Vol.2015-ARC-215 No.7
Vol.2015-OS-133 No.7
2015/5/26

Just-In-Timeコンパイラ生成コードに対する分岐命令の挙動解析に基づく細粒度PG制御

小柴 篤史[1,a)]　和田 基[1,†1]　坂本 龍一[2]　佐藤 未来子[1]　並木 美太郎[1]

概要：本研究では，Just-In-Time コンパイラによる生成コードを対象に，コンパイラによる静的な解析だけでは考慮することが難しい分岐命令の挙動を VM 内で動的に解析し，実行命令列をより適切に把握することで電力削減効果を高めることのできる Geyser 向け細粒度パワーゲーティング制御手法を提案する．提案手法を Dalvik VM に実装し，CaffeineMark，SPECJVM ベンチマークを用いてエミュレータ上で電力性能の評価実験を行い，監視する分岐命令数に制約を設けない理想条件で平均リーク電力を最大 17%削減した．

1. はじめに

回路技術が微細化し消費電力のうちリーク電力が占める割合が大きくなっている．このリーク電力を削減し省電力なシステム LSI を目指す研究が盛んに行われている [1][2][3][4][5]．本研究でも，細粒度パワーゲーティング (PG) 技術を搭載したプロセッサ「Geyser」を提案し省電力化を実現している [6][7]．本 PG 技術では，クロック単位の時間粒度で，演算器が使用される瞬間にのみ電源を供給する．しかし，電源状態が遷移する瞬間には通常時より大きなオーバヘッド電力が発生するため，頻繁な電源の ON/OFF が発生する状況では，かえって平均電力が増加してしまうという特徴がある．そこで，アプリケーションプログラムの各演算器の使用間隔を解析し，その解析情報に基づいてコンパイラや OS においてハードウェアの PG 技術の適用・抑制を制御するというソフトウェア PG 制御手法を各種提案し，ハードウェアとソフトウェアの協調制御の効果を示している [7][8][9][10]．

その協調制御の研究の一つとして本研究では，Java 等のインタプリタ型言語処理系のプログラム実行基盤に着目し，コード実行の高速化手法である Just-In-Time コンパイラ (JIT コンパイラ）の持つ動的コード生成の枠組みにおいて，動的解析に基づく PG 制御手法を提案している [10]．JIT コンパイラのコンパイルの対象となるのは，ループ処理の最内側など頻繁に実行されるコードブロックである．コードブロックは，ターゲットとするコード量は少ないが全体の実行時間に占める割合が高い．そのため，JIT コンパイラが生成したコードブロックにおいて実行時のプロセッサ状態に応じた PG 制御を施すことで，効果的な省電力化が可能となる．例えば，実行時のチップ温度や分岐命令による実行フローの変化などの各種動的要因を解析時に取り入れた PG 制御が可能である．しかし，先行研究 [10] の PG 制御では，動的な要因としてチップ温度のみを考慮する設計となっており，また，複数のコードブロックにまたがるフロー解析を行っていなかったことが原因で，逆にリーク電力が増加してしまうケースも発生した．

本論文では，JIT コンパイラの動的解析に基づく PG 制御手法において，リーク電力増加の要因となっていたフロー解析の精度に着目し，条件分岐命令の統計情報を考慮した PG 制御手法を提案する．本提案により，複数のコードブロックにまたがるフロー解析を実施し，より正確な解析から効果的にリーク電力を削減することを目的とする．提案する制御手法を Geyser 上の Dalvik VM へ実装し，条件分岐統計を考慮することによる PG 制御への効果をベンチマーク評価により検証し，その結果を示す．

2. PG 技術と関連研究

パワーゲーティング（PG）とは回路への電源電圧の供給を遮断することでリーク電力を低減する技術であり，本研究で対象としているのは，1 サイクル単位で電源遮断を可能とする細粒度 PG 技術である．本細粒度 PG 技術では，電源オフのスリープ期間と電源オンのアクティブ期間の遷移時にエネルギー的なオーバヘッドが生じる (図 1 参照)．そのため，遷移頻度を抑えつつスリープ期間をエネルギー

[1] 東京農工大学
[2] 東京大学
[†1] 現在，株式会社ドワンゴ
[a)] koshiba@namikilab.tuat.ac.jp

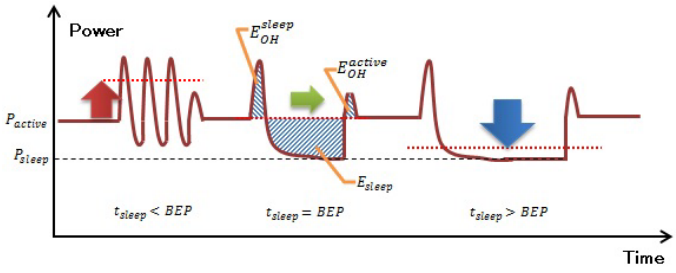

図 1　PG で発生するオーバヘッド電力と BEP の関係

的に得となる損益分岐点（Break Even Point）よりも長くするように PG を制御することがリーク電力削減につながる．そして，適切な PG 制御のためには演算器が使用される時間間隔（アイドルサイクル）をより正しく見積もることが課題となる．また，BEP はプロセッサ温度によって変化するため，PG 制御の際にはプロセッサ温度という動的要因を考慮しながらより適切な制御を行うことも必要となる．

