実用プログラムの STRAIGHT アーキテクチャにおける特性の解析

小泉 透[1,a)]　杉田 脩[1]　塩谷 亮太[1]　入江 英嗣[1]　坂井 修一[1]

概要：STRAIGHT は，偽の依存を生じないような命令セットアーキテクチャである．偽の依存が生じないため，レジスタリネームなしに高効率なアウトオブオーダー実行が可能という特徴を持つ．しかし，命令セットの制約を満たすために余分な命令の実行が必要となることがある．したがって，コード生成の方法によってはかえって性能の低下を招きかねない．本研究では，SPEC CPU 2017 に含まれる C 言語で書かれた実用プログラムを STRAIGHT 向けにコンパイルしたときの特性を解析し，コンパイル方法の改善点を提案する．

1. はじめに

デナードスケーリング則が成り立っている間はトランジスタサイズを縮小しても面積当たりの消費電力が一定であったが，これが崩壊して以降，高性能プロセッサを作るうえで消費電力の削減が大きな課題となっている．この問題に対処するため，様々な省電力プロセッサアーキテクチャが提案されている [1], [2], [3] が，これらの省電力プロセッサアーキテクチャは不規則な処理の高速化には向いていない．Nowatzki らの研究 [3] によれば，不規則な処理を高速化するための選択肢は，複雑さが指摘されているアウトオブオーダースーパースカラプロセッサしか存在しない．

アウトオブオーダー実行は高い性能を実現できるが，一方で電力効率を悪化させることが知られている [4]．アウトオブオーダー実行は，機械語プログラムに記述されている通りの順番で命令を実行するのではなく，実行の準備ができた命令から順次実行していくことで命令レベル並列性を抽出する技法であり，現代の高性能プロセッサで広く採用されている．命令レベル並列性を抽出するためには偽の依存を除去することが重要であり，その目的でレジスタリネーミングが行われる．しかし，レジスタリネーミングは非常に多ポートのメモリを使用し，またそのメモリは非常に高頻度にアクセスされるため，多くの電力を消費する．そのため，レジスタリネーミングはアウトオブオーダースーパースカラコアの電力上のボトルネックの一つとなっている．

この問題を解決するため，我々は STRAIGHT アーキテクチャを提案した [4]．STRAIGHT は，偽の依存が生じないような命令セットアーキテクチャである．したがって，STRAIGHT プロセッサはレジスタリネーミングなしに高効率なアウトオブオーダー実行が可能であり，電力効率の向上が期待できる．一方，命令セット固有の制約を満たすために余分な命令の実行が必要となるため，コード生成の方法によっては実行速度が低下する可能性がある．STRAIGHT の初期評価では，従来型 RISC と比較して命令数が 20%ほど増加するのものの，実行速度においては同等の性能を維持することが可能であるという結果が得られている [4]．

しかし，その評価は合成ベンチマークを用いたものであるため，実用プログラムでも同様の結果が得られるかは明らかではない．そこで本研究では，C 言語標準ライブラリを STRAIGHT 向けに移植し，SPEC CPU 2017 に含まれる C 言語で書かれた実用プログラムを STRAIGHT 向けにコンパイルしたときの特性を解析する．さらに，実用プログラムで高い性能を出すために障害となっている点を洗い出し，改善案を提案する．

2. STRAIGHT アーキテクチャ

2.1 概要

STRAIGHT は，ソースオペランドの指定にレジスタ名

[1] 東京大学 大学院情報理工学系研究科
[a)] koizumi@mtl.t.u-tokyo.ac.jp

ⓒ 2020 Information Processing Society of Japan

ではなく命令間距離を用いる命令セットアーキテクチャである．STRAIGHT では，「いくつ前の命令の結果を使う」といった形式でソースオペランドを指定する．この指定形式を用いるとレジスタの上書きが発生しないため，原理的に偽の依存が発生しない．現実のプロセッサには物理レジスタが有限個しか存在しないため上書きは避けられないものの，その場合でも上書きされるレジスタの寿命は尽きており偽の依存が発生できないことを保証できる．これは指定できる命令間距離に上限があることからレジスタの寿命を保証できることによる．したがって現実のプロセッサでも偽の依存が発生しない．

