情報処理学会論文誌　コンピューティングシステム　Vol. 4　No. 4　36–50 (Oct. 2011)

細粒度な空き時間を利用したコンパイラによるリーク電力削減手法

薦田　登志矢[†1]　佐々木　広[†1]
近藤　正章[†2]　中村　宏[†1]

製造プロセスの微細化にともない，マイクロプロセッサの消費電力の中でもリーク電力の増大が大きな問題となっている．このため，リーク電力削減を目的としたアーキテクチャ技術，コンパイラ技術の研究がなされてきた．本論文ではマイクロプロセッサのリーク電力削減を目標とした，アプリケーション実行時における演算器のパワーゲーティング手法を検討する．演算器の実行時パワーゲーティングでは，スリープにともなうエネルギーオーバヘッドを抑えつつ，できるだけ長い期間演算器をスリープさせるためのスリープ制御手法が重要である．従来のスリープ制御手法は，比較的粗い粒度の空き時間のみを狙って演算器をスリープさせるものであった．このため，細粒度な空き時間が演算器に頻発するアプリケーションでは，リーク電力削減効果が限定的なものとなってしまう．本論文では，細粒度な空き時間において消費されるリーク電力を削減するため，コンパイラを用いたスリープ制御手法を提案する．提案手法ではデータの流れ解析に基づく詳細なプログラム解析により，細粒度な空き時間において効果的なスリープ制御を行うことができる．評価の結果，粗粒度な空き時間に加えて細粒度な空き時間で消費されるリーク電力を削減できる提案手法を用いることで，従来のスリープ制御手法に比べて最大 67%という大きなリーク電力削減率の向上を達成することが分かった．

Compiler Directed Leakage Reduction Technique Considering the Effects of Fine Grained Idle Periods

Toshiya Komoda,[†1] Hiroshi Sasaki,[†1]
Masaaki Kondo[†2] and Hiroshi Nakamura[†1]

As semiconductor technology goes down into sub-micron era, leakage power has been increasing rapidly. Many research have tried to investigate architectural and compiler technique for leakage power reduction of microprocessors. In this paper, we investigate a sleep control technique for microprocessor functional units in order to reduce leakage power consumption. To achieve large amount of leakage power reduction in functional units during CPU execution, it is necessary to minimize energy overhead due to sleep/wakeup mode transitions while maximizing total periods in sleep mode. Conventional sleep control techniques for functional units can reduce leakage power consumed in a coarse-grained idle periods. However, these techniques fail to achieve a large amount of leakage power reduction in applications which use the target functional units frequently. We propose a compiler-directed sleep control technique to reduce leakage power consumed in fine grained idle periods. With a proposed sleep control technique, we can reduce a large amount of leakage power consumed in fine-grained idle periods. Our experiments show that our proposed technique can reduce up to 67% more leakage power compared to conventional techniques.

1. 導　　入

近年，消費電力・エネルギーの削減はマイクロプロセッサ設計における最重要課題の 1 つとなっており，低消費電力化技術の重要性はますます増大している．CMOS 回路の消費電力は，ダイナミック電力とリーク電力（スタティック電力）に大別することができる．特にこれまで無視できるほどの大きさであったリーク電力は，半導体プロセスのさらなる微細化にともないチップ全体の消費電力の中でも大きな割合を占めるようになった．今後のプロセス微細化にともない，さらなる増大が予測されるリーク電力の削減技術が重要な課題となっている．

リーク電力を削減するための回路技術として，パワーゲーティング（Power Gating：以下，PG）と呼ばれる回路技術が知られている．PG 技術では，処理を行っていない回路への電源電圧供給を遮断することで当該回路に流れるリーク電流を削減し，消費リーク電力を大きく削減することができる．このような PG 技術は，バッテリによるエネルギー制約のあるモバイル用途のプロセッサをはじめとし，デスクトップ向けのマルチコアプロセッサにも搭載されるものとなっている[11)]．これら従来の PG 適用例では，チップ全体やコア全体といった大きな回路モジュールを単位として PG が適用されておりチップ全体もしくはコア全体が待機状態にあるシステムアイドル時のリーク電力削減が目的である．一方で今後

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

のリーク電力増大に対処するためには，このようなシステムアイドル時におけるリーク電力削減技術に加えて，システム稼働時，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削減技術の必要性が認識されている．このような背景の中で，PG 技術をアプリケーション実行時におけるキャッシュや演算器といったプロセッサコア内部の細粒度な回路モジュールへと適用しようとする実行時 PG の研究が行われている．

PG 技術は性能的，エネルギー的なオーバヘッドが小さな回路技術である．しかし，1 度にスリープできる期間の長さがシステムアイドル時と比べて圧倒的に短いアプリケーション実行時においては，そのオーバヘッドの影響が無視できない．このためスリープにともなうオーバヘッドを最小化しつつ，回路がスリープしている期間を最大化するためのスリープ制御手法に関して多くの研究がなされている．実行時 PG の初期の研究として，キャッシュにおけるスリープ制御手法が研究されている．文献 4)，9) ではキャッシュのリーク電力削減を目的として，キャッシュラインごとに回路をスリープさせる実行時 PG 技術が提案されている．一方，文献 3) によると演算器などの組合せ回路はスイッチング速度への要求が高く，キャッシュに用いられる回路に比べてトランジスタあたりのリーク電力が大きい．このためキャッシュのリーク電力削減技術に加えて，演算器のリーク電力削減技術を考えることが重要である．このためキャッシュにおける実行時 PG の研究に加えて，演算器をターゲットとしたスリープ制御手法が研究されている．演算器に生じる処理の空き時間は，キャッシュに生じる処理の空き時間に比べてさらに短い傾向があり，したがって，スリープにともなうオーバヘッドの問題はより深刻になる．このため，タイムアウトによるハードウェアベースのスリープ制御手法[6] や，静的なプログラム解析に基づくソフトウェアベースのスリープ制御手法[12),13)] といった様々なスリープ制御手法が提案されている．

演算器における従来のスリープ制御手法では，短い空き時間が頻発するフェーズにおけるスリープを避け，より粗い粒度の空き時間を狙って対象回路のスリープを制御する．短い空き時間が頻発するフェーズにおいて演算器をスリープさせた場合，空き時間の長さがモード切替えにともなうエネルギーオーバヘッドを償却できないほど短い可能性があり，スリープすることでむしろ消費電力増大を招く危険性があるからである[10]．本論文では，スリープ対象演算器が頻繁に使用されるフェーズに発生する短い空き時間を細粒度な空き時間と呼び，従来のスリープ制御手法がターゲットとしてきた粗粒度な空き時間と区別する．従来のスリープ制御手法では，細粒度な空き時間が頻発するアプリケーションにおいてリーク電力削減効果が限定的なものとなってしまうという問題がある．このような場合に大きなリーク電力削減を達成するためには，細粒度な空き時間が頻発するフェーズにおいて演算器のスリープを効果的に制御できるスリープ制御手法が必要である．

本論文ではアプリケーション実行時の演算器における PG 適用を対象に，細粒度な空き時間において消費されるリーク電力を削減するスリープ制御手法を提案する．提案するスリープ制御手法は，静的なプログラム解析による詳細な空き時間予測をベースとしたスリープ制御手法である．プログラムの階層構造を考慮したサイクルレベルの詳細なプログラム解析を行い空き時間の長さを正確に予測することで，細粒度な空き時間が頻発するフェーズにおける演算器のスリープを効果的に制御することができる．また，静的なスリープ制御手法と相性の良いシンプルなハードウェアベースのキャッシュミス検知機構を用いたスリープ制御手法を組み合わせるハイブリッドなスリープ制御手法をあわせて提案する．これにより，メモリインテンシブなアプリケーションも含めたより広い範囲のアプリケーションにおいて，細粒度な空き時間におけるリーク電力を削減することができる．

本論文の構成を示す．2 章でパワーゲーティング技術について概要を示し，本論文で想定するプロセッサアーキテクチャを示す．また，実行時演算器 PG における細粒度な空き時間がリーク電力削減の大きなチャンスであることを示す．3 章では，細粒度な空き時間を効果的に抽出するスリープ制御手法の全体像を示し，提案手法の中で用いる空き時間予測のためのプログラム解析手法を述べる．4，5 章ではシミュレータによる評価を用いて提案手法の有効性を示す．6 章で関連研究について述べ，7 章で結論を述べる．

2. 演算器における実行時 PG

2.1 PG にともなうエネルギーオーバヘッド

PG 回路の模式図を**図 1** に示す．PG 回路ではスリープ対象の回路ブロックとグランド線

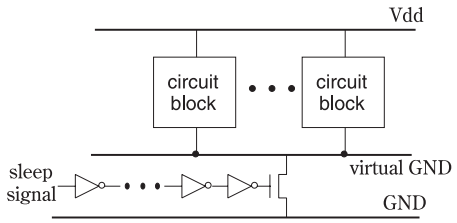


図 1　PG 回路
Fig. 1　The PG circuit.

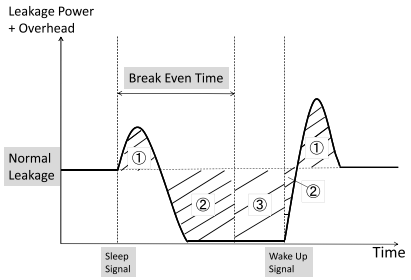


図 2　パワーゲーティングにおける Break Even Time
Fig. 2　Break even time of power gating.

