修士論文

動的および静的プログラム解析を用いた自動メモ化プロセッサ制御手法

指導教員　　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学大学院工学研究科
修士課程情報工学専攻
平成 19 年度入学　19417564 番

島崎 裕介

平成 21 年 2 月 5 日

動的および静的プログラム解析を用いた自動メモ化プロセッサ制御手法

島崎 裕介

内容梗概

これまで、命令レベル並列性やスレッドレベル並列性に基づく様々な高速化手法が提案されてきたが、これらの手法による高速化には限界が見えてきている．一方で，これらの高速化手法とは異なるアプローチとして，計算再利用を用いた高速化手法にメモ化があり，この計算再利用技術に基づく自動メモ化プロセッサが提案されている．

本稿では，自動メモ化プロセッサに電力評価モジュールを実装し，シミュレータレベルでの消費エネルギー評価を行なった．また，メモ化により高速化を図ることは可能だが，プログラムによってはメモ化の効果が現れず，メモ化機構追加分のエネルギーを余分に消費する場合がある．そこで新たに，計算再利用率を動的に解析しこれに応じてメモ化の中断や再開を行わせることで，本来メモ化の効果が得られる時間のみメモ化機構を動作させるモデルを提案・実装した．

まず，メモ化機構を停止させたことにより削減サイクル数の減少によるによる性能低下が予想されたが大きな変化とはならなかった．一方，従来 SPEC CPU95 では平均 +14.6%の消費エネルギーの増加，GA プログラムでは平均 −0.2%とわずかな消費エネルギーの削減が起きていたが，提案手法により SPEC CPU95 では平均 +8.2%，GA プログラムでは平均 −5.1%となり，かなりの消費エネルギーの抑制および削減をすることができた．この結果から，メモ化の中断および再開により，高速性をあまり損なうことなく低消費エネルギー化を実現できることが分かった．

また，自動メモ化プロセッサには前述した消費エネルギー増大のコストだけではなく，メモ化を行う際に発生するオーバーヘッドにより高速化が妨げられるという問題があった．プログラムによってはメモ化を行うことで実行時間が増加してしまう場合がある．そこで，本稿では前述した動的解析とはまた別に，実行すべき対象のプログラムを静的解析し，メモ化の効果が得られないと判断された関数に対してメモ化を行わないようにするモデルを提案・実装した．

SPEC CINT 95 による評価では，すべての関数をメモ化の対象としていた従来手法では平均実行サイクル数が 9.7%減，平均消費エネルギーは 10.8%増であったが，静的解析を用いていくつかの関数をメモ化対象から除外した結果，平均実行サイクル数 9.9%減，消費エネルギーは 10.2%増となる場合があり，オーバーヘッド削減によるわずかな高速化と消費電力の削減を実現できることが分かった．

動的および静的プログラム解析を用いた自動メモ化プロセッサ制御手法

目次

1　はじめに

これまで，SIMD やスーパースカラなどの命令レベル並列性（ILP：Instructure-Level Parallelism）を利用した高速化手法や，マルチコアなどスレッドレベル並列性（TLP：Thread-Level Parallelism）を利用した高速化手法が注目されてきた．しかし，一般に，プログラム中から明示的な ILP や TLP の存在する部分を抽出することは難しく，またプログラム中に存在する ILP や TLP にも限りがあることからこれらの手法による高速化にも限界がある．

一方で，これら従来の手法とは異なる高速化手法としてプログラム中に存在する値の局所性を利用した**メモ化（Memoization）**[1] という手法が存在し，このメモ化による動作をモデル化した自動メモ化プロセッサが提案されている [2]．メモ化とは，計算量の多い関数に対しその入出力を記憶しておくことで，同関数の同一入力による再計算を省略し高速化する手法である．しかし高速化の反面，その入力参照などによるオーバーヘッドや，入出力の記憶機構などによるエネルギーコストには無駄があり，改良の余地がある．

過去の研究では，このメモ化が消費エネルギー削減にも寄与するだろうという予測も立てられている [3] がこれまで厳密な調査が行われたわけではなかった．そこで本稿では，自動メモ化プロセッサに対する消費電力見積もりを可能とし，その上でメモ化機構への供給電力制限によってプロセッサの低消費エネルギー化を目指した制御方法を提案する．それに加えて，メモ化機構へのアクセスにより発生するオーバーヘッドの削減およびメモ化機構のより効率的な利用方法を提案し，自動メモ化プロセッサのさらなる高速化を目指す．

以下，2 章ではメモ化による動作を厳密にモデル化した自動メモ化プロセッサについて述べる．また本稿の課題とする，メモ化の際に必要なコストについても説明する．3 章では，自動メモ化プロセッサに対する電力見積もり機能の実装方法を説明した上で，同プロセッサの消費エネルギー抑制のために提案した動的プログラム解析手法について説明する．一方 4 章では，3 章の動的解析とは異なり，実行する前のプログラムに対して解析を行う．自動メモ化プロセッサにおけるメモ化機構のより効率的な使用を目指した静的プログラム解析手法について説明する．

5 章では，電力測定機能を追加した自動メモ化プロセッサに対して測定シミュレーションを行なった．3 章で説明する動的解析を用いたエネルギー制御方法の提案および 4 章で説明する静的解析を用いたメモ化機構の効率的な使用方法の提案に対する評

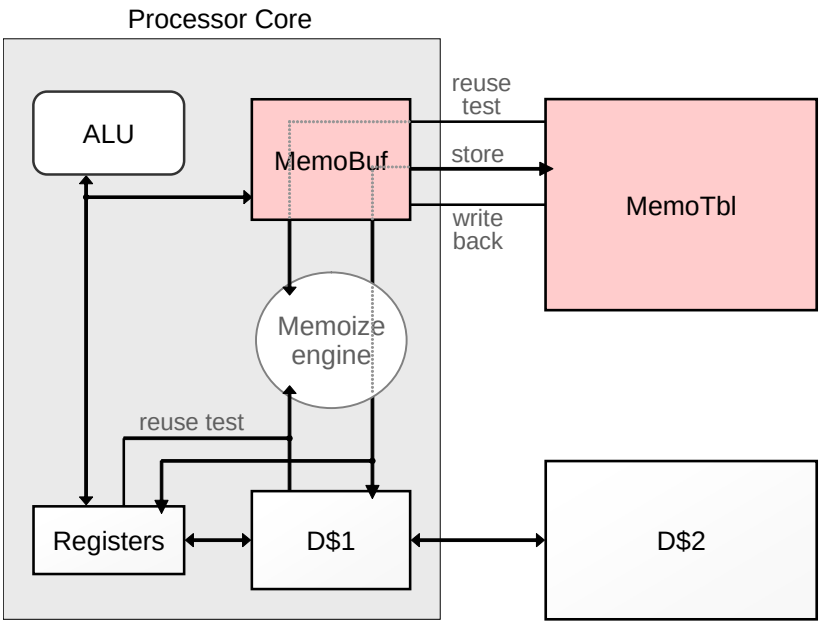

図 1: 自動メモ化プロセッサの構造

価とその考察を行う．最後に 6 章で本稿をまとめる．

2　メモ化と自動メモプロセッサ

ILP を利用したパイプライン処理における高速化手法や TLP を利用した並列計算などの高速化手法が研究されつつも，これら技術を用いた高速化手法には限界が来ている．一方，それらとは異なったアプローチとしてメモ化がある．メモ化とは，関数の入出力セットを配列などに記憶させることで，当該関数の同一入力による実行を省略する高速化手法である．

本章では，このメモ化を行うプロセッサモデルとして提案する自動メモ化プロセッサを取り上げ，その動作やアーキテクチャについて説明する．

2.1　自動メモ化プロセッサの動作モデル

本稿で取り扱う自動メモ化プロセッサは，プログラムの実行中に `call` 命令から `return` 命令までの命令区間を動的に検出し，これを関数として自動的にメモ化する．

図 1 にプロセッサ構成の概略を示す．自動メモ化プロセッサは，入出力を記憶するためのテーブル（以降，**MemoTbl**）および，MemoTbl への書き込みバッファ（以降，**MemoBuf**）を持つ．

MemoBuf および MemoTbl によるメモ化機構の動作を例 2-1 のようなプログラムを

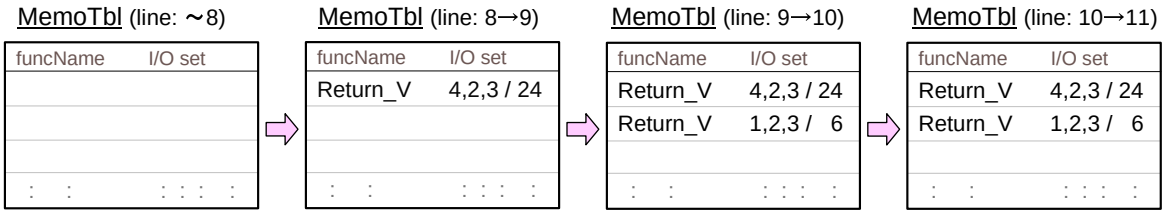

図 2: MemoTbl 内の登録状況

用いて説明する．また，プログラム実行中における MemoTbl のエントリ登録状況を**図 2** に示す.

例 2-1：自動メモ化プロセッサの動作説明用プログラム

```
1)  int Return_V(int L, int W, int H){  // 体積を求める関数
2)        int v;
3)        v = L * W * H;
4)        return ( v );
5)  }
6)  void main(void){
7)        int a=0, b=0, c=0;
8)        a = Return_V( 4, 2, 3 );  // 1度目の呼び出し
9)        b = Return_V( 1, 2, 3 );  // 2度目の呼び出し
10)       c = Return_V( 4, 2, 3 );  // 3度目の呼び出し
11) }
```

例 2-1 のプログラムに対してメモ化を適用した場合，関数 Return_V がメモ化の対象となる命令区間となる．具体的には，Return_V に対する関数呼び出し命令から Return_V 中の関数復帰命令までの間が命令区間として検出される．MemoTbl に記憶されている当該区間の入力アドレスすべてに対応する値をレジスタおよびキャッシュから読み出し，MemoTbl 内に登録された入力値との比較を始める．たとえば 2 度目の関数呼び出しでは，関数 Return_V を呼び出すための引数である 1，2，3 がレジスタもしくはキャッシュ上に存在し，これを読み出して MemoTbl 内に登録された入力値との比較を始める．

1 度目の関数呼び出しの開始時点や 2 度目の関数呼び出しの開始時点のように，現在呼び出している関数そのものおよび関数入力の一致するエントリが MemoTbl 上に存在しなかった場合は通常どおり命令区間を実行していく．その際レジスタおよびキャッ

シュに対する参照を入力とし，それらに対する書き込みを出力として MemoBuf へと一時的に記録していく．その後命令区間の終了を検出すると，MemoBuf 内に蓄積された入出力セットを MemoTbl へと保存する．3 度目の関数呼び出しの時点で，はじめて関数 Return_V とその入力がすべて一致するエントリが MemoTbl 上に存在する．この時に MemoTbl 上の出力が取り出され，Return_V の実行を省略してそのまま変数 c へと代入される．以上で述べたように，過去の結果を利用して同一入力による処理を省略することを**計算再利用**と呼ぶ.

また，関数の入力として扱われるものには，変数 L，W，H などの関数の引数だけでなく関数内で参照された大域変数があり，同様に書き込みが行われた変数は出力として扱われる．ただし関数 Return_V 中の変数 x のように，局所変数は登録する入出力セットからは除外される．自動メモ化プロセッサは SPARC ABI[4] に沿って記述されたプログラムを対象としており，局所変数に関しては OS がデータサイズおよびスタックサイズの上限を決定している．そのため，この上限および関数呼び出しが行われる直前のスタックポインタの値と変数アドレスとの関係から，局所変数かどうかの判別が可能となる．

2.2 自動メモ化プロセッサの構成

前節で述べた MemoBuf および MemoTbl を構成する要素について説明する．一般に関数内では，ある入力の値により後続の入力アドレスが変化する．以下にその例を示す.

例 2-2：後続入力パターンの変化

```
int b = 3,
int c = 5;
int Func(int a){
      if( a > 0 ) return (a+b); // 変数 a,b が参照される（パターン B）
      else        return (a+c); // 変数 a,c が参照される（パターン C）
}
```

例 2-2 は，関数 Func の引き数 a の値が正であった場合に後続の入力が b（パターン B）となり，そうでなかった場合には後続する入力が c（パターン C）となる場合を表わしている.

このように，一般にある関数の入力として扱うべき入力列は，その入力となる値（お

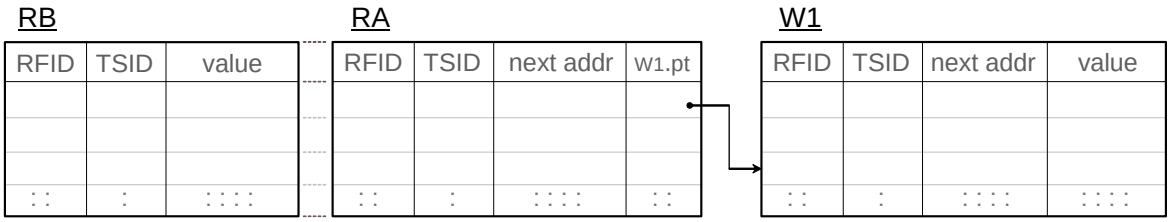

図 3: RB，RA，W1 モデル

よびアドレス）次第で途中から異なるパターンをとる場合がある．よって，ある関数の入力パターンはすべてツリー構造で表現することができ，ある 1 組の入力セットはそのツリー上の一本のパスで表わされる．関数のメモ化を行うにあたり MemoTbl には，この全入力パターンを格納しそれを検索するための構造が必要となる．まず MemoTbl の構成について説明する．

2.2.1 MemoTbl の構成と動作

MemoTbl は以下の 4 つの構成要素からなる．

RF： 命令区間となる関数の開始アドレスを記憶するための表．RAM（Random Access Memory）で構成される．

RB： 命令区間の入力値を記憶するための表．入力値からすぐ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 Adressable Memory）で構成される [5].

RA： 命令区間の入力のアドレスを記憶するための表．RB と同数のエントリを持ち，一対一の対応関係をとる．RAM で構成される．

W1： 命令区間の出力値および出力アドレスを記憶するための表．RAM で構成される．

以下，これら構成要素による MemoTbl 内での検索手順について説明する．RB，RA，W1 のエントリのモデルを**図 3** に示す.

まずレジスタ上にある現在の入力値を取得し，この入力値を用いて RB に対し連想検索を行う．1 つ目の入力値に対し RB 内で一致するエントリが見つかった場合，上の同インデックスを持つエントリから得た次入力アドレスを用いてキャッシュを参照し，次の入力値を得る．これを繰り返してすべての入力の一致を確認した場合，入力セットの終端となった RA エントリには出力表 W1 へのポインタが格納されているため，その値を用いて W1 を参照する．W1 によって出力値が得られ，これをレジスタおよびキャッシュへと書き戻すことにより命令区間の処理を省略する．

TSID パージと RFID パージ

MemoTbl は有限であるため記憶できる入出力情報にも限界がる．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．そこで，MemoTbl 内の RB，RA，W1 は **TSID パージ**（Time-Stamp-ID purge）および **RFID パージ**（RF-ID purge）と呼ばれる 2 種類のエントリ削除の方式を備えている．

まず TSID パージについて説明する．このパージは LRU（Least Recently Used）方式に基づいており，RB，RA，W1 各機構のエントリはそれぞれすべてタイムスタンプを保持し，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から削除されてゆく．なお，ここで指す古いエントリとは，一致比較テスト失敗後に行われる MemoTbl への登録（write）および一致比較テストの最中に行われる MemoTbl からの読み出し（read）の両方が起こらなかったエントリのことである．

次に RFID パージについて説明する．前述した TSID パージだけでは，エントリ削除による登録領域の確保が間に合わない可能性がある．その場合に RFID パージが発生し，現在呼び出していた関数に属するすべての入出力情報エントリが削除される．これを実現するため，RB，RA，W1 各機構のエントリは RF によって割り当てられた ID（RFID）を保持しており，この ID が関数と一対一の対応関係をとっている．つまり，RF から現在呼び出している関数の RFID が分かり，RB，RA，W1 のエントリからこれと同じ RFID を持つものを削除する，という処理が行われる．

2.2.2 MemoBuf の構成と動作

上記 MemoTbl 内の情報を格納するために必要なバッファが MemoBuf である．MemoBuf は小規模で高速な RAM で構成され，命令区間の実行中，RF，RB，RA，W1 すべての情報を一時的に蓄積していく．例 2-3 を用い MemoBuf の動作を説明する．

例 2-3 では，関数 `main` が内部で関数 `F_NonLeaf` を呼び出し，関数 `F_NonLeaf` がまた内部で関数 `F_Leaf` を呼び出している．このように関数が入れ子構造となる場合，最初に検出する命令区間の終端はリーフ関数である `F_Leaf` となるため，この終端検出までの入出力情報（`Func_NonLeaf` および `main` のもの）も蓄えておく必要がある．そのため，MemoBuf は複数の命令区間分その入出力情報を保持することができ，終端を検出した命令区間から随時，スタックのように取り出し RF，RB，RA および W1 への登録を行なっていく．

例 2-3：関数の入れ子構造

```
int F_Leaf(int a, int b){   //内部での関数呼び出しがない
      int x;
      x = a/b;
      return (x);
}
int F_NonLeaf(int a, int b, int c){ //内部で関数 F_Leaf を呼び出す
      int x;
      x = c;
      x += F_Leaf(a,b);
      return (x);
}
void main(void){
      int a=0, b=0;
      a = F_NonLeaf(30,20,10);  //１度目の呼び出し
      b = F_NonLeaf(30,20,10);  //２度目の呼び出し
}
```

関数 main 内での 1 度目の関数呼び出しの実行を終えると，区間 F_Leaf を含んだ命令区間 F_NonLeaf の入出力情報が MemoTbl に登録される．2 度目の関数呼び出しの際には命令区間 F_NonLeaf に対するすべての入力一致比較に成功するため，命令区間 F_Leaf だけでなく命令区間 F_NonLeaf ごと計算再利用を行うことが可能となる．また，このように複数の命令区間に対して計算再利用を行うことを**多重再利用**と呼ぶ.

2.3　コスト軽減の提案

本節ではメモ化を行う際の電力コストやオーバーヘッドの存在について明らかにし，そのコストやオーバーヘッドに対する削減手法を述べる.

