

自動メモ化プロセッサにおける 再利用区間の最適化に向けた調査

久保田 孝弘¹ 森 隆志¹ 宮脇 佑太¹ 小泉 透¹ 塩谷 亮太² 五島 正博³ 津邑 公暁¹

概要：

ゲート遅延に対する配線遅延の相対的な増大から、微細化による高クロック化だけではマイクロプロセッサの性能向上を実現することは困難になってきた。こうした中で、命令間の並列性に着目した高速化手法が注目されるようになった。しかし、プログラム中の並列性を持つ部分全てに並列化を適用するのは難しく、これらの手法による性能向上にも限界がある。これに対し、計算再利用に基づいた高速化手法を採用した自動メモ化プロセッサが提案されている。自動メモ化プロセッサは、計算再利用を自動的に適用可能なプロセッサであり、関数実行時にその関数の入出力を再利用表と呼ばれるルックアップテーブルに記憶する。その後、同一関数を同一入力により再実行しようとした際に、過去に記憶した出力を利用することでその実行自体を省略する。本研究では、自動メモ化プロセッサにおける再利用区間の最適化に向けて、再利用性の高い命令列の特徴について調査を行った。従来の自動メモ化プロセッサは再利用区間を関数単位としていたが、関数の一部に存在する再利用性を活用できないという問題があった。そこで、SPEC CPU2017を用いたトレース解析により、再利用性の高いシーケンスの特徴を分析した。その結果、出現頻度および命令数の観点から、ステンシル計算を含む命令列が多く存在することを確認した。さらに、ステンシル計算の命令列が、複数の load 命令で始まり store 命令・load 命令で終了する構造を持つことに着目し、再利用区間の開始および終了をそれぞれ連続した load 命令および store 命令として定義した。評価の結果、ステンシル計算を多く含むベンチマークにおいて、最大 36.24% の命令を削減できることを確認した。

1. はじめに

21 世紀初頭までは、ムーアの法則およびデナード・スケーリングに基づいて、集積回路の微細化によりクロック周波数を向上させることで、マイクロプロセッサは高速化を続けてきた。しかし、ゲート遅延に対する配線遅延の相対的な増大 [1] や、微細化に伴うリーク電力および発熱量の増大 [2] といった問題から、クロック周波数の向上によるマイクロプロセッサの高速化は困難になっている。 [3]

こうした中で、SIMD 命令やスーパスカラなどの細粒度な並列性に基づく高速化手法が注目されてきた。しかし、一般にプログラム中から細粒度な並列性を抽出することは容易ではなく、またプログラム中に存在する細粒度な並列性にも限りがある。よって、これらの手法による高速化を目指す一方、消費電力や発熱量の問題を解決しつつ、プロセッサあたりの処理能力を向上させるため、クロック周波

数を抑えたコアを 1 つの CPU に複数搭載したマルチコアプロセッサが広く普及している。そして、このマルチコアプロセッサを有効活用するために、スレッドレベル並列性 (TLP: Thread Level Parallelism) に着目した様々な高速化手法が研究されている。これらの手法は、プログラムを複数スレッドで処理できるように分割し、それらをそれぞれのコアに割り当てることで高速化を図っている場合が多い。しかし、プログラマが明示的に並列処理プログラムを記述することは容易ではないという問題がある。これらの高速化技術は、いずれも複数の処理を並列実行することで、処理の総量は変化させずに高速化を図るものである。

これに対し、過去の実行結果を記憶しておき、その結果を再利用することで処理の総量自体を削減し高速化を図る、計算再利用と呼ばれる高速化手法がある。この計算再利用を専用ハードウェアを用いて、プログラム中の一部の処理に自動的に適用する、**自動メモ化プロセッサ** [4] に関する研究が行われてきた。自動メモ化プロセッサは関数を計算再利用の対象区間とし、実行時に関数の入力と出力の組を再利用表と呼ばれる専用表へ登録する。そして、再び同一関数を同一入力を用いて実行しようとした際に、再利用

¹ 名古屋工業大学
Nagoya Institute of Technology

² 東京大学
The University of Tokyo

³ 国立情報学研究所
National Institute of Informatics

用表に登録されている過去の出力を再利用することで、その関数の実行を省略する。

従来の自動メモ化プロセッサでは再利用区間を関数単位にしていた。これは再利用区間の開始は関数呼び出し、終了は関数復帰と再利用区間の開始と終了が明確だからである。しかし、関数単位の再利用には2つの問題点がある。1つ目は、関数の一部に再利用性があっても再利用できないことである。関数単位で再利用を行うため、関数の一部に再利用性があっても、関数全体で再利用性がなければ再利用することはできない。2つ目は、再利用表に登録する情報の多い関数は再利用の対象としないことである。入出力数が多い関数や関数呼び出しによる深い入れ子構造の関数に関しては再利用表に全ての情報を記録することができないため、再利用の対象としていない。これらの要因により、従来の再利用区間が関数単位の自動メモ化プロセッサではメモ化をした場合としなかった場合の結果に大きな差が見られない、もしくは悪化することもあった。本研究では、これらの問題点を解決し、自動メモ化プロセッサの性能を向上させる適切な再利用区間の特定を目指して調査を行った。以下2章では自動メモ化プロセッサについて説明する。3章では従来方式の問題点について説明する。4章では再利用区間の特定に向けて行った調査の結果を示し、5章では今後の方針について説明する。最後に6章で結論を述べる。