（もしくは電源線）との間にパワースイッチと呼ばれる高閾値のトランジスタを挿入し，このパワースイッチへスリープシグナルを送ることで回路ブロックのスリープ/ウェイクアップモードを実行時に切り替える．PG 回路がパワースイッチのスイッチングによってスリープし，再びアクティブになるまでの回路の消費電力遷移を**図 2** に示す．パワースイッチが *sleep signal* を受けたのちに，*wakeup signal* を受け取るまでの期間がリーク電力を削減できるスリープ期間となる．実行時 PG におけるスリープ制御手法を考えるうえで重要なのが，対象回路のスリープ・ウェイクアップにともなって消費される余分なエネルギーである．図 2 では斜線 1 で囲まれた部分に相当する消費エネルギーがこれにあたる．スリープにともなう余分なエネルギー消費はパワースイッチ自体のスイッチングや，スリープ対象回路の寄生容量における電荷の充放電に起因する．このようにモード切替えにともなうエネルギーオーバヘッドが存在するため，PG ではスリープすることで節約できたエネルギー（図中斜線 2）と，エネルギーオーバヘッド（図中斜線 1）が釣り合うスリープ時間である損益分岐時間（*break even time*：以下 BET）が存在する．したがって，実質的に削減できる消費エネルギーは図中斜線 3 部分となる．ここでもし BET が経過するよりも早く *wakeup signal* が送られた場合，スリープすることによりむしろ全体の消費エネルギーが増加する．したがって BET は，実行時 PG におけるスリープ制御を考えるうえで特に重要なパラメータである．

BET は，製造プロセスの特性や回路構成，温度などによって変化することが知られている．実行時 PG を適用した演算器における BET は，100 サイクル未満の短い値をとる[14]．一般的にリーク電力がダイナミック電力に比べて大きいほど，BET は短くなる傾向にある．今後のプロセス微細化や，実行時の温度上昇によるリーク電力の増大を考えた場合 BET はさらに短くなっていくと考えられる．

2.2　実行時 PG を実現するプロセッサアーキテクチャ

この節では，アプリケーション実行時において演算器における PG を実現するためのプロセッサアーキテクチャについて概要を述べる．**図 3** に本論文で想定するプロセッサアーキテクチャを示す．図 3 に示すアーキテクチャは文献 14) で報告されているマイクロプロセッサ，Geyser のアーキテクチャを参考にしている．アーキテクチャのベースは組み込み向け用途を意識したシンプルなインオーダ実行のプロセッサであり，スリープコントローラからのシグナルによって各演算器を実行時にかつ個別にスリープさせる機能を持つ．

実行時 PG の対象となる演算器はプロセッサの演算器構成に合わせて様々なものを考えることができる．たとえば，文献 14) では MIPS R3000 におけるコプロセッサ，CLU，整数乗算器，整数除算器，シフタという 5 つの回路モジュールに実行時 PG を適用している．

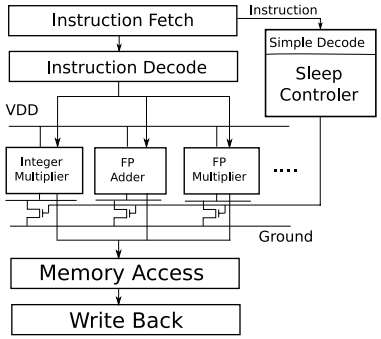


図 3　演算器パワーゲーティングを実現するアーキテクチャ
Fig. 3　Processor architecture for fine grain power gating in functional units.

そのほかにも，浮動小数点演算ユニットや SIMD 演算ユニットなどへ実行時 PG を適用することができる．本論文では，チップ上における実装面積が大きく消費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 種類の演算器をスリープ制御の対象として議論を進めていく．その他の演算器，たとえば整数加算器，整数除算器，浮動小数点除算器などへ実行時 PG を適用することが可能であるが，本論文の評価ではこれらの演算器を対象としていない．これは，整数加算器はほぼ毎サイクル実行されるためどのようなスリープ制御手法を用いてもリーク電力を削減することが難しいこと，および逆に整数除算器，浮動小数点除算器は頻繁に使用されることがなく従来のスリープ制御手法でも十分なリーク電力削減を達成できるためである．しかし，本論文で提案するスリープ制御手法はこれらの演算器やその他の演算器へも適用することが可能である．

なお，スリープモードにともなって生じる性能オーバヘッドに関しては先行研究の中でアーキテクチャ技術により隠ぺいすることが可能であることが示されているため[14]，本論文ではエネルギーのオーバヘッドに焦点を絞って議論を進めていく．

2.3　スリープ制御手法の先行研究

アプリケーション実行時の演算器へ効果的に PG を適用するためには，モード切替えにともなうエネルギーオーバヘッドを最小化しつつ，演算器がスリープモードにいる期間を最大化できるスリープ制御手法が必要である．すなわち，スリープ対象回路に BET 以上長く継続する空き時間が生じた場合のみ回路をスリープし，それ以外の場合にはスリープしないというスリープ制御を実現するためのスリープ制御手法が必要である．このような背景か

ら，アプリケーション実行時の演算器におけるリーク電力削減効果を最大化するためのスリープ制御手法がこれまでに研究されてきた[6),12),13)]．初期の研究において，特別なハードウェア機構を用いて演算器のスリープを制御する手法が提案されている[6)]．この手法は，処理空き時間が一定時間以継続したことをイベントとして当該演算器をスリープさせるタイムアウト方式の制御手法である．一方，静的なコード解析によって演算器に生じる空き時間を予測し，ソフトウェアベースで演算器のスリープを制御しようとする研究がなされている[12),13)]．これらの手法では，当該演算器が長い空き時間に入るフェーズの入口をプログラム解析によりあらかじめ求めておく．この解析情報をもとにプログラムに付加したスリープ制御情報を用いて演算器のスリープを制御する．ループ構造を単位としてプログラム解析を行い，当該演算器を使用しないループの入口で演算器をスリープさせる手法[13)]や，プロファイリング情報を用いてプログラムをその実行頻度によって分割し，このプログラム領域内で演算器を使用しない場合にその領域の入口で演算器をスリープさせる手法[12)]が提案されている．これら従来のスリープ制御手法はスリープ対象の演算器をほとんどもしくはまったく使用しないフェーズを検出することで，モード切替えにともなうエネルギーオーバヘッドを小さく抑えつつ，演算器をスリープさせようとするものである．

2.4　従来のスリープ制御手法の問題点

従来のスリープ制御手法は，スリープ対象の演算器をほとんどもしくはまったく使用しないアプリケーションフェーズにおける粗粒度な空き時間で消費されるリーク電力を削減するものである．しかし従来手法ではスリープ対象の演算器を頻繁に使用し，1 度の期間は短いが多くの回数発生する細粒度な空き時間が全体の実行時間の中で無視できない割合を占めるアプリケーションにおいて，リーク電力削減効果が理想的なスリープ制御が達成された場合と比較して限定的なものとなるという問題がある．この節ではこの問題について説明する．

従来手法が削減しきれないリーク電力削減機会を定量的に考察するため，ある現実的なスリープ制御手法を適用した場合（*real*）に演算器で消費されるリーク電力と理想的にスリープ制御が行われた場合（*opt*）の消費リーク電力の差を考える．理想的なスリープ制御が達成された場合削減できるが，ある現実的なスリープ制御手法 *real* を適用した場合には削減できない消費リーク電力の割合を（*Wasted Leakage Ratio*）*WLR* と呼ぶことにし，以下のように定義する．

$$WLR = (L_{real} - L_{opt})/L_{nosleep} \quad (1)$$

ここで，L_{real}，L_{opt}，$L_{nosleep}$ はそれぞれ，ある現実的なスリープ制御を適用した場合，理想的なスリープ制御を行った場合，まったくスリープを行わない場合に消費されるリーク電力を表す．削減できなかったリーク電力の割合 *WLR* が小さいほど，そのスリープ制御手法は理想に近いスリープ制御を達成していると考えることができ，完全に理想的なスリープ制御を達成する場合の値は 0 である．

表 1　理想的なスリープ制御に対して削減できないリーク電力の割合 *WLR*（BET = 20 サイクル）
Table 1　Wasted leakage ratio (*WLR*, BET = 20 cycles).