2.3.1　メモ化におけるコスト

消費電力／消費エネルギーコスト

プロセッサに対するメモ化機構の搭載により，リーク電流などの待機消費電力や MemoBuf や MemoTbl への読み書きによる動作電力が発生するため，プロセッサ全体の消費電力は上昇する．このため，MemoTbl の RAM サイズおよび CAM サイズを大

きくとることは現実的ではない．プログラムにはメモ化の恩恵を受けやすいものもあれば，逆に恩恵を受けにくいものもまた存在するのだが，どちらにせよ前述した待機消費電力や動作電力は発生する．したがってメモ化によって高速化ができなければ，消費電力が上昇している分だけ必然的に消費エネルギーもまた上昇することになる．

メモ化オーバーヘッドの存在

メモ化を行うことによりオーバーヘッドが発生する．具体的には，関数命令区間の開始時にレジスタ・キャッシュとメモ化機構 RB，RA との間で行われる入力比較のオーバーヘッドと，この入力比較が成功して出力が取り出される際の W1 からの書き戻しオーバーヘッドの 2 つがある．これを例 2-4 を用いて説明する．

例 2-4：入力比較・書き戻しオーバーヘッド

```
int out1=0; //大域変数その１
int out2=0; //大域変数その２
int Func_ex(int a, int b, int c, int d, int e){
      int x;
      x = a + b - c * d / e;  //引数全てを使用
      out1 = x;
      out2 = 2*x;
      return(3*x);
}
void main(void){
      int A=0, B=0, C=0;
      A = Func_ex(1,2,3,4,5);  //１度目の呼び出し
      B = Func_ex(1,2,3,0,0);  //２度目の呼び出し
      C = Func_ex(1,2,3,4,5);  //３度目の呼び出し
}
```

まず，前者の入力比較オーバーヘッドについて説明する．検出した命令区間が RF に登録されていた場合，必ず 1 回以上の入力比較が発生する．例 2-4 の場合，1 度目の Func_ex 呼び出しの際は入力比較オーバーヘッドが発生しないが，2 度目の呼び出しでは 4 つ目の引数が不一致だと判断するまでの計 4 つの引数に対して入力比較が行わる．また 3 度目の呼び出しでは，5 つ目の引数すべてに対しての入力比較が行われオーバーヘッドとなる．一方で後者の書き戻しオーバーヘッドは，入力一致比較がすべて成功

する 3 度目の Func_ex の時点で発生し，大域変数 2 つと 1 つの return 命令による計 3 つの変数に対する書き戻しがオーバーヘッドとなる．

当然だが，この書き戻しオーバーヘッドは入力一致比較の成功時につねに発生するがその失敗時には発生しない．その点入力比較オーバーヘッドは，呼び出した関数が RF に登録されてさえいれば一致比較の成功・失敗によらずつねに発生するため，オーバーヘッドとして占める割合は前者の入力比較によるもののほうが大きい．

2.3.2 コストへの対応

プログラム実行時に行う動的解析およびプログラム実行前に行う静的解析を行うことにより，前項で述べたメモ化におけるコストに対しての対応を図る．

動的解析による消費エネルギー制御

一般に，プログラム中で，メモ化による恩恵があまり得られていないものはメモ化すべきでないと考えられる．そこでプログラムの実行中に，メモ化が効果的に働いているかどうかを動的に解析することで，メモ化すべきか・すべきでないかを判断する．この解析からあまりメモ化の効果がないと判断された命令区間に対して メモ化を中止すると同時にメモ化機構への電力供給の遮断を行えば，余分な電力消費の発生を食い止めることが可能となり，結果的に消費エネルギーコストを削減することが可能となる．

これまで，メモ化により実際にどれほど電力を消費するのかは厳密に調査されてこなかった．そこで本稿では，自動メモ化プロセッサを対象とした消費電力見積もりを評価した上で，その消費エネルギーを削減する手法を提案する．

静的解析によるオーバーヘッド削減・消費電力抑制

実行前に対象プログラムに対し静的解析を行うことで，プログラムを実行する前の段階から実行時におけるなんらかの特徴を得ることができる．たとえばそれは，関数の大きさであったりその内部で存在する引数の数であったりと様々である．そしてこの得られた特徴をもとに，事前にどの命令区間をメモ化すべきか・すべきでないかを把握することが可能ならば，メモ化すべきでない命令区間をはじめからメモ化の対象外とすることで入力比較オーバーヘッドを削減することができる．また当該命令区間の実行中，その入出力情報を MemoBuf に一時的に蓄えておく必要もなくなるため，MemoBuf アクセスによる動的消費電力コストの削減も期待できる．そこで本稿では，自動メモ化プロセッサのためのプログラム静的解析を行う方法を提案し，入力比較オーバーヘッドおよび消費電力を減らす一手法を提案する．

次章より，提案する 2 つの解析手法についてそれぞれ説明していく．まず 3 章では

前者エネルギーコストへの対応手法を，その後の 4 章では，オーバーヘッド削減や電力抑制を図る手法について説明する．

3　動的解析を用いた消費エネルギー抑制

あるプログラムに関し，メモ化の恩恵があまり受けられないと判断できるとき，結果的にメモ化機構により消費される電力（エネルギー）は無駄となる．本章では，まず自動メモ化プロセッサに対する消費電力の見積もり [6] を行った．そのうえで，プログラムの実行中にメモ化が効果的に働いているかどうかを動的に解析するこにより，消費エネルギーを抑制する制御手法を提案する．

3.1　Wattch による消費電力見積もり

これまで，メモ化に関する電力見積もりは行われてこなかったため，まずその電力見積もり方法を準備する必要がある．そこで，自動メモ化プロセッサの消費電力を測定するにあたり，Wattch[7] を参考に，その消費電力見積もりの機構を実装した．なお，Wattch はアーキテクチャレベルでの消費電力シミュレータであり，汎用プロセッサのシミュレータとして広く用いられている SimpleScalar[8] へのパッチという形で一般に公開されている．以下，Wattch における電力の測定方法について説明する．

3.1.1　プロセッサのベース電力

Wattch は，プロセッサ内のユニットを RAM，CAM，組み合わせ回路，クロック回路の 4 つに大別し，それぞれのユニット対して静電容量や電源電圧，クロック周波数を用いてまずベース電力となるものを決定している．ここでベース電力とは，一定のアクセス回数に対し消費するとされる一定の電力であり，プログラムの実行を終了した段階での最終的な（平均）消費電力を求めるために定義されるものである．

ベース電力の算出にあたり，Wattch では cacti（**c**ache **a**ccess/ **c**ycle **ti**me model）[9] を参考に RAM や CAM の実装モデルを決定している．cacti はその名のとおりキャッシュアクセスで発生する遅延を小さくするため，キャッシュにおけるブロックサイズやライン数などのハードウェア構成を決定するツールである．cacti によるモデル化が行われたキャッシュ構造を**図 4** に示す．

たとえば図 4 におけるビットライン（BIT LINES）やワードライン（WORD LINES）など，その静電容量や抵抗値，配線の長さ，電圧等などのパラメータはプロセスルールにより異なってくる．そこで Wattch では，シミュレータが想定するプロセスルールの設定を可能としている．設定したプロセスルールに応じ，これらパラメータに対

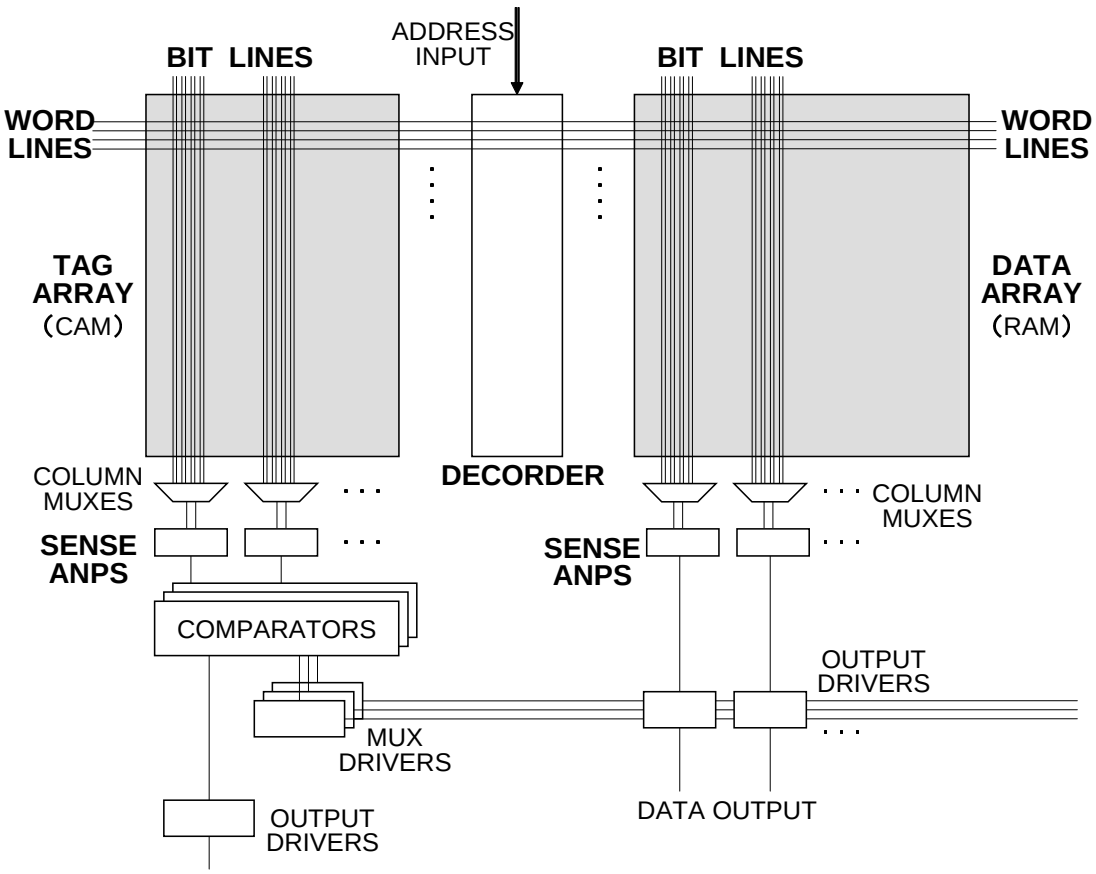

図 4: cacti によるキャッシュ（RAM，CAM）モデル

してのスケーリングファクタがあらかじめ規定されている．スケーリング後の各パラメータに基づきベース電力が決定される．

なお Wattch で設定可能なプロセスルールの種類には，順に 0.10μm，0.18μm，0.25μm，0.35μm（アルミ），0.35μm（非アルミ），0.40μm，0.80μm があり，デフォルトである SimpleScalar へのパッチの段階では 0.35μm（非アルミ）をとっている．各種パラメータの詳細については，文献および Wattch のソースコード（power.h）に記載されている．

以下，大別した各ユニット（RAM，CAM，組み合わせ回路，クロック回路）のベース電力の算出方法について簡単に説明する．

RAM

エントリの数，1 エントリの幅，読み書き転送ポートの数をパラメータとしてベース電力が決定される．レジスタファイルやキャッシュなどがこの RAM に相当する．プロセスルールに基づくレジスタ長やワードライン間隔をもとにデコーダのサイズが求ま

り，そのベース電力が算出される．また，ワードライン長やビットライン間隔に基づきワードラインが消費するとされるベース電力が算出される．ビットラインもワードラインと同様にベース電力が算出され，これら 3 つの合計がレジスタファイルのベース電力となる．キャッシュの場合には，前述した 3 つのベース電力の他にセンスアンプの電力分を追加している．

CAM

RAM におけるビットラインおよびワードラインと同様，CAM ではタグラインとマッチラインがモデル化される．設定したプロセスルールに基づいて決定されるトランジスタ幅からタグラインの静電容量が求まり，これにビットラインとワードラインの静電容量を加えてタグライン全体の静電容量が求まる．マッチラインの静電容量も同様に求められ，タグラインとマッチラインの静電容量から求まる電力が CAM 全体で消費されるベース電力となる．

組み合わせ回路

書き戻しに使われるリザルトバスや ALU など，それぞれの構成に応じて消費電力が計算される．リザルトバスに関しては，プロセスルールごとに規定されたレジスタ長やワードライン間隔からバス長が見積もられ，これにデータ幅を乗じて全体の静電容量が求められる．ALU は演算器と浮動小数演算器とに大別してモデル化されており，それぞれに応じたベース電力が算出される．

クロック回路

プロセッサ全体にクロックを供給するためのクロック線，およびクロック動作のためのトランジスタであるクロックバッファの消費電力に加え，クロックロードのための電力から総消費電力が算出される．

3.1.2 消費電力と消費エネルギーの算出

Wattch では前節で説明したベース電力を基に，各ユニットにおける消費エネルギー E，および単位時間あたりの平均電力 P を以下の式で算出している．

$$E = P_{base} \int_0^T A_{r/w}(t) \cdot a_f(t) dt \tag{1}$$

$$P = E/T \tag{2}$$

P_{base} はそのユニットにおけるベース電力であり，$A_{r/w}(t)$ はユニットへの時刻 t におけるアクセスに対して正の相関を持って変化する係数である．また，$a_f(t)$ はゲート稼働率であり，$0 \leq a_f(t) \leq 1$ と動的に変化する場合と，静的に $a_f(t) = 0.5$ と固定する場合がある．ただし，ユニットによっては計算過程に $a_f(t)$ を含めないものもある．

式（1）でベース電力を稼働時間Tにより積分することで，そのユニットにおける最終的な消費エネルギーを求める．また式（2）では，式（1）で求めたEを用いて単位時間あたりの平均電力Pを求めている．本稿では単に消費電力と記述した場合この平均電力を指すこととする．

3.1.3 自動メモ化プロセッサへの実装

自動メモ化プロセッサに対して電力評価関数の実装を行う．まず，メモ化機構に含まれないプロセッサ要素であるクロック回路とALUに関してはWattchの評価式をそのまま移植する形で実装した．1次データキャッシュ，2次データキャッシュおよびレジスタ部はそれぞれ以下のようにして消費電力評価関数の実装を行った．

1次データキャッシュ 自動メモ化プロセッサの想定するキャッシュは，ブロックサイズが32B（Bはバイト），ライン数は256，総容量が32kBとなっている．また，TLB（Translation Lookaside Buffer）による連想検索の機能は用いていない．そのため，上記キャッシュサイズのパラメータにより電力評価式を見積もる際，Wattchにおける1次キャッシュからCAM部の電力を差し引くことで実装した．またWattch同様，読み書きのためのポート数は2つとしている．

2次データキャッシュ ブロックサイズ32B，ライン数16384，総容量2MBを想定している．1次キャッシュ同様，Wattchにおける2次キャッシュと同じ電力評価式を上記サイズのパラメータを用いて実装した．読み書きのためのポート数はWattchと同じく1つとした．

レジスタ部： 整数レジスタファイル，浮動少数点レジスタファイルおよびこれらレジスタへの書き込みバスをまとめてレジスタ部とし，それぞれで電力評価式を算出する．32bit整数レジスタが72本（内訳：グローバルレジスタ8本，ウィンドウレジスタ16本×4セット），64bit浮動小数点レジスタが32本とし，Wattchにおけるレジスタ部と同じ電力評価式を上述したパラメータを用いて実装した．頻繁に読み書きアクセスが発生するためポート数はWattchと同じく3とした．

また2章で述べたように，メモ化機構にはMemoBufおよびMemoTbl（RF，RA，RB，W1）が存在する．これらについては，それぞれ以下のようなRAMおよびCAMとして消費電力評価関数の実装を行った．

MemoBuf： MemoTblへの書き込みバッファとして頻繁にアクセスされるため，Wattchにおける2ポートの1次キャッシュ（CAM部除く）と同じ電力評価式で実装した．ブロックサイズを32B，総容量を19kBとした．

RF： 命令区間表であり，複数ポートは必要ないため1ポートの1次キャッシュとし

て実装した．サイズは幅 46B，エントリ数 256，総容量が 12kB とした．

RB： 入力値セットのための表であり，連想検索が必要となる．Wattch では キャッシュ内の TLB が CAM を使って構成されているため，これと同じ電力評価式を用いて実装した．幅 36B，深さ 1k 行となり，総容量が 36kB となる．

RA，W1： 入力値のアドレス表および出力値表である．1 ポートの 2 次キャッシュと同じ評価式で実装した．RA，W1 の幅はそれぞれ 25B，75B でありエントリ数が RB と同じ 1k 行であるため，総容量はそれぞれ 25kB，75kB となる．

最後に，ここまで述べた消費エネルギー（消費電力）が算出されるまでの手順を 1 次データキャッシュを例に，簡単な擬似コードを用いて説明する．

例 3-1：電力評価の擬似コード（1 次データキャッシュ）

```
float D1_baseP;        //一次データキャッシュのベース電力 [W]
float D1_Energy = 0;  //同 消費エネルギー（実行開始時は０ [J]）
void Start_Simulation(){
   int D1_accsess;    //アクセス回数の宣言
   for( ;  ; cycle_time++ ){
      D1_accsess = 0; //アクセス回数リセット
      /* 対象プログラムの実行を 1 サイクルづつ進めてゆく */
      if( 1 次データキャッシュアクセス発生 )D1_accses++;

      D1_Energy += Aa( D1_access ) * D1_baseP;
   }
}
void main(){
   D1_baseP = calculate_basePower( D1, 32B, 256lines, 32kB );
   Start_Simulation();
   Print_Power( D1_Energy );
}
```

例 3-1 は，1 次データキャッシュにおける電力評価プログラムを簡易化したものである．まず main 関数内で，はじめに 1 次データキャッシュのパラメータからベース電力の算出を行い（calculate_basePower），そのあと対象プログラムのシミュレーションを行う（Start_Simulation）．シミュレーションは 1 サイクル毎に行われ，そのサイクル

で発生したキャッシュアクセスも計測しておく．そして1サイクル分の動作を終えたら，ベース電力 D1_baseP に対してアクセス回数により算出される計数 Aa(D1_accsess) を乗じ，その1サイクルで発生した消費エネルギーを算出し累計していく．なお，この Aa が前節の式 (1) における $A_{r/w}(t) \cdot a_f(t)$ に相当している．この作業を毎サイクル，シミュレーションを終えるまで繰り返す．シミュレーションの終了後，それまでに累計していた消費エネルギー [J] を出力させる（Print_Power）．以上で述べた手順により1次データキャッシュの消費エネルギーを求めることができる．

3.2 低消費エネルギーモデル

3.2.1 メモ化機構と消費エネルギー

ここまで述べたように，メモ化のためには MemoBuf および MemoTbl（RF，RB，RA，W1）などのユニットが必要となり，これら各ユニットにおける消費電力の増加分を考慮する必要がある．

