卒業研究論文

ループの特徴を利用した再利用表パージ手法による自動メモ化プロセッサの高速化

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 21 年度入学　21115072 番

柴田 裕貴

平成 25 年 2 月 12 日

ループの特徴を利用した再利用表パージ手法による 自動メモ化プロセッサの高速化

柴田 裕貴

内容梗概

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は困難になってきた．こうした中で，SIMD やスーパスカラなどの命令レベル並列性（ILP）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，多くのプログラムは明示的な ILP を持たないことから，これら手法にも限界がある．そこで，計算再利用をハードウェアにより動的に適用する自動メモ化プロセッサに関する研究が行われている．

自動メモ化プロセッサは，関数およびループを計算再利用の対象区間と見なし，実行時にその入力と出力の組を記憶しておくことで，再び同一命令区間を同一入力を用いて実行しようとした際に，その実行自体を省略する．ループでは，1 回のループイタレーション処理に対する入出力が再利用表へ登録される．そして，ループのうち入力が単調変化するものに対し，入力を過去の履歴から予測し，その予測された値を用いて命令区間を別コアで予め実行しておくことで出力を生成・記憶する，並列事前実行と呼ぶモデルが提案されている．これにより，予測が正しかった場合はメインコアによる当該イタレーションの実行が計算再利用により省略できる．

しかし，入力が単調に変化する場合，1 度再利用されたループの入出力は今後再利用される見込みはないと考えられる．さらに，既存の自動メモ化プロセッサでは，関数とループの入出力を区別することなく再利用表から LRU に基づいて追い出すため，再利用成功時に，このような入出力が最新のタイムスタンプで更新され，再利用表から追い出されにくくなってしまう問題があった．そこで，ループの今後再利用が見込めない入出力を再利用表から強制的に追い出す手法を提案する．これにより，再利用される見込みのある他の入出力をより多く再利用表に登録しておくことができるため，再利用率が向上し，自動メモ化プロセッサの高速化が期待できる．

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を実装し，SPEC CPU95 ベンチマークを用いてシミュレーションにより評価した．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 40.5 %，平均 9.1 %のサイクル数の削減であったのに対し，提案手法では最大 41.8 %，平均 9.5 %のサイクル数を削減し，有効性を確認した．

ループの特徴を利用した再利用表パージ手法による
自動メモ化プロセッサの高速化

目次

1　はじめに　1

2　自動メモ化プロセッサ　2

2.1　再利用機構の構成 2
2.2　並列事前実行 6
2.3　パージアルゴリズム 9
2.3.1　TSID パージ 9
2.3.2　FLID パージ 9
2.4　オーバヘッドフィルタ機構 10

3　ループの特徴を利用したパージ手法　11

3.1　ループイタレーションパージ 11
3.2　提案手法の適用による効果 13
3.3　動作モデル 14

4　実装　15

4.1　ハードウェア拡張 15
4.2　動作モデル 17
4.2.1　登録時の動作 17
4.2.2　追い出し時の動作 23

5　評価　24

5.1　評価環境 24
5.2　評価結果 24
5.3　考察 27

6　おわりに　28

参考文献　29

1　はじめに

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プロセッサの性能向上が難しくなってきた．このため，SIMD やスーパースカラ等の命令レベル並列性に基づく高速化手法が注目された．また，近年では高い性能と低消費電力を両立させる観点から，SPARC T4[1] や Opteron[2] などの，複数コアを搭載したマルチコアプロセッサが主流となっている．そして，今後集積度の向上に伴って，100 コア構成の TILE-Gx[3] が予定されているように，搭載コア数をさらに増大させたメニーコアプロセッサが一般化していくと予想されている．

これらのプロセッサ高速化手法はいずれもプログラムの持つ並列性に着目し，高速化を図る手法である．一方で，計算再利用技術に基づいた高速化手法である自動メモ化プロセッサ [4] が提案されている．並列化が，処理全体の総量自体は変化させず複数の単位処理を同時実行により高速化を図るのに対し，計算再利用は処理自体を省略することで高速化を図る手法であり，その着眼点は根本的に異なっている．

自動メモ化プロセッサは，関数およびループを計算再利用の対象区間と見なし，実行時にその入力と出力の組を再利用表と呼ばれる表に登録する．そして，再び同じ命令区間を同一入力を用いて実行しようとした際に，過去の出力を書き戻すことで，その命令区間の実行を省略する．計算再利用の対象区間の 1 つである関数はプログラムの実行中に何度も呼び出されることが多く、またループは同じイタレーション処理が何度も繰り返されることから、これらの命令区間は再利用に成功する見込みがある．しかし，ループは入力の 1 つであるイタレータ変数が単調に変化していくため，実際には再利用に成功しない．そこで自動メモ化プロセッサでは，予測した入力を用いて投機実行コアに命令区間を予め実行させ，その入出力を再利用表に登録しておくことでループ区間の再利用に成功する．

しかし，入力が単調変化するループにおいて 1 度再利用に成功した入出力は，今後再利用される見込みがないと考えられる．このため，再利用表に登録されている入出力を等しく LRU に基づき追い出す自動メモ化プロセッサでは，このような入出力が再利用成功時に最新のタイムスタンプで更新され，再利用表から追い出されにくくなってしまう問題があった．そこで，ループの今後再利用が見込めない入出力を強制的に再利用表から追い出す手法を提案する．これにより，他の入出力をより多く再利用表に登録しておくことができるため，再利用率が向上し，自動メモ化プロセッサの高速化が期待できる．

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について述べる．3 章では，ループの特徴を利用したパージ手法を提案し，4 章でその実装方法について説明する．5 章で本提案手法の評価を行い，最後に 6 章で結論を述べる．

2　自動メモ化プロセッサ

本章では，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，アーキテクチャの構成，動作，再利用表の追い出しアルゴリズムを概説する．

2.1　再利用機構の構成

計算再利用（Computation Reuse）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．また，この手法を命令区間に適用することを**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．ただし，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やバイナリ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メモ化 [6] はオーバヘッドが大きく，限られたプログラムでしか性能向上が得られない傾向がある．そこで，ハードウェアを用いて動的にプログラムを解析し，メモ化を適用することで，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（Auto-Memoization Processor）**である．

自動メモ化プロセッサの構成の概略を**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，コアの内部に 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 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，命令区間およびその入力と出力の組を記憶しておく表である **MemoTbl** とメモ化制御機構（Memoize engine），および MemoTbl への書き込みバッファとして働く **MemoBuf** を持つ．MemoTbl はサイズが大きく，コアからのアクセスコストが大きい．そのため，自動メモ化プロセッサは検出された入出力を MemoTbl ではなく，アクセスコストの小さい MemoBuf に格納しつつ当該命令区間を通常実行し，実行終了時に MemoBuf の内容を一括で MemoTbl に格納する．これにより，MemoTbl へのアクセス回数を減らしオーバヘッドを削減している．

次に MemoBuf の詳細な構成を**図 2** に示す．MemoBuf は複数のエントリを持ち，1

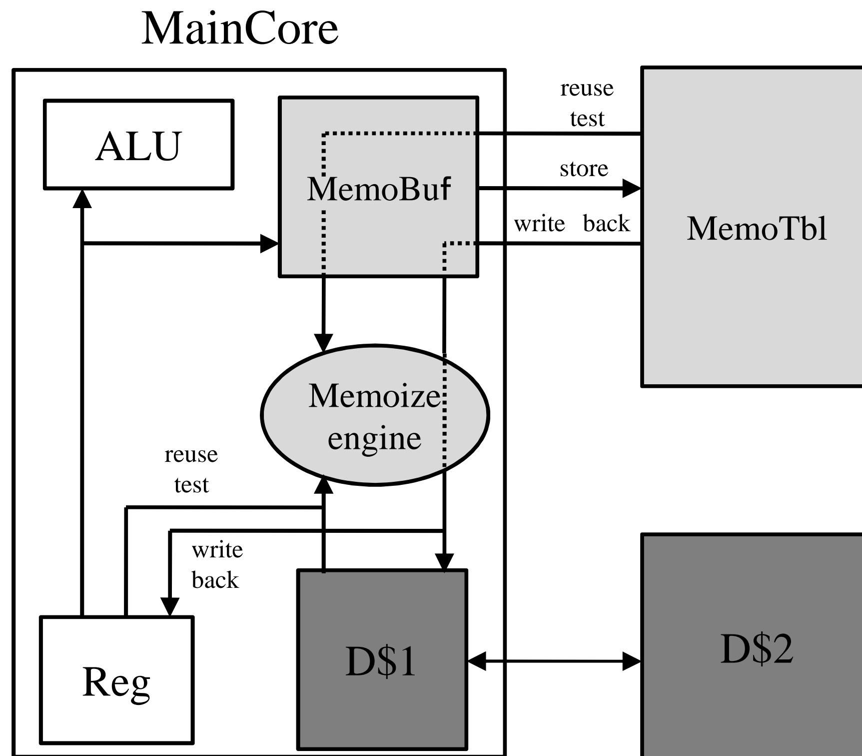

図 1: 自動メモ化プロセッサのハードウェア構成

エントリが 1 入出力セットに対応する．各エントリは，メモ化対象となる命令区間の開始アドレスを記憶する Start Addr，命令区間の実行開始時のスタックポインタを記憶する SP，関数の戻りアドレスとループの終端アドレスを記憶する retOfs，命令区間の入力セットを記憶する Read，出力セットを記憶する Write のフィールドを持つ．また，入れ子構造になった命令区間もメモ化対象とするために，MemoBuf はポインタ Memobuf_top を命令区間の検出時にインクリメントし，命令区間の実行終了時にデクリメントすることで入れ子構造を保持している．