Hu[1] は，マイクロアーキテクチャレベルで演算ユニットのアイドルサイクルを監視し，アイドルサイクルが BEP を超えるとそれを契機に演算ユニットをスリープさせる方式を提案している．しかし，BEP を超えるまでアクティブ状態を維持しておかなければならないなど柔軟な制御はできない．また，Shrivastava[2] は，すべての演算ユニットを起動させておく必要のないときに，リーク電流が大きいユニットを優先的にスリープさせるという，リークセンサを利用した低電力化手法を提案している．ランタイムの温度変化等にまで考慮しているが，温度にしか対応できないこと，PG の時間的粒度が比較的粗くなるなどの問題がある．

一方，ソフトウェアによって PG 制御を行う研究として，You[3] は，コンパイラにおけるデータフローの解析により対象ドメインのアクティビティを見積り，それに基づいて PG 制御用の命令スケジューリングを行う手法を考案している．また，Park[4] や Roy[5] は，プログラムのループ構造を解析し，演算ユニットのアイドルサイクルを予測し，それに基づいて PG 制御用の命令を挿入する手法を考案している．これらは，静的な解析手法を用いて PG 制御を図る方式であり，動的な要因を考慮していない．

本研究では，細粒度 PG 技術とソフトウェア PG 制御によるプロセッサの省電力化を目指している．そして，先行研究においてプログラムの挙動解析や動的な要因となる温度変化への追従などをソフトウェアで制御することで，従来研究よりも柔軟な細粒度 PG 制御を実現している [7][8][9][10]．特に先行研究 [10] は，インタプリタ型言語処理系の JIT コンパイラを備えた仮想マシン (VM) において，JIT コンパイラが動的に生成する各コード (JIT コード) を対象にアイドルサイクル解析を施し PG 制御を行う．VM が検出した頻繁に実行するコードブロックを対象とする PG 制御であるため，対象範囲のみの PG 制御でありながらアプリケーションプログラム全体の省電力化を図れる方式である．しかし，JIT コードのアイドルサイクル解析において，条件分岐命令の挙動や複数 JIT コードをまたぐ挙動などに対する予測が不十分なことにより，一部のプログラムで平均リーク電力を増加させていた．これを解決するため，プログラムの実行フローの変化などを動的に考慮したより正確なアイドルサイクル解析が課題となっている．

3. 目標

そこで，本論文では，JIT コンパイラが動的に生成する JIT コードを対象に，分岐命令による実行フローの変化を動的に予測することでより正確なアイドルサイクル解析を実現し，リーク電力の削減効果を図ることを目標とする．具体的には以下の項目に基づくアイドルサイクル解析と PG 制御によるリーク電力削減効果の関係を明らかにする．

条件分岐命令の挙動を考慮したアイドルサイクル解析

アイドルサイクルは実行命令流によって変動するため，分岐命令の分岐方向を正確に予測し実行される命令列を正しく予測することでアイドルサイクル解析の精度を向上させる．

複数の JIT コードにまたがるアイドルサイクル解析

JIT コンパイラを持つインタプリタ型言語処理系では，コードブロックごとに JIT コードとインタプリタを切り替えて実行するため，複数のコードブロックにまたがるアイドルサイクルを正しく予測することで，アイドルサイクル解析の精度を向上させる．

4. 研究対象システムの構成要素

4.1 動的解析と PG 制御を備えた VM 構成

本研究で提案する，動的要因に基づくアイドルサイクル解析を搭載した VM の全体構成を図 2 に示す．図 2 の星マークは本研究の新しいアイドルサイクル解析機能のために先行研究 [10] に対して追加した構成要素となる．本 VM では，JIT コンパイル直後に行う「動的コード解析処理」と，JIT コードの実行直前に行う「動的コード修正処理」の 2 段階で PG 制御を行う．なお，本設計は Geyser 上の Dalvik VM を設計基盤としているが，プロセッサに PG 技術を制御するための命令仕様や動的要因を検出するためのハードウェアサポートを備え，JIT コンパイラを備えたインタプリタ型言語処理系であれば適用可能である．

「動的コード解析処理」では，JIT コンパイラによって生成された JIT コードを対象に，フローグラフを作成し，各演算器の使用間隔情報であるアイドルサイクルを算出する．新たにフローグラフを生成する設計を加えたことで，条件分岐命令の挙動や，コードブロックをまたぐ実行フローをグラフ上で追うことができるようになり，より正確なアイドルサイクル解析を可能にしている．

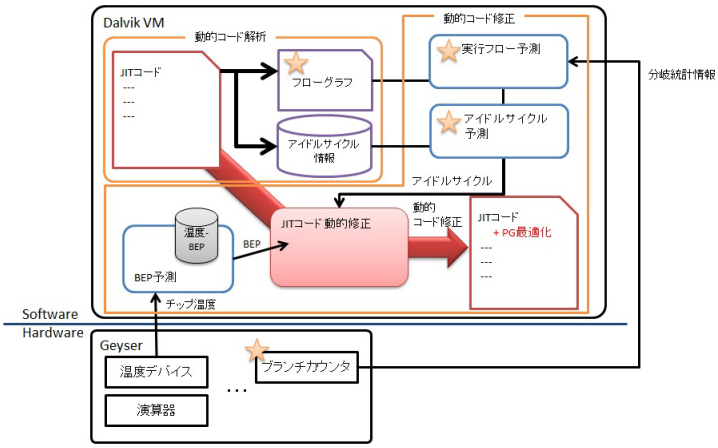

図 2　動的解析と PG 制御を備えた VM の全体構成

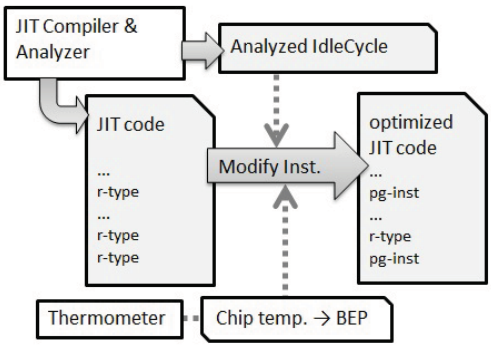

図 3　JIT コード動的修正 [10]

