Vol.2022-ARC-248 No.3
Vol.2022-SLDM-198 No.3
Vol.2022-EMB-59 No.3
2022/3/10

STRAIGHTアーキテクチャにおけるC++コンパイラ開発と性能評価

灘 洋太郎[1,a)]　小泉 透[2]　杉田 脩[2]　塩谷 亮太[2]　門本 淳一郎[2]　入江 英嗣[2]　坂井 修一[2]

概要：命令間の偽の依存関係は，Out of Order スーパースカラプロセッサの効率的な命令発行を妨げる要因の一つである．STRAIGHT は命令間の偽の依存関係が発生しない命令セットアーキテクチャであり，Out of Order スーパースカラプロセッサをスケーラブルかつ省電力に構成することができる．STRAIGHT はオペランドを命令間距離で指定する特色を持っており，この性質から派生する様々な制約を解いてコードが生成される．C 言語ベンチマークを用いた性能評価では，STRAIGHT 特有の最適化によって RISC-V と同等の性能を実現できることが分かっている．しかし，より複雑な言語において，STRAIGHT 固有の制約が命令数や実行時の性能にどのような影響を与えるのかは未知数であった．本研究ではこれを明らかにすべく，STRAIGHT 用 C++コンパイラの開発，SPEC CPU の C++言語ベンチマークに対する性能評価を行った．その結果，C++言語に特有な性能低下はみられなかった．一方で，C 言語と同様に命令間距離の調整のための命令によって，性能が低下していることが分かった．

1. はじめに

Out of Order スーパースカラプロセッサは機械語プログラム中の命令の順序を動的に入れ替えて実行できるプロセッサである．Out of Order スーパスカラプロセッサは命令レベル並列性を効率よく抽出できるため，現代の高性能プロセッサで広く採用されている．この命令レベル並列性の抽出を妨げる要因として，命令間の偽の依存関係がある．命令間の偽の依存関係は，レジスタへの再書き込みによって発生する．

従来アーキテクチャでは，レジスタリネーミング機構を用いて偽の依存を取り除いていた．ただし，レジスタリネーミング機構は回路面積や消費電力が大きく，フロントエンド幅やレジスタ数を増大させる妨げとなっていた．

STRAIGHT [1] は論理レジスタへの再書き込みを行わない命令セットを用いて，そもそも偽の依存が生じないようにする命令セットアーキテクチャ (instruction set architecture, ISA) である．STRAIGHT は複雑なレジスタリネーミング機構を必要としないため，フロントエンド幅や物理レジスタ数を増加させやすいという強みを持つ．

これに加え，STRAIGHT アーキテクチャは分岐予測ミスのペナルティが既存アーキテクチャより小さい．これは次の２つの理由による．まず，STRAIGHT はリネーミングステージを必要としないため，フロントエンドレイテンシを短縮できる．さらに，STRAIGHT はリネーミング機構を持たないため，レジスタのマップ情報を復帰する逐次巻き戻し処理が必要がない．これらの要因により，分岐予測ミス後により早くパイプラインを命令で満たすことができる．

STRAIGHT アーキテクチャの C 言語ベンチマークによる評価では，STRAIGHT 特有の最適化によって RISC-V [2] と同等の性能が出せることが明らかになっている [3]．なお，この評価は STRAIGHT，RISC-V ともに同一のパラメータで行われている．STRAIGHT は RISC-V がレジスタリネーミング機構に割いていた回路面積をフロントエンド幅の増大のために充てることができるため，回路面積を揃えて評価した場合には，RISC-V より STRAIGHT の方が高い性能を出せると考えられる．

しかし，C 言語以外のベンチマークを用いた性能評価はこれまで行われてこなかった．ゆえに，より複雑な言語において，STRAIGHT 命令セット特有の制約がどの程度命令数増加や実行時の性能に影響を与えるのかは不明であった．そこで本研究では，STRAIGHT 用 C++言語コンパイラを開発し，SPEC CPU [4][5] のうち C++言語で書かれたベンチマークを用いて STRAIGHT アーキテクチャの性能評価を行った．

