

手続き間解析による STRAIGHT アーキテクチャ向け 可変長 callee-save 呼出規約の実現

杉田 脩^{1,a)} 入江 英嗣^{1,b)}

概要：STRAIGHT アーキテクチャは、レジスタオペランドを命令間の距離で指定する。そのため、同アーキテクチャの関数呼び出しでは、callee 側で進む総距離を一般に静的に決定できないことから、caller 側でコンテキストの保存を行う必要がある。これに対して、関数の引数と戻り値を一定個数ずつ拡張することでレジスタ経由で値を引き渡し、callee-save の実現手法が提案されている。しかし、固定数の拡張を呼出規約とするこの方法は、保存すべき値の数に関わらず一定のレジスタ操作が発生し、不要なスピルなどのオーバーヘッドが発生する。そこで本研究では、caller が真に必要な個数だけ callee-save スロットを割り当てる可変長 callee-save 呼出規約を提案する。コード生成においてはアセンブリレベルの静的解析により、スロット数に整合する callee 関数へのバインドと、不要な定義のデッドコード削除を行う。本稿では、この実装による効果を評価し、その有効性について検証する。

1. はじめに

高性能かつ高効率なコンピューティング需要の高まりとともにアーキテクチャの変革が求められる中、偽の依存が生じないアーキテクチャ・パラダイムとして距離指定型アーキテクチャが提案されている [4], [7], [10]。その代表例が STRAIGHT アーキテクチャ [4] である。STRAIGHT は、各命令の宛先レジスタをフェッチした順序で割り当て、レジスタオペランドを距離 (index の差分) で指定する。これにより、レジスタ上の値の上書きによる偽のデータ依存は生じない。そのため、電力消費が高くフロントエンド幅の 2 乗の割合で回路規模がスケールする [12]、旧来のリネーム機構を不要となる。よって、高効率かつスケラブルにアウトオブオーダー実行を実現できる。

各命令の宛先レジスタが順番かつ自動的に割り当てられる STRAIGHT では、特定のレジスタ番号を引数・戻り値レジスタにできないため、従来とは異なる呼出規約を用いる。まず、引数と戻り値は、call^{*1} や ret 命令の直前に順番に並べられる。また、callee (被呼出し関数) 内で距離が進むことにより caller (呼出し関数) におけるコンテキストの相対的位置が変わってしまうことから、全てのコンテキストは caller-save する必要がある。つまり、STRAIGHT において callee-save を行う「レジスタ」は存在しない。

この callee-save レジスタの欠如に対して、STRAIGHT ならではの callee-save の実現方法 [8] が提案されている。それは、各関数の引数と戻り値を一定個数拡張し、引数としてコンテキストをレジスタ渡しし、戻り値としてコンテキストをレジスタから復元するというものである。これにより、callee-save が無いことによるメモリアクセスの増加を抑制できる。一方で、固定数の拡張を呼出規約とするこの方法は、保存すべき値が拡張数より少ない場合、不要な値をスタックに退避・復元するオーバーヘッドが発生してしまう。

そこで本稿では、必要な個数だけ callee-save スロットを割り当てる可変長 callee-save 呼出規約を提案する。提案する呼出規約では、既存の callee-save の実現方法とは異なり、同一実行バイナリ内で異なる callee-save の個数の関数呼び出しを混合して使用する。これは、caller と callee で拡張する引数・戻り値の数が一貫していれば、自由に拡張数を決められることを利用している。

また、この呼出規約を実現するコード生成手法を提案する。提案手法では、まず各関数について異なる callee-save 数に対応する複数のアセンブリを生成する。このアセンブリ生成においては、各関数呼び出しに対し、必要な数だけ callee-save を設定する。そして、それらをマージし、割り当てたスロット数に整合する callee 関数へのバインドを行う。また、バイナリサイズが大きくなることを抑制するため、不要な定義のデッドコード削除を行う。