ベンチマーク	FPALU		FPMULT		INTMULT	
	loop	timeout	loop	timeout	loop	timeout
ammp	0.01	0.01	0.2	0.01	0	0
apsi	0.27	0.22	0.15	0.13	0.28	0.29
art	0.04	0.03	0.05	0.02	0	0
equake	0.27	0.20	0.16	0.10	0	0
mesa	0.36	0.11	0.43	0.07	0.40	0.05
mgrid	0.08	0.10	0.59	0.10	0.67	0.05
swim	0.22	0.30	0.37	0.15	0	0
wupwise	0.31	0.19	0.43	0.12	0.30	0.04

表 1 に，従来のスリープ制御手法における *WLR* の値を示す．ここでは，タイムアウトによるスリープ制御（*timeout*）[6)]，およびループを単位としたソフトウェアスリープ制御（*loop*）[13)] における *WLR* の値を，浮動小数点加算器，浮動小数点乗算器，および整数乗算器の 3 種類の演算器をスリープ対象として算出している．アプリケーションには SPEC CPU 2000 FP ベンチマークの中から 8 つのアプリケーションを用いている．値の算出には，4 章で詳述するサイクルレベルのシミュレータによる演算器使用履歴のトレースデータを用いている．表 1 から，理想的には削減可能だが従来手法では削減できていないリーク電力が相当量存在することが分かる．

次に理想的なスリープ制御に対して従来手法では削減できないリーク電力とは細粒度な空き時間の間に消費されるリーク電力であることを説明する．そのために，あるアプリケーションが実行された場合に，ある演算器に生じる細粒度な空き時間の割合（*Fine Grained Idle periods Ratio*）*FGIR* を以下のように定義する．

$$FGIR = C_{fg}/C_{total} \quad (2)$$

ここで，C_{fg} は細粒度な空き時間の合計サイクル数（BET の 2 倍以下の長さを持つ空き時間のサイクル数の総和として定義）であり，C_{total} はアプリケーション実行全体に要したサイクル数である．

あるスリープ制御手法が削減できないリーク電力の割合 *WLR* と細粒度な空き時間の割合 *FGIR* は，実行プロセッサが同一であればどちらもアプリケーションと演算器の組ごと

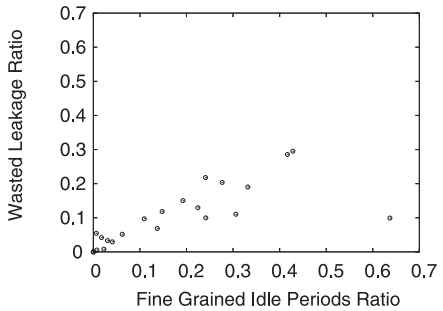


図 4　細粒度な空き時間の割合 *FGIR* と削減できないリーク電力の割合 *WLR* の相関関係（BET = 20 サイクル，*timeout*）
Fig. 4　The correlation between *FGIR* and *WLR* (BET = 20 cycles, *timeout*).

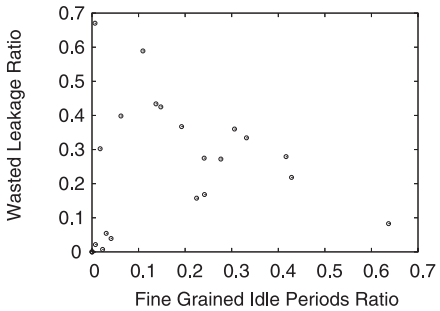


図 5　細粒度な空き時間の割合 *FGIR* と削減できないリーク電力の割合 *WLR* の相関関係（BET = 20 サイクル，*loop*）
Fig. 5　The correlation between *FGIR* and *WLR* (BET = 20 cycles, *loop*).

に定義される指標となる．この 2 つの指標の相関関係を調べるため，**図 4**，および**図 5** に細粒度な空き時間の割合 *FGIR* と削減できないリーク電力の割合 *WLR* の組を二次元平面上にプロットした図を示す．縦軸が削減できないリーク電力の割合 *WLR* の値であり，横軸は細粒度な空き時間の割合 *FGIR* である．1 つの点は 1 つのアプリケーションと演算器の組を表しており，図 4 にタイムアウトによるスリープ制御手法の場合，図 5 にループを単位としたソフトウェアベースのスリープ制御手法の場合を示している．*WLR* および *FGIR* の値は，前述と同様に 4 章で詳述するサイクルレベルのシミュレータによる演算器使用履歴のトレースデータを基に算出している．*WLR* の値は表 1 中のものと同様であり，1 つの図中には 24 個の点がプロットされている[⋆1]．細粒度な空き時間の割合 *FGIR* が増加するとともに従来のスリープ制御手法では削減できないリーク電力の割合 *WLR* も増大していく傾向があることが分かる[⋆2]．すなわち，従来のスリープ制御手法では細粒度な空き時間において消費されるリーク電力を効果的に削減できていないと考えられる．

このように細粒度な空き時間が実行時間の中で大きな割合を占める場合には，従来手法では考慮されてこなかった細粒度な空き時間で消費されるリーク電力が大きな影響を及ぼす．このため，細粒度な空き時間におけるリーク電力をターゲットとした演算器のスリープ制御手法を考える必要がある．細粒度な空き時間において効果的なスリープ制御を行うためには，個別の細粒度な空き時間の長さを正確に予測しつつ演算器のスリープを制御できるスリープ制御手法が必要である．

3. 細粒度空き時間を抽出するスリープ制御システム

3.1 概　　要

この章では，アプリケーション実行時の演算器に生じる細粒度な空き時間で消費されるリーク電力削減を実現するためのスリープ制御手法について説明する．提案手法は，演算器に生じる個々の細粒度な空き時間の長さを予測するためのコンパイラによる詳細なプログラム解析に基づくソフトウェアベースのスリープ制御手法である．このため，サイクルレベルでの空き時間長さの予測を可能にするデータの流れ解析を用いた解析アルゴリズムが提案手法の中心となっており，解析アルゴリズムの詳細は本章の次節以降でくわしく述べる．

命令シーケンスに起因する細粒度な空き時間を静的な解析により予測する一方，メモリアクセス待ちにより生じるパイプラインストールに起因する細粒度な空き時間を静的に予測することは困難である．これは，メモリアクセスがハードウェアのメモリ階層や入力データセットなど動的な要因に強く依存するためである．メモリアクセスに起因する細粒度な空き時間はメモリバウンドなアプリケーションで問題となる．提案手法では，最下位キャッシュミスを検知して演算器をスリープさせるハードウェア機構とソフトウェアベースのスリープ制御手法を併用することで，メモリアクセス待ちに起因する細粒度な空き時間に対処する．キャッシュミス検知によるイベント駆動型のスリープ制御手法は文献 14) の中で提案されており，ハードウェアオーバヘッドが小さくシンプルなスリープ制御手法である．

図 6 に，提案する実行時演算器スリープ制御手法におけるソースコードのコンパイルから実行可能バイナリの実行までの流れを示す．オブジェクトコード生成までのステップは通常のコンパイルと同一である．提案する空き時間解析手法では，サイクルレベルの詳細な空き時間解析を行う（図中 step2）．解析において，より広い範囲のコード情報を利用するために解析はリンク時に行う．コンパイラは，すべての命令に対してスリープ対象演算器に生じる空き時間の予測値を求める．このとき，分岐命令における分岐確率情報を利用してより正確な空き時間予測を行うことができる（図中 step1）[⋆3]．コンパイラによって得られ

⋆1 原点付近で重なっている点があり目視では 20 個〜21 個しかプロットされていないように見えている．
⋆2 *FGIR* = 0.63 に存在する点（*mgrid*，FPALU）は *FGIR* は大きいが *WLR* は小さい，この点では理想的なスリープ制御が達成されたとしても 2 割程度しかリーク電力を削減できないため，*WLR* の値も小さい．

⋆3 分岐命令の分岐確率に関する情報は，マイクロアーキテクチャレベルのシミュレーションを必要とせず容易にかつ高速に取得することが可能である．

た空き時間の解析結果は，スリープ命令を実行可能コードに付加する際に用いられる（図中 step3）．実行可能コードに付加されたスリープ制御情報は，ターゲット CPU によって実行時にデコードされ各演算器のスリープ制御に用いられる（図中 step4）．このようなソフトウェアベースのスリープ制御と並行して，キャッシュミスによるスリープ制御を実現するハードウェア機構が最下位キャッシュミスの発生をトリガとして演算器をスリープさせる（図中 step5）．

提案手法のスリープ制御を実装するためにはハードウェアのサポートが必要になる．ここでは，提案手法のスリープ制御のサポートによって発生するハードウェアコストについて説明する．文献 7) では実行時 PG 機能を有する CPU，Geyser1 の試作チップおよび動作実験結果が報告されている．Geyser1 では，演算器セルにパワースイッチを挿入したことによる面積オーバヘッドが 5.4%–12.6%であったと報告されている．この面積オーバヘッドは，演算器において実行時におけるスリープ機能を実現することによって生じる面積オーバヘッドである．提案手法のスリープ制御手法を実装するためには，これに加えて 1.) ソフトウェアからのスリープ制御情報をデコードするためのハードウェア，2.) キャッシュミスを検知して演算器をスリープさせるためのハードウェア，が必要になる．Geyser1 では，スリープ制御のためのこれら 2 つのハードウェアをサポートしているがその面積オーバヘッド，電力オーバヘッドは無視できる程度のものであった．これは，a.) 命令空間の空きを利用することで，命令長を変更することなくスリープ制御命令を実装することができるため，デコーダロジックの追加は小さい，および b.) キャッシュミスを検知する機構はもともと CPU に備わっている，という 2 つの理由による．また，これら 1.) および 2.) のハードウェアを実現するための回路はクリティカルパス上には存在せず，性能に与える影響はなかった．

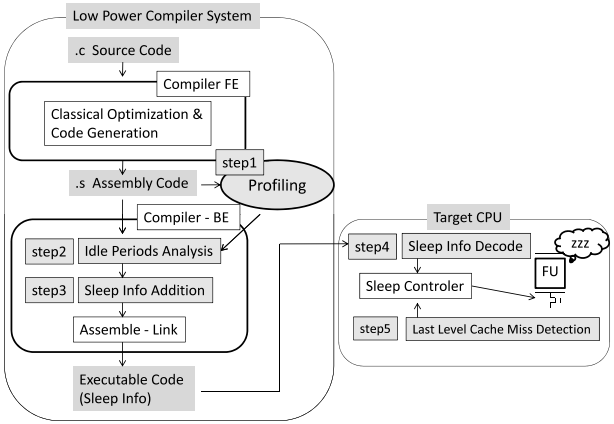


図 6　提案する演算器スリープ制御手法の概要
Fig. 6　An overview of the proposed sleep control method for functional units.

