影響の少ないインスツルメント手法と電力最適化のためのプログラム領域分割

木村英明[†] 佐藤三久[†]
堀田義彦[†] 今田貴之[†]

我々は，プログラムを複数領域に分割するとともにDVFS機構を利用して電力最適化を行う手法を提案してきた．この分割した領域で行う処理はできる限り一定であり，かつ，領域を区別するために挿入したコードによってプログラムの特性が変化しないことが望ましい．そこで本稿では，プログラムの実行に与える影響が少ないコードのインスツルメント手法を提案する．提案手法では，一定時間でタイマ割込みを行いプログラムの実行位置と最適化対象プログラムの実行情報を取得し，ソースコードの静的な解析結果を組み合わせることでプログラムを複数領域に分割する．さらに，分割した領域に対して電力最適化アルゴリズムを適用し，消費電力の削減を行う．提案手法を実際のクラスタ環境に適用し，プログラムの分割を行うとともに電力最適化を行った．その結果，単純な領域分割方法と比較してプロファイル取得のためのオーバヘッドを大幅に削減した．また，プロセッサ単体で22.2%，クラスタシステム全体で12.0%のエネルギー削減を確認した．

Low-impact Instrumentation and Defining Program Region for Power Optimization

Hideaki Kimura,[†] Mitsuhisa Sato,[†] Yoshihiko Hotta[†]
and Takayuki Imada[†]

We have presented an energy reduction algorithm by controlling voltage and frequency with DVFS for each region in a program. The behavior of each regions should have almost the same characteristics. And instrumented codes between regions should have small impact on the behavior so that analysis and control by instrumented code do not change the program behavior. In this paper, we propose a method to define program regions with small impact instrumented codes. We get the program trace execution data by periodic timer interrupt, and analyze the program structure in source code to decide where instrumented code to be inserted. We focus on the reducing energy consumption by using the proposed method. We have designed and implemented our defining program region technique and evaluated on a real platform. The result shows that we can reduce the energy consumption by 22.2% in processor and 12.0% in cluster system.

1. はじめに

近年，PCクラスタなどのHPCシステムの消費電力が増加している．システムの消費電力増加により，信頼性の低下や実装密度の低下，電気代の増加など様々な問題が発生する[1),2)]．これらを解決するために，低電力コンポーネントを使用してHPCシステムを開発する方法[3)~5)]やDVFS（Dynamic Voltage and Frequency Scaling）により最高性能を保ちつつ必要に応じて電力を削減する方法[1),6)~8)]が数多く提案されている．DVFSとはプロセッサの周波数と電圧を動的に変更できる機構である．プロセッサを低い周波数・電圧で動作させることにより消費電力を削減することができる．近年では多くのプロセッサがDVFS機構を搭載するようになり，PCクラスタシステムにおいてもこの機構を利用できるようになってきた．

DVFSを用いた電力最適化として，プロセッサの使用状況などから周波数を決定する方法[9)]や事前実行により得られるプロファイル情報を基にして最適な実行方式を決定する方法[8),10)]がある．我々は，ソースコードにコードを挿入してプログラムを複数領域に分割しプロファイル情報を取得する（以降，プロファイル取得のためのコードを挿入することを“インスツルメン

† 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

ト” とする）とともに，DVFS を利用して電力最適化を行ってきた[11]．これは，

(1) 関数単位やループ単位など，プログラム中の主要な箇所にインスツルメントし，プロファイル情報を取得する．このコードで挟まれた区間を領域と呼ぶ，
(2) 様々な周波数でプログラムを実行し，プロファイル情報を取得する，
(3) 周波数変更のためのオーバヘッドを考慮したうえで，電力が最適になるよう各領域の周波数を決定する，

という手法である．

これまで，(1) については人手で適当な領域を決定していた．類似の研究では，領域の分割方法について詳細を述べていないことも多い．

たとえば，すべての関数開始時と終了時にプロファイル情報取得のためのコードを挿入することで，プログラムを関数単位に分割することが可能である．これは，非常に単純に実現できるが，関数内での変化をとらえることができない．たとえば，関数内に計算と通信といった明らかに異なる処理が含まれていたとしても，これを同一の領域としてしまう．このような状況下では，本稿で対象とする電力最適化が難しい．またインスツルメントしたコードが非常に短い時間間隔で大量に実行される場合，このオーバヘッドが非常に大きくなりプログラムの特性が変化する可能性がある．これにより，プロファイルを正しく取得できず，最適化を行えない可能性がある．これは関数単位で分割した場合に限らず，他の静的な方法，たとえばループ単位に領域を設定した場合についても同様である．

従来の静的なインスツルメント手法では以下の問題点がある．

- 静的に分割した場合，領域内の挙動が同一である保証ができない．
- 領域を指定するために挿入したコードを頻繁に実行することで，プログラムの特性が変化する可能性がある．

プログラムを複数領域に分割し，電力最適化を行うためにはプロファイルの取得方法が非常に重要である．プロファイル取得によってプログラムの実行特性が大幅に変化することがなく，かつ領域内の特性を適切に取得可能なインスツルメントが必要である．

そこで本稿では，インスツルメントがプログラムの特性に与える影響を少なくし，かつ領域の特性を正確にとらえたプログラム分割手法を提案する．本提案手法では，一定時間間隔ごとにタイマ割込みを行うことで，プログラム中のどの部分を実行しているかという情報とコード挿入位置を決定するための指針となる情報を取得する．さらにソースコード情報を解析しこれらを組み合わせることでインスツルメントする箇所を決定する．従来は適切なコード挿入位置を決定するために人手による解析を行う必要がありユーザへの負担が非常に大きかったが，提案手法を用いることでコードの挿入を容易に実現できる．本稿では，プログラムを複数領域に分割した後，電力最適化手法を適用し電力削減を試みる．そのため，インスツルメント箇所を決定するための情報として消費電力を用いる．電力最適化手法は我々がこれまでに提案してきた手法[11] を用いる．

本稿の構成は次のようになっている．2 章では関連研究，3 章ではこれまで提案してきた電力最適化手法を述べる．4 章では提案する領域分割方法について述べる．5 章では評価環境，6 章では評価結果を示し，7 章で考察，最後にまとめと今後の課題を述べる．

2. 関 連 研 究

プログラムをいくつかの領域に分割し，それに対して最適化を行う手法は HPC 分野において数多く行われている．近年では，DVFS を用いた電力最適化が特にさかんである．電力最適化の際にプロファイル情報の取得や DVFS コードなどの数多くのオーバヘッドが存在するが，これらを考慮した電力最適化手法やプロファイル取得方法について述べているものは少ない．

