卒業研究論文

負荷情報を用いた
動的ステージ統合による省電力化

指導教員　　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 17 年度入学　17115107 番

中村　浩俊

平成 21 年 2 月 10 日

負荷情報を用いた動的ステージ統合による省電力化

中村　浩俊

内容梗概

近年，プロセッサの高性能化に伴い，消費電力は増大する傾向にある．これに対して，現在，動的な電源電圧制御手法 (DVS) が導入されている．DVS は，プロセッサ負荷に応じて，電源電圧とクロック周波数を変化させ省電力化する手法である．しかし，DVS はプロセッサ製造技術に依存し，閾値電圧の削減率の減少から将来的に電力削減効率の低下が予測されている．そこで，プロセッサ製造技術に依存することのない，DVS に変わる省電力化手法として動的パイプラインステージ統合 (PSU) が提案されている．PSU はパイプラインステージを統合することで省電力化し，プロセッサ負荷に応じて動的にクロック周波数と統合状態，通常状態を切替えることで性能の低下を防いでいる．負荷の上昇時には通常状態へ切替え，負荷の減少時には統合状態へ切替える．しかし，PSU は命令実行をしつつ状態の切替えを行うサンプリングフェーズと命令の実行を行うだけの実行フェーズの 2 フェーズからなっており，実行フェーズでは状態の切替えを行えない．このことから，実行フェーズ直後の負荷の上昇では統合状態から通常状態への切替えが行えない．これにより，消費エネルギーが増え，効率が低下する恐れがある．そこで本研究では，この PSU の効率の低下を防ぐ新たな手法を提案する．負荷が上昇したときに実行した命令の IPC をここでは負荷情報と呼ぶ．本研究の目標は負荷情報を PSU に導入することにより PSU の効率の改善を図ることである．そのために，記憶バッファの実装を行った．記憶バッファとは，過去の負荷情報を記憶しておくバッファのことである．

評価には SimpleScalar-PISA 命令セットシミュレータを用い，実行は out-of-order 実行で行った．従来の PSU に対し，負荷情報の導入を行ったことで，SPEC95 ベンチマークでは従来手法と比較して，実行時間は 27 %減，消費エネルギーは 5 %増となる場合があった．また，平均で実行時間は 11 %減，消費エネルギーは 3 %増となり，プロセッサ負荷の増加を防ぐことが出来た．

負荷情報を用いた動的ステージ統合による省電力化

目次

1　はじめに

近年、プロセッサの高性能化に伴い，消費電力は増大する傾向にある。しかし，これまでのプロセッサ研究は，性能向上技術に着目したものが大部分であった．一方で，現在ではモバイルコンピュータや組み込みシステムに高性能なプロセッサを搭載するために，プロセッサの省電力化への要求が強まっている．省電力化により，より小さい電源を使用できるため，モバイルプロセッサの小型化，消費エネルギー抑制が可能となり，さらに放熱ファンなどに求められる性能も低減できるため，静音化の観点からも重要である．また，従来よりも少ない電源容量で従来と同じ駆動性能を確保できるということからコスト削減の面でも有効である．

現在のモバイルプロセッサでは，この問題に対し，クロック周波数とプロセッサの電源電圧を動的に変更することで対処している．この手法は **DVS(Dynamic Voltage Scaling)** と呼ばれ，プロセッサ負荷の低い時は電源電圧とクロック周波数を下げて省電力化する手法である．例えば，Intel のプロセッサには，EIST と呼ばれる DVS 技術が採用されている．DVS は，消費電力を削減できる有効な手段であるが，プロセス技術の進歩とともに削減できる電力が減少する可能性が考えられる．この先のプロセス技術の進歩に伴い，最大電源電圧は下がっていくと考えられる．しかし，閾値電圧のスケーリングファクタは小さくなっていくと予想され，このため最大電源電圧と閾値電圧の差が少なくなるためである．

このような背景から，DVS に代わるエネルギーの削減手法として，**PSU(Pipeline Stage Unification)** と呼ばれるパイプラインステージを動的に統合する手法が現在提案されている．近年，プロセッサの命令を処理するユニットは，多くのパイプラインステージに分割されている．例えば，Intel Pentium4 は 20 段に分割されている．このような多段のパイプラインは高いクロック周波数を達成するために必要である．また、高いクロック周波数は高い処理性能を達成するために必要である．このことから，高い処理性能が要求されない場面では多段パイプラインでなくともよい．逆に、多段のパイプラインでは分岐予測ミスペナルティやキャッシュアクセスレイテンシ等が大きいため，同一のクロック周波数でパイプライン段数の少ないプロセッサよりも **IPC(Instruction Per Cycle)** は，低下することとなる．そこで，PSU はプロセッサ負荷が低い時や分岐ミスが起こった時などに，クロック周波数を低下させると同時に，パイプラインを統合させる．これにより，IPC の向上が図れ，クロック周波数を下げたことによる消費電力削減効果も得ることが出来る．また，一般的に，パイプライン段数が少ない方

が分岐ミスペナルティは少ない．PSU はステージの統合によりパイプライン段数を一時的に少なくし，分岐ミスペナルティを削減している．さらに，ステージの統合によりバイパスされたパイプラインレジスタの消費電力も削減できる．過去の研究により，PSU は DVS より低消費電力であると確認されている．このことから，本研究では PSU に着目する．

しかし、現在提案されている PSU はあまり効率的ではない．現在の PSU では定期的に IPC を調査し，調査により得られた情報によりステージの統合を行うかどうかを判断している．しかし，クロック周波数を低下させるタイミングによっては，IPC を定期的に調査することで削減できる消費エネルギーに対し，実行時間が増加してしまう場合がある．このため，本研究では，動的パイプラインステージ統合に負荷情報を導入する．動的に記憶された負荷情報を使用することでパイプラインステージを統合するかどうかの判断を補助し，現在の PSU による実行時間の増加を抑えたい．

本論文では，研究背景として従来手法である DVS と PSU を説明し，その問題点と本研究の目的を述べる．その後，提案手法を述べ，具体的な実装方について述べる．最後に，シングルプロセッサのシミュレータである **SimpleScalar**[1] を用い，SPEC95 CINT ベンチマークプログラムによるシミュレーションによる評価を行い，考察する．

2　研究背景

本研究の背景として，従来手法の動作原理を概説し，その問題点について説明する．

2.1　従来手法

2.1.1　DVS

近年のモバイルプロセッサでは，DVS が導入されており，DVS により低消費電力を実現している．DVS とは，OS によるマイクロプロセッサの電力管理技術の一つであり，動的にクロック周波数と電源電圧を変更し消費電力を削減する手法である．このときの電源電圧の変化幅は限られており，プロセッサが通常動作時に消費している標準の電力とプロセッサが動作するために最低限必要な閾値電力の間で変化させる．CMOS プロセッサの消費電力式を以下に示す．

$$P = pCV^2 f \tag{1}$$

P：消費電力
p：スイッチングするトランジスタの割合
C：総キャパシタンス
V：電源電圧
f：クロック周波数

式 (1) より分かるように，消費電力は，クロック周波数を低下させればそれに比例して低下し，また電源電圧を低下させればその 2 乗に比例して低下する．CMOS 集積回路の時間遅延は，電源電圧にほぼ比例するため，クロック周波数と電源電圧を比例させればプロセッサは正常に動作する．これを利用し，バッテリ持続時間要求が強い時，またはプロセッサに与えられた負荷が低い時に，クロック周波数を低下させ，消費電力を削減する．さらにクロック周波数を低下させることで，クロックサイクル時間が延びる．クロックサイクル時間が延びることにより，命令信号の遅延が発生する．そのため，信号を周波数低下後のクロックに合わせ，電源電圧を低下させる．これにより，プログラム実行に要する消費電力を削減する．また，プロセッサに与えられた負荷が高くなると，クロック周波数を上昇させ，それにより短くなったクロックサイクル時間に命令信号を合わせる．このように，プロセッサに与えられた負荷が低いときにのみクロック周波数を下げることで処理性能を落とし省電力化することで，省電力化しつつ実行時間の増加を抑えている．

2.1.2 PSU

前述の DVS は，消費電力を削減する有効な手法であるが，次節で説明する問題点があるため，DVS に代わる省電力手法として PSU が提案されている．PSU は，動的にパイプラインステージを統合する手法である．PSU は，プロセッサ負荷が低い時，クロック周波数を低下させるという点では，DVS と同じである．しかし PSU は，クロック周波数を低下させ、それにより延長したクロックサイクル時間を利用して複数のパイプラインステージを統合するという点で DVS と異なっている．