可変 callee-save 規約について、コンパイラおよびプリリ

¹ 東京大学, The University of Tokyo

^{a)} sugita@mtl.t.u-tokyo.ac.jp

^{b)} irie@mtl.t.u-tokyo.ac.jp

^{*1} STRAIGHT では jal, jalr に展開される。

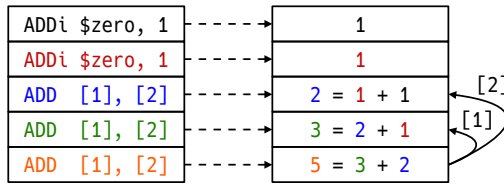


図 1: STRAIGHT の命令列の例

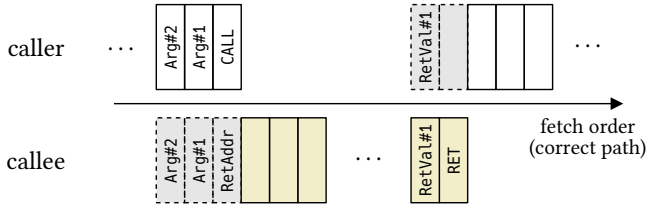


図 2: STRAIGHT の関数呼び出しでの引数・戻り値の配置

ンカに対して提案するコード生成法を実装し、複数の C ベンチマークで評価を行った。その結果、実際に提案する規約のコードが生成され、プログラムの実行が可能であることを確認した。また、既存の固定長の callee-save 規約で生成したコードに対して、提案する規約がロード・ストア命令の実行数を悪化させず、いくつかのベンチマークについて削減することを確認した。また、性能評価では全てのベンチマークについて性能を悪化させず、12 個中の 2 個のベンチマークについて、実行性能を 2-3% 改善することを確認した。

2. STRAIGHT アーキテクチャ

2.1 概要

STRAIGHT は、レジスタオペランドを命令間の距離によって指定するアーキテクチャである。命令間の距離とは、フェッチした順序において何命令前の結果かである。これについて、簡単な STRAIGHT のアセンブリ列を図 1 に示す。図において、 $[x]$ のオペランドは x 個前の命令の結果を示す。このように STRAIGHT は、レジスタを経由せず、値を生成する命令を直接指すことによって、計算を実行する。

この方式では、偽の依存が生じないため、アウトオブオーダー実行に従来の高コストなリネーム機構が不要となる。従来アーキテクチャでは同じレジスタに書き込まれる値を別々の物理レジスタにリネームすることによって並列性を抽出してきた。しかし、STRAIGHT では、各レジスタオペランドは命令と紐づいて参照されるため、他の命令および結果の値によってオペランドが上書きされてしまうということがない。つまり、Write After Read のような偽の依存は STRAIGHT において生じない。これを実現するアーキテクチャ実装については [1], [4] を参照されたい。

2.2 STRAIGHT における関数呼び出し

ここでは本研究に関連する STRAIGHT の呼び出し規約

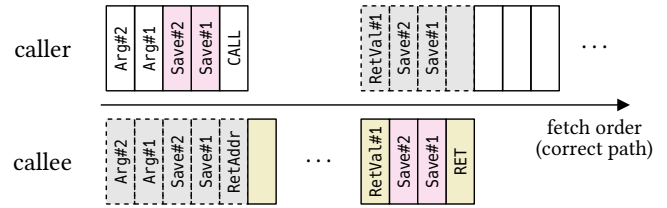


図 3: 引数と戻り値の拡張による callee-save の実現

について述べる。STRAIGHT は、従来 ISA と同様に、特定のレジスタに引数・戻り値を特定の順番に格納するという規約を使えない。これは、すべての汎用レジスタは距離で参照し、特定のレジスタに値を書き込むということもできないためである。

そこで STRAIGHT は、レジスタオペランドの指定と同様な、距離による呼び出し規約を採用する (図 2)。この規約では、関数を呼び出す命令 (jal, jalr) の直前の命令を一つ目の引数、その前を二つ目の引数、... とする。同様に、復帰する命令 (ret) の直前の命令を一つ目の戻り値、そのまを二つ目の戻り値、... とする。この方法を用いることで、STRAIGHT は関数間で引数・戻り値をレジスタ上で引き渡すことができる。

また、関数呼び出しに関してレジスタ保存も STRAIGHT は従来 ISA 同様の規約を使用できない。従来の ISA では、各レジスタについて caller 内と callee 内のどちらでコンテキストを保存するかの規定が存在する。しかし、STRAIGHT ではレジスタの区別が存在しない上、callee 内でどれだけ RP が進むかも一般にわからない。このため、STRAIGHT において callee-save を行うレジスタというのは存在しない。