[1] 東京大学 工学部
The University of Tokyo
[2] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
[a)] nada@mtl.t.u-tokyo.ac.jp

Vol.2022-ARC-248 No.3
Vol.2022-SLDM-198 No.3
Vol.2022-EMB-59 No.3
2022/3/10

2. STRAIGHT アーキテクチャ

2.1 概要

STRAIGHT は，動的命令とその実行結果を書き込む論理レジスタを一対一に対応させるアーキテクチャである．この制約によって論理レジスタへの再書き込みができないため，偽の依存が原理的に発生しない．これは，技術的にはソースレジスタを何命令前のデスティネーションレジスタの値を参照するかという形（レジスタ間距離）で指定することによって実現される．

また，この制約によって高コストなモジュールを用いずに論理レジスタを物理レジスタに対応付けることが可能となる．まず，命令のデスティネーションレジスタは，物理レジスタを順番に割り当てていくことで自動的に命令ごとに異なったレジスタとなる．また，レジスタ間距離はそのまま物理レジスタ番号の差分に読み替えることができる．これらの操作は単純な加減算で実現可能であり，テーブル引き等の高コストな回路が不要である．

ソースレジスタとして指定できるレジスタ間距離の上限（最大参照距離）は ISA で定められている．この制約によって, 一定以上前の命令のデスティネーションレジスタは参照できなくなる（以降，これを「レジスタの寿命が切れる」と表現する）．この寿命の切れたレジスタのみに再書き込みを行うことで，偽の依存を防ぐ．

2.2 STRAIGHT コンパイラ

STRAIGHT コンパイラは，上記の制約を満たすコードを出力する．そのため，従来型 RISC アーキテクチャ向けのコンパイラが行うことに加えて，以下の二種類のコード変形を行う：

長寿命な値がある時の距離調整　寿命が切れたレジスタ上の値は参照できなくなるため，生存変数を必要に応じてレジスタ間転送命令 (RMOV) 命令で最大参照距離以内に移動させる必要がある．RMOV 命令は，ソースレジスタの値をデスティネーションレジスタへコピーする命令である．

実行経路合流時の距離調整　if 文などの分岐後に複数の実行経路が合流する時，合流地点以降で参照される値をレジスタに書き込む命令の配置を調整する必要がある．これは，そのような値の書きこまれているレジスタと合流後の命令のデスティネーションレジスタのレジスタ間距離が，実行経路によらず一定の値である必要があるからである．通常はそのような値を書き込む命令を並び替えることでこれを実現するが，命令の実行順序に制約があり並び替えられない場合もある．その場合，生存変数の値をコピーする RMOV 命令を適切な位置に挿入する必要があり，STRAIGHT において命令数が増加する主要因となっている．

上記のコード変形で追加される RMOV 命令の数は，一般に生存変数が多いほど増加する．

2.3 STRAIGHT アーキテクチャ関数呼び出し規約

STRAIGHT には名前付きのレジスタがないため，以下のような関数呼び出し規約を使う．関数呼び出しの引数が 1 つ以上存在する場合，関数呼び出し命令の 1 つ前の命令の結果に 1 つ目の引数を，2 つ前の命令の結果に 2 つ目の引数を……という方法で引数を並べてゆく．

この規約に従って引数を並べるために，命令の実行順序が制約されることがある．この制約を解決するために RMOV 命令が追加されることがあり，STRAIGHT において命令数が増加する一因となる．

3. 仮想関数呼び出しのオーバーヘッド

C++における仮想関数は virtual というキーワードと共に宣言されたメソッドのことを指し，派生先のオブジェクトからオーバーライドできる特徴を持つ．呼び出される関数は実行時に動的ディスパッチを実現する仕組み（C++では仮想関数テーブル）を用いて決定される．この呼び出しは，一般にインライン展開できない[*1]．これは，コンパイル時に基底クラスと派生クラスのどちらのメソッドが呼ばれているかわからないためである．