実際に PSU を実現するには，統合するステージ間のパイプラインレジスタをバイパスする回路が必要になる．図 1 に PSU に関連する信号線 とパイプラインレジスタの結合の様子を表す．ステージを統合していない状態を通常状態，統合している状態を統合状態と呼ぶこととする．

説明を簡単化するため，LogicA と LogicB，2 ステージの統合例を示す．図 1 から分かるように，パイプラインレジスタには，クロックドライバからの信号が入力されて

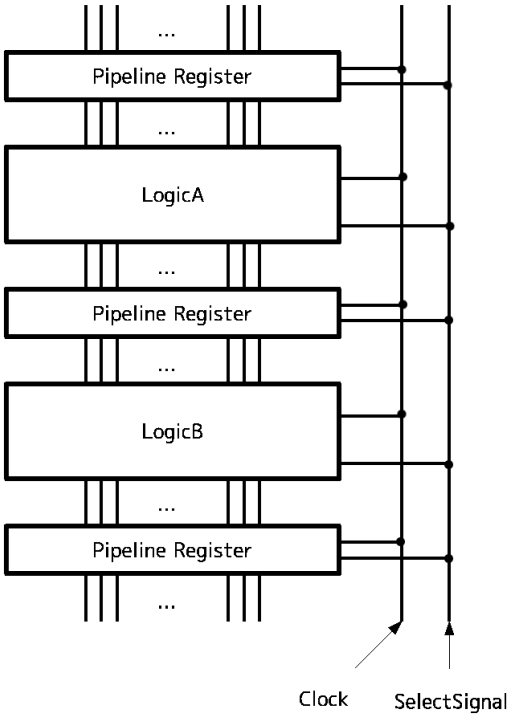

図 1: パイプライン通常

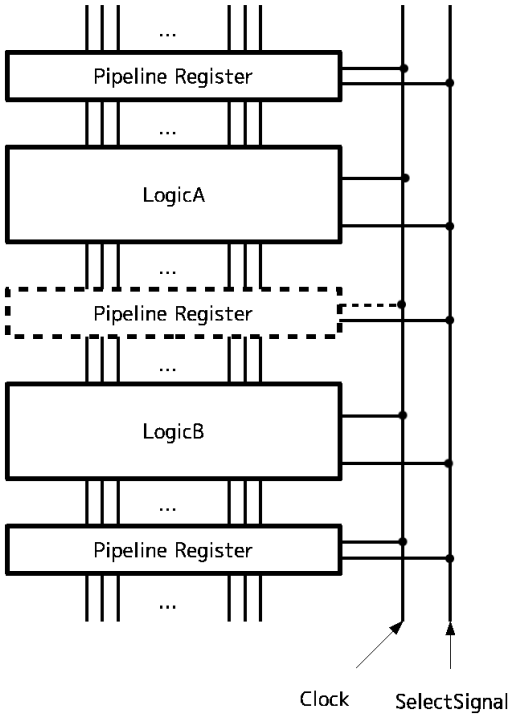

図 2: パイプライン統合

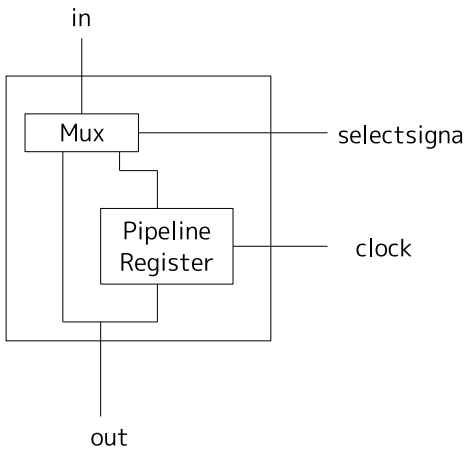

図 3: パイプラインレジスタ

いる．また，PSU のための信号線として，統合用の信号線が追加されている．図 1 はステージを統合していない状態を，図 2 は統合した状態を表す．図中の実線部分は動作部分を表し，破線部分は動作していない部分を表す．

ステージを統合していない場合，隣接する組み合わせ論理回路 A と B は，回路の間

のパイプラインレジスタが動作しており，パイプラインレジスタにLogicAの出力を一度格納するため，論理回路Aと論理回路Bは異なったステージとして動作する．それに対して，ステージを統合している場合，組み合わせ論理回路AとBの間のパイプラインレジスタへ統合信号が入力されるため，パイプラインレジスタは動作しない．図3から分かるように，PSUのためにパイプラインレジスタにはマルチプレクサが追加されている．マルチプレクサにステージ統合信号が入力されることでパイプラインレジスタがバイパスされ，2つの論理回路は1つのステージとして動作する．

パイプラインを統合することで省電力化を実現できる理由は2つある．1つは図1,2から分かるように，統合することでパイプラインレジスタを使用しなくなり，パイプラインレジスタに使用していた電力を省電力化できるためである．

2つ目に，統合によりプロセッサ全体のパイプラインが短くなることで，次のような利点がある．図4のようにパイプラインが動作しているとする．一段目のF，D，I，Eのステージと，二段目のF，D，Iステージ，三段目のF，Dステージは既に動作済みであるとする．現在動作しているのは，一段目の2つ目の実行ステージの列である．図4のように，現在動作している，二段目のパイプラインで分岐ミスが起きたとする．点線のステージは，分岐ミスが起こらなかった場合に動作する予定のステージを表している．通常状態では，図4の様に分岐ミスが起きた時，廃棄されるステージはパイプライン三段目のF，D，Iの3ステージである．1ステージは1サイクルで動作するので，このとき，ストールサイクル数は3サイクルである．しかし，図5に表すように統合された状態ではF，D，Iは1ステージに統合されているので，3ステージが1サイクルで動作する，このことから統合された状態ではストールサイクル数は1サイクルとなり，2サイクル減少している．そのため，分岐ミスが起きた場合のストールサイクル数を削減できる．ストールサイクル数を削減することで，後続命令をフェッチして実行するまでのステージが減少する．また，分岐命令から後続命令フェッチまでのサイクル数が減少することで，分岐ミスペナルティが削減できる．

さらに，図6,7に表すように，命令の依存関係による待ちサイクルも削減することが出来る．図6は通常状態のパイプラインを表しており，3段目のパイプラインの最初の実行ステージが2段目のパイプラインの最後の実行ステージの処理結果に依存しているとする．図7は図6と同様の依存関係を持つ統合状態のパイプラインを表している．図6から分かるように，通常状態では，2サイクル待つ必要がある．しかし，図7では，実行ステージが1ステージに統合されており，1サイクルで動作する．このため，待ちサイクルが2サイクル削減できる．また，クロック周波数を下げることで処

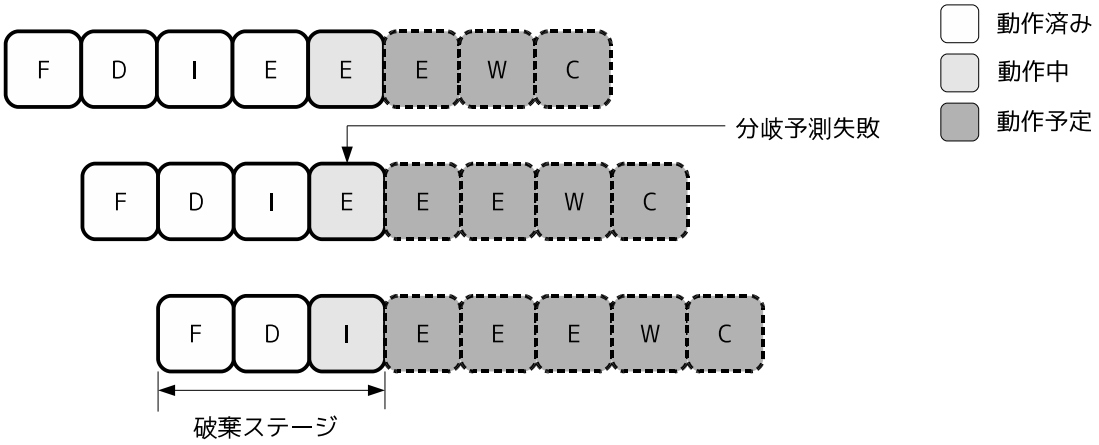

図 4: パイプライン通常

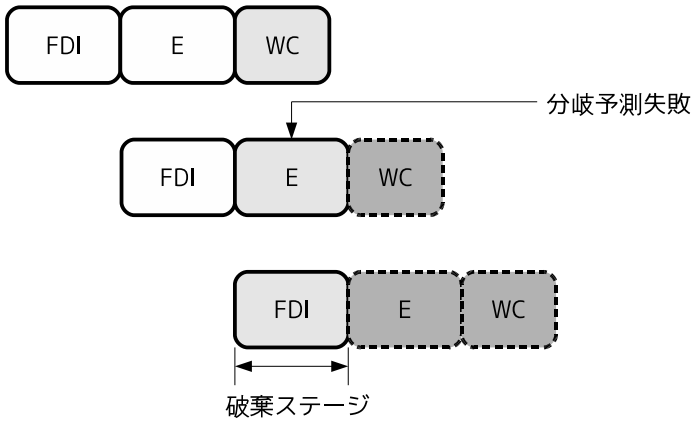

図 5: パイプライン統合

F : fetchステージ
D : dispatchステージ
I : issueステージ
E : executeステージ
W : writebackステージ
C : commitステージ

