卒業研究論文

複数イタレーションの一括再利用による並列事前実行の高速化

指導教員　　松尾 啓志 教授
　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 18 年度入学　18115011 番

池谷 友基

平成 22 年 2 月 8 日

複数イタレーションの一括再利用による並列事前実行の高速化

池谷 友基

内容梗概

近年，CPU の消費電力の増大や発熱量の限界から CPU の動作周波数の向上が難しくなってきている．現在までに，プログラムの実行を高速化する手法として命令レベル並列性 (ILP) に着目するものや，1 つの CPU に複数コアを搭載するマルチコア CPU を用いたスレッドレベル並列性 (TLP) に基づく高速化手法が提案されている．しかし，これら既存手法を用いた高速化は頭打ちになってきている．そこで，既存バイナリを変更することなく，計算再利用をハードウェアで動的に行う自動メモ化プロセッサが提案されてきた．計算再利用とは，ある命令区間の入力に対する出力を表に登録しておき，再度同じ命令区間に対して同じ入力があった場合は，表から出力を読み出して，命令区間の実行を省略することにより，高速化を図る手法である．さらに，この自動メモ化プロセッサに並列事前実行と呼ばれる投機的手法を用いることで，複数コアを有効活用する手法も提案されている．

自動メモ化プロセッサでは，計算再利用の対象を関数とループとしている．ループを対象とした再利用では，1 回のループイタレーション処理に対して，入出力情報が表へと登録される．そのため，有限の大きさである表に対して，登録される入出力情報が多くなってしまう．また，各イタレーションに対して再利用が適用可能か再利用テストを行う．再利用テストとは，表のデータと当該命令区間の入力の比較を行うことである．ループの再利用では，再利用テストを行う回数が増え再利用テストにかかるコストが大きくなってしまう．

本研究の目標は，この再利用テストのコストを削減することにより，自動メモ化プロセッサのさらなる高速化を図ることである．そこで，ループの複数イタレーションを一括で再利用する手法を提案する．これにより，再利用テストの回数を減らし再利用コストを削減する．

提案手法の有効性を検証するため，従来の自動メモ化プロセッサに提案手法のモデルを実装し，SPEC CPU95 ベンチマークでシミュレーションによる評価を行った．その結果，通常通り命令を実行するのと比較し，従来手法では最大 40.9 %，平均で 13.3 %のサイクル数の削減だったのに対し，提案手法では最大 51.0 %，平均で 20.8 %のサイクル数の削減し有効性を確認できた．

複数イタレーションの一括再利用による並列事前実行の高速化

目次

1 はじめに **1**

2 研究背景 **2**

2.1 自動メモ化プロセッサ 2

2.1.1 再利用機構の概要 2

2.1.2 再利用機構の構成 2

2.1.3 再利用機構の動作 5

2.2 並列事前実行 8

2.3 再利用オーバヘッドとオーバヘッド評価機構 10

3 提案 **12**

3.1 複数イタレーションの一括再利用 12

3.2 動作モデル 13

3.2.1 従来手法 13

3.2.2 提案手法 14

3.3 提案手法の適用による効果 16

3.3.1 再利用オーバヘッド削減 16

3.3.2 MemoTbl 使用の効率化 18

3.3.3 ストライド予測コストの増加 18

3.3.4 並列事前実行処理の遅延 19

3.3.5 オーバヘッドフィルタによる再利用対象の変更 20

4 実装 **20**

4.1 実装の概略 20

4.2 実装モデル 21

4.2.1 イタレーション一括処理 21

4.2.2 展開数のループ別動的設定 22

5 評価 **24**

5.1 評価環境 24

5.2 評価結果 25

5.3 考察 27

6 おわりに 28
謝辞 29
参考文献 29

1 はじめに

現在までに，プログラムの実行を高速化する手法として，スーパースケーラのように命令レベル並列性 (Instruction-Level Parallelism: ILP) に着目したものが研究されてきた．しかしながら，プログラム自体に存在する ILP には限界があり，命令レベルの並列化を行うだけでは，プロセッサの性能向上が頭打ちになりつつある．この流れを受け，CPU がマルチコア化されるとともに，命令レベル並列性より粗い並列性であるスレッドレベル並列性 (Thread-Level Parallelism: TLP) が注目されることとなる．並列性の抽出にはマルチスレッドライブラリや，高度なコンパイラが用いられる．ライブラリを用いた場合，プログラム内の並列化できる部分を明示的にスレッドの形で書き下す必要があり，プログラマに負担がかかる．一方，コンパイラによる抽出はその精度に問題がある．よって TLP によるプログラム実行の高速化もまた難しい状況であるといえる．

これらはいずれもプログラム実行の並列化という方法で高速化を図るものであるが，本研究はそれとはまったく異なる計算再利用という手法に着目する．

計算再利用には，ハードウェアによるものとソフトウェアによるもの，またその両方によるものなど，様々なものが提案されている．専用のハードウェアを用いることによりバイナリの変更なしで，既存のプログラムを実行できる区間再利用のモデルに自動メモ化プロセッサ [1] というものが存在する．本論文では，この自動メモ化プロセッサのさらなる高速化手法を提案する．自動メモ化プロセッサは実行した関数やループの入出力を表に記憶しておく．再び同じ関数やループが実行されたときにその入力と過去に実行した関数やループの入力とを比較し，一致すれば過去の出力を利用し実行を省略する．本研究では，ループの再利用に対して複数のイタレーションを一括で再利用することで，再利用適用時に発生するオーバヘッドを削減し，自動メモ化プロセッサのさらなる高速化を図る．

以下，2 章では本研究が扱う自動メモ化プロセッサの動作モデルを述べる．3 章では，自動メモ化プロセッサのさらなる高速化を実現する複数イタレーションの一括再利用手法を提案し，4 章で実装について説明する．5 章で本提案手法の評価を行い，最後の 6 章において結論を述べる．

2　研究背景

本研究で取り扱う自動メモ化プロセッサについて，その高速化の方針と動作原理を概説する．

2.1　自動メモ化プロセッサ

メモ化 (Memoization)[2] とは，関数やループといったプログラム中の命令区間に対して，その入出力を計算再利用可能な状態で記憶しておくことである．このメモ化を，既存のバイナリを変更することなく，ハードウェアを用いて動的に行うプロセッサとして考案されたのが自動メモ化プロセッサである．

2.1.1　再利用機構の概要

計算再利用とは，プログラムの関数呼び出しやループなどの命令区間において，その入力と出力の組を表に記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，過去の記憶された出力を利用することで命令区間の実行自体を省略し，高速化を図る手法である．既知の入力値に対して再び同じ区間を実行する際，正しい出力値を表検索で求めることができる．計算再利用の特長は，入力値さえ一致すれば実行結果を検証する必要がないことである．

2.1.2　再利用機構の構成

自動メモ化プロセッサのアーキテクチャを**図 1** に示す．自動メモ化プロセッサでは，コアの内部に一般的な CPU コアが持つ ALU，レジスタ (Reg)，1 次データキャッシュ (D$1) を持ち，コアの外部に共有の 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が独自に持つ機構として，コアの内部に MemoBuf と再利用機構を管理するための構造 (Reuse_System) を持ち，コアの外部に MemoTbl を持つ．MemoTbl とは命令区間およびその入出力を記憶するための表であり，計算再利用を行うために必要となる．また，コアが命令区間の入出力を MemoTbl に登録する際，サイズの大きい MemoTbl に対して毎回参照を行うとオーバヘッドが大きくなってしまう．そこで，このオーバヘッドを軽減するため，作業用の小さなバッファである MemoBuf をコアの内部に設けている．MemoBuf と MemoTbl の詳細な構成を**図 2** に示す．

まず，MemoBuf について説明する．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する．各エントリは，該当する命令区間を記憶する RFindex，命令区間の実行開始時のスタックポインタ SP，関数の戻り値とループの終端アドレス

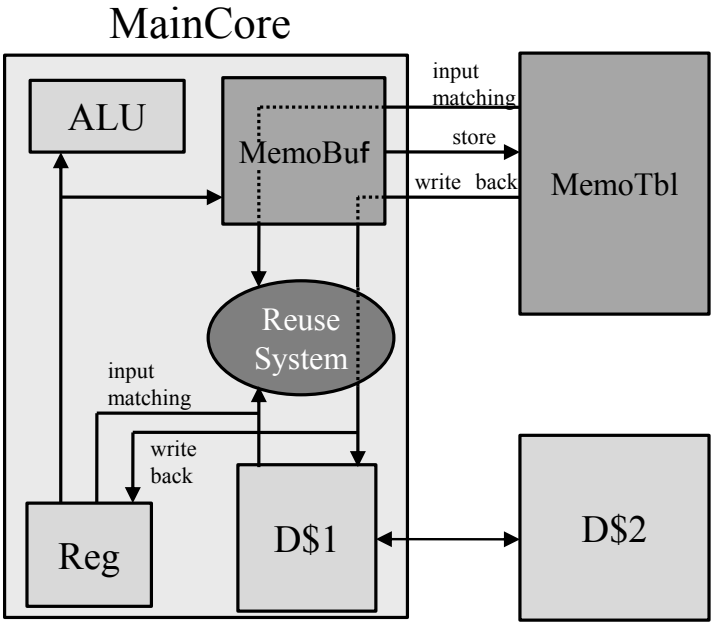

図 1: 自動メモ化プロセッサのアーキテクチャ

を記憶する FOfs，命令区間の入力セット Read，および出力セット Write の各フィールドからなる．命令区間の実行中に出現した入出力は，命令区間の実行を進めながら Read と Write フィールドに記憶されていく．入出力とは，参照したレジスタおよび主記憶アドレスになる．そして，各命令区間の実行終了時に再利用エントリを一括して MemoTbl 本体に登録する．