「動的コード修正処理」では，フローグラフやアイドルサイクル情報を用いて，JIT コードの実行フローを動的要因に基づいて予測し，PG 制御対象の演算器のアイドルサイクルをより詳しく算出する．その結果を基に，JIT コードの実行直前に JIT コード動的修正を行う．JIT コード動的修正処理では，予測したアイドルサイクルと温度によって変化する BEP とを比較して，電力的に得となるようにプロセッサの PG を制御するために JIT コードの命令の一部を PG 制御向けに書き変える（図 3 参照）．この一連の処理を JIT コードの実行直前に行う設計により，静的な解析手法では困難であった実行時の動的要因，すなわち温度変化や分岐命令の挙動などをパラメータとした PG 制御を可能にしている．

なお，本研究では分岐統計情報を得るためのハードウェアとして，第 4.3.1 項で述べるブランチカウンタを新たに設けた．JIT コードの分岐命令の分岐確率をパラメータとして実行時のフローを予測することで，より正確なアイドルサイクル解析を行えるようにしている．

4.2　JIT コードのフロー解析

本節では，動的コード解析処理におけるフローグラフの生成方式，および，アイドルサイクル情報を算出するための演算器使用情報の解析について述べる．

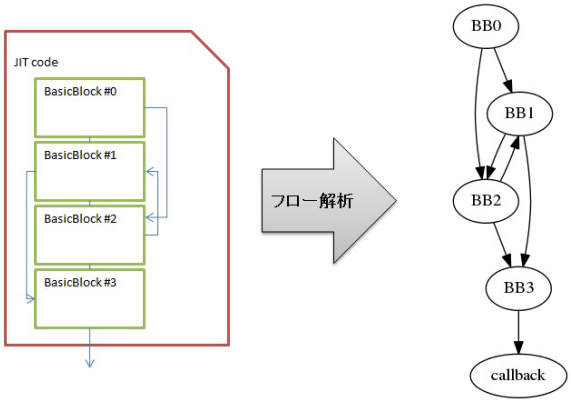

図 4　JIT コードと生成されるフローグラフの構成図

4.2.1　フローグラフの生成方式

本研究では，実行フローを表すグラフを生成することで，JIT コード中の条件分岐命令がどのような挙動をとる場合でも，その実行フローをグラフ上で追うことができるようにし，より正確なアイドルサイクル予測に役立てる．フローグラフの構成を図 4 に示す．各 JIT コードの命令列を，条件分岐命令を境としてベーシックブロック単位に分割し，分岐方向によってベーシックブロック間をつなげることでフローグラフを構成する．フローグラフの頂点は JIT コード中のベーシックブロックと 1:1 で対応させ，頂点間をつなぐ辺によってベーシックブロックの実行順序を保持する．ベーシックブロックは条件分岐命令によって区切られているため，フローグラフの各頂点からは分岐が成立した場合，しなかった場合の 2 本の有向辺がのびる．ただし，JIT コードの終端に位置するベーシックブロックはインタプリタへコールバックするため，1 本のみの有向辺がのびる．

図 4 に例示したように JIT コード中の分岐命令は，JIT コードの終端にあたるコールバックへの分岐を除き，すべて JIT コード内への分岐となる．分割されたベーシックブロック間をすべて 2 辺で接続しておくため，実行時の条件分岐がどの方向に進んでも実行フローを構成できる．

4.2.2　ベーシックブロックのアイドルサイクル解析

アイドルサイクル解析では，フローグラフ作成と同時に各ベーシックブロック内の各演算器の使用状況を事前に解析しておくことで，後のアイドルサイクル予測時の解析の手間を軽減する．アイドルサイクル解析では，PG 対象となる各演算器が使用される時間間隔を解析し，アイドルサイクルを見積もる．さらに，いくつものベーシックブロックにまたぐ任意の実行フローに対しても各演算器のアイドルサイクルを見積もれるように，次の四つの情報をベーシックブロックごとに解析しておく．

- ベーシックブロックの全命令数（以下 len と呼ぶ）
- ベーシックブロック中で PG 対象となる各演算器が最

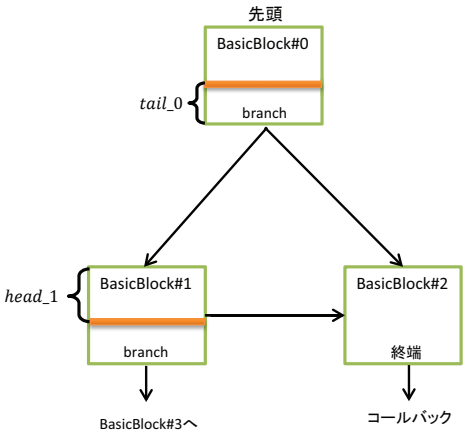

図 5　フローグラフを用いたアイドルサイクル予測の例

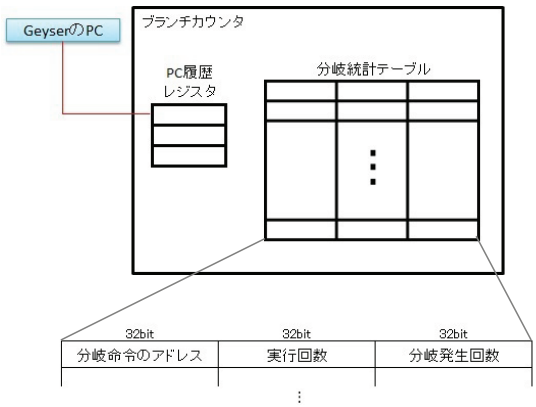

図 6　ブランチカウンタの構成と分岐統計テーブルの概念図

初に使用されるまでのサイクル数（以下 head と呼ぶ）
- ベーシックブロック中で PG 対象となる各演算器が最後に使用されてから JIT コードの実行が終了するまでのサイクル数（以下 tail と呼ぶ）
- ベーシックブロック実行後にコールバックが発生するか否かの情報