2. 自動メモ化プロセッサ

2.1 概要

計算再利用 (Computation Reuse) とは、プログラム中で定義された関数において、その入力組 (**入力セット**) と出力組 (**出力セット**) を実行時に記憶し、再び同じ入力セットでその関数が実行されようとした場合に、過去の記憶された出力セットを再利用することで、その関数の実行自体を省略する高速化手法である。また、この計算再利用を各関数に適用することを**メモ化 (memoization)** [5] と呼ぶ。メモ化は元来、高速化のためのプログラミングテクニックである。しかし、メモ化を適用するためには、プログラムを記述しなおす必要があり、既存のロードモジュールをそのまま高速化することはできない。その上、ソフトウェアによる自動メモ化 [6] はオーバーヘッドが大きく、限られたプログラムでしか性能向上が得られない。

そこで、ハードウェアを用いて自動的にメモ化を施す**自動メモ化プロセッサ (Auto-Memoization Processor)** [4] が研究されている。自動メモ化プロセッサは、プログラムの実行時に動的に関数を検出しメモ化を施すことで、既存のバイナリを変更することなく高速に実行できる。なお自動メモ化プロセッサは、関数呼び出し命令による遷移先の命令から、対応する関数からの復帰命令までの区間を関数として検出する。

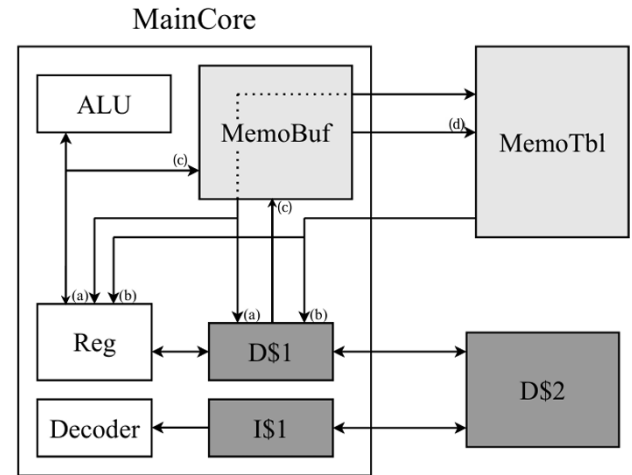


図 1: 自動メモ化プロセッサのハードウェア構成

自動メモ化プロセッサのハードウェア構成の概略を図1に示す。自動メモ化プロセッサは、一般的なプロセッサと同様に、ALU、レジスタ (Reg)、命令キャッシュ、1次データキャッシュ (D\$1)、2次データキャッシュ (D\$2) を持つ。また、自動メモ化プロセッサ独自の機構として、関数およびその入力と出力の組 (入出力セット) を記憶しておくルックアップテーブルである**再利用表 (MemoTbl)**、および MemoTbl への書き込みバッファとして働く**再利用バッファ (MemoBuf)** を持つ。MemoTbl はサイズが大きい、コアからのアクセスレイテンシが大きいため、入出力を検出する度に MemoTbl へアクセスするとオーバーヘッドが大きくなる。そこで、このオーバーヘッドを軽減するために、作業用の小さなバッファである再利用バッファをコアの内部に設けている。なお、自動メモ化プロセッサでは、レジスタやメモリの参照を「入力」、レジスタやメモリへの書き込み、および返り値を「出力」として扱う。

自動メモ化プロセッサは関数呼び出しが実行されると関数を検出する。関数を検出すると、MemoTbl を参照し、検出された関数の入力セットと MemoTbl に記憶されている過去の入力セットとを比較する。これを再利用テスト (reuse test) (図1(a)) と呼ぶ。もし、現在の入力セットが MemoTbl に記憶されたいずれかの入力セットと一致する場合、その入力セットに対応する出力セットをレジスタやキャッシュに書き戻し (writeback) (図1(b))、関数の実行を省略する。一方、現在の入力セットが MemoTbl のいずれの入力セットとも一致しない場合、自動メモ化プロセッサはその関数を通常実行しながら、その入出力を再利用バッファに登録 (図1(c)) し、実行終了時に再利用バッファの内容を MemoTbl に登録 (store) (図1(d)) することで将来の再利用に備える。

さて、一般に関数内では、複数の入力値が順に参照され、使用される。しかし、同じ関数でも、その入力アドレスの

```

1 int a = 3, b = 4, c = 8;
2 int calc(int x){
3     int tmp = x + 1;
4     tmp = tmp + a;
5     if(tmp < 7)
6         tmp = tmp + b;
7     else
8         tmp = tmp + c;
9     return(tmp);
10 }
11 int main(void){
12     int ans = 0;
13     calc(2); /* x = 2, a = 3, b = 4 */
14     b = 5; ans = calc(2); /* x = 2, a = 3, b = 5 */
15     a = 5; ans = calc(2); /* x = 2, a = 5, c = 8 */
16     a = 3; ans = calc(2); /* x = 2, a = 3, b = 5 */
17     return(0);
18 }

```

図 2: サンプルコード 1

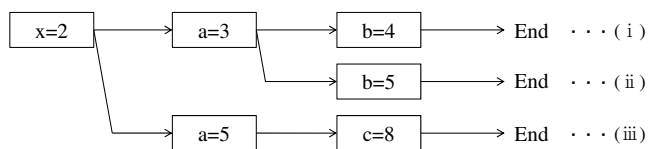


図 3: 入力パターンの木構造

列は分岐していく場合がある。ここで、入力アドレスとは、入力セットに含まれる入力値が記録されているレジスタまたはメモリアドレスであり、一つの入力セットに含まれる入力アドレスをまとめて入力アドレスの列と呼ぶ。例えば、条件分岐命令を実行すると、次に参照される入力アドレスはその条件分岐命令の分岐結果によって変化してしまう。これらの入力セットを、分岐前の共通部分も含めて別々に記録すると、再利用表を無駄に消費してしまう。