3.2　コンパイラによる空き時間解析

細粒度な空き時間を抽出して効果的にリーク電力を削減するためには，サイクルレベルという高い精度での空き時間予測が必要となる．ここでは，プログラムコードの命令シーケンスからスリープ対象の演算器に生じる空き時間の長さをサイクルレベルで予測するプログラム解析手法について述べる．

図 7 にプログラムの命令シーケンスから演算器に発生する空き時間を予測する様子を単純な命令列を用いて示す．図 7 では，乗算器に生じる空き時間の長さを解析している．左端の Opcode は命令のオペコード，Latency はそれぞれの命令実行に要するサイクル数，Predicted Idle Length は当該命令実行直後から次にスリープ対象演算器が使用されるまでに要するサイクル数の予測値，すなわち空き時間長さの予測値を示している．図 7 では，最下段の *mult* 命令から制御の流れと逆方向にさかのぼりながら，各命令の Latency を足しこんでいくことで空き時間長さの予測値を求めている．すなわち，i 行目の命令における Predicted Idle Length は，$i+1$ 行目における Predicted Idle Length の値と $i+1$ 行目の命令の Latency の値の和として求める．ただし，$i+1$ 行目の命令がスリープ対象演算器を使用する場合（図 7 では，*mult* 命令がこれにあたる），$i+1$ 行目の命令における Predicted Idle Length，および Latency の値によらず i 行目の Predicted Idle Length は 0 とする（図中 initialize）．この空き時間の算出方法は，対象アーキテクチャが 1way インオーダ実行のプロセッサであり，各演算器の実行サイクル数が固定である，という仮定に基づいているが，複数命令同時発行可能なアーキテクチャに対しても命令列の ILP を考慮した命令実行モデルを用いることで，同様に空き時間長さの予測値を求めることができる．この予測値が

Opcode	Latency	Predicted Idle Length（乗算器）
mult	3 cycles	4 (= 3 + 1) cycles
store	1 cycle	3 (= 1 + 2) cycles
load	2 cycles	1 (= 0 + 1) cycle
add	1 cycle	0 (initialize) cycle
mult	3 cycles	xx cycle（後続列に依存）

図 7　命令シーケンスと後続の空き時間予測値
Fig. 7　A sequence of instructions and their predicted values of idle periods.

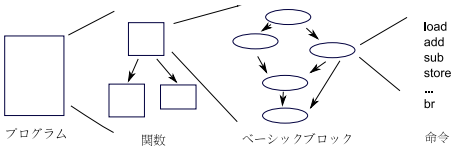


図 8　プログラムの階層構造
Fig. 8　The hierarchical structure of a program.

BET より長い場合に演算器をスリープさせる制御情報を付加することになる．

細粒度な空き時間を抽出する一方で，従来手法でターゲットとされてきた粗粒度な空き時間を取り逃がすことがあってはならない．そこで，提案手法では**図 8** に示すベーシックブロック，関数内の制御構造，関数呼び出し関係というプログラムの階層構造を考慮に入れた解析を行う．これにより，ベーシックブロック内で発生する空き時間から，ループ内で発生する空き時間，さらに関数を跨いで発生する空き時間といった様々な粒度の命令シーケンスに起因する空き時間を正確に予測することができる．次節では上記の空き時間予測解析を，制御フローグラフおよび関数呼び出しグラフ上における命令ノード間の距離を求める問題に帰着させ，空き時間の予測値を求めるアルゴリズムについて説明する．

3.2.1　関数内の空き時間予測

ここでは細粒度な空き時間をサイクルレベルの精度で予測するアルゴリズムについて関数呼び出しのない単一の関数内での解析について述べ，次節でアルゴリズムの関数間解析への拡張について述べる．

提案手法の空き時間予測解析アルゴリズムは，コンパイラにおける静的解析において広く用いられているデータの流れ解析の枠組みを利用する．データの流れ解析の一般論については文献 1) などに詳しく述べられているため，ここでは提案手法において特有な点を中心にアルゴリズムを示す．

データの流れ解析では，まずプログラムをベーシックブロックに分割し制御フローグラフ（*Control Flow Graph*：以下，CFG）を構築する．関数出口を表す単一のノードを s_{exit} とする．提案手法では CFG 上の各命令ノード s に対して，直後の命令アドレスにおける命令ノード $s_{suc}[s]$，分岐先アドレスの命令ノード $s_{bt}[s]$，分岐確率 $q[s]$，命令ノード s の実行に要するサイクル数 $len[s]$，命令 s の実行中にスリープ対象演算器を使用しない確率 $pnu[s]$，という 5 つの値を定義する[⋆1]．分岐確率 $q[s]$，実行サイクル数 $len[s]$ および対象演算器を使用しない確率 $pnu[s]$ は提案手法の解析に特有の情報である．分岐確率 $q[s]$ は 1.) 単純にすべて 0.5 を用いる，2.) ループ構造を考慮してバイアスのかかった値を用いる，3.) プロファイル情報を用いるなどの方法を用いて決定する．なお，4 章における評価実験では 3.) プロファイル情報を用いる方法を用いている．また，実行サイクル数 $len[s]$ および対象演算器を使用しない確率 $pnu[s]$ は以下の式 (3)，(4) を用いて決定する．

$$len[s] := \begin{cases} 0 & \text{(if s uses the Functional Unit)} \\ execLatency[s] & \text{(else)} \end{cases} \tag{3}$$

$$pnu[s] := \begin{cases} 0 & \text{(if s uses the Functional Unit)} \\ probNotUseFU[s] & \text{(else)} \end{cases} \tag{4}$$

命令 s がスリープ対象演算器を使用する場合を，$len[s] = 0$，$pnu[s] = 0$ とすることで表現している．また，$execLatency[s]$，$probNotUseFU[s]$ は命令ノード s に対応する命令の実行に要するサイクル数，およびスリープ対象演算器を使用しない確率を表す．$execLatency[s]$，$probNotUseFU[s]$ の値はプロセッサアーキテクチャのパイプラインや演算器構成情報をもとにあらかじめ与える．通常の命令においては，$execLatency[s]$ は命令 s の実行レイテンシであり，$probNotUseFU[s]$ は 1 である．命令 s が関数呼び出しを行う場合のみ特別な扱いが必要となり，これについては次節で述べる．また，関数出口ノード s_{exit} では，$execLatency[s_{exit}] = 0$ および $probNotUseFU[s_{exit}] = 1$ であると定義する．

CFG 上でデータの流れに沿って演算器に生じる空き時間の情報を解析するため，$idle_{in}[s]$，$idle_{out}[s]$，$pnuExit_{in}[s]$，$pnuExit_{out}[s]$ という 4 つのデータの流れ値を各命令 s に対して定義する．ここで，これらの変数の型は通常のデータの流れ解析で用いられる真理値ではなく実数値とする．各変数の意味はそれぞれ，$idle_{in}[s]$，$idle_{out}[s]$ が，命令ノード s の直前（直後）の地点から次に当該演算器が使用されるまでの平均空き時間，$pnuExit_{in}[s]$，$pnuExit_{out}[s]$ が，命令ノード s の直前（直後）の地点から当該演算器を使用する命令を実行せずに関数の出口（s_{exit}）までたどり着く確率（以下では，演算器未使用確率と呼ぶ）である[⋆2]．また，$Idle_{in}$，$Idle_{out}$，$PnuExit_{in}$，$PnuExit_{out}$ を CFG 上のノードに対する上記 4 変数をそれぞれ並べたベクトル（したがって，ベクトルの長さは CFG のサイズと等しい）であるとする．

⋆1 命令 s が分岐命令でない場合，$s_{bt}[s] = null$（値を持たない），$q(s) = 0$ であるとする．逆に，命令 s が無条件分岐命令のときには $s_{suc}[s] = null$（値を持たない），$q(s) = 1$ であるとする．
⋆2 $s = null$ の場合，各変数の値は 0 であると定める．

演算器の空き時間に期待される性質を考慮し，上記のデータの流れ値が CFG 上で満たすべきデータの流れの等式を CFG 上のすべての s に対して以下のように定義する．

$$idle_{in}[s] = pnu[s] * idle_{out}[s] + len[s] \quad (5)$$

$$pnuExit_{in}[s] = pnu[s] * pnuExit_{out}[s] \quad (6)$$

$$idle_{out}[s] = (1 - q[s]) * idle_{in}[s_{suc}[s]] + q[s] * idle_{in}[s_{bt}[s]] \quad (7)$$

$$pnuExit_{out}[s] = (1 - q[s]) * pnuExit_{in}[s_{suc}[s]] + q[s] * pnuExit_{in}[s_{bt}[s]] \quad (8)$$

式 (5) は，命令ノード s の直前と直後における空き時間予測値の関係を示している．同様に式 (6) は，命令ノード s の直前と直後における演算器未使用確率の関係を示している．式 (7) は，命令ノード s の直後と CFG 上で直接の下流に位置する命令ノード $s_{suc}[s]$，$s_{bt}[s]$ の直前における空き時間予測値の関係を示している．同様に式 (8) は，命令ノード s の直後と命令ノード $s_{suc}[s]$，$s_{bt}[s]$ の直前における演算器未使用確率の関係を示している．