さて，一般に命令区間内では，複数の入力値が順に参照され使用される．しかし，同じ命令区間でも，その入力アドレスの列は分岐していく場合がある．例えば，条件分岐命令を実行すると，次に参照される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．このように，ある命令区間の入力アドレスの列はその入力値によって分岐していく．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 に格納する．例えば，自動メモ化プロセッサが**図 3** に示すサンプルプログラムを実行する場合，関数 func の入力エントリについて**図 4** のような木構造を作る．なお，図 4 の input1，input2，input3 は，関数 func を呼び出してから，何番目に参照

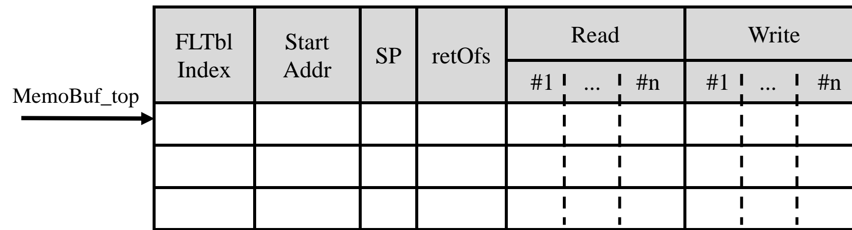

図 2: MemoBuf の構成

```
1   : int a, b, c;
2   : int func(x){
3   :   int tmp = x + a;
4   :   if(tmp<=5)
5   :     return(tmp * b / 2);
6   :   else
7   :     return(tmp * c / 2);
8   : }
9   : int main(){
10   :   a = 4;, b = 2, c = 8;
11 :   func(1); /* x = 1, a = 4, b = 2 */
12 :   b = 6;
13 :   func(1); /* x = 1, a = 4, b = 6 */
14 :   a = 5;
15 :   func(1); /* x = 1, a = 5, c = 8 */
        ...
```

図 3: サンプルコード

された入力値であるかを示している．また，図 4 のノードは命令区間の入力値を，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いる．そして，入力セット（A）,（B）,（C）はそれぞれサンプルプログラムの 11 行目，13 行目，15 行目における関数呼び出しに対応する．この例において，入力セット（A）と入力セット（B）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（C）では，変数 c が 3 番目に参照される．これは，2 番目に参照される変数 a の値が異なることにより，プログラムの

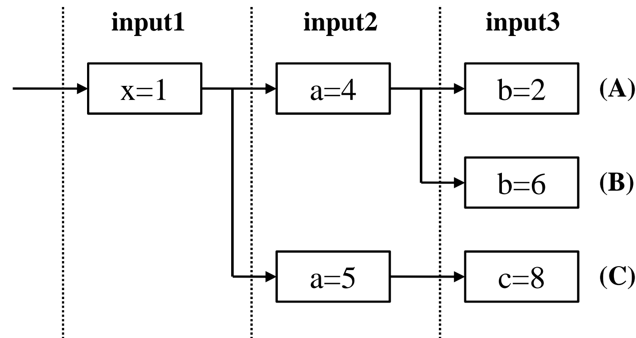

図 4: 入力エントリが成す木構造

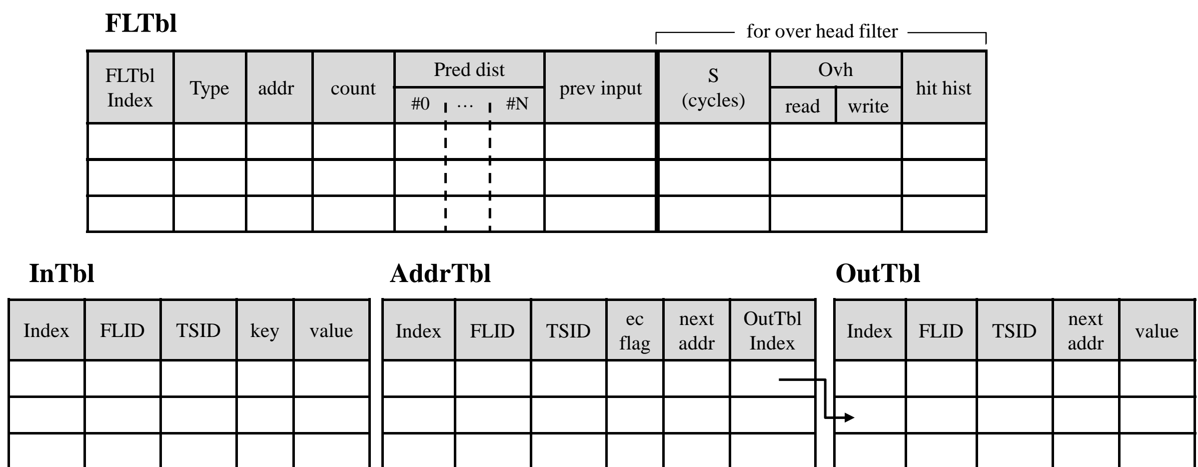

図 5: MemoTbl の構成

4 行目における条件分岐の結果が変化し，次入力アドレスが変化したためである．

次に MemoTbl の詳細な構成を**図 5** に示す．MemoTbl は，命令区間を記憶する **FLTbl**，入力を記憶する **InTbl**，入力アドレスを記憶する **AddrTbl**，および出力を記憶する **OutTbl** の 4 つから構成される．なお，InTbl，AddrTbl，OutTbl をまとめて入出力表と呼ぶ.

FLTbl は再利用対象となる各命令区間に対応する行を持っており，その行番号（FLTbl Index）を各命令区間の識別番号とする．また，各行にはメモ化のためのフィールドおよび後述するオーバヘッドフィルタのためのフィールドを持っている．メモ化のためのフィールドには，関数およびループの区別（Type），命令区間開始アドレス（fadr）を持ち，命令区間ごとに InTbl の使用エントリ数 (count) を記憶する．また，後述する並列事前実行の入力ストライド予測に用いるための直近の入力値セット（prev inputs），

各イタレーションの実行担当コア番号（Pred_dist）を記憶するフィールドがある．オーバヘッドフィルタのためのフィールドには，当該命令区間のサイクル数（S），過去の再利用に要した入力検索及び出力書き戻しオーバーヘッド（Ovh read/write），過去の再利用ヒット履歴 (hit hist) が保持される．

InTbl は，命令区間の入力を記憶する表である．各行は FLTbl の行番号（FLTbl Index）に対応する FLID を持ち、この値によってどの命令区間の入力値を記憶しているのかを判別する．そして，命令区間の全入力パターンを上述した木構造で管理するため，入力値（value）に加えて，親エントリのインデクス（key）を持つ．また，InTbl のエントリを LRU に基づいて追い出すためにタイムスタンプ（TSID）も持つ．

AddrTbl は，命令区間の入力アドレスを記憶する表であり，InTbl と同数のエントリを持ち，同じ Index を持つ InTbl のエントリの次入力アドレス (next addr) を記憶する．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，終端エントリである場合，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クス（OutTbl Index）も持つ．

OutTbl は，命令区間の出力を記憶する表であり，命令区間の出力値（value）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クス（next Index）も持つ．

2.2　並列事前実行

自動メモ化プロセッサでは，関数とループを再利用対象区間とし，これらの命令区間を過去に完全に同一の入力セットで実行した場合にのみ効果が得られる．よって，入力の一つであるイタレータ変数が単調に変化するループでは，計算再利用による効果が全く得られない．そこで，計算再利用をしながら実行を進めるメインコアの他に，メインコアが実行した命令区間の過去の入力に基づいて，未来の入力値を予測し，同一命令区間をメインコアに先駆けて投機実行するコアを複数基備えるシステムが提案されている．これを**並列事前実行**と呼び，以下この投機実行コアを **SpC**（**Speculative Core**）と呼ぶこととする．この SpC は予測された入力値を用いてメインコアが実行しようとする区間を事前に実行しておき，実行結果を MemoTbl に格納する．ここで，入力値予測が正しかった場合，メインコアはその区間の実行を省略することができる．また，入力の予測が外れた場合でも，並列事前実行によるオーバヘッドは MemoTbl の検索に要する時間のみであり，投機実行に起因するオーバヘッドは発生しない．ただし，エントリ数に制限のある MemoTbl に不必要なエントリが登録されることにより，

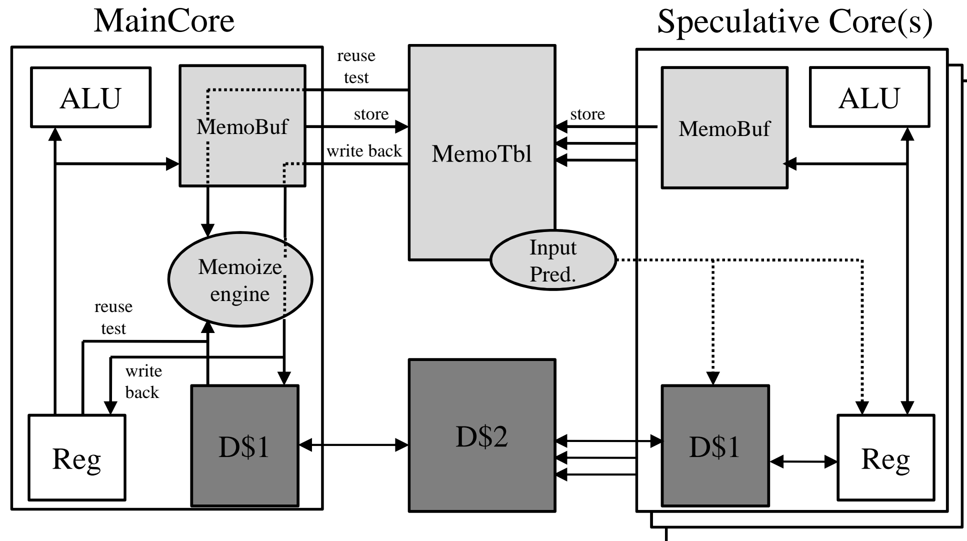

図 6: 並列事前実行機構を備えた自動メモ化プロセッサのハードウェア構成

有効なエントリが追い出され，並列事前実行をしない場合よりも性能が低下してしまう可能性はある．

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ハードウェア構成を示す．SpC は複数基備えることができ，各 SpC は，それぞれ MemoBuf と 1 次キャッシュを持ち，MemoTbl と 2 次キャッシュは全コアで共有する．MemoBuf を各コアごとに持っているので，命令区間の実行終了時に各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．また，MemoTbl は共有されているため，メインコアは SpC によって MemoTbl に登録されたエントリを用いて再利用を適用することができる．