MemoBuf に複数のエントリが存在するのは，再利用コアが実行中に呼び出した関数やループのネスト構造を保持するためである．MemoBuf のどのエントリを使用しているかを判断するためにポインタ memobuf_top を用いる．MemoBuf はスタックのように扱われ，階層の低い方から順に番号が振られている．エントリは番号の値の小さい方から順に使用され，関数呼び出し (call) や多重ループによってネストが増加すると，それに対応して memobuf_top がインクリメントされる．逆に関数から戻った場合 (return) や内側のループの実行が終了したときに，memobuf_top をデクリメントし階層を下げる．このようにして，MemoBuf は，コアが現在実行している関数やループのネスト構造を保持する．

次に，MemoTbl の各構成要素について説明する．まず，RF は命令区間を記憶する表であり，関数とループを判別するフラグ (type) を持ち，命令区間の開始アドレス (fadr) と終端アドレス (eadr) を記憶する．ただし，関数の場合は終端アドレスとして戻りアドレスを記憶している．また，該当する命令区間を記憶する RFindex を持つ．RFindex

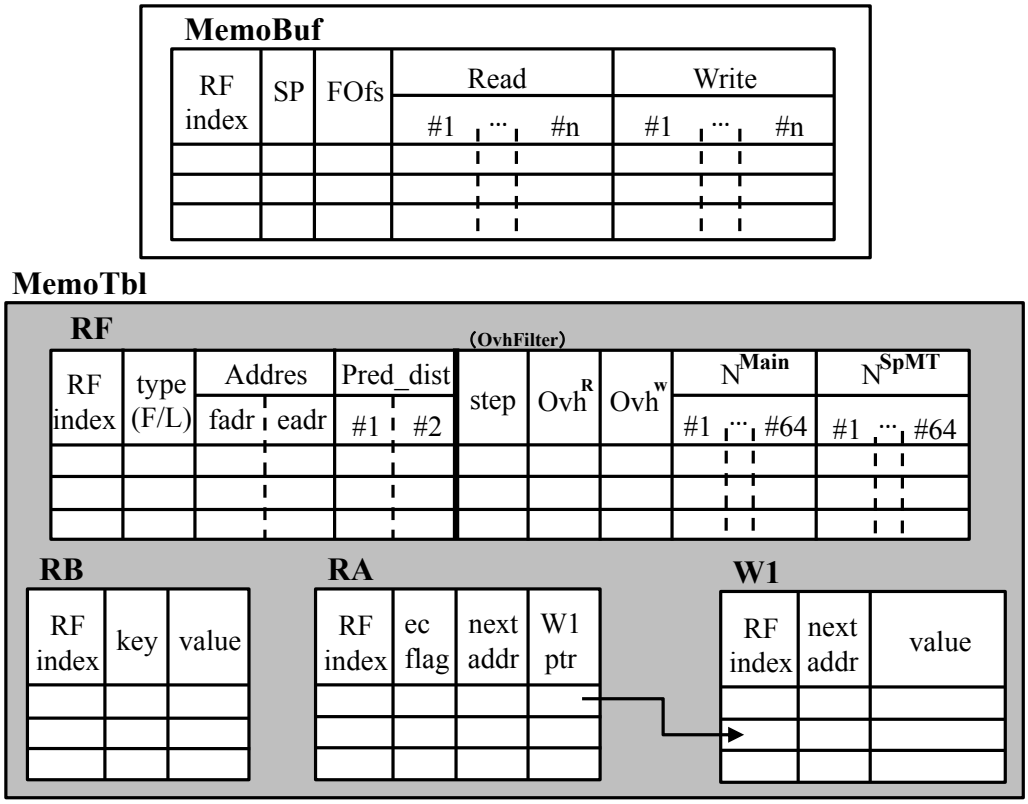

図 2: MemoBuf と MemoTbl の詳細な構造

を RF が一括管理し，他の各要素が RFindex を持つことで，命令区間ごとのエントリの管理ができる．RB は命令区間の入力値を記憶するための表であり，入力値から次入力となるエントリを検索し特定する必要があるため，連想検索が可能な CAM(Content Addresable Memory) で実装されると仮定している．また，入力一致比較を行うときの MemoTbl の検索オーバヘッドを小さく抑えることができる．RA は入力アドレスのセットを記憶する表であり，次に参照すべき入力アドレス (nextaddr) を記憶する．RB と RA は同数のエントリを持ち，各エントリは 1 対 1 に対応する．さらに，RA はそのエントリが入力値セットの終端であることを示すフラグ (ecflag) を持つ．W1 は出力値を記憶する表であり，RA の終端エントリがその出力を記憶している W1 のエントリを指すポインタ (W1ptr) を持つ．

MemoTbl と入力一致比較を行い，入力が完全に一致した場合は W1 のエントリを指すポインタ (W1ptr) により当該入力に対応する出力を読み出し，レジスタやメモリへ書き戻すことで，命令区間の実行を省略することができる．また，MemoTbl の大きさは有限であるため，MemoTbl のエントリが溢れる前に不要なエントリを削除する必要

```
int x,y,z;
int Select(int a){
  if(a+x>5) return(y);
  else      return(z);
}
int main(void){
  x=3,y=4,z=5;  Select(3);
  x=3,y=5;      Select(3);
  x=1,y=4;      Select(3);
  return(0);
}
```

図 3: サンプルプログラム

がある．このエントリの削除アルゴリズムにはLRU(Least Recently Used) 方式を用いている．

2.1.3 再利用機構の動作

計算再利用を行うための再利用機構の動作に，命令区間や入出力値を記憶したエントリの MemoTbl への登録と，当該命令区間が再利用可能か入力一致比較を行う再利用テストがある．

エントリの登録

関数やループの実行が終了したとき，実行開始から MemoBuf に蓄えてきた開始アドレスや入出力値等の情報が MemoTbl へ一括で登録される．このとき，命令区間のアドレスは RF へ，入力値は RB へ，入力のアドレスは RA へ，出力値は W1 へ登録される．

一般に，関数やループの命令区間内では，ある入力の値によって次に参照するべき入力アドレスが変化する．例えば，変数に主記憶アドレスが格納されている場合や条件分岐により次の入力が変化する場合である．そのため，ある命令区間の全入力パターンを保持するために，入力が登録されている入力エントリは木構造を成している．よって，関数やループの 1 入力セットはその木構造における 1 本のパスとして表現できる．**図 3** にサンプルプログラムを示す．ここでは関数を例にあげ，入力エントリの登録について説明する．

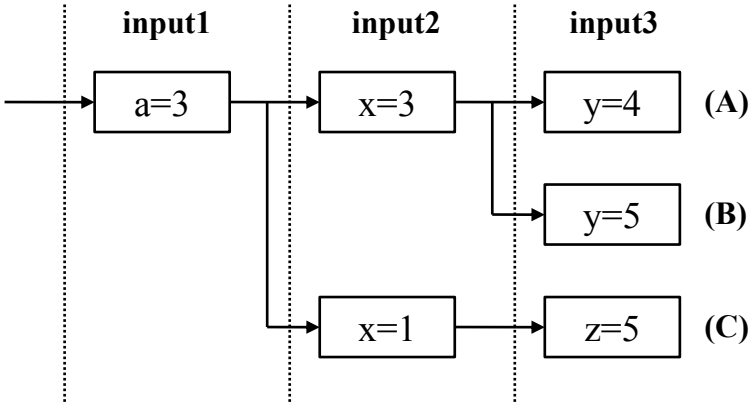

図 4: 入力エントリがなす木構造

関数の入力には引数と，その関数が参照する大域変数や上位関数の局所変数が含まれる．この場合，関数 Select() の入力は，引数 a，大域変数 x,y,z となる．この関数 Select() の入力が MemoTbl 中のエントリにおいて木構造をなす様子を**図 4** に示す．図 4 中，input1,input2,input3 は関数 Select() の入力を表している．

入力は多分木で構成され，RB エントリが節に，RA エントリが枝に対応する．まず，引数 a が 3 で関数 Select() が呼び出される．この時，入力として使われる大域変数 x,y,z はそれぞれ 3,4,5 となっている．関数内の条件分岐が成立するため，返り値は y の値となり入力エントリは図 4 の (A) のように登録される．また，変数 y の値のみが変わり 2 回目の関数 Select() が呼び出されると，図 4 の (B) のように登録される．ここで，これら 2 つの関数呼び出しにおける 2 つ目までの入力 $a = 3$ と $x = 3$ は共通である．同じ命令区間で入力値の共通する部分が存在する場合は，入力エントリを有効活用するため，途中までの入力エントリを共有して記憶する．3 回目の関数 Select() の呼び出しでは，変数 x の値が変化しているため条件分岐が不成立となり，図 4 の (C) のように登録される．変数 x の次に参照される値は変数 y ではなく変数 z となり，次に参照するべき入力アドレスが変化する．そのため，次の入力値の枝分かれの部分では，異なる RA エントリを用いている．これにより，入力アドレスが変化した場合にもレジスタやメモリの参照を可能としている．

再利用テスト

計算再利用を行うためには，関数やループの入力値と，既に MemoTbl に登録されているエントリとを比較する必要がある．**図 5** に MemoTbl の検索手順の様子を示す．ここでは，図 3 のプログラム例を実行後の MemoTbl を用いている．図 5 の RB 中の

