修士論文

コンパイラによるプロセッサの リーク電力削減手法

48-086309 **薦田　登志矢**

指導教員　中村　宏 准教授

2010 **年** 1 **月**

東京大学大学院情報理工学系研究科システム情報学専攻

概要

近年，マイクロプロセッサにおけるリーク電力の増大が大きな問題となっている．本稿では，特にアプリケーション実行時における演算器のリーク電力を削減することを目標として，演算器へのパワーゲーティング適用手法を提案する．演算器におけるパワーゲーティングにおいては，スリープ/ウェイクアップのモード切り替えに伴なうエネルギーオーバーヘッドが問題となる。このエネルギーオーバーヘッドを小さく抑えつつ大きなリーク電力削減を達成するためには，演算器に生じる時間的に細粒度な空き時間を正確に予測しながら，演算器のスリープを制御する手法が不可欠である．本研究では，こうしたスリープ制御手法としてコンパイラを用いた空き時間予測に基づくスリープ制御手法を提案する．本手法の中では，コンパイラによってコード内の命令を解析し，演算器に生じる空き時間の長さを予測する．提案するコード解析アルゴリズムは分岐命令および関数を跨いだ解析を行うことができ，リーク電力削減に有効な長い演算器の空き時間をより多く検出できる．また，キャッシュミスが頻発するためコンパイラによる空き時間予測が有効でないアプリケーションへの対策として，キャッシュミス検知による動的なスリープ制御手法をコンパイラによる静的な制御手法と併用するハイブリッド手法を提案する．シミュレータを用いた評価の結果，提案手法によって，理想的にスリープ制御を行った場合に近い大きな演算器のリーク電力削減を達成できることが分かった．

キーワード　Microprocessor，Leakage Power，Power Gating，Compiler

目次

第 1 章

はじめに

1.1 プロセッサの消費電力の問題

現在，マイクロプロセッサはコンピュータシステムの中心的な部品として社会のあらゆる場面で使用されている．例えば，インターネットを支えるサーバシステム，携帯電話の端末，自動車の制御システムなどその応用範囲はますます広がっている．

このようなマイクロプロセッサの普及を支える原動力となってきたのが，半導体技術の向上である．図 1.1 は，ムーアの法則として広く知られているトランジスタ集積数の年代変化を表すグラフである．横軸が年代，縦軸がチップ上のトランジスタ数を示している．このように，マイクロプロセッサは指数関数的なトランジスタ数の増加とそれに伴なう性能向上を達成してきた．一方，それに伴ない消費電力も急激に増加しており，消費電力はマイクロプロセッサ設計における大きな制約となっている．

本研究は，近年ますます重要になってきたマイクロプロセッサの消費電力削減を目指す．特にマイクロプロセッサの演算器部分のリーク電力に着目し消費電力削減手法を提案する．

1.2 ダイナミック電力とリーク電力

現在のマイクロプロセッサは CMOS 回路によって構成されている．CMOS 回路が消費する電力は二つの成分に大別できることが知られている．１つ目の成分は，ダイナミック電力と呼ばれるものである．これは，トランジスタが ON から OFF，OFF から ON というようにスイッチングを行うことによって消費される電力である．CMOS トランジスタは，スイッチングの際，図 1.2 のように出力側のキャパシタンスにおいて電荷の充放電を行う．このときに，消費されるのがダイナミック電力である．ダイナミック電力とはすなわち，マイクロプロセッサが計算を行うことによって消費される電力である．従来のマイクロプロセッサの消費電力は大部分がこのダイナミック電力であった．

ダイナミック電力を削減するアーキテクチャ技術については，これまでも多くの研究がなされている．図 1.4 に，代表的なダイナミック電力削減技術を示す．

二つ目の成分は，リーク電力と呼ばれるものである．CMOS 回路では電源線からグランド

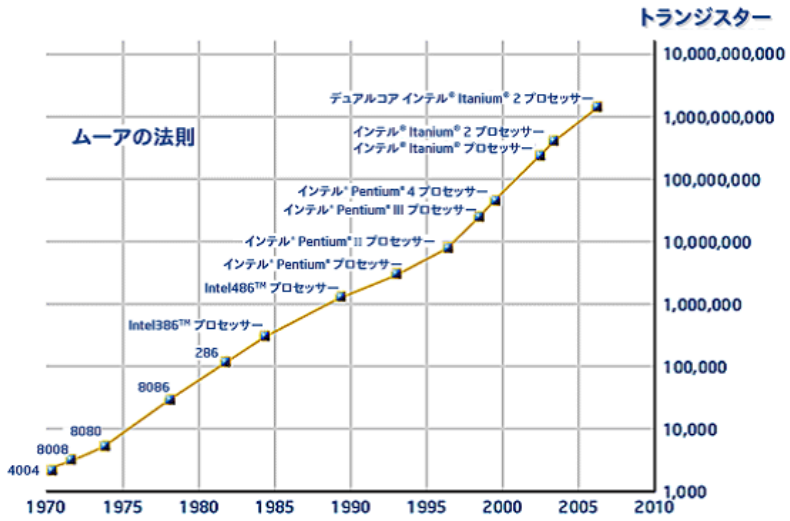

図 1.1. チップ上のトランジスタ数の増加 (Intel 社 Home Page より)

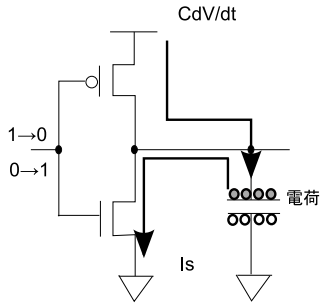

図 1.2. ダイナミック電力の模式図

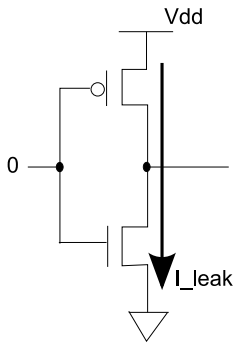

図 1.3. リーク電力の模式図

線へ向かって，電流が漏れ出す現象が存在することが知られている．図 1.3 は，この様子を模式的に表したものである．本来絶縁状態となっているべき電源線 VDD とグランド線の間を，漏れ電流が流れている．この漏れ電流により消費される電力がリーク電力である．リーク電力は，トランジスタのスイッチングにはほとんど依存しない．すなわち，回路に電圧がかかっている状態では常に消費される無駄な電力である．このリーク電力は，従来ダイナミック電力に比べて無視できる程度の大きさであったが，近年のサブミクロン世代のプロセスにおいては無視できない大きさとなっている．

本研究はマイクロプロセッサの消費電力削減を大きな目標としているが，特に近年大きな問題となっているリーク電力を消費電力削減の対象とする．1.3 節ではこのリーク電力の問題について概観し，本研究が取り組む課題を説明する．

手法名	概要
Dynamic Voltage Scaling[13]	性能制約に基づく電圧・周波数制御
Clock Gating[20]	不要なクロックによるスイッチングを止める
Cache Sub-Banking[19]	キャッシュブロックをバンクに分けてアクセス

図 1.4. ダイナミック電力を削減するための技術

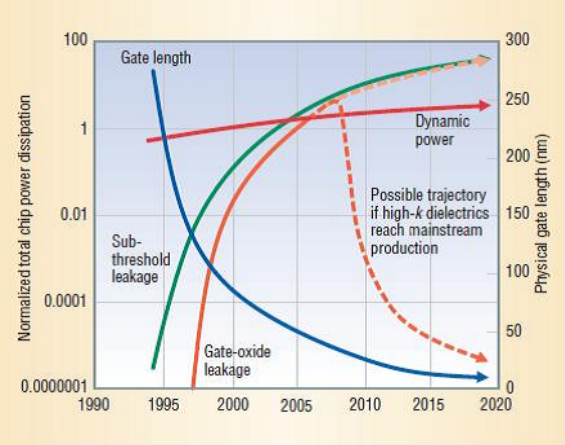

図 1.5. プロセス微細化に伴なうリーク電力増大の予測

1.3 本研究が取り組む問題

1.2 節で触れたように，プロセス微細化に伴なうリーク電力の増大が近年のマイクロプロセッサ設計において大きな問題となっている．図 1.5(文献 [10]) は，プロセス微細化に伴なって生じると予測されているリーク電力の急激な増大を表したグラフである．横軸が，年代，縦軸が正規化されたチップの消費電力およびプロセスのゲート長を表している．図ではリーク電力の一種であるサブスレッショルドリーク (図中の緑色の線) が，年代が進みプロセスが微細化していくにつれてダイナミック電力以上の大きさになることが予測されている．

こうした状況を受け，リーク電力増加の抑制はさらなるプロセス微細化に伴なう最も重要な技術的課題の一つとなっており，リーク電力削減を目指したトランジスタ製造技術，回路設計技術の技術開発が急務になっている．一方で，アーキテクチャレベルの設計においても，回路のリーク電力増大への対策が必要であると指摘されている (文献 [3])．

近年，リーク電力削減のためのアーキテクチャ技術としてパワーゲーティングと呼ばれる手法が注目されている．これは，処理のない回路への電源電圧供給を動的に遮断し回路をスリープさせることで，不要なリーク電力消費を抑える手法である．本研究ではこのパワーゲーティングをマイクロプロセッサの演算器に適用するさいに問題となるエネルギーオーバーヘッドに焦点を当て，より小さなエネルギーオーバーヘッドでより多くのリーク電力を削減するための演算器スリープ制御手法の構築を目指す．そのためにコンパイラによるコード解析を用いて演

算器に生じる処理の空き時間を正確に予測し，演算器のスリープを適切に制御する手法を提案する．

1.4　本論文の構成

本論文の構成は次の通りである．まず，本章 (1 章) においてマイクロプロセッサの消費電力の増大，その中でも特にリーク電力の増大が問題になっていることを述べ，本研究がマイクロプロセッサの演算器におけるリーク電力削減を目標とすることを述べた．

2 章では，本提案手法が想定するパワーゲーティング技術について述べ，本研究が取り組む課題をより詳しく述べる．また，演算器へのパワーゲーティング手法を実現するためのプロセッサアーキテクチャ，命令セット，コンパイラシステムについて述べる．

3 章では，本提案手法の中心部分であるコンパイラによるコード解析に基づくスリープ制御手法の詳細を述べる．また，コンパイラによる静的なスリープ制御手法とハードウェアによる動的なスリープ制御手法との併用によるハイブリッドな制御手法についても述べる．

4 章では，3 章で述べた提案手法によって達成できるリーク電力削減効果をシミュレーションによって評価する．また，本提案手法におけるコード解析の計算量についても評価を行う．

5 章では，4 章でのシミュレーションに加え，コード解析に基づく空き時間の予測精度についてより詳細な検討を行い，提案手法の改善案を検討する．一方で，細粒度パワーゲーティングによるリーク電力削減に適したコード生成についても考察する．

最後に 6 章で，本論文のまとめと今後の課題について述べる．

第 2 章

細粒度パワーゲーティング

2.1 パワーゲーティング技術

2.1.1 パワーゲーティングの先行研究

1 節で述べたように，本研究はマイクロプロセッサのリーク電力削減を目標としている．この目標を達成するために，本研究ではパワーゲーティングと呼ばれる手法を用いる．以下では，このパワーゲーティング手法について概説する．

パワーゲーティング手法は，動作の必要のない回路への電源供給を遮断する（スリープさせる）ことでプロセッサ全体の性能を低下させることなく，リーク電力を削減する手法である．図 2.1 は，パワーゲーティング回路の模式図である．パワーゲーティングにおいては，回路ブロックと電源線，もしくはグランド線の間にパワースイッチと呼ばれるトランジスタを挿入する．図では，回路ブロックとグランド線の間にパワースイッチを挿入している．スリープ信号 (sleep signal) を用いてこのパワースイッチを動的に ON/OFF することで，当該回路ブロックへの電源供給を制御することができる．回路ブロックへ電源電圧が供給されている状態を回路ブロックがウェイクアップモードにあると言う．一方，スリープ信号がアサートされ，パワースイッチにより回路ブロックのリークパスが遮断されている状態を回路ブロックがスリープモード (低リークモード) にあると言う．パワーゲーティングにおいては，スリープ信号の伝搬やスリープトランジスタの駆動，またスリープ期間中に図の VGND にたまった電荷の放電などのために時間的・エネルギー的なオーバーヘッドが生じることが知られている．

このパワーゲーティング技術をマイクロプロセッサに適用する際には

1. どの程度の大きさの回路ブロックに分割してパワーゲーティングするか？
2. どの程度の時間粒度でパワーゲーティングするか？

といった設計上の問題が生じる．

回路ブロックをより細かく，またスリープする時間間隔をより短くすれば，それだけ多くの空き時間を抽出することができると考えられる．一方でこのように細かい単位でスリープを行う場合には，性能やエネルギーへのオーバーヘッドが大きくなるとともに，パワーゲーティン

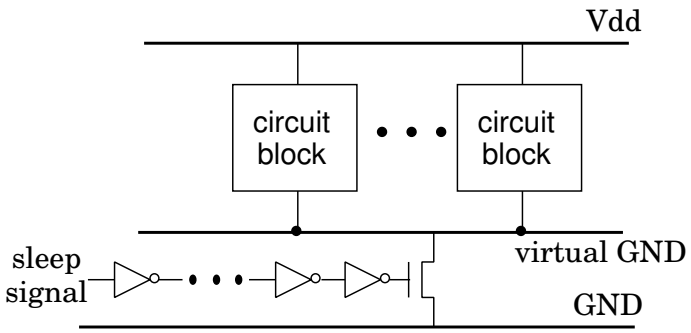

図 2.1. パワーゲーティング回路模式図

グを実現する回路が複雑なものとなってしまうといったデメリットがある．回路ブロックの細かさ，およびスリープ/ウェイクアップモード切り替えの時間間隔の細かさを，パワーゲーティングの粒度と呼ぶ．より回路を細かく，より短い時間間隔でモード切り替えを行う場合を細粒度なパワーゲーティングであるといい，その逆の場合を粗粒度なパワーゲーティングであるという．

空間的に最も粗い粒度のパワーゲーティングは，チップ全体を一つの単位としてスリープするパワーゲーティングであると考えられる．これは，チップ全体がアイドル状態である期間（アプリケーションを実行していない期間）に消費されているリーク電力を狙った電力削減手法である．チップ全体がアイドル状態であれば，チップへ電源電圧を供給する必要がないためこれを遮断することで不要なリーク電力を削減できる．チップ全体を単位としたパワーゲーティングは，バッテリー寿命の問題などが存在する組み込み系のプロセッサを中心にすでに実用化されている．しかし，この手法では削減できるリーク電力はチップ全体がアイドル状態の場合のリーク電力のみである．従って，アプリケーション実行時におけるマイクロプロセッサのリーク電力を削減することはできない．

実行時リーク電力が無視できない大きさになった近年では，アプリケーション実行時のリーク電力を減らす試みも行われている．マルチコアプロセッサにおいては，アイドル状態のコアへの電源電圧供給を遮断するという手法が実用化されている (文献 [14])．また，複数の CPU や専用 IP が 1 チップ上に搭載されている SoC において，多数のパワードメインを個別にスリープ制御できる仕組みも提案されている (文献 [8])．

これらの手法を用いればチップ全体がアイドルでない場合においても，使用していないコアや IP におけるリーク電力を削減することができる点で，従来のアイドル時を狙ったリーク電力削減手法とは異なっていると言える．

さらにこの考え方を推し進めて，一つのコアや IP 内部を細かいパワードメインに分割してパワーゲーティングを施す試みがなされている．マイクロプロセッサは，キャッシュやレジスタ，演算器などの回路要素から構成されているが，アプリケーション実行時といえども，必ずしもこれらの回路要素全てが処理を行っているわけではない．従って，パワーゲーティングをアプリケーション実行時のマイクロプロセッサ内部の回路要素に適用することで，より大きなリーク電力の削減が期待できる．一方で，より細粒度に回路をスリープさせた場合には性能

オーバーヘッドおよびエネルギーオーバーヘッドが大きなものとなる．そこで，スリープによるリーク電力削減を最大化する一方，性能およびエネルギーのオーバーヘッドを最小限に抑えるための手法が必要になる．

このようなアイディアに基づく，コア内部の細粒度パワーゲーティングの先行研究として，キャッシュにおける実行時パワーゲーティングの研究がある．キャッシュは，チップ面積の大きな割合を占めていることから，リーク電力削減の対象として重要である．先行研究 [9] では，キャッシュにおけるデータ利用パターンの特徴を利用したキャッシュのスリープ制御手法を提案している．キャッシュの次にパワーゲーティングのターゲットとして期待できるのが演算器である．演算器がトランジスタ当たりのリーク電力が大きなトランジスタで構成されていること，SIMD 型命令のサポートによる演算器面積の増加などの理由から，チップ全体のリーク電力に対する演算器におけるリーク電力の占める割合が大きくなるからである．演算器へのパワーゲーティング適用の先行研究としては文献 [6] がある．この先行研究では，シンプルな時間ベースのスリープ制御を用いて演算器にパワーゲーティングを適用している．

本研究は先行研究 [6] と同様，演算器にパワーゲーティングを適用する場合に必要となる演算器のスリープ制御手法に関する研究である．先行研究 [6] におけるシンプルな制御手法は，スリープさせたい演算器の使用頻度が低いアプリケーションにおいては十分なリーク電力削減を達成する．一方で，スリープさせたい演算器の使用頻度が高く演算器に生じる一つ一つの処理空き時間が数十〜数百サイクル程度の短いものとなる場合においては十分に演算器のリーク電力を削減できない．単純な時間ベースのスリープ制御は，演算器に生じる空き時間の長さを予測するメカニズムを持っておらず，適切なタイミングで演算器をスリープさせることができないためである．

そこで，本研究ではスリープさせたい演算器の使用頻度が高いアプリケーションを対象とし，演算器に生じる処理の空き時間の長さを正確に予測しながら演算器のスリープ制御を行うことができる手法の実現を目指す．具体的には，コンパイラによるコード解析に基づく演算器の空き時間予測を用いたスリープ制御手法を提案し，演算器のリーク電力を最大限に削減することを目指す．

2.1.2 エネルギーオーバーヘッド

ここでは，マイクロプロセッサの演算器にパワーゲーティングを適用する際に大きな問題となるパワーゲーティングに伴なうエネルギーオーバーヘッドについて詳しく説明する．

パワーゲーティング手法においては回路ブロックへの電源電圧を遮断するとき，および回路ブロックをスリープ状態から復帰させるときに，エネルギーオーバーヘッドが発生する．エネルギーオーバーヘッドを償却するためには，BET（*Break Even Time*：損益分岐時間）と呼ばれる時間より長くスリープしていなければならない．図 2.2 は，回路ブロックがスリープし再びウェイクアップするときの消費電力の遷移を表す．縦軸がエネルギーオーバーヘッドを考慮したリーク電力であり，横軸は時間である．図中では，斜線部２で表わされるスリープ中のエネルギー削減により，斜線部１で表わされるオーバーヘッドを償却している．このようにパ

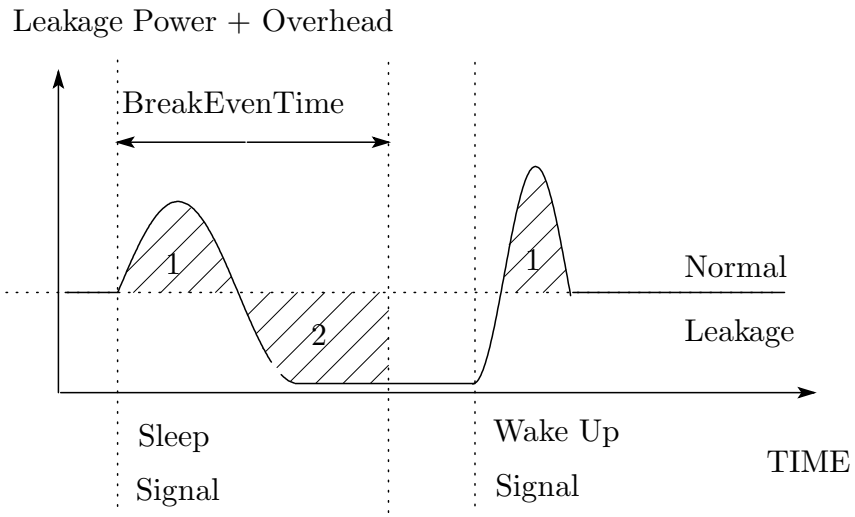

図 2.2. エネルギーオーバーヘッド

ワーゲーティングによってスリープモードに移行した際には BET 以上スリープしなければ逆に消費エネルギーが増大してしまう．この BET は半導体プロセスや対象回路の構成などの静的な要因，また温度（リーク電力が温度によって変化するため）のように動的な要因にも強く依存して変化することが知られている．文献 [17] では 90nm CMOS テクノロジにおいて各種演算ユニットの BET を，温度を 25–125 °C と変化させて測定したところ，最小で 2 サイクル，最大で 114 サイクルであったと報告されている．