理性能を落とすことによっても省電力化を行っている．

PSU では，ステージの統合を動的に行っている．そのために，PSU は 2 つのフェーズを持っている．[2] 本論文でのフェーズとは，一定サイクル数分の期間のことである．1 つ目のフェーズでは，統合状態，通常状態を切り替えるため，定期的に負荷をサンプリングしている．この期間を，**サンプリングフェーズ**と呼ぶ．サンプリングフェーズでは，命令を実行しながら，同時に IPC を測定している．さらに，測定された IPC に応じて統合状態，通常状態を切り替え，クロック周波数の調整をしている．IPC が低けれ

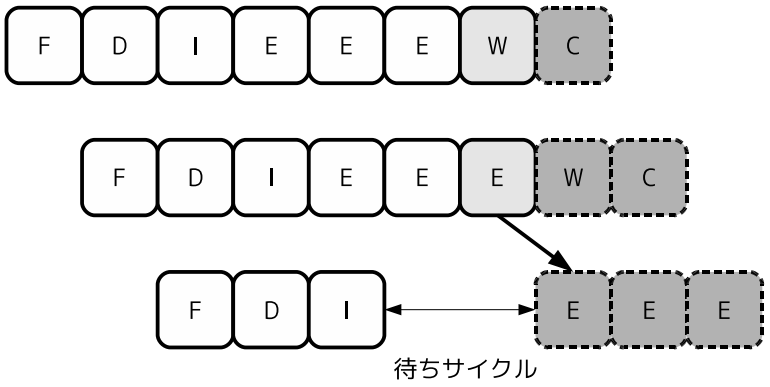

図 6: パイプライン通常

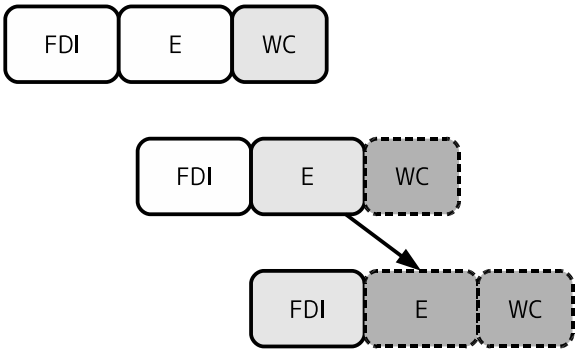

図 7: パイプライン統合

ばプロセッサの処理に余裕があると判断してステージの統合を行い，高ければ処理に負荷がかかっていると判断し通常状態に戻す．2 つ目のフェーズを，**実行フェーズ**と呼び，サンプリングフェーズ以外の期間はすべて実行フェーズにあたる．実行フェーズでは，統合，通常状態の切り替えや周波数の変更を行わず，サンプリングフェーズで変更された状態を引き継いで命令を実行する．図 8 に表すように，PSU では，サンプリングフェーズと実行フェーズを一定期間ごとに交互に切り替えて動作している．通常状態と統合状態を動的に切替えることで，DVS よりさらに性能の低下を緩和し，消費エネルギーを削減している．

2.2 従来手法の問題点

DVS と PSU，2 つの従来手法には，いくつかの問題点が存在する．DVS では，電源電圧は標準の電圧とプロセッサが動作する閾値電圧の間で変化させるものであり，ト

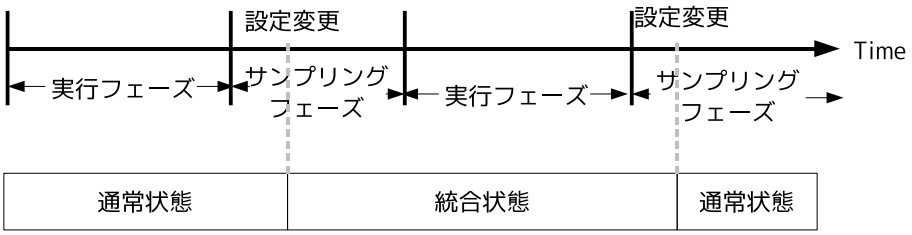

図 8: PSU の制御の概略

ランジスタは閾値電圧以下では動作しない．将来的にプロセッサの製造技術の進歩により標準の電圧は下がっていくと予測されるのに対し，プロセスの微細化が進むと閾値電圧のスケーリングが悪くなると考えられる．閾値電圧のスケーリングが悪くなると考えられる理由としては主に以下の 2 つが挙げられる．

1 つ目は，リーク電流の増大である．リーク電流とは，トランジスタがオフの時に，ドレイン電流を完全に遮断できずに流れてしまう電流のことである．つまり，回路が動作してなくても流れてしまう電流のことで，閾値電力を下げるとリーク電流は大幅に増加する．このリーク電流が発生する原因の一つに，サブスレショナルド電流がある．ゲート電圧が閾値電圧以下であってもトランジスタのドレイン電流はわずかに流れてしまう．ドレイン電流がわずかに流れてしまうことによるリーク電流をサブスレショナルド電流と呼ぶ．サブスレショナルド電流はゲート電圧が 0.6V 下がるごとに 10 倍となり，また温度にも依存している．[3]

2 つ目は，SRAM などにみられる閾値電力のばらつきである．通常，1 つの SRAM には 6 個以上のトランジスタが使用されている．それぞれのトランジスタの閾値電力に大きなばらつきがある場合，SRAM へのデータの読み書きが正常に出来なくなる可能性がある．近年の製造プロセスの微細化によりこのばらつきは増大する傾向にある．

以上で述べた 2 つの理由から，閾値電力が下げづらくなり，その反面標準電力は下がっていく．この，変化電力幅の狭まりから，DVS の削減できる消費電力は将来的に低下することが予測される．

このような DVS の問題から，プロセッサの製造技術に依存しない手法として，PSU が提案されている．しかし，PSU にも削減できる消費エネルギーが限定される可能性が存在する．PSU はサンプリングフェーズと実行フェーズが必ず交互に繰り返されている．統合状態，通常状態を切替えることが出来るのはサンプリングフェーズのみで

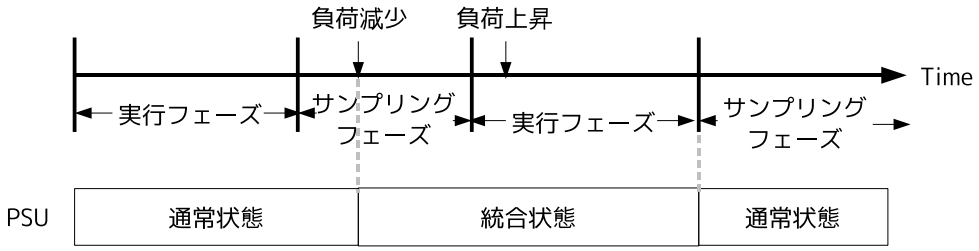

図 9: 制御の概略

あり，実行フェーズはただ命令を実行しているだけである．実行フェーズでは通常状態，統合状態の切替えが出来ない．このことから，負荷のかかるタイミングによっては削減できるエネルギーが限定されることがある．

図 9 に，PSU の削減できる消費エネルギーが限定される場合を表す．サンプリングフェーズが終了した時点で統合状態であり，実行フェーズに入った直後にプロセッサ負荷が上昇した場合，通常状態に戻ることが出来るのは次のサンプリングフェーズである．この実行フェーズの間，高負荷状態のまま，統合状態で実行されることとなる．そのため，本来通常状態に戻し，プロセッサの処理性能を上げるべき場面が統合状態で実行される．それにより，削減出来る消費エネルギーが限定される、つまり消費電力は削減できるが実行時間の増加が起こってしまうと考えられる．