一方で，メモ化はプログラム中の値の局所性を利用した高速化手法であるため，値の局所性のないプログラムの場合にはメモ化による高速化は期待できない．そのような場合に問題となるのは，メモ化機構を稼働させることによる無駄な消費電力およびエネルギーの増大である．

このような時間区間を多く持つプログラムでは，メモ化機構を起動しておく必要はないといえる．なおここで時間区間とは，命令区間の集合に対してかかる実行時間のことである．そこでプログラムの実行中に，動的にメモ化の効果を解析することで前述した区間を判定し，本来メモ化の効果が得られない区間をメモ化対象から除外し余分なエネルギー消費を抑制するアルゴリズムを提案する．

3.2.2 エネルギー制御アルゴリズム

前節で述べた，余分なメモ化を行わない自動メモ化プロセッサの実現にあたり，計算再利用の成功頻度を動的に検知する必要がある．まず，新たに**図 5** に示すようなバイナリカウンタを用意した．なおこれらは，そのビット長が多くても2バイトを越えないほどの比較的小さなカウンタであり，プログラムカウンタに比べてもアクセス頻度はきわめて低く，プロセッサ全体に影響する電力増加は無視できるほどわずかである．このカウンタを用い，対象プログラム実行中に以下の2つを計測する．

(1) 関数呼び出しが発生した回数

(2) 計算再利用が成功した回数

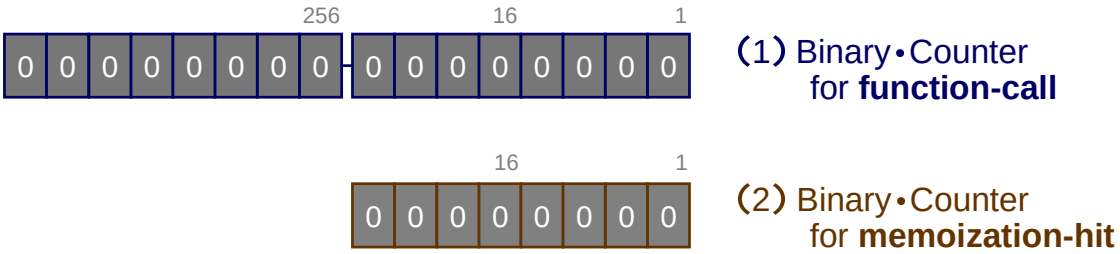

図 5: バイナリカウンタ

なお（2）は 2 章図 1 において，reuse test の成功により writeback が発生した回数に相当する．もちろん，関数呼び出しが起きた時 MemoTbl に登録されていない命令区間には reuse test が行われない．登録された命令区間の中からさらに入力比較まですべて一致したものだけが計算再利用成功となるため，必ず（2）の回数は（1）の回数以下となる．

たとえば（1）の回数に比べ（2）の回数が極端に小さかった場合は，ある時間区間内でメモ化の恩恵が十分得られていなかったことを意味し，逆に（2）の回数が大きい場合にはメモ化が効果的に働いていたことを意味する．以降これらの値を用いて実現した，2 つのエネルギー制御アルゴリズムについて説明する．

メモ化中断アルゴリズム

メモ化による再利用成功率が低い場合，メモ化を中断しメモ化機構への電力供給を遮断することで，無駄なエネルギー消費を抑えることができる．そのメモ化中断を決定する判定方法を以下に述べる．以下，本稿ではこれを**メモ化中断**アルゴリズムと呼ぶ．

中断の判定を行うために，前節（1）の関数呼び出し発生回数に対する閾値として N_{call}，前節（2）の計算再利用成功回数に対する閾値として N_{hit}，の 2 つを定義する．判定時におけるカウンタモデルを**図 6** に示す．

関数呼び出しが N_{call} に達する前に計算再利用が N_{hit} 回発生した場合には，メモ化が有効に働いていると判断し，関数呼び出し回数カウンタをリセットして引き続きメモ化およびメモ化の閾値判定を行なっていく．

なお，counter reset など図 6 内で矢印で示された信号は，そのビットへの桁上がりが発生した瞬間に出力されるものであり，当該ビットが 1 の間つねに出力されているわけではない．

一方，計算再利用が N_{hit} 回成功する前に呼び出し回数が N_{call} に達した場合にはメモ化機構を停止し，計算再利用を行わない通常のプロセッサとして動作する．このと

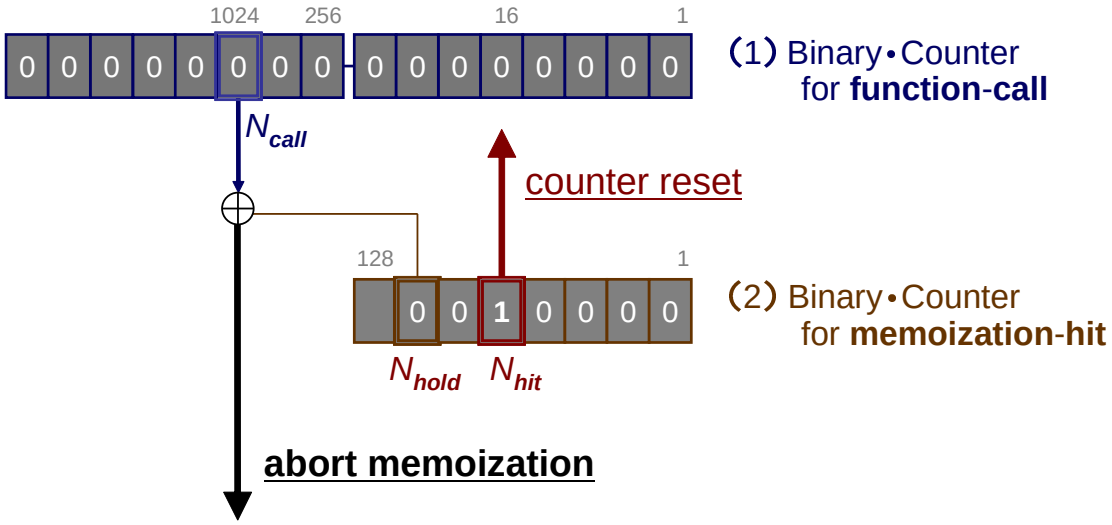

図 6: メモ化中断アルゴリズムのカウンタモデル

きメモ化機構への電力供給をシャットダウンするため，MemoBuf および MemoTbl の内容は揮発する．

なお，一度メモ化の中断が決定されると，それ以降計算再利用による高速化が一切望めなくなるため，メモ化有効性チェックのためのパラメータは比較的中断が起こりにくくなるよう設定している．具体的には，まずメモ化の効果を確認できる十分な時間区間として $N_{call} = 1024$（$= 2^{10}$）とし，その上で $N_{hit} = 16$（$= 2^4$）とした．これは，一般にメモ化によるサイクル数削減効果のみられるプログラムの多くで，関数呼び出し回数のうちの数%以上で計算再利用が起きているからであり，この $N_{call} = 16$ は関数呼び出し回数の閾値 $N_{hit} = 1024$ うちの約 1.5%に相当する．

また，（2）のメモ化ヒット回数カウンタは，何回連続で閾値判定に成功したかの履歴を記憶することができる．これを用いて，メモ化効果の得られているプログラムではメモ化中断が起こりにくくなるよう配慮した．図中に示した N_{hold} がメモ化の効果が得られていることを示すビットであり，4 回連続して閾値判定をパスできた場合にこのビットへの桁上がりが発生し 1 となる．また N_{hold} は飽和ビットであり，再度桁上がりが発生したとしても 0 とはならない．そして，関数呼び出し回数が N_{call} に達した際に行われるメモ化中断に対してこのビット N_{hold} との排他的論理和をとる．この排他的論理和により，今までメモ化による効果が出ていると判断されたプログラムに対して，1 回の閾値判定でメモ化中断を決定してしまうことを防ぐ．もちろん，排他論理和をとった後には飽和ビットごとメモ化ヒット回数カウンタをリセットする．

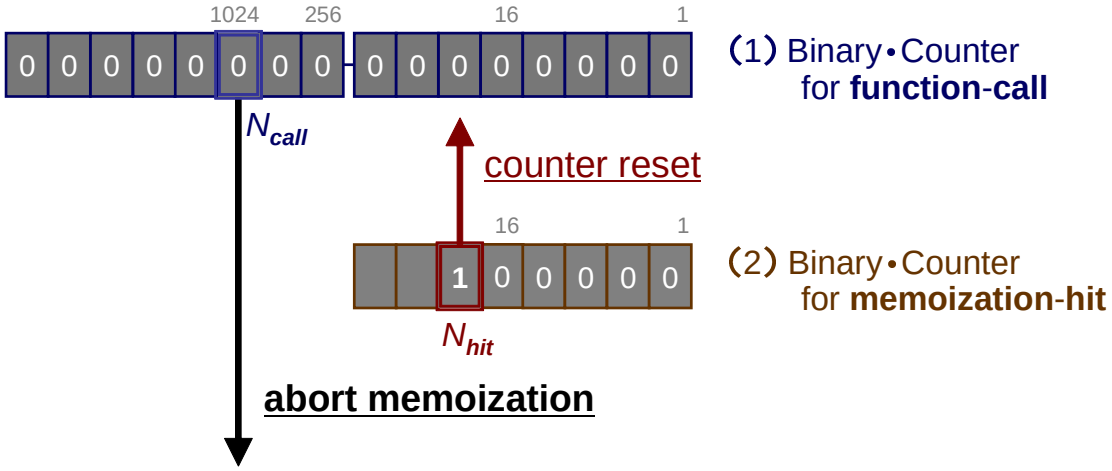

図 7: メモ化再開アルゴリズムのカウンタモデル（中断判定時）

メモ化再開アルゴリズム

先に述べたアルゴリズムはメモ化の中断のみを考慮したものであるが，プログラムにはメモ化効果に時間的局所性のあるものがある．たとえば，メモ化の効果がプログラム実行開始直後には現れないがプログラムの後半で効果が現れてくるようなプログラムに対しては，メモ化を中断した場合，本来得られるべきメモ化による高速化の恩恵が受けられなくなる可能性がある．

一方で，本章で述べるメモ化の有効性チェックは実際にメモ化機構を動作させた状態で初めて可能となる．よって，中断以降メモ化による効果が得られるかどうかを判定するために，メモ化を中断した以降も適宜メモ化を再開し，プログラム中におけるメモ化効果を判定できる状態にしておく必要がある．そこで，メモ化を中断した状態からメモ化を再開するアルゴリズムを提案する．以下，本稿ではこれを**メモ化再開**アルゴリズムと呼ぶ．

メモ化再開アルゴリズムでの，メモ化中断判定におけるカウンタモデルを**図 7** に示す．まず，本再開アルゴリズムを中断アルゴリズムと組み合わせて使用する場合，前節のようにあえて中断を起こりにくくする必要はない．このため，中断を決める閾値 N_{hit} は前節より大きく 32（$= 2^5$）とした．また図 6 で説明した飽和ビット N_{hold} によるメモ化中断決定の回避も行わないこととした．

次に，中断後メモ化再開判定を行うためのアルゴリズムを追加する必要がある．まずメモ化中断および再開が起きているモデルを**図 8** に示す．

まず，メモ化を中断している状態の継続期間は関数呼び出しが $2^k \times N_{call}$（k は整数）

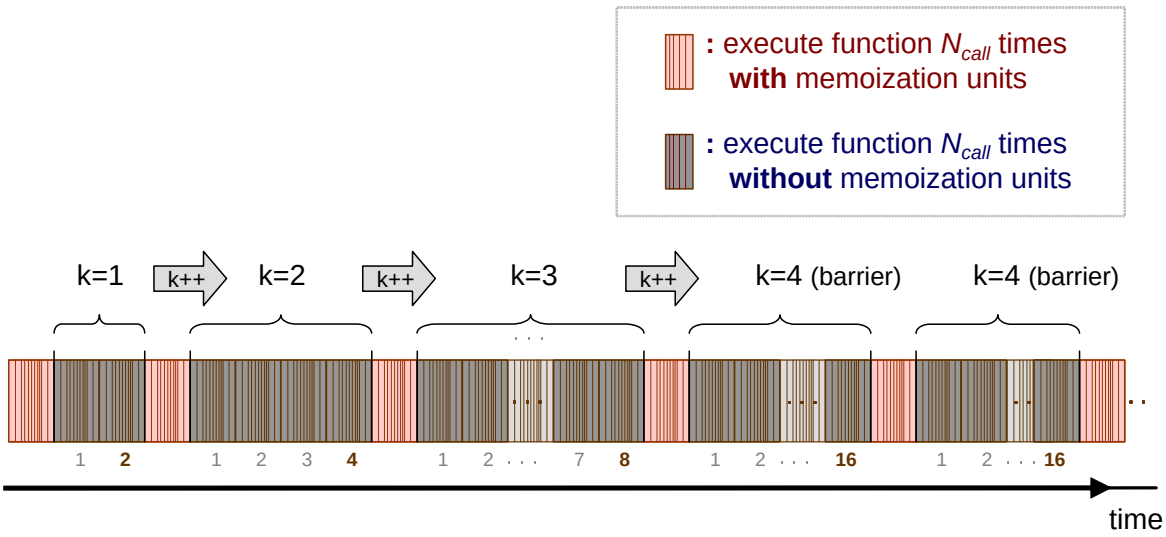

図 8: メモ化中断期間 k の変化の様子

回起こるまでとした．これにともない，k の状態を保持するためのビット長 4 となるカウンタを用意した．メモ化中断状態の継続中は，中断期間の長さを決定する k を動的に変化させる．図 8 はメモ化の効果が得られないプログラムを実行した際の k の変化を表わしており，具体的には初期値を $k = 1$ からメモ化中断と判断されるごとに最大値 $k = 4$ までインクリメントする．そのため，メモ化機構を動作させている時間区間 N_{call} に対し，メモ化機構を動作させていない時間区間は最大で $16 \cdot N_{call}$ となる．メモ化再開条件を満たすと初期値 $k = 1$ にリセットする．この k の導入により，メモ化の効果がほとんど得られないプログラムにおいて頻繁なメモ化再開によるエネルギー消費を抑えることができる．なお，メモ化再開後も本項のメモ化中断アルゴリズムにしたがうものとする．もちろん，中断時以降は供給電力がシャットダウンされているため再開時には MemoBuf および MemoTbl には中断以前の情報は保存されていない．

中断状態におけるメモ化再開判定のカウンタモデルを**図 9** に示す．まず，k を記憶するカウンタは `0001`（$k = 1$），`0010`（$k = 2$），`0100`（$k = 3$），`1000`（$k = 4$）の 4 状態をとる．またメモ化中断の発生直後には関数呼び出しカウンタをリセットして 0 とする．これにより，k の各ビットと関数呼び出しカウンタにおける N_{call} の上位 4 ビットとのマスクを取ることで，関数呼び出しが一定に達したかどうかを判断することができる．

なおメモ化再開時には MemoBuf および MemoTbl のウォームアップ時間を考慮する必要があるが，これはゲーティッド V_{dd}[10] などの他の供給電力制御手法と違い，そ

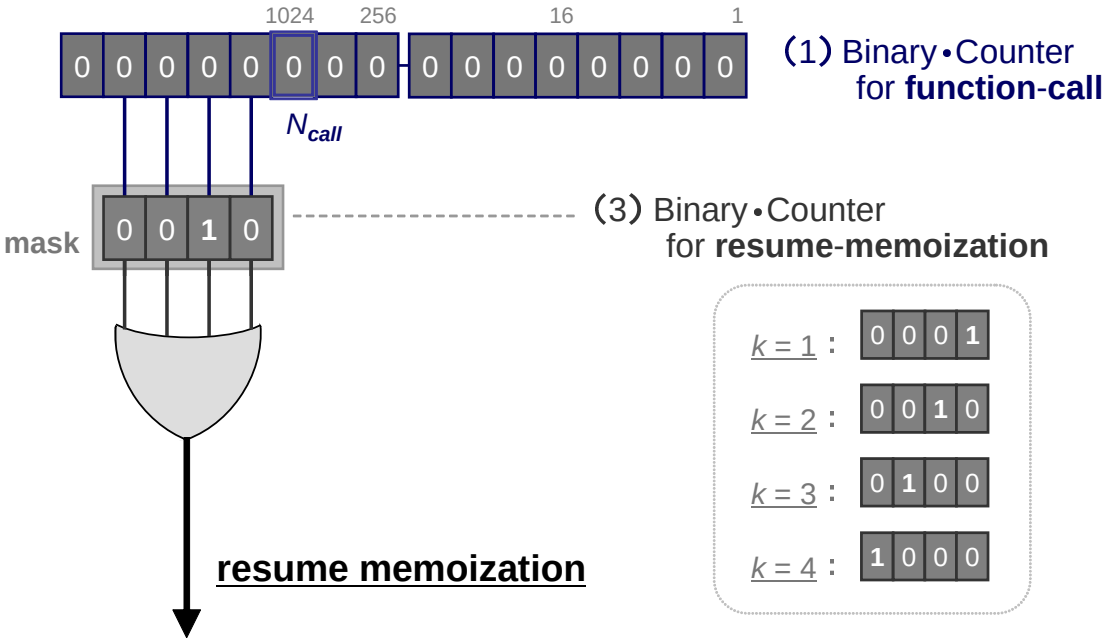

図 9: メモ化再開アルゴリズムのカウンタモデル（再開判定時）

の性能に与える影響はほとんどないといえる．本手法では再開条件を満たしたと判断されるのはある関数呼び出しの発生直後となるため，その関数の命令区間はまだメモ化対象でなくメモ化機構も機能を提供する必要はない．次の関数呼び出しの発生までに機構の準備ができていれば十分である．汎用 GA プログラムである GENEsYs を用いてこの影響の予備評価を行なったが，再開直後に MemoBuf および MemoTbl を使用できない期間が数万クロック程度以下であればほとんど影響が出ないことを確認した．

4　静的解析による高速化・消費電力抑制

3 章では，動的にメモ化の効果を解析しその消費エネルギーを制御する手法について説明した．一方本章では 3 章の動的解析とは異なり，事前に実行対象となるプログラムに対して解析を行う．この静的解析を用いて，関数単位でメモ化による高速化の恩恵が受けられるかどうかの判定を行う．

まず高速化の妨げとなるオーバーヘッドの存在について述べ，これを見積もるためのプログラムの静的解析方法を提案する．そして静的解析で得られた情報をもとにして，どの関数をメモ化すべきか・すべきでないかといったメモ化のヒント情報を自動メモ化プロセッサに対して伝えるための実現方法について説明する．