一方で callee-save の欠如は葉関数を呼ぶ際に不要なメモリアクセスを増加させるため、別の形で callee-save の実現方法が提案されている [8]。この方法では、全ての関数の引数と戻り値を一定個数拡張する。そして、関数呼び出しの際に引数の拡張スロットに保存したい値を渡し、渡した引数をそのまま戻り値の拡張スロットに入れて、レジスタ経由で値を引き渡すという方法である (図 3)。本研究では、提案する可変長 callee-save に対し、この方法を固定長 callee-save と呼ぶ。

2.3 固定長 callee-save の課題

固定長 callee-save は、特に小さい葉関数の呼び出しにおいて、caller-save によるメモリアクセスの増加を大きく抑制できる。しかしその一方で、保存すべき値の数が拡張数より少ない場合に、不要な値を引数から戻り値に移動させる (場合によってはスタックに退避・復元する) オーバヘッドが存在する。

この保存すべき値の数が少なくなる典型的なケースを図 4 にしめす。これは、ループ中で使わない不要な値を積極的にレジスタに退避する aggressive spill 最適化 [8] を適用

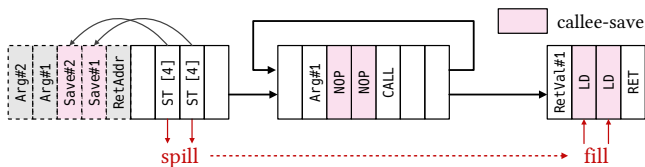


図 4: Aggressive spill によって保存する値の割り当てがなくなる例。ループ内の CALL が自身のような関数を呼び出すとき、NOP による不要な値がメモリアクセスを生むことが分かる。

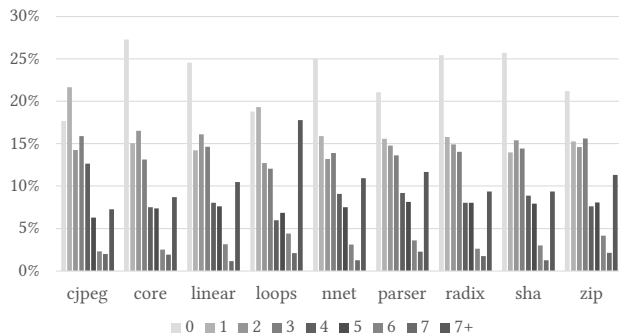


図 5: 関数呼び出しに対する callee-save への割り当て候補数の分布

したコードである。このコード中のループ内の call が自身を呼び出す場合を考える。拡張された callee-save スロットは関数内部で計算に使用されないため、ループが存在すると、この最適化でスロット上の値がスタックに退避される。このように aggressive spill 最適化が適用されるとループ内部で生存する変数が少なくなり、そのようなループ内で起こる関数呼び出しでは保存すべき値が少ないことも多い。

これについて、CoreMark-PRO [3] の全関数において関数呼び出しそれぞれに必要な callee-save 数の分布を予備評価した。この予備評価では、各関数の引数・戻り値の拡張数を 3 で固定し、割り当て候補となる値の数の分布を計測している^{*2}。

予備評価の結果を図 5 に示す。このように、関数呼び出しごとに必要な callee-save 数にはばらつきがある。また、先ほど述べたように、拡張数に対して割り当てたい値が少ないと不必要なメモリアクセスやデータの移動が必要になる。以降では、このような問題を解決するために、一定の範囲内で自由に callee-save 数を決定できる呼び出し規約を提案する。

3. 提案手法

3.1 可変長 callee-save 呼出規約

まず、本稿で提案する可変長 callee-save 呼出規約について説明する。我々の提案する呼び出し規約では、図 6 のよ

うに、caller/callee 間で一貫する範囲内で、callee-save を行う値の数を自由に設定できる。

この呼出規約では以下の 2 つのパラメータを設定する。

- M (maximum): callee-save 出来る値の個数の最大値
- B (baseline): callee-save する値の個数の基本値

この 2 つの値は $0 \leq B \leq M$ の関係である必要がある。また M は、STRAIGHT には値を参照できる距離に最大値 (max reference distance; MRD) が存在するため、自明に MRD 未満となる。

事前に設定した M, B に対し、各関数呼び出しについて M 個以下の範囲で callee-save をする値の個数 (引数・戻り値の拡張数) を自由に決定できる。ただし、いくつか例外が存在する。以降ではそれらを順番に説明する。