並列事前実行のためには，当該命令区間の過去の入力値から未来の入力値を予測し，その予測した入力値を SpC に与える必要があり，そのために小さなハードウェア（Input Pred）を自動メモ化プロセッサに設けている．その際，他のコアが実行しているイタレーションと同一のイタレーションの入力値を SpC に与えてしまうのを回避するために，各コアが実行しているイタレーションを管理する必要がある．そのため，メインコアが実行しているイタレーションを基点として，そこから数個先までのイタレーションの実行担当コア番号を記憶しておくフィールド（Pred_dist）を FLTbl に設けている．これにより，他のコアが実行していないイタレーションが分かるため，FLTbl の FLTbl Hist フィールドに記憶されている過去の入力値から，ストライド予測 [7] により，そのイタレーションを投機実行するための入力値を得る．

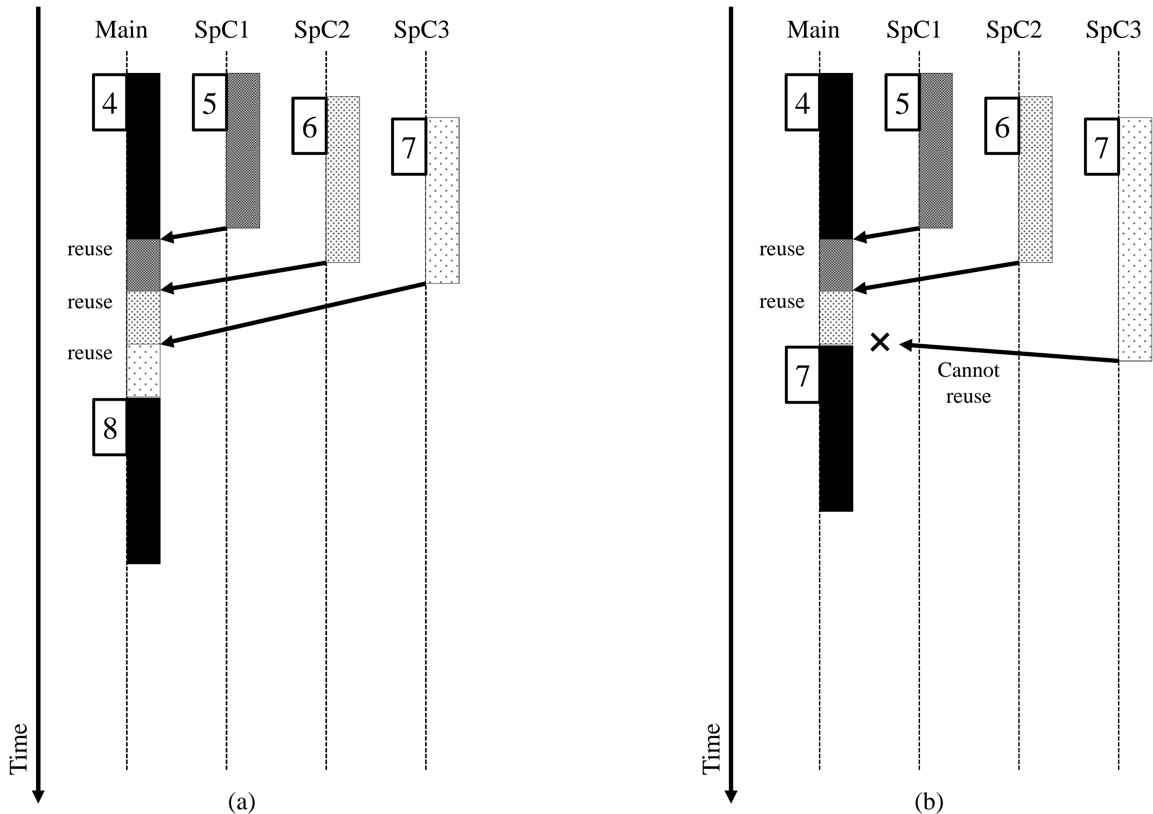

図 7: 並列事前実行の流れ

次に，SpC を用いた場合の並列事前実行のタイミングチャートを**図 7** に示す．この例では，SpC を 3 基とし，あるループに対してメインコアが入力値 4 に対応するイタレーションを通常実行しているとする．また，それに並行して SpC はストライド予測を用いてそれぞれ入力値 5，6，7 に対応するイタレーションを実行するとする．

ここで，図 7 の (a) は最も効率良く再利用を適用できる例を示している．この場合，メインコアが入力値 4 に対応するイタレーションの実行を終了し，入力値 5，6，7 に対応するイタレーションの実行に移ろうとした時，3 基の SpC によりこれらの入力値に対応するイタレーションの事前実行は既に完了しており，MemoTbl にその入出力セットは既に登録されている．一方，図 7 の (b) は，キャッシュミスの発生などにより SpC3 で入力値 7 に対応するイタレーションの実行に遅延が発生してしまった例である．この場合，メインコアは SpC3 と同じ入力値に対応するイタレーションに再利用を適用できず，そのイタレーションの実行を開始してしまう．

なお，メインコアと SpC では 2 次データキャッシュや主記憶が共有されている．このため，コアがこれらの共有領域に対して書き込むと，他の SpC やメインコアの実行

においてデータの不整合が生じてしまう．そこで，このような問題を回避するために，SpC はこれらの共有領域に値を書き込むことはせず，代わりに MemoBuf に命令区間の出力値を記憶する．

2.3　パージアルゴリズム

自動メモ化プロセッサの MemoTbl の容量は有限であるため適宜エントリを追い出す必要がある．そこで，自動メモ化プロセッサは LRU に基づく仕組みと明示的に特定のエントリを追い出す仕組みの 2 つを備えている．

2.3.1　TSID パージ

TSID パージは LRU に基づいた追い出し手法である．MemoTbl から最も長い期間使われていないエントリを追い出すために，自動メモ化プロセッサはリングカウンタを用いて時刻管理を行っている．新しくエントリを MemoTbl に登録するとき，そのエントリの TSID フィールドに現在のリングカウンタの値を記憶し，リングカウンタは MemoTbl に一定数のエントリが登録されるごとに更新される．また，再利用に成功した時，その入出力エントリの TSID を現在のリングカウンタの値に更新する．そして，リングカウンタが更新されるたびに，更新後と同一の TSID を持つエントリを MemoTbl から追い出す．なお，メインコアが入出力表にエントリを登録するときのみリングカウンタを更新し，TSID パージを適用することで，SpC によりエントリが大量に登録された場合にリングカウンタの更新頻度が高くなり，エントリが追い出されやすくなるのを防いでいる．

ここで，TSID パージでは解決できない問題が 2 つある．1 つ目の問題は，命令区間を記憶する FLTbl は TSID パージによる追い出しをしないため，一定量の登録があると FLTbl が溢れてしまう問題である．2 つ目の問題は，多くのエントリが頻繁に再利用に成功しタイムスタンプが更新されると，入出力表へのエントリの登録量が追い出す量より多くなることで，エントリの登録途中で入出力表が溢れてしまう問題である．この問題は，SpC によるエントリの登録が多く発生する場合でも発生する．これら 2 つの問題を解決するために，自動メモ化プロセッサはもうひとつのパージアルゴリズムである FLID パージを備えている．

2.3.2　FLID パージ

FLID パージは FLTbl や入出力表が溢れた場合に，特定の命令区間のエントリを明示的にパージする手法である．FLTbl が溢れた場合は，既に登録されている命令区間のうち，最近検索や登録が行われておらず再利用成功回数も低い命令区間を追い出す．

このとき，追い出された命令区間の FLID を持つ入出力表のエントリは，FLTbl からたどることのできない無効なエントリとなってしまう．そのため，これら無効なエントリを入出力表からすべて追い出す．一方，入出力表が溢れた場合は，エントリの登録が途中までしか完了していないため，当該命令区間の入力エントリの木構造が崩れてしまっている可能性がある．そこで，この問題を解決するために，当該命令区間の FLID を持つエントリも入出力表からすべて追い出す．なお，この入出力表が溢れた際の FLID パージは，大きな性能低下に繋がる可能性がある．これは，当該命令区間が頻繁に再利用される命令区間であった場合，その命令区間の有益なエントリが全て追い出されてしまうからである．

2.4 オーバヘッドフィルタ機構

自動メモ化プロセッサは，計算再利用可能な命令区間の実行を省略することで高速化を図る手法であるが，その際には MemoTbl を検索するコスト，および入力が一致したエントリに対応する出力を MemoTbl からレジスタやメモリに書き戻すコストがオーバヘッドとして発生する．命令区間によっては，これらのオーバヘッドが大きく，再利用を適用することで却って性能が悪化してしまう場合がある．そこで，FLTbl では，各命令区間に対し一定期間における再利用の状況をシフトレジスタ（図 5 中 hit hist.）を用いて記録し，それぞれの命令区間の再利用適応度を算出している．