例えば，ソースコード 1 のように継承関係にある基底クラス A と派生クラス B があり，A のメソッド M を B でオーバーライドしているとする．この場合 16 行目のポインタ p がクラス A・B のどちらのインスタンスを指しているのか，コンパイル時に知ることができない．13-15 行目の “様々な操作”で，ポインタ p にクラス A, B どちらのポインタが代入されるかは一般に，実行してみなければわからない．ゆえに，コンパイル時にクラス A と B どちらのメソッド M が呼び出されるのか分からず，メソッド M をインライン展開することができない．

STRAIGHT は RISC-V などの従来アーキテクチャに比べ，仮想関数呼び出しに追加のオーバーヘッドがかかる．これは，仮想関数呼び出しがインライン展開できないためである．2.3 節で述べたように，STRAIGHT の関数呼び出しのオーバーヘッドが存在する．よって，インライン展開できない呼び出しのたびに，そのオーバーヘッドを支払う必要がある．

[*1] ただし，例外的に仮想関数に対するインライン展開が可能になる場合もある．C++では，メソッドや基底クラスそのものに `final` キーワードが指定されているときがこれに該当する．この場合メソッドのオーバーライドが行われないとコンパイル時に分かるため，呼び出されるメソッドの実装を特定できる．また，静的なコードの解析によって実行時に呼び出されるメソッドを確定できた場合は，メソッドのインライン展開を行うことができる．これを脱仮想化 (devirtualization) [6] と呼ぶ．

Vol.2022-ARC-248 No.3
Vol.2022-SLDM-198 No.3
Vol.2022-EMB-59 No.3
2022/3/10

ソースコード 1: オブジェクト継承の例

```
class A {
public:
    virtual void M();
};

class B : public A {
public:
    void M() override;
};

int main() {
  A* p;
  ...
  // 様々な操作
  ...
  p->M();
}
```

4. STRAIGHT の C++プログラム評価環境実装

4.1 STRAIGHT 用 C++言語コンパイラの実装

STRAIGHT 向け C++言語コンパイラ実装は，既存の STRAIGHT 用 C 言語コンパイラ [3] をベースに開発した．STRAIGHT の C 言語コンパイラは，LLVM7.0 のバックエンドプログラム (llc) へ実装されていた．そのため, 本研究ではこの実装を，研究開始当時の最新版であった LLVM12.0 へ移植した上で流用した．移植の際は，LLVM7.0 から LLVM12.0 までに行われた LLVM の仕様変更に対応して llc の実装を修正する必要があった．変更箇所は，スタックアラインメント幅の表現方法やビルド手順の指定方法など多岐にわたった．

さらに，バックエンドプログラムに C++言語で新たに必要になった命令や機能を実装した．具体的には，LLVM 中間言語 (LLVM IR) の Atomic 命令，Atomic 疑似命令，GEP 命令への対応を行い，正しく機械語命令列へコンパイルできるようにした．これらについて以下で説明する．

まず，LLVM IR の Atomic 命令 (Atomic Load, Atomic Store) に対しては, 対応する STRAIGHT の Atomic 命令へ変換してコード生成する機能を実装した．一方，LLVM IR の Atomic 疑似命令（Atomic Swap など）には, STRAIGHT の命令セット中に対応する命令が存在しない．このため，Atomic 疑似命令は Atomic ではない STRAIGHT コードへ変換し，その上下を FENCE 命令で囲んだコードを生成するよう実装した．

また，GEP 命令は，構造体のメンバへのオフセットを計算する LLVM IR の命令である．この命令は STRAIGHT 用 C 言語コンパイラでも対応されていたが，構造体の中に多重にネストして構造体が存在する，といった複雑な構造体はサポートしていなかった．本研究では，このような構造体にも対応できるよう再実装した．