間接呼び出し

間接呼び出し (indirect call) は、関数ポインタなどを利用した、動的に生成されるアドレスに対する関数呼び出しの形態である。これの大きな問題は、callee のコンパイル時において callee が何であるかが一般に不明なことである。そのため、間接呼び出しで呼び出す関数の callee-save 数は B 個 (基本値) に固定する。これによって、どのような場合においても正しく一貫した callee-save 数による間接呼び出しが可能となる。

末尾呼び出し

末尾呼び出し (tail call) は、ある関数 (caller) が最後に行う処理として他の関数 (callee) を呼び出し、その戻り値をそのまま自身の戻り値とする呼び出し形態である。末尾呼び出し最適化では、callee の引数を所定のレジスタに配置した上で callee の先頭アドレスにジャンプすることで、caller のリターンアドレスを維持し、callee からのリターン時に caller の元々のリターンアドレスへの直接の復帰を可能とする。STRAIGHT は今のところ標準でこの末尾呼び出しをサポートしないが、専用命令の追加や仕様変更でこれをサポートできるため^{*3}、これに対する提案規約の仕様を定める。

末尾呼び出しを行う関数呼び出しでは、caller の callee-save な引数を、ジャンプ先の callee からの復帰時に戻り値として返却する必要がある。すなわち、図 7 のように、末尾呼び出しをする caller と callee は callee-save する数は等しくなければならない。したがって、直接呼び出しの末尾呼び出しでは、caller が呼ぶ関数は必ず同じ callee-save 数である必要がある。一方、間接呼び出しの末尾呼び出しも存在する。上記の通り、間接呼び出しは B 個の callee-save を持

^{*2} 拡張数を 3 としたのは、[8] においてそれが性能最適だったためである。割り当て方法の計算は、提案手法の実装と同等である。

^{*3} STRAIGHT への末尾呼び出しの実装の障害は、通常の関数呼び出しが 1 命令で完了するのにに対し、末尾呼び出しはリターンアドレスの書き込み (move) と callee へのジャンプの 2 命令が必要となるため、通常の引数配置の規約を満たせないことにある。これは、例えば末尾呼び出し専用の tail, tailr 命令を作る、あるいは、Clockhands [7] のようにジャンプ命令は RP を進めないという ISA 設計にすると自然に実装できる。

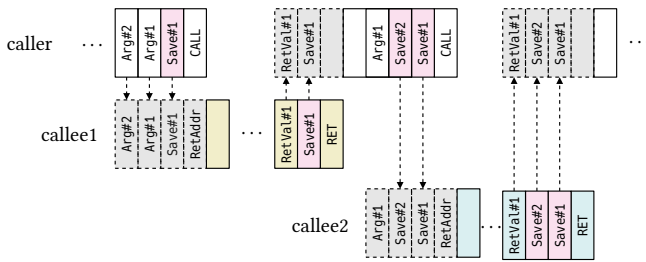


図 6: 提案する可変 callee-save 規約のコード例。1 回目の関数呼び出しでは callee-save 数が 1、2 回目では 2 となっている。caller と callee 間で callee-save 数が一貫していれば関数の実行に問題がない。

表 1: 提案する規約で設定可能な callee-save 数とその条件

	通常 (jal/jalr)	末尾呼び出し (tail/tailr)
直接呼び出し	$0 \leq n \leq M$ の任意の n	caller と同じ個数
間接呼び出し	B 個で固定	caller と callee が共に B 個の時のみ

つ関数を呼ぶと規定するため、callee が B 個の callee-save を持たない限り間接の末尾呼び出しは行えない。

3.2 コード生成

以上の呼び出し規約について、設定できる callee-save の数とその条件をまとめたものを表 1 に示す。次に、この呼び出し規約を実現するコード生成手法について提案する。

本稿で提案するコード生成手法において用い、新たに処理・実装が必要となったのは以下の 2 つである。

コンパイラ 各関数呼び出しに対し、表 1 を満たす範囲で必要な数だけ callee-save スロットを割り当てる。また、その呼び出しに割り当てられたスロット数について、後続の処理でもわかるように印をつける。

プレリンカ 今回新たに実装した、アセンブリをリンカに渡す前に処理を行うツール。異なる callee-save 数で生成したアセンブリを結合させ、呼び出すときの callee-save 数と呼び出される関数の callee-save 数を一貫させる。

