付け結果の表示例

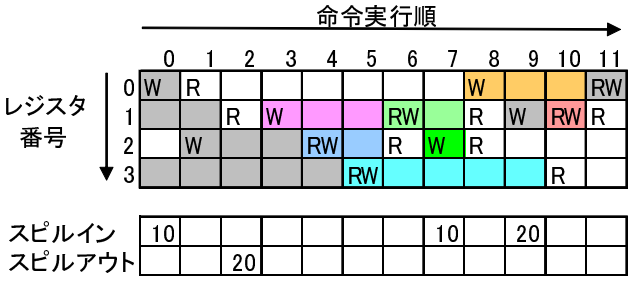

図 3　図 2 のレジスタ割付け結果を変更した例

2.3　不要なスピルコード

レジスタ割付けは，NP 完全問題であることが知られているが，コンパイルの実行時間を実用的な時間で抑える必要もあるため，最適なレジスタ割付けを行うのは困難であり [6]，不要なスピルコードが挿入されることがある．不要なスピルコードとは，データを割付けられておらず空いているレジスタが存在する状況で挿入されるスピルコードであり，レジスタ割付けを改善すれば削除できるスピルコードである．

NPB に対するレジスタ割付けの結果を詳しく解析すると，空いているレジスタがあるにも関わらず，次の 2 パターンのスピルコードが見つかった．これは不要なスピルコードである．

(1)　同じデータについてスピルインを複数回行う
(2)　スピルアウトしたデータをその後スピルインする

不要なスピルコードは，2 パターンとも図 2 に存在している．パターン 1 については，スピルインを見ると，スタック 10 について 2 回繰り返していることが分かる．命令 0 から命令 7 までの間で，連続して空いているレジスタがあれば，その間でスタック 10 からスピルインしたデータを割付けることで，命令 7 のスピルインが不要となる．パターン 2 については，スタック 20 を命令 2 でスピルアウトし，命令 9 でスピルインしている．一度メモリへ書き出した後に，メモリから読み出すという流れであり，命令 2 から命令 9 までの間に空いているレジスタがあれば，そ

のレジスタに割付けることで，スピルアウトとスピルインを削減することができる．

図 2 では，データを割付けたい区間で空いているレジスタが存在せず，2 つのパターンのどちらでもスピルコードを削除することはできない．しかし，レジスタ割付けを変更した図 3 であれば，命令 1 から命令 6 までレジスタ 0 が空いており，パターン 1 のスピルインを削除できる．具体的には，レジスタ 0 の命令 0 にあるデータの生存区間を命令 0 から命令 7 までに変更し，命令 7 のスピルコードを削除すればよい．なお，図 3 に対してパターン 2 のスピルコードを削減する場合は，命令 3 から命令 5 まであるレジスタ 1 の生存区間をレジスタ 0 に，そして，命令 6 から命令 7 まであるレジスタ 1 の生存区間をレジスタ 0 に変更すればよい．そうすると，レジスタ 1 は命令 2 から命令 8 までが空くため，スタック 20 にスピルアウト・スピルインするデータをレジスタ 1 で保持できる．

以上のように，レジスタ割付け結果を図 2 から図 3 へのように変更し，空いているレジスタを作ることができれば，スピルコードを削減できる．

なお，図 2 と図 3 では変数の生存区間が変化した部分を灰色以外で着色している．

3. スピルコードの削減方法

レジスタの割付け結果を変更することで，不要なスピルコードを削減する方法を述べる．

3.1 概要

不要なスピルコードの削減は，以下の手順で行う．

A. 不要なスピルコードの探索
B. 削減するスピルコードの選択．削減可能なスピルコードがなければ終了
C. 空きレジスタの作成を試行．空きレジスタを作れなければ，手順 B へ戻る
D. 空きレジスタへの割付けとスピルコード削減
E. 手順 B へ戻る