特に短い時間間隔で頻繁に利用される演算器を実行時にスリープさせる場合には，スリープモードにとどまることのできる期間も短く，モード切り替えの回数も大きくなるため，エネルギーオーバーヘッドの影響が大きくなる．そこで，エネルギーオーバーヘッドおよび BET を考慮したスリープ制御が重要となる．すなわち，BET 以上長く処理がない場合にのみ演算器をスリープモードへ移行させ，それ以外のときにはスリープモードへ移行せず，次の処理をウェイクアップモードのまま待つべきである．そのためには，演算器に生じる空き時間の長さを予測しながらスリープを制御する手法が必要となる．

なお，スリープ/ウェイクアップモードの切り替え遅延に伴なう性能オーバーヘッドについては，文献 [17] においてその隠蔽技術が提案されている．これについては，付録 A の中で説明している．本研究ではこの性能オーバーヘッド隠蔽技術を用いることを仮定し，性能オーバーヘッドは生じないものとして，エネルギーオーバーヘッドに焦点を絞ってスリープ制御手法を考えていく．

2.2　パワーゲーティングの消費エネルギーモデル

マイクロプロセッサで消費されるリーク電力やリークエネルギーを実チップにおいて実測することは困難である．そこで提案手法の構築，および手法の評価時に用いるためのパワーゲーティングのアーキテクチャモデル，およびアプリケーション実行時に演算器で消費されるリー

cycle	instr	$ip^{mult}(t)$	$ac^{mult}(t)$	$md_1^{mult}(t)$	$md_2^{mult}(t)$
cycle 0	load	1	idl	wu	slp
cycle 1	mult	0	act	wu	wu
cycle 2	b	0	act	wu	wu
cycle 3	store	6	idl	wu	slp
cycle 4	load	5	idl	wu	slp
cycle 5	b	4	idl	wu	slp
cycle 6	b	3	idl	wu	slp
cycle 7	b	2	idl	wu	slp
cycle 8	add	1	idl	wu	slp
cycle 9	mult	0	act	wu	wu
⋮	⋮	⋮	⋮	⋮	⋮

図 2.3. アプリケーション実行中のモデル変数の具体例

クエネルギーのモデルを定義する．

2.2.1 演算器におけるパワーゲーティングのモデル

まず，パワースイッチ適用時の演算器におけるスリープ動作をモデル化する．そのために演算器の状態や消費電力などを表すいくつかのモデル変数を定義する．

アプリケーション実行時の演算器 K に対して，パワーゲーティングに関する以下のような変数，定数を定義する．ここで，t はアプリケーション開始時点からのサイクル数を表す (t は整数)．

- $ip^K(t)$：演算器の空き時間．時刻 t から次に演算器が使用されるまでの時間
- $ac^K(t)$：時刻 t における演算器のアクティビティ．act，idl の 2 つの値を取りうる．
- $md^K(t)$：演算器のモード．時刻 t における演算器のモードを示す．wu，slp の 2 つの値をとりうる．$ac^K(t) = act$ ならば $md^K(t) = wu$ でなければならない．
- l^K：演算器 K で消費される 1 サイクル当たりのリーク電力．
- e_{OH}^K：演算器 K が一回スリープすることによって消費されるエネルギーオーバーヘッド．(現実には定数ではないが，本稿では定数としている)
- $SP^K(t_1, t_2)$：演算器 K が，時刻 t_1 から t_2 の間にスリープした回数．

これらの変数の具体例を図 2.3 に示す．この例では，単一命令発行のインオーダプロセッサにおける乗算器を想定している．各列は，各サイクルにおける変数の値を表す．左から，時刻 t，命令実行パイプラインの EX ステージで実行されている命令，後続の空き時間 $ip^{mult}(t)$，乗算器のアクティビティ $ac^{mult}(t)$，アイドル時にも全くスリープをしない場合の乗算器のモード遷移 $md_1^{mult}(t)$，アイドル時には常にスリープをする場合のモード遷移 $md_2^{mult}(t)$ を表し

ている．実行命令列の b はデータ依存によるパイプラインバブルを表現している．ここでは，乗算命令の実行に 2 サイクル，キャッシュミス時の load 命令の実行に 3 サイクル，その他の命令は 1 サイクルで実行されるものとしている．演算器のモード $md^{mult}(t)$ の値が，$slp \rightarrow wu$ と遷移する cycle1 および cycle9 で乗算器はウェイクアップしている．一方，演算器のモード $md^{mult}(t)$ の値が $wu \rightarrow slp$ と変わるサイクルで演算器はスリープしていることになる．

ここでサイクル 0 からサイクル 9 までのスリープ回数 $SP^{mult}(0,\ 9)$ を考える．全くスリープを行わない場合のモード遷移 $md_1^{mult}(t)$ では，スリープ回数 $SP_1^{mult}(0,\ 9)$ は 0 である．一方，演算器に処理がなくなれば常にスリープする場合のモード遷移 $md_2^{mult}(t)$ では，cycle2 から cycle3 にかけてウェイクアップモードからスリープモードへの切り替えが生じており，スリープ回数 $SP_1^{mult}(0,\ 9)$ は 1 である．

スリープ回数 $SP^K(t_1,\ t_2)$ は，モード切り替えに伴なうエネルギーオーバーヘッドを評価するために用いられる．

2.2.2　リークエネルギーのモデル

本稿では，パワーゲーティングによるスリープ/ウェイクアップのモード切り替えに伴なうエネルギーオーバーヘッドを考慮に入れたスリープ制御手法を提案する．すなわち，パワーゲーティングに伴なうエネルギーオーバーヘッドと消費されたリークエネルギーの和を最小にするようなスリープ制御手法の構築を目指す．そこで，演算器 K において時刻 t_1 から時刻 t_2 にかけて生じたスリープに伴なうエネルギーオーバーヘッドと消費リークエネルギーの和 $Leak^K(t_1,\ t_2)$ を以下の数式でモデル化する．

$$Leak^K(t_1,\ t_2) := (\sum_{t=t_1}^{t_2} md^K(t) * l^K) + SP^K(t_1,\ t_2) * e_{OH} \tag{2.1}$$

この数式モデルを立てる上で用いた仮定は以下の 4 つである．

1. ウェイクアップモードにおける単位時間当たりのリーク電力が常に一定である．
2. スリープモードでのリーク電力は 0 である．
3. スリープ/ウェイクアップのモード遷移に要する時間は 0 である．
4. 1 回のスリープ/ウェイクアップに伴なうエネルギーオーバーヘッドは一定である．

右辺の第一項は，演算器がウェイクアップしている時間に消費されるリーク電力を表し，第二項はパワーゲーティングによるスリープ/ウェイクアップのモード遷移によって消費されたエネルギーのオーバーヘッドを表している．

なお本研究では，スリープに伴なうエネルギーオーバーヘッドと消費リークエネルギーの和を省略して消費リークエネルギーと呼ぶことにする．

図 2.3 の例で消費リークエネルギー $Leak^K$ を計算してみる．演算器が，全くスリープしない場合 $(md_1^{mult}(t))$，$Leak_1^{mult}(0,\ 9)$ は，

$$Leak_1^{mult}(0,\ 9) = 10l^{mult} \tag{2.2}$$

となり，一方常にスリープする場合 ($md_1^{mult}(t)$)，$Leak_2^{mult}(0,\ 9)$ は

$$Leak_2^{mult}(0,\ 9) = 3l^{mult} + e_{OH}^{mult} \tag{2.3}$$

となる．

ここで，スリープをする場合のリークエネルギー $Leak_1^{mult}$[式 (2.2)] とスリープをしない場合のリークエネルギー $Leak_2^{mult}$[(式 2.3)] のどちらが小さいかを考える．これは 1 サイクル当たりのリーク電力 l^{mult} と，一回のスリープ/ウェイクアップに伴って消費されるエネルギーオーバーヘッド e_{OH}^{mult} の比の値に依存する．すなわち，$\frac{e_{OH}^{mult}}{l^{mult}} > 7$ ならば，$Leak_1^{mult} < Leak_2^{mult}$ であり，スリープしない方がリーク電力量は小さくなる．一方，$\frac{e_{OH}^{mult}}{l^{mult}} < 7$ ならば $Leak_1^{mult} > Leak_2^{mult}$ であり，処理がなくなった場合にはスリープする方がリークエネルギーが小さい．

このように，一回のスリープ/ウェイクアップに伴って消費されるエネルギーオーバーヘッドを 1 サイクル当たりのリーク電力で割った値 $\frac{e_{OH}^{mult}}{l^{mult}}$ が，ある時刻 t に演算器に空き時間が生じた場合の，スリープするかしないかの決定において重要であることがわかる．この値は，2.1.2 節で説明した break even time （BET）に対応するモデル変数となっていると考えることができる．そこで，演算器 K の break even time BET を以下の式で定義する．

$$BET^K := \frac{e_{OH}^K}{l^K} \tag{2.4}$$

2.3 細粒度パワーゲーティングを実現するためのシステム

この節では，細粒度パワーゲーティングを実現するためのプロセッサアーキテクチャ，命令セット，およびコンパイラシステムについて説明する．このシステムは，2.1.2 節で触れた先行研究 [17] において報告されているマイクロプロセッサをもとにしたシステムである (付録 A 参照)．本研究では，ここで説明したシステムを想定して，スリープ制御手法を提案し，評価を行っている．

2.3.1 プロセッサアーキテクチャ，命令セット

細粒度パワーゲーティングを実現するためには，演算器を個別にスリープさせるための機構をマイクロプロセッサに実装する必要がある．このため通常のマイクロプロセッサに存在する機構に加えて，各演算器へスリープトランジスタを挿入し，また演算器のスリープ/ウェイクアップを制御するためのスリープコントローラを実装が必要である．図 2.4 に細粒度パワーゲーティングを実現するプロセッサアーキテクチャの例を示す．

パワーゲーティングの対象となる演算器としては様々なものを考えることができる．例えば，先行研究 [17] の実装ではコプロセッサ，整数乗算/除算器，シフタなどがパワーゲーティングの対象となっている．その他にも，浮動小数点演算ユニットや SIMD 演算ユニットなどがパワーゲーティングの対象とすることができる．4 章における評価では，一般的なマイクロ

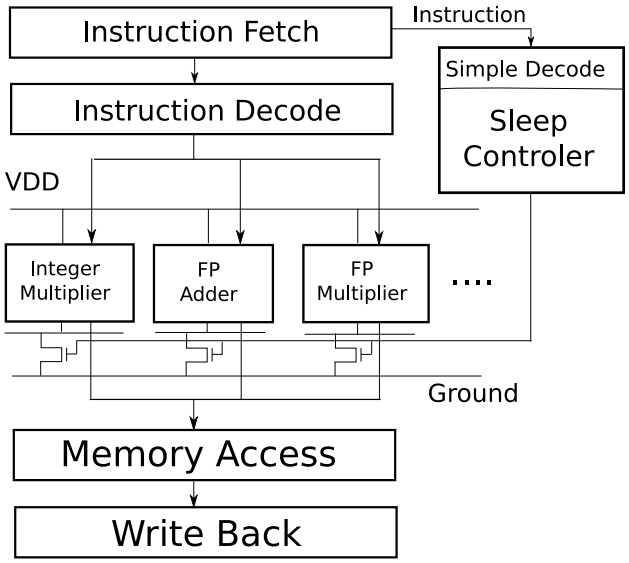

図 2.4. 対象アーキテクチャ

プロセッサに実装されており，またチップ上における実装面積が比較的大きくリーク電力も大きな整数乗算器，浮動小数点加算器，浮動小数点乗算器の３つの演算器をスリープ制御の対象としているが，本提案手法はその他の演算器にも適用することができる．

また，本研究においては単一命令・インオーダー発行のシンプルなプロセッサアーキテクチャを対象としてスリープ制御手法を考える．このようなシンプルなプロセッサは，性能要求は高くないが消費電力削減が特に重要な組み込み市場向けのマイクロプロセッサによく用いられること，および本提案手法で用いるコード解析に基づくスリープ制御手法と相性が良いことなどが理由である．なお，組み込み向けマイクロプロセッサの市場は，携帯電話やノートブックと言った製品への需要が高まっていることなどもあり，今後ますます重要になる分野でもある．

本研究では，コンパイラを用いた静的な演算器のスリープ制御を行う．そこで，コンパイラシステム内部で解析したスリープ制御情報をハードウェアに伝えるためのメカニズムが必要となる．これを実現する手法として，本研究ではスリープビットと呼ばれる１ビットのスリープ制御情報を各命令に埋め込む手法を用いる．このスリープビットは従来の命令セットに，スリープモードへの遷移を引き起こす１ビットを埋め込むことでソフトウェアによるスリープ制御を実現しようとするものである．スリープビットについては 2.3.2 節で詳しく説明する．

2.3.2　スリープビット

ここでは，本提案手法において命令セットに新しく定義するスリープビットについて説明する．このスリープビットは，先行研究 [17] において MIPS 命令セットに対して実装されたものと同様のものである．

スリープビットは，命令中に埋め込まれる１ビット分のスリープ制御情報である．処理して

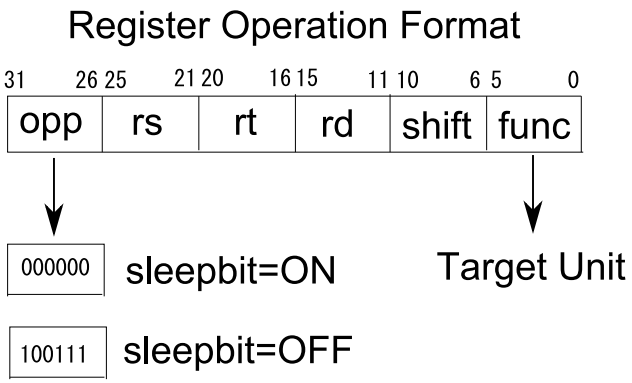

図 2.5. MIPS 命令セットにおけるスリープビットの実装

cycle	instr	$ac^{mult}(t)$	$md^{mult}(t)$
cycle 0	load	*idl*	*slp*
cycle 1	mult(slpbit=OFF)	*act*	*wu*
cycle 2	*b*	*act*	*wu*
cycle 3	store	*idl*	*wu*
cycle 4	load	*idl*	*wu*
cycle 5	*b*	*idl*	*wu*
cycle 6	*b*	*idl*	*wu*
cycle 7	*b*	*idl*	*wu*
cycle 8	add	*idl*	*wu*
cycle 9	mult(slpbit=ON)	*act*	*wu*
cycle 10	*b*	*act*	*wu*
cycle 11	store	*idl*	*slp*
cycle 12	load	*idl*	*slp*
⋮	⋮	⋮	⋮

図 2.6. スリープビットの動作例

いる命令に埋め込まれているスリープビットが ON の場合，当該演算器はその命令の実行終了と同時にスリープする．一方，処理している命令のスリープビットが OFF の場合には，当該演算器はスリープしないで次の命令を待つ．

このスリープビットは，命令空間の未定義領域を利用することで命令長を変更することなく実装することができる．先行研究 [17] では，実際に MIPS 命令セットにおいて命令長を変えることなくスリープビットを実現している．MIPS 命令セットにおいて最上位 6 ビットが「10011」となる命令は未定義であるが，これを利用してスリープビットの機能を実装している．図 2.5 に先行研究 [17] に基づいたスリープビットの実装を示す．

また，スリープビット手法においては演算器のスリープを制御するための特別な命令 (ス

リープ命令) を新しく定義する場合と異なり，実行可能ファイルサイズの増大やスリープ命令の実行に伴なう性能低下などを回避することができる．一方で，ハードウェアに伝えることができる情報が相対的に少ないこと，および命令空間の消費が大きいことなどがデメリットとして考えられる．本研究では，エネルギーオーバーヘッドの問題に焦点を絞るため性能低下を引き起こすことのないスリープビット手法を用いることを想定している．

スリープビットのメカニズムを実現するために必要なハードウェアの変更はデコーダのロジック部分と，スリープコントローラ部分の変更である．しかし，そのハードウェアコストは小さい．ハードウェアにテーブルやカウンタを追加することで演算器の空き時間を予測する手法に比べて，スリープ制御に必要な追加のハードウェアが少ないことがコンパイラによって静的なスリープ制御を行う手法のメリットの一つである．

次に 2.2 節で定義したモデルにおけるスリープビットの動作を示す．図 2.6 は，各サイクルごとの乗算器に関する変数変数の遷移の様子を示しており図 2.3 と同様の図である．ただし，cycle1 と cycle9 における乗算命令 mult にスリープビット (slpbit) が付加されているところが異なる．乗算器のスリープモードへの遷移は，実行した乗算命令に埋め込まれているスリープビットの値によって決まる．cycle1 の乗算命令のスリープビットは OFF であり，乗算器は cycle1 の乗算命令の実行後 (cycle3) スリープしない．一方，cycle9 における乗算命令 mult のスリープビットは ON であり，cycle9 の乗算命令実行直後 (cycle11) に乗算器がスリープする．このようにスリープビットが ON の場合にのみ，その命令の実行直後に当該演算器がスリープモードへ遷移する．

2.3.3　コンパイラシステム

本研究が提案する手法では，細粒度パワーゲーティングにおけるエネルギーオーバーヘッドを最小限に抑えながら演算器のスリープを制御するために，スリープビットを用いた静的な制御手法を用いる．このため，各演算命令のスリープビットが ON/OFF を決定するためのモジュールをコンパイラシステム内部に付け加える必要がある．ここでは，本研究で想定するコンパイラシステムについて概略を述べる．

本研究では，RISC 命令セット向けの一般的なコンパイラシステムを想定する．図 2.7 にコンパイラシステムの模式図を示す．高級言語で記述されたソースファイルは，まず字句解析・構文解析などを経て，中間表現に変換される．ここで，目的とする命令セットおよびプロセッサアーキテクチャに非依存のコード最適化が施される．次に，目的の命令セットおよびプロセッサアーキテクチャの情報を用いてオブジェクトファイルを生成する．このとき，命令スケジューリングやレジスタ割付などが行われ，様々なコード最適化が施される．最後に，リンカによってライブラリや別々にコンパイルされたモジュールを一つにまとめ，実行可能ファイルが完成する．

本研究における各命令へ埋め込むスリープビットの ON/OFF の決定はオブジェクトファイルのリンク時に行う．これは，オブジェクトファイルを解析対象とすることで目的の命令セット，プロセッサアーキテクチャの情報を利用できるからである．また，リンク時にコード解析

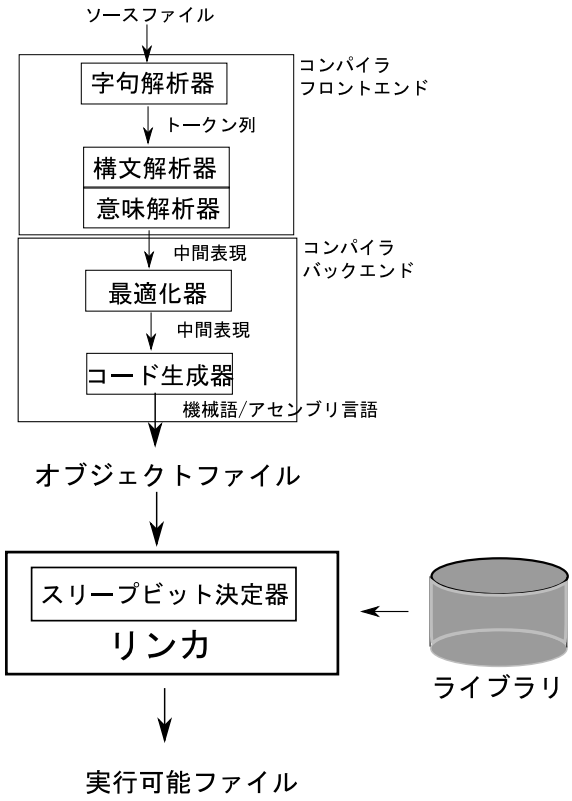

図 2.7. 対象とするコンパイラシステム

を行うことですべての関数モジュールの情報を利用したスリープビットの決定が可能だからである．図 2.7 では，リンカ内に描かれているスリープビット決定器がこの作業を行うモジュールである．