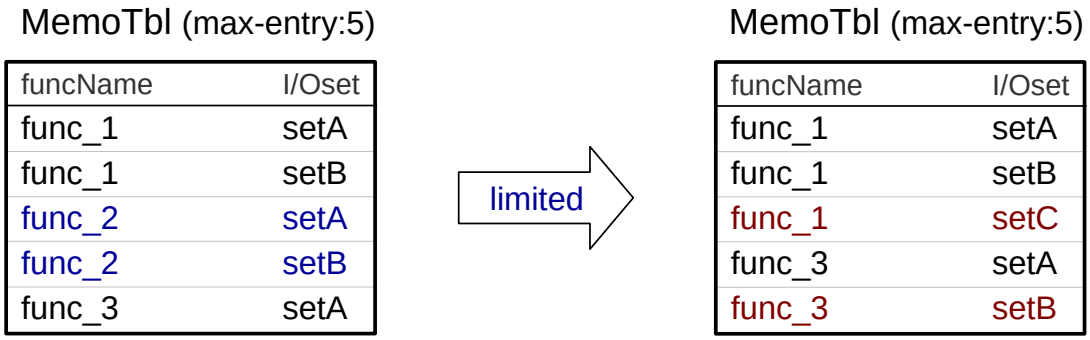

図 10: MemoTbl への登録状況の変化

4.1 メモ化利得とオーバーヘッド評価

2 章で説明したようにメモ化を行う際にはオーバーヘッドが発生する．一方で，本来実行するはずであった関数にかかる実行時間がこのオーバーヘッドを上回っていた場合，それらの差分が 1 命令区間で高速化された時間となる．この時間を**メモ化利得**と名付け，これを以下の式で定義する．

$$G = T_{exe} - Ovh \tag{3}$$

$$Ovh = Ovh_R + Ovh_W \tag{4}$$

G がメモ化利得であり，T_{exe} が関数実行にかかる時間である．そして入力比較に必要となるオーバーヘッド Ovh_R と出力書き戻しに必要なオーバーヘッド Ovh_W との和 Ovh がメモ化の際に発生するオーバーヘッドである．

命令区間の中で，このメモ化利得の値が小さなものに対してはメモ化を行わなければオーバーヘッドの削減が期待できる．加えて，有限である MemoTbl にメモ化する価値のない命令区間が登録されなくなるため，MemoTbl 内には価値のある命令区間の占める割合が上昇する可能性が出てくる．これを**図 10** を用いて説明する．図 10 は入出力情報を最大 5 つまで記憶できると仮定した場合の MemoTbl をモデル化したものである．左側の MemoTbl には関数 func_1，func_2 および func_3 に対する入出力セットが保存されているが，ここで本章で提案するような静的解析の結果より関数 func_2 はメモ化による高速化の効果が得られない関数または計算再利用の成功が見込めない関数と判断されたとする．これをメモ化対象から除外した場合が右側の MemoTbl である．MemoTbl への登録量が減ったため 2 章で説明した TSID パージは起こりにくくなり，func_1 や func_3 の入出力情報がこれまでより多く登録される可能性もある．

以上のことからメモ化ヒット率向上による高速化が期待できる．また図 10 の関数

func_2 のように，メモ化の際に必要となる MemoBuf および MemoTbl への参照自体が削減されるため，その動的消費電力削減の効果も期待できることになる．

次に以下では T_{exe} および Ovh の見積もり方法について説明する．実行するプログラムのメモ化利得解析を行うにあたって，そのアセンブリを解析対象とした．

4.1.1 関数の命令数見積もり

まず関数の実行にかかる時間となる T_{exe} の見積もりを行うにあたり，本章で説明する静的解析は動的に実行した場合と違い，実際にかかる実行サイクル数が分かるわけではない．そのため，実行サイクル数に代わるなんらかの評価基準を用意して関数にかかる実行時間を近似的に推測必要がある．本稿では実行サイクル数と高い相関があると考えれられる，関数内の命令数を解析することにより関数にかかる実行時間を推定した．

命令数の解析を行うにあたり，ただ関数内の命令数を計測すればよいわけではない．それは，たとえばループイタレーション内における命令は何度も実行される可能性があり，また条件分岐先の taken/not-taken 各区間内における命令などは条件により実行されない可能性がある．そこで本節では関数の実行時間を推測するにあたり，分岐命令とその命令に対するラベルを検出することで，できるかぎり動的に実行される場合と同じ命令数となるような近似を行なった．命令数計測近似のために考慮した特徴的な命令を以下に挙げる．

(a) 下方への条件分岐命令
(b) 下方への無条件分岐命令
(c) 上方への無条件分岐命令
(d) 関数復帰命令（リターン命令）

以降これらの命令検出に基づき，モデル化を行なった命令数計測の方法を順に説明する．

（a）下方への条件分岐命令

命令数のモデル化の際，まず考慮したのが下方への条件分岐命令である．本稿では，実際に実行される命令数の期待値を求めるにあたり，その**条件分岐確率を** 1/2 **とする**ことで命令数計上を行なった．そこで，以下の例 4-1 のような関数を例に命令数のモデル化について説明する．

例 4-1：命令数のモデル化・サンプル

```
Func_Sample( ){
        /*  処理 X  */
        if( *** ){ /*  処理 Y  */ }
        /*  処理 Z  */
}
```

関数 Func_Sample は処理 X，処理 Y および処理 Z から構成され，条件分岐区間に含まれる処理 Y 以外は必ず実行される．そこで，これらの処理 X，Y，Z における命令数をそれぞれ x，y，z として**図 11** のようにモデル化を行なった．

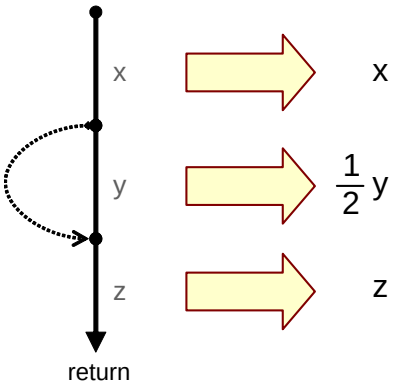

図 11: 命令数計測モデル（a）

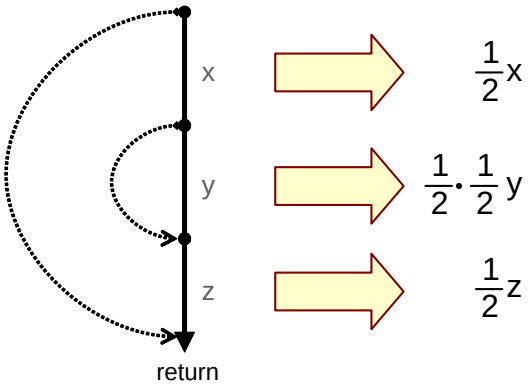

図 12: 多重規則への対応例

図 11 では，命令数 x および z の区間は必ず実行されるが命令数 y の区間は実行されない可能性があることを表現している．つまり下方への条件分岐命令とそのラベルとの間に含まれるのがこの y 命令となる．そこで，該当区間であるこの y に実行確率の 1/2 を掛けることで区間の命令数を計上した．

一方，たとえば if 文の中でさらに if 文が呼び出される場合，前述した下方条件分岐の区間は 2 重となる．この場合のモデルを**図 12** に表わす．図 12 では，命令数 y の区間にて条件分岐区間が 2 重となっており，この場合も（a）に対する命令数計測規則を重ねて適用させる．したがって，x，y，z をもつ全命令区間に条件分岐確率 1/2 が掛けられ，その中で条件分岐が行われる y にはさらに 1/2 を掛けて命令数を計上した．

（b）下方への無条件分岐命令

たとえば if-else 文のように，命令列において taken 処理のまとまりの後に not-taken のまとまりがある場合などには必ず下方への無条件分岐命令が存在する．

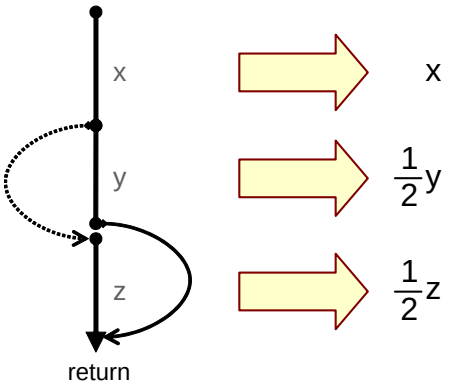

図 13: 命令数計測モデル（b）

これをモデル化したものを**図 13** に表わす.

図 13 では，x は必ず 1 回分実行されるためそのまま x となり，z は前述した（a）の分岐区間に含まれているため $(1/2)y$ として計上されている．一方 z は下方への無条件分岐命令とそのラベルの間に含まれる命令区間であり，y を実行しなかった場合のもう 1/2 の確率で実行されることになる．したがって該当区間を 1/2 倍することでの命令数 z を計上した.

（c）上方への無条件分岐命令

ループによるイタレーション区間が存在する場合，必ず上方への分岐命令が存在する．一方で，たとえば上方分岐命令の跳び先で関数復帰命令を呼ぶ場合などもあり，上方への分岐命令のすべてがループとなるわけではない．また，イタレーション区間は，動的な実行順序をたどった際に 2 回実行された命令を検出することではじめて検出が可能となるため，アセンブリの静的解析でその区間を判別することはきわめて困難である．そこで本解析では，**無条件上方分岐とそのラベルとの区間における命令数を 2 倍で計上する**ことにより，前述したイタレーション区間でない上方への条件分岐区間に対して対応を図った．また，1 イタレーション区間は必ず 1 回は通過するものとしてモデル化を行なった．この例を**図 14** に示す.

図 14 では，x および y の命令区間がループ区間に該当しており，y の区間内だけで描かれる破線のパスがループ区間を抜けるパスである．イタレーション区間を 1 回分実行するという決まりより，このモデルにおける実際の動的な実行順序としては $x \to y \to x \to z$ となり，計 $(2x + y + z)$ 命令となる．これを静的解析では，$y \to x$ のように動的実行順序と同じパスとして考えるのではなく，上方への無条件分岐命令とそのラベルの間に含まれる命令区間 x および z を 2 倍の重みで計測することにより実現した.

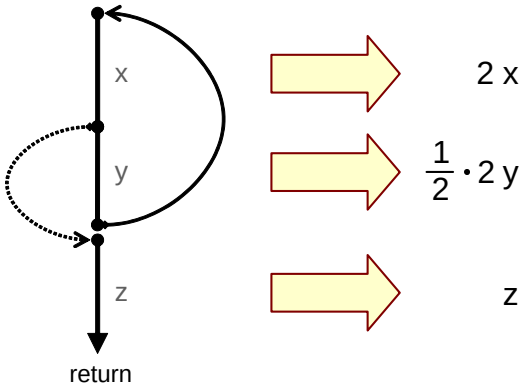

図 14: 命令数計測モデル（c）

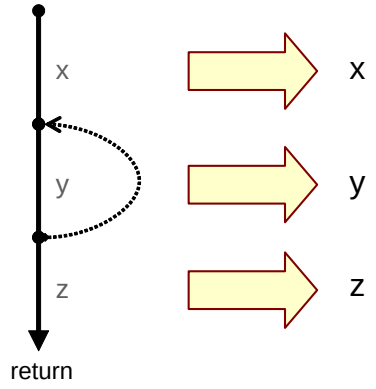

図 15: 上方への条件分岐の例

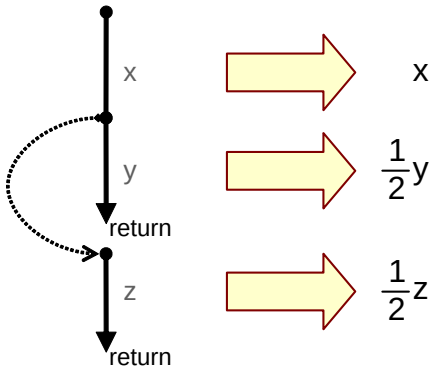

図 16: 命令数計測モデル（d）

したがって，x は実際に 2 回通る $2x$ に計上され，y は（a）によるモデル化の 1/2 倍と本法則（c）の上方分岐での計測規則 2 倍との積となり，実際に通る 1 倍の重みで計上される．

一方，上方への条件分岐の場合の例を**図 15** に示す．図 15 において，上方分岐とそのラベルとで挟まれた命令数 y の区間はループ内のイタレーション区間に該当する．ここで，x，y の区間の実行を終えた直後上方への条件分岐を検出したときには，すでにイタレーション区間 y は必ず 1 回通っていることになる．したがって，当命令を検出しても（a）（b）（c）で説明した『該当区間を n 倍する』というような命令数の計上は行わないこととする．

（d）関数復帰命令（リターン命令）

関数の中にはリターン命令が複数あるものも存在する．たとえば if-else 文の中で，taken 処理の終わりに 1 つのリターン命令，not-taken 処理の終わりにもう 1 つのリターン命令が存在する場合がある．これをモデル化したものを**図 16** に示す．

図 16 では，実際の実行モデルとしては y および z はそれぞれ 1/2 の確率で実行されることになる．一方で，y の区間の最後のリターン命令検出以降，この y を 1/2 の確率で実行していたパスは以降の区間にあたる z には合流しないことが分かる．そこで，リターン命令検出以降の命令数を 1/2 の重みで計測した．これにより z の区間は，実際に通る確率を掛けた $(1/2)z$ として計上される．

4.1.2　オーバーヘッドの見積もり

アセンブリの静的解析によりメモ化が成功した際に発生しうるオーバーヘッドを見積もる．このオーバーヘッドには，式（4）で挙げた入力比較オーバーヘッド Ovh_R と出力書き戻しオーバーヘッド Ovh_W があるが，これらの検出方法について説明する．ある関数中においてオーバーヘッドのために解析の対象としたものを以下に示す．

（A） 読み出しの発生する入力レジスタ
（B） 読み出しの発生する大域変数
（C） 書き込みの発生する出力レジスタ
（D） 書き込みの発生する大域変数

以上の 4 つのうち，（A）と（B）が入力比較オーバーヘッドに該当し，（C），（D）が出力書き戻しオーバーヘッドに該当する．これらを順を追って説明する．

（A）読み出しの発生する入力レジスタ

関数の実行中，読み出しの発生する入力レジスタはその関数の入力（メモ化入力一致比較の対象）として扱われる．ここで入力レジスタとは，SPARC 特有のレジスタウィンドウにおいて関数入力に使われると約束されたレジスタ領域を指す．また，同様に出力レジスタというレジスタ領域も存在する．例 4-2 に入力レジスタのアセンブリ解析例を示す．

例 4-2 の先頭行が関数 `Sample_In` のラベルとなっており，その後の save 命令によりスタックポインタ `%sp` をずらし `Sample_In` の使用領域を確保する．2 つある `!#PROLOGUE#` は，行頭の `!` がコメントアウトを意味するため処理に影響を与えるわけではないが，関数ラベルの開始個所に必ず挿入されるコードである．以降，変数ラベルと関数ラベルを区別するためこの `!#PROLOGUE#` 表記を用いた．これ以降の命令群が，入力値および出力値解析を含めたすべての解析の対象となる．

例 4-2：レジスタ入力のアセンブリ解析

```
Sample_In:
            !#PROLOGUE# 0
            save      %sp,  -96,  %sp
            !#PROLOGUE# 1
            add       %i0,  100,  %i0
            mul       %i1,  %i0,  %i1
            sub       %i2,  %i0,  %i3
            : :         :     :     :
```

add 命令行以下の入力値解析について説明する．SPARC では，%i0〜%i7 が関数の入力値である引数として使用されるレジスタであり，本例では%i0，%i1，%i2，%i3 の 4 つまでが使用されている．加算命令 add のオペランドのうち左端の%i0 および 100 がソースとして使用され，右端の%i0 がデスティネーションとして使用される．したがって，ソースとして使用された%i0 は関数の引数だと特定できる．同様にして乗算命令 mul および減算命令 sub からは，それぞれ%i1，%i2 が引数として特定できる．ここで sub 命令の行に注目すると，これまで使われていなかった%i3 がデスティネーションとして使用されることが分かる．これは，もともと%i3 が引数として使用されておらずそのレジスタ領域を計算過程で使用しただけである．よって，以降%i3 がソースとして参照されたとしても関数の仮引数としては扱わないようにした．

なお，例 4-2 の関数 Sample_In では%i を入力レジスタとして解析しているが，自身に関数呼び出し命令を含まないリーフ関数の場合は，関数の仮引数として入力レジスタ%i の代わりに出力レジスタ%o が用いられる．そのためリーフ関数の場合には%o を対象として同様の解析処理を行なった．

（B）読み出しの発生する大域変数

2 章（例 2-4）で説明したように，メモ化テストの入力比較対象には大域変数も含まれるためそのアクセスオーバーヘッドも計上する必要がある．例 4-3 に大域変数入力に対するアセンブリ解析例を示す．この例では， out が大域変数ラベルであり，値として 200 が格納されている．この大域変数 out に対する参照が関数 Use_out 内で行われている．

例 4-3：大域変数入力解析のアセンブリ解析

```
              : :        :       :          :
out:
              .uaward  200
              : :        :       :          :
Use_out:
              !#PROLOGUE# 0
              : :        :       :          :
              ld         [%fp+68],        %o0
              ld         [%i0+%lo(out)], %i1
              add        %i2,     %i1,    %i2
              : :        :       :          :
```

大域変数に対する参照には必ずロード命令 ld が用いられる．一方で，局所変数によるフレームポインタ%fp を利用したロード命令もあるためすべてのロード命令が入力となるべきではない．そこで大域変数を判別するため，ローカルレジスタ%lo を用いたロード命令のみを入力として検出するようにした．本例の関数 Use_out では，%fp を用いた 1 つ目の ld 命令は入力として検出されず，%lo を用いた 2 つ目の ld 命令のみが大域変数からの参照として検出されることになる．

なお大域変数の解析にあたり，すべてが入力として検出できるわけではない．たとえば前述した局所大域変数 out のようにその格納アドレス領域が静的に確保されたものは入力としての検出が可能である．しかし，malloc 関数など格納アドレス領域が動的に確保された変数に関しては，フレームポインタやレジスタに格納された値を解析する必要がある．そのような場合，ロード命令によりロードした変数アドレス領域が大域変数のものであるかどうかと特定するのは静的解析では困難である．

（C)・(D）書き込みの発生する出力レジスタ・大域変数

関数の実行中，書き込みの発生する出力レジスタがその関数の出力（writeback 対象）として扱われる．また，読み出しの発生する大域変数が入力として扱われる（B）と同じく，書き込みが発生する大域変数も関数の出力として扱われる．例 4-4 に，これら出力に対するアセンブリ解析例を示す．