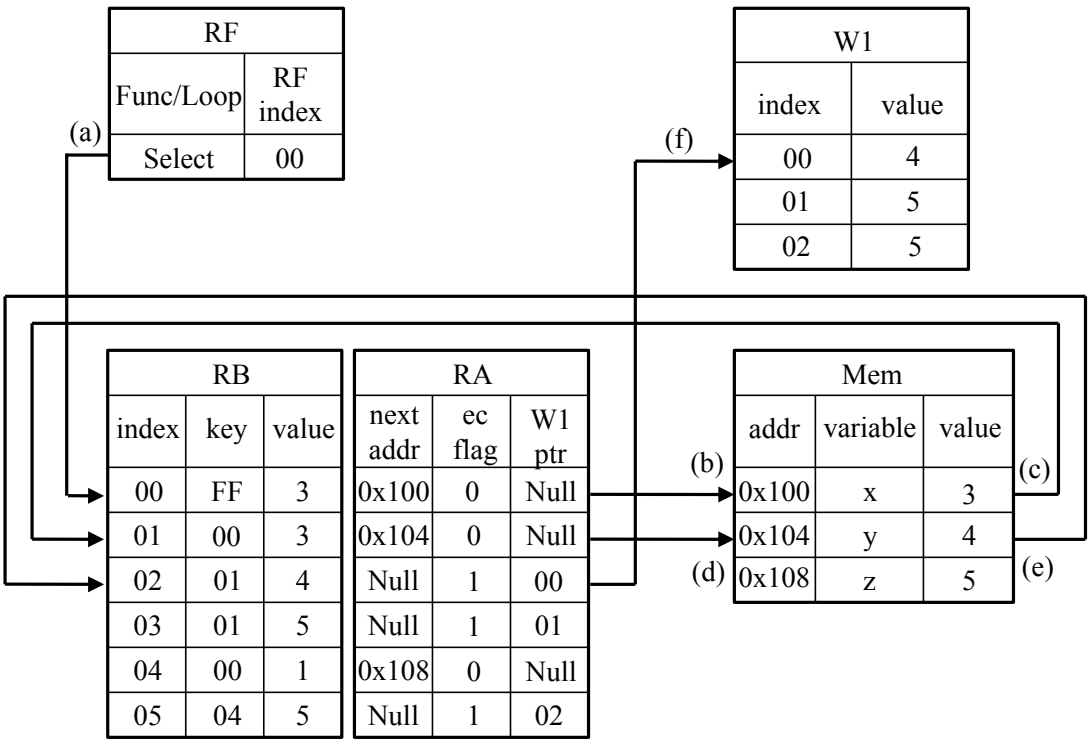

図 5: MemoTbl の検索手順

key は親エントリのインデクスを，value は入力値を指す．index は表のインデクスであり，RFindex は全て同じ値であるため省略する．図 3 のプログラム例の場合では，入力 input2,input3 の親エントリがそれぞれ input1,input2 となり，input1 は親エントリを持たない．親エントリを持たないエントリは key 値として，一番目のエントリであることを示す FF を持つ．また，RA 中の next_addr は次に参照すべき入力アドレスを格納しており，ec_flag はエントリが入力値の最後のエントリであるか否かを示す．W1ptr は，出力値を記憶している W1 のエントリを指すポインタを格納する．そして，W1 中の value は出力値を指す．

関数やループの命令区間の検出時，その開始アドレスを用いて RF を検索する．ここでは，$x = 3$,$y = 4$,$z = 5$ で関数 Select(3) の関数呼び出しが行われたとして，どのように MemoTbl を検索するのかを説明する．まず，RF により得られた RFindex を基に RB エントリの検索を開始する．このとき，最初のエントリであることを示す FF を key 値として持ち，1 つめの入力値 3 を value として持つエントリを検索する (a)．条件に一致した RB エントリに対応する RA エントリから，次に参照するアドレスを得てレジスタや主記憶を参照し，次入力値を得る (b)．次に，一致した RB のインデクス

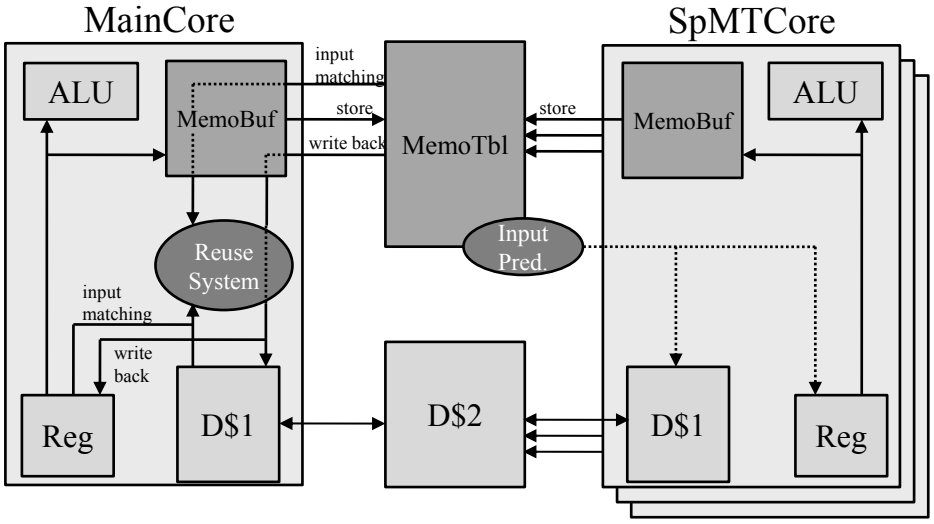

図 6: 並列事前実行機構を備えた自動メモ化プロセッサのアーキテクチャ

00 を key として持ち，取得した 2 つ目の入力値 3 を value として持つエントリを，再度 RB の中から検索する (c)．以上の手順を繰り返し (d,e)，RA エントリの ec_flag の値が 1 であるとき，入力が全て一致し再利用テストが成功する．再利用テストに成功した場合は，検索の終点となった RA エントリの W1ptr に格納されているポインタによって W1 を参照し，当該命令区間の出力値を読み出す (f)．そして，読み出した出力をレジスタやメモリに書き戻すことで，当該区間に対して計算再利用が適用できる．

2.2 並列事前実行

並列事前実行とは，過去の命令区間の入力に基づき，ストライド予測 [3] を用いてその命令区間を将来実行する際に用いられる入力を予測し，該当区間をメインコアとは別のコアで予め実行しておくことである．このコアを並列事前実行コアと呼び，並列事前実行コアは予測された入力を用いて得られた実行結果の入出力を，計算再利用可能な状態で MemoTbl へと登録する．これにより，ループのように入力のパラメータが単調に変化する場合など，過去の実行結果を利用しても計算再利用が全く効果がない命令区間に対しても高速化を図ることができる．

図 6 に並列事前実行機構を備えた自動メモ化プロセッサのアーキテクチャを示す．図 6 に示すように，ALU，レジスタ (Reg)，1 次データキャッシュ(D$1)，MemoBuf は各コアが独立で持ち，MemoTbl と 2 次データキャッシュ(D$2) および主記憶は全てのコアで共有されるものとする．

MemoBuf をコアごとに持っているため，各命令区間の実行終了時にそれぞれのコアは入出力情報を MemoBuf から MemoTbl へ独立して登録することができる．メインコアは自身や並列事前実行コアによって登録された MemoTbl のエントリを用いて再利用を行うことができる．並列事前実行コアは複数備えることができ，それぞれのコアが予測された入力に対する区間を並列に実行することができる．また，並列事前実行では入力の予測が外れた場合でも，再利用を行うメインコアからは予測が外れたことが観測されないため，並列事前実行によるオーバヘッドは生じない．ただし，大きさが有限である MemoTbl に不必要なエントリが登録されることにより，有効なエントリが削除され性能が低下してしまう可能性はある．

次に，並列事前実行コアを用いた場合の動作を説明する．並列事前実行を行うためには過去の再利用エントリの登録情報から，将来の入力を予測して並列事前実行コアへ渡す必要がある．このため，入力を予測して並列事前実行コアに渡すための小さなハードウェア (Input Pred) を MemoTbl へ設ける．具体的な動作は，まず最後に出現した入力および最近出現した 2 組の入力の差分に基づいてストライド予測を行う．予測された入力セットに基づき，並列事前実行コアはメインコアと並列して当該命令区間の実行を開始する．そのため，図 2 で示したように，RF では各命令区間に対してストライド予測に用いる最近出現した 2 組の入力 (Pred_dist) を保持している．

並列事前実行を用いた場合の動作を**図 7** に示す．例として，並列事前実行コアを 3 台とし，ある命令区間に対してメインコアが入力値 4 で通常実行しているとする．それに平行して並列事前実行コアではストライド予測を用いて入力値 5,6,7 でそれぞれ実行を行う．

図 7 の (a) は最も効率良く再利用が適用できる場合である．(a) では，メインコアが入力値 4 に対する処理を終え入力値 5,6,7 の実行に移ろうとしたとき，3 台の並列事前実行コアでそれぞれの入力に対する処理を終えて MemoTbl に入出力情報が登録されている．一方，(b) では，3 台目の並列事前実行コア (SpMT3) で入力値 7 に対する処理が遅延し，メインコアが並列事前実行コアと同じ入力値による実行を開始してしまった場合である．これは，キャッシュミスの発生などにより処理が遅延してしまったためである．

この問題を回避するため，自動メモ化プロセッサでは，並列事前実行コアにおける入力値 5,6,7 に対する処理を，メインコアが入力値 4 に対する処理よりも早めに開始している．つまり，現在メインコアで実行中の入力よりもある程度先まで，並列事前実行コアへの入力割り当てを行っている．