なお，例外機能はコンパイラへ実装しなかった．これに伴って C++の標準ライブラリも，例外機能を使わない設定でビルドした．

4.2 C++標準ライブラリの STRAIGHT 向け移植

本研究で開発した STRAIGHT コンパイラはすべての関数をリンクした状態の LLVM IR を入力として STRAIGHT 機械語プログラムを生成する．このため，C++言語で記述されたプログラムをコンパイルするには，C++言語の標準ライブラリが LLVM IR の形で必要となる．一方で C++言語の標準ライブラリのコンパイルは CMake に記述された非常に複雑な手順を踏んで行う必要があり，CMake ファイルを改変してコンパイル結果を LLVM IR の形式で得ることは困難であった．

これに対し本研究では，ライブラリ全体を単一の LLVM IR ファイルへと変換するために，WLLVM [7] を用いた．WLLVM は，コンパイル結果を LLVM IR の形式で得ることのできるコンパイラドライバである．具体的には，Clang や GCC の代わりに WLLVM を指定してコンパイルすることで，もとの CMake ファイルをそのまま利用して，ライブラリ全体を単一の LLVM IR ファイルへと変換した．

STRAIGHT コンパイラは LLVM の RISC-V 向けバックエンド実装をベースとして実装されているため，コード生成において RISC-V 用の LLVM IR が必要となる．しかし，LLVM の RISC-V 対応が不完全であり，標準ライブラリのインクルードパスの設定も難しいため，x86 マシン上でのクロスコンパイルによる RISC-V 用 LLVM IR の生成は困難であった．このため，標準ライブラリの RISC-V 用 LLVM IR への変換は RISC-V マシン上で行うこととした．

5. 評価

5.1 評価方法

SPEC 2006 の 444.namd, 473.astar と，SPEC 2017 の 631.deepsjeng_s をベンチマークとして用いた．これらのソースコードは全て C++言語で書かれている．

ベンチマークプログラムは，STRAIGHT 向け C++言語コンパイラを用いてコンパイルした．また，比較対象のアーキテクチャとしては，RISC-V [2] を用いた．RISC-V 向けのコンパイラとして，STRAIGHT 向け C++コンパイラと同じバージョンである LLVM12.0 に含まれる llc を用いた．

なお，C 言語の標準ライブラリは musl [8] を，C++言語標準ライブラリは libc++ [9] を用いた．

コンパイル手順は以下の通りである．

(1) RISC-V マシン上で, C の標準ライブラリのソースファイルを clang-12 -c -S -emit-llvm で LLVM IR へ変換
(2) RISC-V マシン上で，C++の標準ライブラリを WL-

表 1: シミュレーション区間

ベンチマーク	区間	シミュレーションの範囲
444.namd	sim0	PairCompute::doWork(PatchList*) 関数の 10 ～ 12 回目の実行
	sim1	SelfCompute::doWork(PatchList*) 関数の 10 ～ 11 回目の実行
473.astar	sim0	regwayobj::createway(regobj*,regobj*,regobj**&,int&) 関数の 10 ～ 135 回目の実行
	sim1	way2obj::createway(int,int,int,int,unsigned char*,pointt*&,int&) 関数の 10 ～ 12 回目の実行
	sim2	wayobj::createway(int,int,int,int, point16t*&,int&) 関数の 10 ～ 24 回目の実行
631.deepsjeng_s	sim0	search(state_t*,int,int,int,int,int) 関数の 2168 ～ 4362 回目の実行

LVM を用いて LLVM IR へ変換

(3) RISC-V マシン上で，ベンチマークのソースファイルそれぞれを clang++-12 -c -S -emit-llvm -stdlib=libc++ -fno-exceptions で LLVM IR へ変換