ある命令区間について，最近の一定回数 T の再利用試行における再利用成功回数 M は上記シフトレジスタから得られる．この値と，当該命令区間の過去の省略サイクル数 S から，実際に削減できたサイクル数を

$$M \cdot (S - Ovh^R - Ovh^W) \tag{1}$$

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，当該命令区間の再利用表検索オーバヘッド，および再利用表からキャッシュ等への書き戻しオーバヘッドである．

また，再利用が行われなかった場合でも，再利用表の検索オーバヘッドは存在する．このオーバヘッドは，

$$(T - M) \cdot Ovh^R \tag{2}$$

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．式 (1) から式 (2) を引いた

ものを $\boldsymbol{Gain}$ とすると，

$$Gain = M \cdot (S - Ovh^W) - T \cdot Ovh^R \tag{3}$$

となり，この $Gain$ が正値であれば再利用の効果があると判断できる．再利用表に小さなハードウェアを付加することによってこれを計算し，再利用の効果が得られると判断された命令区間に対してのみ再利用表への登録および再利用を行っている．

3　ループの特徴を利用したパージ手法

本章では，既存の自動メモ化プロセッサの問題点を述べ，本研究で提案するループの特徴を利用した再利用表パージ手法，およびそれにより得られる効果を説明する．

3.1　ループイタレーションパージ

自動メモ化プロセッサは，関数とループを等しく計算再利用の対象区間としている．しかし，関数とループは異なる特徴を持つ．関数はプログラム中のどの場所からも呼び出される可能性があるため，関数の入出力エントリが再利用されるタイミングは予測できない．そのため，関数の入出力エントリは入出力表に登録されている間，再利用される可能性がある．一方で，ループは一般にイタレーションごとに入力が単調変化していくため，全く同じ入力で再び実行されることはほとんどない．そのため，ループの入出力エントリは，そのエントリに対応するイタレーションの実行をメインコアが開始する際にのみ，再利用される可能性があり，それ以降のイタレーションでは，再利用が見込めない．以降，関数の入出力エントリを関数エントリ，ループの入出力エントリをループエントリと呼ぶ．

既存の自動メモ化プロセッサでは，再利用が見込めないループエントリを他のエントリと平等に扱っているため，このループエントリを入出力表から優先的に追い出していない．そのため，この再利用が見込めないループエントリによって入出力表を効率的に利用できていないという問題がある．

そこで，この問題を解決するために，再利用が見込めないループエントリを入出力表から強制的に追い出す手法を提案する．以降，この手法を**ループイタレーションパージ**と呼ぶ．しかし，再利用が見込めないループエントリを追い出しても，リングカウンタの更新頻度は変わらないため，TSID パージにより他のエントリが追い出される間隔は既存手法と変わらず，入出力表を効率的に利用できない．

そこで，入出力表が溢れた場合にのみ TSID パージを適用する手法をループイタレー

ションパージに組み合わせる．これにより，ループイタレーションパージで再利用が見込めないループエントリを追い出した分，他のエントリが長い間入出力表に登録されるようになり，入出力表を効率的に利用できる．また，入出力表が溢れた際に，性能低下を引き起こす原因となっていたFLIDパージを適用する必要がなくなる．以降，この入出力表が溢れた場合にのみTSIDパージを適用する手法を**on-demand TSIDパージ**と呼ぶ．ただし，リングカウンタを更新するタイミングは既存手法と同じであるため，1度のパージの適用で追い出されるエントリ数はTSIDパージと変わらない．

前述したとおり，ループイタレーションパージは，再利用される見込みのないループエントリを入出力表から強制的に追い出す手法であるが，ループエントリの中には入出力表に登録する時点で再利用が見込めない場合がある．これは，以下の3つの場合で起こると考えられる．1つ目は，メインコアがイタレーションの実行を終了し，そのループエントリを登録する場合である．この場合，入力が単調変化していくループにおいて，メインコアが実行したイタレーションのループエントリはエントリの登録時点で再利用が見込めないエントリである．2つ目は，2.2節の図7(b)のSpC3のように，メインコアのイタレーション実行開始よりも投機実行の終了が遅れてしまったSpCがループエントリを登録する場合である．この場合，SpCが登録するループエントリは，メインコアが既に実行を開始したイタレーションに対するものであるため，1つ目と同様に再利用が見込めないエントリである．また，ループの終了条件は事前に分からないので，繰り返し上限を超えたイタレーションをSpCが実行している場合がある．そのため，3つ目は，メインコアがループ区間の実行を抜けた後，当該命令区間を投機実行しているSpCがループエントリを登録する場合である．この場合，SpCはメインコアが今後実行することのないループのイタレーションを実行していることになるので，そのループエントリはエントリの登録時点で再利用が見込めないエントリである．そこで，このような3つの場合においてループエントリの入出力表への登録を中止する手法も併せて提案する．以降，この手法を**ループイタレーション登録フィルタ**と呼ぶ．

これらの提案手法により，再利用される見込みのないループエントリが入出力表から削除され，他のエントリが長い間入出力表に保持されるようになるため，入出力表を効率的に利用でき，再利用率が増加し高速化を期待できる．

3.2　提案手法の適用による効果

ループイタレーションパージと on-demand TSID パージ，そしてループイタレーション登録フィルタの 3 つの提案手法を適用することによる効果および影響について考察する．

関数再利用率の増加　提案手法では，再利用される見込みのないループエントリを入出力表から排除し，エントリを追い出すタイミングを入出力表が溢れた時に変更するため，既存手法よりも関数エントリが入出力表に長い間登録されるようになる．また，関数エントリの再利用間隔はさまざまであるため，提案手法により関数エントリの再利用される可能性が上昇することで，関数の再利用率が増加する．

検索オーバヘッドの減少　検索オーバヘッドとは，命令区間に再利用を適用できるかどうかを確かめるときに，MemoTbl を検索するオーバヘッドである．なお，自動メモ化プロセッサは，入出力表の各命令区間のエントリ数をカウントし，それを FLTbl の count フィールドに記憶しているため，入出力表に当該命令区間のエントリが存在しないときは入出力表を検索しない．さて，既存の自動メモ化プロセッサでは，メインコアは実行したイタレーションのループエントリを入出力表に登録する．これにより，SpC が登録したループエントリが入出力表に存在しない場合でも，MemoTbl にその命令区間のループエントリが存在するため，メインコアは MemoTbl を検索する．しかし，入力が単調変化するループでは，同一の入力で同イタレーションが再度実行される可能性は非常に低いため，この検索は無駄である．一方，提案手法ではメインコアはループエントリの入出力表への登録を中止しており，またメインコアが実行したイタレーションに関するループエントリを追い出している．これにより，SpC が登録したループエントリが入出力表に存在しない場合は，入出力表にそのループに対するエントリが存在しない．そのため，この場合は当該命令区間の count フィールドの値が 0 となり，入出力表を検索しなくなることで，検索オーバヘッドが削減される．

ループ再利用率の低下　既存の自動メモ化プロセッサでは，入力値予測が外れるイタレーションでも過去のイタレーションと入力が全く同じになる場合においては再利用に成功する．しかし，提案手法ではメインコアはループエントリの登録を中止し，また実行を終了したイタレーションに対するエントリを追い出すため，このような場合において再利用の機会を失う．ただし，このようなことは一般に入力が単調に変化するループではほとんど起こらないと考えられる．そのため，提案手法ではループ再利用率が低下する可能性があるが，その低下率は僅かであり自動メモ化プロセッサの性能に与える影響は小さい．

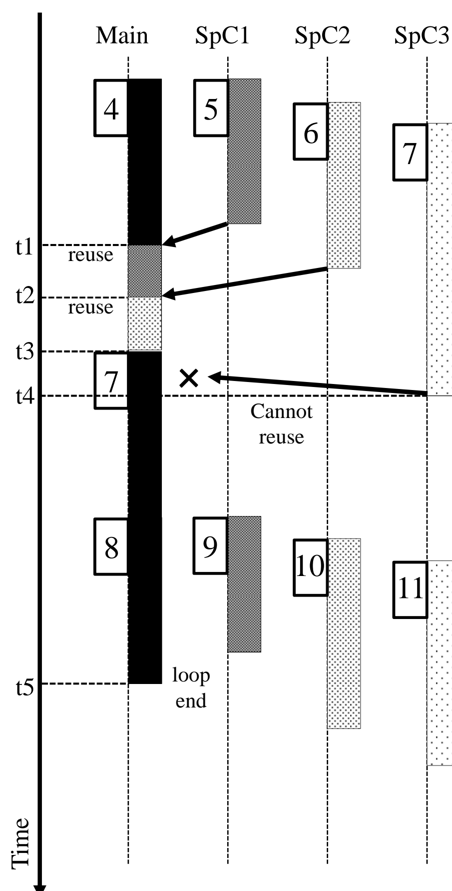

図 8: 並列事前実行の流れ

3.3 動作モデル

本節では，提案手法によるループエントリの登録中止や追い出しの様子を，並列事前実行のタイミングチャートを示す**図 8** を用いて説明する．なお，図 8 において，SpC は 3 基とし，入力値 8 に対応するイタレーションがループの最後のイタレーションであるとする．まずメインコアは入力値 4 に対応するイタレーションの実行を開始し，SpC もそれぞれ入力値 5，6，7 に対応するイタレーションの投機実行を開始する．そしてメインコアは時刻 t1 において，入力値 4 に対応するイタレーションの実行を終了する．このとき，メインコアが実行していたイタレーションのループエントリは，今後再利用される見込みが無いため，入出力表への登録を中止する．その後，メインコアは並列事前実行により登録されていたエントリを再利用することで入力値 5 に対応するイタレーションの実行を省略し，時刻 t2 において，この再利用されたループエントリは今後再利用される見込みがないため，入出力表から削除する．そして，メインコアは