MIR とレジスタ割付けのログから，図 2 のように表現できるレジスタ割付け結果の情報を得る．この結果から不要なスピルコードを探索する（手順 A）．そして，探索したスピルコードのなかから 1 つを選択し（手順 B），その削減のために必要となる空きレジスタを作る（手順 C）．空きレジスタの作成では，スピルコードが存在する区間を指定し，その区間で空きレジスタの作成を試行する．空きレジスタを作ることができれば，そのレジスタでスピル対象のデータを保持し，スピルコードを削除する（手順 D）．その後は手順 B へ戻り，削減するスピルコードについて処理を繰り返す．このような流れでスピルコード削減を繰り返すと，空いているレジスタへ割付けが行われて空きレジスタが徐々に少なくなるため，削減可能なスピルコードの数も少なくなる．削減可能なスピルコードがなくなれば，処理を終える（手順 B）．また，割付けが進むと空きレジスタの作成も困難になるため，空きレジスタが作れなければ手順 B へ戻る（手順 C）．空きレジスタの作成については，詳細を次節で述べる．

本稿では，削減対象の不要なスピルコードは，その範囲が単一の基本ブロック内に限るものとする．スピルコードの範囲が複数の基本ブロックにまたがっていても削減は可能と考えられるが，ループや分岐などの制御フローを考慮した空きレジスタを作成する必要があり，アルゴリズムが複雑になる．本稿では，実装が容易な単一の基本ブロックに限って，スピルコード削減の効果があるのかどうかを示す．

3.2 空きレジスタの作成

連続した空きのあるレジスタを作る方法について述べる．

空きレジスタの作成処理を行うための入力は図 2 のようなレジスタ割付けの解析結果と，空きを作る区間 `in_range` である．出力は，作成した空きレジスタがどれかという情報と，空きレジスタを作るために必要となる入れ替え操作情報である．入れ替え操作情報は，入れ替える 2 つのレジスタと区間を，入れ替え操作の順番に並べたものである．

空きレジスタの作成は，再帰呼び出しによって実現することができる．レジスタ `regA`

空きレジスタの作成は，区間 `in_range` で空きを作ることができれば，それがどのレジスタでも良い．そのため，あるレジスタ `regA` について区間 `in_range` で空きを作る処理を実現すれば，この処理を任意のレジスタについて繰り返せば良い．そして，レジスタ `regA` について区間 `in_range` で空きを作る処理は，再帰処理で実現できる．具体的には，レジスタ `regA` が区間 `in_range` 内で保持しているデータについて，その生存区間を調べて，レジスタ `regA` のデータの生存区間ごとに空きを作る処理を行えばよい．再帰的に処理を行っていく過程で，区間 `in_range` で連続した空きのあるレジスタ `regB` が見つかれば，レジスタ `regA` とレジスタ `regB` を入れ替える．

レジスタ `regA` について区間 `in_range` で空きを作る手順は以下の通りである．

C1. レジスタ割付けの状況を保存
C2. 区間 `in_range` 内でレジスタ `regA` がデータを保持する区間 `rangeA` を探索．データ保持区間 `rangeA` は，一般には複数の区間である
C3. データ保持区間 `rangeA` が見つからなかったら，処理を終了
C4. データ保持区間 `rangeA` の中からもっとも早い区間を `rangeX` とし，区間 `rangeX` 内で空いているレジス

タ regB を選択．空いているレジスタがなければ，失敗として，処理を終了

C5. 区間 rangeX 内でレジスタ regB がデータを保持する区間 rangeB を探索．データ保持区間 rangeB は，一般には複数の区間である

C6. データ保持区間 rangeB が見つからなかったら，区間 rangeX で regA と regB を入れ替え，手順 C4 に戻る

C7. データ保持区間 rangeB のいずれかが基本ブロックの最後まで達する場合は，失敗として，処理を終了

C8. データ保持区間 rangeB の中からもっとも早い区間を rangeY とし，レジスタ regB について区間 rangeY の空きを作成（再帰呼び出し）