これにより，複数のベーシックブロックにまたがるアイドルサイクルも各ベーシックブロックのアイドルサイクル情報から算出可能とする．なお本解析では，Geyser プロセッサを対象にして，1 命令実行に 1 サイクルかかるものと仮定し，命令数をそのままサイクル数として解析している．

例えば図 5 に示すフローグラフにおいて，head と tail を解析しておくことで，BasicBlock #0 から BasicBlock #1 にまたがるアイドルサイクルは，BasicBlock #0 の tail（図 5 の tail0）と BasicBlock #1 の head（図 5 の head1）の和で求めることができる．各ベーシックブロックにおけるこれらの解析情報は，プロセッサの温度情報や分岐方向などの動的要因には関係しない基本情報となる．したがって，JIT コンパイル時に一度解析しておけば，ベーシックブロックを実行する度に解析し直す必要はなく，本解析情報から，温度情報にともなう BEP の変化や分岐予測結果などの動的要因をふまえたアイドルサイクル予測を行える．

4.3　PG 制御向けアイドルサイクル予測

本節では，動的コード修正処理における実行フロー予測，PG 制御向けアイドルサイクル予測について述べる．

4.3.1　ブランチカウンタを用いた実行フロー予測

本研究では，アイドルサイクル予測の精度を向上させるために，分岐命令の分岐方向を考慮した実行フロー予測を行う．分岐予測に関しては，コンパイラによる静的な分岐予測技術 [11] がある．主にキャッシュや内部リソースの最適化を目的とした手法ではあるが，本研究にも応用することは可能だと考える．しかし，JIT コード処理の動的な分岐予測処理に適用した場合，処理オーバヘッドの増加が懸念される．また，Yeh[12][13] はハードウェアで過去数十回の分岐パターンを基に次の分岐方向を予測する手法を提案しているが，ハードウェアの複雑化とリソース量の多さが問題となる．そこで本研究では，監視対象の分岐が成立することが多いのか，しないことが多いのかを単純に計測することのできるブランチカウンタを設計し，アイドルサイクル予測の精度を向上させる方針とした．

図 6 に示すように，本ブランチカウンタは分岐命令のアドレス，命令の実行回数，分岐成立回数を複数個保持することのできる分岐統計テーブルで構成される．本テーブルへ登録した分岐命令アドレスに対して，命令の実行回数，分岐成立回数をハードウェアが自動的に記録するという単純な設計としハードウェア量の軽減を図っている．また，ブランチカウンタへの分岐命令の登録，破棄をすべてソフトウェアで制御する設計としたことにより，計測対象となる分岐命令の選択に柔軟性を持たせている．

本研究では，前節で述べたフロー解析によって生成したフローグラフを，本ブランチカウンタから得られる分岐命令の分岐確率と照らし合わせ，最も起こりやすいひとつの経路を選択しながらフローを予測する．そして次節に示す方法でアイドルサイクルを算出して，適切な PG 制御へ生かす．

4.3.2　アイドルサイクル予測

ベーシックブロック内のアイドルサイクルはすでにアイドルサイクル解析において情報を採取している．しかし，条件分岐でベーシックブロックを複数またぐ実行フローにおいては，以下に示す方法によりアイドルサイクルを算出し，より適切な PG 制御を実現する．

アイドルサイクル全体の長さは，PG 対象となる演算器を使う先頭の命令と同じ演算器が使用されるベーシックブロックが実行されるまでの実行経路によって変化する．すなわち，PG 対象となる演算器を使う先頭の命令を含むベーシックブロックから BB_0, BB_1, ...，BB_n の順に実行されアイドルサイクルが終了する場合，全体のアイドルサ

Vol.2015-ARC-215 No.7
Vol.2015-OS-133 No.7
2015/5/26

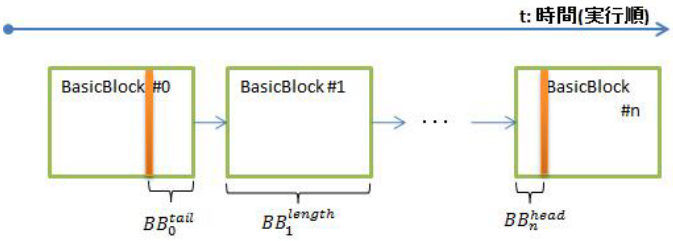

図 7　複数のベーシックブロックをまたぐアイドルサイクル予測の例

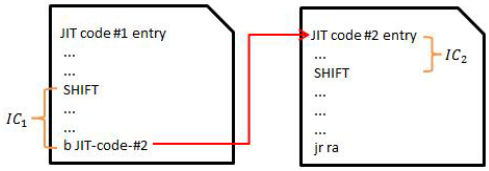

図 8　複数の JIT コードをまたぐアイドルサイクル予測の例

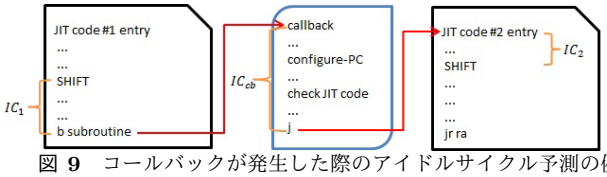

図 9　コールバックが発生した際のアイドルサイクル予測の例

イクルの長さ IC_{len} は式 (1) となる．

$$IC_{len} = \sum_{k=0}^{n} IC_{BB_k} \tag{1}$$

IC_{BB_k} が，各ベーシックブロック内がどれだけアイドルサイクルを伸ばしているかを表している．それぞれのベーシックブロックが伸ばすアイドルサイクルの長さ (IC_{BB_k}) は，次式にしたがって定める．

$$IC_{BB_k} = \begin{cases} BB_k^{tail} & (k=0) \quad (2a) \\ BB_k^{head} & (k=n) \quad (2b) \\ BB_k^{len} & (0<k<n) \quad (2c) \end{cases}$$