2.2 コンパイラアルゴリズム

RISC と異なり，STRAIGHT には名前付きの汎用レジスタが存在しない．そのため，C 言語等の高級言語で書かれたプログラムを STRAIGHT 機械語にコンパイルする際，一般的な RISC 向けコンパイラ手法をそのまま適用することはできない．特に問題となるのは，複数の実行経路が合流する地点において経路に依存してオペランドを選択したい場合である．RISC の場合，全ての実行経路で同じ汎用レジスタに書き込むことでこの問題を解決できる．一方，STRAIGHT の場合，全ての実行経路で命令間距離を同じにすることで問題を解決する．以下は，これを実現するアルゴリズムである [5].

(1) 複数の実行経路が合流する地点において生存している値を列挙する．
(2) それぞれの経路について，参照する値が共通の順番になるよう，合流点の直前に `RMOV` 命令（レジスタ間転送命令）を追加する．

このアルゴリズムは指定できる命令間距離に上限がない場合，全てのプログラムをコンパイルすることができる．指定できる命令間距離が有限の場合でも，適宜スピルすることで常にコンパイルできることを保証できる [6]．また，このアルゴリズムでは多量の `RMOV` 命令を追加するが，大半の `RMOV` 命令は冗長であり，有向閉路除去問題を解くことで除去できる [7].

以下，実用プログラムを STRAIGHT 向けにコンパイルする上で重要であり，かつ後の分析で性能に大きく関わることが判明する二点，関数呼び出し規約および積極的スピルについて説明する．

2.3 関数呼び出し規約 [8]

STRAIGHT には RISC のような名前付きレジスタがないため，「第一引数を `a0` レジスタに入れて渡す」のような特定のレジスタを用いた関数呼び出し規約を定めることができない．代わりに，関数呼び出し・復帰に用いられる分岐命令（`JAL` 命令や `RET` 命令など）との相対距離を用いた呼び出し規約を定める．具体的には，関数呼び出し命令の直前の命令の結果が第一引数，二個前の命令の結果が第二引数，……となるように引数値を生成する命令を配置する．また，復帰命令の直前の命令の結果が返り値となるように返り値を生成する命令を配置する．

RISC アーキテクチャの関数呼び出し規約では，呼び出し側保存レジスタと呼び出され側保存レジスタを設定している．しかし，関数内で何命令実行されるかは一般にわからないため，関数呼び出し前に実行された命令の結果を距離で参照することはできない．よって STRAIGHT では呼び出され側保存レジスタを設定せず，すべてを呼び出し側保存レジスタとする．関数呼び出しをまたいだ参照がある場合，関数呼び出し側の責任で適宜スタックに退避する．

2.4 積極的スピル [9]

ループが何回回った時点でもループ内定数を静的に定められた単一の距離で参照するためには，`RMOV` 命令の実行が不可欠である．この `RMOV` 命令の実行は，ループ内で参照されない値に対しても必要であり，実行命令数が大きく増加する原因となる．この問題は，ループ前にスピルアウトしループ後にスピルインすることで解決可能である．

このような，レジスタ数に余裕がある場合でも積極的にスピルすることで実行命令数を削減する技法が積極的スピル最適化である．積極的スピルでは，ループをまたいだ参照は関数呼び出しをまたいだ参照と同様に扱われる．ある値をレジスタに書き込む命令とその値を読み出す命令の間に，ループも関数呼び出しも存在しないことが静的にわかる場合のみレジスタ上で値を受け渡す．それ以外の場合，つまり実行経路によってはループや関数呼び出しをまたぐ可能性がある場合，その値を使用する命令の直前でスピルインする．

合成ベンチマークである CoreMark[10] を用いた評価では，積極的スピルの適用により実行命令数が約 2/3 となり，実行速度も向上することがわかっている．

3. STRAIGHT における実用プログラムの特性

3.1 実験方法

実用プログラムとして SPEC CPU 2017[11] の中から C 言語で書かれている 2 ベンチマーク，`605.mcf_s` および `657.xz_s` を選択した．また，C 言語標準ライブラリの実装は，musl libc[12] を採用した．