第 3 章

コード解析に基づくスリープ制御

本章では，スリープ/ウェイクアップのモード遷移に伴なうエネルギーオーバーヘッドを抑えつつ，細粒度パワーゲーティングを効率的に演算器に適用するためのスリープ制御手法を提案する．

はじめに 3.1 節において本提案手法と同様に，コンパイラによる静的なコード解析を用いた演算器のスリープ制御手法の先行研究を紹介する．次に，3.2 節において本研究で提案する空き時間の予測値を陽に用いたスリープ制御の枠組みを述べる．3.3 節では，空き時間を予測するためのコード解析アルゴリズムを提案する．このコード解析アルゴリズムは，コンパイラにおけるコード解析において従来から用いられているデータの流れの解析の枠組みを拡張し，分岐命令や関数を越えたグローバルなコード解析を行うものである．最後に 3.4 節で，提案するコード解析に基づくスリープ制御と相性の良い動的なスリープ制御手法としてキャッシュミス検知に基づくスリープ制御について説明する．キャッシュミス検知手法は，スリープビットによるスリープ制御では捉えることのできない動的な要因に起因する空き時間を捉えることが可能であり，提案するコード解析に基づくスリープ制御を効果的に補助することができる．

3.1 コード解析に基づくスリープ制御：先行研究

本提案手法と同じように，演算器の実行時リーク電力削減を目標として，演算器に対してコンパイラを用いた静的なスリープ制御を行おうとする手法がすでにいくつか提案されている．文献 [15] では，本提案手法と同様コンパイラによるソフトウェアレベルでのスリープ制御手法を提案しているが，議論の中心は性能的なオーバーヘッドについてである．文献 [16] ではループに着目し，プロファイリングを用いることによって長い空き時間の予測を行うスリープ手法を適用している．本研究は，プロファイリングを用いることを仮定していない点がこれとは異なっている．本提案手法は手法の適用対象となるプロセッサアーキテクチャとして，インオーダ実行のシンプルなプロセッサアーキテクチャを想定している．これとは異なるプロセッサアーキテクチャを扱った研究としてクラスタ化 VLIW アーキテクチャにおける演算器パワーゲーティングのためのコードスケジューリングの研究がある [12]．また，コンパイラによるリーク消費電力手法とハードウェアによる手法を定量的に比較している研究も存在する [4]．

本提案手法は，これらの先行研究では明確に定義されていない演算器の空き時間を明確に定義し，コード解析によってこの空き時間の値を陽に予測する手法である．また提案手法では関数を越えたコード解析を行っていること，および静的なスリープ制御を効果的に補助する動的なスリープ制御手法と併用することを想定していることが先行研究とは異なる．

3.2 提案手法の概要

3.2.1 空き時間予測の問題

以下では，2.2 節で定義した細粒度パワーゲーティングのモデルを用いて考える．ここでの議論は，各演算器ごとに独立に適用できるため変数の右肩にある演算器 K を省略する．なお，以下ではスリープさせようとしている演算器を目的演算器と呼ぶことにする．

2.2 節における例で説明したように，演算器におけるスリープの制御においては break even time を考慮することが必要である．エネルギーオーバーヘッドを抑えつつ，最大限の実行時リーク電力を削減するためには以下のような制御を行う必要がある．

あるサイクル t において目的演算器の処理がなくなったとき，

1. 空き時間 $ip(t)$ が BET 以上ある場合には，スリープする．
2. 空き時間 $ip(t)$ が BET 未満の場合には，スリープせず次の命令を待つ．

ここで問題になるのが時刻 t において演算器に処理の空きが生じたとき，その後に続く空き時間 $ip(t)$ が BET よりも長いか短いかをどのように知るかである．空き時間 $ip(t)$ は時刻 t には分からないことから，ある種の予測を行う必要がある．この予測を正確に行うことができれば，がパワーゲーティングによって大きなリーク電力を削減できる．

本提案手法ではより精度の良い空き時間予測に基づく効率的なスリープ制御を実現するため，空き時間 $ip(t)$ をコード解析によって陽に予測し，これを用いたスリープ制御手法を提案する．

3.2.2 空き時間予測の枠組み

ここでは，本提案手法におけるコード解析によって求まる予測空き時間と実行時空き時間 $ip(t)$ の間の関係を詳しく述べる．以下では，簡単のため目的演算器を使用する命令を目的命令と呼ぶことにする．

本提案手法におけるコード解析では，目的命令が実行されたあとに生じる平均空き時間を命令アドレスごとに予測し，これをもとに演算器のスリープを制御する．一般的に，あるアドレスの目的命令 a は複数回実行される可能性があり，各実行ごとの空き時間は必ずしも一定ではない．これは，後続の分岐命令の結果やパイプラインストールなどによって次に目的演算器を使用するまでの時間が動的に変化するからである．目的命令 a からフェッチされた命令の実行

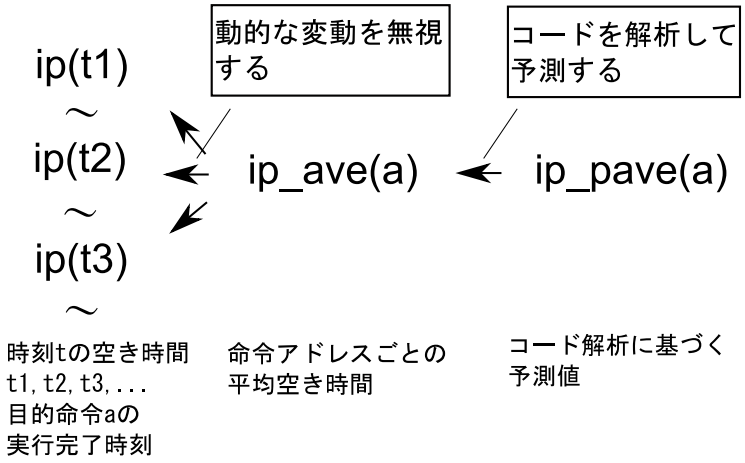

図 3.1. 静的な空き時間予測によって生じる 2 段階の誤差

完了直後の時刻の集合を $T(a)$ として，目的命令 a の平均空き時間を

$$ip_{ave}(a) \quad := \sum_{t \in T(a)} \frac{ip(t)}{|T(a)|} \tag{3.1}$$

と定義する．ここで $|T(a)|$ は，目的命令 a の実行が完了した時刻集合の要素数，すなわち目的命令 a が実行された回数を示す．

目的命令ごとに平均空き時間を予測しておけば，2.3.2 節で紹介したスリープビットの ON/OFF を決定する際に都合が良い．すなわち，コード解析によって求めた目的命令 a の予測平均空き時間 $ip_{pave}(a)$ が BET より長ければ目的命令 a に挿入するスリープビットを ON にし，そうでなければ OFF にすれば良い．

本提案手法の枠組みでは，まず目的命令 a が実行された直後の時刻 t における空き時間 $ip(t)$ を時刻 t に実行が完了した目的命令 a の平均空き時間 $ip_{ave}(a)$ で近似する．この目的命令 a に対する平均空き時間 $ip_{ave}(a)$ を，コード解析によって予測する．この予測値を $ip_{pave}(a)$ と表記することにする．

従って，ある時刻 t において実際に生じる空き時間と本提案手法で求まる予測値との間には図 3.1 のように 2 段階の誤差が生じることになる．

誤差 1
目的命令 a の平均空き時間 $ip_{ave}(a)$ をコード解析による目的命令 a の予測平均空き時間 $ip_{pave}(a)$ で近似

誤差 2
目的命令 a の実行が完了した時刻 t における空き時間 $ip(t)$ を目的命令 a の平均空き時間 $ip_{ave}(a)$ で近似

上記の 2 つの誤差が，本提案手法においてリーク電力削減効果に及ぼす影響については 5 章で検証している．

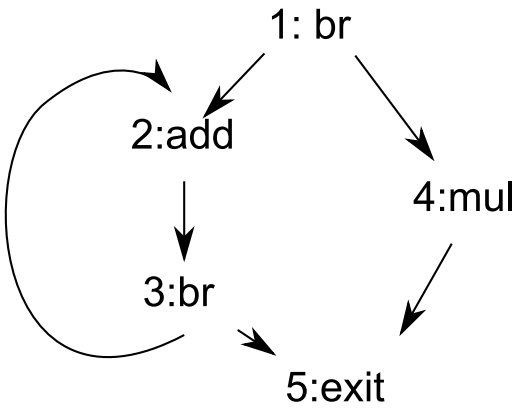

図 3.2. CFG の具体例

本節の最後に本提案手法における演算器のスリープ制御の流れを以下に示す.

1. 通常どおりアプリケーションをオブジェクトファイルにコンパイルする.
2. リンク時において，コード解析を行い各目的命令の予測平均空き時間を求める.
3. 各目的命令の予測平均空き時間をもとに, それぞれの命令のスリープビットを決定する.
4. 実行可能ファイルを生成する.
5. アプリケーション実行時に，スリープコントローラが実行可能ファイル中のスリープビットを判別し演算器のスリープを制御する.

3.3 平均空き時間を求めるためのコード解析

3.3.1 概要

本手法では，ある命令アドレスの命令が実行されたあとに生じる空き時間の平均値をコード解析によって求める．このような値を求めるために，コード最適化の分野で従来から用いられているデータの流れの解析の枠組みを用いている．データの流れの解析においては，コード内の各命令をノードとし，制御の流れをエッジで表現した制御フローグラフ（以下，*CFG*：*control flowgraph*）や関数間の呼び出し関係を表現した関数呼び出しグラフ (以下，*CG*：*call graph*) を用いて分岐命令や関数呼び出し命令を越えたコードの解析を行うことができる．これらについては，3.3.2 で詳しく定義する.

図 3.2 に CFG の例を示す．ノード 1:br，2:add，3:br，4:mul，5:exit が各命令を表し，各エッジはその間の制御の流れを表している．ここで，br を分岐命令，add を加算命令，mul を乗算命令，exit を関数からリターンする命令としている．例えば，乗算器における空き時間を予測するためにはコード中の乗算命令に着目し，3.3.3 節で説明するデータの流れの解析を行えば良い．本提案手法におけるデータの流れの解析では，目的命令とその目的命令より CFG 上で後方に存在する目的命令との間に挟まれる平均のノード数を求める．これをもとに，あるアドレスの目的命令が実行された後，どのくらいの長さの空き時間が生じるかを予測する．こ

の時に，関数呼び出しを越えて後方の命令を解析することで，スリープしたときにリーク電力削減効果の大きい長い空き時間を精度良く判別することができるように工夫している．

本提案手法では，単一命令発行のインオーダプロセッサを対象としており，各命令は 1 サイクルで終了すると仮定して演算器の空き時間を予測している．通常のプロセッサアーキテクチャでは，各命令が 1 サイクルで終了するという仮定は厳密には成立していないが，パイプラインが理想的に流れている場合などには実質的に 1 サイクルごとに 1 命令が発行されるため無理のある仮定ではない．なお，スーパースカラプロセッサなどより複雑なプロセッサアーキテクチャの場合においては IPC 予測などと組み合わせて空き時間を予測する必要があると考えられるが，それは本研究の範囲を越えているのでこれ以上は触れない．

3.3.2　データの流れ解析

ここでは，データの流れの解析について以下の節で必要な用語を定義する．なお，この定義は文献 [1] および文献 [21] を参考にしている．データの流れの解析は，理論的に良く研究されており実用的にも広く使用されているプログラムの解析手法である．

ベーシックブロックとは，命令の列でその間に分岐も合流もない命令のかたまりである．ベーシックブロックの中の命令列は，先頭から最後まで必ず一直線に実行される．

制御フローグラフ $G_f(V_f, E_f)$ は，プログラムの分岐や合流の様子を表現する有向グラフである．制御フローグラフは，関数ごとに定義される．通常，制御フローグラフにおけるノード v_f は，ベーシックブロックであるがここでは以降のデータの流れの解析の議論を行いやすくするため，ノード v_f を個々の命令とする．すなわち，グラフ中のノード v_f に対して，プログラム中の命令アドレスが１：１で対応することになる．ここでは，制御フローグラフ上のノードに対する命令アドレスを $a(v_f)$ と表記する．

また，エッジの定義もベーシックブロック間の分岐や合流だけではなく，アドレスの単純な増加によって生じるプログラムカウンタの遷移も含めた制御の流れに対して定義する．すなわち，

$$E_f = \{(v_f, v'_f) | a(v_f) \rightarrow a(v'_f) \text{ への制御の流れが存在する } \} \tag{3.2}$$

となる．

関数呼び出しグラフ $G_c(V_c, E_c)$ は，関数の呼び出し関係を有向グラフとして表現したものである．ノード v_c は，プログラム中の各関数である．従って，関数呼び出しグラフはプログラムに一つだけ定義される．エッジ e_c は，

$$E_c = \{e_c = (v_c, v'_c) | v_c \text{内で } v'_c \text{が呼び出される } \} \tag{3.3}$$

と定義される．

プログラム中におけるベーシックブロック，制御フローグラフ，関数呼び出しグラフの階層構造を図 3.3 に示す．

データの流れの解析においては，CFG 上の各ノード v_f の前後の位置にコードの性質を抽出するための変数群 $IN(v_f)$, $OUT(v_f)$ を定義する．次に，解析したいコードの性質に応じてそ

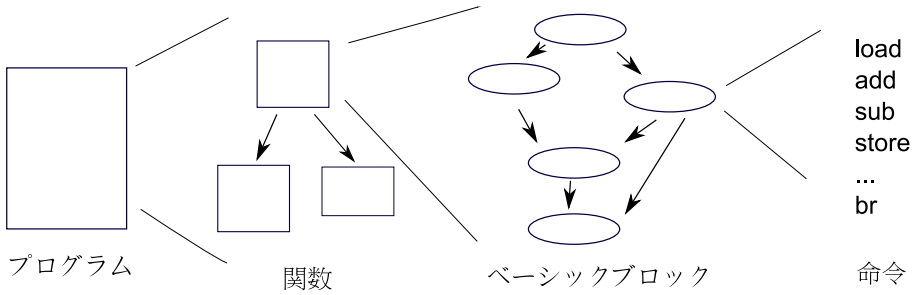

図 3.3. プログラムを表現するグラフ構造の階層

れらの変数群 $IN(v_f)$, $OUT(v_f)$ が満たすべき等式を CFG 上の各ノードおよび各エッジに対して定義していく．そして，適当に与えた初期値から，これらの等式群を満たす解を求める．データの流れの解析の一般論については，文献 [1] の第 7 章に詳しく書かれている．

通常のデータの流れ解析において CFG 上で定義される変数は，例えば「到達する定義」のように「命令の集合」であることが多い．一方，本手法では演算器の空き時間の予測値を表す「実数値」を定義して解析を行う．また，関数間解析を行う際，補助変数を定義する必要が生じるところも従来の解析とは異なる点である．これらについて，次節以降詳しく述べる．

3.3.3 関数呼び出しのない関数における平均空き時間予測アルゴリズム

まずはじめに，関数呼び出しがない関数内での解析について説明する．基本的な枠組みは，3.3.2 節で紹介したデータの流れの解析を用いている．

この解析においては制御の流れと逆の方向に向かってデータが流れる解析となる．また，各ノード（命令）には命令の種類ごとに使用する演算器 K が割り当てられているものとする．

以下では，次の 3 つのステップに分けて解析アルゴリズムを説明する．

1. CFG 上の各ノード v_f の前後に，実数値変数を定義する．
2. 各ノード v_f の前後の変数 $IN(v_f)$, $OUT(v_f)$ が満たすべき等式群を定義する．
3. 反復計算のための，各変数 $IN(v_f)$, $OUT(v_f)$ の初期値および関数出口での境界値を定義する．

最初に，以下のような実数値変数を CFG 内の各ノード s の前後に定義する．

- $IN_D[v_f]$：ノード v_f の直前の地点から次に目的の演算器を使用するノードまでのノード数の期待値.
- $OUT_D[v_f]$：ノード v_f の直後の地点から次に目的の演算器を使用するノードまでのノード数の期待値.
- $IN_P[v_f]$：ノード v_f の直前の地点から目的の演算器を使用する命令を実行せずに関数の出口までたどり着く確率.
- $OUT_P[v_f]$：ノード v_f の直後の地点から目的の演算器を使用する命令を実行せずに関数の出口までたどり着く確率.

ここで，$OUT_D[v_f]$ は 1 サイクルに必ず 1 つずつ命令が発行されるという仮定のもと，アドレス $a(v_f)$ の命令における平均空き時間 $ip_{ave}(a(v_f))$ の予測値としてなる．従って，$OUT_D[v_f]$ の値を用いてスリープビットの値を決定することができる．

次に，CFG 上の各ノードに定義されたこれらの変数が満たすべき流れの等式を与える．まず，等式中で使用する 2 つの定数 $T_D[v_f]$，$T_P[v_f]$ を CFG の各ノード v_f に対して定義する．これら 2 つの定数は当該ノードが使用する演算器が目的の演算器であるかどうかによって以下のような値を取る．

$$T_D[v_f] = \begin{cases} 0 & (\text{if } v_f \text{ uses the target unit}) \\ 1 & (\text{otherwise}) \end{cases} \tag{3.4}$$

$$T_P[v_f] = \begin{cases} 0 & (\text{if } v_f \text{ uses the target unit }) \\ 1 & (\text{otherwise}) \end{cases} \tag{3.5}$$

これらの変数を用いて流れの等式は,

$$\begin{aligned} IN_D[v_f] &= T_P[v_f]OUT_D[v_f] + T_D \\ IN_P[v_f] &= T_P[v_f]OUT_P[v_f] \end{aligned} \tag{3.6}$$

$$OUT_D[v_f] = \begin{cases} q[v_f] * IN_D[v_{f_{suc1}}] + (1 - q[v_f]) * IN_D[v_{f_{suc2}}] \\ (\text{if } v_f \text{ is a branch}) \\ IN_D[v_{f_{suc}}] \\ (\text{otherwise}) \end{cases} \tag{3.7}$$

$$OUT_P[v_f] = \begin{cases} q[v_f] * IN_P[v_{f_{suc1}}] + (1 - q[v_f]) * IN_P[v_{f_{suc2}}] \\ (\text{if } v_f \text{ is a branch}) \\ IN_P[v_{f_{suc}}] \\ (\text{otherwise}) \end{cases} \tag{3.8}$$

と与えられる．ここで，$v_{f_{suc}}$ は v_f が分岐命令でないときの v_f の後続命令を，$v_{f_{suc1}}$ および $v_{f_{suc2}}$ は v_f が分岐命令のときの 2 つの後続命令を表す．$q[v_f]$ は分岐命令を表すノード v_f ごとに与えることができるアルゴリズムのパラメータであり，分岐結果が $v_{f_{suc1}}$ の方を指し示す確率を表すこの $q[s]$ はループ構造を考慮に入れる，あるいはダイナミックなプロファイリングを取るなどして各分岐ごとに設定することができるパラメータである．なお，4 章における評価では，簡単に全ての分岐命令に対し $\frac{1}{2}$ という値を用いた．

最後に，上記流れの等式（3.6）–（3.8）を用いて反復計算を行うための初期値，および関数出口での境界値を与える．CFG 内の出口以外の各ノード v_f の初期値は,

$$\begin{aligned} &IN_D^0[v_f] = T_P[v_f] * T_D[v_f],\ IN_P^0[v_f] = T_P[v_f], \\ &OUT_D^0[v_f] = 0,\ OUT_P^0[v_f] = 0 \end{aligned} \tag{3.9}$$