図 7 に複数のベーシックブロックのアイドルサイクル情報を用いてアイドルサイクル予測を行う例を図示する．このように一つの実行フロー中におけるアイドルサイクルは式 (1) によるアイドルサイクルの総和として求めることで予測可能である．一方で，実際には単一 JIT コード内でアイドルサイクルが発生するだけなく，図 8，図 9 に例示するように複数の JIT コード，もしくは，インタプリタへのコールバック関数へとまたがることも多い．そこで，本研究では，アイドルサイクル解析情報の head と tail を活用し，JIT コードをまたぐアイドルサイクルを予測する．また，コールバックが関数をまたぐ場合に関しては，コールバック処理をあらかじめ解析してアイドルサイクル予測に必要なサイクル情報（IC_{cb}）を事前に算出しておくことで対応する．

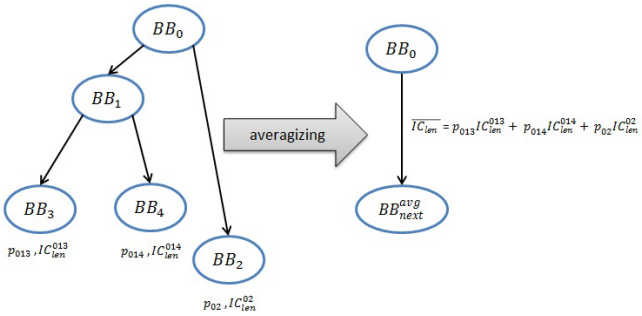

図 10　全経路のアイドルサイクル予測と平均アイドルサイクル長

4.3.3　平均アイドルサイクル長を用いたアイドルサイクル予測

本論文で示した先の実行フロー予測では，最も起こりやすいひとつの経路を分岐確率から割り出し，そのアイドルサイクルを予測して PG 制御を施す．そのため，最も起こりやすい経路が実行される場合は正しい PG 制御によってリーク電力を削減することができるが，それ以外の経路が実行された時は実行フローの予測が外れるため，適切な PG 制御を保証できない．もっとも起こりやすい経路がほぼ全体を占めるのであれば，制御ミスによるリーク電力の増加は少ない．しかし，他の経路も無視できない確率で起こる場合，その経路が実行されることも考慮にいれてアイドルサイクルを決定することが望ましい．そこで，全経路の情報を取り入れた平均的な PG 制御を行うために，「平均アイドルサイクル長」を用いたアイドルサイクル予測方式を別途設計した．本方式では，PG 制御対象となる演算命令実行後の全経路のアイドルサイクルを算出し，その平均値を PG 制御に用いる．

全経路の各アイドルサイクル長 IC_{len}^{path} を，発生確率 p_{path} を係数として線形結合した値を平均アイドルサイクル長 $\overline{IC_{len}}$ は次式で表せる．

$$\overline{IC_{len}} = \sum_{path} p_{path} \times IC_{len}^{path} \tag{3}$$

この $\overline{IC_{len}}$ をアイドルサイクル予測結果とし PG 制御に用いる．例えば，図 10 の左図に示すフローグラフにおいては，$BB_0 \rightarrow BB_1 \rightarrow BB_3$，$BB_0 \rightarrow BB_1 \rightarrow BB_4$，$BB_0 \rightarrow BB_2$ の 3 つの経路が存在し，それぞれ p_{013}，p_{014}，p_{02} の確率で実行されるとする．このとき，右図のように全経路を $BB_0 \rightarrow BB_{next}^{avg}$ という 1 つの経路にまとめたとして，この経路のアイドルサイクルを，

$$\overline{IC_{len}} = p_{013}IC_{len}^{013} + p_{014}IC_{len}^{014} + p_{02}IC_{len}^{02} \tag{4}$$

で算出して PG 制御に用いる．前節のアイドルサイクル予測と本節の平均アイドルサイクル長でのアイドルサイクル予測とでは，実行するプログラムによって省電力効果の違いが生じると思われる．次章にて評価結果を示す．

5. 評価

5.1 評価環境

本評価で用いた開発環境，評価環境を**表 1** に示す．提案する PG 制御手法を実装した Dalvik VM を Android システム上で動作させる．Android システムは QEMU をベースに Google 社によって改変された Android Emulator(以降単にエミュレータと記す) 上で動作させる．Android 上のシェルからベンチマークを実行させ，エミュレータがシミュレートしたパワーゲーティングによるスリープ分布をもとに，平均リーク電力の評価を行う．

表 1 開発環境と評価環境

名称	製品名など
評価環境	Android Emulator
CPU アーキテクチャ	MIPS32 Rev2
細粒度 PG 方式	Geyser アーキテクチャ
Android システム	MIPSAndroid 4.2.1 (JB)