以降では、このコード生成法におけるそれぞれの実装と実行方法について詳しく説明する。

コンパイラ処理

コンパイラでは、callee-save 数 N を指定し、コンパイルする各関数の引数と戻り値を N 個拡張する。この時、コンパイルする関数に suffix csr N を付ける。また、その関数が呼ぶ callee について、表 1 の範囲内で自由に callee-save する値の数 K を決める。そして、呼び出す関数に suffix csr K をつける。これを $N = 0 \dots M$ の全ての整数でコンパイルし、全ての条件に対するアセンブリを生成する。

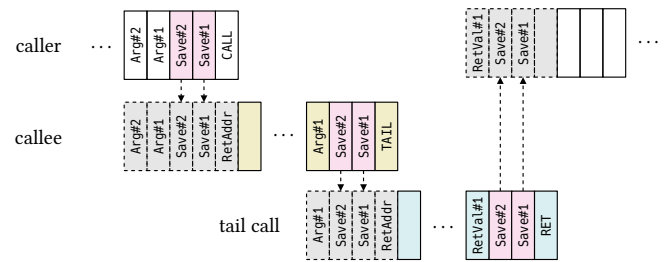


図 7: 末尾呼び出しにおける callee-save 例。callee が末尾呼び出しで呼び出す関数が自身の callee-save 数と同じでない場合、callee の引数に存在する callee-save 値を caller に返却することができない。

プレリンカ処理

プレリンカでは、コンパイラで生成した callee-save 数 $N = 0 \dots M$ のアセンブリを結合させる。このとき、ラベルやオブジェクトが同じものがあると、リンカで処理できないため、別々のオブジェクトに見えるよう名前を変更する。また、間接呼び出しが callee-save 数 B の関数を呼び出せるように、先頭のアドレスを作るときの関数のラベルは suffix csr B を付ける。

単純な結合は不要な関数定義や共通する定義が含まれバイナリサイズが大きくなるため、プレリンカではデッドコード・共通定義の削除も行う。デッドコード削除では、関数間の呼び出し関係から call graph を作り、不要な関数定義を削除する。本研究の実装では、全ての callee-save 数 B の関数が呼び出される可能性があるとして仮定し、それらの関数と start 関数から辿り着く全ての関数を必要な関数定義とした。また、文字列など関数固有の情報を持たない明らかに共通な定義に関しては、一つを残して、残りは全てそれを参照するようにした。

4. 評価

4.1 実験設定

本研究で提案した規約およびコード生成法が有効に機能することを実証するために、複数のベンチマークに対してバイナリを生成しシミュレーション評価を行う。本研究で用いたベンチマークは、SPEC CPU2017 [2] から mcf, lbm, xz と、CoreMark-PRO [3] の cjpeg, core, linear, loops, nnet, parser, radix, sha, zip の合計 12 個である*4。SPEC CPU2017 のベンチマークは実行区間が長いため、[8] を参考に代表的な区間のみを計測に用いている。

本実験において比較するのは以下の 3 つの方針・規約によって生成されたバイナリである。

ベースライン 拡張数 3 の固定長 callee-save

可変長 callee-save (variable) ベースライン B を 3、最大数 M は十分な数に設定

葉関数を考慮した可変 callee-save (leaf-aware)

*4 これらの名称は全て正式ではないが、先頭の文字列をとっている。

表 2: 評価に使用したプロセッサパラメータ

Frontend Width	10
Scheduler	256
ROB (R)	768
LSQ	LQ: 192, SQ: 128
Frontend Latency	RISC-V: 7, STRAIGHT: 5
Execution Units	Int \times 8, iMul \times 2, Load \times 3, Store \times 2, Float \times 4, iDiv \times 1, fDiv \times 1
Phys. Reg.	RISC-V: Unified \times R, STRAIGHT: Unified \times R + 127
Branch Pred.	64 KiB TAGE
BTB	4-way, 8192 entries
RAS	16 entries
Mem. Dep. Pred.	PHAST [5]
L1I/D Cache	128 KiB, 8-way, 3 cycles
L2 Cache	8 MiB, 16-way, 12 cycles
Prefetcher	Stream prefetcher at L2
Main memory	100 cycles

variable と同様の規約設定だが、callee-save の割り当ての際に葉関数かを考慮する。具体的には、葉関数である（あるいは関数を呼ぶパスの頻度が少ない*5）ときのみ、callee-save を割り当てている。

割り当て方針 leaf-aware の意図は以下の通りである。call を含む関数は、callee-save によってレジスタ上で値を受け渡しても、call を跨ぐことによりスピルされることが多い。そのようなケースでは、call の前にスピルを行ってもメモリアクセスの実行回数は変わらない。また、callee-save をする場合、値を所定の位置に配置するコスト（move や nop）がかかる。このため、葉関数以外では callee-save を割り当てないことのメリットが大きい。