Lim らはユーザがプログラム中に定義したコード（以降，ユーザコードとする）と MPI 関数呼び出しコードの 2 種の実行状態から領域を分割する手法を提案している[12]．これは，複数の連続する MPI 関数呼び出しコードをまとめて 1 つの領域と見なす，1 度の処理に要する時間が極端に短い場合には領域としない，などの特徴がある．この手法は，領域分割によるオーバヘッドの大幅な増加を避けることができる．しかしながら，ユーザコードと MPI 関数呼び出しコードのみの区別しかないため，挙動の異なるユーザコードを同一の領域としてしまう．

Wu らは，平均 L2 キャッシュミス回数，メモリ転送トランザクションの回数などから領域を決定している[7]．Freeh らは，領域を決定するために operations per miss を用いている[8]．いずれも，観測値の変動を基にして領域を決定し，動的な制御を行っている．この手法では観測値が非常に短い時間間隔で大幅に変動するような場合，最適化のためのコードが頻繁に実行され，性能が大幅に低下する可能性がある．

プログラムを関数単位に分割し，特に実行時間の長い関数を領域として最適化を行う方法もある[11),13)]．この方法は，最適化制御コードの挿入位置を単純に決定でき，わずかな最適化制御コードを挿入することで消費電力を大幅に削減できる．しかし，1 回の実行が非常に短い時間で完了する領域を大量に実行する場合に性能が大幅に低下する可能性がある．このような問題を避けるために，人手による解析やヒントの付与が必要となる．また，周波数を切り替える際のオーバヘッドを考慮した最適化[11)] もある．しかしながら関数単位で制御を行うため，関数内での特性変化をとらえることができない．

ソースコードの静的な解析結果を利用して領域を決定する手法も提案されている[1)]．この手法はプログラムの構造だけで領域と判断するため，まったく別の挙動を示す箇所を同じ領域と見なす可能性がある．また，各処理の実行時間を予測できない場合，分割した領域が極端に細粒度や粗粒度となってしまう可能性がある．

Isci らは，ランタイムシステムにおいて DVFS を用いた電力最適化を提案している[9)]．プログラムを複数領域に分割し，プログラムの実行状態から次に実行する領域を予測し，適切な周波数に遷移することで電力最適化を実現する．

3. 領域分割をともなう電力最適化手法とその問題点

我々は，プログラムを複数の領域に分割しプロファイル情報を基にして電力最適化を行う手法[11)] を提案してきた．これは，以下に示す手順で最適化を行う．

(1) ある決まった方針によってプログラム中にプロファイル取得のためのコードを挿入し，プログラムを複数の領域に分割する．
(2) アプリケーションを実行するとともに，プロファイル情報を取得する．プロセッサの周波数を変更しながら，各領域のエネルギーや EDP（Energy Delay Product）[14)] を取得する．EDP は $EDP = energy * time$ によって与えられる．
(3) プロファイル情報から，各領域に対して最適な周波数を決定する．このとき，周波数を切り替えるためのオーバヘッドを考慮する．すなわち，周波数を変更することによるエネルギーや EDP の削減が，変更のためのオーバヘッドより小さいならば周波数の変更を行わない．

これまで (1) については，通信部と計算部を分割するといった方法や，プログラムの動作を人手で解析することによりプロファイル取得コードの挿入位置を決定してきた．しかしながら，たとえば前者ではメモリアクセスが頻発する部分と主に計算を行う部分を別領域と見なせない可能性がある．後者では，ユーザはプログラムの動作を熟知していなければならずユーザに与える負担が大きいといった問題がある．

4. 影響の少ないインスツルメント手法

本稿では，プログラム実行に与える影響の少ないインスツルメント手法を提案する．本手法では，ソースコードと実行情報からコード挿入位置を決定し，プログラムを複数の領域に分割する．我々の手法によりオーバヘッドと領域内の挙動を同時に考慮したインスツルメントを行うことが可能になる．

4.1 プログラム領域分割の基本方針

以下に示す 3 つの条件を同時に満たすような領域分割が本稿の目的である．

(1) ソースコードに対してコードを挿入することにより，プログラムを複数領域に分割する．
(2) 適切な最適化を行うために，1 つの領域内の挙動ができる限り一定である．
(3) インスツルメントによってプログラムの挙動や特性が変化しない．

我々は，ソースコード中にコードを挿入し，領域を定義することを想定している．HPC アプリケーションの多くは，プログラム中の特定箇所を何度も繰り返す．したがって，最適化を行う部分をソースコード上から見つけ，その部分を最適化することで大幅な電力削減が期待できる．

分割した領域に対してプロファイルの取得や最適化制御を行う．したがって，領域内の挙動は同一であることが好ましい．プロセッサなどの動作を追尾することでより精密な挙動をとらえることができるが，これらの情報を取得するためには新たなシステムの構築や環境の整備が必要となる．本稿では消費電力の削減を対象としており電力測定装置が利用可能であり，これを利用することで新たなシステムを構築する手間を省くことができる．そのため，本稿では指標として消費電力を用い，消費電力の変動によって特性の変化を判断する．

インスツルメントしたコードが頻繁に実行されると，性能が大幅に低下する可能性がある．コードの挿入によって，プログラム本来の実行とは関係のないコードが実行されることになる．この頻度が極端に高い場合，このオーバヘッドによってプログラムの特性が変化する．したがって，インスツルメントがプログラムの実行に与える影響を考慮しなければならない．

並列プログラムを複数領域に分割する単純な方法と

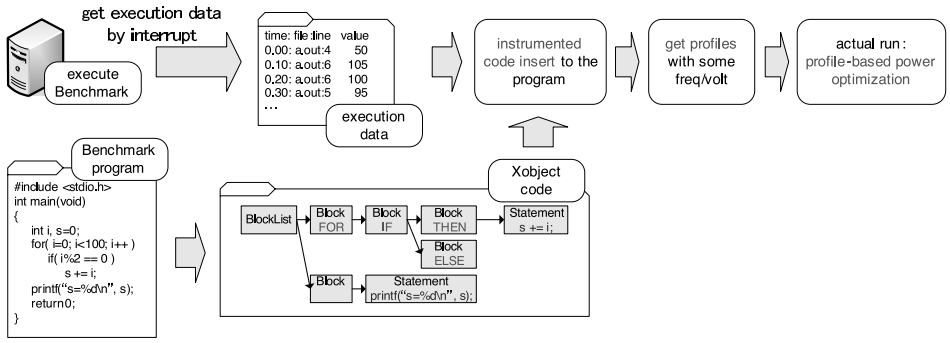


図 1 最適化制御単位の粒度を考慮したプログラム分割の流れ
Fig. 1 Overview of defining program region where grain was considered.