同様に入力値 6 に対応するイタレーションの実行を省略し，時刻 t3 において，再利用したループエントリを削除する．その後，入力値 7 に対応するイタレーションのループエントリが入出力表に存在しないため，メインコアは再利用を適用できず，そのイタレーションの実行を開始する．一方同時刻において，SpC3 は入力値 7 に対応するイタレーションを投機実行中であり，時刻 t4 においてそのイタレーションの投機実行を終了する．このとき，既にメインコアは入力値 7 に対応するイタレーションの実行を開始しているため，そのイタレーションに対するループエントリは今後再利用される見込みがない．そこで，SpC3 はループエントリを入出力表に登録することを中止する．その後実行が進み，メインコアが時刻 t5 において，このループ区間の最後のイタレーションである入力値 8 に対応するイタレーションの実行を終了する．このとき，当該命令区間に対するループエントリは，メインコアが今後実行することのないループに対するエントリであり，再利用が見込めない．そこで，メインコアがループ区間の実行を終了するとき，SpC1 が登録した入力値 9 に対応するイタレーションのエントリを削除する．そして，メインコアがループ区間の実行を終了した後，入力値 10，11 に対応するイタレーションの投機実行を終了する SpC2 と SpC3 はループエントリの登録を中止する．

4　実装

本章では提案手法の具体的な実装方法について説明する．

4.1　ハードウェア拡張

提案手法を実装するために拡張した MemoTbl を**図 9** に示す．初めに，ループイタレーションパージを実現するためのハードウェア拡張について説明する．これを実現するためには，ループエントリとイタレーションを対応づける必要がある．なお，SpC によるループエントリ登録時に，実行したイタレーションはメインコアが実行しているイタレーションから何個先のイタレーションであるのかを Pred_dist から知ることができる．そこで，各イタレーションのループエントリに，他のイタレーションと区別するための固有の数値を割り当てる．そのため，メインコアが実行しているイタレーションを基点として，そこから数個先までのイタレーションをそれぞれ区別するための値を記憶しておくフィールド（liid_list）を FLTbl に設ける．以降，このイタレーションを区別する liid_list の値のことを**イタレーション ID** と呼ぶ．この liid_list のフィールド数を Pred_dist と同じにすることで，Pred_dist のコア番号から特定できるイタレー

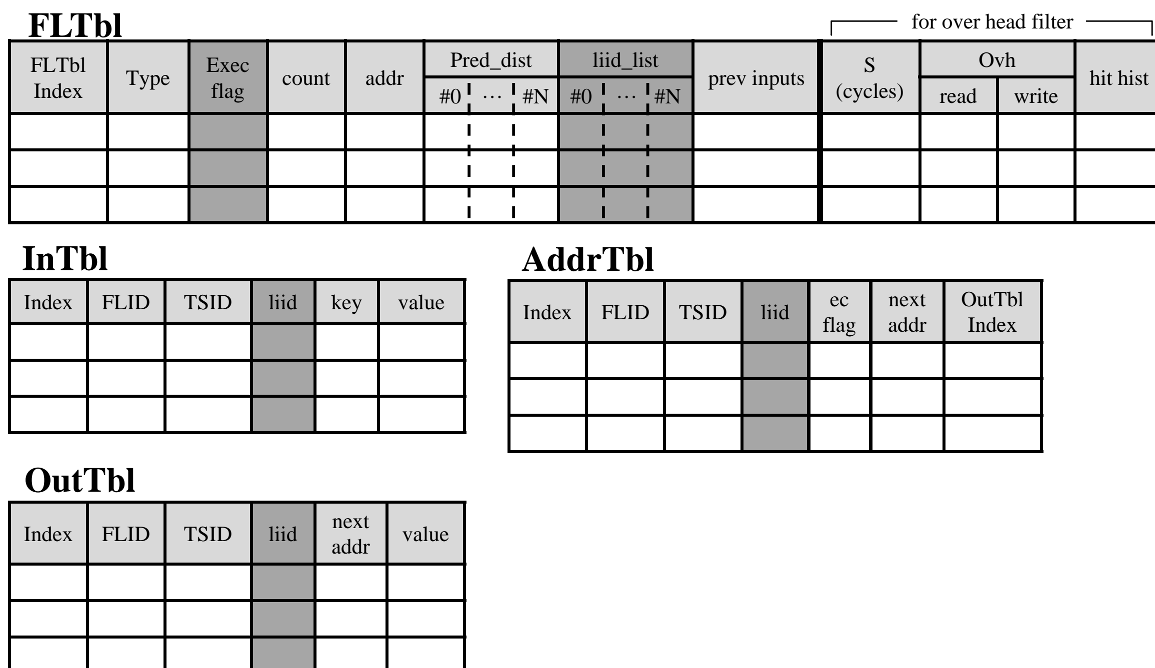

図 9: 拡張後の MemoTbl

ション全てをイタレーション ID で区別することができる．そして，入出力表にこのイタレーション ID を記憶するフィールド（liid）を追加することによりループエントリとイタレーションを対応づけることができる．なお，既存の自動メモ化プロセッサでは，SpC 台数 n に対し，$2(n+1)$ 個のイタレーションに対する実行担当コア番号を記憶する Pred_dist を用いることで，実行中の入力より $2(n+1)$ ストライド先の入力までを予測し，メインコアおよび各 SpC に割り当てている．そのため，この liid_list も $2(n+1)$ 個のイタレーション ID を記憶する必要がある．また，Pred_dist は左端をメインコアが実行しているイタレーションとし，メインコアがイタレーションの実行を終了したとき，および再利用に成功し実行を省略したときに，左に 1 つシフトする．このとき，右端にはまだどのコアにも割り当てられていないことを表す数値を記憶する．そのため，liid_list も同様に左端をメインコアが実行しているイタレーションとし，実行を終了した時点で左に 1 つシフトする．そして，右端には左端の追い出した値をセットするようにすることで，イタレーション ID をできる限り少ないビット数で表現することができる．

次に，メインコアがループ区間の実行を終了した後，当該命令区間を実行している SpC にループエントリの登録を中止させることを実現するためのハードウェア拡張について説明する．これを実現するためには，メインコアがループ区間の実行を終了し

ているか否かを判断するフラグが各ループごとに必要である．そこで，FLTbl に 1 ビットのフラグ（Exec flag）を追加し，メインコアがループを実行中にはその命令区間に対応するフラグをセットし，そのループ区間の実行を終了したらそのフラグをクリアする．

最後に，on-demand TSID 法を実現するためのハードウェア拡張について説明する．on-demand TSID 法は，入出力表が溢れるまでエントリを追い出さず，入出力表が溢れた際に最も古いタイムスタンプを持っているエントリを追い出す．これを実現するためには入出力表が溢れるまで，リングカウンタによって時刻管理されているタイムスタンプが枯渇しないことが必要である．既存手法では，リングカウンタの更新後の値と同一のタイムスタンプを持つエントリを MemoTbl から追い出すため，タイムスタンプが枯渇することはなかった．しかし，もしこのまま入出力表が溢れるまでエントリを追い出さないと，リングカウンタの全ての値を使用しても入出力表が溢れず，タイムスタンプが枯渇してしまう場合がある．なお，1 つのタイムスタンプで表現できるエントリ数は既存手法と同じとしているため，この数と入出力表の容量から，入出力表が溢れるまでタイムスタンプが枯渇しないリングカウンタの大きさが計算できる．そこで，従来のリングカウンタをその大きさに拡張する．

4.2　動作モデル

本節では，提案手法におけるループエントリの登録時の動作とループエントリの追い出し時の動作を説明する．

4.2.1　登録時の動作

提案手法では，入出力表へのループエントリ登録時にそのエントリの再利用が今後見込めるかどうかを判断し，再利用が見込めないのであれば登録を中止する．まず最初に再利用が見込めない場合の動作について説明する．3.1 節で述べたように，提案手法では 3 つの場合でループエントリの登録を中止している．

再利用が見込めない場合　メインコアはループエントリの登録を中止するために，命令区間の実行終了時に，FLTbl の Type フィールドを調べる．そして，Type フィールドにループを示す値が記憶されているのであれば，メインコアはそのループエントリの登録を中止する．

次に，メインコアがイタレーションの実行を開始するよりも，当該イタレーションの投機実行の終了が遅れてしまった SpC が，ループエントリの登録を中止する時の動作について説明する．SpC はイタレーションの投機実行を終了した時に，Pred_dist を