シミュレーションは、公開されている STRAIGHT のシミュレーション環境 [6] を用いて行った。評価に用いたプロセッサパラメータを表 2 に示す。これは、近年発売された高性能プロセッサ [9], [11] を参考に設定している。

4.2 実験結果

4.2.1 ロード・ストアの実行数への影響

まず、三つの条件で生成したバイナリについてエミュレーション実行を行い、ベンチマークの実行命令数を測定した。このエミュレーション実行において、可変長 callee-save 規約を用いたバイナリも、ベースラインと同じ出力結果が得られた。すなわち、提案するコード生成手法によって正しく動作するコードが得られることを確認できた。

次に、提案する規約によるコードへの影響を測定結果から確認する。まず、固定長 callee-save のバイナリに対するロード命令とストア命令の実行数の削減率をそれぞれ図 8、

図 9 に示す。ロード命令については、ベンチマーク parser で 1.5%、zip で 2.0% の実行命令数の削減が得られた。ストア命令については、ベンチマーク xz と zip で 2.0–3.0%、parser では 10% 以上の実行命令数の削減が得られた。

一方で、いくつかのベンチマークでは命令の削減効果はほとんど得られなかった。この原因に関して、ロード・ストア命令の実行回数の変化を関数呼び出しの回数で割った値を図 10 に示す。例えば、lbn は削減率はほとんど変わらないものの、この値では変化が現れている。すなわち、適切な callee-save 量を選択できているが、計測区間において関数呼び出しがほとんど行われていないため、実行命令数への影響が少ないと考えられる。逆にこのグラフにおいても値の変化が現れないようなベンチマークは元々の割当量 (3) が最適、あるいは今の callee-save 値の割り当て方では効果が得られないベンチマークと言える。

また、割り当て方針 leaf-aware に関して、図 9 を見るとベンチマーク core でストア命令が増加している。このベンチマークは call が中に存在するが、そのほとんどの実行において call を呼ばない関数を持つ。つまり、これは葉関数の誤判定が原因であり、profile guided optimization を行うことで更に最適化ができることを示唆している。

4.2.2 実行性能への影響

次にシミュレーション実行を行ったときの実行性能（実行サイクル数の逆数）についての比較を図 11 に示す。radix, sha では性能がそれぞれ 3.0%, 2.2% 程度向上したが、その他のベンチマークについては向上しなかった。特にロード・ストア命令の高かった parser ベンチマークについては、全体の実行命令数に対するロード・ストア命令の実行の割合が少なかったことから、性能向上が得られなかった。

性能が向上した radix, sha と全体の実行命令数の削減幅が大きかった zip について、実行命令数の変化を図 12 に示す。この図では、規約を変えることによって実行命令数が変化した命令種のみを、baseline における全実行命令数を 100% として正規化して表示している。図から sha と zip ベンチマークは可変長 callee-save 規約による影響が MOVE と RPINC (NOP) 命令に表れていることがわかる。これは、各関数呼び出しの必要量に合わせて callee-save を行うことにより、不必要な nop やデータの持ち運びである move が減り、それが性能に対して良い影響をもたらしたことを意味している。ベンチマーク radix の性能向上についても、図上の割合が少ないものの同様である。radix では RPINC が実行命令数全体の 1% 削減している。この削減は一つの関数 precise_random_f64 に集中しており、この命令実行においてフェッチ効率が上がったことにより、性能向上が得られたと考えられる。

また、baseline, variable, leaf-aware の順番で命令数が削減している。variable と leaf-aware の差は、内部に call を含む命令に対して callee-save を行うかである。内部で call