例 4-4：レジスタ・大域変数出力のアセンブリ解析

```
            : :          :      :      :
out2:
            .uaward  400
            : :          :      :      :
Change_out2:
            !#PROLOGUE# 0
            : :          :      :      :
            st          %i2, [%i1+%lo(out2)]
            : :          :      :      :
            ret
            restore    %g0,   %l3,   %o0
            : :          :      :      :
No_Return:
            !#PROLOGUE# 0
            : :          :      :      :
            ret
            nop
```

例 4-4 に示した関数ラベルのうち，Change_out2 が大域変数による出力 out2 と返り値による出力%o0 の 2 つを持つ関数だが，No_Return は何も出力を返すことのない関数である．大域変数 out2 には値 400 が格納されており，また関数 Change_out2 では大域変数へのストア命令 st が使用されている．(B) で述べた大域変数からの読み出し検出と同じように，ローカルレジスタ%lo を用いたストア命令を関数の出力として検出するようにした．

また，関数の返り値ももちろん出力である．SPARC では従来から遅延スロット [11] という概念があり，これにより関数復帰命令 ret を検出しても，先にその 1 つ後にある命令を実行する仕様となっている．本解析では関数 Change_out2 の場合のように関数復帰命令 ret の次命令で出力レジスタ%o0 をデスティネーションとして使用している関数は返り値ありとし，関数 No_Return の場合のように空命令 nop などデスティネーションが存在しない命令がある場合を返り値なしとした．

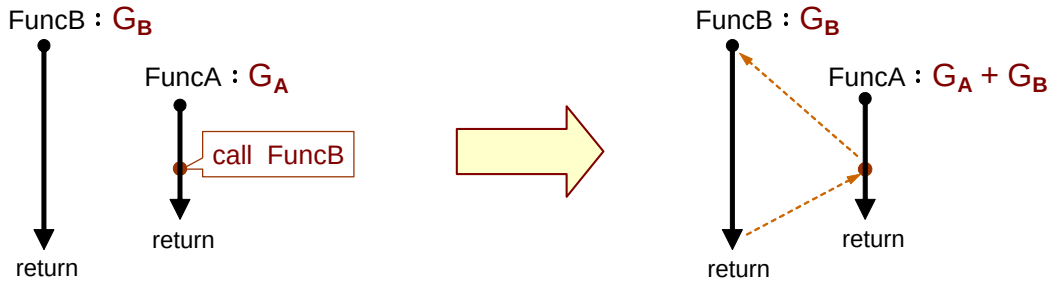

図 17: 関数呼び出しとメモ化利得算出モデル

4.1.3 利得と関数呼び出し

ここまで説明した関数命令数とオーバーヘッドを見積もることで，関数ごとにメモ化利得を算出することができる．一方で， 自動メモ化プロセッサは関数の多重再利用を可能としているためメモ化利得の算出には呼び出された関数の親子関係を考慮する必要がある．そこで本解析では**関数呼び出しを検出した場合，呼び出し先のメモ化利得を加算する**ものとした．このメモ化利得の算出をモデル化した例を**図 17** に示す．

図 17 は関数呼び出しにより FuncA のメモ化利得が変化する場合を表わしている．まず，関数 FuncA および FuncB のメモ化利得がそれぞれ G_A および G_B であるとする．ここで関数 FuncA は内部で関数 FuncB を呼び出しており，FuncA の計算再利用が成功した場合は同時に FuncB の計算再利用も行われることになる．したがって FuncA のメモ化利得は，自身のメモ化利得 G_A に FuncB のメモ化利得 G_B を足し合わせた G_A+G_B となる．なお，関数呼び出しによるメモ化利得加算を行う場合も前項で説明した（a）～（d）による計上方法にしたがうものとする．

一方で再帰関数の場合，自身を呼び出す回数が定まらない限りメモ化利得のモデル化を行うことはできない．そこで，呼び出す回数の確率を級数的に解くことにより当該関数のメモ化利得を算出した．**図 18** に，再帰関数とそのメモ化利得算出のモデル化の例を示す．

まず，利得算出のモデル化を行う前の関数 FuncA のメモ化利得を G_A とする．図 18 の場合，FuncA の関数呼び出しは（a）で説明した下方分岐区間に属しており，1/2 の確率で呼び出しが発生することになる．そのため，自分自身を呼び出す確率は 1/2 となり，自身を呼び出さずそのまま関数復帰命令を呼び出す確率がもう 1/2 となる．

そして自身を呼び出した後，また自身を呼び出す確率は $1/2 \times 1/2 = 1/4$ となる．こ

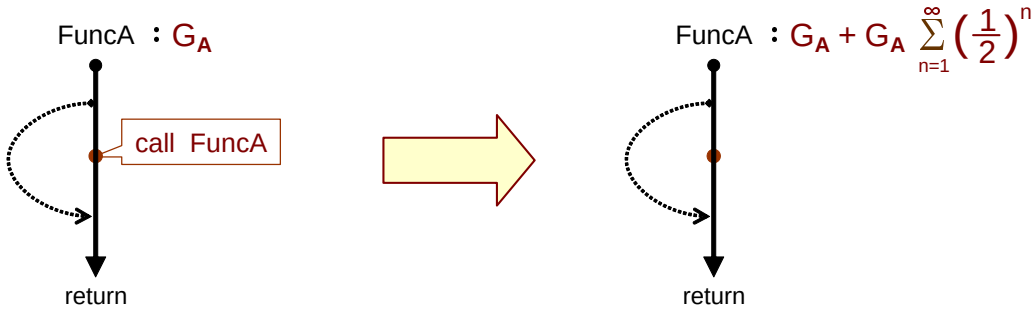

図 18: 再帰関数とメモ化利得算出モデル

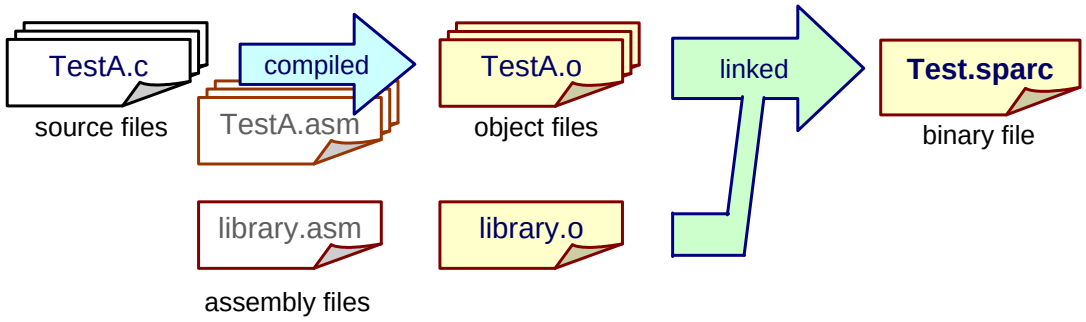

図 19: 実行バイナリ生成までの過程

れを級数的に解くと，関数呼び出し回数の期待値 E_{call} は

$$E_{call} = \frac{1}{2} + (\frac{1}{2})^2 + (\frac{1}{2})^3 \cdots = \sum_{n=1}^{\infty} (\frac{1}{2})^n = 1$$

となる．したがって図 18 の場合，自身を呼び出す回数の期待値は 1 回と定まり，自関数のメモ化利得 G_A を加えることで再起関数の利得を算出することができる．

4.2 解析手順と実装

図 19 に実行バイナリを生成するまでの過程を示す．本章で行う静的解析はすべてアセンブリを対象としている．この理由は，シミュレーション対象となるプログラム（source files）をコンパイルする過程で，アセンブリ（assembry files）がコンパイラによる最適化が行われた後の実行バイナリ（binary file）とほぼ一対一の関係を持っているからである．異なるのはリンカにより条件ラベルや局所大域変数ラベルが削除される程度にとどまる．

前節で説明したアセンブリの静的解析より，メモ化利得といったメモ化のためのヒント情報が得られる．これを元にメモ化すべき関数・すべきでない関数の分類を行い，オーバーヘッド削減・メモ化ヒット率向上による高速化・動的消費電力抑制などを図

例 4-6：メモ化禁止情報の挿入

オリジナルのアセンブリ

```
: :          :       :       :
add          %i0,  %i1,  %i0
call         Func_N, 0
nop
: :          :       :       :
: :          :       :       :
```

メモ化禁止情報挿入後

```
: :          :       :       :
add          %i0,  %i1,  %i0
call         %g0
call         Func_N, 0
nop
: :          :       :       :
```

る．その際，どの関数に対しメモ化制限を行うかというヒント情報をプロセッサに渡すには以下の 2 通りの方法が考えられる．

（方法 1） ヒント情報ファイルを別途生成する

（方法 2） 実行バイナリにヒント情報を付加させる

（方法 1）により実現する場合，自動メモ化プロセッサにはメモ化のヒント情報を格納する領域が必要となる．プログラム中で定義されている関数の数は，あるプログラムでは数十種類であったり別のプログラムでは数千種類であったりとプログラムごとに変動するため，格納領域の上限サイズを大きくとる必要も考えられる．また，その格納領域に対し当該命令区間がメモ化除外対象の可能性があるかどうかを調べるための参照コストも考慮する必要がある．いずれにせよ，実際のプロセッサにおける実装およびヒント情報参照とためのオーバーヘッドコストは大きいと考えられる．

（方法 2）により実現する場合，メモ化除外というヒント情報を意味する命令行の追加，もしくはバイナリ内のメモ化ヒント情報領域を参照させる命令が必要となる．一方で，（方法 1）で述べたヒント情報格納領域およびそれに対する参照コストは不必要となる．また過去の研究では，本稿で提案する自動メモ化プロセッサのためのヒント情報とは異なるのだが，コンパイル時に計算再利用のための情報を埋め込むもの [12] [13] も提案されている．

本手法では，ハードウェアの拡張とそのハードウェアへの参照オーバーヘッドが必要ない（方法 2）による実現を行なった．以降，この実装方法について述べる．

メモ化ヒント情報の実装

例 4-6 に，該当命令区間をメモ化対象から除外することを意味するヒント情報（以降，これを**メモ化禁止情報**と称す）を対象プログラムに組み込んだ場合の例を示す．例 4-6 は関数 Func_N を呼び出している部分を抽出したものを表わしており，左側が Func_N

に対してメモ化禁止情報を**挿入する前**のアセンブリ，右側がメモ化禁止情報を**挿入した後**のアセンブリである．

メモ化禁止情報の挿入前では，加算命令 add の直後に関数呼び出し命令 call が続いているが，挿入後ではその間に call %g0 という命令を追加している．この命令がメモ禁止情報に該当する．ここで%g は SPARC のもつグローバルレジスタのことを指しており，その 1 本目である%g0 にはその値が常に 0 となるよう取り決めがなされている．つまりこの call %g0 という命令は，アドレス 0 番地に存在する関数への呼び出しを意味する命令となる．しかしこの 0 番地は一般にプログラムコードが置かれる領域には含まれず，SPARC の場合もプログラムコードが配置されるのは 10000 番地以降となっていることから，この命令が実際のプログラム中で出現する可能性はない．以上の理由により，以前から存在はしたのだが決して使われることのなかった命令 call %g0 をメモ禁止情報として採用し，自動メモ化プロセッサがこれを解釈してメモ化を行わないようにした．

5 評価

3 章で説明したように，自動メモ化プロセッサに消費電力評価の機能を追加した．そのうえで，計算再利用の成功率に基づいてメモ化中断および再開の動作を動的に行なう機構を実装し評価を行なった．また 4 章で述べたように，静的解析を用いることで実行対象プログラムに対しメモ化禁止情報を挿入し，その評価を行なった．

5.1 動的解析による低消費エネルギー評価

5.1.1 評価環境

シミュレータは単命令発行の SPARC-V8 をベースとし，命令レイテンシは SPARC64-III[14] を参考にした．また 3.1 節で述べた，消費電力算出に使用される様々なパラメータを規定するプロセスルールは，UltraSPARC III 相当の 0.18μm とした．

なお上記のように，評価モデルとしては比較的単純なスカラプロセッサアーキテクチャを採用している．メモ化は従来の ILP などに基づく高速化手法とは大きく考え方の異なる手法であり，スーパスケーラなどの今日一般的となっているアーキテクチャではその有効性が活かしきれないためである．今後は ILP に基づいた高速化手法との併用も考えていく必要があるが，本稿では純粋にメモ化の効果を測定するためこのようなモデルを仮定した．評価に用いたパラメータを**表 1** に示す．

ベンチマークプログラムとして，SPEC CPU 95 および GENEsYs を用い，以下の 4

表 1: シミュレータ諸元

Register 幅	32 bits
Register 本数	72 entries
FloatingPoint Register 幅	64 bits
FloatingPoint Register 本数	32 entries
プロセスルール	0.18 μm
D1 Cache 容量	32 KBytes
ラインサイズ	32 Bytes
ウェイ数	4
レイテンシ	2 cycles
ミス ペナルティ	10 cycles
D2 Cache 容量	2 MBytes
ラインサイズ	32 Bytes
ウェイ数	4
レイテンシ	10 cycles
ミス ペナルティ	100 cycles
Register Window 数	4 sets
Window ミス ペナルティ	20 cycles/set
MemoBuf. 容量	19 kB
RF 容量（RAM）	12 kB
RF ウェイ数	256
RB 容量（RAM）	36 kB
RA 容量（CAM）	25 kB
W1 容量（RAM）	75 kB
RB・RA・W1 ウェイ数	1024
MemoTbl ⇔ レジスタ 比較	32 Byte/cycle
MemoTbl ⇔ キャッシュ 比較	32 Byte/2cycles

種類の場合に対してシミュレーションおよび評価を行なった.

(O) オリジナル（メモ化なし）

(M) 自動メモ化プロセッサ（従来手法）

(E_A) 省エネルギーモデル 1（メモ化中断）

(E_{AR}) 省エネルギーモデル 2（メモ化中断＋再開）

省エネルギーモデル 1・2 となる (E_A) および (E_{AR}) が提案手法である．なお，添字の A はメモ化の中断 (**A**bort)，R はメモ化の再開 (**R**esume) を意味している．

5.1.2 SPEC CPU 95

まず SPEC CPU95（train） を gcc 3.0.2（-O2 -mcpu=supersparc）によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いて評価を行なった．**図 20**，**図 21** および**表 2** に結果を示す．

両図の横軸に示した 101.tomcatv から 146.wave5 までの計 9 つが浮動小数点演算を使用するベンチマークプログラム（CFP）であり，099.go から 147.vortex までの計 7 つが整数演算のみを使用するベンチマークプログラム（CINT）である．

図 20 は各ベンチマークプログラムに要した実行サイクル数を表わしており，左から順に (O) オリジナル，(M) 従来手法，(E_A) 中断手法，(E_{AR}) 中断＋再開手法，における実行サイクル数である．図 21 は図 20 と同じく前述した 4 パターンにおけるプロセッサの消費エネルギーを表わしている．図 21 の凡例は消費エネルギーの内訳であり，下から順にクロック回路（Clock），一次データキャッシュ（D$1），二次データキャッシュ（D$2），演算器 （ALU），レジスタ部（Regfiles），およびメモ化のための機構の各構成要素（MemoBuf，RF，RB，RA，W1）における消費エネルギーの割合を示している．なお，図 20，図 21 では (O) の値ですべて正規化してある．また表 2 は SPEC CPU95 を平均した結果であり，各値はやはり (O) を基準とした増減で表わしている．以降，各手法による結果について述べる．

従来手法

従来手法 (M) では，124.m88ksim や 147.vortex のようにおよそ 20%以上のサイクル数が削減できている場合に，消費エネルギーをオリジナルと同程度もしくはそれ以下に抑えることができている．しかし，SPEC CPU95 は全体で平均削減サイクル数 5.3%，特に CFP に至っては 1.7%となっており，メモ化による効果が得にくいプログラムが多いことが分かる．その結果平均消費エネルギーは全体で 14.6%増となり，比較的現実的な値に抑えられているとはいえ，CFP の多くや CINT の 101.ijpeg，129.compress などメモ化の効果が現れないプログラムにおいて無駄なエネルギー消費が目立っている．

メモ化中断手法

図 21 より，メモ化中断手法 (E_A) では 102.swim や 103.su2cor をはじめ計 8 つのプログラムにおいてメモ化の中断が起きているが，101.tomcatv 124.m88ksim および