(4) 以上で得られた LLVM IR すべてを llvm-link で結合し，単一の LLVM IR ファイルにする

(5) llc-12 -mattr=+m,+f,+d,+a -O2　で，(4) で得られた LLVM IR ファイルをコンパイルし，RISC-V アセンブリを得る

(6) gcc [10] を用いて RISC-V アセンブリをアセンブルし，RISC-V バイナリを得る

(7) llc-12 -O2 -march=straight で，(4) で得られた LLVM IR ファイルをコンパイルし，STRAIGHT バイナリを得る

手順 (1), (2), (3) は全て RISC-V マシン上で行った．これは STRAIGHT コンパイラが入力として用いる RISC-V の LLVM IR へとソースコードを変換する必要があるためである．

上記手順で得られた RISC-V と STRAIGHT のバイナリに対し，test 入力を用いてコンパイル結果が正しいことを確認した．まず，RISC-V と STRAIGHT の結果は一致した．またその結果は一部，リファレンス出力と異なる値もあったが，その場合でも誤差は SPEC が許す範囲内であった．この誤差は浮動小数点数演算に由来するものと思われる．x86 でのネイティブ実行を含め多数の環境で実験し，環境により結果が異なりうることを確認している．

次に，性能を評価するため鬼斬弐 [11] を用いたシミュレーションを行った．鬼斬弐はパイプラインプロセッサの動作を一クロックごとの単位で正しくシミュレート（サイクルアキュレートなシミュレーション）できるシミュレータである．ただし，ベンチマークの全体をシミュレーションすると時間がかかりすぎるため，ベンチマークプログラムの主要な関数を抽出し，その関数が実行される部分に対してシミュレーションを行った．シミュレーション区間は表 1 のとおりである．

シミュレーションに用いたプロセッサパラメータを表 2 に示す．プロセッサパラメータ Medium は最新の性能重視の CPU を意識したパラメータであり，Performance x86 Core [12]，Zen3 [13]，Power10 [14]，M1 [15] を参考にした．2xMedium, 3xMedium は Medium のフロントエンド，バックエンド幅をそれぞれ 2，3 倍にしたパラメータである．なお，物理レジスタは十分な量があるものとした．また，STRAIGHT アーキテクチャのフロントエンドレイテンシは RISC-V より 2 サイクル短く設定した．これは，STRAIGHT アーキテクチャがレジスタリネーミングステージを必要としないためである．

さらに仮想関数の引数を配置する RMOV 命令が，STRAIGHT で実行命令数をどの程度増加させるかも調べた．これにより，3 節で述べた，仮想関数呼び出しの STRAIGHT に特有なオーバーヘッドを測定した．ただし，どの RMOV 命令が引数並び替えに用いられているかを判別するのが困難だったため，仮想関数呼び出しに使われる間接呼び出し命令（JALR）を含むベーシックブロック中の RMOV 命令の個数を測定した．測定には鬼斬弐の skip 実行機能を用いた．

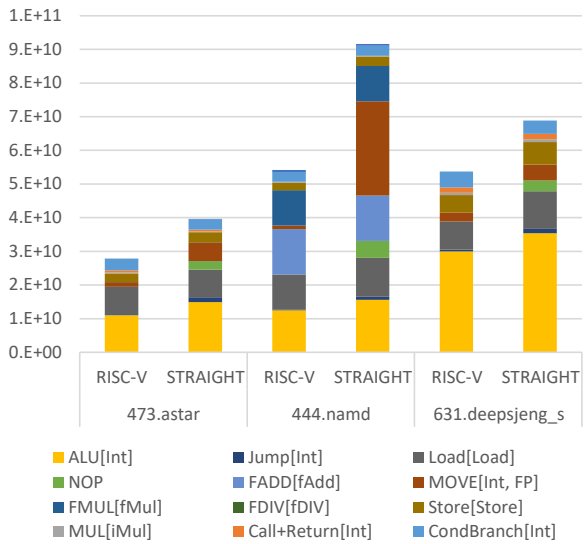

図 1: test を全実行する際の命令タイプごとの命令数

5.2 評価結果

図 1 に，命令の種類ごとの実行数を示す．凡例の角括弧内の文字は，命令を実行するユニットを表す．図中の MOVE は STRAIGHT においては RMOV 命令，RISC-V では mv 命令を意味する．