して，計算部と通信部に分けるといった方法が考えられる．しかしながら，この手法では挙動の異なる計算部分を異なる領域と見なすことができない．たとえば，演算部とメモリアクセス部を異なる領域と見なすことができない．一方，我々の提案する手法では計算部と通信部といった単純な分割だけでなく，それぞれについてさらに詳しい分割を行えると考えられる．

4.2 インスツルメントがプログラムに与える影響を考慮したプログラムの領域分割

我々は，コード挿入による挙動の変化を考慮したインスツルメント手法を提案する．本手法では，プログラムのソースコードと実際の実行情報からプロファイル取得コードの挿入位置を決定し，プログラムを複数の領域に分割する．実行情報を用いることで実際の動作を反映させた領域分割を行うことが可能となる．なお，実行情報（execution data）はプロファイル取得コードを挿入する位置を決定するために用いるデータ，プロファイル（profile）は提案手法によって分割した領域に対して取得するデータを意味する．

一方で，ソースコードを解析した結果を AST（Abstract Syntax Tree）形式で表現される中間コードに変換する．これにより，ソースコードの情報を反映させることができる．これらを組み合わせることにより，プログラム特性を大幅に変化させず，かつ領域の特性をとらえた分割を行うことができる．**図 1** に処理の流れを示すとともに，次節以降に詳細を述べる．

4.3 プログラムの実行情報解析

プログラムを実行するとともに実行情報の取得を行う．ここでは，2 種のデータを取得する．

1 つは “プログラム中のどの部分を実行しているか” を示す情報である．これは，取得したデータをソースコード上にマッピングするために必要である．プログラムの実行中に割込みによってプログラムカウンタ（PC）値を取得することで実現する．割込みは一定時間間隔で行い，そのつど PC 値を保存する．Linux において，標準のタイマ割込み間隔は kernel 2.4 以前で 100 Hz，それ以降 kernel 2.6.13-rc1 までが 1,000 Hz，現在は 250 Hz となっている．本稿の評価では，250 Hz で割込みを行い実行情報を取得する．

実行時に保存した PC 値と実行ファイルのデバッグ情報から関連付けられているソースファイル名や行番号を取得する．Linux 環境では

addr2line -e a.out < interrupt_data

によってソースファイル名や行番号情報を取得することができる．

もう 1 つの情報は “プログラムを複数の領域に分割するための指針となる情報” である．たとえば，消費電力やメモリアクセス回数，単位時間に実行した命令数などがこれにあたる．これは，先に述べた実行位置情報と同じ時間間隔で取得する．

これら 2 つの情報から，“その瞬間にどの命令を実行し，そのときどのような状態であったか” を知ることができる．それぞれの情報を取得する機器は異なるが，ともに時系列に沿って取得を行うためこれらを対応付けることが可能である．今回は電力最適化を目的としているため，プログラムを複数領域に分割するための指針として消費電力を用いる．

4.4 ソースコードの静的構造解析

プログラムの実行情報を解析するとともに，ソースコードの静的解析を行う．HPC アプリケーションの多くは，プログラムの構造を解析することにより，より効果的な最適化を行うことができる．

ここでは，Omni OpenMP Exc compiler tool kit [15] の Xobject 形式を用いる．Xobject 形式は AST（Abstract Syntax Tree）形式で構成される中間コードである．このツールを利用することで，プログラムのソースコードを Block や Statement などの要素に分割する．Statement は分岐を持たない文を表現し，

Statement を複数個まとめて Block を形成する．Block が基本データ構造になる．複数の Block から Block-List を構築する．これらを組み合わせることにより，ネスト構造も表現可能である．なお，関数の呼び出し側（caller）が異なる場合，呼び出される関数（callee）が同じであっても別 Block として扱うことにした．これにより，プログラムは単純な木構造に帰着できる．なお，同じ関数であっても異なる Block と見なすため，関数に対してインスツルメントするコードは callee 側ではなく，caller の前後に挿入することになる．

4.5 粒度を考慮したプログラム分割

実行情報とプログラムの静的構造解析結果から，

- 領域内は同一の挙動を示し，領域の特徴を適切に抽出できる，
- プログラムの実行に与える影響が少ない，

を同時に満たす領域分割を行う．

まず，プログラムの実行情報解析結果とソースコードの静的構造解析結果を対応付ける．これにより，各コードの実行情報を木構造で表現することができる．本稿では，電力データを木構造で表現する．また，プログラムを静的に解析することにより Block が何回実行されるかを求めることができ，タイマ割込みによって取得した実行情報からプログラム中の特定部分をどの程度実行したかを判断することができる．これらより，Block 内の処理を 1 回実行するために要する時間を推定できる．

次に，木構造に沿って特性の変化を調べる．各 Block 内の特性がすべて同一であると見なせれば 1 つの領域にまとめ，見なせない場合は複数の領域に分割する．また，挿入したコードが頻繁に実行される状況を避けるため各 Block を 1 回実行するために要する時間を考慮する必要がある．1 度実行するために要する時間が極端に短い Block は領域分割の対象にしない．

Algorithm 1 に領域分割アルゴリズムを示す．Block 内の処理を 1 回行うために要する時間の長短，その Block 内における挙動の変化の有無によって 4 通りに分類する．作成した AST に対し，深さ優先で以下の処理を再帰的に実行する．

4.5.1 子 Block の実行時間がしきい値より長く，すべて同じ挙動を示す場合

自 Block 以下の挙動はすべて同一であると見なす．したがって，自 Block は 1 つの領域，または上位 Block で定義される領域の一部であると見なす．この場合，子 Block の実行情報をまとめ自 Block に保存する．自 Block の前後にプロファイル取得コードを挿入する必要はない．また，1 つの子 Block のみが実行時間のしきい値を超えている場合や実行時間のしきい値を超えている子 Block の挙動がすべて同じである場合についても本手順と同様の処理を行う．**図 2** (1) に例を示す．ここで，val は領域分割の指標として用いる実行情報，T_{th} は領域分割するか否かを決定する実行時間のしきい値を示す．図 2 (1) において Block C1 と C2 の val が同じであるため，B1 以下の Block は同一の領域であると見なすことができる．同一領域と見なす Block を実線楕円で示す．Block C1 と C2 の実行情報（たとえば消費電力データ）を B1 に渡す．また，C1 と C2 を実行する際に要する時間から B1 の処理時間を算出する．上位の Block に対しても同様の処理を行ってゆき，1 つの領域を可能な限り拡大してゆく．

Algorithm 1 領域分割：**define_region**(*block_id*)

```
while block_id's child which must be checked do
  call define_region( block_id's child )
end while
if block_id's child.time > T_th then
  if block_id's children run by the same behavior
  then
    One Region or a part of Region
                ← block_id's children
  else
    Region ← block which run by same behavior
  end if
else
  if block_id's children run by the same behavior
  then
    One Region or a part of Region
                ← block_id's all descendant
  else
    One Region ← block_id's all descendant
  end if
end if
return
```

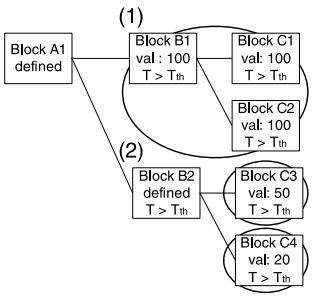


図 2 Block 内の実行時間が長い場合
Fig. 2 The case of it takes long time to execute block.