ベンチマークプログラムは，STRAIGHT 向け最適化コンパイラを用いてコンパイルした．STRAIGHT 向け最適化コンパイラは，最適化コンパイラ基盤である LLVM[13] をベースに作成した．LLVM のバージョンは，作成開始当初最新であったバージョン 7 である．また，比較用の RISC

アーキテクチャは RISC-V[14] とした．RISC-V 向け最適化コンパイラは，動作する中で STRAIGHT 向けコンパイラと最も近いバージョンである LLVM のバージョン 9 に含まれる `llc` を用いた．

コンパイル手順は以下のとおりである．

(1) C 言語標準ライブラリである musl を `clang7 -target=riscv64-pc_linux-gnu -S -emit-llvm` でコンパイルし C 言語標準ライブラリの LLVM-IR ファイルを手に入れる
(2) ベンチマークプログラムを `clang7 --target=riscv64 -pc_linux-gnu $CFLAGS -S -emit-llvm` でコンパイルし，ベンチマークプログラムの LLVM-IR ファイルを手に入れる
(3) `llvm-link7` でまとめて一つの LLVM-IR ファイルとする
(4) `llc9 -march=riscv64 -mattr=+m,+f,+d -O2` で LLVM-IR ファイルをコンパイルし，RISC-V アセンブリを手に入れる
(5) `gcc` を用いて RISC-V アセンブリをアセンブルし，RISC-V 用バイナリを手に入れる
(6) `llc7 -march=straight -O2` で LLVM-IR ファイルをコンパイルし，STRAIGHT 用バイナリを手に入れる

上記手順で得られた RISC-V/STRAIGHT バイナリを，`test.0` 入力を用いてサイクルアキュレートなシミュレータである鬼斬弐 [15] でシミュレーションすることで，実行命令数と実行にかかるサイクル数を測定した．ただし，ベンチマークプログラムすべてをシミュレーションするのは時間がかかりすぎるため，ベンチマークプログラムのうち主要な関数を事前に調べ，その部分のみをシミュレーションした．主要な部分として選択した関数およびシミュレーション対象区間は以下のとおりである．

- `605.mcf_s`：`primal_bea_mpp` 関数，初回実行から 1000 回目の実行まで
- `657.xz_s`：`sha_compress` 関数，初回実行から 3000 回目の実行まで

シミュレーションの際に使用したプロセッサパラメータは，表 1 のとおりである．このプロセッサパラメータは，最先端の高性能プロセッサ，具体的には Intel 社の Sunny Cove [16]，AMD 社 Zen2 [17]，IBM 社の POWER9 (Scale Out) [18] を参考にして決定した．分岐命令やキャッシュライン境界をまたいでも分岐予測器の予測通りにフェッチ幅分だけの命令がフェッチできる（ideal）としているのは，密なループを構成する命令列がフェッチ境界をまたいだ場合の性能低下を防ぐためである．今回使用した LLVM のコンパイラはそれを考慮した命令列の配置を行わないため，これに起因する性能低下がランダムに発生する．これを防ぎアーキテクチャ本来の性能を比較するため，このオプションを設定した．

表 1　シミュレーションに用いたプロセッサのパラメータ

	STRAIGHT	RISC-V
Logical register	127	Int 31, Float 32
Fetcher	6-way, 4 cycle, ideal	
Renamer	N/A	2 cycle
Dispatcher	2 cycle	
Scheduler capacity	96	
Issue width	9	
Issue latency	4	
Exec unit	Int×4, Float×2, Load×2, Store×2, iMul×1 iDiv×1	
Load/Store queue	Load 128, Store 72	
Reorder buffer	256	
Branch predictor	8-component TAGE 130-bit history, 8 KiB storage	
Branch target buffer	4-way, 8 Ki entries	
Return address stack	8 entries	
Mem. dep. predictor	Store set 9-bit producer ID, 4 Ki entries	
L1 cache	32 KiB + 32 KiB, 8-way, 4 cycle	
L2 cache	256 KiB, 8-way, 8 cycle	
L3 cache	2 MiB, 16-way, 30 cycle	
Prefetcher	Stream prefetcher at L3 distance 8, degree 2, 16 entries	
Main memory	150 cycle	