Node	5:exit	4:mul	3:br	2:add	1:br
step 0	(0, 1, 0, 0)	(0, 0, 0, 0)	(1, 1, 0, 0)	(1, 1, 0, 0)	(1, 1, 0, 0)
step 1	(0, 1, 0, 0)	(0, 0, 0, 1)	$(\frac{3}{2}, 1, \frac{1}{2}, 1)$	$(\frac{5}{2}, 1, \frac{3}{2}, 1)$	$(\frac{9}{4}, \frac{1}{2}, \frac{5}{4}, \frac{1}{2})$
step 2	(0, 1, 0, 0)	(0, 0, 0, 1)	$(\frac{9}{4}, 1, \frac{5}{4}, 1)$	$(\frac{13}{4}, 1, \frac{9}{4}, 1)$	$(\frac{21}{8}, \frac{1}{2}, \frac{13}{8}, \frac{1}{2})$
⋮	⋮	⋮	⋮	⋮	⋮
ideal result	(0, 1, 0, 0)	(0, 0, 0, 1)	(3, 1, 2, 1)	(4, 1, 3, 1)	$(3, \frac{1}{2}, 2, \frac{1}{2})$

図 3.4. 図 3.2 におけるデータの流れの解析の具体例, 各マス内は (IN_D, IN_P, OUT_D, OUT_P)

とする．また，$v_{f_{exit}}$ を関数の出口を表すノードとして境界値を

$$IN_D^b[v_{f_{exit}}] = 0,\ IN_P^b[v_{f_{exit}}] = 1 \tag{3.10}$$

で与える．

反復計算の手順を，図 3.2 を用いて具体的に説明する．図の 1:br，3:br は分岐命令，4:mul のみが目的の演算ユニットを使用する命令であるとしている．また，5:exit は関数の出口を表す．まず，5:exit 以外の各ノードに対して流れの等式中で用いる定数 T_D，T_P を式（3.4）と式（3.5）を用いて計算すると，

$$\begin{aligned} T_D[1:br] = T_P[1:br] = 1 \\ T_D[2:add] = T_P[2:add] = 1 \\ T_D[3:br] = T_P[3:br] = 1 \\ T_D[4:mul] = T_P[4:mul] = 0 \end{aligned} \tag{3.11}$$

これらの定数値，式（3.9），（3.10）で与えられる初期値，境界値を設定し，初期値流れの等式（3.6）– （3.8）に従って反復計算を行っていく過程を図 3.4 に示す．図中の，各マス内はそれぞれのノードの各ステップにおける（IN_D，IN_P，OUT_D，OUT_P）を表す．図中の最下段には理論的に求められる反復計算の収束先の解を示している．この解析は，制御の流れとは逆方向に解析を行うため，ノード訪問の順序を制御フローグラフの出口から入り口へ向かって行うことで収束を加速することができる．このため，図では 4:mul→ 3:br→2:br... というように左から右へ変数の更新を行っている．ここで，ノード 5:exit は関数の出口であり，変数更新の必要はない．また，制御フローグラフ内のノードを 1 度ずつ訪問し変数を更新する作業を 1 step としている．

以下に本アルゴリズムの擬似コードを示す．

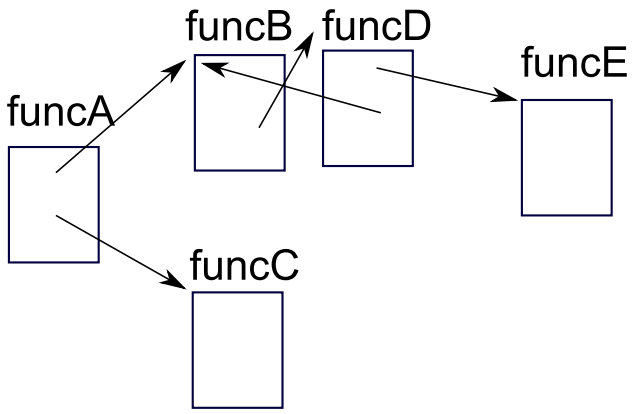

図 3.5. 関数呼び出しグラフ

Intra-Procedure Analysis

INPUT 関数の制御フローグラフ.
OUTPUT 関数内の各命令位置での予測平均空き時間の値.

/*式 (3.10) に従って境界値を設定*/
$IN_P[s_{exit}] \leftarrow 1$, $IN_D[s_{exit}] \leftarrow 0$
for each node $s \in$ CFG other than s_{exit} **do**
　/* 式 (3.9) に従って変数の初期値を設定 */
　Set initial values of $IN_P[s]$,　$IN_D[s]$, $OUT_P[s]$,　$OUT_D[s]$
end for
while changes to any OUT_P or OUT_D occur **do**
　for each node $s \in$ CFG other than s_{exit} **do**
　　/*式 (3.6), (3.7), (3.8) に従って変数の値を更新*/
　　update $IN_D[s]$, $IN_P[s]$, $OUT_D[s]$, $OUT_P[s]$
　end for
end while

厳密解を求めるためには無限回の反復計算が必要であるが，演算器の休止可能時間の予測値としては整数値が得られれば十分である．従って厳密な解を求める必要はないが，それでも値が収束するまでにはある程度の反復回数が必要になる．これについては 4 章で実際のアプリケーションにおける反復回数の評価を示している．

3.3.4　関数を跨いだ平均空き時間の予測アルゴリズム

3.3.3 節において，関数内部に関数呼び出し命令のない関数に対する解析について説明した．本節では，一般に関数内部に存在する関数呼び出しをどのように取り扱うかについて述べる．

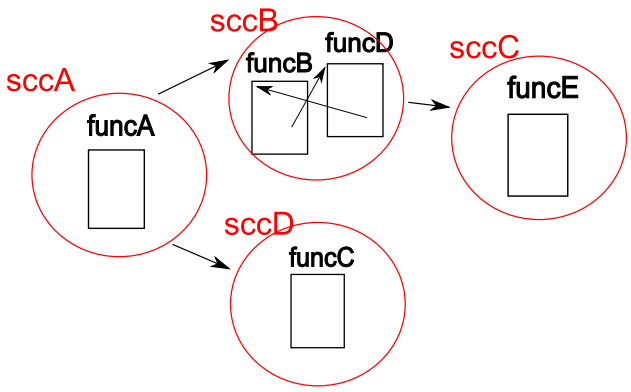

図 3.6. 強連結成分分解，各円内が強連結成分

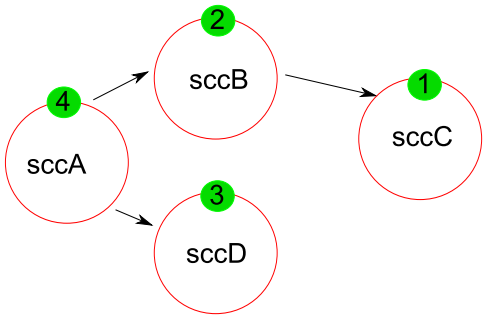

図 3.7. 深さ優先探索によるラベル付け

関数呼び出しの最も簡単な取り扱い方としては，呼び出す関数の先頭に目的命令が存在するという仮定をおき，関数呼び出し命令を目的命令を使用する命令として取り扱うことが考えられる．しかし，この方法は非常に保守的であり空き時間の予測値が実際のものに比べて大幅に短くなる．多くの関数呼び出しが存在する一般のプログラムにおいては，演算器の長い空き時間をコードから十分に抽出することができない．

そこで，関数をまたいだコード解析が必要となる．なお，説明のため以下では関数呼び出し命令として，*jal* なる疑似命令を用いる．

まず，関数間解析の全体像について述べる．関数間解析は大きく

1. 呼び出しグラフの強連結成分分解を行い，関数の解析順序を決定する．
2. グラフの各関数の伝達定数（後述）を求める．
3. プログラム全体の出口から情報を各関数へ伝える．

という 3 つのステップに分けられる．これらのステップは，文献 [1] で説明されているリージョンベースな解析（*region based analysis*）を参考にしている．

関数間解析では各関数を順番に解析していくが，各関数内の解析では自身を呼び出す関数および自身が呼び出す関数の解析情報を用いる．このため，上記のステップ 2，3 では関数を解析する順序が重要になる．そこで，ステップ 1 で呼び出しグラフに対して強連結成分分解および後行順のラベルづけを行い，関数の解析順序をあらかじめ決定しておく．

ステップ 1 ではまず，関数呼び出しグラフ内のループ構造を取り除くため強連結成分分解を行う．図 3.5 に呼び出しグラフの例を，その強連結成分分解を図 3.6 に示す．ここで，強連結成分をノードとする新しいグラフを CG' とする．次に，CG' の各ノード後行順のラベル付を行う．後行順のラベル付けは深さ優先探索によって行うことができる．図 3.7 に後行順にラベル付けされたグラフの例を示す．大きな円が関数呼び出しグラフの各強連結成分を，その内部の小さな円が各関数を表し，矢印が関数の呼び出し関係を表している．

各強連結成分内に関数が複数存在する場合にはこれらの解析順序に任意性が存在する．様々な解析順序が考えられるが，本研究における実装では強連結成分のサイズと同じ回数だけ，強連結成分内に存在する関数の解析を順番に繰り返すことで強連結成分内部の関数間での情報を交換するようにしている．

次に，ステップ 2 においてどのように関数間で情報をやりとりするかについて述べる．

まず，3.3.3 節で各命令に対して定義した伝達定数 T_D，T_P を今度は各関数に対して定義する．これらの値を関数の伝達定数と呼ぶことにする．これらは，

$$T_D[prc] := IN_D[v_{f_{ent}}],\ T_P[prc] := IN_P[v_{f_{ent}}] \tag{3.12}$$

と定義する．ただし，prc は関数を表し，$v_{f_{ent}}$ は関数 prc の入り口における命令を表す．これら関数の伝達定数を用い，3.3.3 節で定義した各命令の T_D，T_P の定義を次のように置き換える．

$$T_D[v_f] = \begin{cases} 0 & (\text{if } v_f \text{ uses the target unit}) \\ T_D[prc_{called}] & (\text{if } v_f \text{ is “jal”}) \\ 1 & (\text{otherwise}) \end{cases} \tag{3.13}$$

$$T_P[v_f] = \begin{cases} 0 & (\text{if } v_f \text{ uses the target unit}) \\ T_P[prc_{called}] & (\text{if } v_f \text{ is “jal”}) \\ 1 & (\text{otherwise}) \end{cases} \tag{3.14}$$

prc_{called} は関数呼び出し命令 jal が呼び出す関数を表す．このように流れの等式中の定数を定義することで，3.3.3 節の解析アルゴリズム自体を変更することなく，関数呼び出し命令を扱うことが可能になる．本アルゴリズムでは関数を呼び出してからその地点にリターンするまでの一連の処理を非常に複雑な操作を行う一つの命令と考えていることになる．

ステップ 2 では，上記で定義した関数の伝達定数を計算する．ここでは，ステップ 1 でラベル付けした順序に従って各強連結成分内の関数を解析する．すなわち，関数呼び出しグラフの根から葉に向かって各関数の伝達定数 T_D, T_P が決定していくことになる．

このときの関数内部の解析においては，前章で示した式 (3.4)，(3.5) の代わりにに式 (3.13) と式（3.14）を用いて前節と同様の初期値（3.9）および境界値（3.10）のもと，同様の反復計算を行い解を求める．自身の計算結果から，その関数の伝達定数 T_D，T_P が得られるが，この値はこの関数を呼び出す別の関数内部の解析で用いられる．

ステップ 3 ではステップ 2 で求めた関数の伝達定数 T_D, T_P を用いて，プログラム全体の出口から逆方向にデータを伝搬させ関数内の解析を行っていく．このため，ステップ 1 におけるラベル付けの逆順に各強連結成分内の関数を解析する．すなわち，今度は関数呼び出しグラフの根から葉に向かって解析が行われる．これは，プログラム全体の出口が，関数呼び出しグラフの根に当たる関数に存在するからである．

ステップ 3 において，プログラムの出口から制御の流れと逆方向にプログラムの入り口まで解析情報を伝えていくことで，プログラムの全ての位置における演算器の使用間隔を得る．ここでの各関数内の解析方法は，CFG の出口における境界値を除いてステップ 2 と同様である．CFG 出口における境界値は，自身を呼び出関数における解析結果を用いて決定する．すなわち，式（3.10）の代わりに関数出口ノード $v_{f_{exit}}$ における変数 $IN_D[v_{f_{exit}}]$ の値を，自身を呼び出す命令を表すノード $v_{f'_{call}}$(通常，他の関数内に存在) における変数 $OUT_D[v_{f'_{call}}]$ の値を用いて決定する．自身を呼び出す命令は，複数の関数の中に複数存在し得るがそれらの平均値をとるなどの方法によって関数出口における境界値を得ることができる．

以上のように，式（3.12)，(3.13)，(3.14）と前章で示した流れの等式（3.6）–（3.8）を用いて，関数間解析に基づく各命令アドレスからの演算器空き時間の予測値を求めることができる．以下に，関数間解析アルゴリズムの擬似コードを示す．

Inter-Procedure Analysis

INPUT 呼び出しグラフ CG と 関数呼び出しグラフ内の各関数の制御フローグラフ.
OUTPUT 呼び出しグラフ中の各関数内の各命令位置における予測平均空き時間の値.

/*関数呼び出しグラフを強連結成分分解し，解析順序を決定する*/
Find strongly-connected componets in CG.
Reduce the graph by treating a strongly-connected componet as one node.
Denote the reduced graph as CG'.
Perform depth-first search on CG', giving nodes post-order labels.

/*呼び出しグラフの葉から根の方向に向かって，各関数の T_D,T_P を計算していく*/
for $P \in CG'$ according to the post-order **do**
  for $i = 0$ to the number of procedures in P **do**
    for each procedure $prc \in P$ **do**
      /*呼び出す関数の T_D,T_P を用いる*/
      do intra-procedure analysis for prc
    end for
  end for
end for

/*呼び出しグラフの根から葉の方向に向かって，各関数内の各命令の IN, OUT を決定していく**/
for $P \in CG'$ according to the depth-first order(Reverse of the post-order). **do**
  for $i = 0$ to the number of procedures in P **do**
    for each procedure $prc \in P$ **do**
      /*呼び出す関数の T_D,T_P，および自身を呼び出す命令の OUT_D を用いる*/
      do intra-procedure analysis for prc
    end for
  end for
end for

3.4　動的なスリープ制御とのハイブリッド手法

3.2 節で述べたように提案手法におけるコード解析では，コードに起因する演算器の空き時間を正確に予測することが可能である一方，ハードウェアの命令実行方法に起因する動的な演

算器の空き時間を予測することが原理的に困難である．また同一アドレスの命令が実行された後に生じる空き時間は常に一定ということはなく，動的な要因によって変動し得る．このような動的な要因による演算器の空き時間への影響をコンパイラによって対処することは難しい．そこで空き時間の動的な変動の影響もスリープ制御に反映するため，静的に決定する演算器の空き時間をコンパイラで予測することに加えて，キャッシュミス検知によるスリープ制御を併用するハイブリッド手法を提案する．

近年のプロセッサにおいては，キャッシュミスはプロセッサをストールさせる最も大きな要因である．キャッシュミスが頻発するアプリケーションにおいては，コンパイラによる演算器の空き時間予測の精度が低下する恐れがある．これは，キャッシュミスの発生をコンパイラによって知ることが難しいためである．そこで，本研究ではキャッシュミスをトリガーとして演算器をスリープさせるという動的なスリープ制御手法とコード解析に基づく静的なスリープ制御手法を併用するハイブリッド手法を提案する．キャッシュミスはその構成やキャッシュのレベルにもよるが，通常長いミスレイテンシを伴ないプロセッサストールを発生させる．自身の BET がこのキャッシュミスレイテンシよりも短い演算器はキャッシュミスの発生と同時にスリープモードへ移行することで，安全にリーク電力を削減することができる．

キャッシュミスをトリガーとしたこの手法はハードウェアを利用した手法であるが，キャッシュミスを検知するための追加ハードウェアはごくわずかである．従って，キャッシュミス検知によるスリープ制御はスリープ制御のための追加ハードウェアが少ないというコンパイラによる静的なスリープ制御の利点を損なわない．キャッシュミス検知によるスリープ制御手法は，すでに文献 [17] において実際の LSI として実装されている．

第 4 章

評価実験

ここでは，3 章で提案したコンパイラによる空き時間予測に基づくスリープ制御手法について，その効果をプロセッサシミュレータを用いた評価によって示す．はじめに，評価した値，評価におけるぱパラメータ，比較したスリープ制御手法などについて述べたあと，提案手法を適用した場合の演算器のリークエネルギーのグラフを示す．また，本提案手法で用いているコード解析アルゴリズムの計算量に関する評価も行う．なお，ここで評価した値は電力ではなくある一定の時間の間に消費されたリークエネルギーであるが，この値をシミュレーションした総時間で割れば平均消費電力となる．

4.1 評価環境

評価には，*SimpleScalar Tool Set*[2] 内のサイクルレベルシミュレータである sim-outorder を使用した．本稿ではインオーダープロセッサを想定しているため，オプションとして *-inorder* を設定し，また，キャッシュをインオーダープロセッサに多く使用されるブロッキングキャッシュに変更して評価を行っている．

詳細な設定を，表 4.1 に示す．リークエネルギーを評価した演算ユニットは整数乗算器（以下 IntMul），浮動小数点 Alu（以下 FPAlu），浮動小数点乗算器（以下 FPMul）の 3 つである．

ベンチマークプログラムとして，SPEC CPU2000[5] の浮動小数点演算ベンチマークから 9 つのベンチマークプログラムを使用した．SPEC CPU2000 の浮動小数点ベンチマークの中のアプリケーションは，上記 3 種類の演算器を特に頻繁に使用し，リークエネルギーを削減するためには精度の良い空き時間予測が必要となる．

使用した命令セットは MIPS 命令セットをベースにした SimpleScalar シミュレータ用の命令セットである PISA 命令セットを用いている．*SimpleScalar Tool Set* 内の gcc を用いてコンパイルし最適化オプションは-O2 とした．また，データセットには ref インプットセットを使用した．

評価にはプログラムの実行開始から数えて 10 億命令から 38 億サイクル分の実行のシミュレーション結果を用いている．

また，*SimpleScalar Tool Set* にはリークエネルギーを評価する機能がないため，2.2 節で

表 4.1. 評価に用いたパラメータ

ISA	PISA
Issue	in-order
Fetch & decode & issue & commit width	1
Branch prediction	Combined predictor (bimodal & 2-lev., 4K-entry)
Functional units	Int ALU: 1 Int Multiplier/Divider: 1 FP ALU: 1 FP Multiplier/Divider: 1
L1 I-cache	32KB, 32B line, 2way 1 cycle latency
L1 D-cache	32KB, 32B line, 2way 1 cycle latency
L2 unified cache	1MB, 128B line, 8way 6 cycle latency
Memory latency	100 cycle

定義したリークエネルギーのモデル式を用いてアプリケーション実行中に消費されたリークエネルギーを評価している．具体的には，シミュレータによって演算器 K がウェイクアップモードであった総サイクル数 $\sum_t wa^K(t)$ および，スリープした回数 SP^K を取得し，これを式 2.1 に代入して消費されたリークエネルギーを求めている．なお，評価結果は全て相対的な値を用いているため，1 サイクル当たりのリークエネルギー l^K および 1 回のスリープ/ウェイクアップのモード切り替えに伴うエネルギーオーバーヘッド e_{OH} の値を指定する代わりに break even time $BET^K = \frac{e^K_{OH}}{l^K}$ を指定している．ここでは，この BET^K を，それぞれ 20 および 80 サイクルとした 2 つの値についての評価結果を示す．これらの値は文献 [17] の中で評価されている break even time の値を参考にしている．

提案手法のリークエネルギー削減効果を比較検証するために，4 つの異なるスリープ制御手法を各演算器に適用した場合の消費リークエネルギーを評価している．評価を行ったスリープ制御手法は次の通りである．

- compinter：3 章で提案したコード解析手法に基づくスリープ制御.
- compintra：関数間解析を行わずに，関数呼び出し命令を保守的に扱ってコード解析を行った場合のスリープ制御.
- L2：L2 キャッシュミス発生時にスリープするスリープ制御．本シミュレーションにおいては L2 キャッシュは最下位であり，そのミスレイテンシは常に BET 以上である.

- hybrid：3 章で提案したコード解析手法に基づくスリープ制御と L2 キャッシュミス検知によるスリープ制御を同時に適用するスリープ制御.

