社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－HPC－108 (13)
2006／10／6

DVFS制御を目的としたプログラムの領域分割

木村英明† 佐藤三久† 堀田義彦†
今田貴之† 朴泰祐† 高橋大介†

本稿では，プログラムに与える影響の少ない領域分割方法を提案する．近年，多くのCPUにDVFS (Dynamic Voltage and Frequency Scaling) 機構が搭載されるようになり，これを適切に制御することで電力性能を向上させることができるようになってきた．すでに，プログラムをいくつかの領域に分割しプロファイルを取得することで電力制御を行う方法が数多く提案されている，しかしながら，プロファイルの領域分割について詳しく述べているものは少なく，プロファイル取得のためのオーバヘッドを考慮していないことが多い．そこで，我々はプロファイル取得のためのオーバヘッドを考慮したプログラムの領域分割方法を提案する．本手法は，実行データと電力データ，プログラムの静的な解析結果を組み合わせて適切な粒度の領域に分割し，領域区分コードを挿入するものである．

我々は逐次プログラムによる評価を行った．その結果，プログラムの構造のみに頼った単純な分割方法と比較して領域区分コードの実行回数を大幅に削減することができることが分かった．さらに，適切な周波数選択アルゴリズムと組み合わせることで電力を削減できることが分かった．

Dividing program into regions for controlling DVFS

HIDEAKI KIMURA,† MITSUHISA SATO,† YOSHIHIKO HOTTA,†
TAKAYUKI IMADA,† TAISUKE BOKU† and DAISUKE TAKAHASHI †

In this paper, we propose a new technique to divide into region with low impact. Recently, many high performance microprocessors have DVFS(Dynamic Voltage and Frequency Scaling) mechanism, we can reduce energy consumption by using of this mechanism adequately. While many researchers have proposed a lot of techniques of controlling the power by dividing the program into some regions, a few reports address the division of the profile, and the overhead for the profiling was not considered. We propose a technique to divide the program into region, taking the overhead for the profiling into account. Our algorithm enables to divide the region by instrumenting the code using the execution profile, the power profile, and the structure of program.

We have applied our technique to a serial program. As a result, the execution times of the instrumented code decreases comparing to that with a simple division technique of relying on only the program structure. And, we found that the power consumption can be reduced by combining our algorithm with the DVFS frequency selection algorithm.

1. 序　論

近年，CPUの性能向上とともにCPUの消費電力が増加している．CPUの消費電力増加に伴う発熱の増加は，システムの信頼性を低下させるとともに実装密度を低下させる．そのため，近年では多くのCPUで周波数と電圧を動的に変更するDVFS (Dynamic Voltage and Frequency Scaling) 機構を搭載するようになってきており，HPC分野においてもDVFSを利用して電力を削減する手法が数多く提案されてきている．我々もプロファイル情報を用いて最適周波数選択アルゴリズム[1]を提案してきた．これは，プログラムをいくつかの小領域に分割しプロファイル情報を用いて各小領域の最適な周波数を決定するものである．しかしながら，プログラム分割方法の詳細について言及しておらず，計算時と通信時といったような経験則に基づいて人手で分割をしていた．

DVFS制御のためのプログラムの領域分割についていくつかの手法[2][3][4]が提案されている．これらは，キャッシュミス回数やプログラムの構造から領域を分割するとともにプロファイルを取得し，電力制御を行っている．いずれもプログラムの動作が変化する箇所に着目しているのみであり，領域を分割したことによるオーバヘッドを考慮していない．

電力最適化を目的としたプロファイルを取得するために，領域区分のためのコードを挿入し，領域を分割

† 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

する場合を考える．細粒度に領域を指定すると，挿入した領域区分コードを頻繁に実行し，プロセッサの挙動が変化する可能性がある．このような場合，正確なプロファイルを取得できない．一方，あまりにも粗粒度にすると取得できるプロファイル情報が不適当になる．複数の挙動を示す部分を同一の領域とみなすことは適当でない．

そこで本稿では，適切な粒度の領域にプログラムを分割方法する方法を提案する．本稿では特にDVFS制御のための領域分割を対象とする．我々が提案する手法は，電力データと実行データ，プログラムの構造からプログラムを分割するものである．定期的な割り込みにより実行データを取得し，電力データを対応付けて実行電力データを作成する．この実行電力データとプログラムの構造解析結果から適切な粒度の領域を決定する．これにより，"プロファイルを取得するためのオーバヘッドが少ない領域分割" を実現する．

さらに，我々が提案してきた周波数選択アルゴリズム[1] と組み合わせることでエネルギー削減を試みる．

本稿の構成は次のようになっている．2章では，関連研究について述べる．3章では，電力情報とプログラムの構造から領域を分割する方法について述べる．4章では，測定環境について述べる．5章では，提案手法を利用した周波数制御アルゴリズムを適用した際の結果を示す．6章では，考察と今後の課題について述べる．最後に，まとめを述べる．

2. 関連研究

プロファイル情報を元に電力最適化制御を行う手法はすでに数多く提案されている．プログラム分割方法について言及しているものもあるが，いずれもプロファイル取得のためのオーバヘッドを考慮していない．