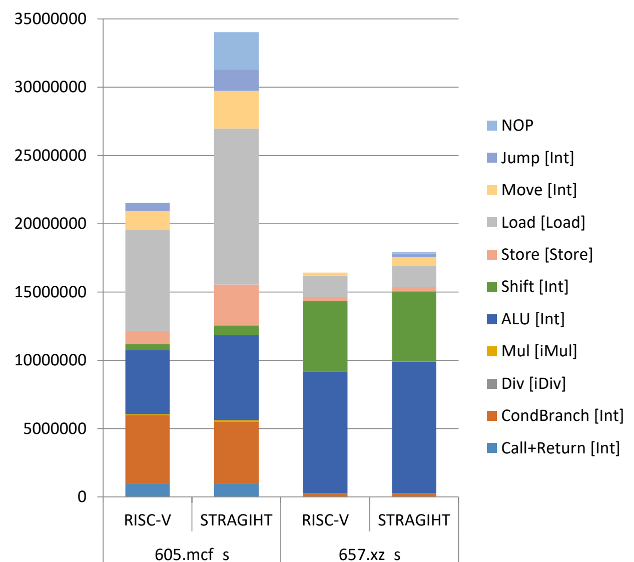

図 1　命令種類ごとの実行命令数

3.2　実験結果

図 1 は，命令の種類ごとの実行数を示したものである．凡例の角かっこ内はその命令の実行ユニットを表す．レジスタ間転送命令（RISC-V における `mv` 命令及び STRAIGHT における `RMOV` 命令）は Move で示されている．`605.mcf_s` を STRAIGHT で動かした場合，RISC-V で動かした場合と比べ，レジスタ間転送命令およびロードストア命令が大きく増加している．一方，`657.xz_s` の場合，RISC-V と

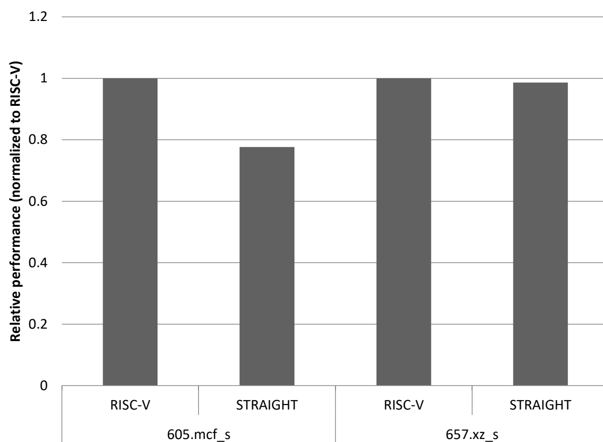

図 2 RISC-V を基準とした相対実行速度

STRAIGHT で命令数は大きく変わっていない．

図 2 は，RISC-V を基準とした相対実行速度（シミュレーション対象区間を実行するのにかかったサイクル数の逆数）を示したものである．605.mcf_s を STRAIGHT で動かした場合，RISC-V で動かした場合と比べて約 1.3 倍の時間がかかっている．一方，657.xz_s の場合には STRAIGHT と RISC-V で実行速度の差はほとんどない．

レジスタリネーミングを廃することと引き換えに実行命令数が増えてしまう STRAIGHT において，実用プログラムである 657.xz_s が RISC と同等の速度で実行できることは，特筆すべき結果である．

4. 分析

先ほどの実験では，657.xz_s の場合，STRAIGHT は RISC とほぼ同等の実行速度を実現できるのに対し，605.mcf_s の場合は大きく実行速度が低下した．605.mcf_s の実行速度低下の要因を分析したところ，以下の二つが原因であることが判明した．

- クイックソートを行う spec_qsort 関数の中で比較関数である cost_compare 関数を呼び出す際，スピルが多発する
- price_out_impl 関数の中で，値をレジスタ上で受け渡せるにもかかわらずスピルアウト・スピルインする部分がクリティカルパスを伸ばしている

これらの問題は合成ベンチマークである CoreMark などでは発生しなかった問題であり，実用プログラムをコンパイルして初めて問題の存在が確認できた．以下，これらの問題について詳しく述べ，対処方法を提案する．