*5 静的分岐予測で判定する。例えば expf の例外検知など。

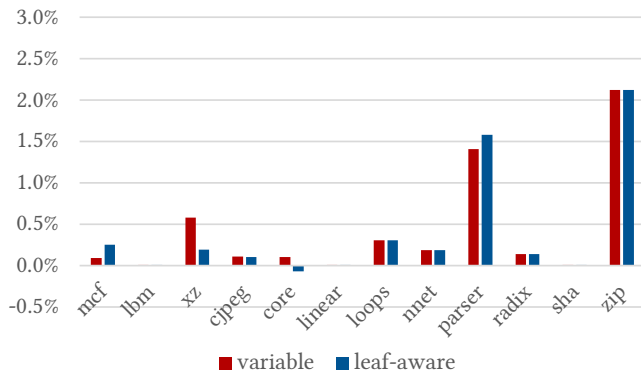


図 8: ロード命令の実行数に対する削減率

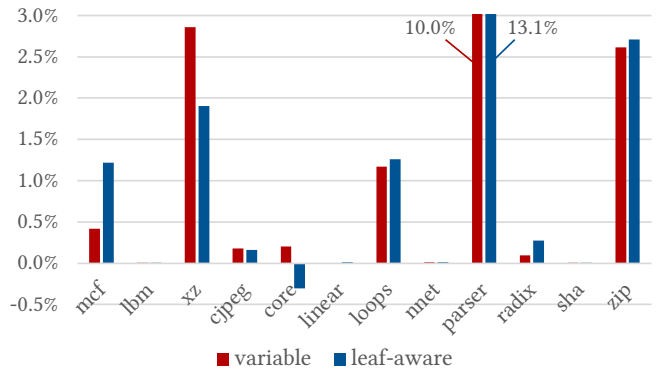


図 9: ストア命令の実行数に対する削減率

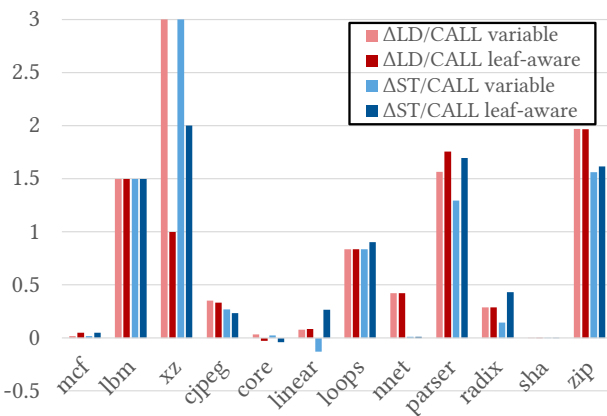


図 10: ロード/ストア命令の実行数の変化量を関数呼び出しの回数で割った平均

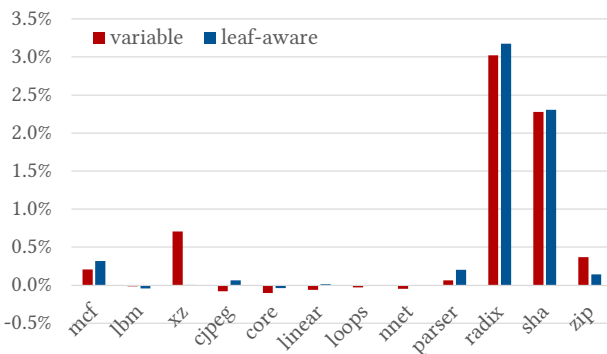


図 11: ベースラインに対する実行性能の向上率

を行うような関数では、callee-save 値を引数として渡されても、関数の prologue, epilogue でスタックに退避/復帰するケースが多い。そのため、call をする前にその値を保存しても変わらず、callee-save をするとかえって値の位置を調整するコストがかかる。これが原因で、leaf-aware は命令数を更に削減できていると考えられる。