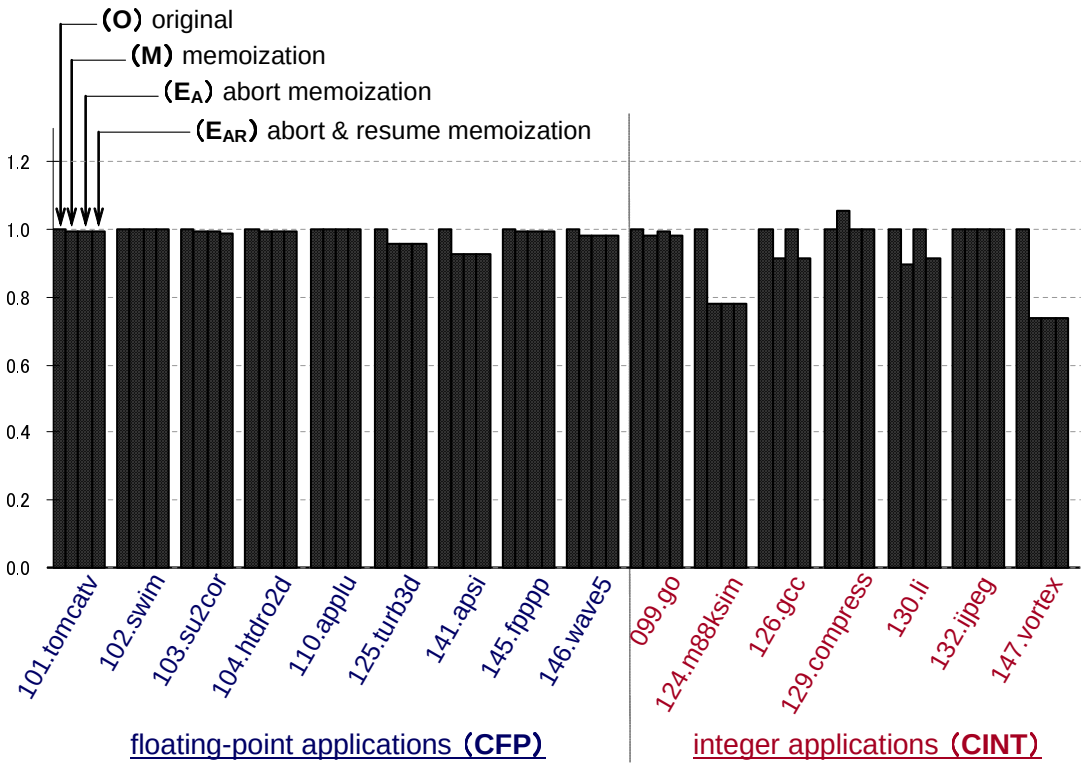

図 20: 総実行サイクル数（SPEC CPU95）

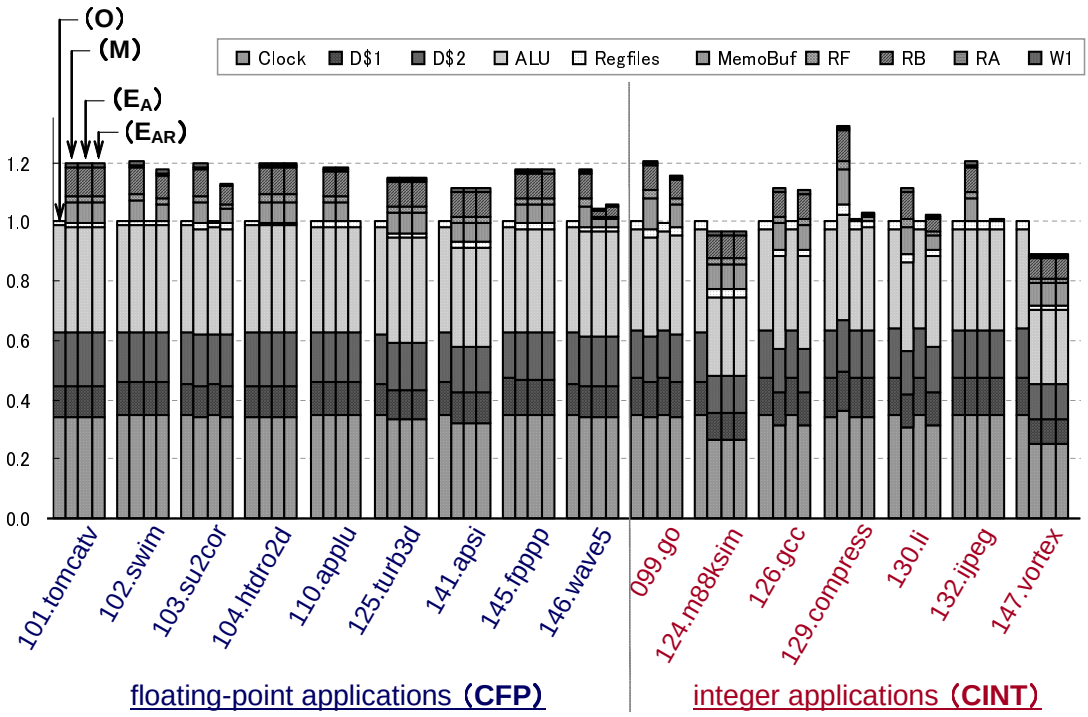

図 21: 総消費エネルギー（SPEC CPU95）

表 2: SPEC CPU95 の結果（平均）

		サイクル数	消費エネルギー
SPEC CPU95 全体	(M) 従来手法	−5.3%	+14.6%
	(E_A) 中断手法	−4.3%	+5.4%
	(E_{AR}) 中断＋再開手法	−5.5%	+8.2%
（ CFP のみ ）	(M) 従来手法	−1.7%	+17.7%
	(E_A) 中断手法	−1.6%	+11.5%
	(E_{AR}) 中断＋再開手法	−1.7%	+13.0%
（ CINT のみ ）	(M) 従来手法	−9.7%	+10.8%
	(E_A) 中断手法	−7.7%	−2.0%
	(E_{AR}) 中断＋再開手法	−10.1%	+2.2%

147.vortex などメモ化中断が起こらず (M) と同じエネルギー消費となっている．

消費エネルギーは全体で 5.4% 増となり，(M) の 14.6%と比べかなりのエネルギー消費の抑制が実現できている．また，中断の起きた 8 つのプログラムの消費エネルギーに限っては，(M) では平均で 19.7%増だったものが，わずか 0.6%増にまで減少した．中断による消費エネルギー抑制が効果的に行われたことが分かる．

一方で全体の平均削減サイクル数は 4.3%となり，従来手法の 14.6%と比べ削減サイクル数は大きく減少してしまった．これは本来メモ化の効果が得られるものまでメモ化中断してしまったことによると考えられる．したがって，(E_A) はエネルギー消費を抑制するが，同時にメモ化による高速化も抑制してしまう可能性の高い手法だといえる．

メモ化中断＋再開手法

最後に中断＋再開手法 (E_{AR}) では，メモ化の再開動作により (E_A) でも高速化が起きていた 124.m88ksim や 147.vortex だけでなく，126.gcc，130.li においても高速化がみられた．特に 130.li に関しては，本来メモ化による高速化の恩恵を受けられる時間区間とメモ化の効果がない時間区間をうまく判定できていると考えられ，(M) で余分に消費していたエネルギーが削減され (O) と同程度のエネルギー消費に抑えられている．

(E_{AR}) による全体の平均削減サイクル数は 5.5%となり，(E_A) の 4.3%と比較しても，従来手法 (M) の高速性を維持できている．また全体での平均消費エネルギーは 8.2%増となり，(M) の 14.6%と比べても消費エネルギーの抑制が実現できている．

以上の結果から，メモ化の再開動作を取り入れることで，中断アルゴリズムだけで

表 3: GENEsYs のパラメータ

交叉率	60.0 %
突然変異率	0.1 %
個体数	50
世代数	25 世代
他のパラメータ	default 値

は損なわれてしまう計算再利用による高速性を維持しつつ，消費エネルギーを抑制できることが分かった．

5.1.3 GENEsYs

次に，汎用 GA ソフトウェア GENEsYs 1.0 [15] を用いて評価を行なった．GENEsYs には，GA のベンチマークや定量的評価に良く用いられる De Jong のテスト関数，巡回セールスマン問題，フラクタル関数などの標準的な関数をはじめとする，24 種の適合度関数が実装されている．

メモ化効果はプログラムの記述方式に依存して変化するため，今回は，メモ化効果の高い適合度関数をメモ化効果が得られやすいように書き換えたもの [16] を用いて評価した．また，交叉アルゴリズムは 2 点交叉とした．

表 3 に GENEsYs の実行パラメータを，**図 22**，**図 23**，および**表 4** にシミュレーション結果を示す．図 20，図 21 と同様に，24 種の各適合度関数の結果はすべてオリジナル (O) の値により正規化したものである．使用した gcc のバージョンやコンパイルオプションなども前項で説明した SPEC CPU95 のものと同じである．

GENEsYs では全般的に大きなメモ化効果が得られており，全 24 適合度関数のうち 6 関数で，総消費エネルギーがメモ化なし (O) の場合より抑えられていることが分かる．また GENEsYs では，特に全体に対して適合度計算量の占める割合の大きい関数，すなわち処理時間が長くかかる関数ほどメモ化の効果が得られることが分かっており GA は処理時間面においても消費エネルギー面においてもメモ化の恩恵を受けやすいプログラムであると言える．以下，各手法による結果を順に追って説明していく．

従来手法

従来手法 (M) の全体における平均削減サイクル数は 18.1% と大きく削減されており，消費エネルギーも平均で 0.2%減となりわずかにではあるがメモ化によりエネルギー削減が起きている結果となった．一方で SPEC CPU95 同様，やはりメモ化による高速

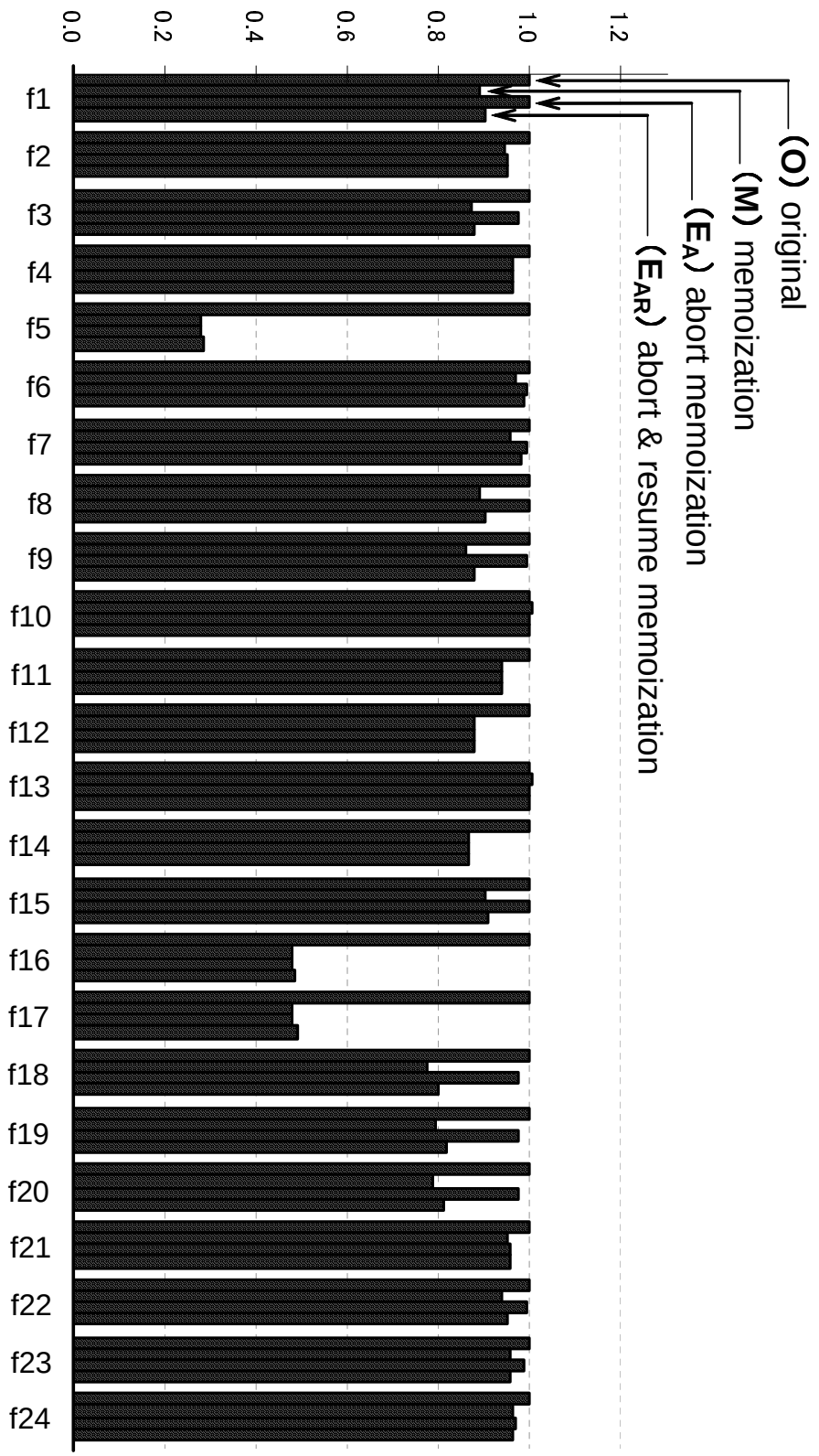

図 22: 総実行サイクル数（GENEsYs）

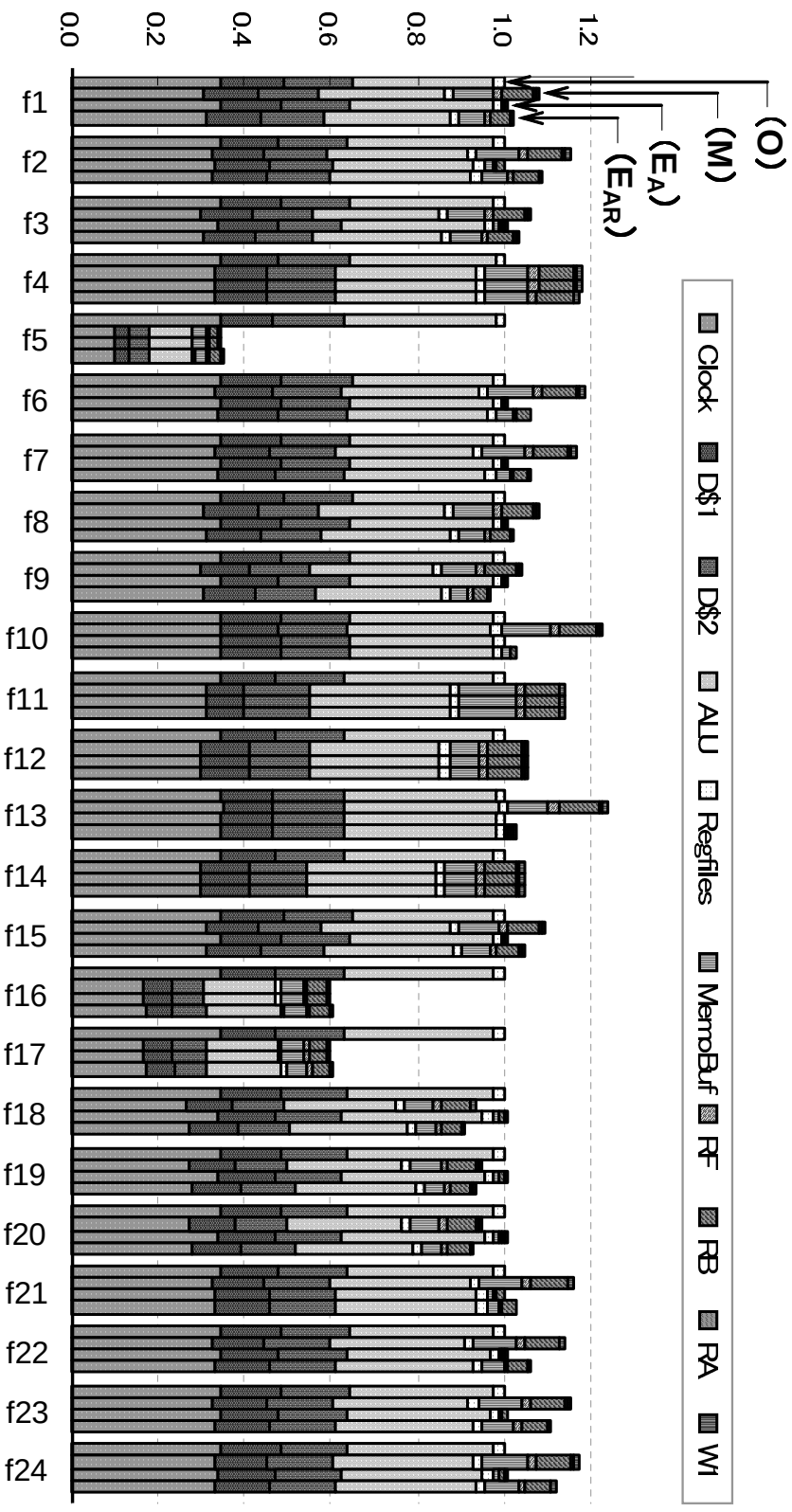

図 23: 総消費エネルギー（GENEsYs）

表 4: GENEsYs の結果

	サイクル数	消費エネルギー
(M) 従来手法	−18.1%	−0.2%
(E_A) 中断手法	−13.7%	−6.5%
(E_{AR}) 中断＋再開手法	−17.3%	−5.1%

化が起きていないものも存在し，いくつかの適合度関数で無駄にエネルギーを消費しているのがわかる．

メモ化中断手法

中断手法 (E_A) では，f1 をはじめ計 15 の適合度関数でメモ化の中断が起きているが，f2，f3，f4，f5，f11，f12，f14，f16，f17 ではメモ化の中断が起こらず (M) と同じ低エネルギー消費となっている．平均消費エネルギーは 6.5% 減となり，メモ化中断の導入によってかなりの消費エネルギーの削減が可能となった．

中断の起きた計 15 の評価関数の消費エネルギーに限れば，(M) では 10.1%増だったのが (E_A) では 0.5%増にまで抑えられており，効果的に消費エネルギーの抑制ができたことが分かる．一方，f18，f19，f20 の 3 関数に関しては，本来高速化が見込めるにもかかわらずプログラムの開始間際でメモ化の中断が起きたため，高速化も消費エネルギー削減もできていない結果となった．このため全体の実行サイクル数削減率は 13.7%となり，(M) の 18.1%と比較すると大きく低下してしまっている．

メモ化中断＋再開手法

中断＋再開手法 (E_{AR}) では，メモ化の再開動作により，メモ化中断手法で大きく高速化できていた f5 や f16，f17 だけでなく，(M) で高速化できているすべての評価関数において，メモ化再開による高速化がみられた．特に f9 や f18，f19，f20 に関しては，SPEC CPU95 の 130.li 同様，定期的にメモ化を再開したことにより，メモ化なし，さらにはメモ化中断手法と比べても消費エネルギーが削減される結果となった．

全体の平均削減サイクル数は 17.3%となり，(E_A) で犠牲となっていた高速性は克服され，より (M) に近い高速性を実現できた．また平均消費エネルギーは 5.1%減となり，0.2%減程度であった (M) と比べでもさらなる消費エネルギーの削減が実現できている．

評価関数ごとに分類してまとめると，まったくメモ化による高速化が得られない関数は f10 および f13 であるが，これらでは 20% 以上の消費エネルギー削減ができており，メモ化機構により生じる無駄な消費エネルギーをほぼ完全に抑制できたと言える．

また，f1，f8，f9，f15 などで顕著であるが，(E_{AR}) が (E_A) よりもエネルギー消費が増大している一方，(E_A) では損なわれている速度向上が (E_{AR}) ではほぼ維持できている．つまり (E_{AR}) の (E_A) に対する消費エネルギー増は，必要な場合にメモ化機構が動作した結果であり，逆に (E_A) はメモ化が必要な場面においてもメモ化機構への電力供給を停止してしまっていることがうかがえる．また f18，f19，f20 のようにメモ化を行うことによって大きくサイクル数が削減され，結果的に消費エネルギーが (E_A) よりも抑制できたものもある．

以上の結果から，(E_{AR}) は (E_A) に比べて対象を無駄なエネルギー消費に絞り，その増加を有効に抑制できていることが分かる．(E_{AR}) ではメモ化効果の高い区間においてのみメモ化機構を動作させることをほぼ実現できており，従来手法 (M) の高速性をほとんど損なうことなく効果的に消費エネルギーの削減を行うことができると分かった．

5.1.4 考察

消費した電力の割合について

オリジナル (O) および従来手法 (M) では電力的な遮断を行なってはいない．また式（2）で示したように，消費電力に対し図 20 や図 22 に示している時間を乗算したものが消費エネルギーであり，これが図 21 や図 23 の結果である．したがって (O) や (M) においては，図 21（図 23）で示した消費エネルギーの割合がそのまま消費電力の示す割合となる．