4.1 呼び出され側保存レジスタの欠如に起因する問題

primal_bea_mpp 関数は spec_qsort 関数を呼び出している．spec_qsort 関数は，C 言語標準ライブラリに存在する qsort 関数と同等の関数である[*1]．この spec_qsort 関数は関数ポインタで比較関数を指定するインタフェースとなっており，また再帰関数となっているため，比較関数がインライン展開されていない．実際に実行される比較関数である cost_compare 関数は 30 命令程度の小さい関数であるがインライン展開されておらず，前後にスピルを行うメモリアクセス命令が追加されている（図 3(a)）．一方 RISC-V では，呼び出され側保存レジスタを活用しているため，当該部分はレジスタ間転送命令の実行のみとなっている（図 3(b)）．

ここで，比較関数内部では分岐予測ミスが多発するため，前後でのスピルインにかかるレイテンシはアウトオブオーダー実行によって隠蔽することができず，性能に大きな影響を与えている．

このように小さな葉関数が頻繁に呼び出される場合，呼び出され側保存レジスタの欠如は大きな性能低下を招く．この問題は，STRAIGHT の呼び出し規約に呼び出され側保存レジスタを定義することで解決可能である．STRAIGHT に k 個の呼び出され側保存レジスタを定義する方法として，以下の方法を提案する．

- すべての関数呼び出しの引数を疑似的に k 個増やす
- すべての関数からの返り値を疑似的に k 個増やす
- 呼び出し側では，増えた引数に呼び出され側で保存してほしい値を渡す
- 呼び出され側では，増えた引数を増えた返り値に転送する

このようにすることで，関数内で何命令実行されるかがわからなくとも，関数呼び出し前に実行された命令の結果の値を関数呼び出し後に距離で参照することが可能になる．この手法による性能向上の評価および最適な k の値の探索については今後の課題となる．

4.2 積極的スピルに起因する問題

primal_bea_mpp 関数の途中には，ある実行経路では値をレジスタ上で受け渡せるが，別の実行経路では間にループが挟まるためにループ前にスピルアウト，ループ後にスピルイン，といったコードが生成されている箇所がある．問題となっている部分の primal_bea_mpp の制御フローを図 4(a) に示す．ここで，BB1 に存在する命令で生成され，BB5 で参照される値のことを考える．この値は，参照の途中にループが存在しうるため，積極的スピルによりスピルアウトされ，BB5 での参照の直前にスピルインされる．しかし，右側の BB1-BB3-BB5 の実行経路の場合，値をレジスタで受け渡すことが可能である．また，左側の実行経路