4.2.3 バイナリサイズ

最後に、この呼び出し規約の採用によってバイナリサイズがどのように変化するかについて評価した。結果を図 13

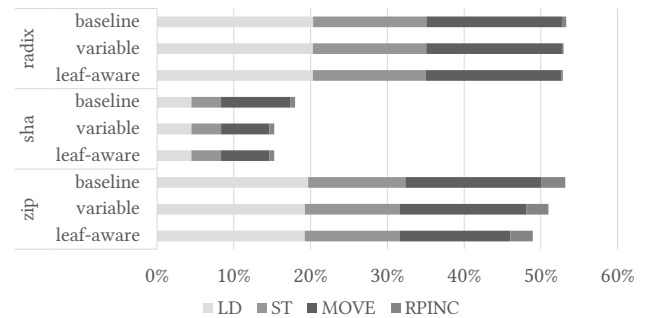


図 12: radix,sha,zip の実行命令数の変化。規約によって実行数が増える命令種のみを、baseline における全実行命令数との比で表示している。

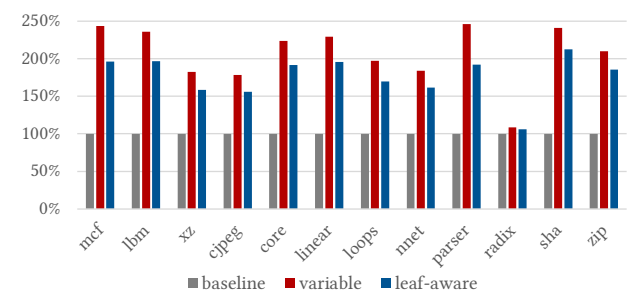


図 13: バイナリサイズの比較。値は baseline を 100% として正規化した。

に示す。条件 baseline と比較したとき、バイナリサイズは幾何平均で、条件 variable が 202%、条件 leaf-aware が 174% となった。

どのベンチマークと条件についても baseline のバイナリサイズの最大 2.5 倍以内に収まっている。これは、デッドコード削除が機能していることを示す。また、variable の条件と leaf-aware の条件だと、leaf-aware の方が小さい。これは葉関数でない関数呼び出しについて、callee-save を設定しないと固定することによって、呼び出す関数の callee-save 数のバリエーションが少なくなるためである。

5. おわりに

命令間距離でオペランドを指定する STRAIGHT では、従来アーキテクチャのようにレジスタごとに caller-save あるいは callee-save するという規約を使えない。そのため、call 後に必要なコンテキストをレジスタ上で保存するためには、引数・戻り値と同様の形でレジスタ渡しをする必要がある。しかし、全ての関数について同じ数だけ引数・戻り値を拡張する既存手法は、保存の必要量とその拡張数が異なる場合に無駄なメモリアクセスや命令の実行が生じていた。

これに対して本研究は、caller と callee で拡張された引数・戻り値の数が一貫する範囲内で自由に callee-save する値の数を決定できる、可変長 callee-save 規約を提案した。そして、新しい規約のコード生成法を提案し、実装したことで実際にこの規約でコード生成・プログラムの実行が可能であることを示した。また、提案規約の採用により性能が悪化したとしてもその割合は無視できるほど小さく、逆に radix, sha のベンチマークで 2-3%性能向上することを示した。

本論文の評価により、提案する可変 callee-save の規約の有効性は、その callee-save の割り当て方やスピルの生成によって大きく変わることが示唆された。今後は、この規約の有効性を最大化するために、profile guided optimization を通じた手続き間解析による更なる最適化や、スパイル最適化の改善に取り組んでいく。

謝辞 本研究の成果は一部、Premo Inc. および JSPS 科研費 23K28050 の助成による。

参考文献

- [1] Amano, T., Kadomoto, J., Mitsuno, S., Koizumi, T., Shioya, R., Irie, H. and Sakai, S.: An Out-of-Order Superscalar Processor Using STRAIGHT Architecture in 28 nm CMOS, *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5 (2023).
- [2] Bucek, J., Lange, K.-D. and v. Kistowski, J.: SPEC CPU2017: Next-Generation Compute Benchmark, *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18*, pp. 41–42 (2018).
- [3] EEMBC: CoreMark-PRO (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, *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 121–133 (2018).
- [5] Kim, S. and Ros, A.: Effective Context-Sensitive Memory Dependence Prediction, *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 515–527 (2024).
- [6] Koizumi, T. et al.: straight-dev/env, GitHub (online), available from <https://github.com/straight-dev/env/>

(accessed 2022/02/24).

- [7] Koizumi, T., Shioya, R., Sugita, S., Amano, T., Degawa, Y., Kadomoto, J., Irie, H. and Sakai, S.: Clock-hands: Rename-Free Instruction Set Architecture for Out-of-Order Processors, *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–16 (2023).
- [8] 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)*, pp. 400–408 (2021).
- [9] Lam, C.: Lion Cove: Intel's P-Core Roars (2024).
- [10] Matsuo, R., Koizumi, T., Irie, H., Sakai, S. and Shioya, R.: Enhancing GPU Performance through Complexity-Effective Out-of-Order Execution using Distance-based ISA, *IEICE Transactions on Information and Systems*, Vol. E108.D, No. 6, pp. 558–569 (2025).
- [11] Schor, D.: Arm Launches Next-Gen Flagship Cortex-X925 (2024).
- [12] Tatsumi, Y. and Mattausch, H. J.: Fast quadratic increase of multiport-storage-cell area with port number, *Electronics Letters*, Vol. 35, pp. 2185–2187 (1999).