コード解析に基づくスリープ制御手法に用いる空き時間予測のためのコード解析アルゴリズムは Java によって実装している．各分岐命令における分岐確率は，簡単に全て $\frac{1}{2}$ としている．また，関数ポインタなどによる関数の間接参照についてはポインタ解析を行っておらず，関数の間接参照については関数間解析を行っていない．コンパイル時に各演算器の BET が既知であると仮定している。

4.2　リークエネルギー削減効果

4.2.1　スリープ制御手法ごとの結果

図 4.1〜4.6 は，$BET = 20$ および 80 とした場合の，FP 加算器，FP 乗算器，整数乗算器で消費されるリークエネルギーのシミュレーション結果を表している．縦軸のリークエネルギーは，全くスリープしない場合のリークエネルギーを 1 として正規化して表示している．各スリープ制御手法の結果は，演算器，BET，アプリケーションという 3 つのパラメータの組に対して与えられている．以下では，これらのパラメータの組を簡単のために，演算器-BET-アプリケーション，と表記することにする．例えば，図 4.1 中の wupwise ベンチマークに対する各スリープ制御手法の結果を参照する場合には，FPAlu-20-wupwise，と表記する．また，() の中に，「,」で区切られた複数のパラメータが表記されている場合には，あり得る全ての組み合わせを参照することにする．例えば，(FPAlu，FPMul)-(20，80)-wupwise，と表記した場合には FPAlu における BET=20 の wupwise の結果，FPMul における BET=20 の wupwise の結果，FPAlu における BET=80 の wupwise の結果，FPMul における BET=80 の wupwise の結果，という 4 つのパラメータの組の結果をあわせて参照している．また，表 4.2 に各アプリケーションにおいて，総実行サイクルに対するメモリアクセスによる待ちサイクル数および各演算器が処理を行っているサイクル数の割合を示す.

まず，関数間解析を行うことによるリークエネルギー削減効果の向上を見る．関数間解析を行うコード解析によるスリープ制御（compinter）は，関数間解析を行わないコード解析によるスリープ制御（compintra）に比べて，FPMul-80-mgrid における場合を除いて，全ての場合においてより大きなエネルギー削減効果を達成していることが分かる．また，FPMul-20-wupwise の場合のように関数をまたぐことによるリークエネルギー削減効果の向上が非常に大きい場合も存在している．一方関数間解析を行わない場合，IntMul-(20，80)-equake の場合のように目的演算器がほとんど使用されていないアプリケーションに対しても，スリープする機会を逃してしまうことがあることが分かる．このようなことから，演算器のスリープ制御のための空き時間予測においては，関数間解析が重要であることが分かる.

次に，関数間解析を行うコード解析に基づくスリープ制御，および L2 キャッシュミス検知によるスリープ制御を単独で適用した場合の結果を比較する．全体的な傾向としては，コード解析に基づくスリープ制御の方が L2 キャッシュミス検知によるスリープ制御よりも大きなエネ

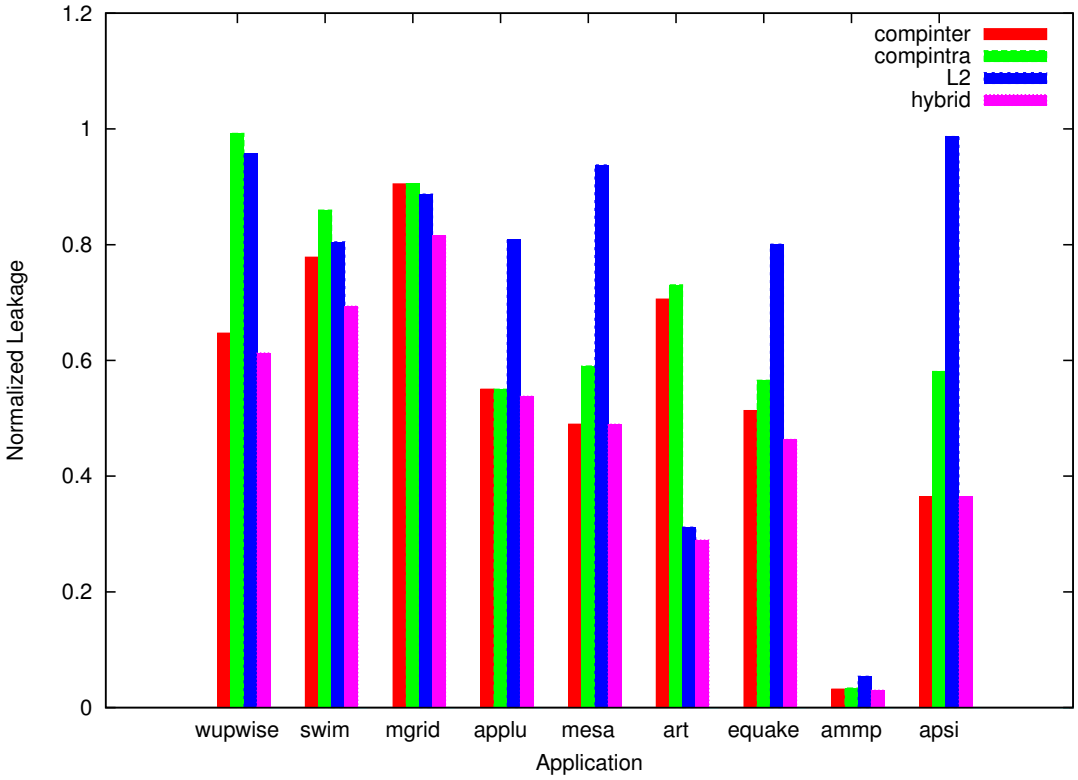

図 4.1. 提案手法を適用した場合の正規化された消費リークエネルギー，FP 加算器，BET=20

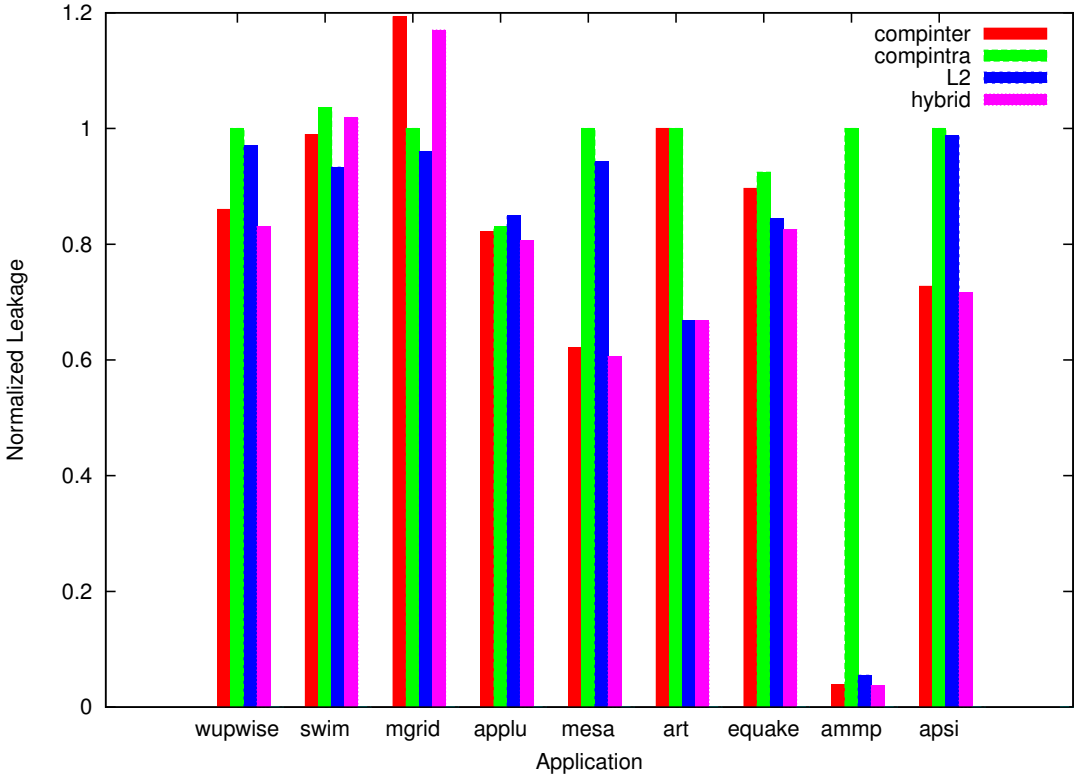

図 4.2. 提案手法を適用した場合の正規化された消費リークエネルギー，FP 加算器，BET=80

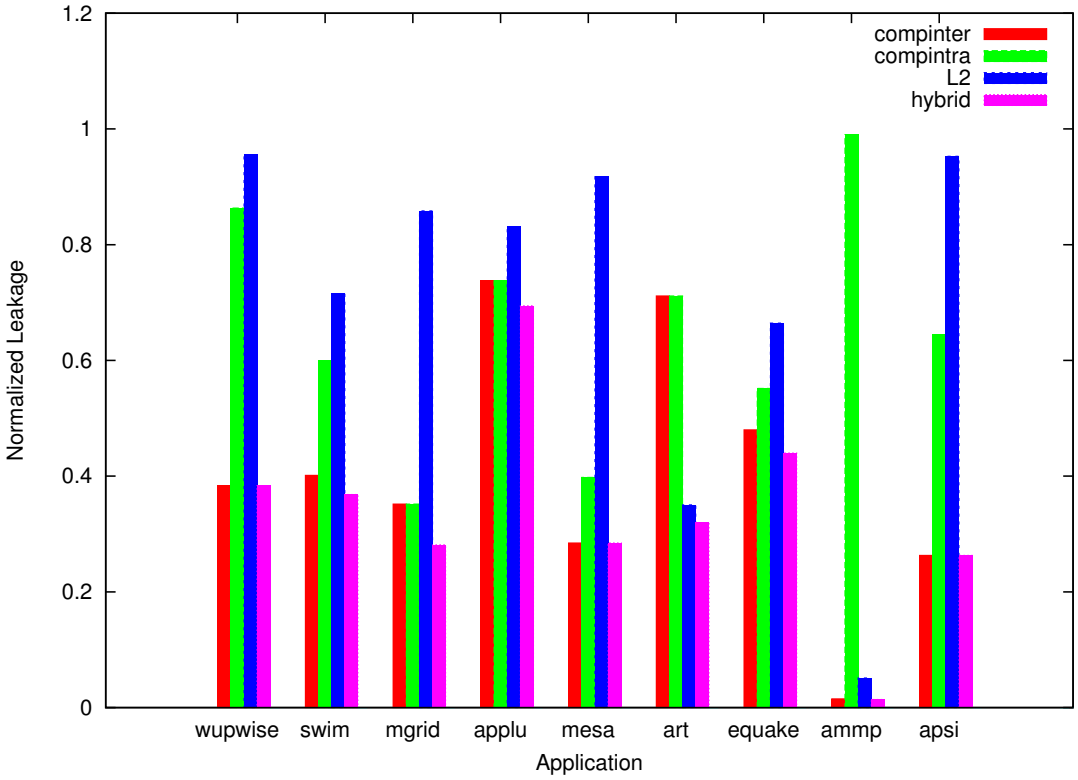

図 4.3. 提案手法を適用した場合の正規化された消費リークエネルギー，FP 乗算器，BET=20

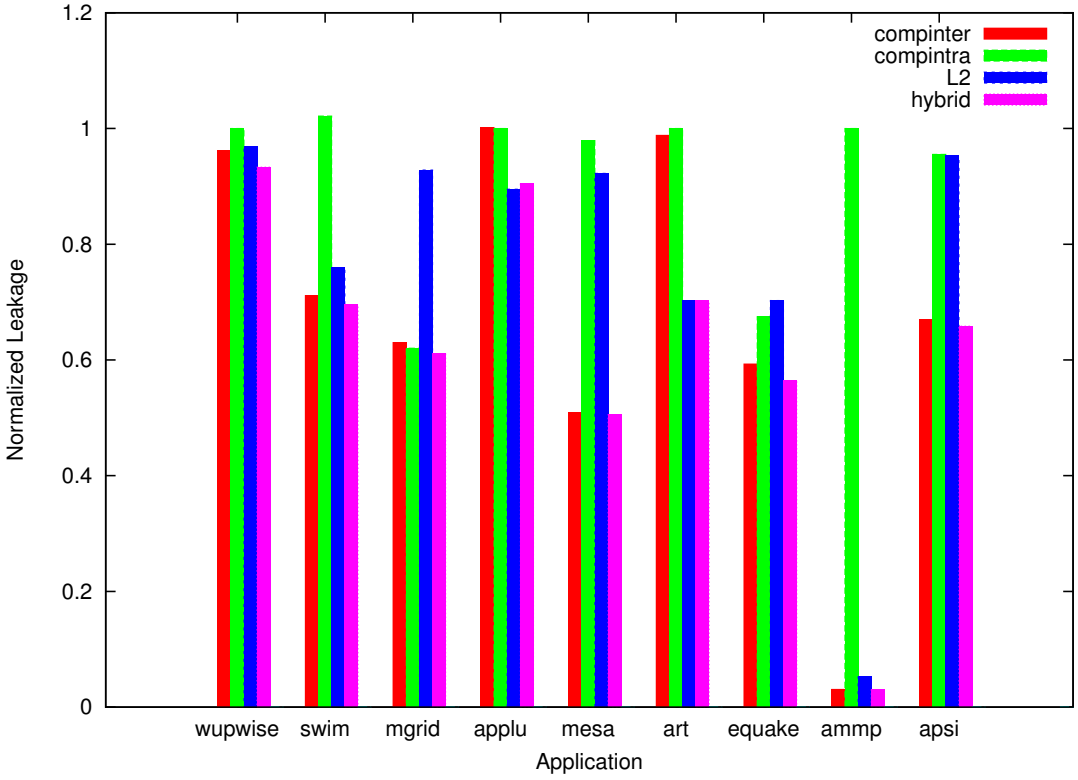

図 4.4. 提案手法を適用した場合の正規化された消費リークエネルギー，FP 乗算器，BET=80

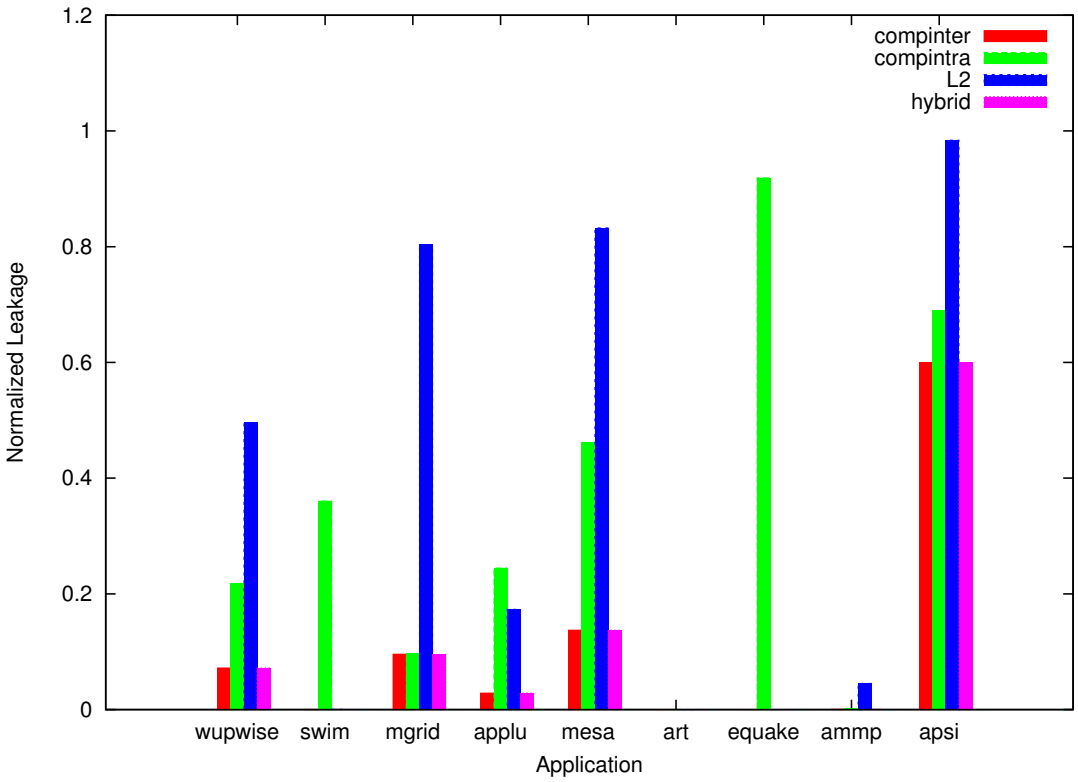

図 4.5. 提案手法を適用した場合の正規化された消費リークエネルギー，整数乗算器，BET=20

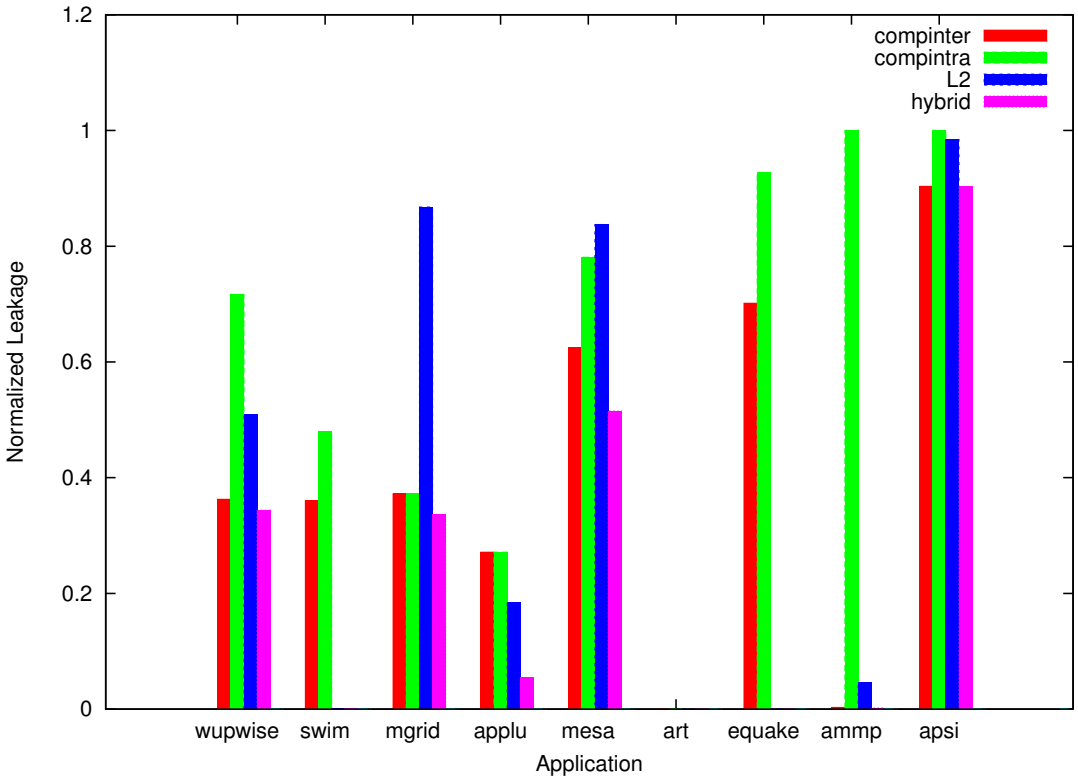

図 4.6. 提案手法を適用した場合の正規化された消費リークエネルギー，整数乗算器，BET=80

ルギー削減を達成している．一方で, FPAlu-80-(swim, mgird) および (FPAlu, FPMul)-(20, 80)-art の結果では，L2 キャッシュミスによる制御の方がエネルギー削減効果が大きくなっている．FPAlu-80-(swim，mgrid) の場合には，コード解析に基づく空き時間予測がうまくいっておらず，BET 未満の短い空き時間しか生じないタイミングでのスリープが頻発し，エネルギーオーバーヘッドが大きくなってしまっているのがその原因である．そのため，コード解析に基づくスリープ制御を適用した場合の正規化されたリークエネルギーが 1 を越えており，全くスリープをしない場合より消費エネルギーが増大してしまっている．一方，(FPAlu, FPMul) - (20，80) - art の場合には実行しているアプリケーションが L2 キャッシュミスを頻発することが原因である．表 4.2 を見ると art の実行時間のほとんどはメモリアクセス待ちの状態である．静的なコード解析手法では，メモリアクセス待ちに起因するストール時間を考慮することができないため，キャッシュミスが頻発するアプリケーションでは有効にリークエネルギー削減を達成することができない場合があることが分かる．一方，アプリケーション ammp もキャッシュミスが頻発するアプリケーションであるが関数間解析を行うコード解析に基づくスリープ制御では，単独でも十分なリークエネルギー削減を達成している．アプリケーション ammp の場合には，命令中に含まれる目的命令の数が少なくそれら命令の間の距離がコードの中でも大きいと考えられる．そのためキャッシュミスの影響を考慮していない場合でも，予測空き時間が BET 以上あると予測されスリープビットを ON にすることができたと考えられる．