提案手法のデータの流れ解析では流れの等式 (5)〜(8) を満たす実数値ベクトル $Idle_{in}$，$Idle_{out}$，$PnuExit_{in}$，$PnuExit_{out}$ を求める．そのために流れの等式 (5)〜(8) を，左辺に右辺を代入する代入式と読み替え，値が収束するまで代入計算を繰り返す反復計算アルゴリズムを用いる．提案手法における解析では，空き時間に関する情報を関数出口からさかのぼって伝搬させていくため，制御の流れに対して逆向きの反復計算を行う．反復計算のための初期値は以下のように与える．

$$idle^0_{in}[s] := pnu[s] * len[s] \quad (9)$$

$$pnuExit^0_{in}[s] := pnu[s] \quad (10)$$

$$idle^0_{out}[s] := 0 \quad (11)$$

$$pnuExit^0_{out}[s] := 0 \quad (12)$$

各変数の右肩の 0 は反復計算の初期値であることを示している．

等式 (5)〜等式 (8) を満たす予測空き時間 $Idle_{in}$，$Idle_{out}$ および演算器未使用確率 $PnuExit_{in}$，$PnuExit_{out}$ は複数存在しうる．関数出口ノード s_{exit} における $idle_{out}[s_{exit}]$，$pnuExit_{out}[s_{exit}]$ に任意性が存在するためである．このため，式 (9)〜(12) で定まる反復計算の初期値に加えて関数出口ノード s_{exit} における境界値を以下のように与える．

$$idle_{out}[s_{exit}] := 0 \quad (13)$$

$$pnuExit_{out}[s_{exit}] := 1 \quad (14)$$

式 (13) は呼び出し元の関数呼び出し命令の直後でスリープ対象演算器が使用される場合を想定した境界値である．解析対象の関数を呼び出す関数における空き時間解析情報を用いて $idle_{out}[s_{exit}]$ を設定することで空き時間解析の精度を高めることができるが，これについては次節において説明する．一方，式 (14) は関数出口では（すでに関数出口に到達しているので）関数出口まで到達するときにスリープ対象演算器は確率 1 で使用されないことを表している．

アルゴリズム

INPUT 各命令ノード s に分岐確率 $q[s]$，実行サイクル数 $len[s]$，当該演算器の未使用確率 $pnu[s]$ が定義された CFG
OUTPUT 全命令直後からの当該演算器に生じる平均空き時間の予測値 ($idle_{out}[s]$)
$idle_{out}[s_{exit}] \leftarrow 0, pnuExit_{out}[s_{exit}] \leftarrow 1$
for each node $s \in$ CFG **do**
　/* 式 (9)〜(12) に従って変数の初期値を設定 */
　Set initial values of $idle_{in}[s]$, $pnuExit_{in}[s]$, $idle_{out}[s]$, $pnuExit_{out}[s]$
end for
while changes to any $idle_{out}$ or $pnuExit_{out}$ occur **do**
　for each node $s \in$ CFG **do**
　　/*式 (5)〜(8) の左辺に右辺を代入して，空き時間の平均値および未使用確率を更新*/
　　update $idle_{in}[s], pnuExit_{in}[s], idle_{out}[s], pnuExit_{out}[s]$
　end for
end while

図 9　空き時間を予測するデータの流れの解析
Fig. 9　Data flow analysis for idle length prediction.

図 9 に，解析アルゴリズムの疑似コードを示す．

なお，無限ループが存在しないことを前提として提案手法における反復アルゴリズムの収束性を線形代数を用いて理論的に保証することができる．

3.2.2　関数間での空き時間予測

実用的なプログラムは多数の関数呼び出しによって構成されている．このため，演算器に生じる空き時間の長さを正確に予測するためには関数間解析を行うことが必要である．前項で説明したデータの流れの解析は単一関数の CFG 上で動作するものであった．ここでは，解析対象関数内に存在する関数呼び出し命令 s_{call} に定義されるパラメータ $len[s_{call}]$，$pnu[s_{call}]$ および関数出口における空き時間長さの境界値 $idle_{out}[s_{exit}]$ の値を工夫することで関数間の情報を用いた空き時間解析を実現する方法について説明する．本提案手法は，文献 1) で説明されている *Region-Based Analysis* 手法を基にしている．

まず，関数間の呼び出し関係を表現した呼び出しグラフ（*Call Graph*：以下，CG）を構築する．CG は関数をノードとし，呼び出し関係にある関数どうしを有向辺でつないだグラフである．このとき，本提案手法に特有のパラメータとして各関数ノード f に対して 3 つ

の実数値変数 $lenF(f)$，$pnuF(f)$ および $idle_{out}F(f)$ を定義する．これらの変数は，関数 f の入口から関数 f の出口またはスリープ対象演算器を使用する命令に到達するまでに要する平均サイクル数，関数 f の入口からスリープ対象演算器を使用せずに関数 f の出口に到達する確率，関数 f を呼び出す命令（複数存在しうる）実行直後における空き時間長さの平均値，をそれぞれ表している．具体的な値は前節で定義したデータの流れ値を用いて，以下の式 (15)〜(17) によって与える．

$$lenF(f) := idle_{in}[s_{ent}] \tag{15}$$

$$pnuF(f) := pnuExit_{in}[s_{ent}] \tag{16}$$

$$idle_{out}F(f) := \frac{\Sigma_{s_{call} \in Callers(f)} idle_{out}[s_{call}]}{NCallers(f)} \tag{17}$$

s_{ent} は，関数 f の入口に存在する命令ノードを表す．ただし，式 (15)，(16) の右辺の値は関数出口の境界値 $idle_{out}[s_{exit}]$ の値に 0 を用いた場合におけるデータの流れ値であるとする．$Callers(f)$ は，関数 f を呼び出す関数呼び出し命令全体を表しており，$NCallers(f)$ はそのような関数呼び出し命令の個数を表している．提案手法の関数間解析では，関数呼び出し命令 s_{call} における $len[s_{call}]$，$pnu[s_{call}]$ および関数出口における境界値 $idle_{out}[s_{exit}]$ の値に，CG 上の関数 f に対して定義される $lenF(f)$，$pnuF(f)$，$idle_{out}F(f)$ を用いることで関数間での空き時間解析結果の受け渡しを実現する．

式 (15)，(16)，(17) によって CG 上の関数に定義した値を前項で説明したデータの流れの解析において利用する手順は以下のようになる．関数呼び出し命令 s_{call} における実行に要するサイクル数 $execLatency[s_{call}]$，およびスリープ対象演算器を使用しない確率 $probNotUseFU[s_{call}]$ の値には，呼び出し先関数 f_{callee} の平均実行サイクル数 $lenF(f_{callee})$ および演算器未使用確率 $pnuF(f_{callee})$ を用いる．また，関数出口ノード s_{exit} における空き時間の予測値 $idle_{out}[s_{exit}]$ には現在解析している関数 f の空き時間長さの平均値 $idle_{out}F(f)$ の値を用いる．この様子は，式 (18)，(19)，(20) のように整理される．

$$len[s_{call}] := lenF(f_{callee}) \tag{18}$$

$$pnu[s_{call}] := pnuF(f_{callee}) \tag{19}$$

$$idle_{out}[s_{exit}] := idle_{out}F(f) \tag{20}$$

提案手法における関数間解析では，関数内の変数間で情報を伝搬する流れの等式 (5)〜(8) と関数間で情報を伝搬する等式 (15)〜(20) を満たすデータの流れ値を求めることですべての関数内のすべての命令における空き時間の予測値を求めることができる．

図 10 は，提案手法の関数間解析によって解析情報が関数間で伝搬する様子を模式的に示したものである．sa_{call} が関数 Fb を，sb_{call} が関数 Fc を呼び出している．関数間解析は 2 ステップに分けて行われる．Step1 では，関数 Fc → 関数 Fb → 関数 Fa の順に各関数内の解析を行う．このとき，出口ノード sa_{exit}，sb_{exit}，sc_{exit} における境界値は $idle_{out}[] = 0$ と設定し，式 (15)，(16)，(18)，(19) を用いて呼び出し先関数の解析結果を用いた解析を行う．Step1 により，CG 上のすべての関数における $lenF()$，$pnuF()$ の値が決定される．続く Step2 では，Step1 の解析とは逆の順序（関数 Fa → 関数 Fb → 関数 Fc）に解析を行う．このとき $idle_{out}[sa_{exit}] = 0$ とすることでプログラム全体の出口における境界値を設定し，式 (17)，(20) を用いることで，呼び出し元関数の解析結果を用いながら各関数の出口における境界値 $idle_{out}[]$ を決定していく．Step1 における関数の解析順序は，一般には深さ優先探索を用いて決定することができる[1]．Step2 における関数の解析順序は，一般に Step1 における解析順の逆順を用いればよい．

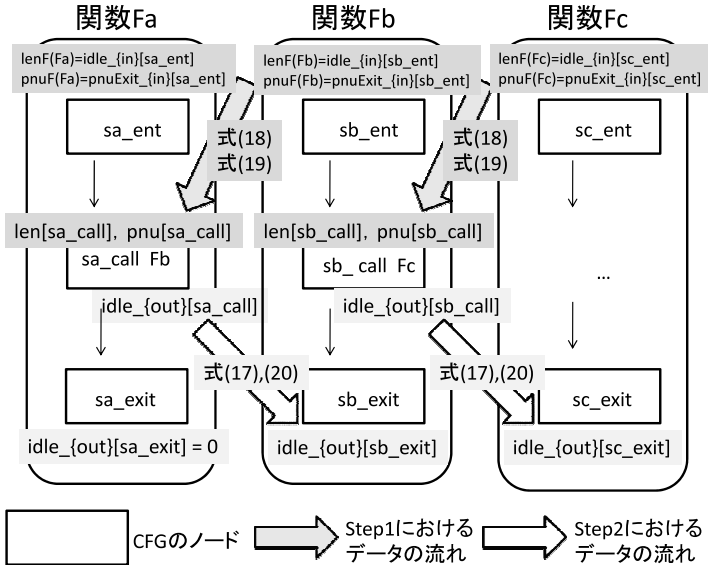


図 10　関数間の情報の受け渡し
Fig. 10　Analysis information passing between procedures.