本評価では実際の消費電力を計測するかわりに，ベンチマーク実行時のパワーゲーティングの挙動を QEMU に搭載したパフォーマンスカウンタを用いて記録し，平均リーク電力 $\overline{P}$ を次式で算出する．

$$\overline{P} = \frac{\overline{P_{sleep}} \times T_{sleep} + \overline{P_{active}} \times T_{active}}{T_{total}} \quad (5)$$

ここで，T_{total} はベンチマーク実行にかかった総サイクル数，T_{sleep} は総スリープサイクル数，T_{active} は総アクティブサイクル数である．また，P_{active} はアクティブ時の平均リーク電力，P_{sleep} はスリープ時の平均リーク電力を表す．P_{sleep} はベンチマーク実行時のスリープ分布に基づき，(6) 式で求められる．

$$P_{sleep} = \frac{\sum_i (P_{sleep_i} \times T_{sleep_i})}{\sum_i T_{sleep_j}} \quad (6)$$

P_{sleep_i} は i サイクルスリープにおける 1 サイクルあたりの平均リーク電力を表している．この各スリープ長さ毎の平均リーク電力 (P_{sleep_i}) とアクティブ時の平均リーク電力 (P_{active}) は，Synopsys 社の Power Compiler を用いた Geyser アーキテクチャのシミュレーションから求めた値であり，文献 [14][15] に示されている．

評価実験のベンチマークプログラムとして，CaffeineMark v3.0（Logic, Loop, Method, Sieve）と SPECJVM98（compress）を用いた．各ベンチマークをエミュレータ上で複数回実行し，プロセス全体の平均リーク電力の平均を算出する．25 °C，65 °C，100 °C，125 °C の四つの温度下で，提案手法によるソフトウェア制御の有無による平均リーク電力の削減率を評価する．

評価においては二種類の実験を行う．一つは，ブランチカウンタを用いて高い分岐確率を示す方向へ分岐すると仮定したアイドルサイクル予測に基づく PG 制御の実験，もう一つは，平均アイドルサイクル長を用いたアイドルサイクル予測に基づく PG 制御の実験を行う．

5.2 分岐確率に基づくアイドルサイクル予測の評価実験

5.2.1 概要

本実験では，分岐確率に基づくアイドルサイクル予測による PG 制御を行う場合の平均リーク電力の削減率を評価し，本提案方式の効果を示す．また，これと同時に，ブランチカウンタの実装個数が，PG 制御の効果に影響するかを調べるために，全分岐命令の分岐確率を同時に計測できる理想的な場合と，有限個のブランチカウンタしか使えないとして，ブランチカウンタで監視する命令を切り替えながら PG 制御を行った場合の電力削減率について評価する．

5.2.2 実験結果と考察

まず，無限個のブランチカウンタの場合，全ベンチマークの全温度で平均約 4%の削減率を達成し，削減率が最大となったのは 100 °C における Sieve ベンチマークで約 17%の削減に成功している．また，先行研究 [10] の方式と比較した場合，Logic，Loop の 2 種類は本研究の提案手法によって更にリーク電力の削減率が向上され，平均リーク電力が増加してしいた Method ベンチマークにおいても，演算器別にみると Shift で平均 10%，全演算器合計では平均 1%の平均リーク電力を削減できた．しかし，Sieve ベンチマークでは，制御なしと比較して本研究の手法でリーク電力の削減に成功したものの，削減率は先行研究の結果に劣るものとなってしまった．また，compress に関してはほぼ横ばいという結果になった．PG 制御関連の処理を解析したところ，プログラム実行時の分岐挙動を計算する際に使っている Div の電力が増加するなど，PG 制御で発生するオーバヘッドが原因となっていた．本提案方式は，適切なアイドルサイクル予測は行えるものの，そのために要する電力が増加するため，PG 制御に見合う省電力化可能なプログラムに対しては，効果的な方式であるといえる．

次に，ブランチカウンタの個数を変化させた場合，どの個数においても 65 °C における Sieve ベンチマークで最もリーク電力を削減でき，削減率は 9.0〜9.2%となった．また，無限個のブランチカウンタを備えるとした理想値に削減率が最も近かったのは，カウンタ数 8 個および 64 個，100 °C 環境における Sieve ベンチマークであった．図 11 に Sieve ベンチマークの結果を示す．この結果は，本提案方式を適用した PG 制御において，ブランチカウンタの個数を増やしてもリーク電力削減効果は極端に向上しないことを示している．この傾向は他のベンチマークも同様であり，JIT コード中の分岐命令全ての分岐確率を詳細に反映できなくても PG 制御の効果が得られるということを示している．