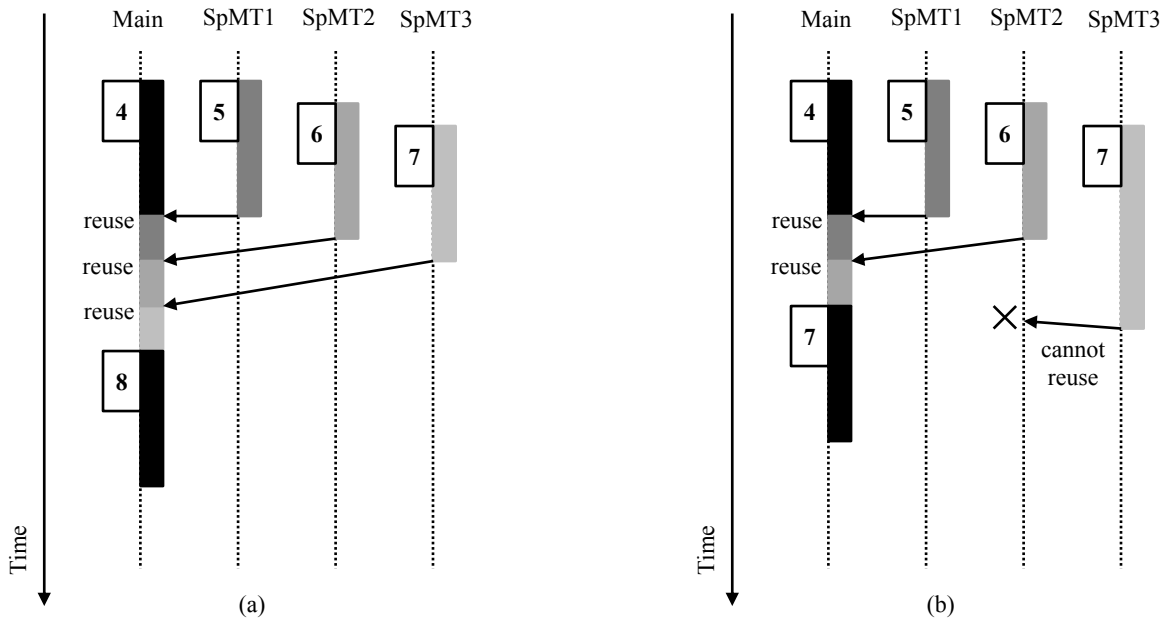

図 7: 並列事前実行の流れ

また，メインコアと並列事前実行コアでは 2 次データキャッシュや主記憶を全てのコアで共有している．このため，これらの共有領域に対して書き込みを行うと，他の並列事前実行コアやメインコアがプログラムの実行を行う際にデータの不整合が生じてしまう．よって，並列事前実行コアでは MemoBuf を主記憶書き込みの際のバッファとして扱い．コア間での共有しているデータに対する書き込みを行わないようして，メモリデータの不整合の発生を回避している．

2.3 再利用オーバヘッドとオーバヘッド評価機構

自動メモ化プロセッサがある命令区間に対して計算再利用を適用するためには回避不可能なオーバヘッドが生じる．まず，再利用テストの入力値検索のときに，メインコアのレジスタや主記憶の値と MemoTbl に登録されている値とを比較するための検索コストがある．この検索コストは入力値検索の成功・失敗に関わらず発生する．また，入力値検索が成功した際に，出力を MemoTbl からレジスタやキャッシュへ書き戻すための書き戻しコストがある．入力値の検索コストと出力の書き戻しコストをあわせて再利用オーバヘッドと呼ぶ．

命令区間によっては，再利用オーバヘッドが大きく，計算再利用を行わずに実際に命令を実行した方が早く実行を終えることができる場合も存在する．その場合，計算再利用により性能が悪化するばかりか，必要としない入出力を MemoTbl に登録して

いることになり，MemoTbl が有効活用されない．そこで，自動メモ化プロセッサでは，MemoTbl への無駄なアクセスを抑制する再利用オーバヘッド評価機構を備えている．再利用オーバヘッド評価機構を使用して，再利用オーバヘッドと計算再利用により高速化できる実行サイクル数を見積もり，計算再利用により効果を得られると判断した命令区間に対してのみ再利用テストを行う．具体的には，命令区間の再利用により削減できるサイクル数と，その再利用に必要となるオーバヘッドについて概算を行う小さなハードウェアを MemoTbl に付加する．この機構をオーバヘッドフィルタと呼ぶ．

前述の並列事前実行では，並列事前実行コアの対象とする命令区間をいかに選択するかが重要である．そのため，図 2 で示したように，オーバヘッドフィルタの機構は，命令区間を記憶する表である RF の拡張として備えられている．メインコアによる登録頻度が高いと呼び出し回数が多い命令区間となる．また，並列事前実行コアが登録したエントリの使用頻度が高くなると，並列事前実行コアが有効に働いている．そのため，命令区間のうちでも，メインコアによる登録頻度が高く，並列事前実行コアが登録したエントリの使用頻度が高いものが，最も並列事前実行による効果が得られる．再利用機構では，この動的に変化する登録頻度や使用頻度を把握するために，一定期間における登録および再利用の状況をシフトレジスタを用いて記録している．このシフトレジスタの記録を元に再利用オーバヘッドの算出を行う．既存のオーバヘッドフィルタでは 64 ビットのシフトレジスタを用いて，過去 64 回分の状況を記録している．

ある命令区間における，最近の一定期間 T でのメインコアによる登録回数 N^{Main} と並列事前実行コアが登録したエントリの使用回数 N^{SpMT} は，上記のシフトレジスタから得られる．これらの値と当該命令区間の過去の省略サイクル数 S から実際に削減できるサイクル数を

$$(N^{Main} + N^{SpMT}) \times (S - Ovh^R - Ovh^W) \tag{1}$$

として計算する．なお，Ovh^R，Ovh^W はそれぞれ，過去の履歴より概算した MemoTbl の検索コストと書き戻しコストである．

また，再利用が行われなかった場合でも，MemoTbl の検索コストは存在する．このオーバヘッドは，

$$(T - N^{Main} - N^{SpMT}) \times Ovh^R \tag{2}$$

として計算できる．

ここで発生したオーバヘッド (2) よりも，削減できたサイクル数 (1) が大きいような命令区間は，再利用の効果が得られると考えられる．なお，過去に再利用が行われたこ

とがあり，再利用効果が得られる命令区間であっても，最近の T の間に再利用が一度も行われなかった場合，$N^{Main} + N^{SpMT} = 0$ となってしまい，再び再利用対象とならなくなってしまう．この場合には，N^{SpMT} を T で初期化することでこれを回避する．

3 提案

自動メモ化プロセッサでは，関数とループを計算再利用の対象区間としている．ループの場合，イタレーションごとに再利用テストが行われるため，再利用テストの回数が増加し再利用オーバヘッドが大きくなる．また再利用エントリが増え MemoTbl が圧迫される原因となる．そこで，本研究ではループの再利用に対して複数のイタレーションをまとめて，一括で再利用を行う手法を提案する．本章では，提案手法を適用した場合の再利用機構の動作と期待される効果について述べる．

3.1 複数イタレーションの一括再利用

従来の再利用機構では，ループ再利用時にイタレーション毎に登録や再利用テストなどの再利用に関する処理が実行される．各イタレーションでの入出力情報が MemoTbl へと登録され，全てのイタレーション処理の開始時に再利用テストが毎回行われる．そのため，基本的に処理が繰り返して行われるループでは，再利用テストの回数が増加してしまい，1 つのループでみるとその再利用オーバヘッドが大きなものとなってしまう．また，全てのイタレーションの入出力情報が再利用エントリとして登録されるため，MemoTbl が圧迫される原因となる．そこで，本研究では，複数のイタレーションをまとめる手法を提案する．いわゆるループアンロールのように複数回分のイタレーションの処理を 1 回とみなす．ただし，実際にプログラムをループアンロールして書き換えるわけではなく，再利用機構が動的にループの複数イタレーションを 1 つの再利用対象区間として扱う．これにより，従来と同様にバイナリ変更なしでの動作が可能となる．

具体的には，イタレーション複数回分の入出力情報を 1 つの再利用エントリとして MemoTbl へと登録を行う．ループ実行中に，あるイタレーションの入出力情報の登録が始まった場合，従来では 1 回分のイタレーションの処理が終了した時点で入出力情報を MemoTbl へと書き戻すが，提案手法では処理区間が続いているとみなし，入出力情報の登録処理を継続させる．ある一定数のイタレーション処理の終了に到達した時点で，複数回分のイタレーションの入出力情報を 1 つの再利用エントリとして MemoTbl へ登録することにする．以降，複数のイタレーションをまとめる時のこの一定値を**展**

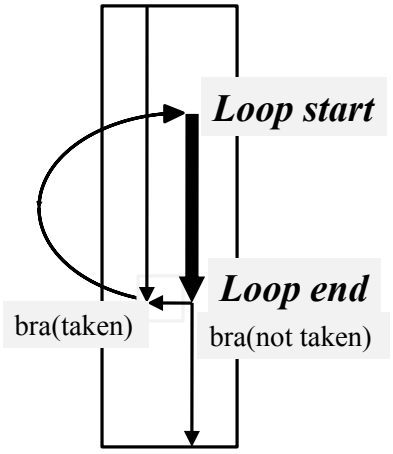

図 8: プログラム中のループ

開数と呼ぶ.