2.3 研究目的

DVS は有用な手段であるが，プロセッサの製造技術に依存していることから，将来的に削減できる消費電力が減少することが予測され，DVS に代わる手法として，PSU が提案されている．過去の研究から，PSU はプロセッサの製造技術に依存せず，また DVS より省エネルギー化が可能であると示されている．[3] しかし，前節で述べた，PSU も削減できる消費エネルギーが限定される可能性がある．このため，本研究ではこの PSU に着目し，PSU が削減できる消費エネルギーの限定を防ぐことで効率をあげ有用性を高めることを目的とする．ここで問題となるのは，PSU は実行フェーズにおいては統合状態，通常状態の切替えが不可能という点である．このため，実行フェーズ直後に負荷上昇が起きた場合，実行時間が増加してしまう恐れがある．本研究では，この点に着目し，実行フェーズにおいて負荷が上昇したとき，統合状態を解除し，通常状態に切替えることを可能にしたい．そこで，実行フェーズでの通常，統合状態の

切り替えを実現するために，負荷情報を導入する．負荷情報とは，過去の命令のサンプリングで記憶した負荷の上昇する命令区間の情報のことである．負荷情報の導入により，過去に負荷の上昇した命令区間が再び呼び出されるとき，実行フェーズで負荷の検知が出来るよう改良を加える．これにより，PSU で実行時間が増加し，削減できる消費エネルギーが限定されてしまうことを防ぐことが出来ると考えられる．

3　提案手法

ここでは，提案手法である負荷情報の導入の動作原理，従来手法との違いを述べる．

3.1　負荷情報の導入

これまで，既存の PSU では実行フェーズにおいて統合状態，通常状態の切替えが行えないため，実行時間が増加してしまうので，削減できる消費エネルギーが限定される可能性があるということを述べた．これを改善するため，実行フェーズで負荷が上昇したときに通常状態に切替えたい．そのために，負荷情報の導入を提案する．ここで述べる負荷情報とは，プロセッサの負荷が上昇した時，負荷上昇を起こす関数の開始命令の**アドレス**のことを指す．これは，プログラムの命令列中，負荷の上昇する命令区間があるとすると，その命令区間の開始命令のアドレスを記憶しておくことで，次回，当該命令が実行されたときに負荷上昇を予測しようというものである．まず，通常の PSU と同様に動作しながら，負荷の上昇を確認するとその負荷情報を記憶しておく．このとき，既に記憶済みの負荷情報であるなら，記憶しない．また，命令実行時に負荷情報と照合し，負荷上昇が予測される命令である場合，統合状態を解除し，通常状態へ切り替える．こうすることで，実行フェーズ中での負荷上昇時，PSU の実行時間の増加を防ぐ．この手法を実現するために，記憶バッファの追加を行った．記憶バッファとは，負荷情報である命令のアドレスを記憶しておくバッファである．実行フェーズではこのバッファ内のアドレスと照合することで負荷上昇が予測される命令を知ることが出来る．

3.2　動作説明

3.2.1　提案手法の動作

記憶バッファを追加した際の，全体の動作について述べる．提案手法の制御の様子を図 10 に示す．まず，提案手法のサンプリングフェーズと実行フェーズの動作について述べる．サンプリングフェーズでは，IPC の測定を行い，測定結果に応じて統合状

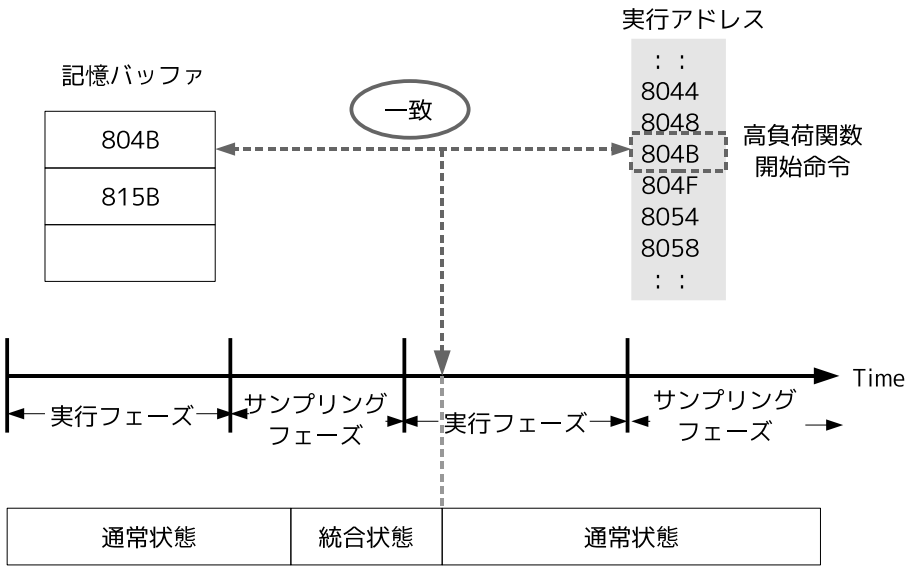

図 10: 制御の概略

態、通常状態の切替えを行う．サンプリングした時に通常状態であり，かつ IPC が低ければ統合状態に切替える．逆に，サンプリングした時に統合状態であり，かつ IPC が高ければ通常状態へと切替える．ここまでは PSU と同じであるが，提案手法では，IPC の測定時にある命令区間で負荷の上昇が認められた場合，その開始命令のアドレスを記憶バッファ内に記憶してある情報 (アドレス) と照合する．記憶バッファ内にその開始命令の情報がなければ，その命令の情報を記憶バッファへ書き込んでおく．一方で，負荷が変動していない場合や減少した場合，もしくは開始命令のアドレスが既に記憶バッファに登録されている場合は記憶バッファには登録せず，PSU と同様の動作をする．図 10 で表す例では，まず，過去のサンプリングフェーズにより，高負荷関数の開始命令のアドレス 804B と 815B が記憶バッファ内に格納されているとする．実行アドレスの列は，命令の実行列であり，中央の時間軸と対応している．つまり，時間軸に合わせて実行アドレスのアドレスにある命令が実行されていく．また，最下段はパイプラインの状態を示している．

実行フェーズでは，PSU と同様に直前のサンプリングフェーズで設定された状態を維持してプログラムの命令列の実行を行う．さらに，実行を行う前に命令のアドレスを記憶バッファ内の情報と照合する．照合の結果，実行する命令のアドレスが記憶バッファに登録されていなければそのままの状態で命令が実行される．しかし，記憶バッファに登録されている命令のアドレスと合致した場合，その時点での状態が統合状態

であるのなら，統合状態を解除し通常状態に切替える．これにより，その命令は通常状態で実行される．例では，実行フェーズ中に高負荷関数の開始命令のアドレス 804B を検知し，このとき記憶バッファには過去のサンプリングによりこのアドレスが格納されている．このため，それまで統合状態であったのを解除し，通常状態へと状態を切替えている．すでに通常状態であった場合，引き続き実行を続ける．提案手法はこのように動作し，PSU の削減できる消費エネルギー量の減少を防いでいる．また，初出の高負荷な命令区間でも状態を切替ることが出来る場合もある．例えば，プロセッサに高い処理性能を要求する関数 A があるとする．この関数 A は実行するプログラム中で何度も呼ばれており，ある時，同等の処理性能を要する関数 B から呼び出されたとする．この関数 B がパイプラインが統合状態のときに呼ばれたとする．関数 B の命令区間は初出であるため，開始命令で通常状態に戻ることはないが，関数 A の開始命令のアドレスが登録されているため，関数 A が呼び出されたときに通常状態に切り替わる．このような場合，初出の命令区間でも状態を切替えることが可能である．

3.2.2 従来手法との違い

ここでは，提案手法と従来手法との違いを述べる．従来手法では，実行フェーズ中に負荷が上昇 (IPC が上昇) した場合，負荷の上昇を検知し，統合状態から通常状態へ切替えることが出来るのはサンプリングフェーズのみであった．このことから，ある実行フェーズで負荷上昇が起きた場合，状態が切り替わるのはその実行フェーズの次のサンプリングフェーズとなる．このため，その実行フェーズは高負荷状態で動作しており，命令実行に多くの時間がかかり，PSU の削減できる消費エネルギーが限定されていた．これに対し，提案手法である実行フェーズ中に負荷が上昇した場合を考える．ここ，負荷上昇を起こした命令区間は，過去のサンプリングフェーズ中でも実行されたとする．この時，記憶バッファに開始命令のアドレスが登録されているので，実行フェーズ中でも，その開始命令の PC を検知し，統合を解除することが出来る．これにより，図 11 に表すように実行フェーズ中の PSU の実行時間の増加を防ぐことが出来る．提案手法は，統合解除のみ実行フェーズで可能となるため，消費電力自体は PSU より上がってしまうが，統合解除したことにより実行時間が削減される．このため，実行エネルギーは PSU よりも削減できると考えられる．また従来の PSU は，削減できる消費エネルギーが負荷上昇を起こす命令区間の偏りに多分に依存している．前述のとおり，実行フェーズにその命令区間が偏ってしまった場合，あまり消費エネルギーを削減できないと考えられる．これに対し，提案手法では，実行フェーズ中での負荷上昇に対応することにより負荷上昇を起こす命令区間が偏っていても切替えが可能で