そこで、自動メモ化プロセッサは、全入力パターンを木構造で表現し、MemoTbl に登録する。例えば、自動メモ化プロセッサが図 2 に示すサンプルプログラムを実行する場合、関数 calc の全入力セットは図 3 に示すような木構造で表現されることになる。なお、図 3 のノードは関数の入力値を、エッジは入力値と次に参照される入力値との対応関係を示しており、End はそれ以上の入力値が存在しないことを示す。また、図 3 の (i) は図 2 の 13 行目、(ii) は 14 行目および 16 行目、(iii) は 15 行目における関数呼び出し時の入力セットに対応する。この例において、入力セット (i) および入力セット (ii) では、変数 b が 3 番目に参照されるのに対して、入力セット (iii) では、変数 c が 3 番目に参照される。これは、変数 a の値が異なり、プログラムの 5 行目における if 文の条件を満たすか否かが変わったためである。

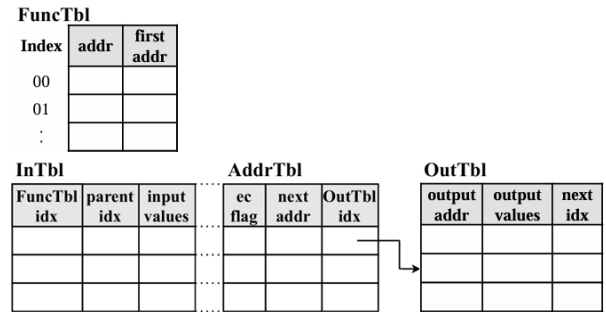


図 4: MemoTbl の構成

次に、この木構造で表現された全入力パターンを登録するためのルックアップテーブルである MemoTbl の構成を図 4 に示す。MemoTbl は関数を記憶する関数表 (FuncTbl)、入力を記憶する入力表 (InTbl)、入力アドレスを記憶するアドレス表 (AddrTbl)、および出力を記憶する出力表 (OutTbl) の 4 つの表から構成される。FuncTbl、AddrTbl、OutTbl は RAM で実装され、InTbl は 3 値 CAM (Ternary Content Addressable Memory) で実装される。CAM は全てのエントリに対して、一致比較を同時に行うことができるため、RAM と比較して高速な連想検索が可能である。このとき、キャッシュライン中の入力として参照されている値が格納されている部分以外については 3 値 CAM のドントケアを用いて登録する。このようにするため、CAM ではなく 3 値 CAM を用いる。

FuncTbl は 1 エントリが 1 関数に対応しており、その行番号 (Index) を各関数の識別番号とする。addr には記録する関数の関数アドレスを記録し、first addr は関数において初めに参照されるレジスタまたはメモリのアドレスを記録する。この表を用いて、再利用テストを行う関数の入出力が再利用表に記録されているか、および再利用テスト時に初めに一致比較を行う入力のアドレスを確認する。

InTbl の各エントリは FuncTbl の行番号 Index を格納するフィールド (FuncTbl idx) を持ち、この値を用いてどの関数の入力値を記憶しているかを判別する。また、関数の全入力パターンを木構造で管理するために、入力値 (input values) に加えて親エントリのインデックス (parent idx) を持つ。この input values は、入力を含むキャッシュライン全体を記憶する。これは、関数で用いられる複数の入力のうち、同じキャッシュライン中に存在するものがあった場合に、それぞれの値の一致比較を同時に行うことができるためである。例えば、ループ構造で配列の各要素にアクセスするような場合、各要素が記録されているメモリアドレスは連続していることが予想される。そのため、もしその配列の各要素が関数の入力であった場合に、複数の要素を同時に一致比較することで、再利用テストにかかる時間を短縮することができる。このとき、キャッシュライン中の、入力として参照される値が格納されている部分以外について

ては、3 値 CAM のドントケアを用いて登録する。なお、レジスタの値が入力値となる場合も同様に、複数レジスタの入力値をまとめて 1 つの入力エントリに記憶する。

AddrTbl は InTbl と同数のエントリを持ち、各エントリは 1 対 1 に対応する。AddrTbl の各エントリは入力値検索のために、次に参照すべきアドレス (next addr) を持つ。また、入力セットの終端エントリか否かを保持するフラグ (ec flag) を持ち、出力を記憶している表である OutTbl のエントリを指すインデックス (OutTbl idx) も持つ。

OutTbl の各エントリは関数の出力先のアドレス (output addr), 出力値 (output values) を持つ。また、出力セットの各エントリをリスト構造で管理するため、次に参照すべきエントリのインデックス (next idx) を持つ。

このような MemoTbl への書き込みバッファが MemoBuf である。MemoBuf の詳細な構成を図 5 に示す。MemoBuf は複数のエントリを持ち、1 エントリが 1 つの入出力セットに対応する。各エントリは、関数の開始アドレス (StartAddr), その関数の実行開始時のスタックポインタ (SP), 関数の入力セット (Read) および出力セット (Write) のためのフィールドを持つ。この、Read フィールドおよび Write フィールドは、複数の入出力値を保持できるようになっている。なお、SP は関数の入出力判定時に関数スタック内に含まれる値 (ローカル変数) の参照を入出力の対象として記録しないようにするために使用する。これは、自動メモ化プロセッサは関数のすべての処理に対して再利用するため、そもそも関数を再利用する場合は関数の関数スタックを用意する必要がなく、関数内で定義されるローカル変数を入出力として記録する意味がないためである。また、入れ子構造になった関数もメモ化対象とするために、MemoBuf は現在実行中の関数に対応する各エントリをスタック構造として保持する。そのため、MemoBuf は現在使用しているエントリをポインタ (MemoBuf_top) で指しており、関数の検出時にそのポインタをインクリメントし、関数の実行終了時にデクリメントすることで入れ子構造に対応している。このスタック構造は、リングバッファをスタックとして扱うことで実現する。なお、スタックがいっぱいになった場合でもポインタのインクリメントは行われる。これは最も古いエントリが上書きされることになるが、そもそもそのような場合には関数が深い入れ子構造となっていることを示しており、そのエントリに記録されている情報を MemoTbl に記録しても再利用の機会が見込めないことが予想されるため、大きな問題とはならない。