最後に，コード解析 (関数間) に基づくスリープ制御手法と L2 キャッシュミス検知によるスリープ制御を組み合わせたハイブリッド制御（hybrid）のリークエネルギー削減効果を見る．ハイブリッド制御 (hybrid) は，FPAlu - 80 - (swim，mgrid) の場合を除く，全ての場合において最も大きなリークエネルギー削減を達成している．前述のように，コード解析に基づくスリープ制御および L2 キャッシュミス検知によるスリープ制御を単独に適用した場合には，それぞれ効果の大きなアプリケーションが異なっていた．これらを組み合わせることでほぼ全ての場合において大きなリークエネルギー削減を達成している．

4.2.2　時間経過に伴なうリークエネルギーの推移

ここでは，いくつかの (演算器-BET-アプリケーション) の組に対して，アプリケーションの実行経過に伴なう消費リークエネルギーの推移を示す．図 4.7〜4.10 は，横軸に時間，縦軸にスリープしない場合に対して正規化された消費リークエネルギーを示している．横軸の単位は，１億クロックサイクルである．それぞれの折れ線が，各戦略が当該の区間において達成したリークエネルギー削減率を示している．図 4.7〜4.10 は，評価を行った演算器-BET-アプリケーションの組の中でも，各スリープ制御手法の特徴をよく表していると考えられるものを選んでいる．

図 4.7(FPMul-80-mgrid) のグラフにおいて，まず特徴的のは 5 億サイクルから 7 億サイクルおよび 25 億サイクルから 27 億サイクルにかけての部分である．ここでは，全ての戦略のリークエネルギーが急激に下がっている．図 4.7(FPMul-80-mgrid) において，さらに特徴的

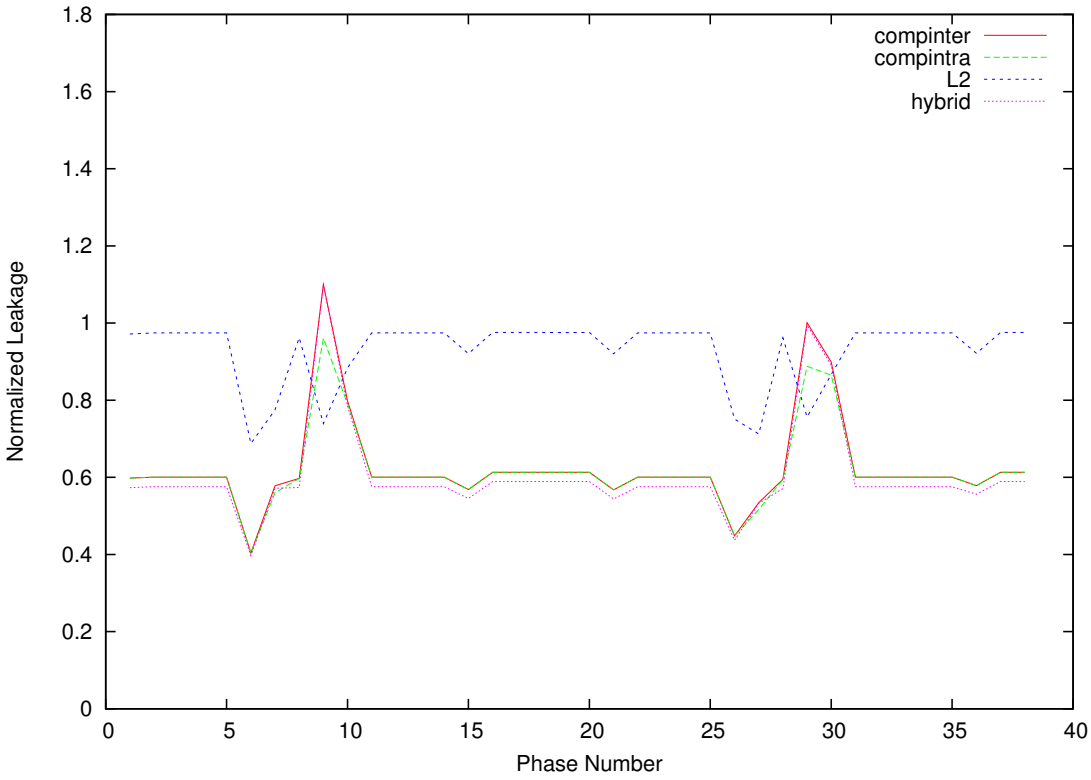

図 4.7. 時間経過とリークエネルギー削減率，mgrid，FPMul，BET=80

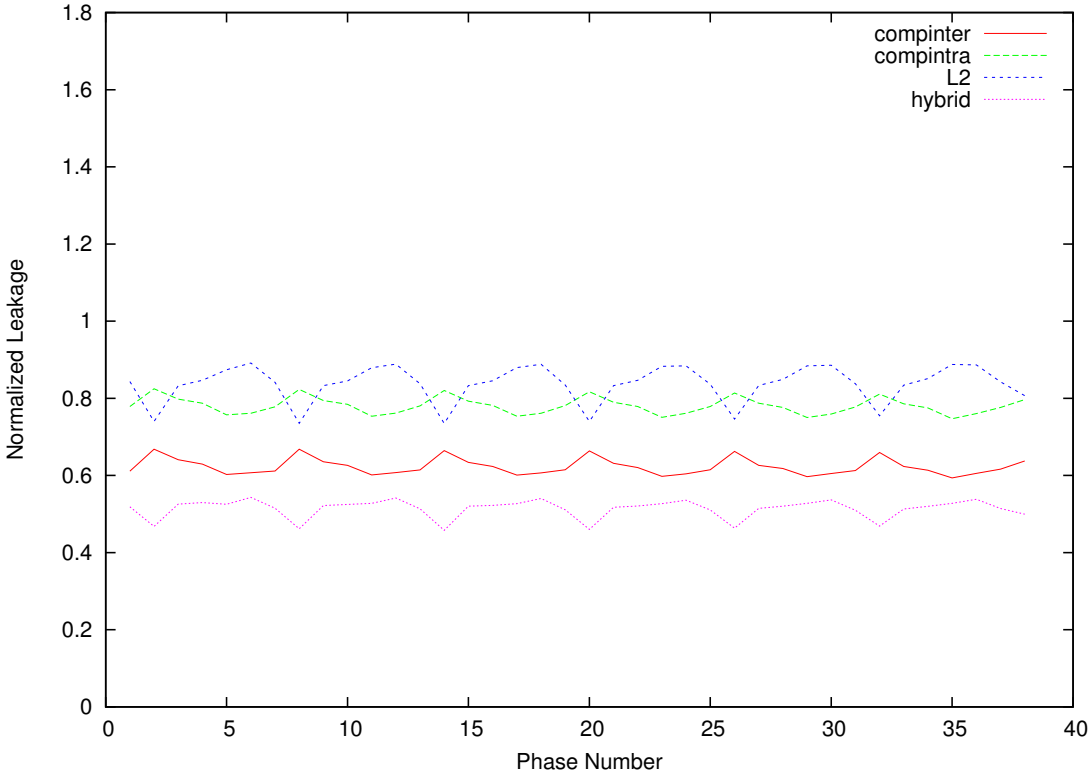

図 4.8. 時間経過とリークエネルギー削減率，mesa，IntMul，BET=80

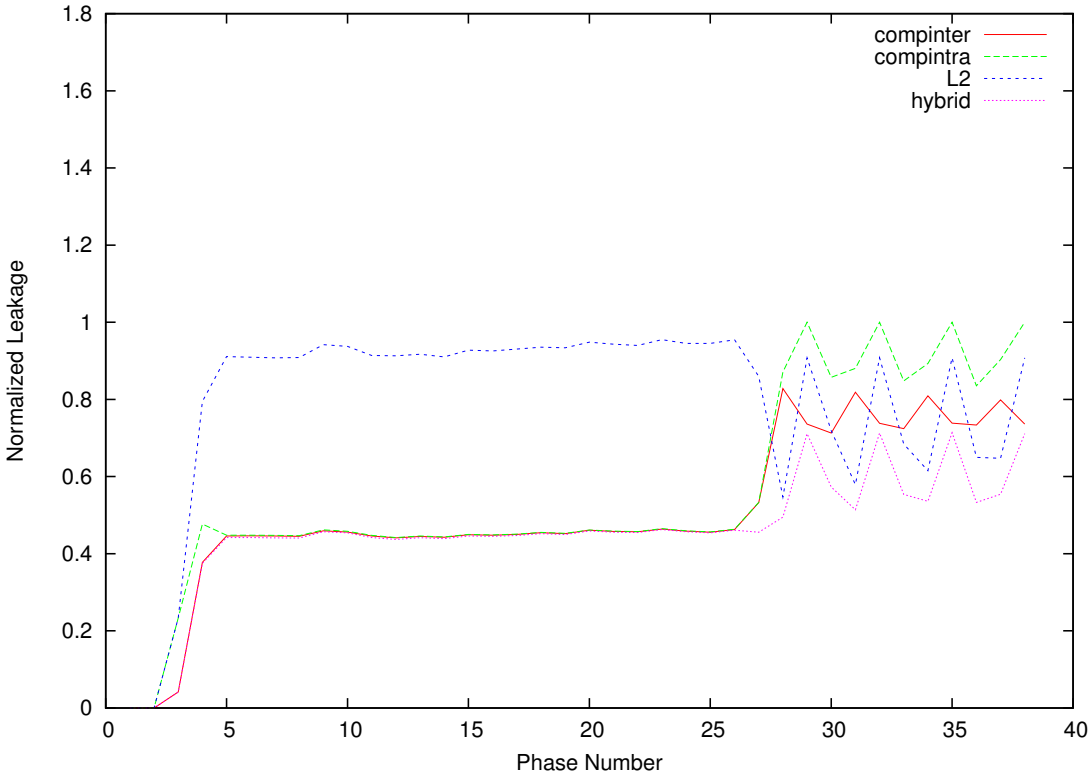

図 4.9. 時間経過とリークエネルギー削減率，equake，FPAlu，BET=20

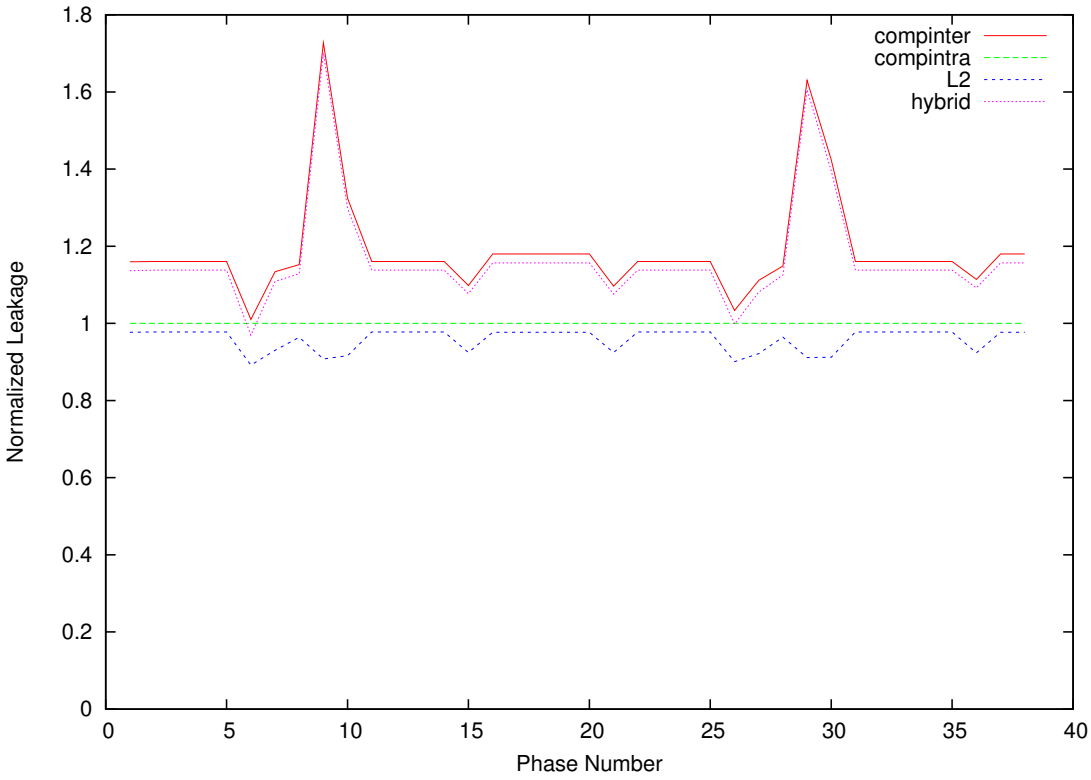

図 4.10. 時間経過とリークエネルギー削減率，mgrid，FPAlu，BET=80

	メモリアクセス	FPAlu 使用	FPMul 使用	IntMul 使用
wupwise	0.028	0.074	0.129	0.015
swim	0.266	0.102	0.121	0.000
mgrid	0.129	0.216	0.066	0.014
applu	0.130	0.066	0.263	0.009
mesa	0.009	0.101	0.103	0.041
art	0.755	0.025	0.050	0
equake	0.133	0.067	0.141	0.000
ammp	0.802	0.005	0.004	0.000
apsi	0.005	0.045	0.057	0.114

表 4.2. 各処理の総実行サイクル数に占める割合

なのは 8 億サイクルから 10 億サイクルおよび 28 億サイクルから 30 億サイクルにかけての部分である．ここでは，L2 キャッシュミス戦略のリークエネルギーが急激に下がると同時に，その他の戦略のリークエネルギーが急激に上がっている．

同様の現象が，図 4.8(IntMul-80-mesa)，および図 4.9(FPAlu-20-equake) の 27 億サイクルから 40 億サイクルの部分で生じている．すなわち，これらのフェーズにおいてはコード解析に基づくスリープ制御とキャッシュミス検知によるスリープ制御の効果の間に負の相関が見られる．すなわち，キャッシュミス検知によるスリープ制御を適用した場合のリークエネルギーが上がれば，コード解析に基づくスリープ制御を適用した場合のリークエネルギーが下がり，逆にキャッシュミス検知によるスリープ制御を適用した場合のリークエネルギーが下がれば，コード解析に基づくスリープ制御適用した場合のリークエネルギーが上がるという対応関係が現れている．また，2 つの制御手法を同時に適用したハイブリッド手法におけるリークエネルギー削減幅が 2 つの戦略によるリークエネルギー削減を足しあわせた形になっており互いの制御手法が悪影響を及ぼすことなく独立にリークエネルギーを削減していることが分かる．

これらのグラフから，コード解析に基づくスリープ制御とキャッシュミス検知を利用したスリープ制御では，効果の大きな局面が異なっていることが分かる．コード解析に基づくスリープ制御は，特に目的演算器が頻繁に利用される場合に有効性が高い．一方，キャッシュミス検知によるスリープ制御特ではキャッシュミスが頻発する場合において有効性が高い．ハイブリッド手法においては，この 2 つのスリープ制御手法が互いに悪影響を与えることなく独立に働くことで，より多くのアプリケーションおよび様々なアプリケーションのフェーズにおいて有効なスリープ制御を実現している．

一方で，図 4.10(FPAlu-80-mgrid) では，9 億サイクル付近および 29 億サイクル付近で，関数間解析を行うコード解析に基づくスリープ制御およびハイブリッド手法のリークエネルギーが急激に大きくなっており，80% 近くも消費電力が増大してしまっている．このフェーズにおいては，実際には短い空き時間しか生じない場合であるにも関わらず，コンパイラが長い空

き時間を予測してしまったことで，不適切なタイミングでのスリープが頻繁に繰り返され大きなエネルギーオーバーヘッドが生じている。このようなケースを防ぐためには，コード解析による空き時間予測の精度を向上させる必要がある．提案手法によるコード解析の空き時間予測が実行時の空き時間をどの程度予測できているかについては，5 章でより詳細に検討する．

4.3　コード解析の計算量

ここでは，空き時間予測のためのコード解析手法自身の実行速度について評価する．実際の計算時間は，プラットフォームおよび実装によるため，ここでは，データの流れの解析で行う反復計算の収束に要する反復回数，すなわち 3.3.3 節の Intra-Procedure Analysis 擬似コード中の while 文が停止するまでに要するループの反復回数を評価する．この反復計算は，各関数の制御フローグラフ上で実行されるものであり，その反復回数がデータの流れ解析の計算量および実行速度を決定するものである．反復計算の回数は，データの流れの解析で解析する変数の性質や制御フローグラフ中に含まれるベーシックブロックの個数やトポロジーによって変化する．ここでは，提案する空き時間予測アルゴリズムにおいてデータの流れの解析が収束するまでに要する反復計算回数を様々なアプリケーションで評価した．

図 4.11〜4.16 に，アプリケーション applu，mesa，apsi における制御フローグラフ中に含まれるベーシックブロックの個数と空き時間予測を得るために要した反復回数の関係を示す．縦軸が反復回数，横軸が制御フローグラフ中に含まれるベーシックブロックの個数である．

図から反復回数はベーシックブロックの個数にはあまり依存せず，ほとんどの場合に３０回以下の反復計算で値が収束していることが分かる．しかし，通常のデータの流れの解析における反復回数が１０回未満であること (文献 [1]) を考慮すると本提案手法は数倍の計算量を必要とすることが分かる．これは通常のデータの流れ解析で扱われる変数が集合であるのに対し，本提案手法では変数に実数を用いているため，情報の伝播に加えて実数値の収束を待つ必要があるからだと考えられる．

本提案手法の，データの流れ解析を高速化するための方針はいくつか考えられる．まず，データの流れの解析の収束速度は，各回の反復計算において制御フローグラフを訪問する順序によって変化することが知られている．

すなわち，3.3.3 節の Intra-Procedure Analysis 擬似コード中の、while 文内の for 文における各ノードのの訪問順序を工夫することで反復計算の回数を減らすことができると考えられる．現在の実装では単純な訪問順序を用いており，文献 [1] で紹介されている Depth-First-Ordering を用いた訪問順序の工夫などが考えられる．また，本提案手法を線形代数の言葉で表現すると一般的な連立一次方程式を解いていることに相当する．そこで，線形代数の分野において研究されている連立一次方程式救解の高速化手法が適用できると考えられる．

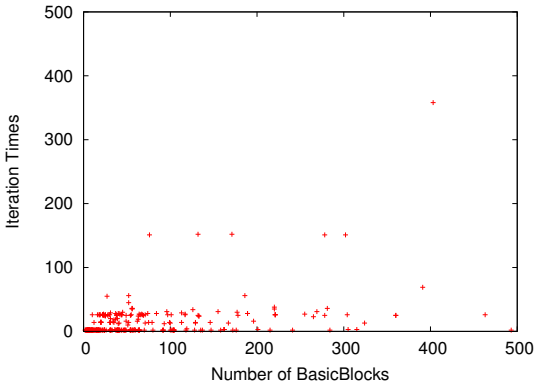

図 4.11. 制御フローグラフサイズと反復回数 (applu, FPAlu)

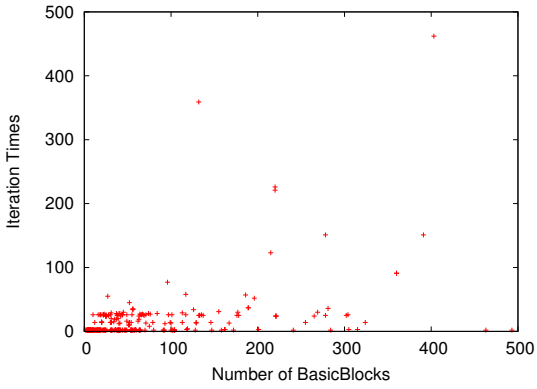

図 4.12. 制御フローグラフサイズと反復回数 (applu, IntMul)