4.5.2 子 Block の実行時間がしきい値より長く，挙動が変化する場合

子 Block の実行時間が長いため，最適化のための

コードを挿入してもプログラムの実行に与える影響は少ないと考えられる．ゆえに，子 Block の挙動が変化する部分にプロファイル取得コードを挿入し，複数の領域に分割する．自 Block 内で挙動の変化を N 回観測した場合，その Block を $(N+1)$ 個の領域に分割する．親 Block にはすでに自身以下で分割済みであることを示す情報を返す．

実行時間の短い Block と実行時間が長く特性の異なる Block が混在する場合もこれに該当する．この場合，実行時間の短い Block は実行時間の長い Block にまとめ，コードを挿入する．これは，まとめられた Block による影響は小さく無視できると考えられるためである．具体的には，実行時間の短い Block の前後でより特性が近い方にまとめる．図 2 (2) に例を示す．この場合，Block C3 と C4 の処理は異なる．いずれも，実行時間がしきい値よりも大きいため，C3，C4 はそれぞれ別領域と見なす．したがって，C3，C4 はそれぞれ別の楕円で囲ってある．

4.5.3 子 Block の実行時間がしきい値より短く，子孫がすべて同じ挙動を示す場合

このような状況下で領域分割を行うと，実行性能が大幅に低下する可能性が高い．よって，子 Block への探索を打ち切り，自 Block 以下の挙動をいっせいに調べる．

自 Block 以下の挙動がすべて同一であるならば，自 Block 以下を複数の領域に分割する必要はない．実行時間が長い場合と同様に子孫の情報をまとめ，自 Block に保存する．また，この Block は上位 Block の一部となる可能性がある．**図 3** に例を示す．Block B1，B2 の実行時間はともにしきい値より短いため，Block A 以下の特性をいっせいに調査する．Block A 以下の特性に変化が見られなければ，単一の領域であると見なす（図中では実線楕円で示す）．また，他の特性の似た Block とともに 1 つの領域を形成することもできるため，結果を上位 Block に返し上位 Block において領域分割アルゴリズムを再度適用する．図 3 では，破線楕円で示すような大きな領域を定義できる可能性があるため，○付き矢印方向にある上位 Block に対して実行情報を渡す．

4.5.4 子 Block の実行時間がしきい値より短く，子孫の挙動が変化する場合

自 Block 以下のすべての要素をいっせいに調べ，その挙動が変化する場合である．本来ならば，挙動が変化した箇所で領域を区分するべきであるが，このような領域分割を行ってしまうと性能が大幅に低下する可能性がきわめて高い．したがって，領域分割を許す最小の単位である自 Block の前後にプロファイル取得コードを挿入し，1 つの領域と見なす．これは，自 Block 以下で定義されるべき複数の領域を 1 つにまとめたものと等しい．これにより，領域内で複数の挙動を示すものの，プロファイル取得コードの実行回数を大幅に削減しプログラム実行に与える影響を少なくすることができる．この Block は他 Block とともに 1 つの領域を形成することはない．**図 4** では Block B1，B2 の実行時間がしきい値より短く，かつ特性が変化している．ここでは，特性が変化する B1，C3，C4 の 3 領域に分割するのではなく，A1 を 1 つの領域とする．これにより，短時間でプロファイル取得コードが頻発することを避け，実行に与える影響を減らすことができる．図 4 ではこれ以上の大きな領域を形成しない．よって，図 3 で示したような破線楕円は存在せず，上位に実行情報を渡して再評価を行うこともない．ゆえに上位 Block に対して×付き矢印が向いている．

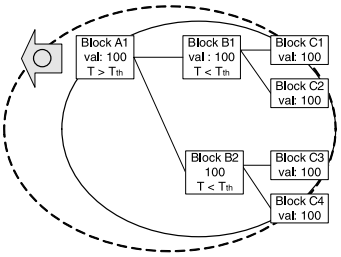


図 3 Block 内の実行時間が短く，挙動が変化しない場合
Fig. 3 The case of it takes short time to execute same behavior block.

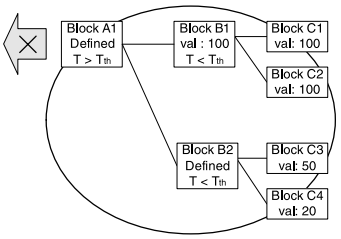


図 4 Block 内の実行時間が短く，挙動が変化する場合
Fig. 4 The case of it takes short time to execute different behavior block.

5. 電力最適化への応用

5.1 評 価 方 法

プログラムを複数領域に分割する最も単純な方法として，関数単位による分割が考えられる．これは，ソースコードの情報のみから単純に領域を決定することができるが，挿入したコードを頻繁に実行することでプログラムの特性が変化する可能性があり，また領域内が同一挙動を示すことも保証できない．本稿では

提案手法による領域分割と関数単位での分割について，それぞれベンチマークの実行時間を測定し，領域分割を行わなかったときと比較する．実行時間が大幅に増加するということは，すなわちインスツルメントしたコードを頻繁に実行しているということである．したがって，実行時間が増加すれば増加するほど，もとのベンチマークとは異なった動作を示しているといえる．

また，本手法によって分割した領域に対してプロファイル情報を用いたDVFSによる電力最適化を行う．これは，3章で示したように，動作可能なすべての周波数でプログラムを実行し，あらかじめ指定した領域に対してそれぞれ最適な周波数を選択するというものである．評価指標として，エネルギーとEDPを用いる．我々がこれまでに提案した手法[11]は，周波数切替えのためのオーバヘッドを考慮した電力最適化手法である．これによりプロファイル取得後の実行時間や電力増加を防ぐことができる．しかしながら，どのような領域が適切であるといった議論はなされておらず，経験的にプロファイル取得コードを挿入することで最適化を行ってきた．プロファイルの取得方法によっては，そのオーバヘッドによってプログラムの特性が変化してしまう可能性もあった．