また, LD/ST は整数のロード/ストア命令と浮動小数点数のロード/ストア命令を足した値を図示した．STRAIGHT は RISC-V のような浮動小数点数レジスタを持たず，整数のロード/ストア命令で浮動小数点数も扱う．ゆえに浮動小数点数専用のロード/ストア命令を用いる RISC-V との比較を容易にするため，このような図示の仕方を選んだ．

表 2: シミュレーションで用いたプロセッサパラメータ

	Medium	2xMedium	3xMedium
フェッチ幅・リネーム幅	8	16	24
発行幅	10	20	30
リタイア幅	10	20	30
フロントエンドレイテンシ	STRAIGHT: 6, RISC-V: 8		
発行レイテンシ	4		
演算レイテンシ	Int: 1 cycle MUL: 3 cycle (pipelined) IDIV, FDIV: 15 cycle Load, Store: 5 cycle FPADD, FMUL: 3 cycle		
リカバリレイテンシ	L1 命令キャッシュレイテンシ以下 (RISC-V, STRAIHGT ともに 0)		
演算器数	Int 5 MUL 2 LD 3 ST 4 DIV 1 FPDIV 1 FPADD 2 FPMUL 2	Int 10 MUL 4 LD 3 ST 4 DIV 2 FPDIV 2 FPADD 4 FPMUL 4	Int 15 MUL 6 LD 3 ST 4 DIV 3 FPDIV 3 FPADD 6 FPMUL 6
スケジューラサイズ	320	640	960
ロードキューサイズ	128	256	384
ストアキューサイズ	80	160	240
リオーダーバッファサイズ	600	1200	1800
分岐予測器	8-component TAGE 130-bit history, 8 KiB storage		
分岐先バッファ	4-way, 8 K entries		
リターンアドレススタック	32 entries, with rename		
メモリ依存予測器	Store set 13-bit producer ID, 32 K entries		
スケジューリングポリシー	古い命令優先 選択的フラッシュ 実行終了時脱退		
L1 キャッシュ	64 KiB （命令）+ 32 KiB （データ） 8-way, 64 B line, 4 cycles		
L2 キャッシュ	256 KiB 8-way, 64 B line, 8 cycles		
L3 キャッシュ	2 MiB 16-way, 64 B line, 30 cycles		
プリフェッチャ	Stream prefetcher at L3 distance 8, degree 2, 16 entries		

いずれのベンチマークにおいても，STRAIGHT では RISC-V に比べて距離調整に用いられる MOVE，NOP，JUMP が増加した．なお，JUMP 命令が増加する理由は距離調整を行うコード領域を飛び越すために用いられているためである．

また STRAIGHT では ALU も増加した．これは，生存変数を RMOV 命令で移動させる代わりに，ALU 命令で同じ値を再計算することがあるためである．これは依存の連鎖を短くするために行われる最適化が原因である．

また，STRAIGHT で test 入力を全実行したとき，仮想関数呼び出しと同じ BB 内の RMOV 命令は実行された命令数全体のうち以下の割合を占めていた．括弧内の数字は，実行された RMOV 命令全体に対する割合を示す．

- 473.astar: 1.4% (9.8 %)
- 444.namd: 3.9×10^{-6}% (1.3%)
- 631.deepsjeng_s: 3.3×10^{-5}% (4.8×10^{-4} %)