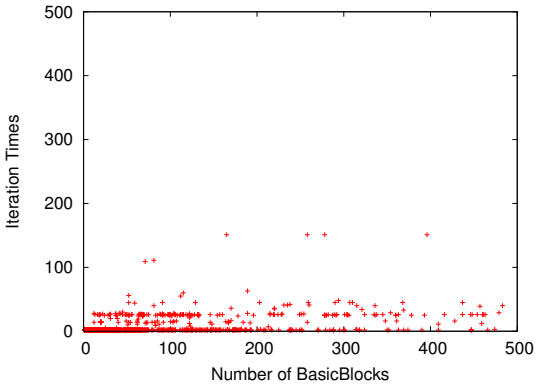

図 4.13. 制御フローグラフサイズと反復回数 (mesa, FPAlu)

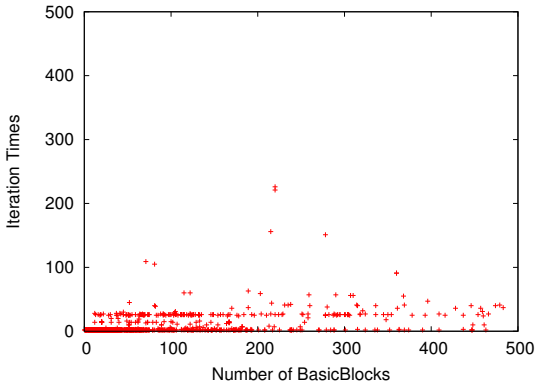

図 4.14. 制御フローグラフサイズと反復回数 (mesa, IntMul)

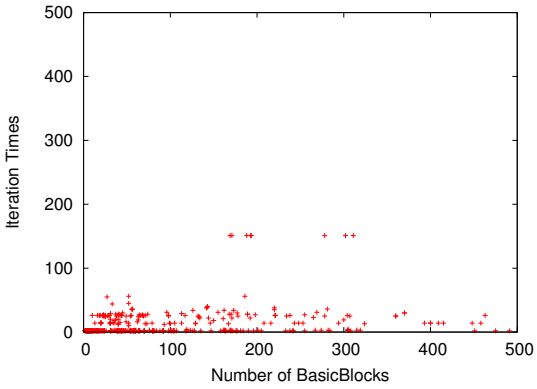

図 4.15. 制御フローグラフサイズと反復回数 (apsi, FPAlu)

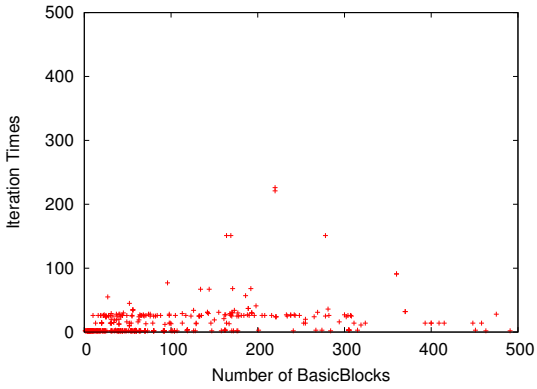

図 4.16. 制御フローグラフサイズと反復回数 (apsi, IntMul)

第 5 章

考察

この章では，5.1 節で，コード解析によるスリープ制御によって達成できたリークエネルギー削減と理想的なスリープ制御が実現できた場合のリークエネルギー削減との比較を通じて，提案手法によるスリープ制御をさらに効果的なものにするための考察を行う．また，5.2 節では，細粒度パワーゲーティングによるスリープのためのコード最適化の可能性について考察を行い，スリープ制御だけにとどまらない演算器のリーク電力削減におけるコンパイラ手法の可能性を検討する．

5.1 達成可能なリークエネルギー削減の上限との比較

5.1.1 評価方法

ここでは，スリープ制御によって達成可能なリークエネルギー削減の上限と提案手法によって達成できたリークエネルギー削減を比較する．リークエネルギー削減の上限は 4 章で行ったシミュレーションで得た演算器使用履歴から評価することが可能である．この比較評価により，提案手法が演算器のリークエネルギー削減という目的をどの程度達成しているのか，さらにリークエネルギーを削減する余地は残っているのか，を知ることができると考えられる．

スリープ制御によって達成可能なリークエネルギー削減の上限としては，２種類のものが考えられる．

１つ目は，スリープビットを用いたスリープ制御を行う場合に達成可能なリークエネルギー削減の上限である．これは，コード解析において (動的な要因によって生じる空き時間も含めて) 目的命令の平均空き時間をを完全に知ることができた場合の制御に相当する．以下では，これを理想的なコンパイラ制御 (bestcomp) と呼ぶことにする．

２つ目は，スリープビットを用いたスリープ制御に限らずあらゆるスリープ制御を考えた場合に達成可能なリークエネルギー削減の上限である．これは，演算器に空き時間が生じる瞬間にその時刻 t における空き時間を完全に知ることができる場合のスリープ制御である．すなわち，空き時間が BET 以上続くものならばスリープし，BET 未満ならスリープしない，という制御を理想的に行う場合に消費されるリークエネルギーである．以下では，これを理想的なス

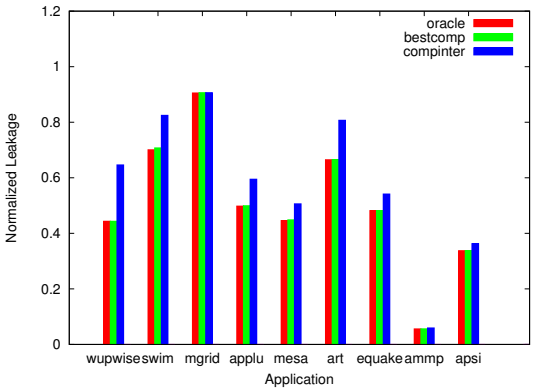

図 5.1. 理想的なスリープ制御と提案手法の比較 (FPAlu, BET=20)

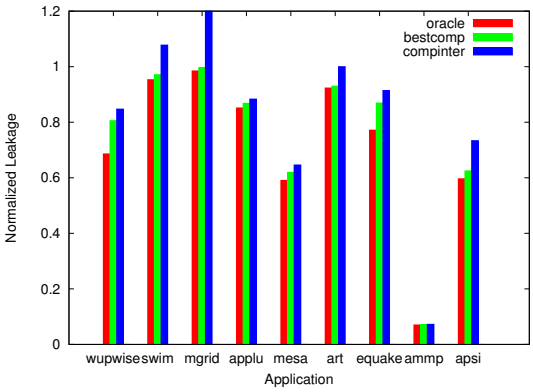

図 5.2. 理想的なスリープ制御と提案手法の比較 (FPAlu, BET=80)

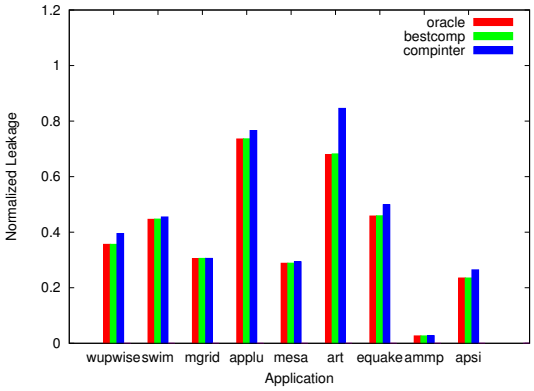

図 5.3. 理想的なスリープ制御と提案手法の比較 (FPMul, BET=20)

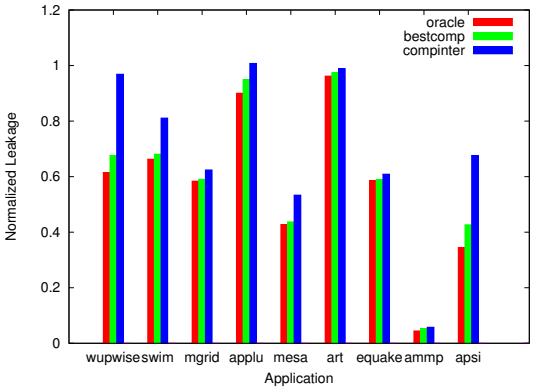

図 5.4. 理想的なスリープ制御と提案手法の比較 (FPMul, BET=80)

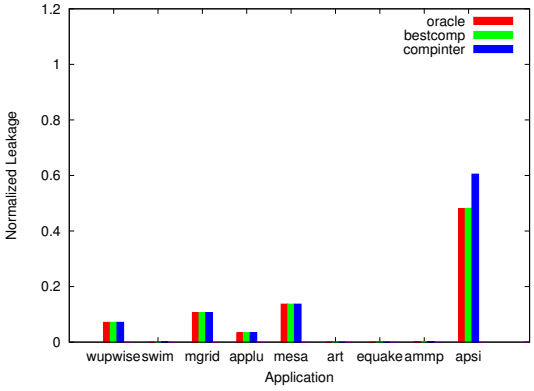

図 5.5. 理想的なスリープ制御と提案手法の比較 (IntMul, BET=20)

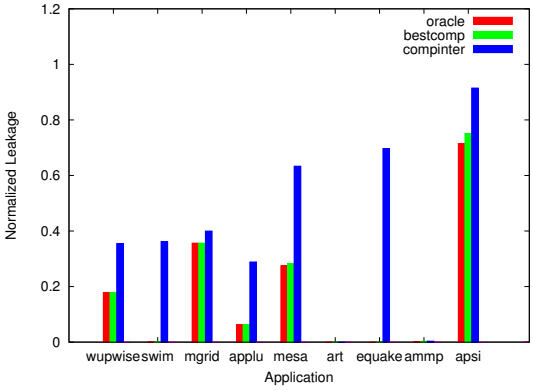

図 5.6. 理想的なスリープ制御と提案手法の比較 (IntMul, BET=80)

リープ制御 (oracle) と呼ぶことにする．

図 5.1〜5.6 に，2 つの理想的なスリープ制御，(bestcomp，oracle) と提案手法によるコンパイラ制御 (compinter) に対するアプリケーション実行時の消費リークエネルギーを示す．図は 4 章における図 4.1〜4.6 と同様のものである．縦軸が全くスリープしない場合を 1 として正規化した消費リークエネルギー，横軸がアプリケーションおよびスリープ制御手法となる．

ここでのシミュレーションでは，キャッシュミスの影響を取り除くため，L2 キャッシュ，およびメモリアクセスレイテンシを 1 サイクルとしている．この設定は現実的ではないが，提案手法によるコンパイラ制御の限界と可能性を考察するためには有効であると考えられる．その他のアーキテクチャパラメータは，表 4.1 で示したものと同様である．

メモリアクセスレイテンシを 1 サイクルとして評価を行う場合，L2 キャッシュミスをトリガーとするスリープ制御には意味がない．このためここでは，3 章で提案したスリープビットとキャッシュミス検知を組み合わせたハイブリッド制御ではなく，スリープビットによる単独の制御を提案手法として，2 種類の理想的な制御と比較している．

以下では，図 5.1〜5.6 をもとに提案手法による予測平均空き時間の精度，およびスリープビットによる静的な制御と動的な制御の比較などを議論していく．

5.1.2　空き時間予測精度の検証

まず，スリープビットによる静的な制御における削減の上限 bestcomp と，提案手法によるコンパイラ制御 compinter との差を比較する．この比較を行うことで提案手法の中で行っている空き時間予測アルゴリズムがどの程度正確に空き時間を予測できたかを知ることができる．

個別の結果を見てみると，(FPAlu，FPMul，IntMul)-20-mgrid では，どの演算器においても理想的なコンパイラ制御 bestcomp と提案手法によるコンパイラ制御 compinter の差はほとんどない．この場合には，提案した手法による予測空き時間が実行時空き時間を精度良く近似できている．

一方で，FPAlu-20-wupwise，FPMul-20-art，FPAlu-80-swim，FPAlu-80-mgrid，IntMul-80-apsi などでは，理想的なコンパイラ制御 bestcomp と提案手法によるコンパイラ制御 compinter との差が大きくなっている．特に FPAlu-80-swim および FPAlu-80-mgrid では，提案手法によるスリープ制御を行うことで，全くスリープしない場合よりもリークエネルギーが増加している．これらは，提案手法における予測平均空き時間が実行時の平均空き時間をうまく予測できなかった結果である．

図 5.7 に実際に実行された命令のうち，コード解析時の予測空き時間 ip_{pave} が実行時の平均空き時間 ip_{ave} よりも短かったものの割合を示す．ほとんどの場合において，8 割以上の命令の予測空き時間 ip_{pave} が実行時空き時間 ip_{ave} よりも短くなっていることが分かる．予測空き時間が実行時空き時間よりも短い場合には，BET 以上長い空き時間が生じているにも関わらずスリープできない場面が生じる可能性がある．すなわち，削減可能なリークエネルギーを削減できないことになる．しかし，平均空き時間 ip_{pave} を短めに評価することは，BET 未満

	FPAlu	FPMul	IntMul
wupwise	0.86 (74)	0.86 (59)	0.88 (78)
swim	0.77 (140)	0.86 (72)	0.69 (42)
mgrid	0.88 (113)	0.47 (19)	0.85 (143)
applu	0.94 (468)	0.78 (1095)	0.80 (65)
mesa	0.86 (419)	0.81 (355)	0.77 (40)
art	0.90 (72)	1.00 (57)	—- (0)
equake	0.88 (284)	0.82 (263)	0.64 (31)
ammp	0.78 (107)	0.85 (40)	0.75 (4)
apsi	0.85 (169)	0.79 (83)	0.83 (186)

図 5.7. 予測平均空き時間 ip_{pave} が平均空き時間 ip_{ave} より短かった命令アドレスの割合 (() 内は，当該演算器を使用する命令のアドレス数)

の短い空き時間においてスリープしてしまうことでエネルギー的に損をする事態を招くことはなく，またキャッシュミス検知によるスリープ制御によって有効なスリープを行える可能性が残っている，という意味で深刻な空き時間予測ミスではないと考えることができる．

一方で，予測平均空き時間が実行時平均空き時間よりも長い命令も存在することが分かる．予測平均空き時間が実行時平均空き時間よりも長い場合には，その逆の場合よりも悪影響は大きい．すなわち，この場合には BET 未満の短い空き時間しか生じない場合に演算器がスリープしてしまい余分なエネルギーオーバーヘッドが生じる可能性があるからである．実際に，FPAlu-80-mgrid や FPAlu-80-swim の場合には，このような現象が起き，全くスリープしない場合よりもリークエネルギーが増加している．

提案手法による予測空き時間が実行時空き時間とずれる原因都しては，大きく分けて３つの要因が考えられる．

１つ目は，パイプラインストールによる命令実行の遅れである．ここでの結果では，メモリレイテンシを１としているためキャッシュミスによるストールの影響は取り除いているが，データ依存や分岐予測ミスによるストールは存在している．空き時間予測の中では，命令はパイプライン中を理想的に流れるものとしてモデル化されているため，ここで予測空き時間と実行時空き時間がずれる可能性がある．このずれは，必ず予測空き時間が実行時空き時間よりも短くなる方向に働く．

２つ目は，関数ポインタを用いた関数呼び出しである．現在のコード解析の実装においては，関数ポインタを用いた関数呼び出しを検出していない．従って，関数ポインタによって，間接的に呼び出される側の関数は実質的に関数間解析が行われないことになる．4 章で示したように，関数間解析を行わない場合にはコード中において BET 以上長い空き時間が生じる部分を判別することが困難である．結果として，スリープビットを ON にすべき目的命令のスリープビットを ON にすることができず，リークエネルギー削減のチャンスを逃している．

IntMul-80-mesa においては，間接参照にによって呼び出される関数が頻繁に使用されてい

る．この関数に対しては，実質的に関数間解析が行われないため，当該関数の中の目的命令に対する予測空き時間が実行時の空き時間に比べて大幅に短くなっている．これによって，リークエネルギー削減の大きなチャンスを逃している．この差が，提案手法によるコンパイラ制御 compinter と，理想的なスリープビット制御との差に現れている．

図 5.7 において，多くの命令において予測平均空き時間が実行時平均空き時間が短くなる理由は，上記の 2 つが要因となっていると考えられる．

3 つ目は，分岐結果の偏りである．空き時間予測においては，分岐命令の分岐確率を分岐命令ごとに与えることができるようになっている．しかし，今回の評価では，これらの分岐確率を一律に $\frac{1}{2}$ としてコードを解析している．従って，ある分岐命令の分岐結果に偏りがあると，その直前部分における予測平均空き時間と実行時平均空き時間に大きな差が生じる可能性がある．実際，ループの底に存在する分岐命令など多くの分岐命令の分岐結果には，偏りがあることが知られている．分岐結果の偏りによって生じる予測平均空き時間と実行時空き時間の差は予測平均空き時間が実行時空き時間に比べて短くなる方向にも，長くなる方向にも働きうる．しかし，より深刻なのは予測が実行時のものよりも長くなるケースである．

FPAlu-80-mgrid の例では，ループの底に存在する目的命令の予測空き時間の誤差が，リークエネルギー増大の原因となっている．これらのループでは，ループ処理から抜け出した後に長い空き時間が生じる一方，ループ内では目的演算器を頻繁に利用する．現在のコード解析では，ループのバックエッジとループから抜け出すエッジの分岐確率を等しいとして解析を行っているため，当該目的命令の予測平均空き時間がループから抜け出したあとの長い空き時間の影響によって，実行時の平均空き時間よりも長く評価されてしまう．これによってループ処理の間中，空き時間が BET 未満にも関わらず，スリープモードへの遷移を繰り返しリークエネルギー増大という結果を招いている．

5.1.3　予測精度向上のためのアイディア

5.1.2 節での考察をもとに，提案手法の予測精度を改善するための方法としてに以下のようなものが考えられる．

- コード解析時に用いる命令実行モデルを見直す．パイプラインストールや命令実行にかかる遅延も考慮した命令実行モデルを構築し，各命令が実行されるサイクル数をより正確に予測する．
- エイリアス解析を用いて，関数の間接参照を考慮したコード解析を行う．
- 実行プロファイルをもとに，各分岐命令の分岐確率パラメータとして実行時の分岐確率に近いものを用いる．特にループにおける分岐確率の設定が重要であると考えられる．

これらの改善策を適用したときの，空き時間予測精度の評価は今後の課題である．

5.1.4 静的なスリープ制御手法の削減上限と動的な制御手法の削減上限の比較

ここでは，理想的なコンパイラ制御 bestcomp の結果と理想的なスリープ制御 oracle による結果を比較することで，スリープビットを用いた静的なスリープ制御の削減の上限が，動的な制御も含めたスリープ制御全体の削減の上限と比べて，どの程度のリークエネルギー削減を達成するのかを考察する．

はじめに，理想的なコンパイラ制御 bestcomp と理想的なスリープ制御 oracle の結果に差が生じる原因を考える．その原因は，同一アドレスの命令が実行された後に生じる空き時間に動的な変動が存在すること以外にはない．従って，理想的なコンパイラ制御と理想的なスリープ制御の結果に差があるアプリケーションでは，あるアドレスの命令が実行された後の空き時間が BET を跨いで変動している．従って，キャッシュミスの影響を除いた場合にこの空き時間の動的な変動が実際のアプリケーションにおいてどの程度生じているのかを見ることになる．

図 5.1〜5.6 を見ると，２つの理想的な制御戦略，理想的なコンパイラ制御 bestcomp と理想的なスリープ制御 oracle の結果には，多くの場合において大きな差が見られないことが分かる．特に，BET=20 の場合における FPAlu，FPMul，IntMul，BET=80 における IntMul では，全てのアプリケーションでその差は無視できる程度のものになっている．一方で，FPAlu-80-wupwise，FPMul-80-apsi の場合などでは，１割程度の差があることが分かる．

3 章で述べたように，本提案手法ではこのような動的な変動を無視して，同一アドレスの命令実行後の平均空き時間を，各回の演算器使用における空き時間だと思って，演算器のスリープを制御している．ここでの評価から，キャッシュミスによる影響を取り除いた場合，同一命令アドレスにおける後続空き時間の変動は一部の場合を除いて大きくないことが分かった．従って，キャッシュミス検知によるスリープ制御と同時に適用した場合，提案手法において空き時間の動的な変動を無視していることは大きな問題にはならない．

5.2 細粒度パワーゲーティングのためのコード最適化

ここまで，3 章で提案した演算器のスリープ制御手法について検討してきた．この手法は，アプリケーション実行時に生じる演算器の空き時間を予測し効率的なスリープ制御を実現する手法であり，多くのアプリケーションにおいて有効なスリープ制御を実現することが分かった．一方で，一部の場合にはそもそもリークエネルギーを削減する余地が非常に小さいということが分かった．例えば，図 5.2 からは mgrid ベンチマークにおける FP ALU では理想的なスリープ制御を行った場合でも，削減できるリークエネルギーはもとの 10% 未満であることが分かる．