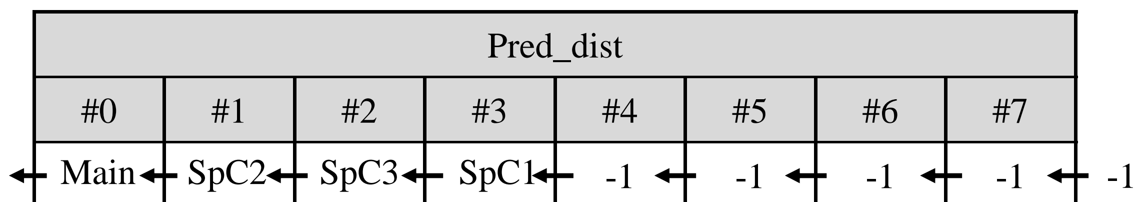

図 10: SpC を 3 基とした場合の Pred_dist の構成

確認することでループエントリの登録を行うか否か判断する．SpC を 3 基とした場合の Pred_dist の詳細な構成を**図 10** に示す．図 10 において，Pred_dist には各コアのコア番号が記憶されているが，−1 はまだどのコアも実行していないことを表している．そして，Pred_dist の#0 はメインコアが実行しているイタレーションを表し，メインコアがイタレーションの実行を終了したとき，および再利用に成功し実行を省略したときに Pred_dist の値を 1 つ左にシフトする．これにより，Pred_dist の#N にはメインコアが実行しているイタレーションより N 個先のイタレーションを実行中のコア番号が記憶される．なお，#0 はメインコアが実行しているイタレーションを表しているため，メインコアがイタレーションの実行を開始した時点で#0 をメインコアのコア番号で上書きする．例えば，図 10 の場合，メインコアが実行しているイタレーションより 1 個先のイタレーションを実行対象とするコアの番号は#1 に記憶されており，それは SpC2 のコア番号である．この後，メインコアがイタレーションの実行を終了すると，#0 には SpC2 のコア番号が記憶され，メインコアは SpC2 が実行対象としていたイタレーションの実行を開始すると，#0 の値をメインコアのコア番号で上書きする．そこで，SpC はループエントリ登録時に Pred_dist に自身のコア番号が記憶されていない場合，自身が投機実行したイタレーションをメインコアが実行している，もしくは実行を終了していると判断し，このループエントリの登録を中止する．

次に，メインコアがループ区間の実行を終了した後，当該命令区間を投機実行している SpC がループエントリの登録を中止するときの動作について説明する．メインコアはループ区間の実行を開始するとき，その命令区間に対応する Exec フラグをセットし，そしてループ区間の実行を終了するとき，その命令区間に対応する Exec フラグをクリアする．そのため，SpC はループエントリ登録時に当該命令区間に対する Exec フラグがクリアされていれば，メインコアは既にそのループを実行を終了したと判断し，ループエントリの登録を中止する．ここで，並列事前実行のタイミングチャートを示す**図 11** を用いてこの動作を説明する．なお，この図において SpC は 3 基とし，入力

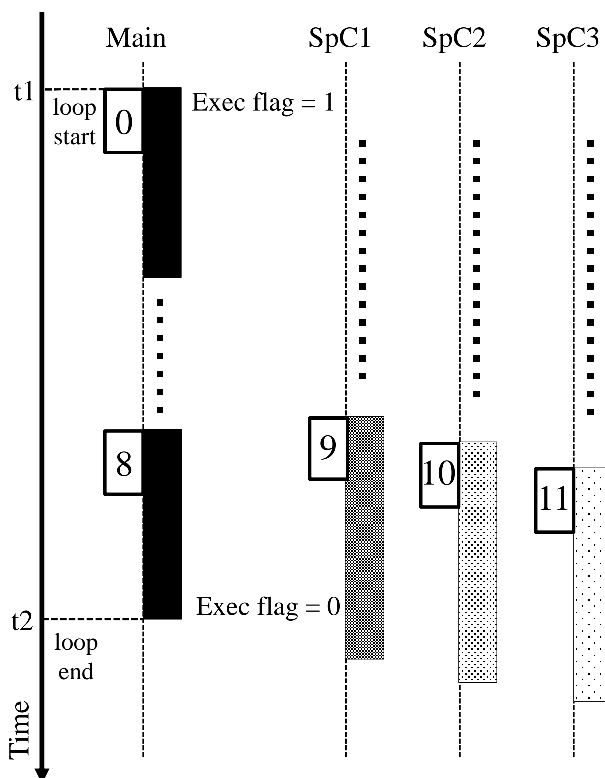

図 11: 並列事前実行の流れ

値 0 に対応するイタレーションがループの最初のイタレーションであり，入力値 8 に対応するイタレーションがループの最後のイタレーションであるとする．まず時刻 t1 にて，メインコアがループ区間の実行を開始する．このとき，当該命令区間に対応する Exec フラグをセットする．このフラグがセットされている間，SpC はメインコアがそのループ区間を実行していると分かる．この後実行が進み，時刻 t2 にて，メインコアが入力値 9 に対応するイタレーションの実行を終了し，このループ区間の実行を終え，当該命令区間に対応する Exec フラグをクリアする．その後，各 SpC が入力値 9, 10，11 に対応するイタレーションの実行を終了し，そのループエントリを登録しようとするが，当該命令区間の Exec フラグがクリアされているため，これらのエントリの登録を中止する．

再利用が見込める場合　次に，ループエントリの登録時にそのエントリでの再利用が，今後見込める場合の動作について説明する．この場合では，SpC はイタレーションの実行を終了した後，ループエントリを登録する時に liid_list からそのイタレーション ID を取得する．ここで，SpC を 3 基とした場合の liid_list の詳細な構成を**図 12** に示す．この図において，liid_list の値は各イタレーションに対する固有のイタレーション

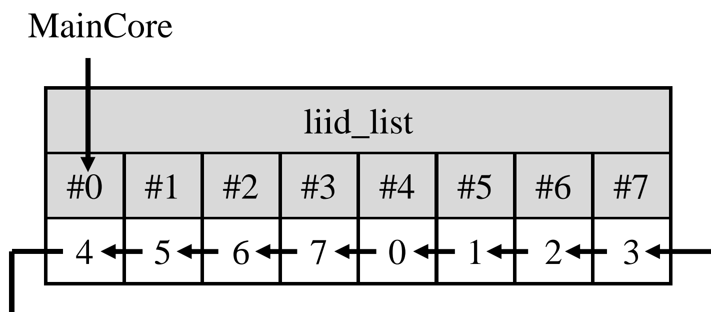

図 12: SpC を 3 基とした場合の liid_list の構成

ID を表している．そして，liid_list では Pred_dist と同様に，#0 はメインコアが実行しているイタレーションを表す．また，メインコアのイタレーションの実行終了時に liid_list の値を 1 つ左にシフトし，#0 から追い出される値を#7 に記憶する．これにより，liid_list の#N にはメインコアが実行しているイタレーションより N 個先のイタレーションに対するイタレーション ID が記憶される．そこで，まず SpC がループエントリを登録する際，実行したイタレーションはメインコアが実行しているイタレーションより何個先のイタレーションであるのかを Pred_dist から調べる．そして，そのイタレーションに対するイタレーション ID を liid_list から取得する．例えば，SpC3 がループエントリを登録するときに，Pred_dist と liid_list が図 10 と図 12 の状態であった場合，まず，Pred_dist から SpC3 のコア番号が記憶されているフィールドを検索する．そして，そのフィールドが#2 であると分かるため，liid_list の#2 からそのイタレーション ID として 6 を取得する．その後，入出力表に登録するループエントリの liid フィールドに 6 を記憶する．なお，関数エントリの liid フィールドにはループエントリではないことを示す値として −1 を記憶する．

さて，自動メモ化プロセッサでは入力エントリを木構造で表現しているため，2.1 節の図 4 のようにエントリを共有する場合がある．また，ループイタレーションパージはメインコアが実行を終了したイタレーションのイタレーション ID と，入出力表のループエントリの liid フィールドの値を比較し，一致しているのであれば，そのループエントリを追い出す．そのため，共有している入力エントリを追い出してしまうことで，ある入力エントリの親エントリがなくなり，その木構造が崩れる可能性がある．そこで，共有している入力エントリのイタレーション ID を，ループイタレーションパージによって追い出される順番が最も遅い入力エントリのイタレーション ID に更新することで，この問題を解決する．具体的には，ループエントリ登録時に，登録す

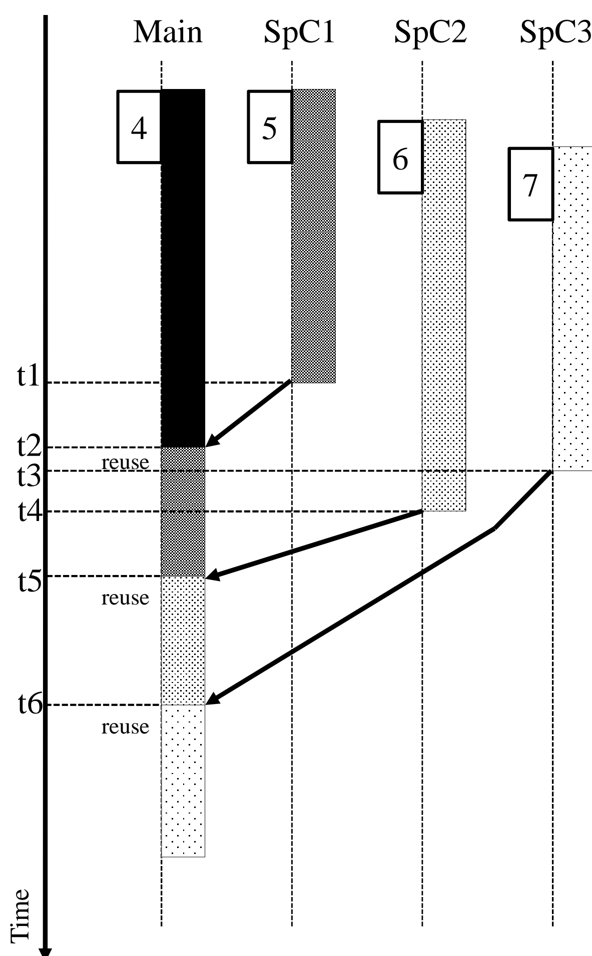

図 13: 並列事前実行の流れ

るループエントリのイタレーション ID が記憶されている liid_list のフィールドと，共有している親エントリのイタレーション ID が記憶されている liid_list のフィールドを比較する．このとき，そのフィールドがより右側にあるほど，ループイタレーションパージによって追い出される順番が遅い．そのため，その右側にあるイタレーション ID を liid フィールドの値とする．