図 2，図 3 にベンチマークを実行した際の性能を示す．性能は，シミュレーションの実行サイクル数の逆数を，RISC-V のプロセッサパラメータ Medium での実行サイクル数で正規化したものを用いた．473.astar の sim1，sim2 を除くと，いずれのシミュレーション区間とプロセッサパラメータでも STRAIGHT の性能は RISC-V の 8，9 割程度となった．一方 473.astar の sim1 のパラメータ 3xMedium，sim2 のパラメータ 2xMedium，3xMedium において同一パラメータの RISC-V より 3 ∼ 5%高い性能が出た．

6. 分析

6.1 RMOV 命令増加の要因

5.2 節の結果より，命令数増加の主要因は距離調整に用いられる RMOV 命令であることが分かった．また，仮想関数呼び出しでの RMOV 命令増加の影響は全実行命令数の最大 1%程度にとどまり，RMOV 命令増加の主要因でないことも明らかになった．以下では，RMOV 命令を最も多く実行していた関数をベンチマークごとに抽出し，その特徴について分析してゆく．

6.1.1 444.namd

444.namd で RMOV 命令を特に多く実行していたのは，以下に示す ComputeNonbondedUtil クラスのメンバ関数であった．

- calc_pair_energy_fullelect
- calc_pair_fullelect
- calc_pair_energy_merge_fullelect
- calc_pair_energy

これらの関数は，いずれも生存変数を大量に含んだループを持っていた．このためループ間で生存変数を保存するために大量の RMOV 命令が追加され，実行命令数の増加を招いた．さらに，これら RMOV 命令と後続の浮動小数点演算が依存関係にあったため，浮動小数点演算の発行を妨

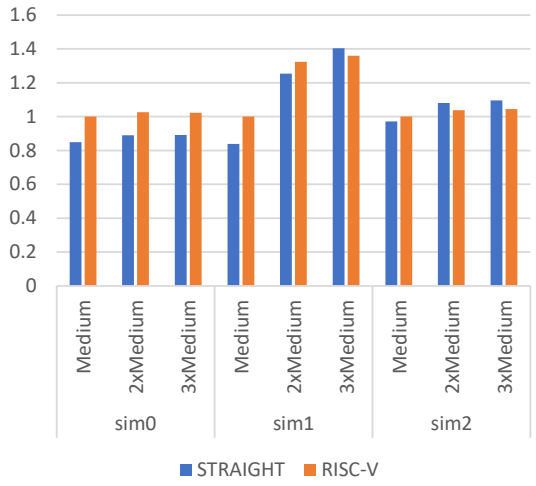

図 2: 473.astar の実行速度

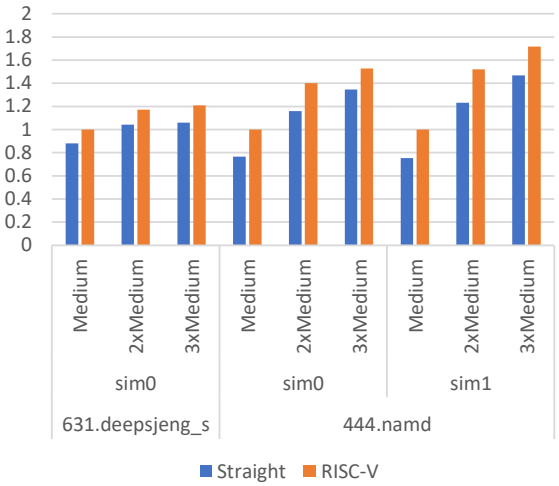

図 3: 444.namd，631.deepsjeng_s の実行速度

げて性能を低下させていた．

6.1.2 473.astar

473.astar で RMOV 命令を特に多く実行していたのは，以下に示す関数であった．

- wayobj::makebound2(int*,int,int*)
- way2obj::releasepoint(int,int)
- regwayobj::makebound2(flexarray<regobj*>&, flexarray<regobj*>&)