いずれの評価でも，領域分割のために必要な実行情報として消費電力を用いる．

5.2 評価環境

評価環境として，電力計測システムや周波数制御ライブラリなどからなるPowerWatch[11]を用いる．周波数制御ライブラリをソースコードの任意の位置に挿入することで，プログラム中の任意部を指定した周波数で動作させることができる．

測定対象マシンとしてOpteron148（2.2 GHz, L2：1 MB）を搭載した16ノードクラスタを用いた．各ノードは電力測定装置を介してコンセントに接続されている．メモリはDDR-SDRAM 1 GBであり，カーネルはLinux kernel 2.6.19，コンパイラはgcc 4.1.1，MPIライブラリとしてLAM MPI 7.1.3を用いた．評価ベンチマークとしてNPB（NAS Parallel Benchmarks）3.2.1-MPIを用いた．カーネルベンチ5種について，それぞれCLASS=Cで実行する．今回の評価では，実行情報取得時のデータセット，プロファイル取得時のデータセット，および電力最適化時のデータセットはいずれも同じものを使用した．小さなデータセットから大きなデータセットでの振舞いを予測できればよいが，MPIの実装によってはデータセットの大小によって通信アルゴリズムが変化するなど，正確な予測は難しい．したがって，本稿では共通のデータセットを使用した．

表1 周波数と電圧の組
Table 1 The pair of voltage and frequency.

frequency [MHz]	voltage [mV]
2,200 (Standard)	1,400
2,000	1,350
1,800	1,300
1,000	1,100

Linux kernel 2.6.19の標準割込み間隔は250 Hzである．できる限り精密な実行情報の取得が望まれるが，一方でこの割込みによるオーバヘッドを考慮しなければならない．250 Hz間隔で実行情報を取得した場合，各種ベンチマークの実行時間は割込みを行わないときと比較してわずかな増加（おおむね1%以下の増加）であった．この程度の差は誤差の範囲内と考え，250 Hz間隔で実行情報を取得する．

今回使用した周波数と電圧の組を**表1**に示す．周波数制御ライブラリによって，CPUの動作周波数を変更する．この変更に要する時間はおおむね30 μsec程度であった．この時間より大きな時間単位で領域を分割しなければ最適化による効果が得られない．今回の評価では，周波数変更のためのオーバヘッドを考慮し実行時間が5 msec以下である領域については周波数変更を行わず，隣接する領域と同一の周波数を選択した．

AC電源線に流れる電流を複数ノードまとめて測定し，これを実効値に変換することでクラスタ全体の消費電力を評価する．ゆえに，クラスタ全体の電力データにはATX電源におけるAC-DC変換損失も含まれる．また，同時にノード0のATX電源線（CPU線，12 V線，5 V線，3.3 V線）について電力測定を行う．全ノードのATX電源線に対して電力測定を行うことが望ましいが，測定装置のポート数制約によって，今回はこのような手法をとった．なお，この問題については測定器の拡張により対応することができる．回数を分けて電力測定を行う方法もあるが，通信を含むようなプログラムでは性能にばらつきが発生し，正確な最適化を行えない可能性が高いため採用しなかった．

6. 評価結果

本章では，提案した手法によるインスツルメントがプログラムの実行に与える影響を評価するとともに，分割した領域に対して電力最適化アルゴリズムを適用する．並列プログラムにおいては，通信時に動作周波数を下げることにより，性能をそれほど低下させるこ

となく消費電力を大幅に削減できることが知られている．本提案手法では，このような区別だけでなく計算部についてもその動作に応じて複数個に分割できると考えられる．

提案手法では，以下に示す順序で領域分割，電力最適化を行う．

(1) 領域分割を行うために実行情報を取得する．
(2) プロファイル取得のための予備実行．
(3) 電力最適化アルゴリズムを適用した後の本実行．

それぞれの処理について次節以降で評価する．

ベンチマークプログラムをそのまま実行したときとプロファイル取得のためのコードを挿入したときの実行時間を比較する．プロファイル取得コードの挿入方法としては，ユーザが定義したすべての関数単位，その中でも実行時間の長い上位 5 つの関数のみに適用した場合，提案手法の 3 種類について比較する．

6.1 分割した領域とその特性

本節では実行情報を取得，および領域分割について評価する．

図 5 に IS（CLASS=C）の主要ループを実行した際のノード 0 における消費電力遷移と，そのときに実行している領域を示す．ここでは，IS 実行における主要ループ 2 回分を拡大表示している（loop1 は主要ループ 1 回目を意味する）．2,200 MHz で処理を実行し，消費電力が 1%以上変動した場合に特性が変化したと見なした．これは，測定対象として用いた PC の構成と電力計測装置の精度を考慮したものである．また，同一の領域を実行している部分を矢印で示している．

rank 関数では，MPI 関数を実行する以前に主要なループが 2 つ存在する．1 つは bucket 中のキー数を数えるループ（ここでは rank_1 とした），もう 1 つは bucket のソートを行うループ（ここでは rank_2 とした）である．これらを実行したときの消費電力は異なっており，提案手法によって別の領域と見なした．関数単位のみによる分割では，このような変化に対応することができない．評価結果より，提案手法では関数単位のみ，ループ単位のみといった分割に制限されないことが確認できる．

また，MPI_Alltoallv 関数内で大幅な電力変動が確認できる．MPI_Alltoallv 関数内の処理を解析し電力最適化を行うことも可能であるが，ネットワークトポロジやホスト数の変化，さらにはパケット衝突の有無によって挙動が変化する可能性もある．したがって，MPI 関数内の最適化は難しいと判断し，MPI 関数はそれ自身を最小の Block としている．

図 6 に FT（CLASS=C）の主要ループを実行した際のノード 0 における消費電力遷移とそのときの実行領域を示す．動作周波数は 2,200 MHz である．主要な関数は fft() であり，この中で計算と通信が発生する．fft() の後半に cffts1() と cffts2() が呼ばれるが，これらの消費電力の差はほとんどない．したがって，これらはまとめて 1 つの領域と見なした．なお，これらは内部で同じ関数を呼ぶなど非常に似た動作を示している．

各ベンチマークにおいて分割した領域の内訳を**表 2** に示す．呼び出された関数（callee）がつねに 1 つの領域と見なされるような分割が行われたとき，これを関数単位とした．すなわち，callee 側でコードをイン

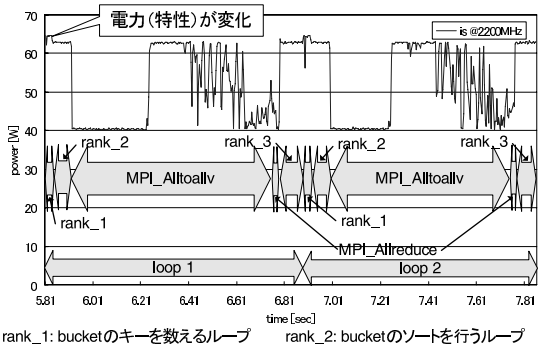


図 5 IS 実行時の電力遷移（標準周波数）
Fig. 5 Power profile of execution of IS (default frequency).