ここで，並列事前実行のタイミングチャートを示す**図 13** を用いて，ループエントリを登録するときの動作例を説明する．なお，SpC は 3 基とし，このときの Pred_dist と liid_list を**図 14** に示す． 図 14 の (A) は時刻 t1 における Pred_dist と liid_list の値を示し，(B) は時刻 t2，時刻 t3 および t4 における値を示している．まずメインコアは入力値 4 に対応するイタレーションの実行を開始し，SpC もそれぞれ入力値 5，6，7 に対応するイタレーションの投機実行を開始する．そして，SpC1 が時刻 t1 にて入力値 5 に対応するイタレーションの実行を終了する．このとき，SpC1 は Pred_dist を自身

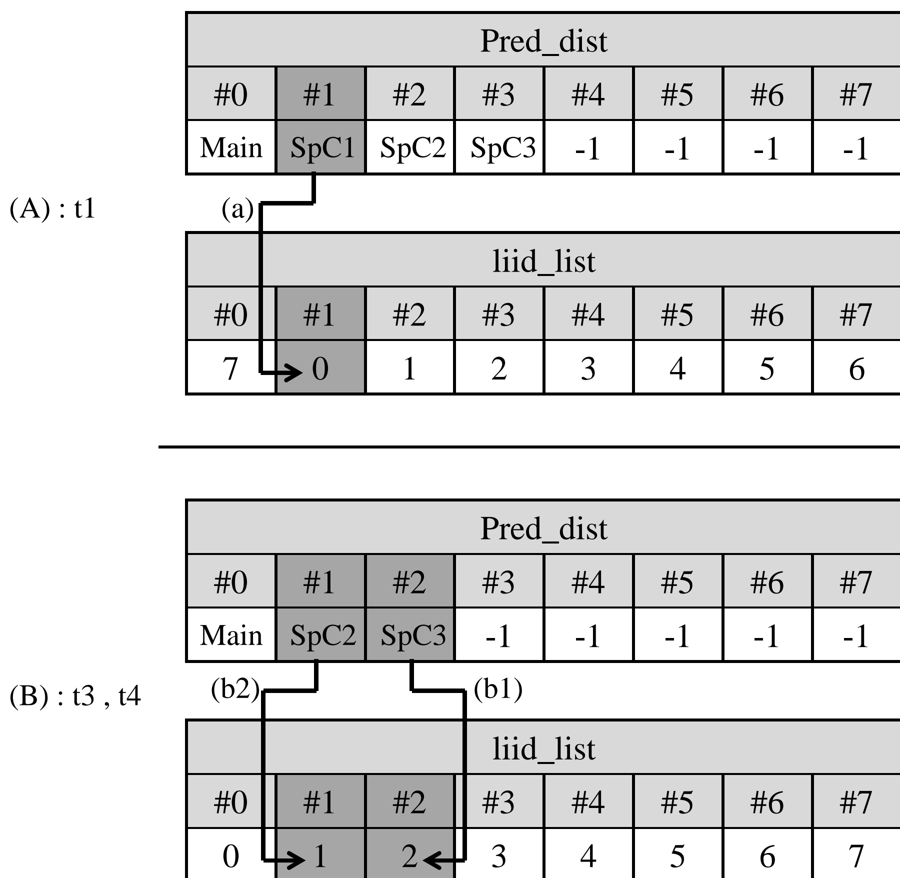

図 14: Pred_dist と liid_list の値

のコア番号で検索し，そのフィールドに対応する liid_list のイタレーション ID として 0 を取得する (a)．そして，登録するループエントリの liid フィールドに 0 を記憶する．次に，時刻 t2 において，メインコアが入力値 4 のイタレーションの実行を終了する．このとき，図 14(A) に示す Pred_dist と liid_list が更新され，(B) の状態となる．次に，SpC3 が時刻 t3 において，入力値 7 に対応するイタレーションの実行を終了する．このとき，SpC3 は SpC1 のイタレーションの実行終了時の動作と同様にイタレーション ID として 2 を取得し (b1)，登録するループエントリの liid フィールドに 2 を記憶する．ここで，もし SpC3 が登録する入力エントリを共有する入力セットが MemoTbl に存在する場合，その liid フィールドの値は 0 である．そのため，SpC3 が登録するループエントリのイタレーション ID のほうが，ループイタレーションパージによって追い出される順番が遅いため，その共有するエントリの liid フィールドの値を 2 に更新する．次に，時刻 t3 にて SpC2 が入力値 6 に対応するイタレーションの実行を終了する．このとき，SpC1，SpC3 のイタレーションの実行終了時の動作と同様にイタレーション

(C) : t5

liid_list							
#0	#1	#2	#3	#4	#5	#6	#7
1	2	3	4	5	6	7	0

(D) : t6

liid_list							
#0	#1	#2	#3	#4	#5	#6	#7
2	3	4	5	6	7	0	1

図 15: liid_list の値

ID として 1 を取得し (b2)，登録するループエントリの liid フィールドに 1 を記憶する．ここで，もし SpC2 が登録する入力エントリを共有する入力セットが MemoTbl に存在し，その liid フィールドの値が 0 であれば，登録するループエントリのイタレーション ID である 1 のほうがループイタレーションパージによって追い出される順番が遅いため，その liid フィールドの値を 1 に更新する．一方で，liid フィールドの値が 2 であれば，登録するループエントリのイタレーション ID である 1 のほうがループイタレーションパージに追い出される順番が早いため，liid フィールドの値は更新しない．

4.2.2 追い出し時の動作

提案手法では，メインコアが再利用によってイタレーションの実行を省略した時，および再利用を適用できずに実行を終了した時に，そのイタレーションに関するループエントリを入出力表から追い出す．これを実現するために，メインコアが実行しているイタレーションを表す liid_list の#0 からイタレーション ID を取得する．そして，そのイタレーション ID と入出力表の liid フィールドが記憶しているイタレーション ID が同一であるようなループエントリを入出力表から追い出す．

ここで，前項で示した図 13 と図 14 を用いてループエントリ追い出し時の動作例を説明する．なお，図 13 における時刻 t5，および時刻 t6 での更新後の liid_list を**図 15** の (C)，および (D) に示す．まずメインコアが時刻 t2 において，入力値 4 に対応するイタレーションの実行を終了する．その後，メインコアは並列事前実行により登録されているエントリを再利用することで入力値 5 に対応するイタレーションの実行を省略する．このとき，時刻 t5 において，メインコアはループイタレーションパージのために図 14(B) に示す liid_list の#0 からイタレーション ID として 0 を取得する．そして，入出力表のループエントリの liid フィールドに 0 を持つ SpC1 が登録したエントリ

を追い出す．このとき，(B) に示す liid_list が更新され，図 15(C) の状態となる．その後，メインコアが入力値 6 に対応するイタレーションの実行を省略し，時刻 t6 において同様に (C) に示す liid_list の#0 からイタレーション ID として 1 を取得する．そして，入出力表のループエントリの liid フィールドに 1 を持つエントリを追い出す．このとき，(C) に示す liid_list が更新され，(D) の状態となる．その後，メインコアが入力値 7 に対応するイタレーションの実行を省略し，(D) に示す liid_list の#0 からイタレーション ID として 2 を取得する．そして，入出力表のループエントリの liid フィールドに 2 を持つエントリを追い出す．このような動作により，他のエントリが長い間入出力表に保持されるようになるため，入出力表を効率的に利用できる．

5　評価

以上で述べた拡張を既存の自動メモ化プロセッサシミュレータに対して実装した．また，提案手法の有効性を示すために，ベンチマークプログラムを用いてサイクルベースシミュレーションによる評価を行った．

5.1　評価環境

評価には，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．評価に用いたパラメータを**表 1** に示す．なお，全てのモデルはメインコア 1 つに，3 つの投機実行コアを加えた合計 4 コア構成とし，キャッシュや命令レイテンシは SPARC64-Ⅲ[8] を参考とした．また，MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288[9] を参考にし，サイズは 32Bytes 幅 ×4K 行の 128KBytes とした．

5.2　評価結果

提案手法の有効性を確かめるため，汎用ベンチマークプログラムである SPEC CPU95 を用いて実行サイクル数を評価した．この結果を**図 16** に示す．各ベンチマークプログラムの結果を 4 本のグラフで示しているが，それぞれは

（**N**）　メモ化を行わないモデル

（**M**）　従来モデル

（**O**）　on-demand TSID パージのみを適用したモデル

（**P**）　ループイタレーションパージ，ループイタレーションフィルタおよび on-demand TSID パージを適用した提案モデル

表 1: 評価環境

MemoBuf	64 kBytes
MemoTbl CAM	128 kBytes
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 cycles/32Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32Bytes
D1 cache	32 KBytes
line size	32 Bytes
ways	4 ways
latency	2 cycles
miss penalty	10 cycles
D2 cache	2 MBytes
line size	32 Bytes
ways	4 ways
latency	10 cycles
miss penalty	100 cycles
Register windows	4 sets
miss penalty	20 cycles/set

が要した総実行サイクル数を表しており，メモ化なしモデル（N）を 1 として正規化している．また凡例はサイクル数の内訳を示しており，exec は命令サイクル数，read はレジスタやキャッシュと MemoTbl との比較に要したサイクル数（検索オーバヘッド），write 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（書き戻しオーバーヘッド），D$1 および D$2 は 1 次および 2 次データキャッシュミスにより要したサイクル数，window はレジスタウインドウミスペナルティのサイクル数である．