2.2 自動メモ化プロセッサの動作例

計算再利用を適用するためには、現在の入力セットと MemoTbl に記憶しておいた過去の入力セットとを比較する必要がある。MemoTbl の検索手順を図 6 に示す。ここで、図 6 の MemoTbl は図 2 を 15 行目まで実行した状態

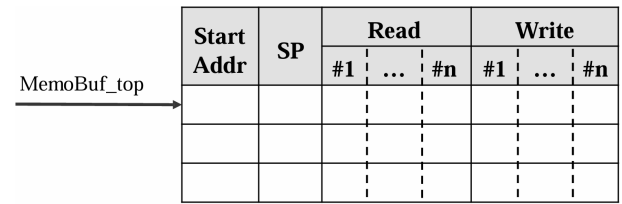


図 5: MemoBuf の構成

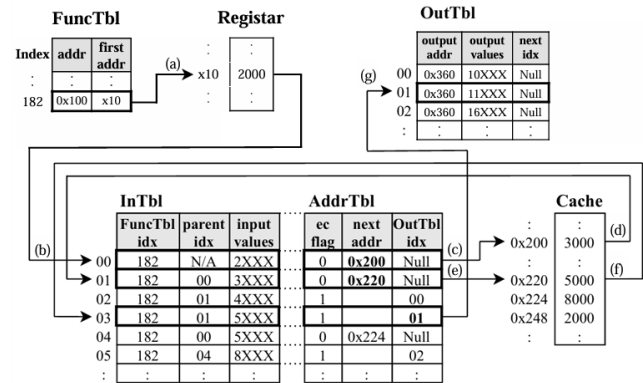


図 6: MemoTbl の検索手順

を表している。なお、図中の InTbl における X はドントケアを示しており、キャッシュライン内のその値は検索時に比較されない。また、図中の InTbl の parent idx における “N/A” は親エントリが存在しないこと、すなわち、parent idx の値が “N/A” であるエントリは図 3 で示したような入力パターンの木構造の根に相当するエントリであることを示している。また、図 6 内の矢印は図 2 における関数呼び出し時の MemoTbl 検索フローを示している。

次に、どのような手順で計算再利用を適用するのかを、図 2 の 16 行目を実行する場合を例に説明する。16 行目を実行し関数 calc が呼び出され、再利用区間の実行開始アドレスが検出されると、まず、FuncTbl が関数 calc の開始アドレス (0x100) をキーとして検索される。このエントリの first addr が x10 を指しているため、レジスタの x10 を参照する (図 6(a))。そして、関数 calc の開始アドレスである 0x100 を FuncTbl idx フィールドに持ち、input values が現在のレジスタ上の入力値と一致し、かつ parent idx が “N/A” であるようなルートエントリが検索される (図 6(b))。次に、該当するエントリがライン 00 で発見され、対応する AddrTbl の next addr が 0x200 番地を指しているため、そのアドレスに対応するキャッシュラインを参照する (図 6(c))。そして、得られた値を input values として持ち、かつ parent idx が 00 であるエントリを InTbl から検索する (図 6(d))。以降、同様に検索を続ける (図 6(f))。その後、検索対象のエントリがライン 03 で発見されたとき、当該エントリの ec flag が 1 であることが検出される。これは、当該エントリが入力セットの終端エントリであることを表しているため、一連の入力セットの比較を終え

```

1 int a = 3, b = 4, c = 8;
2 int calc(int x){
3     int tmp = x + 1;
4     tmp = tmp + a;
5     int rand = random.nextInt(14);
6     if(rand < 7)
7         tmp = tmp + b;
8     else
9         tmp = tmp + c;
10    return(tmp);
11 }
12 int main(void){
13     int ans = 0;
14     ans = calc(2); /* x = 2, a = 3, b = 4, c =
15                    8*/
16     ans = calc(2); /* x = 2, a = 3, b = 4, c =
17                    8 */
18    return(0);
19 }

```

図 7: サンプルコード 2

る。なお、検索が終端エントリに達するのは、現在の入力セットと過去の入力セットが全て一致した場合のみであるため、自動メモ化プロセッサは再利用テストに成功したと分かる。このとき、終端エントリに対応する AddrTbl エントリの OutTbl idx に登録されているインデックスが指す OutTbl エントリを参照し (図 6(g)), 読み出した出力値をレジスタやメモリに書き戻すことで関数の実行を省略することができる。

3. 従来方式の問題点

3.1 関数単位の問題点

従来の自動メモ化プロセッサでは再利用区間が関数単位となっている。これは 1 章でも述べたように再利用区間の開始と終了が明確だからである。しかし、再利用区間が関数単位となっていることで再利用の機会を損失する場合がある。