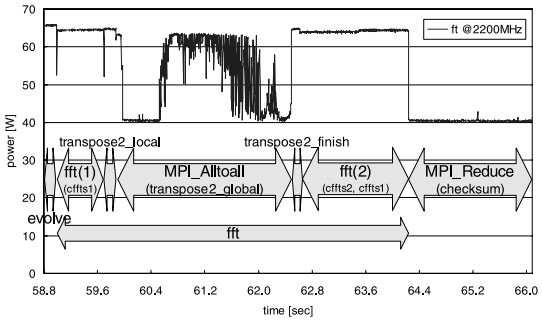


図 6 FT 実行時の電力遷移（標準周波数）
Fig. 6 Power profile of execution of FT (default frequency).

表 2 ベンチマークと領域の内訳
Table 2 Detail in region of benchmark.

benchmark	ユーザ関数	ループ	MPI 関数	その他
IS	1	3	2	0
CG	2	0	0	0
FT	5	0	0	1
MG	0	1	0	6
EP	1	0	0	0

表 3 ベンチマークの実行時間
Table 3 The execution time of benchmark.

benchmark	分割なし [sec]	提案手法 [sec]		関数単位 [sec]		関数単位（5 つ）[sec]	
IS	14.9	15.2	(+1.58%)	>7,200	(—)	15.2	(+1.7%)
CG	119.4	122.0	(+2.10%)	1,342.5	(+1,024%)	747.4	(+526%)
FT	167.4	168.8	(+0.88%)	2,872.7	(+1,616%)	2,616.1	(+1,463%)
MG	32.6	34.5	(+3.67%)	105.3	(+216%)	36.3	(+8.8%)
EP	137.4	137.5	(+0.07%)	137.5	(+0.07%)	137.5	(+0.07%)

スツルメントしても結果が変わらない場合である．領域とする場合としない場合が混在する関数についてはその他に分類した．

NPB のカーネルベンチマークに提案アルゴリズムを適用した結果，その多くでプログラム中で定義した関数（以降，ユーザ関数とする）による分割が行われた．これは，NPB のプログラムが関数ごとに機能分割されているためである．MPI 関数を 1 つの領域と見なすことは少なかった．これは，MPI 関数とわずかな計算処理を実行するユーザ関数単位を 1 つの領域と見なすことが多かったためである．また，ユーザ関数内で特定のループとわずかな処理を行う場合も多く，これもユーザ関数単位に分類されている．一方，IS においては rank() 関数内のループで挙動の変化を観測し，ループ単位による領域分割を実現している．FT において複数の関数からなる 1 つの領域を定義する場合があった．これはユーザ関数，ループ単位のいずれにも該当しないためその他に分類した．MG では関数呼び出しによって領域と見なす場合と見なさない場合に分かれる関数が 6 つあった．これは，引数によって関数の特性が変化するためである．これらは関数単位に分類せず，その他とした．なお，今回は関数の複製を作成することで caller の異なる関数呼び出しに対応している．従来，CG を人手による解析で最適化を行う際には MPI_Send を 1 つの領域と見なしていたが，提案手法では領域としなかった．これは，MPI_Send 関数を実行するために要する時間を短いと判断したためである．

6.2 コード挿入がプログラム実行に与える影響

本節では，プロファイル時について評価を行う．主に，プロファイル取得コードを挿入した際の実行時間の変化について議論する．

NPB 3.2.1-MPI を実行したときの実行時間を**表 3** に示す．ここでは，プロファイル取得のためのコードを挿入しない場合と提案手法，すべての関数の前後にプロファイル取得コードを挿入した場合，全体の実行時間が長い上位 5 つの関数についてのみコードを挿入した場合について比較する．なお，関数単位で EP にコードを挿入する場合，時間計測以外のサブルーチンは vranlc() のみである．よって，vranlc() とその前後で領域を分割するのみである．また，関数単位で分割した IS については，実行時間が 2 時間を超えても終了しなかったためベンチマークの実行を打ち切った．

IS，CG，FT，MG において，関数単位でプロファイル取得コードを挿入すると実行時間が大幅に増加した．これは，プロファイル取得コードを大量に呼び出したためである．特に IS においては 500 倍近い時間を要しても処理が終了しなかった．プロファイル取得コードの挿入数を制限することでコードの実行回数が減少し，実行時間の増加を抑えることができるが，それでも CG，FT のオーバヘッドは大きい．FT や MG に対して提案手法を適用した場合，指定した領域数は多いもののオーバヘッドは小さいという結果になった．これは，極端に実行時間の短い関数を大量に実行する場合に，プロファイル取得コードを挿入しなかったためである．MG では，特定の関数呼び出し時のみ領域と見なす場合があった．これにより，同じ関数であっても比較的実行時間の長い場合のみを 1 つの領域と見なした．仮に，一部の呼び出しだけではなくその関数を呼び出す際に必ずプロファイル取得コードを実行すると実行時間は 38.4 sec となり，領域分割を行わないときと比較して性能が 15.2%低下する．

EP においては，領域の分割方法による実行時間の差はほとんど見られない．これは，いずれの方法においてもプロファイル取得コードがほとんど実行されなかったためである．

以上より，提案手法ではオーバヘッドが少なく，プログラム本来の特性を大幅に変化させずにプロファイルを取得することができることが分かる．

6.3 電力最適化アルゴリズム適用結果

本節では，電力最適化アルゴリズムを適用した際の電力について評価する．

提案手法によりプログラムを複数領域に分割し，プロファイルを用いた電力最適化を行う．プロファイルを用いた電力最適化はすでに数多く提案されている[1),8),10)]が，ここでは我々が提案した周波数切替えオーバヘッ

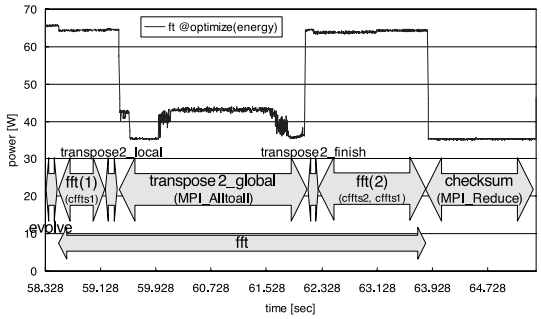


図 7 電力最適化時の電力遷移（FT）
Fig. 7 Power profile of execution of FT (optimized power consumption).