まず,（O）の結果より，従来モデル（M）と比べて 134.perl，125.turb3d において高速化していることが分かる．これは，on-demand TSID パージを適用することで，性能低下を引き起こす原因となっていた FLID パージを起こさなくしたためであると考えられる．そこで,（M）における入出力表が溢れた際の FLID パージの回数を**表 2** に示す．（M）において，この表に示す FLID パージ回数が抑制されたぶん，この FLID パージ

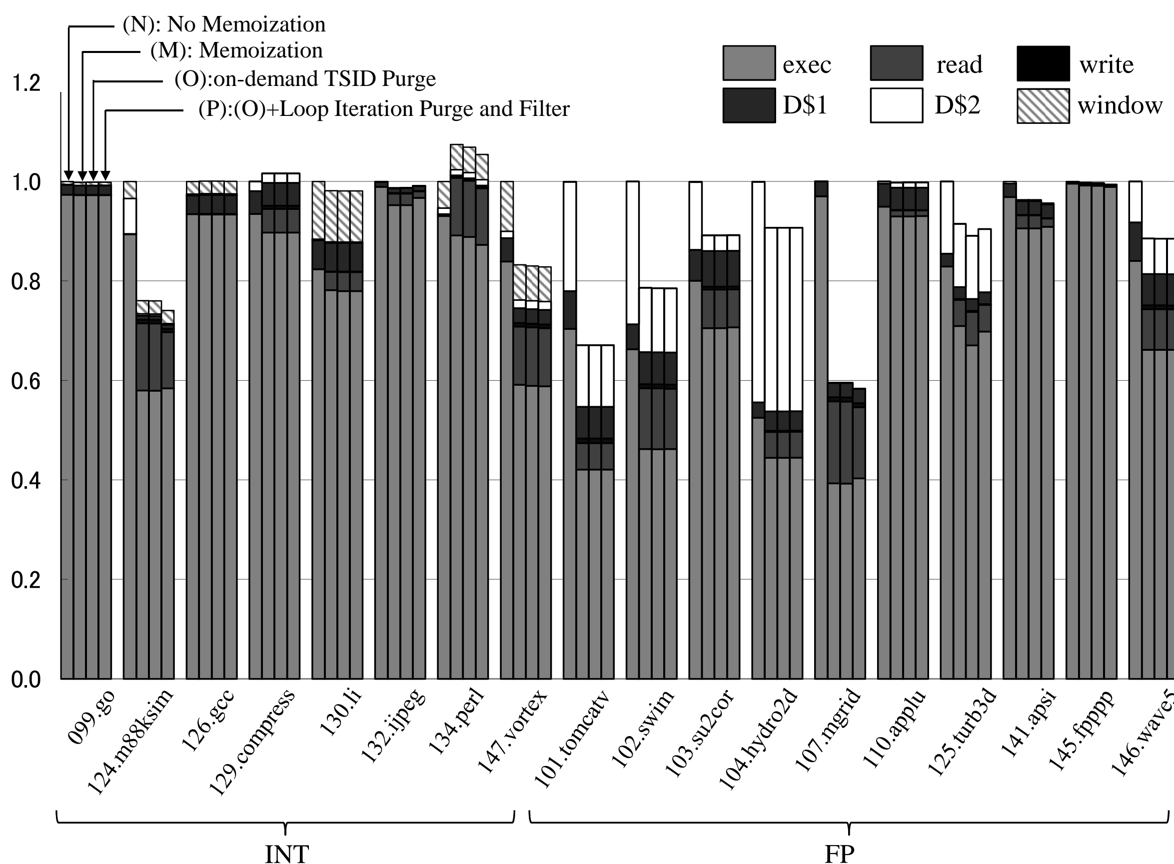

図 16: 実行サイクル数

表 2: 入出力表が溢れた際の FLID パージ回数

	134.perl	125.turb3d
（M）従来モデル	548	60992
（O）on-demand TSID パージモデル	0	0

で追い出されていた有益なエントリが（O）では入出力表に保持し続けられたことで，高速化したと考えられる．次に（P）と（O）を比較すると，124.m88ksim，134.perl，147.vortex，107.mgrid，141.apsi，145.fpppp の 6 つのプログラムにおいて高速化していることがわかる．特に，134.perl，145.fpppp においては，exec を削減できたことで高速化している．これは，ループエントリの登録中止と強制的な追い出しにより，入出力表を効率的に利用できるようになったため，関数エントリが再利用される機会が増え，（O）よりも関数の実行を省略することができたからである．また，124.m88ksim，147.vortex，107.mgrid，141.apsi では，検索オーバヘッドを削減できたことで高速化している．これは，ループ区間の実行時に（O）では入出力表に登録されていた当該

表 3: 124.m88ksim のループ区間（2d6e8〜2d7e4）における再利用成功回数

	再利用成功回数
（O）on-demand TSID パージモデル	26782
（P）提案モデル	0

命令区間のループエントリが，提案手法により入出力表から排除されていたからである．しかし，124.m88ksim，132.ijpeg，107.mgrid，125.turb3d，141.apsi では exec が増加してしまっている．これは,（O）では再利用を適用できていたループ区間に対して，提案手法では再利用を適用できなくなってしまったためであると考えられる．また，その他のプログラムではほとんど性能が変化しなかった．

結果をまとめると，SPEC CPU95 ではメモ化なし（N）に比べ，従来モデル（M）では最大 40.5 %，平均 9.1 %のサイクル数の削減であったのに対し，提案モデル（P）では，最大 41.8 %，平均 9.5 %のサイクル数の削減に成功した．

5.3　考察

本節では，exec が増加してしまった 124.m88ksim，132.ijpeg について考察する．
124.m88ksim は，アドレス 2d6e8〜2d7e4 のループ区間において（O）では頻繁に再利用されていたが,（P）では全く再利用されなかったことが exec の増加の原因であった．**表 3** にそのループ区間の再利用成功回数を示す．この表のように,（P）では，このループ区間での再利用成功回数が 0 になっていた．このループ区間において,（O）では SpC が並列事前実行し，ループエントリを入出力表に登録した後，メインコアがそのループエントリを再利用することで実行を省略していた．しかし（P）では，検索オーバヘッドが削減された分,（O）で再利用を適用できていたイタレーションの実行をメインコアが開始する時刻が（O）より早くなった．そのため，そのイタレーションにおいて，SpC のループエントリの登録がメインコアの実行開始に間に合わず,（P）はそのループ区間で再利用を適用できなかった．なお，並列事前実行の対象区間が再利用頻度が低いループ区間であると，自動メモ化プロセッサはそのループ区間の並列事前実行を停止する．これにより,（O）で再利用に成功していたループ区間が,（P）では並列事前実行の対象区間から外れ，再利用を適用できなくなってしまった結果,（O）よりも exec が増加したと考えられる．

132.ijpeg は，アドレス 2a9a0〜2ab14 のループ区間において，再利用成功回数が大きく減少したことが原因であった．**表 4** にそのループ区間の再利用成功回数を示す．

表 4: 132.ijepg のループ区間（2a9a0〜2ab14）における再利用成功回数

	再利用成功回数
（O）on-demand TSID パージモデル	246663
（P）提案モデル	101508

132.ijpeg は，124.m88ksim の exec 増加の原因と同様に，検索オーバヘッドが削減された分，SpC のループエントリ登録が間に合わず，再利用を適用できなくなったイタレーションがあった．しかし，SpC のループエントリ登録が間に合い，再利用に成功するイタレーションも存在したため，124.m88ksim のように並列事前実行の対象区間から外れ，再利用成功回数が 0 になることはなかった．そのため，この表のように再利用成功回数が（O）に比べ約 6 割ほど減るのみとなったが，これによる exec の増加が全体としての性能低下に繋がってしまった.

6　おわりに

本研究では，従来の自動メモ化プロセッサの更なる高速化手法として，再利用される見込みがないループエントリの入出力表への登録中止と入出力表からの強制的な追い出しを提案した．また，入出力表が溢れた場合にのみに TSID パージを適用する手法も併せて提案した．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した結果，メモ化無しで通常実行するのと比較して，従来モデルでは最大 40.5 %，平均 9.1 %のサイクル数の削減であったのに対し，提案手法では，最大 41.8 %，平均 9.5 %のサイクル数の削減となり，提案手法の有効性を確認できた.

今後の課題としては，まず，さらなる再利用表の効率的な利用を目指すことが挙げられる．本研究では，命令区間の特徴のみに着目したが，エントリの再利用回数やサイクル削減数等を考慮することで，さらなる高速化を図ることができると考えられる．また，命令レベル並列性に基づく高速化手法とメモ化とを組み合わせた手法を探っていくことも挙げられる.

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，斎藤彰一准教授，松井俊浩准教授，梶岡慎輔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，松井研究室の方々に深く感謝致します.

参考文献

[1] Shah, M., Golla, R., Grohoski, G., Jordan, P., Barreh, J., Brooks, J., Greenberg, M., Levinsky, G., Luttrell, M., Olson, C., Samoail, Z., Smittle, M. and Ziaja, T.: Sparc T4: A Dynamically Threaded Server-on-a-Chip, *IEEE Micro*, Vol. 32, No. 2, pp. 8–19 (2012).

[2] Conway, P. and Hughes, B.: The AMD Opteron Northbridge Architecture, *IEEE Micro*, Vol. 27, No. 2, pp. 10–21 (2007).

[3] Tilera Corporation: *TILE-Gx Processor Family Product Brief* (2009).

[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 International Symposium on High-Performance Computer Architecture*, pp. 106–114 (1999).

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

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

[9] MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).