MemoTbl にはイタレーション複数回分の入出力情報が登録されているため，再利用テストに成功したときにそのエントリを用いることで，一括での再利用を行う．ループの数が多いプログラムでは，本提案手法の適用機会が増える．また，回数の大きいループが存在しているプログラムでは，再利用テストの回数をより抑えることができる．そのため，これらの傾向のプログラムで提案手法の適用により高速化が見込める．

3.2 動作モデル

ここでは，ループ時における従来の再利用機構の動作を詳しく説明した後，複数イタレーションを一括で再利用するために，提案手法を適用した再利用機構の動作を従来手法と比較しながら説明する．

3.2.1 従来手法

ループは，**図 8** に示すように，一度後方分岐に到達し，後方分岐が成立した後に，再度同じ後方分岐命令に到達してはじめて，それまでの命令列がループを形成していたことがわかる．本質的に，ループの第 1 回目のイタレーションを再利用することはできないものの，一般的にループは繰り返し実行されることを考えると，この制限は軽微なものである．後方分岐命令の分岐先 (Loop Start) に始まり，同一の後方分岐命令 (Loop End) に終わる命令区間について，この間の入出力を登録しておくことにより，ループを再利用することができる．

ループの入出力登録が中止されなければ，登録中のループに対応する後方分岐命令を検出した時点で，登録中の入出力表エントリを MemoTbl に登録し，現イタレーショ

ンの登録を完了する．さらに，後方分岐命令が成立する場合は，次イタレーションが再利用可能かどうか判断する．すなわち，後方分岐する前に，後方分岐先アドレスを検索し，入力値が完全に一致するエントリを選択し，関連する主記憶アドレスすべてを参照して，一致比較を行う．全ての入力が一致した場合，登録済みの出力を書き戻すことにより，次イタレーションの実行を省略することができる．再利用した場合，MemoTbl に登録されている分岐方向に基づいて，さらに次イタレーションに関して同様の処理を繰り返す．一方，次イタレーションが再利用不可能であれば，次イタレーションの実行および MemoTbl への登録を開始する．

従来手法でのループ再利用の動作の様子を**図 9** に示す．第 1 回目の後方分岐に到達し，後方分岐が成立した時に，それまでの命令列がループの処理区間であると判断する．以降のイタレーションに対して再利用テストが行われ，テストに失敗した場合にはそのイタレーションの入出力情報を登録していく．第 2 回目と第 3 回目のイタレーションの処理が終わると，その 2 つの入出力情報の差分からストライド予測を行い，次イタレーションの入力を予測して並列事前実行コアが有効に動作するようになる．

第 1 回目の後方分岐に到達し後方分岐が成立した時に，それまでの命令列がループの処理区間であると判断する．そのため，図 9 で示すように第 1 回目のイタレーションは通常実行される．ここでは，説明のために第 N 回目のイタレーションをイタレーション N とする．以降のイタレーションに対して再利用テストが行われ，再利用テストに失敗した場合にはそのイタレーションの入出力情報を登録していく．イタレーション 2 とイタレーション 3 の処理が終わると，その 2 つの入出力情報の差分からストライド予測を行い，次イタレーションの入力を予測して並列事前実行コアが有効に動作するようになる．

メインコアがイタレーション 4 の処理を実行しているのと平行して，予測された入力セットに基づき，並列事前実行コアも当該命令区間の実行を開始する．各並列事前実行コアでは，それぞれイタレーション 5,6,7 が実行される．これにより登録されたエントリを用いることでメインコアではイタレーション 5,6,7 の処理を再利用によって高速化できる．以降，並列事前実行で作られた再利用エントリが全て再利用に適用できた場合，図 9 のような動作となる．

3.2.2 提案手法

複数のイタレーションを一括で再利用するために，複数回分のイタレーションを 1 回としてみなす．提案手法でのループ再利用の動作の様子を**図 10** に示す．提案手法では第 1 回目のイタレーションによってループを判別した後，図 10 のように複数のイタ

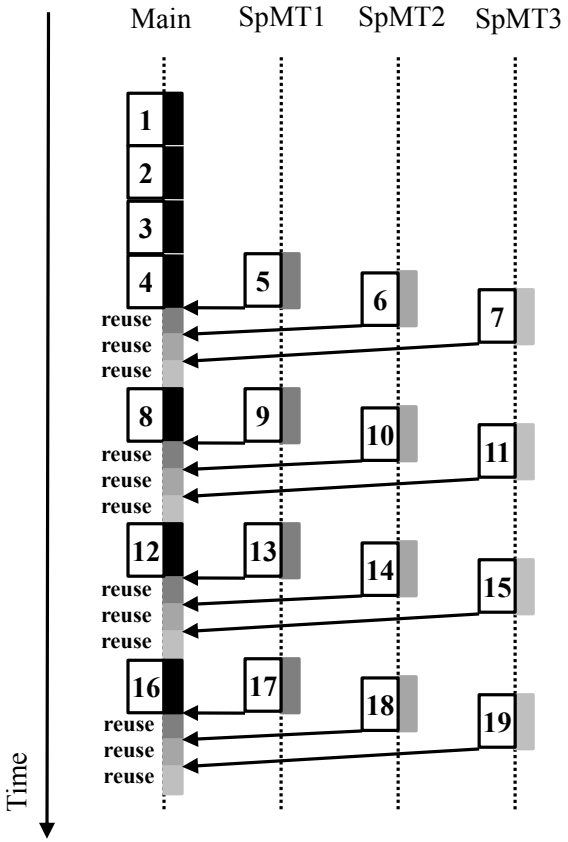

図 9: 従来手法におけるループ再利用の動作の様子

レーションをまとめて 1 回とみなし処理を行う．ここでは，展開数を 2 とする．第 2 回目と第 3 回目の処理であるイタレーション 2,3 と 4,5 の入出力情報が登録されると，2 つの差分を用いてストライド予測が可能となる．並列事前実行コアでの処理も，イタレーション 8,9 やイタレーション 10,11 のように，複数のイタレーションをまとめて実行する．

なお，複数のイタレーションをまとめた場合でも，再利用テストは従来通りに行うことができる．一致比較する入力エントリには命令区間の情報が含まれておらず，入力を全て比較することで，登録された時点での展開数に相当する入力について確認したことが保証される．そのため，展開数を意識しなくても入力一致比較が実行できる．さらに，イタレータ変数などの出力は展開数分の処理終了後の状態で登録されているため，従来通りに書き戻すことによって適切な展開数分だけ処理を省略できる．

また，ループのように入力パラメータが単調に変化する場合には，過去に登録された再利用エントリが使われることが少なく，主に並列事前実行による効果の方が大き

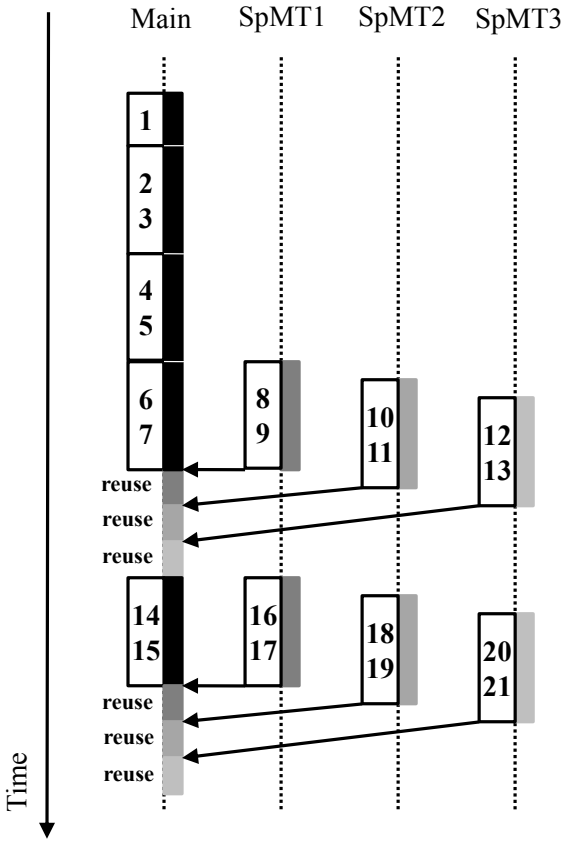

図 10: 提案手法におけるループ再利用の動作の様子

い．そのため，複数イタレーションをまとめたとしても，再利用テストの成功率が急激に悪くなることはないと予想される．展開数が 2 であるこの例では，従来手法での 2 回分の再利用オーバヘッドより，提案手法での 1 回の再利用オーバヘッドが減っていれば高速化ができる．

3.3 提案手法の適用による効果

提案手法を再利用機構に適用した場合，以下にあげる効果が生じると考えられる．

3.3.1 再利用オーバヘッド削減

再利用機構では 2 章で示したように，再利用テスト時の入力値検索にかかるコストと再利用成功時の出力書き戻しにかかるコストの両方の再利用オーバヘッドがある．複数のイタレーションをまとめて一括で再利用することで，再利用オーバヘッドの削減が見込める．検索コストは，MemoTbl に登録されている入力エントリと比較を行うため，入力の数に応じて大きくなる．また，書き戻しコストは，MemoTbl から出力をレ

```
int main(void){
  int i,sum=0;
  int array[10]={1,2,...,10};

  for(i=0;i<10;i++){
    sum=sum+array[i];
  }

  return(0);
}
```

図 11: ループプログラム例

```
int main(void){
  int i,sum=0;
  int array[10]={1,2,...,10};

  for(i=0;i<10;i+=2){
    sum=sum+array[i];
    sum=sum+array[i+1]
  }

  return(0);
}
```

図 12: ループアンロール後

	入力	出力	ループ回数	一致比較	書き戻し
従来手法	i, array[i], sum	i, sum	10	30	20
提案手法	i, array[i], array[i+1], sum	i, sum	5	20	10

表 1: 再利用オーバヘッド算出

ジスタやメモリに書き戻すため，出力の数に応じて大きくなる．

図 11 に C 言語によるプログラムの一例を示す．また，そのプログラムをループアンロールしたプログラムを**図 12** に示す．ここでは，説明のために提案手法の適用を図 12 で示したプログラムのようにループアンロールした場合で考える．ループアンロールをイタレーション処理 2 回分で行うため，提案手法での展開数を 2 とした時と同等であるといえる．また，全てのイタレーションが再利用できたと仮定する．

従来モデルでは，入力が sum と $array[i]$，そしてイタレータである i も「$i = i + 1$」とインクリメントされているため含まれることになり 3 種類となる．出力は sum と i である．一方，提案モデルでは入力が sum と $array[i]$，$array[i + 1]$，そして i となり，出力は従来と同じで sum と i である．ループ全体で見たとき，一致比較が必要となる変数と書き戻しが必要となる変数の延べ数は，入 (出) 力数 × ループ回数となり**表 1** のようになる．