CG が循環構造（自己ループまたはサイズが 2 以上の強連結成分）を持つ場合には特別な扱いが必要になる．この場合，解析対象関数を解析する前にすべての呼び出し先関数（また

は呼び出し元関数）の解析を終了させることが不可能になるためである．これに対処するため，上述の関数間解析を行う前に強連結成分分解を行い，強連結成分を 1 つのノードに縮約したグラフ上で解析の順序付けを行う（再帰呼び出し部は強連結成分に含まれる）．縮約された強連結成分内での解析では，各関数をランダムな順序で個々に解析する操作を固定回数繰り返す．このとき，式 (15)〜(20) を用いることで強連結成分内で解析情報を伝搬させる．ただし，Step1 中の解析で未解析の関数を呼び出す場合には $len[s_{call}] = 0$，$pnu[s_{call}] = 0$ とし，Step2 中の解析で未解析の関数に呼び出される場合には $idle_{out}[s_{exit}] = 0$ として解析を行う．強連結成分全体を 1 度に解析するわけではないため，強連結成分全体の入口ノードおよび出口ノードを定義する必要はない[⋆1]．

4. 評　　価

4.1 評価環境

本論文で提案する実行時における演算器スリープ制御手法の効果を評価するため，サイクルレベルシミュレータを用いた評価実験を行った．2.2 節で示したプロセッサアーキテクチャのシミュレーションを実現するため，Simple Scalar Tool Set [2] をもとに，演算器の実行時 PG 機能，およびキャッシュミス検知によるイベント駆動のスリープ制御機構を持つサイクルレベルのプロセッサシミュレータを構築した．シミュレーションに用いた詳細なパラメータを，**表 2** に示す．スリープ制御対象とした演算器は浮動小数点加算器（FPALU），浮動小数点乗算機（FPMULT），整数乗算器（INTMULT）の 3 種類である．実行時 PG をシミュレーションするプロセッサシミュレータに加えて，提案手法のコンパイラによるスリープ制御手法を GNU Compiler Collection のバックエンドプログラムとして実装した．コンパイラによる解析結果をプロセッサに伝えるソフトウェアとハードウェアのインタフェースには，スリープ制御ビット[14] による ISA 拡張を用いている．

評価実験では SPEC CPU 2000 [5] FP ベンチマークプログラムの 8 つのベンチマークを用いた．これらのアプリケーションでは，評価対象演算器に細粒度な空き時間が全体の実行時間に対して無視できない割合で生じる．コンパイル時の最適化オプションは-O2，入力データには ref インプットセットを用い，最初の 10 億命令実行後の 2 億命令分の実行を評価対象としている．また，BET の値にはリーク電力がダイナミックな電力に対して大きい場合，すなわち BET が短い値をとる場面を想定して，20 サイクル，40 サイクルという値を用いている．

表 2　評価に用いたプロセッサの構成
Table 2　Evaluation setup of the processor arcchitecture.

Branch Predictor	bimodal (2K Entry)
Num. Functional Unit	
-Int ALU	1
-(load/store)	1
-Int Mult	1
-FP ALU	1
-FP Mult	1
L1 I/D cache	32 KB 2-way 1-cycle latency
L2 unified cache	1 MB 8-way 6-cycles latency
Memory Latency	100 cycles

4.2 比較するスリープ制御手法

本章の評価実験における最も重要な指標は，演算器における正規化した消費リーク電力である．これは演算器に従来のスリープ制御手法または提案スリープ制御手法を適用した場合に消費されるリーク電力とスリープにともなって消費された電力オーバヘッドの和を，スリープをまったくしない場合に消費されるリーク電力で正規化した値であり，そのスリープ制御手法の効果を示すものである．この指標が小さいほど，多くの電力を削減できていることになる．

提案手法（*proposal*）における消費リーク電力と比較するため，従来手法適用時の正規化した消費リーク電力を評価した．従来手法として，ループを単位としてプログラム解析を行うコンパイラによるスリープ制御手法（*loop*）[13]，およびハードウェアメカニズムに基づくタイムアウト方式を用いたスリープ制御（*timeout*）[6] の 2 つのスリープ制御手法を評価した．タイムアウト制御におけるタイムアウトまでの待ち時間は BET と同じ値を用いている[⋆2]．

⋆1 関数内解析における場合と異なり，強連結成分内の解析では値が収束するまで各関数の解析を順番に繰り返す反復計算を行うことができない．たとえばループの中で再帰する再帰関数などで，空き時間の予測値が無限大に発散し値が収束しないためである．このため，解析の回数を固定回数に限定して情報の伝搬を行っている．この処理によって得られる空き時間の予測値は保守的なものとなるが，このようなケースは稀でありスリープ制御における予測精度に大きな影響を与えない．4 章の評価では，強連結成分のサイズと同じ回数だけ強連結成分内関数の解析を繰り返して評価を行っている．

⋆2 タイムアウトスリープ制御において最大のリーク電力削減を達成する最良の待ち時間は，一般に BET と同一ではない．タイムアウト制御のみを適用する場合には，演算器に生じる空き時間長さの確率分布を用いて最良の待ち時間を決定することができるが，このためにはサイクルレベルのシミュレーションを行う必要があり，これは現実的な手法ではない．さらに，本評価ではキャッシュミス検知手法と併用しているため，確率分布を用いた最良の待ち時間決定自体が困難である．このため，最良に近い消費リーク電力削減を達成することが多いことが経験的に分かっている BET の値をタイムアウト制御の待ち時間として用いている．

また，最下位キャッシュミスペナルティが BET よりも大きい今回の評価条件では，ベースとなるスリープ制御手法にかかわらずキャッシュミス検知手法を併用することでリーク電力削減効果はつねに大きくなる．このため，ここでの評価では従来手法の評価においてもキャッシュミス検知によるスリープ制御手法を併用させて評価を行っている．

4.3 評価結果

図 11〜図 16 に各演算器における正規化した消費リーク電力の結果を示す．**図 11**，**図 13**，**図 15** は BET が 20 サイクルの場合，**図 12**，**図 14**，**図 16** は BET が 40 サイクルの場合の結果である．図中では，各アプリケーションごとに 3 つのスリープ制御手法における正規化されたリーク電力に対応する 3 つのバーが表示されている．青色のバーが，提案手法適用時の消費リーク電力である．右端の *average* はアプリケーション群を通じての幾何平均を示している．ただし，整数乗算器についてはまったく使用されずシミュレーション期間中を通じてつねにスリープできる場合があるため，このような場合を除外して平均を求めている．

提案手法はすべてのケースにおいて，従来手法であるタイムアウト制御およびループを単位としたコンパイラによるスリープ制御よりも大きなリーク電力削減を達成している．提案手法によるプログラムの階層構造を考慮したサイクルレベルの空き時間予測により，従来手法よりも効果的なスリープ制御を広い範囲のアプリケーションに対して実現できていることが分かる．**表 3** は従来のスリープ制御手法と比べた場合の提案スリープ制御手法による消費リーク電力削減率の向上をまとめたものである．リーク電力削減率の向上は，従来手法で消費されたリーク電力と提案手法で消費されたリーク電力の差をとることで求めている．表中には BET とスリープ対象演算器ごとに，リーク電力削減率向上の最大値（*max*），およびアプリケーション全体における平均値（*average*）を示している．提案手法は，タイムアウトによるスリープ制御手法と比較した場合には最大で 19%，ループを単位としたコン

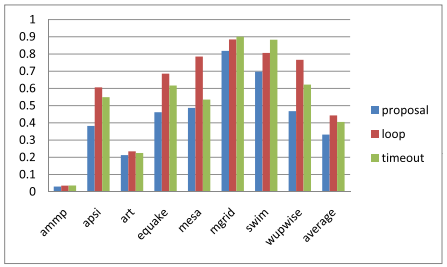


図 11　正規化した消費リーク電力（BET = 20 cycles，FPALU）
Fig. 11　Normalized leakage power (BET = 20 cycles, FPALU).

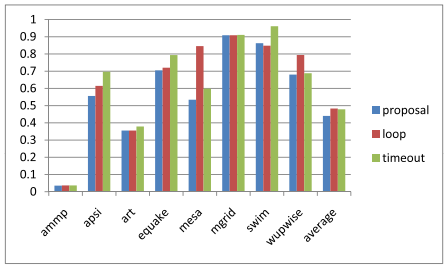


図 12　正規化した消費リーク電力（BET = 40 cycles，FPALU）
Fig. 12　Normalized leakage power (BET = 40 cycles, FPALU).

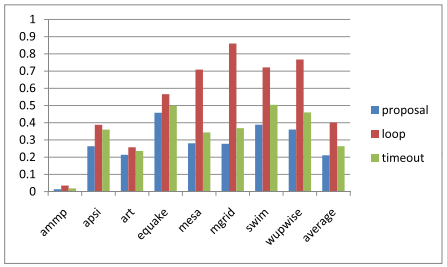


図 13　正規化した消費リーク電力（BET = 20 cycles，FPMULT）
Fig. 13　Normalized leakage power (BET = 20 cycles, FPMULT).