ドを考慮した電力最適化手法[11] を用いた．今回提案した領域分割手法によってプロファイル取得コードの挿入位置を決定し，動作可能な複数の周波数でプログラムを実行する．このプロファイル情報を用いて，プロファイル取得コードを挿入した箇所の中から実際に周波数変更コードを挿入する位置を決定する．我々が提案してきた手法[11] では，周波数変更のためのオーバヘッドを考慮している．しかしながら，プロファイル取得コードの挿入方法によってはプロファイル取得の際の特性が変化してしまう可能性があった．

今回提案した領域分割手法を用いてプログラムを複数領域に分割し，電力最適化を行う．通信待ち時間に低い周波数・電圧を選択することで，電力を削減できる．提案手法により通信を検出し，これを 1 つの領域に指定できると考えられる．また，メモリアクセス部などの CPU 性能を必要としない部分においても周波数を下げることによって消費電力を削減できる可能性がある．

NPB3.2.1-MPI（CLASS=C）について評価を行う．EP はプログラムの大部分が計算であり，周波数を落とすことで性能が大幅に低下することが知られている．電力最適化アルゴリズムを適用したところ，つねに最高周波数を選択しエネルギーおよび EDP に変化はなかった．

提案手法により FT を複数領域に分割し，電力最適化を行った．**図 7** に電力最適化を行った際のノード 0 の電力遷移示す．主要ループに対して 6.1 節で示した領域分割が行われた．電力最適化を行った結果，MPI 関数で 1,000 MHz，それ以外の部分では 2,000 MHz を選択し，5.48%のエネルギーを削減した．

NPB-MPI を用いて，EDP 最適化，エネルギー最適化を行った際の結果を**表 4** に示す．いずれも，2,200 MHz で実行したときとの比較である．

エネルギー最適化では，実行時間に制限を加えることなくエネルギーが最小になるよう実効系列を決定している．そのため，EDP 最適化と比較して低い周波数を選択し，実行時間が増加する傾向にある．今回の評価において，エネルギー最適化時における実行時間増加は低い周波数を使用したことが主な要因である．周波数変更コードが頻発し，これによってプログラムの実行時間が大幅に増加することはなかった．

表 4 EDP/エネルギー最適化の結果
Table 4 Result of optimizing EDP/Energy.

	EDP 最適化 [%]	エネルギー最適化 [%]	
benchmark	EDP	エネルギー	実行時間
IS	−6.72	−8.09	+1.79
CG	0	−1.66	+4.64
FT	−1.46	−5.48	+8.22
MG	−4.38	−12.0	+12.5

電力最適化アルゴリズムでは，プロファイル取得コードの挿入位置から実際に周波数変更を行う位置を決定する．それぞれの領域について最適周波数を計算し，必要に応じて周波数変更コードを実行する．今回の評価では，電力最適化手法によってオーバヘッドが大きいと判断することはなかった．最適な動作周波数が連続する領域間で偶然一致した場合に周波数制御コードを挿入しなかったが，それ以外についてはプロファイル取得コード挿入位置と同じ部分に周波数制御コードを挿入した．

CG では，主要な関数（conj_grad）中で各種通信や計算処理などの処理が行われるが，大幅な消費電力の変動は見られなかった．初期化部と conj_grad とでは挙動が異なるが，EDP 最適化時には 2,200 MHz，エネルギー最適化時には 2,000 MHz と一定の周波数を選択した．EDP 最適化ではすべて標準周波数である 2,200 MHz で実行しているため，EDP の削減率は 0 となっている．

関数単位でプロファイルを取得したときの消費電力最適化結果と比較する．6.2 節の結果からも分かるように多くの場合においてプロファイル取得のためのオーバヘッドが大きくなり，正常なプロファイルを取得することができない．オーバヘッドの比較的少ない IS や MG（実行時間の長い上位 5 関数を領域とした場合）では，電力最適化を行った結果はほぼ同じとなった．提案手法によって領域を分割した後，様々な周波数での実行結果を基にして電力最適化を行うが，異なった領域間で同じ周波数を選択する場合がある．領域の区切り方は異なっても実際に同じコードを実行する際の周波数は同じになるという現象が発生したため，電力削減率が同等という結果となった．

7. 考　　察

7.1 領域分割の粒度とプログラムに与える影響

提案した領域分割手法により，

- 極端に実行時間の短い挙動の変化を除いては，挙動を正確にとらえた領域分割を行うことができる，
- 領域に対して最適化制御を行う際に，そのオーバヘッドにより性能が大幅に低下することを防ぐことができる，

を同時に満たす領域分割を行うことができた．6.1 節，6.2 節の結果からこれを確認することができる．

7.2 エネルギー削減効果

本稿で提案した方法によりプログラムを複数領域に分割し，電力最適化を行った．その結果，IS において 8.09%の電力削減を確認した．

本環境において，最高動作周波数である 2,200 MHz で実行したときの各 ATX 電源線に流れる電流を測定し，消費電力の測定を行った．結果，プロセッサインテンシブな場面において CPU に要する消費電力は 40 W 弱，その他構成要素に要する消費電力は 30 W 程度であることが分かった．AC 電源線での電力測定結果は各ノード 100 W 程度であり，これらの差は電源における AC-DC 変換損失によるものであると考えられる．各ノードに供給される電力の約 40%がプロセッサに供給されており，DVFS によってプロセッサの電力を削減するものの，システム全体としては電力削減の割合は小さい．

ノード 0 のプロセッサに供給される消費電力を比較する．NPB-MPI IS を実行したところ，電力最適化により標準周波数（2,200 MHz）で実行したときと比較して 22.2%のエネルギーを削減した．しかしながら，実行時間の増加にともなうその他構成要素のエネルギー消費により，システム全体としては 8.09%の削減にとどまっている．よって，今後はプロセッサのみではなく周辺の構成要素を含めた電力最適化を検討しなければならないと考えられる．