図 7 のサンプルコード 2 は図 2 のサンプルコード 1 の関数 calc を、random 関数を用いて 0 以上 14 未満のランダムな値を持つ変数 rand を追加し、if 文の分岐の条件を「変数 tmp が 7 未満」から「変数 rand が 7 未満」に変更したコードである。サンプルコード 2 中の 14 行目と 15 行目はどちらも関数 calc を入力となる変数 x, a, b, c を同じ値で実行している。サンプルコード 1 の関数 calc の場合、サンプルコード 2 中の 14 行目、15 行目のような同じ入力で行うものは再利用することができる。しかし、サンプルコード 2 の関数 calc では再利用することはできない。3 行目から 4 行目は同じ入力であれば常に同じ計算処理を行うが、5 行目の変数 rand が関数実行ごとに異なる値となり、6 行目以降が変数 rand の値により計算処理も関数実行ごとに異なるためである。この場合、再利用区間を 3 行目

から 4 行目の部分のみに変更すればこの部分は同じ入力であれば常に同じ計算処理を行うため再利用ができる。しかし、従来方式では関数単位で再利用をするため、関数内の一部に再利用できる部分があったとしても、関数内の一部が再利用できないだけで再利用ができなくなる。このように再利用区間が関数単位となっていることで再利用の機会を損失する場合がある。

3.2 従来方式が再利用の対象としない部分の再利用性

ここまで、従来方式の自動メモ化プロセッサでは関数を再利用の対象としていると述べていたが、すべての関数を再利用の対象とはしていない。再利用の対象にならない関数には主に以下の 2 種類がある。

- MemoBuf に記憶しておく入出力数が多い関数
- 関数呼び出しによる深い入れ子構造の関数

まず、1 つ目の MemoBuf に記憶しておく入出力数が多い関数が再利用の対象にならない理由は、MemoBuf の 1 エントリに格納できる入出力データの数に物理的な上限があるためである。MemoBuf は複数のエントリを持ち、1 エントリが 1 入出力セットに対応している。そのため、1 エントリに格納できる入出力データの数には物理的な上限があり、関数がこの上限を超える数の入力や出力を持つ場合、その入出力セットを 1 つのエントリ内に収めることができず、再利用表への登録自体が不可能となるためである。そのため MemoBuf に記憶しておく入出力数が多い関数が再利用の対象にならない。

また、2 つ目の関数呼び出しによる深い入れ子構造の関数が再利用の対象とならないのは、MemoBuf のエントリ数の物理的な制約が原因である。2 章で述べたように入れ子構造 (ネスト) を持つ関数を再利用の対象とするために、MemoBuf は現在実行中の各階層に対応するエントリをスタック構造として保持しているが、この MemoBuf の物理的な容量には上限があり、関数呼び出しによってスタックが不足した場合は他の入出力情報で上書きされるためである。そのため関数呼び出しによる深い入れ子構造の関数が再利用の対象にならない。

4. 新たな再利用区間に向けた調査

4.1 新たな再利用区間の方針

そこで本研究では関数単位に代わり、自動メモ化プロセッサの性能を向上させる新たな再利用区間の定義を目指して調査を行った。新たな再利用区間では再利用対象として関数に縛られない実行命令列の連続する一部分とする。これは 2 つの利点が考えられるためである。

- 関数単位では再利用できなかった関数でも、その一部のみであれば再利用できること
- 関数単位では再利用の対象としていなかった関数も再利用できること

従来の関数単位の再利用では、関数呼び出しから関数復帰までの全てを再利用するため、関数内の一部に再利用できる部分があったとしても、関数内の一部が再利用できないだけで再利用ができなくなる。また、MemoBuf に記憶しておく入出力数が多い関数や関数呼び出しによる深い入れ子構造の関数、関数外の部分に関しては再利用の対象としていないため、これらに再利用ができる箇所が含まれていても再利用することはできなかった。しかし、再利用区間を関数に縛られないコードの一部とすれば関数単位で再利用できなかった関数や対象としなかった関数も、関数内の再利用できる箇所のみを切り取って再利用することが出来る。

新たな再利用区間の要件として主に 2 つが考えられる。1 つ目はプログラム中の出現回数が多い区間にあることである。再利用表のサイズは有限のため記録しておける入出力セットの数も有限であり、再利用対象となる全ての入出力セットを記録するのは不可能なためである。そのため再利用対象となる区間の 1 つ 1 つの出現回数は出来る限り多いことが望ましい。2 つ目は再利用区間がある程度の長さを持つことである。これは再利用区間が大きいほど再利用できた時には大量の命令列の計算処理を省略でき、メリットが大きいためである。この 2 つの要件を満たす再利用区間の特定を目指して調査を行った。

4.2 再利用性が高いシーケンスがもつ特徴の調査

4.2.1 調査内容

新たな再利用区間の調査として、再利用性が高い再利用可能シーケンスがもつ特徴について調査を行った。まずシーケンスを pc, srcReg の値, dstReg の値の組からなる列と定義し、プログラム実行列に複数回出現するシーケンスを再利用可能シーケンスと定義する。また、ここでの再利用性が高いとは「再利用可能シーケンスの出現回数が多い」、「再利用可能シーケンスの長さが長い」という 2 つの条件を満たすことである。ここで、シーケンスの長さとはシーケンスに含まれる命令数とする。本調査では、各ベンチマークについて、エミュレータ上でプログラムを実行し、得られたトレースを用いて解析を行った。トレースの生成には、汎用的なマシンエミュレータである QEMU 8.0.94 [7] を用いた。命令セットには RISC-V を用いた。ベンチマークには、The Standard Performance Evaluation Corporation が提供する SPEC CPU2017 を用いた。コンパイラには GCC 12.2.0 を用い、最適化オプションには -O2 を指定した。また、プログラム全体の実行挙動を網羅しつつ効率的に解析するため、統計的サンプリング手法である SimPoint を用いて代表区間を選定した。具体的には、まず各ベンチマークについて全実行命令を 10M 命令ごとのインターバルに分割し、各インターバルにおける基本ブロックベクトルを抽出した。次に、これらのベクトルに対して

```
1 for (int i = 1; i < N-1; i++) {
2   for (int j = 1; j < M-1; j++) {
3     B[i][j] = (A[i][j]      // 自分
4               + A[i-1][j]   // 上
5               + A[i+1][j]   // 下
6               + A[i][j-1]   // 左
7               + A[i][j+1]) // 右
8               / 5.0;
9   }
10 }
```