複数イタレーションを 1 つにまとめると，登録される入出力が多くなるため，1 回の

再利用にかかるコストは増加する．しかし，再利用テストの回数自体が減るため，結果として再利用オーバヘッドを削減できる．この場合では，従来モデルにおける 2 回分の一致比較と書き戻しに要する変数の数より，提案モデルにおける 1 回分の変数の数が少ないため，再利用オーバヘッドが削減できている．よって，複数イタレーションをまとめて一括で再利用することで再利用オーバヘッドが削減できることがわかる．

3.3.2 MemoTbl 使用の効率化

MemoTbl は有限であるため記憶できる入出力情報にも限界があり，プログラム終了時まで全ての再利用エントリを保持しておくことができない場合が多い．入出力の登録領域を確保するためには，適宜 MemoTbl のエントリを削除していく必要がある．そのため，自動メモ化プロセッサでは，MemoTbl から不要なエントリを削除する機能を備えている．このエントリの削除をパージと呼ぶ．

再利用エントリのパージアルゴリズムには LRU 方式を用いている．各エントリにタイムスタンプを持たせ，再利用が行われた際にタイムスタンプを更新する．命令区間の実行ごとにタイムスタンプが更新されていない古いエントリを不要なエントリと判断しパージしていく．また，エントリの削除による登録領域の確保が間に合わない場合には，現在呼び出している当該命令区間で，過去に登録されたすべての再利用エントリを削除することで，新たなエントリの登録を可能にする．

複数のイタレーションをまとめると登録するべき入出力情報が減り，MemoTbl が溢れてしまう可能性が減る．また，MemoTbl への登録回数が減ることで，タイムスタンプの更新頻度が低くなり定期的に行われるパージの回数も減少する．そのため，MemoTbl にエントリが長く残り続けることになる．エントリがパージされるまでの期間が従来より長くなることで，MemoTbl の効率が向上し再利用テストに成功する確率が高まる．

3.3.3 ストライド予測コストの増加

並列事前実行を行うためには，将来その命令区間を呼び出す時の入力を予測する必要がある．過去の命令区間の入力に基づき，最近出現した 2 組の入力の差分を用いてストライド予測を行う．ループの場合，第 1 回目のイタレーションは命令区間判定のために再利用エントリは登録されないため，第 2 回目と第 3 回目のイタレーションの入出力の登録情報を得て初めて，そのループに対する有効な並列事前実行が行われるようになる．

第 1 回目のイタレーションにより再利用機構がループの命令区間を判別する．図 10 では，それ以降，展開数が 2 として処理が行われていく．ストライド予測のために第 2

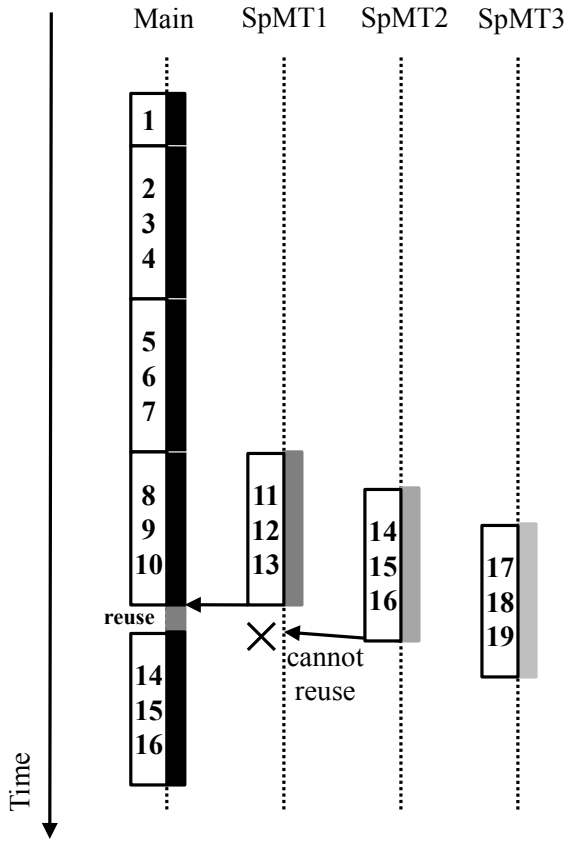

図 13: 展開数が 3 のときの再利用機構の動作の様子

回目と第 3 回目の入出力の登録情報が必要となる．展開数が 2 の場合だと，ストライド予測のために合わせて 4 回分のイタレーションを実行することになる．そこから並列事前実行が有効に働くことになるため，並列事前実行による再利用が有効になるのはイタレーション 8 の処理を開始するときになってしまう．従来ではイタレーション 5 から再利用が効くようになっていた．つまり，その差分が再利用ができなくなってしまったといえる．ループ回数が多い場合はこの差はあまり影響しないが，小さなループだと無視できないものとなる．

3.3.4 並列事前実行処理の遅延

イタレーションの展開数を大きくすると，1 回分の計算量も多くなる．展開数を 3 とした場合の再利用機構の動作の様子を**図 13** に示す．

図 13 では，第 12 回目からのイタレーションに対して再利用テストが失敗している．SpMT2 ではメインコアに並列してイタレーション 12,13,14 の入出力情報を登録を行っているが，メインコアが再利用テストを行う段階ではまだ SpMT2 による処理が終わっ

ておらず，再利用エントリの登録が完了していない．そのため，メインコアでは再利用テストが失敗したことにより通常の実行が開始される．SpMT2 での並列事前実行処理が遅延することで再利用を適用できなくなることがあり，並列事前実行コアの処理も無駄になってしまう．従来手法ではこの 3 回分のイタレーションが再利用できていたとすると，従来手法に比べて処理効率が下がってしまうことになる．この問題を回避するためには，展開数を適切な値に設定する必要がある．

3.3.5　オーバヘッドフィルタによる再利用対象の変更

2.3 節で述べたように，自動メモ化プロセッサに備え付けられているオーバヘッドフィルタでは，再利用の適用により性能が落ちてしまうと判断された命令区間はそれ以降再利用に関する処理を行わなくなる．

複数イタレーションをまとめることで，通常実行のサイクル数は展開数に比例して増加するが，3.3.1 項で述べたように再利用オーバヘッドは削減できることが見込める．オーバヘッドフィルタにより再利用を適用しないと判断されたループも，展開数を大きくすることでサイクル数の削減が見込めると再認識され，再利用が適用されるようになる場合がある．再利用率が上がることで，再利用オーバヘッドだけでなく，通常実行のサイクル数も削減ができる．

一方，並列事前実行などの遅延などが原因で，一定期間内の再利用効率が下がってしまうと，オーバヘッドフィルタが当該ループを再利用の対象区間から外してしまうことも考えられる．先ほども述べたが，展開数は単調に増加させるべきではなく，ループによって適切な値を見極めて設定する必要がある．

4　実装

複数のイタレーションをまとめて処理し，一括で再利用をおこなうために，自動メモ化プロセッサの拡張が必要となる．そこで，本章では提案する機構の実装について述べる．

4.1　実装の概略

ループの再利用では，後方分岐に到達し分岐が成立した時にループの命令区間がわかる．また，後方分岐命令の分岐先 (Loop Start) に始まり，同一の後方分岐命令 (Loop end) に終わる命令区間が，1 イタレーションの処理区間となる．そこで，後方分岐命令を判断基準とし，複数回の後方分岐命令で処理を継続させることで，複数イタレーションの入出力情報をまとめた再利用エントリを作る．具体的には，後方分岐命令実

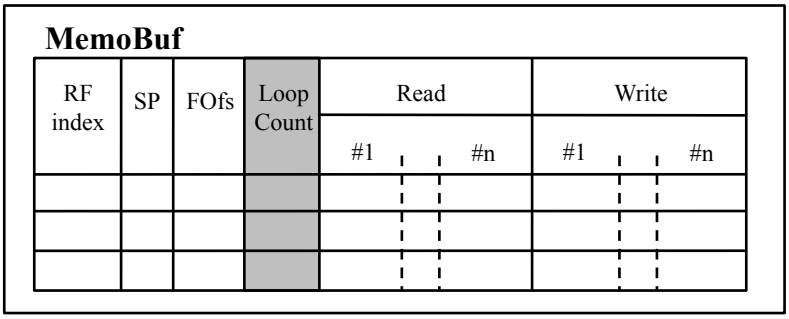

図 14: 拡張後の Memobuf

行時に現在のイタレーション処理回数が展開数に達していない場合，MemoTbl へのエントリ登録をせず，さらに次イタレーションの再利用テストを省く．

ループによって，イタレーションの回数や 1 イタレーションの処理量が異なるため，ループ毎に展開数を設定する．しかし，プログラムを実行する前に，展開数を適切な値に設定することは困難である．そのため，プログラム実行中に再利用機構が展開数を最適な値へと動的に設定する仕組みを導入する必要がある．

4.2 実装モデル

4.2.1 イタレーション一括処理

複数イタレーションを一括で処理するために，複数回の後方分岐命令で処理を継続させる必要がある．そこで，現在処理しているイタレーションが，登録が始まってから何回目のイタレーションであるかを知るために，MemoBuf にカウンタ (LoopCount) を設ける．**図 14** に拡張した MemoBuf を示す．

カウンタの値はプログラム開始時に 0 で初期化される．後方分岐命令に到達した時に，カウンタの値が 0 である場合，再利用テストが行われる．再利用テスト成功時には従来の書き戻しの処理が行われるが，テスト失敗時には入出力情報の登録が開始されるため，このタイミングにカウンタの値を 1 に設定する．再度，後方分岐命令に到達し入出力情報を MemoTbl へと登録する段階になった時，現在のカウンタの値とループ毎に設定されている展開数の値とを比較する．比較の結果，両方の値が一致したときには現在までの入出力情報を 1 つの再利用エントリとして MemoTbl へと登録を行いカウンタの値を 0 とする．一方，比較の結果，カウンタの値が展開数の値に達していなかった場合には，登録処理を行わない．再び，再利用テストが行われるステップとなるが，カウンタの値が 0 ではないため，再利用テストをスキップする．次イタレー