C9. 再帰呼び出しの結果，失敗ならレジスタ割付けの状況を復帰して処理を終了

C10. 手順 C4 に戻る

手順 C7 は，複数の基本ブロックをまたがるデータを保持するレジスタがあった場合に，そのレジスタを入れ替えないようにするために必要である．

手順 C4 での空いているレジスタの探索は，レジスタ番号が小さい順に探索すればよい．

3.3 スピルコードの削減例

レジスタ割付け結果が図 2 の場合を例に，スピルコードの削減手順を述べる．

図 2 から，不要なスピルコードを探索し（手順 A），削減するスピルコードとして，命令 0 と命令 7 にあるスピルインを選択したとする（手順 B）．このスピルインを削減するためには，レジスタ 0 で命令 1 から命令 6 までの空きを作成すればよい．レジスタ 0 で，命令 1 から命令 6 までの区間におけるデータ保持区間 rangeA は命令 3 から命令 5 までである（手順 C2）．区間 rangeA が見つかったので，手順 C3 はとばす．区間 rangeX は命令 3 から命令 5 までで，その最初の命令 3 で空いているレジスタ 1 を選択する（手順 C4）．レジスタ 1 が命令 4 から命令 5 までデータを保持することが分かると（手順 C5），その区間は基本ブロックの最後まで続かないため（手順 C7），レジスタ 1 について命令 4 から命令 5 までの空きを作成する（手順 C9）．このようにしていくと，レジスタ 2 について命令 5 から命令 9 までの空きを作成，レジスタ 3 について命令 6 から命令 7 までの空きを作成，レジスタ 0 について命令 7 の空きを作成，という順に再帰呼び出しが深くなる．レジスタ 0 の命令 7 の空きはレジスタ 1 に存在しているので，これを入れ替える．この入れ替え後は，レジスタ 3 の命令 6 から命令 7 まで保持するデータをレジスタ 0 と入れ替えができる．ここまでの入れ替えを行った状態が図 4 の (a) である．そして，レジスタ 2 について命令 5 から命令 9 までの空きを作成するため，レジスタ 3 の命令 8 から命令 9 までの空き

図 4 図 2 から図 3 に変更する過程

表 1 図 2 から図 3 に変更する入れ替え操作情報

順番	regA	regB	区間
1	0	1	[7,7]
2	3	0	[6,7]
3	0	1	[10, 10]
4	3	0	[8, 10]
5	2	3	[5,9]
6	1	2	[4,5]
7	0	1	[3,5]
8	1	2	[7,7]
9	0	1	[6,7]

を作成，と処理が続く．処理を続けていくと，レジスタの入れ替えは表 1 のような順番で行うことがわかる．入れ替えに対応したレジスタ割付け変更の様子は，図 4 に示されている．

4. 評価

NPB[2] を Clang/LLVM 7.0.0 で AArch64 向けにコンパイルし，レジスタ割付け後の MIR から，スピルコード数および 2 パターンの不要なスピルコード数を調べた結果を表 2 に示す．不要なスピルコード数は，提案方法で削減できる可能性のある数であり，削減できるスピルコードの上限値である．そのため，不要なスピルコードが見つからなかった EP などは提案手法では，スピルコードを削減でき

表 2　NPB 中のスピルコード数

	スピル	不要なスピルコード数	
	コード数	パターン 1	パターン 2
BT	2133	12	72
CG	50	2	0
EP	4	0	0
FT	56	0	0
IS	22	0	4
LU	4802	115	480
MG	775	0	0
SP	5552	5	58

表 3　NPB 中のスピルコード削減結果

	パターン 1		パターン 2	
	削減数	削減率	削減数	削減率
BT	12	0.56%	24	1.13%
CG	2	4.00%	0	0%
IS	0	0%	2	9.09%
LU	17	0.35%	50	1.04%
SP	5	0.09%	12	0.22%

ない．

提案手法を適用した結果，削減できたスピルコード数と，その削減率を表 3 に示す．削減率は，プログラム中に存在するスピルコードの中から削減できたスピルコードの割合である．LU は不要なスピルコードが多く見つかったが，空きレジスタが不足したために実際に削減できたスピルコードは多くない．