図 8: サンプルコード 3

K-means 法を用いたクラスタリングを行い、プログラムの実行フェーズを分類した。クラスタ数 K の決定には、統計的指標である BIC 法 (Bayesian Information Criterion) を用い、各ベンチマークにおいて最適フェーズ分割を行った。得られたクラスタのうち、実行時間への寄与が大きい上位 5 つを選出し、各クラスタの重心に最も近いインターバルを代表区間として調査対象とした。このトレースを用いて、トレースうちのすべての再利用可能シーケンスに対し、「出現回数」と「長さ」の積を計算した。この値を ReuseScore と定義する。各トレースについて、ReuseScore が上位 30 件の再利用可能シーケンスを抽出した。本研究ではトレースは 100 個であるため、合計 3000 件の再利用可能シーケンスが得られる。さらに、これら 3000 件の中から、ReuseScore が上位 100 件の再利用可能シーケンスを抽出した。以上で得られたシーケンスを対象として、再利用性の高いシーケンスが持つ特徴について調査を行った。

4.2.2 調査結果

調査の結果、100 個中 18 個がステンシル計算に該当するシーケンスであることが分かった。ステンシル計算とは、格子状に配置されたデータに対して、各点の値をその周囲の近傍点の値から計算する処理である。サンプルコード 3 に、簡単なステンシル計算の例を示す。このコードでは、2 次元格子上の各格子点 (i,j) に対して、その周囲の近傍点の値を用いて値を更新している。具体的には、格子点 (i,j) の値は、自身および上下左右の 4 近傍の値の平均として計算される。このように、各格子点の値を隣接する格子点の値に基づいて計算する処理は、ステンシル計算と呼ばれる。このステンシル計算を含むベンチマークは多くあり、603.bwaves, 607.cactuBSSN, 619.lbm, 621.wrf, 649.fotonik3d, 654.roms の 6 個がステンシル計算を含む代表的なベンチマークである。そのためステンシル計算を再利用の対象にすることでこれらのベンチマークにおいて計算削減による性能向上が期待できる。

4.3 ステンシル計算に対応した再利用区間

ステンシル計算を再利用の対象とするためには、ステンシル計算に対応した再利用区間の開始および終了を定義す

```
1 ld      s0, 0(s8)          # A[i][j]
2 ld      s1, 0(s7)          # A[i-1][j]
3 ld      s2, 0(s9)          # A[i+1][j]
4 ld      s3, -8(s8)         # A[i][j-1]
5 ld      s4, 8(s8)          # A[i][j+1]
6
7 add     s5, s0, s1
8 add     s5, s5, s2
9 add     s5, s5, s3
10 add    s5, s5, s4
11 div    s6, s5, 2
12
13 sd      s6, 0(s10)
```

図 9: サンプルコード 4

る必要がある。ステンシル計算の大まかな流れをサンプルコード 4 を用いて説明する。サンプルコード 4 は、サンプルコード 3 をアセンブリレベルで表現したものである。ステンシル計算は、大きく 3 つのフェーズに分けられる。第 1 のフェーズは、計算に使用する周囲の近傍点の値を load するフェーズである。第 2 のフェーズは、load した値を用いて計算を行うフェーズである。第 3 のフェーズは、計算結果をメモリに store するフェーズである。サンプルコード 3 を単純にアセンブリへ変換した場合、近傍点の値の load と計算が交互に繰り返される。しかし、このままでは効率が悪いため、コンパイラは最適化により、サンプルコード 4 のように複数の load をまとめて実行し、その後に計算をまとめて実行する形へ変換する。このことから、ステンシル計算に対応する命令列は、複数の load 命令で始まり、store 命令で終わる構造を持つと考えられる。また、store 命令に関しても load 命令同様に複数の store 命令がまとめて実行される。加えて、ある点の計算直後に直ちに store せず、次の点の計算に必要な近傍点の値を続けて load し、複数の点の計算が終わった後にまとめて store する場合がある。したがって、本研究では、ステンシル計算に対応した再利用区間として、再利用区間の開始を 3 つの連続した load 命令、終了を 3 つの連続した store 命令もしくは 3 つの連続した load 命令と定義する。ここで、再利用区間の開始を 3 つの連続した load 命令、終了を 3 つの連続した store 命令・load 命令とした理由について述べる。本研究では、複数の条件を試行した結果、これらの設定が最も多くのステンシル計算を適切に捉えられることを確認した。したがって、本研究では実験的に良好な結果が得られた設定として採用する。

4.4 ステンシル計算に対応した再利用区間の再利用性の調査

4.4.1 調査内容

ステンシル計算に対応した再利用区間が実際に再利用性があるかの調査を行う。調査にあたっては、プロセッサシ

ミュレータを用いるのではなく、理想的な状況における削減命令数をモデル上で算出した。調査方法として、まず、ステンシル計算に対応した再利用区間に該当するシーケンスを全て記憶しておき、このシーケンスの出現回数とシーケンスの長さを記録しておく。「シーケンスの出現回数-1」と「シーケンスの長さ」の積の合計を計算する。この値は再利用表のサイズを無限とした時の削減命令数となる。ここで「シーケンスの出現回数」から 1 引くのは最初の出現では再利用できず、命令の実行を省略できないためである。本調査では、ステンシル計算を含む 6 個のベンチマークを調査対象とした。また、測定対象とする区間は、4.2 節の調査に用いた区間とした。