これらの関数はいずれも多数の if 文を含んでいた．このため，ベーシックブロックの合流点が多くなり，距離調整用の RMOV 命令が大量に追加されていた．

6.1.3 631.deepsjeng_s

- memset
- make(state_t*, int)

これらの関数は多くの if 文で構成されており，複数のベーシックブロックが合流する際の距離調整用の RMOV 命令が大量に用いられていた．特に musl の memset 実装は早期リターンを多く含む技巧的なコードであり，リターンするベーシックブロックを一つにまとめるという，STRAIGHT と相性の悪い RISC 向けコンパイラ最適化がかかってしまっていた．

6.2 473.astar で高い性能が出た理由

473.astar の sim1，sim2 では，パラメータが十分に大きい場合，STRAIGHT のほうが RISC-V より高い性能が出ていた．これらのシミュレーション区間では，分岐予測ヒット率がいずれのプロセッサパラメータでも 77~78%と非常に低かった．ゆえにフロントエンドレイテンシが小さい STRAIGHT のほうが，分岐予測ミスからのリカバリが早いため RISC-V より高速に動作したと考えられる．一方パラメータ Medium ではプロセッサが小さいため，命令数増加による実行速度低下の効果がそれを上回った．

7. おわりに

本研究では，STRAIGHT アーキテクチャ用の C++コンパイラを開発し，SPEC CPU ベンチマークでその性能を評価した．性能を評価した結果，大半のシミュレーション区間・シミュレーションパラメータにおいて，STRAIGHT は RISC-V の 8，9 割程度の性能であった．一方，分岐予測ミスが多発する 473.astar のシミュレーション区間では，プロセッサパラメータを大きくすることで，RISC-V より約 4%高い性能が出た．

また命令数増加の主な要因は，C 言語ベンチマークと同様にベーシックブロック間の距離調整に必要な RMOV 命令，NOP 命令であることが分かった．一方，C++言語特有の，仮想関数の引数並び替えの RMOV 命令による命令数の増加は，全実行命令数の最大 1%程度にとどまることが分かった．

参考文献

[1] Irie, H., Koizumi, T., Fukuda, A., Akaki, S., Nakae, S., Bessho, Y., Shioya, R., Notsu, T., Yoda, K., Ishihara, T. et al.: STRAIGHT: Hazardless Processor Architecture without Register Renaming, *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, pp. 121–133 (2018).

[2] RISC-V-Foundation: RISC-V Foundation — Instruction Set Architecture(ISA), RISC-V Foundation (online), available from ⟨https://riscv.org⟩ (accessed 2022-01-27).

[3] Koizumi, T., Sugita, S., Shioya, R., Kadomoto, J., Irie, H. and Sakai, S.: Compiling and Optimizing Real-world Programs for STRAIGHT ISA, *2021 IEEE 39th International Conference on Computer Design (ICCD)*, IEEE, pp. 400–408 (2021).

[4] SPEC CPU(R) 2017, `https://www.spec.org/cpu2017/`.

[5] SPEC CPU(R) 2006, `https://www.spec.org/cpu2006/`.

[6] 中田育男:コンパイラの構成と最適化 (第 2 版), 朝倉書店 (2009).
[7] WLLVM, https://github.com/travitch/whole-program-llvm.
[8] Musl Libc, https://www.musl-libc.org/.
[9] libc++, https://libcxx.llvm.org/.
[10] GCC, https://gcc.gnu.org/.
[11] Onikiri2, https://github.com/onikiri/onikiri2.
[12] Intel Architecture Day(2021), https://www.intel.com/content/www/us/en/newsroom/resources/press-kit-architecture-day-2021.html.
[13] Zen3, https://www.amd.com/en/technologies/zen-core-3.
[14] Power10, https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_IBM_Starke_POWER10_v33.pdf.
[15] Jani, A.: Apple Ships Its First PC Processor, pp. 1–5 (2021).