ション開始時に，登録開始処理を省き，カウンタの値をインクリメントすることで，処理が継続される．

カウンタの値が展開数に達していないが，ループ自体が終了してしまう場合がある．例として，イタレーションがループの規定回数を終了した時などが挙げられる．これは，後方分岐命令の不成立により判定ができる．この場合には，例外処理として現在までの入出力情報を MemoTbl へと登録を行い処理を終了する．

この実装により，展開数を任意の値で定めることで，従来手法どおりにバイナリ変更なしで複数イタレーションの一括処理が可能となる．

4.2.2 展開数のループ別動的設定

プログラム中に複数のループがある場合，それぞれのループはイタレーション回数や 1 イタレーションの処理量が異なっている．そのため，一括処理をする最適な展開数はループ毎に異なる．そこで，ループ毎に展開数を設定できるようにするために，再利用対象の命令区間を記憶する RF を拡張し展開数を保持するフィールドを付加する．MemoBuf では，該当する命令区間の開始アドレスを記憶する RFindex を保持しているため，いつでも展開数を参照できる．

プログラム実行前に，最も再利用の効果が出る最適な値に展開数を設定することは困難である．そこで，プログラム実行中に展開数を動的に最適な値へと近づけていくことにする．新規ループを見つけたときの展開数の初期値を 1 とする．これはループを通常実行するときと同じである．

1 つの再利用エントリに登録できる入出力の数には限界がある．また，展開数を変えるとストライド予測からループ再利用をやり直すことになる．ストライド予測にかかってしまうコストも考えると，展開数には上限値を設ける必要がある．そこで，まず MemoTbl の RF の拡張を行った．**図 15** に拡張後の RF を示す．図 15 で示すように展開数 (Collect Value) を追加した．また，展開数が上限値に達するなど判定によって展開数が定まった場合，これ以上変更処理を行わないためにフラグ (stop flag) を設けた．

最適な展開数の判定には既存のオーバヘッドフィルタを利用する．オーバヘッドフィルタでは，命令区間に対して再利用適用時に効果が得られるかを判定し，再利用対象区間とするかの判別を行っている．そこで，展開数の値を初期値から順に増やしていき，変化前と変化後でのオーバヘッドフィルタの判定に用いる値を判断基準として用いる．

具体的には，展開数を変化させた時に，変化前と変化後での再利用により得られる効果を比較する．2 章で示したように，オーバヘッドフィルタでは 2 つの計算式を用い

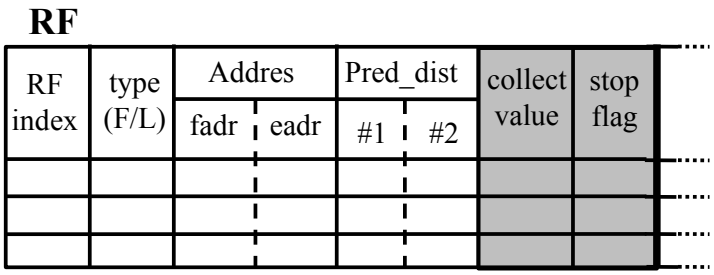

図 15: 拡張後の RF

て判定を行っている．1 つは再利用により実際に削減できるサイクル数であり，もう 1 つは再利用失敗時における MemoTbl の検索コストである．この 2 つの値の差分が再利用により得られる効果となる．以下，この値を**ゲイン**と呼ぶ．

展開数を変化させた時にこのゲインの値が増えるようなら更なる高速化ができることになる．そこで，各展開数でゲインの値を記録するようにオーバヘッドフィルタの拡張を行った．**図 16** に拡張後のオーバヘッドフィルタを示す．図 16 に示すゲイン (gain) は，各展開数での値を覚えるために配列となっている．またその要素数は展開数が上限値に達するまでに取り得る値の数だけ用意する．これにより，記録した値を用いて比較を行う．

また，ゲインの値は展開数に依存している．例えば，展開数が 2 である場合には，イタレーション 2 回分の再利用によって得られる効果となっている．そのため，各展開数毎のゲインを比較するには，イタレーション 1 回分のゲインに換算する必要がある．そこで，拡張後のオーバヘッドフィルタでは，2 つの計算式の差分から得られた値を展開数で割ったものをゲインとして保持する．2.3 節で示した数式 (1),(2) をそれぞれ $cycle1$，$cycle2$ とすると，ゲインは

$$cycle1 = (N^{Main} + N^{SpMT}) \times (S - Ovh^R - Ovh^W) \tag{1}$$

$$cycle2 = (T - N^{Main} - N^{SpMT}) \times Ovh^R \tag{2}$$

$$gain = (cycle1 - cycle2)/CollectValue$$

として計算する．

ゲインの計算には，一定期間におけるメインコアによる登録回数 N^{Main} や並列事前実行コアが登録したエントリの使用回数 N^{SpMT} を用いる．そこで，展開数を変化させた場合，一定期間は展開数を固定して再利用状況を判定に加える必要がある．既存のオーバヘッドフィルタでは，N^{Main} と N^{SpMT} に 64 ビットシフトレジスタを用いて

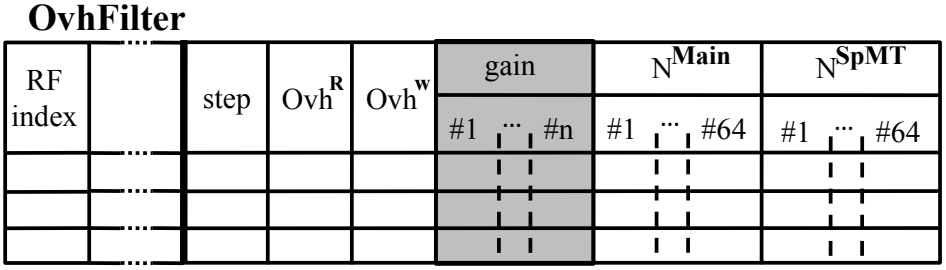

図 16: 拡張後のオーバヘッドフィルタ

いるため，それに合わせて 64 回の再利用テストが行われるまでを一定期間とする．また，展開数 $CollectValue$ の取り得る値をシフトレジスタに合わせて 2 のべき乗値とする．これにより，シフト演算で展開数やゲインの計算が可能となり，追加ハードウェアが単純なものになる．

また，展開数を変化させた時には，RF やオーバヘッドフィルタの値の一部を初期化する必要がある．変化前のストライドでは並列事前実行の効果を得られないし，オーバヘッドフィルタの値をそのまま用いては新しい展開数での効果を正しく判定できないからである．そのため，ストライド予測に用いる最近出現した 2 組の入力 (Pred_dist) や，オーバヘッドフィルタの判定に用いる値 (S,Ovh^R,Ovh^W,N^{Main},N^{SpMT}) は，展開数を変化させた時に初期化を行う．

以上により，プログラムを通常通り実行するだけで自動的に展開数が設定されていき，複数のイタレーションが一括で再利用できるようになる．

5 評価

本提案手法を実現するために既存の自動メモ化プロセッサに対して追加実装を行った．提案手法の有効性を示すためベンチマークプログラムを用いて評価と考察を行った．

5.1 評価環境

実行環境には，計算再利用のための機構を実装した単命令発行の SPARC-V8 シミュレータを用いた．想定する CPU 環境を**表 2** に示す．なお，キャッシュや命令レイテンシは SPARC64-III[4] を，MemoTbl 内の RB に用いる CAM の構成はルネサステクノロジ社の R8A20410BG[5] を参考にしている．また，評価対象のプログラムには，汎用ベンチマークプログラムである SPEC CPU95 ベンチマークを用いた．

再利用テストのコストとして，レジスタと CAM を 32 バイト比較するのに 9 サイク

表 2: 評価環境

D1 Cache 容量	32KByte
ラインサイズ	32Byte
ウェイ数	4way
レイテンシ	2cycle
Cache ミスペナルティ	10cycle
共有 D2 Cache 容量	2MByte
ラインサイズ	32Byte
ウェイ数	4way
レイテンシ	10cycle
Cache ミスペナルティ	100cycle
Register Window 数	4set
Window ミスペナルティ	20cycle/set
MemoTbl サイズ	32byte × 4K 行 (16KByte)
レジスタ ⇔ CAM	9cycle/32byte
メモリ ⇔ CAM	10cycle/32byte
CAM ⇒ レジスタ or メモリ	1cycle/32byte

ル，メモリと CAM を 32 バイト比較するのに 10 サイクルを要するものとする．現在の一般的なアーキテクチャでは，CPU コア内部のクロック速度は，外部のメモリパスのクロック速度の 10 倍程度で動作している．そのため，レジスタやメモリと，CPU コア外部にある MemoTbl との比較に要するサイクル数は現実的なコストとなる．

また，提案手法におけるパラメータの値設定として，展開数の上限値を 32 とした．つまり，展開数の取り得る値は 1,2,4,8,16,32 となる．展開数変動後の固定期間は先ほど述べたように再利用テスト実行 64 回とする．

5.2 評価結果

評価結果のグラフは，左から順に

(N) メモ化なし

(M) メモ化あり

(C2) 2 イタレーションの一括再利用 (展開数を 2 で固定)