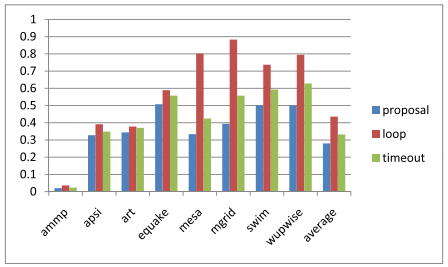


図 14　正規化した消費リーク電力（BET = 40 cycles，FPMULT）
Fig. 14　Normalized leakage power (BET = 40 cycles, FPMULT).

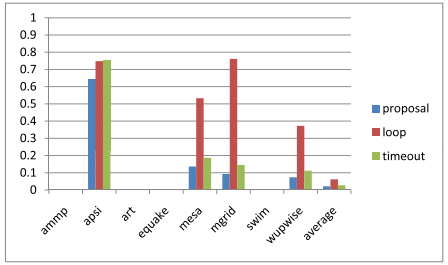


図 15　正規化した消費リーク電力（BET = 20 cycles，INTMULT）
Fig. 15　Normalized leakage power (BET = 20 cycles, INTMULT).

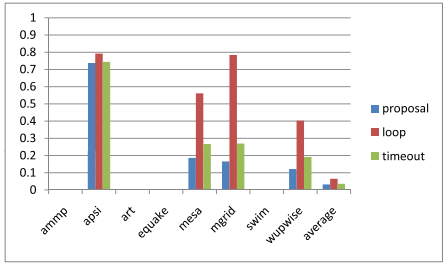


図 16　正規化した消費リーク電力（BET = 40 cycles，INTMULT）
Fig. 16　Normalized leakage power (BET = 40 cycles, INTMULT).

表 3　従来のスリープ制御手法に対するリーク電力削減率の向上
Table 3　Improvement of leakage power reduction compared to conventional methods.

method	BET (cycles)	FPALU		FPMULT		INTMULT	
		max	average	max	average	max	average
timeout	20	0.19	0.07	0.12	0.05	0.11	0.03
	40	0.10	0.04	0.16	0.05	0.10	0
loop	20	0.30	0.11	0.58	0.19	0.67	0.04
	40	0.31	0.04	0.49	0.16	0.62	0.03

パイラによるスリープ制御と比較した場合には最大で 67%という大きなリーク電力削減率向上を達成している．

apsi，*mgrid*，*swim*，*wupwise* における FPALU，FPMULT では提案手法によるリーク電力削減率向上が特に大きい．BET が 20 サイクルの場合，これらのアプリケーションに限定した平均値をとると FPALU で *timeout* に比べて 15%，*loop* に比べて 19%，FPMULT では *timeout* に比べて 10%，*loop* に比べて 34%のリーク電力削減率の向上を達成している．これらのアプリケーションでは浮動小数点系演算器を頻繁に使用するため，FPALU および FPMULT に細粒度な空き時間が頻発する．このため，細粒度な空き時間で消費されるリーク電力の削減を狙った提案手法の効果が結果に強く表れていると考えることができる．また，*mesa* において *loop* スリープ制御手法に比べた場合の提案手法のリーク電力削減率向上が大きくなっている．これは従来の *loop* スリープ制御手法ではループを単位としたスリープ制御を行うため，*mesa* のように長い期間にわたってループが存在しないコード領域が実行されるアプリケーションでは粗粒度な空き時間で消費されるリーク電力削減にも失敗してしまうからである．一方で，整数乗算器（INTMULT）においては従来のスリープ制御手法，提案手法どちらの場合にも平均的に十分なリーク電力削減が達成できている．これは，今回実験に用いたアプリケーション群では整数乗算器において細粒度な空き時間があまり生じないためである．

全体の傾向として，BET が 40 サイクルの場合は BET が 20 サイクルの場合と比較して従来手法に対する提案手法によるリーク電力削減率の向上幅が小さくなっている．これは，細粒度な空き時間を狙ってスリープを行う提案手法では従来手法に比べてスリープ・ウェイクアップのモード切替え回数が多くなり，結果として BET の増大すなわち相対的なエネルギーオーバヘッドの増大の影響を受けやすいためである．

5. 考　　察

5.1 キャッシュミス検知手法の効果

提案手法では，コンパイラによるソフトウェアベースのスリープ制御手法とキャッシュミス検知によるイベント駆動のスリープ制御手法が併用される．ここでは，キャッシュミス検知によるスリープ制御手法を併用することによる効果について議論する．**図 17**，**図 18** に，2 つのスリープ制御手法が提案手法の中でどの程度リーク電力削減に貢献しているかを示す．図中では，2 つのスリープ制御手法を併用する提案手法（*proposal*），提案手法における静的なスリープ制御だけを提供した場合（*compiler-only*），およびキャッシュミス検知によるスリープ制御手法だけ（*cache*）を適用した場合の消費リーク電力を示している．結果からは，2 つのスリープ制御手法が協調的かつ相補的にリーク電力削減に寄与していることが分かる．たとえば，*art* のようにメモリインテンシブなアプリケーションでは，キャッシュミス検知によるスリープ制御手法の効果が高く，*mesa* のような計算インテンシブなアプリケーションでは，コンパイラによるスリープ制御手法の効果が高くなっている．また，2 つのスリープ制御手法を併用することによる悪影響は無視できる程度であり，BET が 40 サイクルにおける *swim*，*mgrid* を除けば，2 つの手法を併用する提案手法がつねに一番大きなリーク電力削減を達成している．コンパイラによるスリープ制御手法とキャッシュミス検知によるスリープ制御手法を併用することにより，様々な特性に起因する演算器空き時間を効果的にとらえたスリープ制御が実現できていることが分かる．

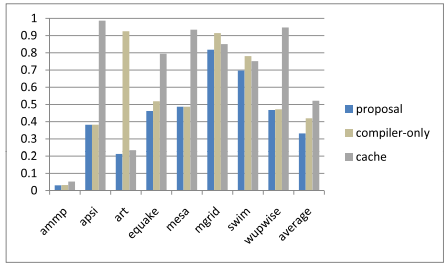


図 17　キャッシュミス検知によるリーク電力削減率の向上（BET = 20 cycles，FPALU）
Fig. 17　Improvement of lekage power reductin via cache miss driven sleep control (BET = 20 cycles, FPALU).

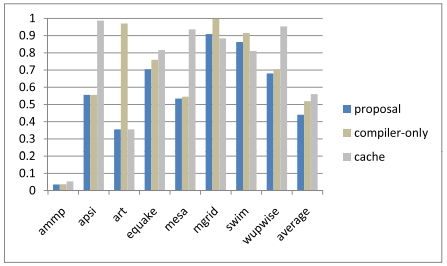


図 18　キャッシュミス検知によるリーク電力削減率の向上（BET = 40 cycles，FPALU）
Fig. 18　Improvement of lekage power reductin via cache miss driven sleep control (BET = 40 cycles, FPALU).

5.2 空き時間粒度と提案手法の効果

2.4 節で示したように従来のスリープ制御手法では，細粒度な空き時間において消費されるリーク電力を効果的に削減できない（図 4，図 5）．ここでは全実行時間に占める細粒度な空き時間の割合 *FGIR* の値が大きい場合にも，提案するスリープ制御手法によって効果的にリーク電力削減が達成できることを示す．**図 19**，**図 20** は，図 4，図 5 と同様のグラフであり，理想的なスリープ制御が実現された場合に比べて提案手法のスリープ制御手法では削減できなかったリーク電力の割合 *WLR* と，細粒度な空き時間の割合 *FGIR* の組をアプリケーションと演算器の組ごとに二次元平面上にプロットしたものである．提案手法では細粒度な空き時間の割合 *FGIR* が大きくなった場合にも *WLR* の値が小さくおさまってお

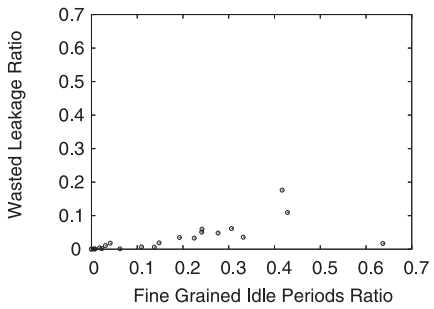


図 19　細粒度な空き時間の割合 *FGIR* と削減できないリーク電力の割合 *WLR* の相関関係（BET = 20 サイクル，*proposal*）
Fig. 19　The correlation of *FGIR* and *WLR* (BET = 20 cycles, *proposal*).

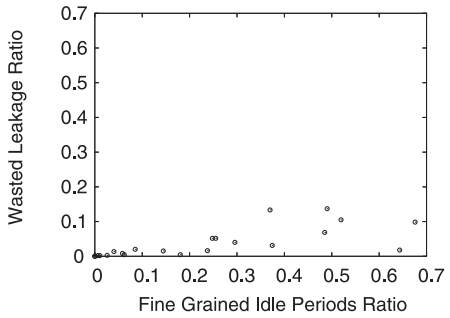


図 20　細粒度な空き時間の割合 *FGIR* と削減できないリーク電力の割合 *WLR* の相関関係（BET = 40 サイクル，*proposal*）
Fig. 20　The correlation of *FGIR* and *WLR* (BET = 40 cycles, *proposal*).