不要なスピルコードを削減できない場合については，空いているレジスタ数が不足する以外に，空きレジスタの作成に失敗することがあった．その要因について詳しく分析できていないが，空きレジスタを探して入れ替えるという 3.2 節のアルゴリズムでは対応できないケースがある．例えば，図 5 のような割付け結果だった場合である．3.2 節のアルゴリズムでは，レジスタ 0 について命令 1 から命令 6 までの空きレジスタを作りたい場合は，再帰呼び出しを繰り返すと，レジスタ 3 について命令 8 から命令 11 までの空きを作成する処理を行う必要がある．しかし，このとき，命令 10 の時点で空いているレジスタが見つからず，空きレジスタの作成に失敗する．ところが，レジスタ割付け結果を図 5 から図 6 へ変更できれば，レジスタ 0 について命令 1 から命令 6 までの空きレジスタができる．このような場合でも，空きレジスタを作成できるようアルゴリズムを改善することは今後の課題である．

5. おわりに

LLVM のレジスタ割付け結果について，不要なスピルコードが発生しており，不要なスピルコードは 2 パターンあることを述べた．これらを削減するため，従来通り LLVM のレジスタ割付けを行った後に，その結果を変更して改善する方法を提案した．提案方法を NPB に適用した結果，不要なスピルコードのパターン 1 については平均 1.25%，パターン 2 については平均 2.87%のスピルコードを削減できることを示した．

今後の課題として，より多くのスピルコード削減のため，割付け結果の変更アルゴリズムの改善，レジスタ割付けの変更を基本ブロック内だけでなく関数などのより大きな処理単位で行えるように拡張することが挙げられる．

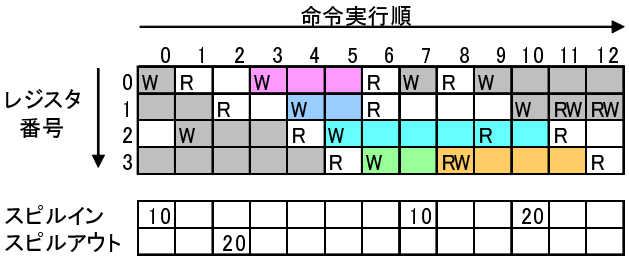

図 5　空きレジスタ作成に失敗する例

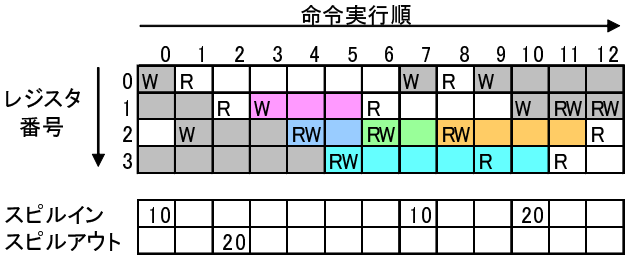

図 6　図 5 のレジスタ割付け結果を変更した例

参考文献

[1] Masaki Arai, “HKG18-506 - HCQC : HPC Compiler Quality Checker, ” Linaro Connect Hong Kong (2018), https://connect.linaro.org/resources/hkg18/hkg18-506/.

[2] University of Versailles Saint Quentin en Yvelines: NAS Parallel Benchmarks 3.0 Unofficial OpenMP C Version (2014), https://github.com/benchmark-subsetting/NPB3.0-omp-C.

[3] The LLVM Compiler Infrastructure Project, http://llvm.org/.

[4] Clang: a C language family frontend for LLVM, http://clang.llvm.org/.

[5] T. C. d. S. Xavier, G. S. Oliveira, E. D. d. Lima and A. F. d. Silva, ”A Detailed Analysis of the LLVM’s Register Allocators,” 31st International Conference of the Chilean Computer Science Society, pp. 190-198 (2012).

[6] Goodwin, David W., and Kent D. Wilken. ”Optimal and Near - optimal Global Register Allocation Using 0–1 Integer Programming,” Software: Practice and Experience 26.8, pp. 929-965 (1996).