メモ化のための機構により消費電力は 2 割程度増加しており，そのメモ化機構の消費電力の大部分は MemoBuf および RB（CAM）によって占められていることが分かる．MemoBuf は一次データキャッシュと同様の構成を想定しており容量も小さいがその参照頻度がレジスタに次ぐほどきわめて高いため，一次データキャッシュ以上の電力消費となっている．また RB はサイズおよび参照頻度はそれほど大きくないのだが，ベースが CAM で構成されているためやはり消費電力が大きくなっている．

実行サイクル数と，メモ化機構を含まない部分の消費電力を比較すると多くのプログラムで，サイクル数の減少以上に消費電力の方が減少していることが分かる．これは，計算結果の再利用によりキャッシュミスやキャッシュアクセス自体が減少したことがある．また入出力登録時，命令区間内で書込みが発生した変数はその後キャッシュではなく MemoBuf から参照されることに起因していると考えられる．

提案手法の効果について

SPEC CPU95 では，特に CFP において従来手法 (M) で高速化が起きていないためそもそもメモ化を行う必要のないプログラムが多い．しかし中断手法 (E_A) や中断＋再

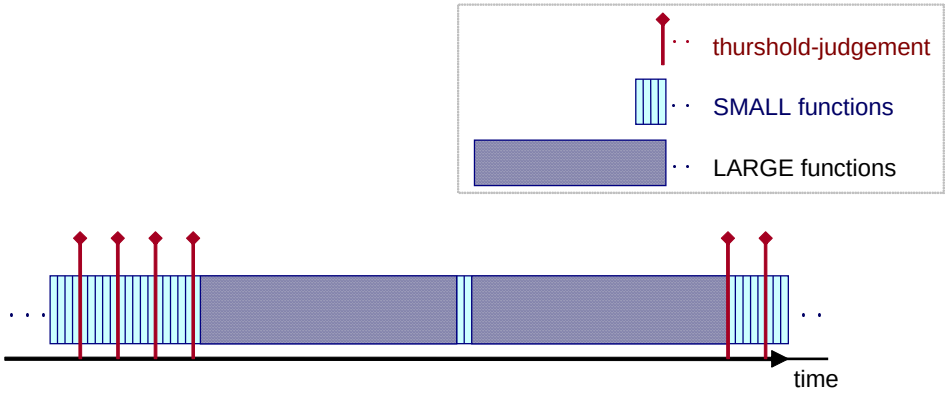

図 24: メモ化中断が発生しない場合のモデル

開手法 (E_{AR}) を用いた場合でもメモ化の中断は発生せず，結果的に無駄にエネルギーを消費してしまっているプログラムが存在する．一般にメモ化の中断が効果的に行われない理由として次の 3 つが考えられる．

(1) プログラム全体に値の局所性がある

(2) 閾値判定が発生する前にプログラムが終了する

(3) 実行サイクル数の非常に大きな関数がある

（1）はつまり，プログラムの一部分だけでなく全体においてメモ化の効果が出やすいことを意味する．SPEC CPU95 では，CINT の 124.m88ksim や 147.vortex などがこれに該当しており，メモ化により実行サイクル数が 20%以上削減できている．

一方，126.gcc や 130.li では従来のメモ化 (M) で実行サイクル数を 10%ほど削減できていたが，プログラムの前半に値の局所性がなかったため (E_A) ではメモ化が中断されてしまっている．一方 (E_{AR}) では，中断後も定期的に値の局所性の有無を確認したことで，このようなプログラムに対しても高速化を図ることが可能となった．

（2）については，閾値判定の間隔を狭めることである程度対応可能だが，メモ化による MemoTbl への登録自体があまり起きていない可能性が高く，そもそもメモ化には適していないと考えられる．

（3）には，おもに 101.tomcatv や 104.hydro2d などの CFP のプログラムが該当する．この（3）に該当するようなプログラム中で行われる閾値判定のモデルを**図 24** に示す．図 24 は，あるプログラム中において実行にかかるサイクル数が比較的小さな関数（SMALL function）およびそれが非常に大きな関数（LARGE function）の 2 種類が呼び出され，関数 5 つの呼び出しごとに閾値判定が行われるモデルを表わしている．たとえば SMALL function でのメモ化ヒット率がきわめて高かった場合，SMALL function

の 5 つ分に当たる時間区間の閾値判定は成功となるが，一方で LARGE function を含んだ区間も成功となる可能性がある．このような場合に，短い関数のメモ化効果から区間全体としても再利用率が高いと判断されてしまう．実際にかかる実行時間やメモ化オーバーヘッドの多くは LARGE function などの大きな関数が占めているため，メモ化の中断が起こらないまま，無駄なエネルギー消費を計上し続けてしまう．

5.2 静的解析による高速化・低消費電力評価

前節の動的解析では，おもに消費エネルギー制御を目的としていた．一方，本節の静的解析ではメモ化により発生するオーバーヘッドの削減および動的消費電力の抑制を図る．

5.2.1 評価環境

シミュレータは前節で用いたものと同じものを用いた．評価の際に用いたパラメータも前節の表 1 とすべて同じである．また前節で述べたように，シミュレータは SPARC64-III の命令レイテンシを参考としており，その命令レイテンシの多くが 1 命令あたり 1cycle となっている．そのため，4 章で述べた命令数やメモ化利得の静的解析を行う際も 1 命令を 1cycle として計上した．ただし，1 命令にかかるレイテンシの大きい乗算命令（4cycles）や除算命令（70cycles）などは個別に計上している．また MemoTbl との入力比較コストも表 1 に基づき，1 入力あたり 1cycle ないし 2cycle として計上した．出力書き戻しコストはシミュレータの仕様と同じく 1 入力あたり 1cyle とした．

なお，ユーザ関数が静的解析の対象であり 4 章で説明したメモ化禁止情報の挿入対象なのだが，ユーザ関数はライブラリ関数を呼び出す場合もある．そこで，あらかじめシミュレータが用いている環境のライブラリ関数のメモ化利得を求めておき，その値を使用することにした．

5.2.2 SPEC CINT 95

SPEC CINT 95 の（train）を gcc 3.0.2 （-O2 -mcpu=supersparc）によりコンパイルし，途中で生成されるアセンブリを解析しメモ化のためのヒント情報を加えた後，スタティックリンクにより生成したロードモジュールを用いて評価を行なった．

理想性能の評価

静的解析による評価を行うにあたり，その前にまず，提案したメモ化禁止情報を取り入れた場合最大でどれくらいの性能が見込めるかを調べておく必要がある．そこで，まず実行対象プログラムに対して事前にシミュレーションを行い，関数ごとの計算再利用成功回数を把握する．その上で，計算再利用が 1 度も成功しなかった関数すべて

に対してメモ化禁止命令を挿入し，再度シミュレーションを行なった．なお，メモ化禁止情報を取り入れたことで，従来の MemoTbl に比べて無駄な入出力情報が登録されなくなるため，メモ化ヒット率向上が期待される．そこで，従来の MemoTbl サイズによる事前シミュレーションだけでなく，より MemoTbl に対して入出力情報を格納できるモデルとして，2 倍，4 倍，8 倍の MemoTbl サイズの場合による事前シミュレーションを行なった．評価を行なったのは以下の 6 種類である．

(O) オリジナル（メモ化なし）
(M) 自動メモ化プロセッサ（従来手法）
(S_{i1k}) 理想値 A（メモ化禁止情報を付加）
(S_{i2k}) 理想値 B（メモ化禁止情報を付加）
(S_{i4k}) 理想値 C（メモ化禁止情報を付加）
(S_{i8k}) 理想値 D（メモ化禁止情報を付加）

(S_{i1k}) はメモ化禁止情報を従来手法で計算再利用が一度も成功しなかった関数に対し付加した場合であり，(S_{i2k})，(S_{i4k}) および (S_{i8k}) が，それぞれ MemoTbl サイズが従来の 2，4，8 倍とした場合でも計算再利用が成功しなかった関数に対してメモ化禁止情報を付加した場合である．これらのシミュレーションによる結果を**図 25** に示す．

図 25 は各ベンチマークプログラムに要した実行サイクル数を表わしており，左から順に (O) オリジナル，(M) 従来手法，そして (S_{i1k}) 理想値 A，(S_{i2k}) 理想値 B，(S_{i4k}) 理想値 C，(S_{i8k}) 理想値 D である．凡例は，その実行サイクル数の内訳を表わしており，下から順に命令実行（exe），メモ化を行う際に発生するオーバーヘッドである，MemoTbl とレジスタとの比較コスト（reg），MemoTbl とキャッシュとの比較コスト（mem）そして計算再利用成功時の書き戻しコスト（writeback），メモ化禁止情報のフェッチにかかるオーバーヘッド（call %g0），およびキャッシュやレジスタウィンドウにおけるミスペナルティ（D$1，D$2，window）である．なお，命令実行（exe）からは，重複を避けるためメモ化禁止情報のフェッチ（call %g0）の分を省いて表示した．これらはすべて (O) の値で正規化してある．以降，各手法による結果の詳細について述べる．

従来手法

従来手法 (M) では，129.compress のプログラムでレジスタおよびメモリとの入力比較オーバーヘッド（reg，mem）がかなり大きくなっていることが分かる．一方，その命令実行時間には変化がないことから，行われた入力比較が結果的に無駄となったことが分かる．他にこの入力比較オーバーヘッドの占める割合が比較的大きなプログラ

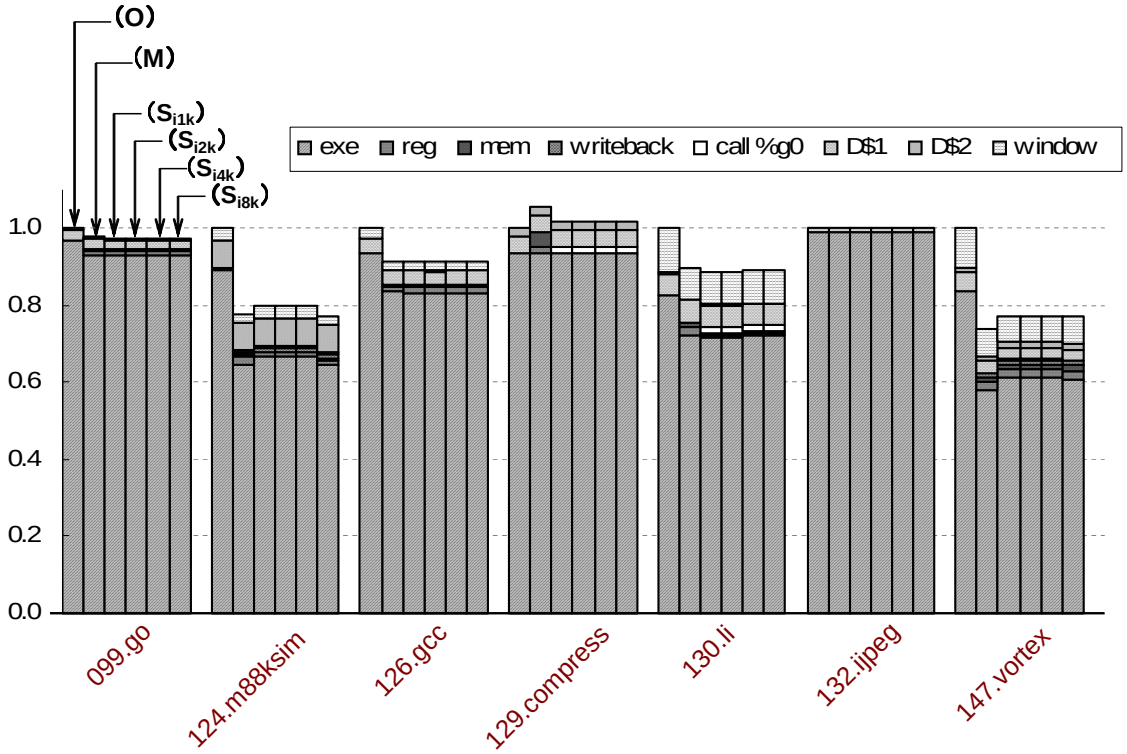

図 25: 総実行サイクル数（SPEC CINT 95）

ムとしては 124.m88ksim や 130.li，147.vortex が挙げられるのだが，それ以上に命令実行時間（exe）が大きく削減されているため，入力比較が無駄ではなかったといえる．

理想値 A，B，C，D

まず理想値 A(S_{i1k}) では，129.compress のように (M) で無駄となっていた入力比較がほぼ 0 となっており，代わりにメモ化禁止情報のフェッチによるオーバーヘッド（call %g0）が発生している．しかし，従来の入力比較オーバーヘッドがこのメモ化禁止情報分のを大きく上回っていたため，メモ化なし (O) までとはいかないが，かなりのオーバーヘッドの抑制ができていることがわかる．

また 130.li では，少しではあるが (M) に対してさらなる高速化が起きている．これは，メモ化禁止情報によるコストが増えた反面，命令実行にかかる時間が削減されていることから，メモ化ヒット率が上昇し計算再利用の成功回数が増加したと考えられる．

一方，124.m88ksim や 147.vortex などではメモ化禁止情報を挿入した結果，(M) に比べ命令実行が増加してしまった．メモ化ヒット率が下がり，計算再利用の成功回数が減ったことが考えられる．これらについては次項で考察を行う．

理想値 B(S_{i2k})，C(S_{i4k})，D(S_{i8k}) についても大部分が (S_{i1k}) の結果とほぼ一緒である

が，124.m88ksim では大きな変化が見られる．(S_{i1k})，(S_{i2k}) および (S_{i4k}) では従来手法 (M) に比べ命令実行が増加してしまったのに対し，(S_{i8k}) では (M) と同等の命令実行となっている．また入力比較コストが削減されているため，124.m88ksim では (S_{i8k}) の総実行サイクル数が最も小さい結果となった．

以上より，各ベンチマークプログラムにおける (M) を含めた (S_{i2k}) や (S_{i4k}) の結果の中から，その実行サイクル数が最も小さい値となった場合を理想値 (S_i) とする．その理想値との比較を含めた静的解析の評価を行う．

静的解析による性能評価

前述した結果の理想値に近づけるよう，静的プログラム解析のみからメモ化禁止情報の付加対象を決定し，シミュレーションを行なった．メモ化禁止情報の付加対象は，4 章で説明した関数ごとのメモ化利得に対して閾値を設け，この閾値をパラメトリックに扱うことで決定した．オリジナル (O)，従来手法 (M) そして前述した理想値 (S_i) に加えて，静的解析で求めたメモ化利得をに対する閾値として，新たに以下の 6 種類によるシミュレーションおよび評価を行なった．

(S_0) 閾値 0 （メモ化利得が 0 以下の関数をメモ化禁止）
(S_5) 閾値 5 （メモ化利得が 5 以下の関数をメモ化禁止）
(S_{10}) 閾値 10 （メモ化利得が 10 以下の関数をメモ化禁止）
(S_{15}) 閾値 15 （メモ化利得が 15 以下の関数をメモ化禁止）
(S_{20}) 閾値 20 （メモ化利得が 20 以下の関数をメモ化禁止）
(S_{25}) 閾値 25 （メモ化利得が 25 以下の関数をメモ化禁止）

これらに対するシミュレーション結果を**図 26**，**図 27** および**表 5** に示す．もちろん，各値はメモ化なし (O) の値により正規化した．以下，各結果について述べる．

閾値 0 (S_0) の場合は，すべての関数で従来のメモ化 (M) とほとんどもしくはまったく同じ実行サイクル数となった．これは，メモ化利得が 0 以下となる関数はほとんど存在せず，バイナリに対してメモ化禁止情報が付加されなかったためである．そのため，消費エネルギーもまたほぼ一緒となっている．

閾値 5 (S_5) の場合は，126.compress でメモ化禁止情報の影響が見られる．メモ化禁止情報によるオーバーヘッドは増加したがこれにより，従来 (M) におけるレジスタやメモリでの入力比較コストが削減されているため，少しではあるがオーバーヘッドの抑制効果を確認できた．しかし依然として，129.compress 以外の関数では (M) および (S_0) に対してほぼ横ばいとなる結果となった．以上の結果より，(S_5) の SPEC CINT 全体での平均実行サイクル数は 9.9%減となり，(M) の 9.7%減に比べわずかに高速化さ