*1 標準ライブラリの違いによりベンチマークの動作が大きく変わることを防ぐため，SPEC 独自の関数が用いられていると考えられる．

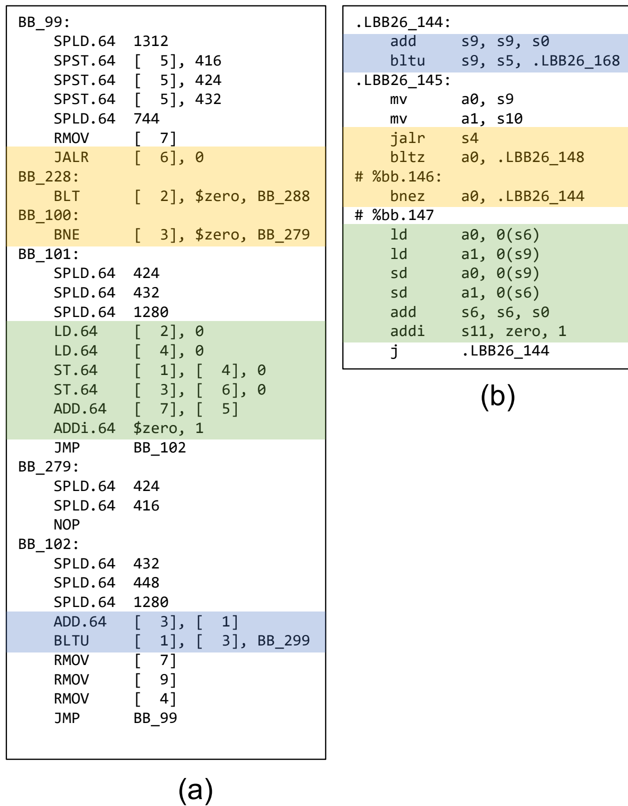

図 3 spec_qsort 関数の一部分．(a)STRAIGHT アセンブリ．スタックへの対比である SPST 命令やスタックからの復帰である SPLD 命令が多い．(b)RISC-V アセンブリ．呼び出され側保存例ジスタである s0，s6，s9，s10，s11 等を用いた効率の良いコードとなっている．

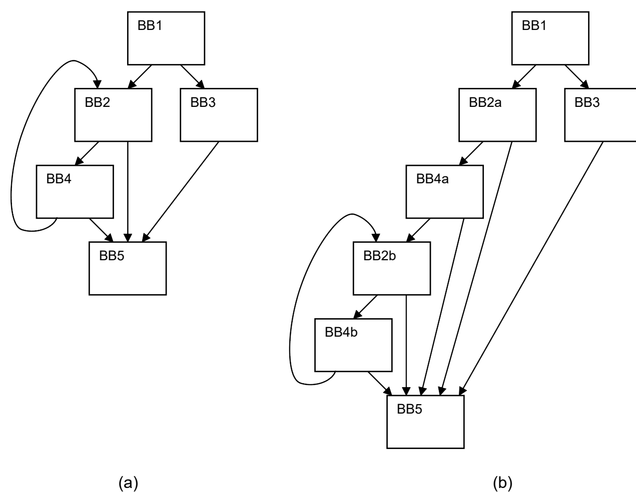

図 4 primal_bea_mpp 関数の制御フロー（一部分）．(a) もともとの制御フロー．(b) ループピーリングを適用した後の制御フロー．

であっても，BB1-BB2-BB4-BB5 と遷移した場合はループを実行していないため，やはり値をレジスタで受け渡すことが可能である．それにもかかわらずメモリを経由して値が受け渡されるため，レイテンシが長くなり，クリティカルパスとなってしまっている．

このように，参照がループをまたぐ可能性があっても実際にはループをまたがない場合，積極的スピルはかえってレイテンシを伸ばしてしまう．これは，積極的スピルが「ループ構造があればそのループは多数回実行される」と暗黙に仮定した最適化であることが原因である．合成ベンチマークである CoreMark と異なり，実用プログラムではループ構造があってもそのループが実行されるとは限らないため，この仮定は見直す必要がある．

この問題に対処するため，コンパイラアルゴリズムに以下の二つの改良を加えることを提案する．

(1) 実行経路によっては関数呼び出しやループをまたがずに値をレジスタで受け渡せる場合，値をなるべくレジスタ経由で受け渡す．参照の直前にスピルインするのではなく，レジスタ経由で受け渡せない実行パスとレジスタ経由で受け渡せる実行パスの合流点直前でスピルインするようにする．

(2) ループの初回実行をループの中からループの前に移動する（ループピーリングを行う）．

この二つの提案手法の意図を先の例を用いて説明する．一つ目の提案手法により，実行が BB1-BB3-BB5 と遷移する場合，値がレジスタ経由で渡されるように変更される．二つ目の提案手法により，制御フローは図 4(b) のように変換される．これにより，元の制御フローで BB1-BB2-BB4-BB5 と遷移していた場合は BB1-BB2a-BB4a-BB5 と遷移することとなり，間にループが存在しなくなる．これを一つ目の提案手法と合わせると，やはり値がレジスタ経由で渡されるように変更される．

この二つを実装し，提案手法 1 のみおよび提案手法 1 と 2 の両方を適用して予備評価を行ったところ，いずれの場合も実行命令数が増加し実行速度は低下してしまった．これは合流点が複雑になったことで，命令間距離を等しくするために追加される RMOV 命令がかえって増加してしまったことが原因と考えられる．この問題を解決し，性能向上につなげることは今後の課題となる．