4.4.2 削減命令数の割合

表 1: 削減命令数の割合

	603	607	619	621	649	654
区間 1	0.00%	0.10%	0.00%	14.63%	0.00%	17.27%
区間 2	0.00%	0.00%	0.00%	0.00%	20.12%	8.20%
区間 3	8.47%	0.00%	0.00%	26.26%	0.00%	0.00%
区間 4	0.00%	0.00%	0.00%	10.52%	36.24%	0.01%
区間 5	0.00%	0.08%	0.00%	6.73%	0.00%	31.10%
average	1.69%	0.03%	0.00%	11.63%	11.28%	11.32%

表 1 はベンチマークの区間ごとの全命令のうちのステンシル計算に対応した再利用区間により削減できた命令数の割合を示している。縦軸が区間、横軸がベンチマークの種類となっている。区間は実行時間への寄与が大きい順となっている。603.bwaves では平均して 1.69%、607.cactuBSSN では 0.03%、619.lbm では 0.00%、621.wrf では 11.63%、649.fotonik3d では 11.28%、654.roms では 11.32%の命令を削減できたことができた。619.lbm では全ての区間で、削減できた命令数の割合は 0.00%となった。ここで、この原因が単にこれらの区間にステンシル計算が含まれないことによるものかを確認するために 619.lbm の今回調査した 5 つの区間で実行された命令の内、ステンシル計算に含まれる命令の割合を求めた。結果としてどの区間でも約 15%の命令がステンシル計算に含まれる命令であった。これらの区間に含まれていたステンシル計算をアセンブリにしたものを確認すると、3 つの load 命令から始まり、3 つの store 命令で終わっており、今回の再利用区間の再利用の対象になっている。ただ、これらの区間に含まれたステンシル計算に含まれる命令の入出力をみると、入出力のパターンが多かった。そのため、ステンシル計算を実行するたびに異なる計算結果となるため、命令数を削減することはできなかったと考えられる。また、603.bwaves と 607.cactuBSSN では比較的命令削減数が少なかった。この原因を調査するために、619.lbm と同様に全ての区間で実行された命令の内、ステンシル計算に含まれる命令の割合

を求めた．一部の区間はステンシル計算を含んでいなかったが，その他の区間ではステンシル計算を含んでいた．これらのステンシル計算も 619.lbm 同様に，今回の再利用区間の対象になっていたが，入出力のパターンが多かった．これが原因となり命令数を削減することはできなかったと考えられる．比較的命令削減数が多いベンチマークである 621.wrf, 649.fotonik3d, 654.roms でも命令削減数が 0.00% の区間がある．これらのベンチマークの全ての区間に対しても同様に，ステンシル計算のために実行された命令の割合を求めると，命令削減数が 0.00% の区間ではステンシル計算が全く含まれない，または割合が低いことが分かった．そのため，これらのベンチマークで命令削減数が 0.00% の区間があるのはこれらのベンチマークに含まれるステンシル計算に再利用性がないのではなく，単にその区間にステンシル計算が含まれないためである．よって，603.bwaves, 607.cactuBSSN, 619.lbm に含まれるステンシル計算は再利用性が低く，621.lbm, 649.fotonik3d, 654.roms に含まれるステンシル計算は再利用性が高いと分かる．

4.4.3 シーケンスの出現回数の調査

表 2 はベンチマークごとに命令削減数が最大の区間のステンシル計算に対応した再利用区間に該当するシーケンスすなわち，3つの連続した load 命令で始まり，3つの連続した store 命令・load 命令で終わるシーケンスの出現回数の分布を示している．619.lbm は命令削減数がすべての区間で 0.00% だったため除外した．縦軸が出現回数，横軸がベンチマークの種類となっている．全てのベンチマークで出現回数が 1 回のシーケンスの割合が最も多い．シーケンスの内容を確認するとステンシル計算を行っていないシーケンスが多くを占めていた．これはステンシル計算に対応した再利用区間の開始を 3つの連続した load 命令，終了を 3つの連続した store 命令・load 命令としたが，この再利用区間にステンシル計算以外の部分も該当してしまったためであり，これらのシーケンスは再利用性がないことが分かる．603.bwaves では出現回数が 1 回のシーケンスを除くと，出現回数が 2～10 回以下のシーケンスが大半を占めている．この調査では再利用の対象となるシーケンスは全て記憶したが，実際の自動メモ化プロセッサの再利用表の大きさは有限であり，記憶できるシーケンス数も有限になっている．そのため，今回の再利用区間の自動メモ化プロセッサではこの調査の結果より効果は薄くなると思われる．削減命令数の平均した値が最も低かった 607.cactuBSSN では出現回数が 11 回以上のシーケンスが 8 個しかなく，大半が出現回数が 10 回以下のシーケンスである．そのため削減命令数も少なくなったと思われる．621.lbm は出現回数が 101 回以上のシーケンスがない．それでも削減命令数の平均した値が最も高かったのは出現回数が 11～100 回のシーケンスの長さが長かったためである．649.fotonik3d はこの 5つのベンチマーク中では出現回数が 101 回以上の

シーケンスの個数が最も多くなっている．654.roms はこの 5つのベンチマーク中では出現回数が 11 回以上 100 回以下のシーケンスの個数が最も多いが，出現回数が 1 回のシーケンスの個数も最も多くなっている．