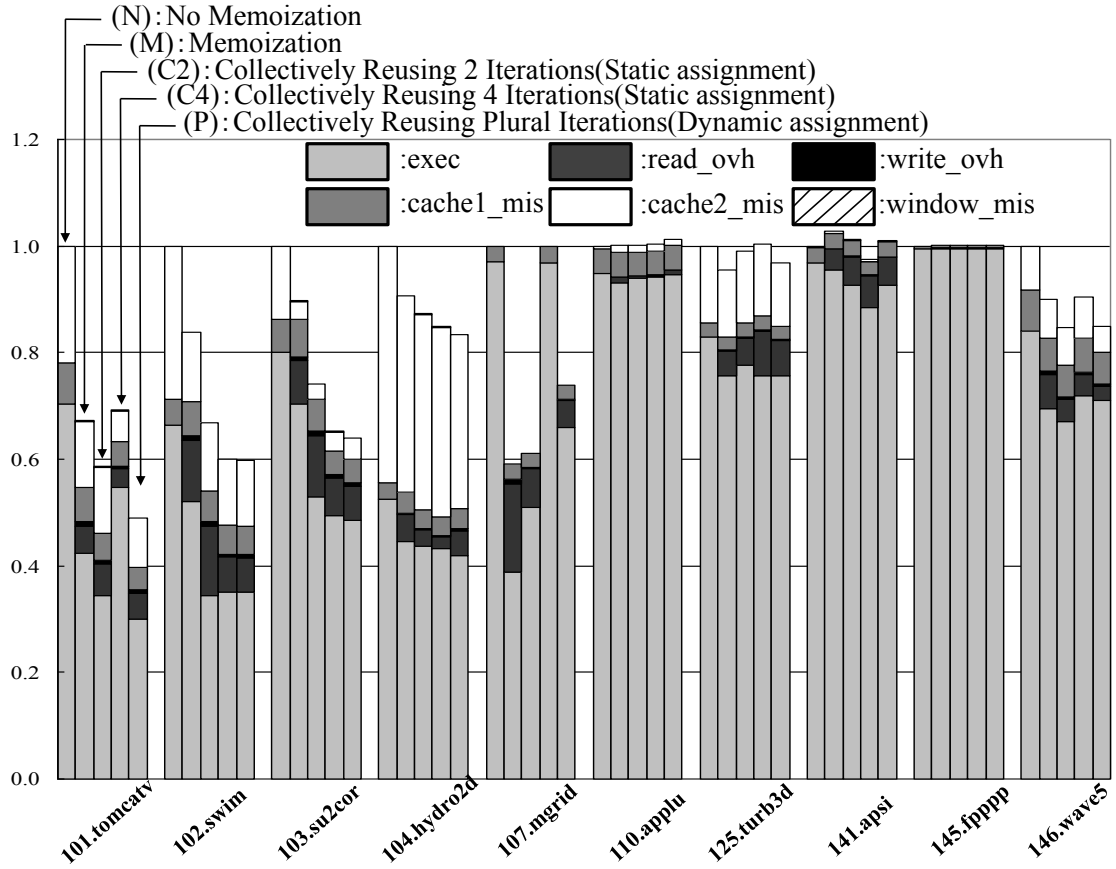

図 17: 評価結果

(C4) 4 イタレーションの一括再利用 (展開数を 4 で固定)

(P) 複数イタレーションの一括再利用 (提案手法)

となる．グラフは各モデルの総実行サイクル数を表しており，それぞれメモ化なし (N) を 1 として正規化した．自動メモ化プロセッサには，再利用を行うメインコア 1 つと並列事前実行コア 3 つが割り当てられている．また，本研究の有効性を確認するために，関数再利用は行わずループの再利用のみ適用した．比較検証のために，全てのループに対して展開数をそれぞれ 2 と 4 に固定した静的なモデル (C2，C4) での評価も行っている．従来手法のメモ化あり (M) は，展開数が 1 で固定された静的モデルと同等であるといえる．SPEC CPU95 ベンチマークの結果を**図 17** に示す．

図 17 の凡例は順に，exec は命令の実行に要したサイクル数，read_ovh は MemoTbl の比較に要したサイクル数 (検索コスト)，write_ovh は MemoTbl の出力をレジスタやメモリに書き込む際に要したサイクル数 (書き戻しコスト)，cache1_mis および cache2_mis は 1 次と 2 次データキャッシュミスにより要したサイクル数，window_mis は SPARC アー

キテクチャ特有のレジスタウインドウミスによって要したサイクル数である．read_ovh と write_ovh を合わせて再利用オーバヘッドとなる．

図 17 の SPEC CPU95 ベンチマークにおける評価結果を説明する．全てのループの展開数を 2 と 4 で固定した静的モデル (C2)(C4) では，従来のメモ化を行う (M) に比べて多くのプログラムで高速化できている．特に，102.swim や 103.su2cor では大幅にサイクル数を削減している．しかし，101.tomcatv や 146.wave5 のように (C2) では (M) に比べて高速化できているが，(C4) になると総サイクル数が (M) よりも増加しているプログラムも存在している．また，107.mgrid や 125.turb3d では (C2)，(C4) のどちらともが (M) に比べ，総サイクル数が増加してしいる．特に，107.mgrid は (M) の結果からわかるように，並列事前実行による高速化が有効であるにも関わらず，(C4) では再利用なしの (N) と同等になってしまっている．以上のことから，ループ毎に最もサイクル数の削減できる展開数が異なる．つまり，ループ毎に最適な展開数があることがわかる．

提案手法である (P) では，従来手法である (M) と比べると概ね良好な結果が得られている．特に，101.tomcatv,102.swim,103.su2cor では大幅なサイクル数を削減できている．他にも，104.hydro2d や 146.wave5 といったプログラムで提案手法適用による効果が得られている．しかし，110.applu や 145.fpppp などのプログラムでは，元々並列事前実行による効果が得られていないため，並列事前実行の拡張である本提案手法ではあまり効果を得られなかった．また，107.mgrid では提案手法適用により (M) に比べてサイクル数が増えてしまっている．それでも，メモ化なし (N) に比べて 30 %ほどのサイクル数の削減はできている．

結果をまとめると，SPEC CPU95 ベンチマークでメモ化なし (N) に比べ，従来手法では最大で 40.9 %，平均で 13.3 %のサイクル数の削減だったのに対し，提案手法では最大で 51.0 %，平均で 20.8 %のサイクル数を削減できた．

5.3 考察

従来手法 (M) と提案手法 (P) を比べると多くのプログラムでサイクル数が削減できている．複数イタレーションの一括再利用により実行サイクル数が削減できる理由は 2 つあると考えられる．1 つは，3.3.1 項でも述べたように再利用オーバヘッドの削減が挙げられる．146wave5 は再利用オーバヘッド削減分が総実行サイクル数の削減に繋がった例であるといえる．また，静的モデルの例となるが，102.swim では (C2) から (C4) へと結果が顕著に現れている．もう 1 つは，exec のサイクル数の削減である．

101.tomcatv,102.swim,103.su2cor では，exec のサイクル数の大幅な削減により総実行サイクルが削減できている．これは，オーバヘッドフィルタによる再利用対象が増えたと考えられる．今までは再利用によるサイクル数の削減が見込めなかったループが，複数イタレーションをまとめたことで再利用による効果を得られると判断されるようになったからである．再利用対象が増えることで再利用オーバヘッドが増えてしまうが，得られるゲインの値によって展開数を判定しているため，増加する再利用オーバヘッドより削減できる exec サイクル数のほうが大きい．

一方，複数イタレーションの一括再利用によって従来手法より実行サイクル数が増加してしまう場合がある．107.mgrid や 125.turb3d のプログラムでその傾向が見られ，特に 107.mgrid では (M) に比べ (P) の総実行サイクル数が大きく増加してしまっている．この 2 つのプログラムの従来手法である (M) と静的モデルの (C2,C4) を比較すると，(M) の総実行サイクル数が一番少ないことから，ほとんどのループの最適な展開数は 1 であることがわかる．提案手法では，性能比較のためにゲイン値の比較を展開数毎に行っている．そのため，どうしてもゲイン値が減少する展開数で動作する期間が存在することになる．例えば，最適な展開数が 1 であるループの場合，ゲイン値が減少する展開数が 2 でゲイン値を判定するために動作する期間が存在する．この期間の存在が実行サイクル数の増加をまねく原因であると考えられる．107.mgrid では展開数 4 である (C4) でほぼ再利用が行われずメモ化なし (N) と同じような結果になってしまっている．展開数が 4 になると再利用が行われなくなってしまうため，107.mgrid の中で最適な展開数が 2 になるループは全く効果の出ない期間が存在することになり，これだけの実行サイクル数が増加してしまったといえる．性能が急激に悪化してしまう期間を避けることが今後の課題となる．

6　おわりに

本研究では，従来までの自動メモ化プロセッサに対し更なる高速化手法として，複数イタレーションを一括で再利用する手法を提案した．提案手法の有効性を確認するため，SPEC CPU95 ベンチマークを用いて評価を行った．その結果，通常通りに命令を実行するのと比較して，従来手法では最大で 40.9 %，平均で 13.3 %のサイクル数の削減だったのに対し，提案手法では最大で 51.0 %，平均で 20.8 %のサイクル数を削減し有効性が確認できた．

本研究の今後の課題として，ソフトウエアにより自動メモ化プロセッサを支援し，更なる高速化を目指すことが考えられる．バイナリのみでなく，ソースコードが公開さ

れているプログラムについては，プログラム内のメモ化のためのヒント情報を埋め込む等のソフトウェアによる支援を行うことで更なる高速化が可能であると考えられる．このような研究は既にある程度行われており，有用であることが示されている．しかし，それをマルチスレッドへ応用した場合については未だ検証されていない．そこで，並列事前実行の際に用いるストライド値をプログラム内に埋め込んで，並列事前実行により確実に有効なエントリが登録できるようにするなど，ソフトウェアによるメモ化とマルチスレッド技術を組み合わせることで，新たな知見が得られると考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室の方々に深く感謝致します．

参考文献

[1] 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).

[2] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).

[3] Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction Using Hybrid Predictors, *30th MICRO*, pp. 281–290 (1997).

[4] HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).

[5] ルネサステクノロジ: ニュースリリース: R8A20410BG (2009).