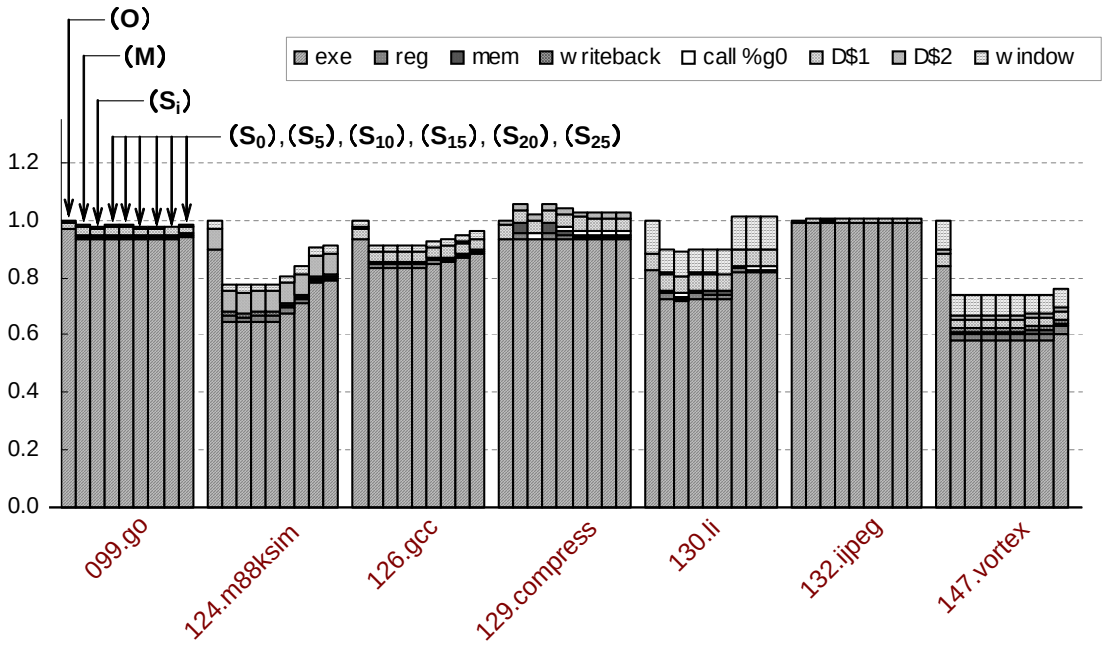

図 26: 総実行サイクル数（SPEC CINT 95）

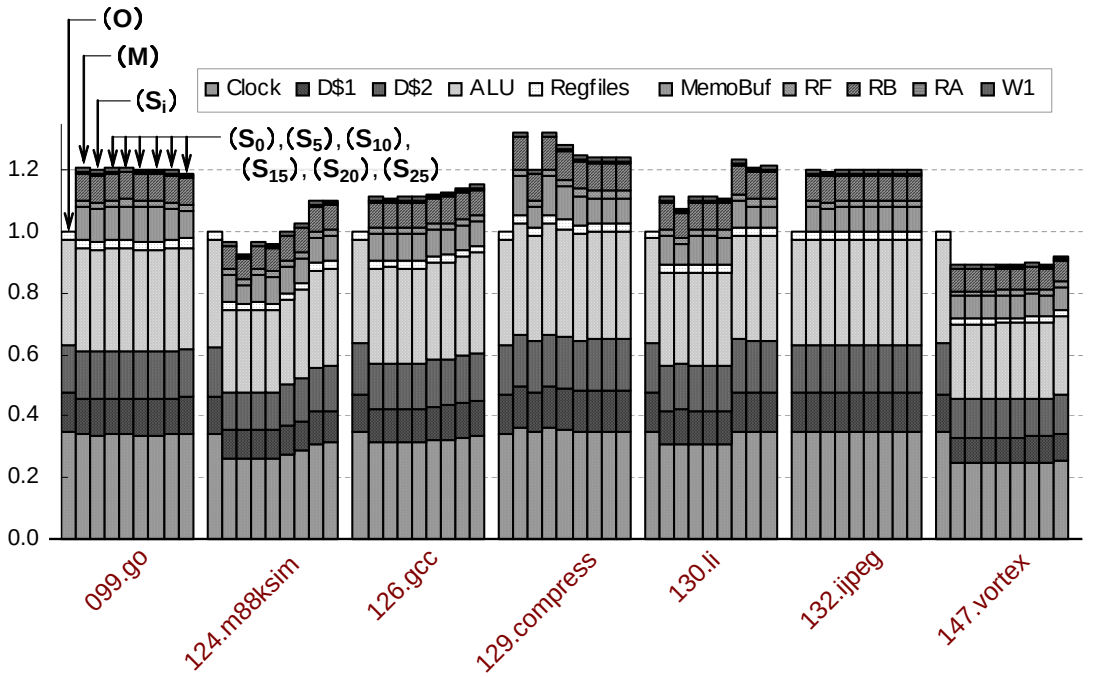

図 27: 総消費エネルギー（SPEC CINT 95）

表 5: SPEC CINT 95 の結果（平均）

	手法	サイクル数	消費エネルギー
(M)	従来手法	−9.7%	+10.8%
(S_i)	理想値	−10.8%	+8.0%
(S_0)	閾値 0	−9.7%	+10.8%
(S_5)	閾値 5	−9.9%	+10.2%
(S_{10})	閾値 10	−9.5%	+10.4%
(S_{15})	閾値 15	−7.2%	+12.7%
(S_{20})	閾値 20	−6.0%	+13.6%
(S_{25})	閾値 25	−5.3%	+14.1%

れた．またその消費エネルギーも，(M) では 10.8%増であったが (S_5) では 10.2%増となり，多少は抑制することができたといえる．

閾値 10 (S_{10}) となると，129.compress で生じていたメモ化によるオーバーヘッドはさらに削減される．一方で， 124.m88ksim や 126.gcc ではメモ化禁止情報の付加により，本来メモ化の恩恵を受けるはずであった関数をメモ化対象外と判定した可能性が高く，命令実行にかかるサイクル数が増加した，その結果，実行サイクル数は平均で 9.5%減となり (M) の 9.7%減と比べ悪化する結果となった．

残りの閾値 15 (S_{15})，閾値 20 (S_{20}) および閾値 25 (S_{25}) の場合では，より多くの関数に対してメモ化禁止情報が付加されるようになる．そのため 129.compress のようにメモ化によるオーバーヘッドが大きいプログラムだけでなく，もともとメモ化により高速化されていたプログラムの多くが閾値上昇ともに従来 (M) にメモ化禁止情報のオーバーヘッドを加えた結果へと漸近する．そのため，(S_{15})，(S_{20})，(S_{25}) における平均実行サイクル数および消費エネルギーは，どれも (M) と比べて大きな性能低下がみられた．

以上の結果より，(S_5)，(S_{10}) の，メモ化利得に対する閾値 5 および 10 あたりが静的解析を行なった上での最良の結果といえるのだが，そもそも 129.compress では (S_{25}) などの高い閾値の場合でも理想値 (S_i) に比べてオーバーヘッドを抑制しきれていない．また，閾値を少しでも上昇させればメモ化の効果を打ち消してしまう可能性があることからも，今回提案したメモ化利得とそのメモ化禁止情報の付加方法が万能ではないことがいえる．

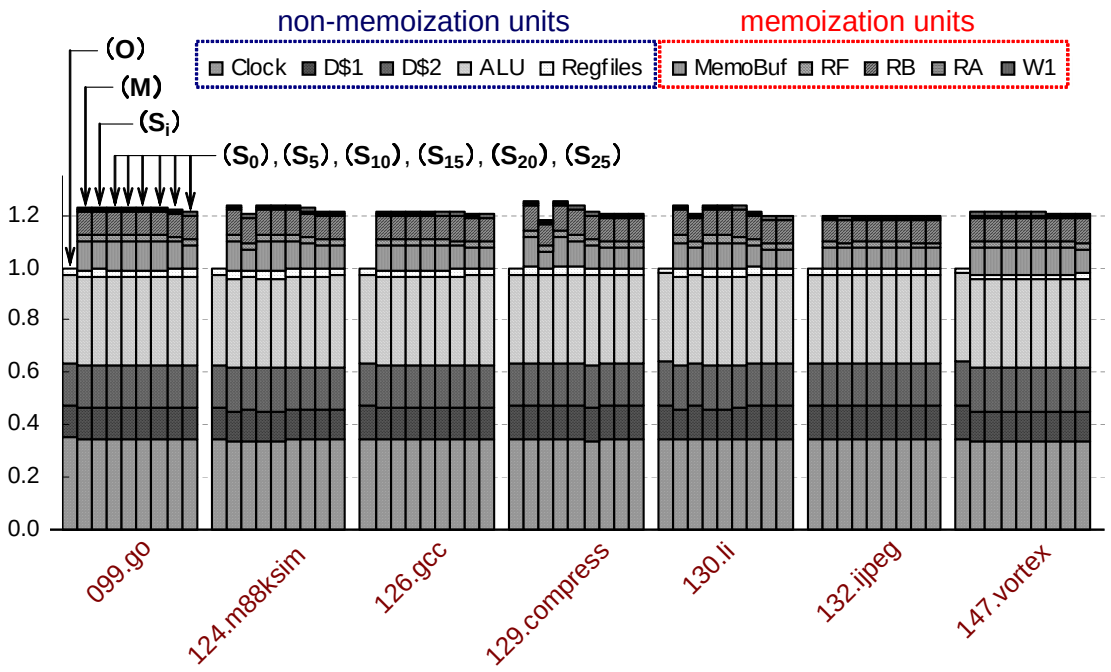

図 28: 消費電力の内訳（SPEC CINT 95）

5.2.3 考察

動的消費電力・静的消費電力について

式（2）および前節で説明したように，消費エネルギーを実行時間で除算することによりプロセッサの消費電力が求まる．ここでは，メモ化禁止情報によってメモ化機構の消費電力をどれほど削減することができたかを考察する．SPEC CINT におけるでの消費電力の内訳を**図 28** に示す．なお図 28 は，図 26 の実行サイクル数で図 27 に示した消費エネルギーの結果を除算したものと同じであり．また凡例である消費電力の内訳も図 27 と同様であるため省略する．

前項で述べたように，147.vortex のなどではキャッシュアクセスが減少したため，わずかな消費電力の変化がみられるものもあるが，ほぼすべてのプログラムにおけるメモ化機構を含まない分の消費電力にはあまり大きな変化は見られない．

さて，従来手法 (M) と理想値 (S_i) とを比べてみると，124.m88ksim や 129.compress, 130. などでメモ化機構の総消費電力に変化が出ている．その内訳を見てみると，メモ化機構の消費電力のなかで大きな割合を占めていた MemoBuf の消費電力が大きく下がっていることが分かる．これはメモ化禁止情報を取り入れたことで，(M) で発生していた MemoBufTbl に対しての頻繁なアクセスが大きく抑えられたことに起因しており，反対に RF，RB，RA，W1 は (M) の段階でもともと頻繁にアクセスが発生してい

なかったためである．つまり，MemoBuf の電力は動的消費電力が大部分を占めており，それ以外のユニットは静的消費電力が大部分を占めていると考えられる．

メモ化禁止情報の効果について

図 25 の理想値による結果では，メモ化すべきでない関数を把握してメモ化禁止情報を付加したにもかかわらず，いくつかのプログラムで理想値 (Si1k)〜(Si8k) の結果がオリジナル (O) のものより悪くなったものがある．この理由として以下の 2 つが考えられる．

(1) MemoTbl に登録されることのない関数にメモ化禁止情報を付加した

(2) メモ化禁止情報により TSID パージの発生頻度が低下した

まず（1）について説明する．MemoTbl に登録されない関数としては，たとえば read 関数などのシステムコールや，入力や出力を多く持つ関数などが存在する．前者の関数は，必ず入力待ちが発生するためメモ化を行うことは困難であり，また後者の関数は，エントリ幅などの制約から MemoBuf に入出力情報を登録するごとができないため，自動メモ化プロセッサはこれらの関数をメモ化対象から除外している．そして 2 章で説明したように，これらの関数を呼び出す親関数もまたメモ化を行うことはできない．したがって（1）の場合には，メモ化禁止情報を読み出すためのオーバーヘッドのみが増加することになる．ただし，一方で従来メモを行うことができないと判断されるまで起きていた MemoBuf アクセスは 0 回となるため，動的消費電力が削減される可能性はある．

次に（2）の場合について説明する．2 章で説明したように MemoTbl 内のエントリの削除方法には TSID パージと RFID パージがあるのだが，このうち TSID パージは MemoTbl のアクセスの際に行われる．

従来では，計算再利用が成功しない関数（以下，関数 **F_not**）をメモ化した際に TSID パージが発生しており，これにより，計算再利用が成功する可能性のある関数（以下，関数 **F_hit**）の古いエントリも同時に削除していた．しかしメモ化禁止情報を取り入れたことで，F_not をメモ化する際に発生していた TSID パージがなくなり，その結果 F_hit の古いエントリは削除されにくくなる．したがって F_hit の全入出力情報を削除する F_hit に属するすべての入出力情報を削除する RFID パージが発生しやすくなり，プログラム全体での計算再利用の成功回数が下がったと考えられる．

（2）で発生しうる問題を解決するためには，たとえばメモ化禁止情報の検出に応じて TSID パージ間隔を狭めたり，またメモ化する際ではなく，ある一定の関数呼び出しごとにパージを発生させる，といった対応が考えられる．以上で述べたようなパー

ジを実現した上で，改めてメモ化禁止情報による効果を確認するのがまず課題となる．

5.3 両解析手法に対する考察

本稿の 5.1 節で評価を行なった，プログラム実行中の計算再利用率に対する動的解析では，SPEC CFP 95 ベンチマークでは消費エネルギーをうまく抑制しきれなかった場合もあったが，その他の CINT や GENEsYs ベンチマークを含め，多くのプログラムで目的であった消費エネルギー制御の効果が出ていることを確認できた．当然だがこれは，解析の対象としていた『関数呼び出しの発生回数』および『計算再利用の成功回数』というメモ化の効果を表わす具体的な指標が正しく取得され，そしてこれらの値がエネルギー制御のために効果的に使用されたからである．

一方 5.2 節で評価を行なった，プログラム中の各関数に対するメモ化利得を求める静的解析では，ある閾値を設けることでオリジナルよりわずかに良い結果とはなったが，目的であったオーバーヘッドの抑制そしてメモ化ヒット率の向上による効果は期待したほど得られなかった．これは，前項で述べたパージ間隔の改良も前提にあるのだが，これに加えて，静的解析の行程全体があまり厳密でなかったことが原因にある．

静的解析が厳密でなかった点としては，静的に算出されたメモ化利得が動的実行されたものと比べてどうしても誤差が発生してしまうこと，メモ化利得は計算再利用が成功した場合を前提とした値であり，関数ごとのメモ化ヒット率をほとんど考慮していないこと，実行対象とするプログラムに応じたメモ化利得の閾値を定めることが困難なこと，などが挙げられる．もちろん，本稿で提案した静的解析のみでなく，様々な解析方法による評価を今後行なっていき，より厳密な静的解析方法のモデルを見つけるということも考えられるが．いずれにせよ，多くの工程を静的解析のみで行なった場合，動的解析のように大きな解析効果を得ることは難しいと考えられる．

6 おわりに

本稿では，計算再利用技術に基づく自動メモ化プロセッサに対し，その消費電力をアーキテクチャレベルで評価した．命令区間の入出力を記憶するための CAM を含むメモ化機構による消費電力増加は 2 割ほどであった．また消費エネルギーでは，SPEC CPU95 で平均 14.6%の増加，GENEsYs で平均 0.2%程度の削減となることが確認できた．このことから，実行するプログラムにおけるメモ化の効果の違いによりエネルギーが増加することもあるが逆にわずかにエネルギーを削減することもできると分かった．

次に，まず計算再利用率を動的に解析して得られた情報をもとにメモ化機構を停止・

再作動させ，メモ化機構のエネルギー消費を抑制するモデルを実装した．シミュレーションにより評価したところ消費エネルギーは，SPEC CPU95 で平均 5.4% の増加，GENEsYs で平均 6.5% の減少となり，かなりのエネルギー抑制が可能であることが分かった．また平均削減サイクル率は，メモ化機構を動作し続けた場合は SPEC CPU95 で 5.3%，GENEsYs で 18.1%であったのに対し，メモ化機構を停止・再作動させた場合にそれぞれ 5.5%，17.3%となり，本来メモ化によって得られる高速化の効果をほぼ維持しつつ，省エネルギー化を図ることができた．

また，計算再利用の成功の際にどれだけ恩恵があるのかを示す指標としてメモ化利得を定義し，実行対象のプログラムごとにこれを算出する静的解析を行なった．このメモ化利得が一定値に満たなかった関数に対し，メモ化対象から除外する意味を持つ情報をバイナリに付加させた．SPEC CINT 95 による評価では，従来手法の場合平均実行サイクル率は 9.7%削減，平均消費エネルギーが 10.8%増であったが，静的解析およびメモ化対象からの除外情報を導入したことでそれぞれ 9.9%削減および 10.2%増となり，わずかではあるが，オーバーヘッド削減による高速化および消費電力の削減を実現できた．しかし，メモ化によるオーバーヘッド抑制にはまだまだ改善の余地があるため，メモ化利得に代わる選別方法を見出すことが今後の課題となる．

また，たとえば本稿で提案した計算再利用率の動的解析のみでは消費エネルギーを抑制できなかったプログラムに対しても，静的解析で取得したメモ化のヒント情報を用いることでエネルギー抑制が可能となるのではないかと考えられる．つまり，静的解析を動的解析を組み合わせることによる消費エネルギー制御やさらなる速度性能の向上なども今後の課題として挙げられる．

謝辞

本研究の為に多大な御尽力をいただき，日頃から熱心な御指導を賜った名古屋工業大学の津邑公暁准教授，ならびに奈良先端科学技術大学院大学の中島康彦教授に心から御礼申し上げます．また，本研究に対しての熱心なご検討・御協力をいただきました，名古屋工業大学の松尾啓志教授に深く感謝致します．これに加え，たびたびご検討・助言をしていただいた同大学の齋藤彰一准教授ならびに松井俊浩助教授にも深く感謝致します．そして最後となりましたが，数多くの助言・協力をしていただいた松尾・津邑・齋藤研究室メンバーの皆様に対しても深甚なる感謝の意を表します．ありがとうございました．

参考文献

[1] Norving, P. *Paradigms of Artificial Intelligence Programmimg*, Morgan Kaufmann (1992).

[2] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H., Nakashima, Y. : Design and Evaluation of Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp.245-250 (2007).

[3] Yang, J., Gupta, R. : Energy-Efficient Load and Store Reuse, *Intl. Symp. on Low Power Electronics and Design*, pp.72-75 (2001).

[4] Paul, R. : *SPARC Architecture, Assembly Language Programming and C*, Prentice-Hall (1999).

[5] 津邑公暁，中島康彦，五島正裕，森眞一郎，富田眞治：汎用 CAM を用いた既存プログラム高速化手法の提案，第 1 回リコンフィギャラブルシステム研究会 論文集，電気情報通信学会, pp.21–26 (2003).

[6] 島崎裕介，池内康樹，津邑公暁，中島　浩，松尾啓志，中島康彦：自動メモ化プロセッサの消費エネルギー評価，情報科学技術レターズ（FIT2007），pp.51–54 (2007).

[7] Brooks, D., Tiwari, V. and Martonosi, M. : Wattch: A Framework for Architectual-Level Power Analysis and Optimizations, *Proc. 27th Annual Intl. Symp. on Computer Architecture*, pp.83–94 (2000).

[8] D.Burger and T.M. Austin : The SimpleScalar Tool Set, Version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin-Madison Computer Sciences Department (1997).

[9] Wilton, S.J. and Jouppi, N.P. : An Enhances Access and Cycle Time Model for On-chip Caches, Techunical report, DEC Western Research Laboratory (1993).

[10] Powell, M., Yang, S.-H., Falsafi, B., Roy, K. and Vijaykumar, T. N. : Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories, *Proc. Intl. Symp. on Low Power Electronics and Design* (*ISLPED*), pp.90–95 (2000).

[11] Sun Microsystems UltraSPARC^{TM}-II*i* CPU Module Data Sheet, (1998)

[12] Huang, J. and Lilja, D. J. : Exploring Sub-Block Value Reuse for Superscalar Processors, *PACT* (2000).

[13] Connors, D. A., Hunter, H. C., Cheng, B. and Hwu, W. W. : Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse, *9th ASPLOS*, pp.222–233 (2000).

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

[15] Bäck, T. : GENEsYs 1.0. Software distribution and install notes (1992)

[16] 鈴木郁真，池内康樹，津邑公暁，中島康彦，中島 浩：再利用による GA の高速化手法，情報処理学会論文誌：コンピューティングシステム， Vol.46， No.SIG 16（ACS 12）, pp.129–143 (2005).