り，理想的なスリープ制御に近いスリープ制御が達成できている．細粒度な空き時間の長さを正確に予測しつつ，細粒度な空き時間が頻発するフェーズにおいてスリープ制御を行う提案手法では，細粒度な空き時間において消費されるリーク電力を効果的に削減できていることが分かる．

6. 関連研究

実行時におけるプロセッサのリーク電力削減手法の初期の研究として，キャッシュを対象とした研究がある[4),9)]．これらの研究では，2 章で説明した文献 6) における演算器スリープ制御手法と同様の，タイムアウトによるスリープ制御手法が提案されている．本論文が対象とした実行時における演算器リーク電力削減の研究には，2 章で説明したハードウェアベースのスリープ制御手法[6)] や，ソフトウェアベースのスリープ制御手法[12),13),15)] がある．また，演算器におけるリーク電力削減を VLIW アーキテクチャにおいて実現しようとする研究もなされている[16)]．これら従来のスリープ制御手法は，スリープ対象演算器をほとんど使用しないアプリケーションフェーズに起因する長い期間継続する粗粒度な空き時間をターゲットとしたスリープ制御手法である．演算器への実行時 PG 適用のこうした研究を受け，近年ではリーク電力のプロセス依存性，温度依存性，および本論文で取り上げた細粒度な空き時間の影響を扱う研究がなされている[8),10)]．Kannan らは，リーク電力が温度やプロセス変動に大きく依存することを考慮にいれて，リーク電力消費の少ない演算ユニットへ優先的に命令発行を行う機構を提案している[8)]．また，Lungu らは細粒度な空き時間の存在によって既存のスリープ制御手法ではリーク電力が増大する恐れがあることを指摘し，リーク電力増大を防止するためにスリープ制御を動的にオフするガードシステムを提案している[10)]．Lungu らは細粒度な空き時間におけるスリープを完全にあきらめることで，リーク電力増大の危険性を回避する手法を提案しているといえる．これに対し本論文の提案手法では，サイクルレベルの空き時間予測手法を用いることで，細粒度な空き時間を逆にリーク電力削減のチャンスに変えることができる．

7. 結　　論

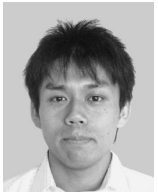
本論文では，演算器における実行時 PG におけるスリープ制御手法を提案した．提案手法では，従来手法が取り逃してきたリーク電力削減のチャンスである細粒度な空き時間におけるリーク電力を削減することが可能である．スリープにともなうエネルギーオーバヘッドを最小限に抑え，効果的にリーク電力を削減するスリープ制御を細粒度な空き時間が頻発するフェーズにおいて実現するためにはサイクルレベルでの詳細な空き時間解析が必要となる．提案手法ではデータの流れ解析を利用してプログラムを解析するソフトウェアベースのスリープ制御手法とキャッシュミス検知によるスリープ制御手法を併用することで，細粒度な空き時間における効果的なスリープ制御を実現している．シミュレータを用いた評価の結果，提案手法を適用することで従来手法に比べて最大で 67%という大きなリーク電力削減率の向上を達成できることが分かった．

参 考 文 献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986).
2) Burger, D. and Austin, T.M.: The SimpleScalar tool set, version 2.0, *SIGARCH Comput. Archit. News*, Vol.25, No.3, pp.13–25 (1997).
3) Butts, J.A. and Sohi, G.S.: A Static Power Model for Architects, *MICRO 33: Proc. 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp.191–201, New York, NY, USA, ACM (2000).
4) Flautner, K., Kim, N.S., Martin, S., Blaauw, D. and Mudge, T.: Drowsy Caches: Simple Techniques for Reducing Leakage Power, *ISCA '02: Proc. 29th annual international symposium on Computer architecture*, pp.148–157, Washington, DC, USA, IEEE Computer Society (2002).
5) Henning, J.L.: SPEC CPU2000: Measuring CPU Performance in the New Millen-

nium, *Computer*, Vol.33, No.7, pp.28–35 (2000).
6) Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H. and Bose, P.: Microarchitectural techniques for power gating of execution units, *ISLPED '04: Proc. 2004 international symposium on Low power electronics and design*, pp.32–37, New York, NY, USA, ACM (2004).
7) Ikebuchi, D., Seki, N., Kojima, Y., Kamata, M., Zhao, L., Amano, H., Shirai, T., Koyama, S., Hashida, T., Umahashi, Y., Masuda, H., Usami, K., Takeda, S., Nakamura, H., Namiki, M. and Kondo, M.: Geyser-1: A MIPS R3000 CPU core with fine grain runtime power gating, *Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian*, pp.281–284 (2009).
8) Kannan, D., Shrivastava, A., Bhardwaj, S. and Vrudhul, S.: Power Reduction of Functional Units Considering Temperature and Process Variations, *VLSI Design, International Conference on*, Vol.0, pp.533–539 (2008).
9) Kaxiras, S., Hu, Z. and Martonosi, M.: Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power, *ISCA '01: Proc. 28th annual international symposium on Computer architecture*, pp.240–251, New York, NY, USA, ACM (2001).
10) Lungu, A., Bose, P., Buyuktosunoglu, A. and Sorin, D.J.: Dynamic power gating with quality guarantees, *Proc. 14th ACM/IEEE international symposium on Low power electronics and design*, *ISLPED '09*, pp.377–382, New York, NY, USA, ACM (2009).
11) Rajesh Kumar and Glenn Hinton: A Family of 45 nm IA Processors, *Proc. IEEE International Solid-State Circuits Conference*, California, USA (2009).
12) Rele, S., Pande, S., Önder, S. and Gupta, R.: Optimizing Static Power Dissipation by Functional Units in Superscalar Processors, *Proc. 11th International Conference on Compiler Construction*, *CC '02*, London, UK, pp.261–275, Springer-Verlag (2002).
13) Roy, S., Katkoori, S. and Ranganathan, N.: A Compiler Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors, *VLSID '07: Proc. 20th International Conference on VLSI Design held jointly with 6th International Conference*, pp.215–220, Washington, DC, USA, IEEE Computer Society (2007).
14) Seki, N., Zhao, L., Kei, J., Ikebuchi, D., Kojima, Y., Hasegawa, Y., Amano, H., Kashima, T., Takeda, S., Shirai, T., Nakata, M., Usami, K., Sunata, T., Kanai, J., Kanai, M., Kondo, M. and Nakamura, H.: A Fine-grain Dynamic Sleep Control Scheme in MIPS R3000, *ICCD '08: Proc. 2008 international conference on computer design*, pp.612–617, Washington, DC, USA, IEEE Computer Society (2008).
15) You, Y.-P., Lee, C. and Lee, J.K.: Compilers for Leakage Power Reduction, *ACM Trans. Des. Autom. Electron. Syst.*, Vol.11, No.1, pp.147–164 (2006).
16) Zhang, W., Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Duarte, D. and Tsai, Y.-F.: Exploiting VLIW schedule slacks for dynamic and leakage energy reduction, *Proc. 34th annual ACM/IEEE international symposium on Microarchitecture*, *MICRO 34*, pp.102–113, Washington, DC, USA, IEEE Computer Society (2001).

(平成 23 年 1 月 28 日受付)
(平成 23 年 6 月 16 日採録)



薦田登志矢（学生会員）
2008 年東京大学工学部計数工学科卒業．2010 年同大学大学院情報理工学系研究科修士課程修了．2010 年より同大学院工学系研究科博士後期課程に在学中．計算機アーキテクチャ，コンパイラの研究に従事．

佐々木　広（正会員）
2003 年東京大学工学部計数工学科卒業．2005 年同大学大学院情報理工学系研究科修士課程修了．2008 年同大学院工学系研究科博士課程修了．博士（工学）．同年東京大学先端科学技術研究センター特任助教．2010 年より東京大学大学院情報理工学系研究科特任助教．計算機アーキテクチャ，オペレーティングシステムの研究に従事．IEEE，ACM，USENIX 各会員．

近藤　正章（正会員）
1998 年筑波大学第三学群情報学類卒業．2000 年同大学大学院工学研究科博士前期課程修了．2003 年東京大学大学院工学系研究科先端学際工学専攻修了．博士（工学）．独立行政法人科学技術振興機構戦略的創造研究推進事業 CREST 研究員，2004 年東京大学先端科学技術研究センター特任助手，2007 年同特任准教授を経て，現在電気通信大学大学院情報システム学研究科准教授．計算機アーキテクチャ，ハイパフォーマンスコンピューティング，ディペンダブルコンピューティングの研究に従事．電子情報通信学会，IEEE，ACM 各会員．

中村　　宏（正会員）

1985 年東京大学工学部電子工学科卒業．1990 年同大学大学院工学系研究科電気工学専攻博士課程修了．工学博士．同年筑波大学電子・情報工学系助手．同講師，同助教授を経て，1996 年東京大学先端科学技術研究センター助教授，2010 年東京大学大学院情報理工学系研究科教授．この間，1996〜1997 年カリフォルニア大学アーバイン校客員助教授．高性能・低消費電力 VLSI システム，省電力コンピューティング，ハイパフォーマンスコンピューティングの研究に従事．情報処理学会より論文賞（平成 5 年度），山下記念研究賞（平成 6 年度），坂井記念特別賞（平成 13 年度）各受賞．IEICE，IEEE，ACM 各会員．