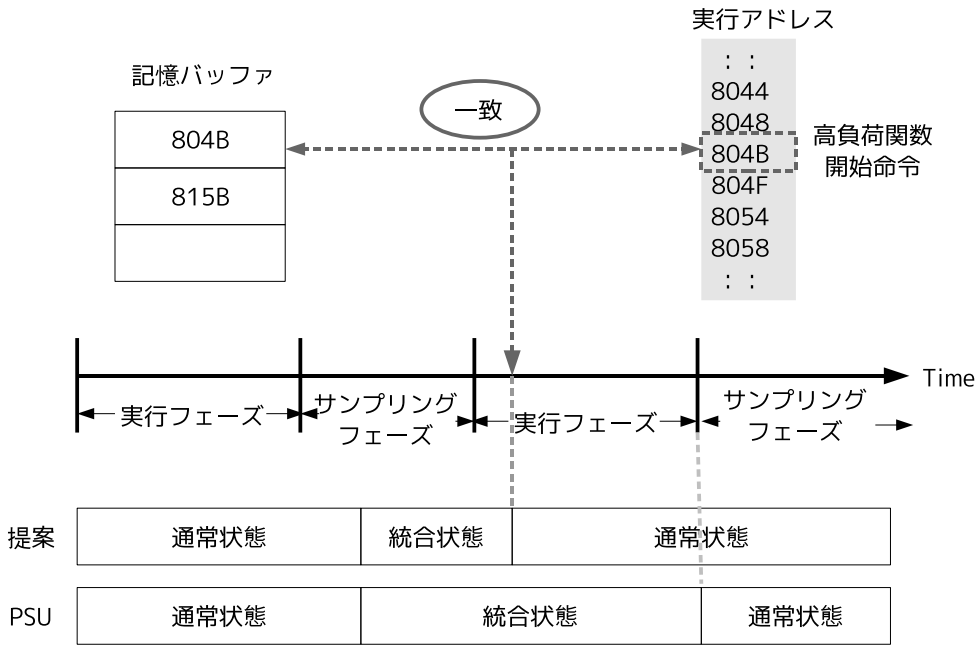

図 11: 制御の概略

ある．しかし，この方法は負荷の低いプログラムでは，削減できる消費エネルギーは PSU と変わらない．また，外的要因などで，一時的に負荷が上昇しただけの命令区間が登録されてしまうと，負荷上昇をしていないのに統合を解除してしまい PSU に比べて，削減できる実行時間に対して消費エネルギー量が増加する恐れがある．また，実行フェーズ中の負荷の減少には対応していないため，実行フェーズ対して負荷上昇を起こす命令区間の割合が少ない時，実行フェーズ中の残りの負荷の低い命令も通常状態で実行されてしまい，実行時間は減少するが，消費エネルギー量が増加する恐れがある．

4 実装

4.1 パイプラインの仕様

4.1.1 パイプライン段数

Simple Scalar の sim_outorder に本提案手法を実装した．PSU は，Simple Scalar のパイプライン部分を用いて行う．[4][5] sim_outorder は，命令を Out of Order 実行するシミュレータであり，Sim-outorder では，パイプライン段数は図 12 のように 6 段となっている．また，点線はステージの区切りを示し，矢印はデータの流れを示してい

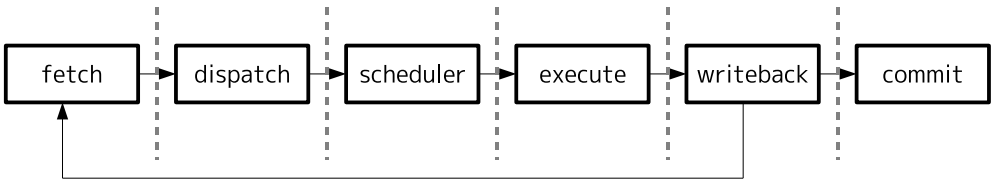

図 12: sim-outorder のパイプライン

る．図 13 に示すように，パイプラインは，Simple Scalar プログラム上では for 文により実装されている．

次に，各パイプラインステージの説明および Simple Scalar の関数との対応を以下に述べる．

Fetch ステージ　命令をキャッシュから読み出すステージである．具体的には，ユーザーが指定した命令幅分の命令を I-Cache(Instruction-Cache) または主記憶から取り出し，IFQ(Instruction Fetch Queue) に格納している．ここで言う命令幅とは，一度に読み出す命令数のことである．また，デフォルトの命令幅は 4 である．Simple Scalar では ruu_fetch() 関数に対応し，命令幅分の命令を I-Cache または Memory から読み出し，IFQ に格納する．

Dispatch ステージ　Fetch ステージで格納された命令を IFQ より取り出し，デコードを行う．その際，メモリアクセスアドレス，レジスタの依存関係を調べ，依存関係が無ければ readyqueue(キュー) へ格納する．デコードした命令を RUU(Register Update Unit) へ配置する．Simple Scalar では，このステージの動作は ruu_dispatch() 関数に含まれている．

Scheduler ステージ　メモリアクセスアドレス，レジスタの依存関係などによって，

```
for(;){
ruu-commit();
ruu-rerease-fu();
ruu-writeback();
ruu-issue();
ruu-dispatch();
ruu-fetch();
}
```

図 13: Simple Scalar パイプライン

readyqueue の順番を入れ替える．readyqueue の先頭から命令を取り出し，その命令が使用するポートやレジスタ等が確保できた場合，その命令を issue する．Simple Scalar では，このステージの動作は ruu_dispatch() 関数に含まれている．

Execute ステージ　命令の実行や，メモリアクセスを行う．また，実行したのち，実行結果およびメモリから読み出したデータを eventqueue(キュー) へ格納する．Simple Scalar では，このステージの動作は ruu_dispatch() および ruu_rerease-fu() 関数に含まれる．

Writeback ステージ　Execute ステージで格納されたデータを eventqueue より取り出し，分岐予測の成功，失敗を判定する．分岐予測が失敗していた場合，後続命令をフェッチし直す．また，命令の実行が終了したことによって，issue 可能となった命令を readyqueue に格納する．Simple Scalar では，このステージの動作は ruu_writeback() 関数に含まれる．分岐予測の成功か失敗かを判断し，失敗していた場合終了し，後続命令を ruu-fetch() からやり直す．また，実行が終了した命令を readyqueue に格納する．

Commit ステージ　RUU の先頭から命令を取り出し，その命令の実行が終わっていた場合，終了処理を行う．終了処理とは，実行結果でレジスタを更新し，ストア命令の場合には，実行結果をメモリに書き込むことである．Simple Scalar では，このステージの動作は ruu_commit() 関数に含まれる．RUU から命令を取り出し，その命令が writeback イベントを終えていた場合その命令の終了処理を行う．また，Store 命令の場合，メモリアクセスポートを確保する．

Simple Scalar の ruu-dispatch() 関数と ruu-rerease-fu() 関数　Simple Scalar の ruu_dispatch() 関数と ruu_rerease_fu() 関数の 2 つの関数は，out of order の Execute，Scheduler，Dispatch の 3 ステージの動作をしている．ruu_dispatch() 関数は，IFQ に送られた命令を In order で取り出し，その命令のデコード及び実行，メモリアクセスやレジスタの更新をすべて同時に行う．その後，その命令を RUU に送る．さらに，メモリアクセスアドレス，レジスタに依存関係が無ければ readyqueue に格納し，実行可能な順序に並び替える．ruu_rerease_fu() 関数は，eventqueue 内の命令を先頭から取り出し，その命令が利用していた Functional Unit を開放する．

4.1.2　パイプライン統合

本研究では，消費エネルギー削減のためにパイプラインステージを統合する．Simple Scalar sim_outorder では，前節の通り 6 個のパイプラインステージを有している．この，パイプラインステージが 6 個の状態を通常状態とする．ここで，ステージを統合