5. おわりに

STRAIGHT はソースオペランドの指定にレジスタ名ではなく命令間距離を用いることで偽の依存を生じないような命令セットアーキテクチャである．これはレジスタリネーミング無しに高効率なアウトオブオーダー実行を可能にするが，その性能の評価は合成ベンチマークを用いたものにとどまっていた．本研究では，SPEC CPU 2017 に含まれる 605.mcf_s および 657.xz_s を STRAIGHT 向けにコンパイルし，その特性を解析した．

その結果，RISC と同等のハードウェア構成において，657.xz_s の実行サイクル数は同等となることが示された．一方，605.mcf_s においては STRAIGHT における呼び出され側保存レジスタの欠如や積極的スピルが原因で RISC ほどの速度が出ていないことが判明した．これらの問題は

ⓒ 2020 Information Processing Society of Japan

合成ベンチマークによる評価では発生していなかった問題であり，実用プログラムによる評価により初めて明らかとなった．STRAIGHT においても RISC と同様に，呼び出され側保存レジスタの設定やスピルコストの考慮が必要であることが示唆された．

謝辞 本研究の一部は JSPS 科研費 JP19H04077, JP20H04153, JP20J22752 による．

参考文献

[1] Greenhalgh, P.: Big. little processing with arm cortex-a15 & cortex-a7, *ARM White paper*, Vol. 17 (2011).
[2] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F. and Mahlke, S.: Composite cores: Pushing heterogeneity into a core, *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, IEEE, pp. 317–328 (2012).
[3] Nowatzki, T., Gangadhar, V. and Sankaralingam, K.: Exploring the potential of heterogeneous Von Neumann/dataflow execution models, *Int. Symp. on Computer Architecture (ISCA)*, pp. 298–310 (2015).
[4] Irie, H., Koizumi, T., Fukuda, A., Akaki, S., Nakae, S., Bessho, Y., Shioya, R., Notsu, T., Yoda, K., Ishihara, T. and Sakai, S.: STRAIGHT: Hazardless processor architecture without register renaming, *Microarchitecture (MICRO), 2018 51st Annual IEEE/ACM International Symposium on*, IEEE, pp. 121–133 (2018).
[5] Nakae, S.: The STRAIGHT code generation algorithm and optimization of register utilization, Master's thesis, The University of Tokyo (2018).
[6] 杉田 脩，小泉 透，入江英嗣，坂井修一：距離制限を保証する STRAIGHT コード生成アルゴリズムの提案，*xSIG non-published manuscript*, pp. 1–13 (2020).
[7] 小泉 透，中江哲史，福田晃史，入江英嗣，坂井修一：STRAIGHT コンパイラによる冗長転送命令の削減，研究報告システム・アーキテクチャ (ARC)， Vol. 2018, No. 2, pp. 1–10 (2018).
[8] 福田晃史，小泉 透，門本淳一郎，中江哲史，入江英嗣，坂井修一：STRAIGHT アーキテクチャの評価環境の実装，電子情報通信学会技術研究報告，Vol. 118, No. 375, pp. 43–47 (2018).
[9] Koizumi, T., Nakae, S., Fukuda, A., Irie, H. and Sakai, S.: Reduction of instruction increase overhead by STRAIGHT compiler, *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, IEEE, pp. 92–98 (2018).
[10] CoreMark, `https://www.eembc.org/coremark/`.
[11] SPEC CPU(R) 2017, `https://www.spec.org/cpu2017/`.
[12] Musl libc, `https://musl.libc.org`.
[13] The LLVM Compiler Infrastructure, `http://llvm.org/`.
[14] RISC-V: The Free and Open RISC Instruction Set Architecture, `https://riscv.org/`.
[15] Processor simulator onikiri 2, `https://github.com/onikiri/onikiri2`.
[16] Intel Architecture Day (2018).
[17] The Path to "Zen 2", `https://www.slideshare.net/AMD/the-path-to-zen-2` (2019).
[18] Sadasivam, S. K., Thompto, B. W., Kalla, R. and Starke, W. J.: IBM Power9 Processor Architecture, *IEEE Micro*, Vol. 37, No. 2, pp. 40–51 (2017).

ⓒ 2020 Information Processing Society of Japan