そこで，ここでは空き時間を予測して演算器のスリープを制御するだけでなく，細粒度パワーゲーティングを適用したときに削減できるリーク電力が大きくなるようなコード生成/最適化を行うことを考える．

コード最適化が消費電力に与える影響を調査した先行研究としては [7] が知られている．しかし，この研究はコード最適化のダイナミック電力への影響に対する考察のみであり本研究が対象としているリーク電力を扱っていない．コード最適化がリーク電力に与える影響を調査した研究としては文献 [18] などがあり，様々な既存のコード最適化技術が細粒度パワーゲーティングを適用した場合のリーク電力削減率に及ぼす影響を調査している．しかしこの先行研究においては，リーク電力削減を目的とした特別なコード生成手法を提案しているわけではない．

この節では，演算器にパワーゲーティングを施すことを前提としてより多くのリーク電力削減を達成可能なコードを生成するためのコード生成/最適化手法について検討する．

5.2.1　細粒度パワーゲーティングに適したコードの具体例

図 5.8 に示したソースコードの例を用いて，細粒度パワーゲーティングのためのコード最適化の具体例を説明する．これは配列の要素同士の掛け算を独立に 2 つ行っているプログラムの主要部分である．このソースコードをコンパイルしたアセンブリコードには，もとのソースコード中の 2 つの乗算命令に対応するアセンブリ言語の乗算命令 (図中では，mul) が含まれる．この 2 つの乗算命令の位置として図 5.9，および図 5.10 の二つのケースが考えられる．for 文中の 2 つの文の間には依存関係がないため，このような 2 通りの乗算命令の配置を考えることができる．これらは，どちらもソースコードの意味を正しく表現することができる．

この例の 2 つのアセンブリコードを実行するときに演算器に生じる空き時間を模式図にすると図 5.11 のようになる．ラインが高くなっている部分が演算器がアクティブな状態，低くなっている部分がアイドル状態を表している．code1 の場合に存在する 2 つの空き時間が，コード 2 ではより長い 1 つの空き時間になっていることが分かる．Break Even Time の長さにもよるが，この場合 code2 の方が細粒度パワーゲーティングによる電力削減効果が大きい．すなわち，code 1 では 2 回スリープ/ウェイクアップを繰り返す必要があるのに対し，code2 では一度のスリープ/ウェイクアップだけで同じだけの時間スリープしていることができる．結果，code2 の方がエネルギーオーバーヘッドが少なくて済むため，電力削減効果が高くなる．

このように，コード中の命令の並びかたが異なればモード切り替えに伴なうエネルギーオーバーヘッドの大きさが異なってくる．直感的には，短い空き時間が少なく逆に長い空き時間は多く生じるようなコードである方がエネルギーオーバーヘッドが少ない．

通常のコンパイラでは，レジスタ使用量等を考慮して命令をスケジューリングするため，図 5.9 のようなアセンブリコードを生成する．一方で，先に説明したようにリーク電力削減を考慮した場合には，スリープ回数が少なくエネルギーオーバーヘッドを小さく抑えることができる図 5.10 の命令スケジューリングの方が好ましい．しかし，図 5.9 のスケジューリングに対して，図 5.10 のスケジューリングは性能で劣る可能性がある．例えば，乗算命令の連続実行によって資源ハザードが生じたり，レジスタ使用数が増加して余分なメモリアクセス命令が入り込むおそれがあるからである．

このように最適化や命令スケジューリングといったコード生成過程は，性能だけでなく消費電力にも影響を及ぼすと考えられる．そこで，以下の節ではパワーゲーティングのためのコー

```
for(i=0;i≤N;i++){
a[i] = b[i]*c[i]
d[i] = e[i]*f[i]
}
```

図 5.8. 例：ソースコード

BET=15 cycle

Loop Head:
アドレス計算 etc： ２０ 命令
mul
アドレス計算 etc： ２０命令
mul
branch check

図 5.9. 例：assembly code1

BET=15 cycle

Loop Head:
アドレス計算 etc： ４０ 命令
～
mul
mul
branch check

図 5.10. 例：assembly code2

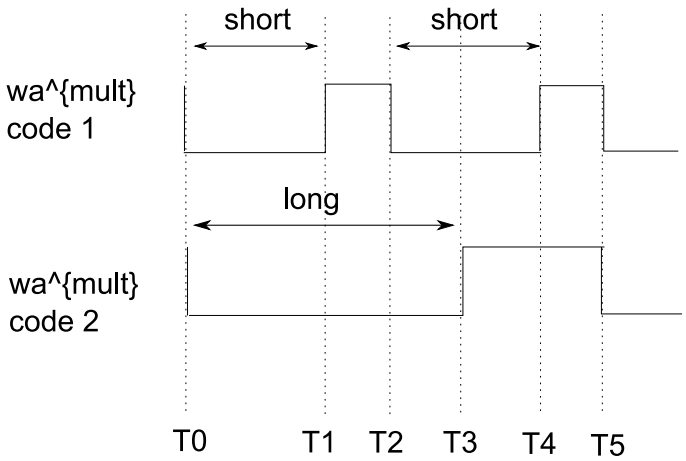

図 5.11. コードの違いによる空き時間の違い

ド最適化の可能性を検討する．

5.2.2 コンパイラによる最適化の例：命令スケジューリング手法の変更

コンパイラにおけるコード生成過程は，多くの段階に分かれており，各段階においてより良いコード (通常は，実行性能が高いコード) を生成するための様々な最適化が行われる．パワーゲーティングに適したコード最適化も同様に，コンパイラシステム内部の様々な段階において考えることが可能である．ここではコンパイラにおけるコード生成過程の中でもパワーゲーティングへの影響が大きいと考えられる目的語のコードスケジューリングに焦点を絞って，コード最適化の可能性を検討する．

コードスケジューリングとは，プログラム中の命令間の依存関係を守りながら，一列に並んだ機械語を生成する作業である．通常，スケジューリングによって得られる機械語列は一意には定まらず異なるスケジューリングアルゴリズムを用いれば，異なる並びの機械語列が生成される．そのため，命令スケジューリング手法が異なれば結果として得られるコードの実行性能や消費電力も変化することが知られている．

パワーゲーティングによるリーク電力削減を考えた場合でも，5.2.1 節で説明したように機械語列の並びによってリーク電力削減の余地が変わる．ここでは，図 5.8 のソースコードからリーク電力削減の余地が大きい図 5.10 の機械語列を生成するようなコードスケジューリング手法を検討する．

パワーゲーティングを考慮したリストスケジューリング

実用的なシステムにおける，コードスケジューリングにおいてはリストスケジューリングと呼ばれるスケジューリング手法を用いることが多い (文献 [21] 参照)．ここでは，このリストスケジューリングと呼ばれる手法を改良することでパワーゲーティングに適したコードを生成する手法を考える．ここではコード最適化の可能性を検証する最初のステップとしてベーシックブロック内におけるコードスケジューリングのみを考えている．

リストスケジューリングは，スケジュール可能な命令のスケジュールと命令間の依存関係の更新を交互に行うシンプルなスケジューリング手法である．一般的なリストスケジューリングアルゴリズムの擬似コードを以下に示す．

リストスケジューリングアルゴリズム

$ConstraintTree$：命令間の依存関係を表す木
$readyList$：スケジュール可能な命令のリスト
while $readyList \neq \phi$ **do**
　$readyList$ の中の命令を一つ選んで $next$ とする
　$next$ をスケジュール
　$next$ を $readyList$ から削除
　$next$ を $ConstraintTree$ から削除
　新しくスケジュール可能になった命令を $readyList$ に加える
end while

ここで，重要になるのが 4 行目の操作である．4 行目はスケジュール可能な命令の集合 $readyList$ の中から，実際にスケジュールする命令を一つ選択する操作を行っている．一般的に，スケジュール可能な命令は複数存在し，どの命令を優先的にスケジュールするかによって生成される機械語列が異なってくる．ここでは $readyList$ 中の命令の優先度として，一般的な優先度付けのルールに加えて下記のような優先度付けのルールを加えることでリーク電力削減に有利なコードを生成する実験を行った．

パワーゲーティングを考慮した優先度付けのルール

目的命令が一つもスケジュールされていない状態では，目的命令の優先度は最低とする．一方，目的命令が一つでもスケジュールされた場合には優先度は最高とする．

これは，ベーシックブロック内で最初に現れる目的命令から最後に現れる目的命令までの距離を短くすることを狙った直感的な手法である．

簡単なアプリケーションに対する実験

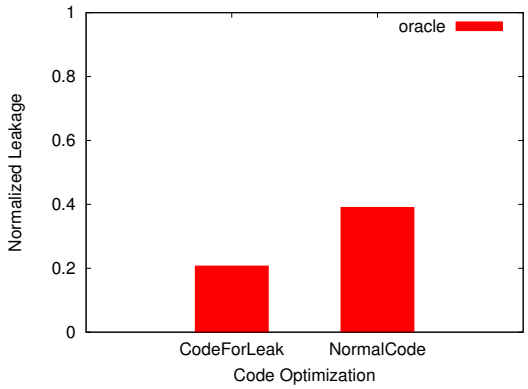

図 5.12. 配列同士の単純な計算，パワーゲーティングを考慮したスケジューリングと通常のスケジューリング，整数乗算器，BET=20cycle

5.2.2 節で説明したパワーゲーティングのためのリストスケジューリングを簡単なアプリケーションに対して，適用した場合のリークエネルギー削減率の変化を示す．図 5.12 は，5.8 に示したソースコード断片を，NormalCode – 通常のリストスケジューリングを用いてコード生成した場合，CodeForLeak – パワーゲーティングを考慮した優先度付けを行ったリストスケジューリングを用いてコード生成した場合，の 2 つの場合についてのリークエネルギーを示している．目的演算器は整数乗算器，BET=20cycle の場合に理想的なスリープ制御 (oracle) を適用した場合の結果である．シミュレーションには，4 と同様，*SimpleScalar Tool Set* [2] を利用し，パワーゲーティングを考慮した優先度付けを行うリストスケジューリングの実装には，コンパイラインフラストラクチャ *Low Level Virtual Machine*[11] を用いた．コード生成時における最適化オプションは-O0 を用いている．

図 5.12 では, 通常のリストスケジューリングにおける正規化したリークエネルギー (NormalCode) が 0.39 であるのに対し, リークを考慮したリストスケジューリング (CodeForLeak) では 0.21 となっており，リストスケジューリングアルゴリズムの変更によってパワーゲーティングによるリークエネルギーの削減効果が大きくなったことが分かる．なお，この場合 2 つのコードの実行性能に差は見られなかった．

簡単な例における実験ではあるが，ここでの実験に用いた図 5.8 で示したソースコードと類似した構造は，より一般的なアプリケーションにおいても存在する．従って，より一般的なアプリケーションにおいても今回実験を行ったコード生成手法を適用することでパワーゲーティングに適したコードを生成できる可能性があると考えられる．

第 6 章

まとめと今後の課題

本論文では，近年その増大が問題となっているマイクロプロセッサのリーク電力削減を目標にして，演算器へパワーゲーティングを適用するさいに必要となるスリープ制御手法を提案した．演算器へのパワーゲーティングの適用は，演算器を頻繁に使用するアプリケーションにおいて特に困難であり，本研究はそのようなアプリケーションを対象とした演算器の実行時スリープ制御手法を提案している．本論文では，パワーゲーティングに伴なうエネルギーオーバーヘッドを最小限に抑えつつ，十分なリーク電力削減を達成するために，コード解析に基づく演算器の空き時間予測手法を提案し，これを用いたソフトウェアベースの演算器スリープ制御手法を提案した．提案するコード解析においては，データの流れの解析を用いてリーク電力削減に効果の大きな長い空き時間を判別するため，関数間解析が行えるように工夫している．さらに，コンパイラでは対応できないハードウェアの動的な要因に起因する空き時間に対処するため，最小限のハードウェア追加で実現可能なキャッシュミス検知による動的なスリープ制御を併用するハイブリッド手法を提案した．

シミュレータを用いた評価の結果，提案手法によって多くのアプリケーションで演算器のリーク電力を大きく削減できることが分かった．特に，コンパイラによる静的なスリープ制御とキャッシュミス検知による動的なスリープ制御を併用したハイブリッド手法では，理想的にスリープ制御をした場合と同程度のリーク電力削減を達成する場合も多く，提案手法が演算器のリーク電力削減に有効であることが分かった．

今後の課題としては，まず提案した空き時間予測手法の精度のさらなる向上が上げられる．これについては，具体的なアイディアを 5.1.3 節中で示した．提案手法以外のスリープ制御手法との比較や，スリープビットやキャッシュミス検知によるスリープ制御を実現するためのハードウェア実装コストの評価なども今後の課題である．また，本論文においては単一発行のインオーダプロセッサを想定した評価を行ったが，コード解析時に仮定するアーキテクチャモデルを変更することで，提案手法をより複雑なアーキテクチャのプロセッサへ拡張することができると考えられる．特に，近年注目されている SIMD 拡張命令を有するプロセッサやアクセラレータを持つプロセッサアーキテクチャなど，演算器におけるリーク電力削減が相対的に重要なプロセッサアーキテクチャへの手法の拡張が考えられる．

また，5.2 節でその可能性に触れた演算器パワーゲーティングのためのコード最適化の問題

は，大きな可能性を持っている．非常に大きく複雑なコンパイラシステムにおいて，性能低下を抑えつつ，パワーゲーティングに適したコード最適化手法を実現することは容易なことではないが，スリープ制御だけでは達成することのできないリーク電力削減を達成できる可能性を秘めておりその価値は高いと考えられる．

謝辞

この修士２年間を通して様々な方と出会い，様々な経験をさせていただき，様々なご指導をいただくことができました。本当に幸運な２年間だったと思います。

南谷先生には，研究室ミーティングのさいに多くのご指摘やご指導をいただきました。ありがとうございました。

中村先生には，プロセッサアーキテクチャの基本な的な事柄から研究の進め方，取得したデータにたいする考察の仕方に至るまで様々なご指導をいただきました。ありがとうございました。

近藤先生には，私の研究が行き詰まったときに多くのご助言をいただきました。また，アーキテクチャに関する様々な興味深いお話を聞かせていただきました。ありがとうございました。

今井先生には，特に研究室の端末やサーバシステムに関連する事柄で多くのご助言・ご指導をいただきました。ありがとうございました。

佐々木先生には，研究室生活における事柄や論文の書き方，シミュレータの使い方など細かな事柄に渡る数多くのご助言，ご指導をいただきました。ありがとうございました。

また，いつも私のとりとめのない議論に真剣に向きあってくださった研究室の皆様に心から感謝いたします。

参考文献

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[3] J. A. Butts and G. S. Sohi. A Static Power Model for Architects. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, 2000.

[4] Y.-H. Fang, Y.-S. Hwang, Y.-P. You, and J.-K. Lee. Compiler-based vs. Hardware-based Power Gating Techniques for Functional Units. ODES-07: workshop on Optimizations for DSP and Embedded Systems, March 2007.

[5] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[6] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37, 2004.

[7] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 304–307, 2000.

[8] Y. Kanno and et.al. Hierarchical Power Distribution with 20 Power Domains in 90-nm Low-Power Multi-CPU Processor. In *Proceedings of the 2006 IEEE International Solid-State Circuits Conference*. 2006.

[9] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, 2001.

[10] N. S. Kim, T. M. Austin, D. Blaauw, T. N. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. T. Kandemir, and N. Vijaykrishnan. Leakage Current: Moore's Law Meets Static Power. *IEEE Computer*, 36(12):68–75, 2003.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[12] R. Nagpal and Y. N. Srikant. Compiler-assisted leakage energy optimization for clustered VLIW architectures. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 233–241, 2006.

[13] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81, 1998.

[14] Rajesh Kumar and Glenn Hinton. A Family of 45nm IA Processors. In *Proc. IEEE International Solid-State Circuits Conference* , Feb 2009.

[15] S. Rele, S. Pande, S. Önder, and R. Gupta. Optimizing Static Power Dissipation by Functional Units in Superscalar Processors. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 261–275, 2002.

[16] S. Roy, S. Katkoori, and N. Ranganathan. A Compiler Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, pages 215–220, 2007.

[17] N. Seki, L. Zhao, J. Kei, D. Ikebuchi, Y. Kojima, Y. Hasegawa, H. Amano, T. Kashima, S. Takeda, T. Shirai, M. Nakata, K. Usami, T. Sunata, J. Kanai, M. Kanai, M. Kondo, and H. Nakamura. A Fine-grain Dynamic Sleep Control Scheme in MIPS R3000. In *ICCD '08: Proceedings of the 2008 international conference on computer design*, pages 612–617, 2008.

[18] Soumyaroop Roy and Nagarajan Ranganathan and Srinivas Katkoori. Compiler-Directed Leakage Reduction in Embedded Microprocessors. In *Proc. IEEE International Conference on Computer Design* , Oct 2009.

[19] C.-L. Su and A. Despain. Cache designs for energy efficiency. volume 0, page 306, 1995.

[20] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 732–737, 1998.

[21] 中田 育男. **コンパイラの構成と最適化**. 朝倉書店, 東京, 1999.

発表文献

研究会・シンポジウム・口頭発表

[1] 薦田　登志矢, 佐々木広, 近藤正章, 中村宏, “リーク電力削減のためのコンパイラによるスリープ制御の初期検討”, 情報処理学会研究報告, ARC-180, pp.33–38, 2008 年 10 月.

[2] Toshiya Komoda, Hiroshi Sasaki, Masaaki Kondo, Hiroshi Nakamura, “Compiler-Directed Fine Grain Power Gating for Leakage Power Reduction in Microprocessor Functional Units”, ODES-7: 7th Workshop on Optimizations for DSP and Embedded Systems, , pp.39–48, March 2009.

[3] 薦田　登志矢, 佐々木広, 近藤正章, 中村宏, “リーク電力削減のためのコンパイラによる細粒度スリープ制御”, 先進的計算基盤システムシンポジウム SACSIS2009, pp.11–18, 2009 年 5 月.

付録 A

演算器におけるパワーゲーティングの実装：Geyser プロセッサ

ここでは本研究と関連の深い先行研究 [17] の中で報告されているマイクロプロセッサ Geyser について説明する．

Geyser は MIPS R3000 をベースにしたマイクロプロセッサである．乗算器や除算器，シフターといった演算器を細粒度にパワーゲーティングできるよう設計されている．演算パイプラインは 5 段であり，各演算器のスリープを個別に制御するためのスリープコントローラを搭載している．

このマイクロプロセッサに用いられているパワーゲーティング回路は，演算器のスリープに伴なう性能オーバーヘッドが従来に比べて小さくなるように設計されている．Geyser ではスリープ/ウェイクアップのモード切り替えを，数ナノ秒という短い時間で行うことができると報告されている．これは，従来利用されてきたパワーゲーティングのモード切り替え速度がマイクロ秒のオーダであることと比較して 100 倍以上高速なモード切り替えである．また，パワーゲーティングによる性能低下を防ぐために Geyser では演算器のプリウェイクアップが

図 A.1. Geyser チップ写真

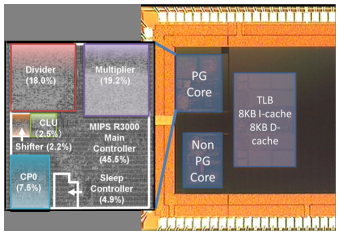

図 A.2. Geyser レイアウト写真

実装されている．演算器のスリープ/ウェイクアップを制御するスリープコントローラは，パイプラインの IF ステージにおいて命令をプリデコードし，使用される演算器にあらかじめウェイクアップ信号を送る．これにより，当該命令が IF ステージから実際に演算器を使用する EX ステージに到達するまでのサイクル (Geyser では 2 サイクル) を利用して，演算器をスリープモードからウェイクアップモードに遷移させることが可能である．従って，演算器のウェイクアップ遅延による性能オーバーヘッドを隠蔽することができる．2.3.1〜2.3.3 節で説明したプロセッサアーキテクチャ，命令セット，およびコンパイラシステムは Geyser におけるシステムを参考にしている．

Geyser のチップ写真，およびレイアウト写真を図 A.1，A.2 に示す．