7.3 他の最適化への適用可能性

本提案手法は，プログラムを木構造に変換し，実行情報を付与して領域分割を行う．今回，付与する情報として電力情報を用いた．これは，電力最適化を行うためには電力情報を取得することが必要不可欠であり，かつ電力情報を用いた領域分割が適切であると考えたためである．しかしながら，キャッシュアクセスなど CPU の振舞いを用いることも可能である．実行情報としてどのような項目を用いるべきかについては，今後検討する必要がある．

また，今回はプロセッサの電力削減機構である DVFS 最適化を対象としたが，他の最適化へ利用することも可能である．たとえば，ディスクを停止させることによって電力を削減する手法がある[16]．これは，DVFS に比べて非常にオーバヘッドが大きく，細粒度での制御は避けなければならない．提案手法を用いることで，極端に細粒度な制御を回避することができる．また電力最適化だけでなく，様々な最適化に利用できると考えられる．

8. 結　　論

本稿では，プログラム実行に与える影響の少ないインスツルメント手法を提案した．本提案手法は，タイマ割込みによって実行情報を取得し，ソースコードの解析結果と組み合わせてコード挿入位置を決定する．これにより，プロファイル取得がプログラムに与える影響を少なくするとともに，領域の挙動を正確にとらえたプログラム分割を行える．これにより，プログラムの精密な挙動を知らずに適切な領域分割を行えるようになった．

提案手法によってプログラムを複数領域に分割した後，プロファイルを利用した電力最適化を行った．その結果，NPB-MPI（MG）実行時にシステム全体で 12.0%のエネルギーを削減し，プロセッサのみに着目した場合 NPB-MPI（IS）実行時に 22.2%のエネルギーを削減した．

今後は，特性抽出や領域分割をより容易に行うために本手法をコンパイラに組み込むことを考えている．現在，MPI 関数内部の処理については最適化の対象としておらず，また，呼び出し方法の異なる関数についても人手による補助が必要となっている．コンパイラと協調することでこれらの問題を解決することができる．また，小さなデータセットから実際に実行するプログラムの特性を予測する機構を開発することで，提案した領域分割手法をより有効に活用できると考えられる．

謝辞　本研究の一部は科学技術振興機構・戦略的創造推進研究事業（CREST）「低消費電力化とモデリング技術によるメガスケールコンピューティング」，および「省電力でディペンダブルな組込み並列システム向け計算プラットフォーム」による．

参　考　文　献

1) Hsu, C.-H. and Kremer, U.: The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction, *PLDI*,

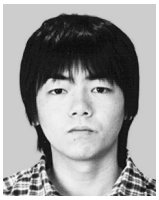
pp.38–48 (2003).
2) Hsu, C.-H. and Feng, W.-C.: A Feasibility Analysis of Power Awareness in Commodity-Based High-Performance Clusters, *Cluster Computing* (2005).
3) Adiga, N.R., et al.: An Overview of the BlueGene/L Supercomputer, *Supercomputing*, p.60 (2002).
4) Warren, M.S., Weigle, E.H. and Feng, W.-C.: High-Density Computing: A 240-Processor Beowulf in One Cubic Meter, *Supercomputing*, p.61 (2002).
5) Nakashima, H., Nakamura, H., Sato, M., Boku, T., Matsuoka, S., Matsuoka, S., Takahashi, D. and Hotta, Y.: MegaProto: 1 TFlops/10kW Rack Is Feasible Even with Only Commodity Technology, *SC06*, p.28 (2005).
6) Chen, G., Malkowski, K., Kandemir, M.T. and Raghavan, P.: Reducing Power with Performance Constraints for Parallel Sparse Applications, *HPPAC in IPDPS* (2005).
7) Wu, Q., Martonosi, M., Clark, D.W., Reddi, V.J., Connors, D., Wu, Y., Lee, J. and Brooks, D.: A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance, *MICRO-38*, pp.271–282 (2005).
8) Freeh, V.W. and Lowenthal, D.K.: Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster, *PPoPP*, pp.164–173 (2005).
9) Isci, C., Contreras, G. and Martonosi, M.: Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management, *MICRO-39* (2006).
10) Ge, R., Feng, X. and Cameron, K.W.: Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters, *SC05* (2005).
11) Hotta, Y., Sato, M., Kimura, H., Matsuoka, S., Boku, T. and Takahashi, D.: Profile-based Optimization of Power-Performance by using Dynamic Voltage Scaling on a PC cluster, *HPPAC in IPDPS* (2006).
12) Lim, M.Y., Freeh, V.W. and Lowenthal, D.K.: Adaptive, Transparent Frequency, and Voltage Scaling of Communication Phases in MPI Programs, *SC06* (2006).
13) 佐々木広，浅井雅司，池田佳路，近藤正章，中村宏：統計情報に基づく動的電源電圧制御手法，情報処理学会論文誌：コンピューティングシステム（ACS16），pp.80–91 (2006).
14) Brooks, D.M., Bose, P. and Schuster, S.E., et al.: Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors, *IEEE Micro*, Vol.20, No.6, pp.26–44 (2000).
15) Sato, M., Satoh, S., Kusano, K. and Tanaka, Y.: Design of OpenMP Compiler for an SMP Cluster, *EWOMP*, pp.32–39 (1999).
16) Gniady, C., Butt, A.R., Hu, Y.C. and Lu, Y.-H.: Program Counter-Based Prediction Techniques for Dynamic Power Management, *IEEE Trans. Comput.*, Vol.55, No.6, pp.641–658 (2006).

(平成 19 年 1 月 22 日受付)
(平成 19 年 4 月 25 日採録)

木村　英明（学生会員）
昭和 58 年生．平成 16 年小山工業高等専門学校電子制御工学科卒業．平成 18 年筑波大学第三学群情報学類卒業．現在，同大学大学院システム情報工学研究科在学中．低消費電力クラスタシステムに関する研究に従事．

佐藤　三久（正会員）
昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年通産省電子技術総合研究所入所．平成 8 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．平成 13 年より，筑波大学システム情報工学研究科教授．同大学計算科学研究センター勤務．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グリッドコンピューティング等の研究に従事．IEEE，日本応用数理学会各会員．

堀田　義彦（学生会員）
昭和 54 年生まれ．平成 15 年筑波大学第三学群情報学類卒業．現在，同大学大学院システム情報工学研究科在学中．クラスタシステム等の低消費電力化に関する研究に従事．IEEE 会員．



今田 貴之（学生会員）

昭和 58 年生．平成 18 年青山学院大学理工学部情報テクノロジー学科卒業．現在，筑波大学大学院システム情報工学研究科在学中．並列処理，低電力高性能クラスタシステムに関する研究に興味を持つ．