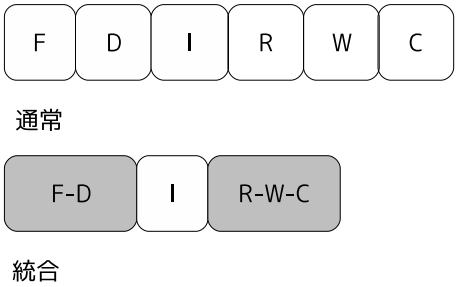

図 14: 統合

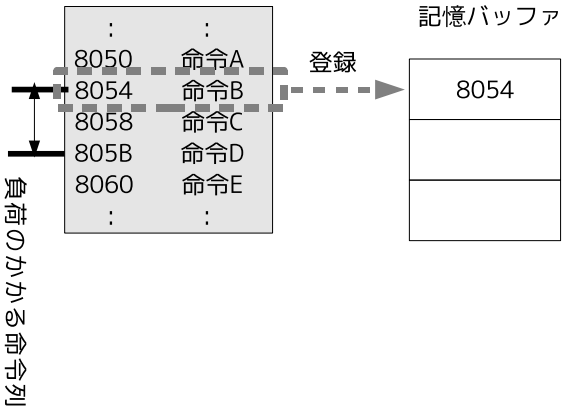

図 15: 記憶バッファ登録

する．統合するステージは，図 14 のように，隣り合うステージ同士とする．これにより，パイプライン段数は 6 段から 3 段となる．本研究では，統合前のパイプラインを通常状態，統合後のパイプラインを統合状態としている．

4.2 記憶バッファ

4.2.1 仕様

ここでは，提案手法を実現するために必要となる記憶バッファについて述べる．記憶バッファは，**FIFO(First in First out)** の構造をとっている．サンプリングフェーズで，命令が実行されるごとに負荷を計測していき，負荷の上昇を計測した場合その命令区間の開始命令のアドレスを格納する．

記憶バッファは，負荷上昇が発生する命令区間の開始命令のアドレスである．例えば，図 15 に表す命令区間があるとする．この命令区間で，記憶バッファに格納される

のは始めの命令 b のアドレスである．負荷のかかる命令区間なので他の命令にも負荷は掛かっているが，負荷が上昇する区間の開始命令のアドレスのみ登録する．この時，すでに記憶バッファには最大格納数までアドレスが格納されていた場合，記憶バッファは始めに格納した命令のアドレスから削除していく．また，実行フェーズで，命令を読み出した時，読み出した命令のアドレスを記憶バッファに格納されているアドレスと照合する．読み出した命令のアドレスがバッファに格納されているアドレスと合致した場合，現在統合状態であるなら，通常状態へ切替える．また，記憶バッファの最大格納命令数は 20 として評価を行う．また，サンプリングフェーズのサイクル数は 10k サイクル，実行フェーズのサイクル数は 500k サイクルと 1M サイクルの 2 通りで評価した．

4.2.2　アクセスタイミング

アドレスの格納　負荷の上昇を検知すると，記憶バッファにその命令のアドレスを格納する。アドレスの格納は，サンプリングフェーズでのみ行われる．負荷の計測は IPC で行われるため，負荷が変動するのは命令の実行後である．このため，負荷上昇時の記憶バッファへのアドレス格納は実行ステージの後に行われる．しかし，4.1.1 の図 13 から分かるように，Simple Scalar でのシミュレーション上の処理は，命令の実行やメモリアクセスの際に行われる．また，各関数の説明を見ると分かるように，レジスタの更新は ruu-dispatch() を実行する際に同時に行われる．その後のステージでレジスタやメモリアドレスの依存関係，実行時間やメモリアクセス時間求められ，Functional Unit，メモリアクセスポートの確保を行っている．このことから，命令の実行は ruu-dispatch() ステージで行わるため，IPC が変動するのはこの直後である．そのため，SimpleScalar 上で負荷情報の検出は ruu-dispatch() 関数の実行直後に行う．

アドレスの照合　提案手法は実行フェーズにおいて，読み出した命令と記憶バッファに格納してある命令のアドレスの照合を行っている．この照合により，以前負荷のかかった命令は実行フェーズにおいても統合状態で実行することが出来る．このため，アドレスの照合は命令が実行される前に行う必要がある．Simple Scalar においても，命令の読み出しは fetch ステージにおいて行われているので，記憶バッファに登録されているアドレスとの照合は fetch ステージの直後に行う．

5　評価

ここでは，実験に用いた環境，使用したベンチマークプログラムについて述べ，実験から得られた結果より考察する．

表 1: プロセッサ構成

発行命令幅		4 命令
RUU		16 エントリ
LSQ		8 エントリ
メモリポート		2 ポート
機能ユニット	整数	4 個
	整数乗除算	1 個
	浮動小数点	4 個
	浮動小数点乗除算	1 個
TLB	命令	16 エントリ/4Way
	データ	32 エントリ/4Way
分岐予測	予測手法	gshare
	BTB	512 エントリ/4Way
	RAS	8 エントリ
キャッシュ	L1 命令	512KB/32B ライン/1Way
	L1 データ	128KB/32B ライン/4Way
	L2	1MB/64B ライン/4way

5.1 評価環境

評価には PSU などの評価に一般的に用いられる SimpleScalar Toolset 中の out-of-order 実行シミュレータを用いて，パイプライン段数を変化させたときの性能を測定した．また，命令セットは PISA を用いた．このシミュレーションで仮定したプロセッサの構成を表 1 に表す．表 1 にロード，ストアユニット数がないが，これは SimpleScalar では整数ユニットがロード，ストアユニットのアドレス生成器を兼ねているためである．また，可変パイプラインを構成する際に，レイテンシを変更している．変更した値を表 2 に表す．ベンチマークプログラムは SPEC95 CINT のうち，表 3 に示すものを用いた．

5.2 結果

前節で示した環境で評価を行った．まず，実行フェーズのサイクル数を 1M サイクルとした場合の結果を図 16 から図 20 に表す． 図 16 は各ベンチマークプログラム実行

表 2: 変更するレイテンシ (サイクル)

	通常	統合
分岐予測ミスレイテンシ	10	5
L1 キャッシュヒットレイテンシ	4	2
L2 キャッシュヒットレイテンシ	4	2

表 3: ベンチマークプログラム

ベンチマーク名	入力	命令数
099.go	2stone9.in	548M
124.m88ksim	ctl.in	124M
129.compress	test.in	35M
130.li	train.lsp	183M
134.perl	scrabbl.in	40M

に要した総サイクル数を表しており左から順に，手を加えていない SimpleScalar(SS)，PSU(PSU)，提案手法を実装した PSU(提案) となっている．図 18 までの図は SS を基準として正規化されており，全て図 16 と同じ 3 パターンについて計測している．図 17 は各ベンチマークプログラム実行に要した時間を表している．図 18 は各ベンチマークプログラム実行にかかった消費エネルギーを表している．図 19 は，総サイクル中の統合状態でのサイクル数を表しており，各手法の総サイクル数を基準として正規化してある．また，図 19 は PSU と提案の 2 パターンで計測した．

次に，実行フェーズのサイクル数を 500k サイクルとした場合の評価を図 20，21 に示す． これまでのグラフと同様，3 パターンについて計測している．

5.3 考察

図 17 から，提案手法は既存手法に比べ，最大で 099.go で約 27 %実行時間が減少しており，平均で約 11 %減少していることがわかる．統合状態でクロック周波数を落とすことにより，プロセッサの処理性能は低下するため，実行時間は増大する．このことから，提案手法は従来手法に比べて処理性能は上がっていると考えられる．また，図 18 から，最大で 099.go で約 6 %消費エネルギーが増加しており，平均で約 3 %増加している．提案手法は，高負荷時には統合状態を解除するので，消費エネルギーは増加