この原因として，JIT コードひとつひとつの長さ (命令

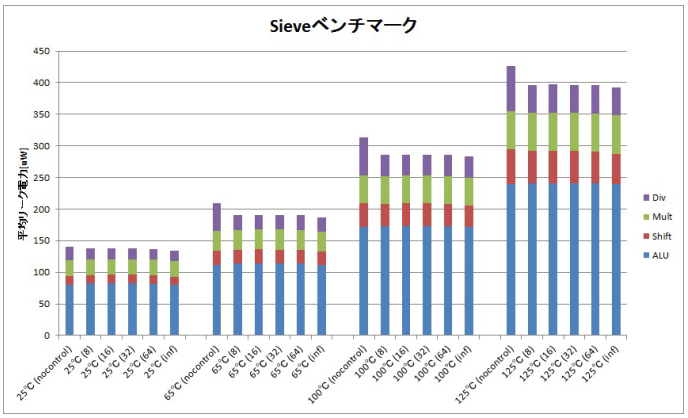

図 11　Sieve ベンチマークの平均リーク電力

表 2　JIT コードの平均命令数

ベンチマーク	平均命令数
Logic	35.0 命令
Loop	56.9 命令
Method	51.3 命令
Sieve	50.3 命令

数) が短いことが影響したと考えている．**表 2** に，評価実験で用いた Logic, Loop, Method, Sieve ベンチマークの JIT コードの平均命令数を示す．Dalvik VM の JIT コンパイラはバイトコードのベーシックブロック単位でコンパイルするため，生成される JIT コードが比較的短い傾向にある．そのため，JIT コード実行中に発生するアイドルサイクルが BEP を超えるか超えないかを決める要因が，ベーシックプロック中の局所的な分岐ではなく，インタプリタの挙動におけるコールバックの発生や JIT チェインなどであったと考えている．ただし，Oracle 社の提供している JavaVM である HotSpotVM など，メソッドベースな JIT コンパイラを採用している言語処理系においては，JIT コンパイルをメソッド単位で行うため，各 JIT コード長が長い傾向にある．この場合，JIT コード中での分岐命令の挙動によってアイドルサイクルが大きく変わることが予想される．このようなメソッドベースの JIT コンパイラを持つインタプリタ型言語処理系においては，本提案手法を用いて分岐命令の挙動を動的に監視し，アイドルサイクル解析の精度を向上させることにより，PG 制御による省電力効果を高めることができると考えられる．

5.3　平均アイドルサイクル長を用いた評価実験

5.3.1　概要

項で述べた平均アイドルサイクル長を用いた PG 制御方式と，前節で評価した分岐確率に基づくもっとも起こりやすい経路を用いた PG 制御方式を，それぞれソフトウェアでの PG 制御を行わない方式の平均リーク電力で比較し，PG 制御方式の違いを考察する．

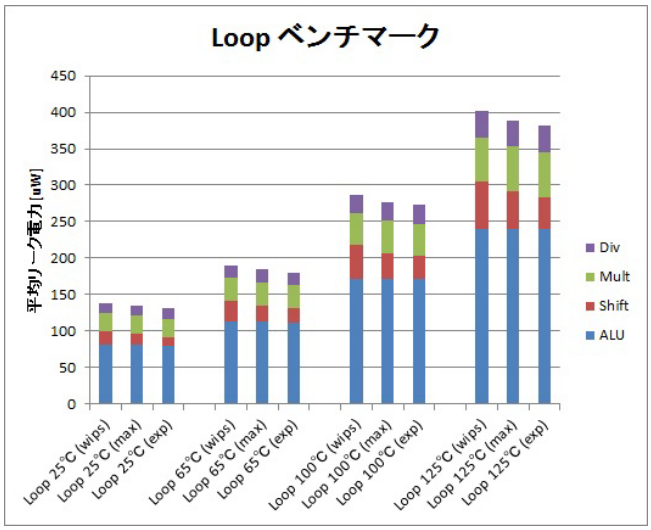

図 12　二つの方式による loop ベンチマークの平均リーク電力

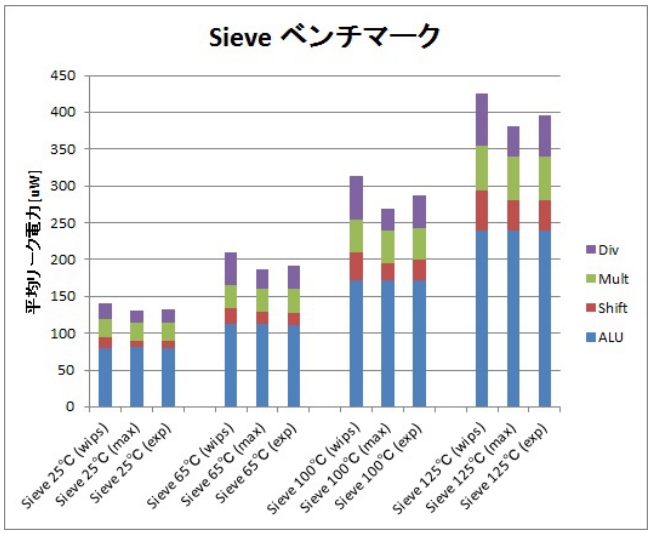

図 13　二つの方式による Sieve ベンチマークの平均リーク電力

5.3.2　実験結果と考察

Caffeine Mark の四つのベンチマークと SPECJVM より一つのベンチマークのうち，平均アイドルサイクル長を用いることで平均リーク電力を削減できたのは，Logic, Loop, Method ベンチマークであった．残りの二つのうち Shieve は分岐確率に基づく方式のほうが平均リーク電力を削減できており，compress はどちらの方式でも違いはなかった．図 12，13 に Loop と Sieve ベンチマークの結果を示す．

平均アイドルサイクル長を用いた PG 制御を行うことによって，全演算器合計の削減率は 2.2%から 3.5%に向上し，特に Shift 演算器の削減率は 9.8%から 20.0%に向上する．しかし，一方で，平均アイドルサイクル長を用いて平均リーク電力が増加してしまった Sieve などは，Div 演算器の削減率は小さくなっている．特に 100 °C における Sieve ベ

ンチマークでは，分岐確率に基づく方式では 50.5%の平均リーク電力が削減できていたが，平均アイドルサイクル長を用いたところ 29%にまで抑えられてしまった．これは，PG 制御によるオーバヘッドが一つの要因だと考えている．平均アイドルサイクル長を求める際に Div 演算器を多く使うため，ここで BEP 未満のスリープが発生しリーク電力が増加したと考えられる．

このように，ベンチマークプログラムによって適用するべきアイドルサイクル予測方法が異なるということが，PG 制御効果の結果より明らかになった．本研究では二つの方式を評価したが，まだ未着手となっているキャッシュミスといった動的要因を加味する方式など，様々なアイドルサイクル予測方法を提案することができる．またそれらの方式をプログラム実行時に選択することができれば，プログラムの挙動に応じて適切な PG 制御を行えるプログラム実行基盤を提供することも可能だと考える．