表 2: 各ベンチマークにおけるシーケンス出現頻度の分布

出現頻度	603	607	621	649	654
1 回	114,462	68,967	78,084	23,718	285,382
2-10 回	31,168	387	6,517	866	6
11-100 回	229	4	1,394	57	2,952
101-1k 回	25	4	0	161	0
1k-10k 回	0	0	0	18	0
10k 回-	2	0	0	0	2

4.4.4 再利用性のない再利用対象の個数の比較

再利用を行うためには再利用区間を検出する度に再利用テストを行う必要がある．この再利用テストにはオーバーヘッドがかかるため，削減命令数が多くても再利用テストの回数が多ければサイクル数は悪化する．そのため，再利用できない箇所に関しては再利用の対象としないことが望ましい．そこで，関数単位の再利用で再利用できないにも関わらず，再利用の対象となった個数とステンシル計算に対応した再利用で再利用できないにも関わらず，再利用の対象となった個数，すなわち出現回数が 1 のシーケンスの個数を比較する．関数単位の再利用ではステンシル計算に対応した再利用と条件を揃えるために，再利用表のサイズは無限にし，フィルタリングを OFF にして調査を行う．調査は 603.bwaves, 607.cactuBSSN, 619.lbm, 621.wrf, 649.fotonik3d, 654.roms の 6つのベンチマークで行い，調査は命令削減数が最大の区間で行った．619.lbm のみ実行時間への寄与が大きい区間とした．表 3 はベンチマークごとの再利用区間の種類による再利用性のない再利用対象の個数となっており，縦軸が再利用区間の種類，横軸がベンチマークの種類となっている．関数単位の再利用では 4つのベンチマークで再利用性のない再利用対象の個数が 0 となっているが，これはまずこの区間で 1つの関数も関数呼び出しから関数復帰が行われなかったためである．621.lbm や 654.roms では関数単位では再利用性のない再利用対象の個数が少ないが，一度も関数単位の再利用が成功していない．そのため，これらの 5つのベンチマークではステンシル計算と関数単位の再利用テストによるオーバーヘッドの比較はできない．ただ，649.fotonik3d では関数単位で再利用が成功しており，削減命令数の割合は 9.44% となっている．これはステンシル計算に対応した再利用の削減命令数の割合より少ない．また，再利用性のない再利用対象の個数に関してはステンシル計算の方が関数単位と比べて少なく，再利用テストによるオーバーヘッドも少ないと考えられる．

表 3: 再利用区間の種類による再利用の再利用性のない再利用対象の個数

再利用区間の種類	603	607	619	621	649	654
ステンシル計算	114,462	68,967	88,883	78,084	23,718	285,382
関数単位	0	0	0	9	79,909	80

5. 今後の方針

今回はステンシル計算に対応した再利用区間の開始を 3 個連続した load 命令, 終了を 3 個連続した store 命令・load 命令にした。これにより, 多くのステンシル計算を再利用の対象と出来たが, ステンシル計算以外の箇所も再利用の対象となっていた。それらのステンシル計算以外の箇所のシーケンスは出現回数が 1 回のもが多く, これにより出現回数が 1 回のシーケンスが多く再利用の対象となってしまった。そのため, ステンシル計算のみを再利用の対象とできるような再利用区間の定義方法について調査していきたい。また, ステンシル計算の中でも 603.bwaves, 607.cactuBSSN, 619.lbm に含まれるステンシル計算のように再利用性が低いものと, 621.lbm, 649.fotonik3d, 654.roms に含まれるステンシル計算のように再利用性が高いものがある。調査よりステンシル計算の再利用性は入出力のパターンと相関関係があることが分かった。今回の再利用区間では 603.bwaves, 607.cactuBSSN, 619.lbm に対してはあまり, 効果が見られなかった。ただ, これらのベンチマークに含まれるステンシル計算も一部分であれば入出力のパターンが少なく, 再利用性が高い可能性がある。そのため, ステンシル計算の中で入出力のパターンが少ない箇所を特定し, 多くのベンチマークで効果のある再利用区間を特定したい。

6. おわりに

本研究では, 自動メモ化プロセッサにおける新たな再利用区間の特定に向けて, 再利用性の高い命令列の特徴について調査を行った。その結果, 再利用性の高いシーケンスとして, ステンシル計算を含むシーケンスが多く存在することが明らかとなった。また, ステンシル計算においては, コンパイラ最適化により, load と計算が分離される傾向があり, 命令列として明確なフェーズ構造を持つことが確認された。この特徴に基づき, 再利用区間の開始を 3 つ連続した load 命令, 終了を 3 つ連続した store 命令・load 命令と定義することが有効であると考えられる。今後は, 本研究で得られた知見に基づき, より高い再利用性を持つシーケンスの抽出手法について検討を進める予定である。

謝辞 本研究の一部は JSPS 科研費 21H03408, 23K21652, 25K03093 の助成を受けたものである。

参考文献

- [1] Matzke, D.: Will physical scalability sabotage performance gains?, *Computer*, Vol. 30, No. 9, pp. 37–39 (1997).
- [2] Gargini, P.: The international technology roadmap for semiconductors (itrs): “past, present and future”, *IEEE Gallium Arsenide Integrated Circuits Symposium. 22nd Annual Technical Digest*, IEEE, pp. 3–5 (2000).
- [3] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 5th edition (2012).
- [4] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).
- [5] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).
- [6] Huang, J. and Lilja, D. J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th Int’l Symp. on High-Performance Computer Architecture (HPCA-5)*, pp. 106–114 (1999).
- [7] Fabrice Bellard: QEMU, a fast and portable dynamic translator, *In USENIX Annual Technical Conference, FREENIX Track*, Vol. volume 41, California, USA, pp. 1–46 (2005).