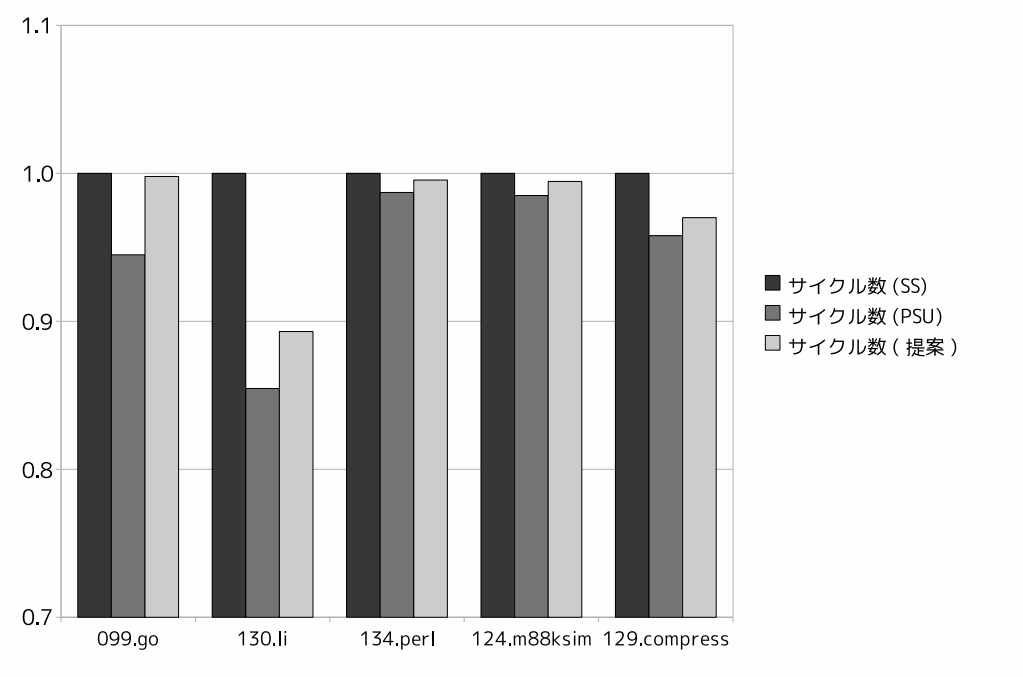

図 16: 実行サイクル数

すると考えられる．しかし，実行時間の削減に対して，消費エネルギーの増加は小さく抑えることが出来た．さらに，図 16 から，最大で 099.go で 37 %，平均で 14 %総サイクル数が増加していることがわかる．これは，統合状態で動作した期間が短いことを意味し，図 19 からも分かるように，総サイクル数中に占める統合状態で動作したサイクル数が減少している．また，総サイクル数が従来手法に比べ増加したことで IPC は低下している．このことから，従来手法では，プロセッサ負荷の高い場合も統合状態で動作していたと考えられ，提案手法によりプロセッサ負荷の改善が出来たと考えられる．

サンプリングフェーズのサイクル数を 500k とした場合，図 20 から，提案手法は既存手法に比べて最大で 099.go で 24 %，平均で約 10 %実行時間が減少している．また，図 21 から，消費エネルギー増加は最大で 099.go の 5 %，平均で約 3 %に抑えることが出来た．このことから，実行フェーズのサイクル数を 1M サイクルとしたときより悪い結果が得られている．これは，実行フェーズのサイクル数が短くなることにより，各サンプリングフェーズ間の間隔が狭まったためであると考えられる．例えば，実行フェーズ中からサンプリングフェーズへの切替え間近実行される高負荷関数があるとする．このとき，提案手法では切替えが起こり，負荷を減少させることが出来る．しかし，実行フェーズが短くなると，この関数は次のサンプリングフェーズに実行され

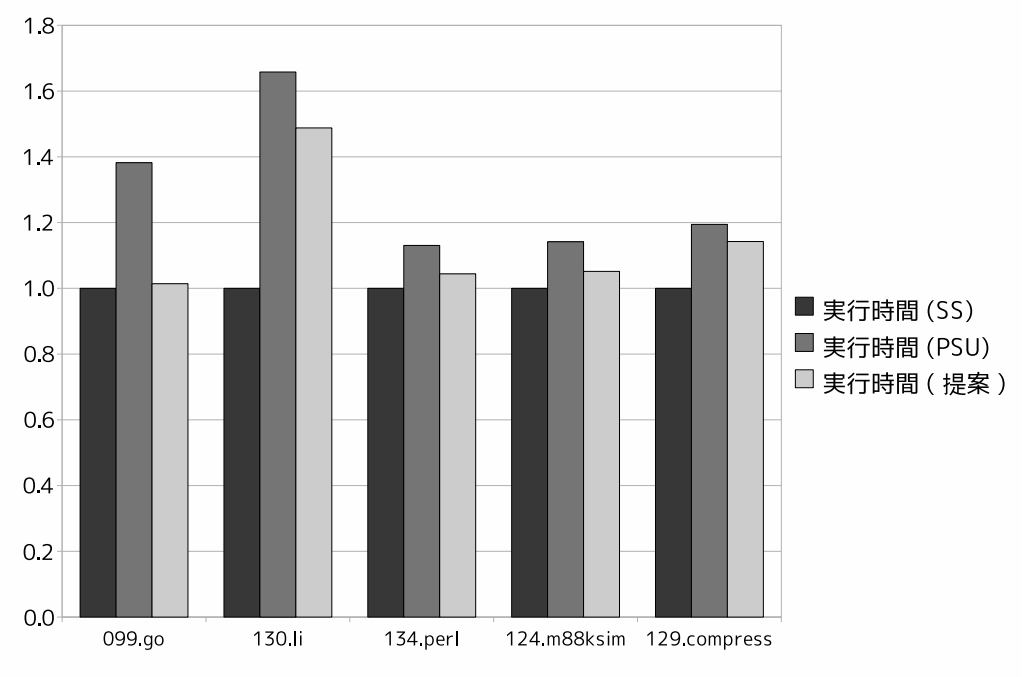

図 17: 実行時間

ることとなり，PSU でも同様の動作が起こる．このため，PSU との動作の違いが小さくなり，実行時間の減少が延びなかったと考えられる．

6 まとめ

近年のモバイルプロセッサは低消費電力と高性能の両立が求められている．その要求に応えるために，DVS という方式が導入されている．しかし DVS は将来的に効率の低下が懸念されており，それに代わる方式として PSU という方式が提案されている．過去の研究により，PSU は現在主流の消費エネルギー削減手法である DVS よりも消費エネルギー削減効果が大きいことが示されている．また，その有効性は半導体の製造技術が進歩するにつれて大きくなることが示されている．しかし，これまでの PSU ではサンプリングフェーズ以外での負荷上昇に対応していないため，実行フェーズ中での負荷上昇時に効率低下の可能性がある．実行フェーズ中での負荷上昇の可能性は大いにありえるため，このことへの対応は重要であると考えられる．そこで，本研究は負荷情報の導入を実現し，提案手法の有効性を調査した．本論文では，PISA 命令セットで out-of-order シミュレーション環境の拡張を行い，SPEC95 CINT ベンチマークプログラムで評価した．その結果，従来手法に比べて，提案手法では，実行時間は 27 %減，消費エネルギーは 5 %増となる場合があった．平均では，実行時間は 15 %減，

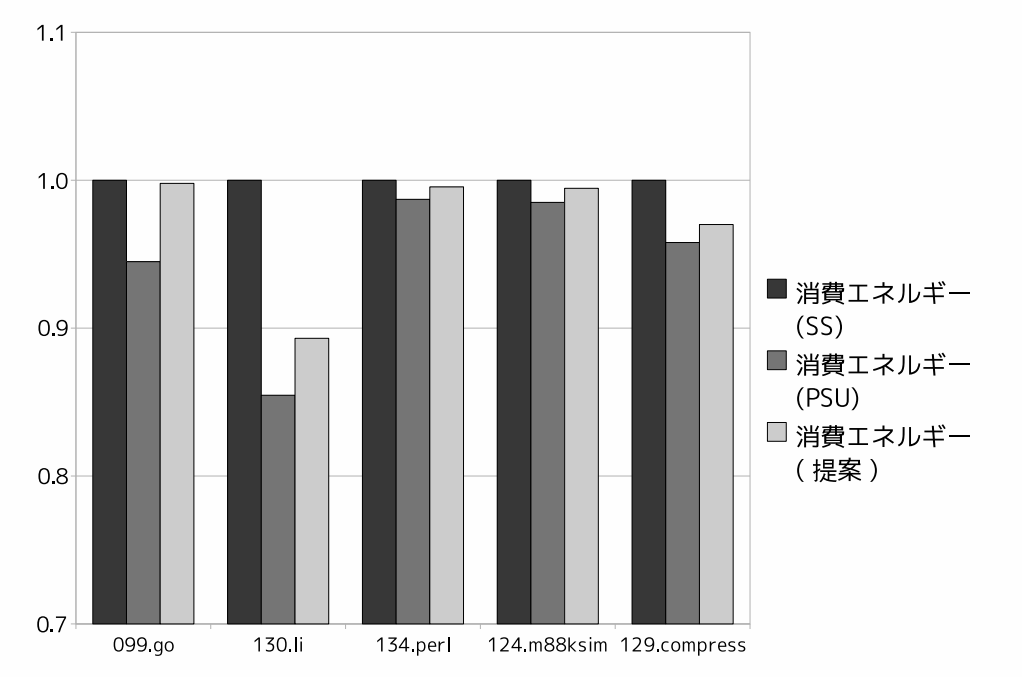

図 18: 消費エネルギー

消費エネルギーは２％増となった．また，総サイクル数は５％増となる場合があり，平均で約３％増となった．このことから，従来手法に比べ，プロセッサの省電力化における性能の低下を抑えることが出来たと考えられる．しかし，本研究では実行フェーズ中での負荷の上昇にしか対応しておらず，負荷の低下にも対応することでさらなる効率の上昇が望める．また，PSU という方式自体が命令セットや環境の変化により IPC の向上率が変化すると，消費電力量も変化するため，異なる命令セットや環境下での有効性の調査が必要である．このことから，本研究で提案した手法も同様の理由からさらなる調査が必要であると考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教授に深く感謝します．また，本研究の際に多くの助言，協力をしていただいた松尾・津邑研究室の方々に深く感謝致します．中でも，同研究室所属の島崎裕介氏，新美明仁氏，神谷優志氏，高木伴彰氏には本研究に関する多大の助言を頂きました．ここに深く感謝の意を表します．