6. おわりに

本研究では，JIT コンパイラの持つ動的コード生成機構に着目した，動的なコード解析に基づく PG 制御手法の提案，評価を行った．コード解析をプログラムの実行時に行うことによって，動的な要因として今回は分岐命令の分岐統計情報を反映したアイドルサイクル予測を実現した．また，アイドルサイクル解析に反映する分岐統計情報を実行時に計測するため，研究対象プロセッサ向けにブランチカウンタを設計した．提案する PG 制御を実装した Dalvik VM と QEMU によるエミュレーションにより行った評価実験では，JIT コード中の分岐命令の挙動を全て監視することができると仮定した場合はシステム全体で平均 4%，最大で 17% の平均リーク電力削減を実現した．そして，先行研究 [10] の方式と比較して，本提案手法によって更にリーク電力を削減可能なことも示した．さらに，ブランチカウンタによって同時に計測できる分岐命令の個数を変化させた評価実験では，平均リーク電力の削減率に有意差が見られず，その原因が JIT コンパイル対象となるコードブロックの長さに関係すると考察した．本評価実験では二種類のアイドルサイクル予測方法で PG 制御を行い，それぞれの方式で省電力効果に違いが生じた．今後は，さらにアイドルサイクル予測の精度を上げるための手法の提案や，VM 自身の PG 制御などにより，さらなる省電力化を目指す．

謝辞

本研究は JSPS 科研費基盤研究 S 25220002 の助成を受けたものである．

参考文献

[1] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose: “Microarchitectural techniques for power gating of execution units”, In Proc. of the 2004 International Symposium on Low Power Electronics and design, pp. 32?37, 2004.

[2] A. Shrivastava, D. Kannan, S. Bhardwaj, and S. Vrudhula: “Reducing functional unit power consumption and its variation using leakage sensors”, IEEE Transactions on VLSI Systems, Vol. 18, No. 6, pp. 988?997, 2010.

[3] Yi-Ping You, C. Lee, and J. K. Lee: “Compilers for leakage power reduction”, ACM Transactions on Design Automation of Electronic Systems, Vol. 11, pp. 147?164, 2006.

[4] Hanmin Park, Jong Kyung Paek, Jinho Lee, and Kiyoung Choi: “Leakage power reduction of functional units in processors having zero-overhead loop counter”, SoC Design Conference (ISOCC) 2009 International, pp.492-495, Nov. 2009.

[5] S. Roy, N. Ranganathan, and S. Katkoori: “A Framework for Power-Gating Functional Units in Embedded Microprocessors”, IEEE Transaction of VLSI Systems vol.17, pp.1640-1649, Nov. 2009.

[6] N. Seki, L. Zhao, J. Kei, D. Ikebuchi, Y. Kojima, Y. Hasegawa, et al.: “A Fine Grain Dynamic Sleep Control Scheme in MIPS R3000”, The 26th IEEE International Conference on Computer Design(ICCD2008), pp.612-617, 12-15 Oct. 2008.

[7] M. Kondo, H. Kobyashi, R. Sakamoto, M. Wada, J. Tsukamoto, M. Namiki, et al.: “Design and Evaluation of Fine-Grained Power-Gating for Embedded Microprocessors”, DATE 2014, pp.1-6, Mar. 2014.

[8] 小林 弘明, 佐藤 未来子, 天野 英晴, 近藤 正章, 中村 宏, 並木 美太郎: “Linux における細粒度パワーゲーティング制御向けコードの実行時管理機構”, 先進的計算基盤システムシンポジウム (SACSIS) 2012, 2012-05-18.

[9] 小柴 篤史，塚本 潤, 和田 基, 坂本 龍一, 佐藤 未来子，小坂 翼，他: “Linux スケジューラによるリークモニタを用いた細粒度パワーゲーティング制御手法と実チップにおける評価”, 情処研報 Vol.2014-OS-129, No.14, pp.1-9, 2014-05-07.

[10] Motoki Wada, Mikiko Sato, Mitaro Namiki: “A Fine Grained Power Management supported by Just-In-Time Compiler”, IEEE Symposium on Low-Power and High-Speed Chips(CoolChips) XVII, Session XIII, No.3, Apr. 16, 2014.

[11] Brian Deitrich, Ben-Chung Cheng, and Wen mei Hwu: “Improving static branch prediction in a compiler”, In Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques, pp. 214-221, 1998.

[12] Tse-Yu Yeh, et al.: “Two-Level Adaptive Training Branch Prediction”, MICRO 24 Proceedings of the 24th annual international symbosium on Microarchtecture, pp.51-61, 1991.

[13] Tse-Yu Yeh, et al.: “Alternative Implementations of Two-Level Adaptive Branch Prediction”, ISCA ’92 Proceedings of the 19th annual international symposium on Computer architecture, pp.124-134, 1992.

[14] 中田光貴, 白井利明, 香嶋俊裕, 武田清大, 宇佐美公良, 関直臣, 他: “ランタイムパワーゲーティングを適用した回路での検証環境と電力見積もり手法の構築”, 信学技報, vol.107, no.414, VLD2007-111, pp. 37-42, Jan. 2008.

[15] 白井利明，香嶋俊裕，武田清大，中田光貴，宇佐美公良，長谷川揚平，他: “ランタイムパワーゲーティングを適用した MIPS R3000 プロセッサの実装と評価”，信学技報, vol.107, no.414, VLD2007-112, pp. 43-48, Jan. 2008.