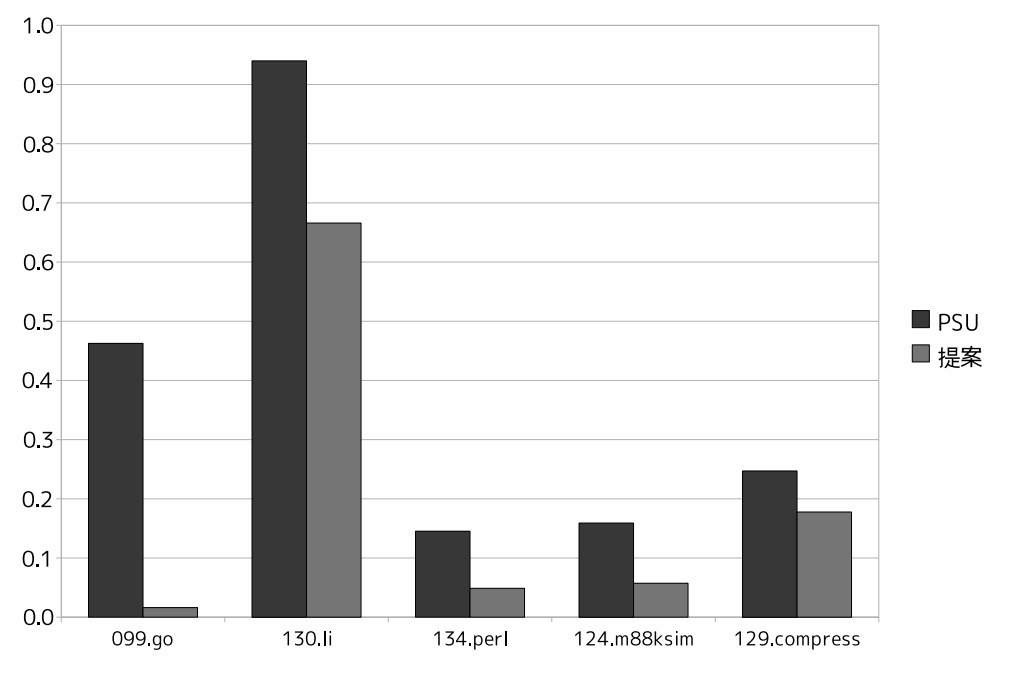

図 19: 統合状態のサイクル数

参考文献

[1] *A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set*.

[2] 嶋田創, 安藤英樹, 島田俊夫: パイプラインステージ統合とダイナミックボルテージスケーリングを併用したハイブリッド消費電力削減機構, 先進的計算基板システムシンポジウム (2004).

[3] 嶋田創, 安藤英樹, 島田俊夫: 低消費電力化のための可変パイプライン, 情報処理学会論文誌計算機アーキテクチャ, No. 145-9 (2001).

[4] 谷口英樹: パイプラインステージ統合を用いた低消費電力プロセッサの設計, 修士論文, 名古屋大学 (2006).

[5] 岩田浩明: ARM 命令セットにおけるパイプラインステージ統合の有効性の調査, 修士論文, 京都大学 (2006).

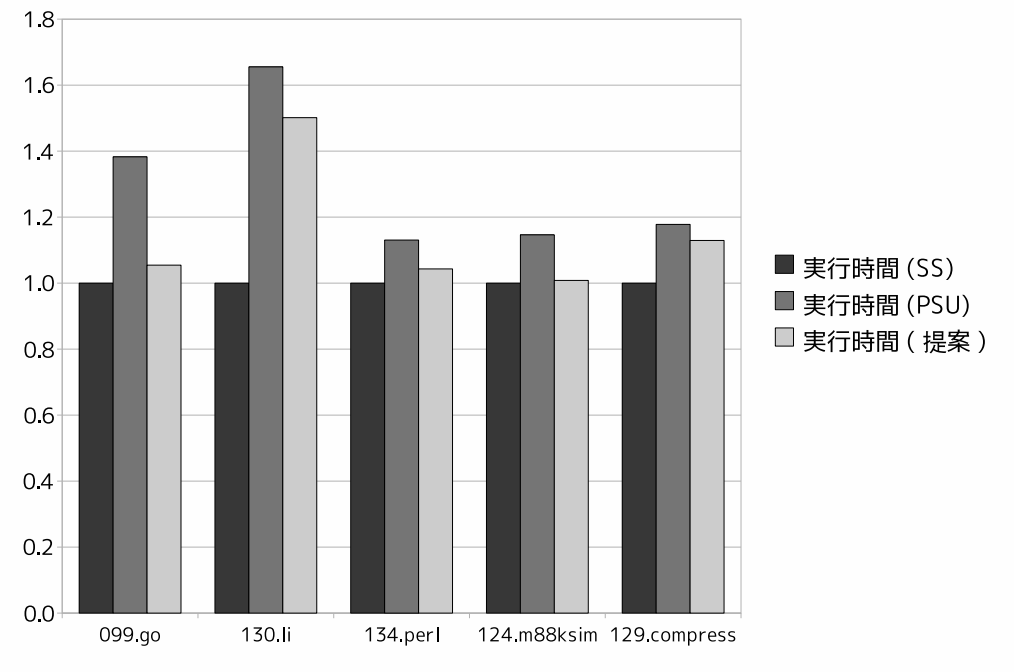

図 20: 実行時間 (500k)

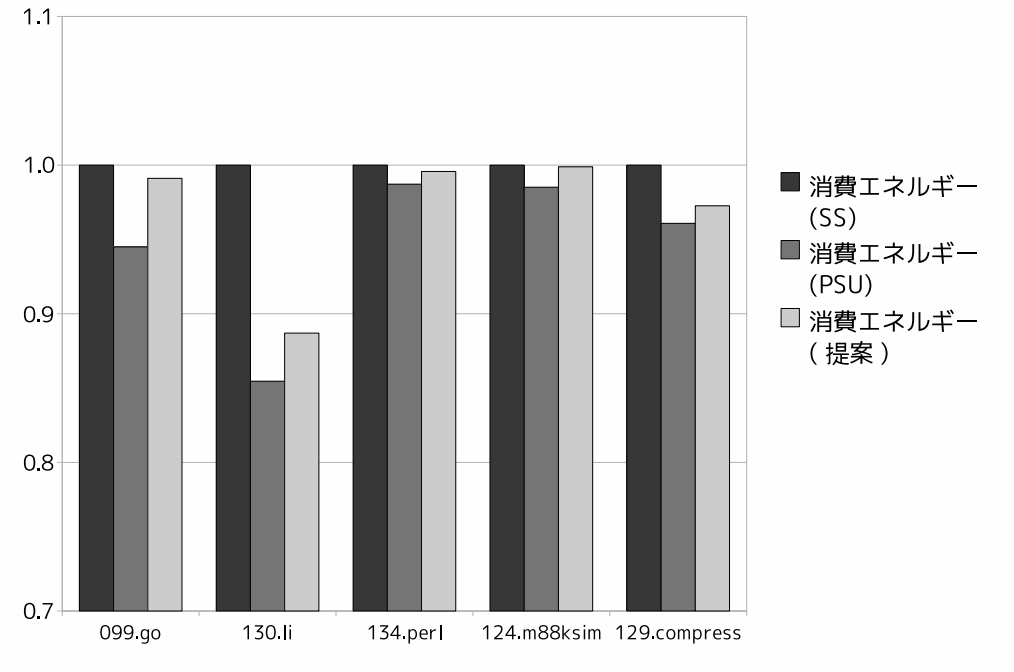

図 21: 消費エネルギー (500k)