Wu ら[2] は，平均L2キャッシュミス，平均メモリ転送などを用いてプログラムを分割している．Freeh ら[3] は，プログラムの領域分割のために operations per miss を用いている．いずれも，DVFS制御のための領域分割が目的である．両者ともに観測値の変動のみで領域を決定しており，観測値の変動よってはプロファイル取得時のオーバヘッドが大きくなる可能性がある．また，キャッシュミスやオペレーション数を測定するために専用のツールが必要となる．

Hsu ら[4] は，ソースコードの解析結果からプログラムを複数の領域に分割し，これを用いて周波数制御を行っている．ソースコードを解析し，処理の流れや繰り返し回数等を求め，これらを基にして領域に分割する．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，全く別の挙動を示す箇所を同じ領域とみなす可能性がある．

3. プログラムの領域分割手法

本章ではDVFS制御を目的としたプログラムの領域分割方法ついて述べる．領域に分割したことでプログラムの挙動が大きく変化せず，かつ領域の特徴を捉えられるような分割を目的とする．また，この分割を行った後に領域区分コードを挿入しプロファイルを取得するとともに最適周波数選択アルゴリズム[1] によって電力最適化を行う．

3.1 プログラム領域分割の基本方針

我々が目的とする領域分割とは以下に示す3つの条件を同時に満たす分割である．

(1) ソースコードに対して分割する領域を指定する．
(2) プロファイル取得のためのオーバヘッドが小さくなる領域分割．
(3) 挙動を正確に捉えたプロファイルを取得できる領域分割．

我々は，プログラムのソースコードに対して領域を分割する．これは，ソースコードの情報を反映することができるためである．具体的には，ソースコード中に領域区分コードを挿入し，これに囲まれた部分を領域として認識する．

ソースコード上にプロファイル取得コードを挿入する場合，その粒度が重要になる．例えば，細粒度に分割すると挿入したコードを頻繁に実行しプログラムの挙動が変化してしまう可能性がある．これは，領域区分コードを頻繁に実行することによるものである．特に，1つの領域を1度実行するために要する時間が極端に短い場合はこれが問題になる．

逆に粗粒度に分割した場合，プログラムの挙動を正確に捉えられない可能性がある．我々はプログラムの特性を正確に示すためにプログラムを領域に分割する．ゆえに，同一領域内で行われる処理の挙動は似ているべきである．

3.2 プロファイル取得のためのオーバヘッドを考慮したプログラムの領域分割

我々は，プロファイル取得時の振る舞いを考慮したプログラムの領域分割方法を提案する．本手法では，電力データとプログラム実行データ，ソースファイルの解析から適切な領域に分割する．**図1** にプログラムを複数の領域に分割する際の処理の流れを示す．

3.2.1 実行データの取得

プログラムを実行するとともに実行データを作成する．実行データは以下の処理によって作成する．

(1) 一定時間間隔でシグナルを発生させ，プログラムカウンタ値（PC data）を取得する．
(2) addr2line によってプログラムカウンタ値をファイル名と行番号（Exec. data）に変換する．

setitimer システムコールを利用することで，一定時間間隔でシグナルを発生させることができる．この

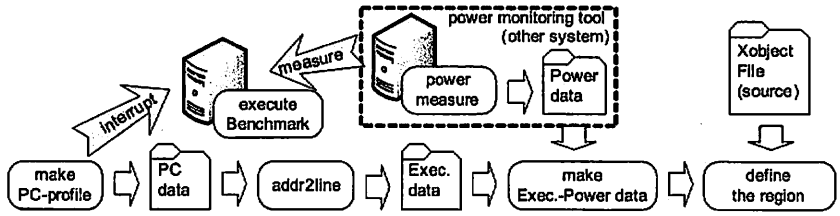

図 1　プログラムの領域分割

シグナル発生時にプログラムカウンタ値を取得することで，“その瞬間にどの処理を実行していたか” を調べることができる．

プログラム終了後，記録したプログラムカウンタ値に対して addr2line を実行する．addr2line を実行すると，実行ファイルのデバッグ情報からプログラムカウンタ値に関連付けられているソースファイル名と行番号を求めることができる．この時系列に並んだ実行行番号を実行データとする．

3.2.2　電力データの取得

プログラムを実行するとともに電力測定システムを利用して電力データを取得する．電力データを実行データと時系列に対応付けるため，両者の測定間隔が同じであるようにする．また，実行データと電力データは別マシンで測定することになる．したがって，これらの開始時刻を合わせる必要がある．

3.2.3　実行-電力データの作成

実行データと電力データを対応付ける．実行データの行番号情報から，各コードの実行時間とその時の電力を求めることができる．命令が同じであればデータが異なっても同じ処理であるとみなし，ソースコードの行ごとに集計する．各行ごとに実行している時間とそのコードを実行する際に要する平均電力を求める．このコード別の実行時間，平均電力データを実行-電力データと呼ぶ．

3.2.4　プログラム構造解析

プログラムの実行によってデータを取得するとともに，ソースファイルの構文解析を行いプログラムの構造を調べる．プログラムの構造を調べることで，適切な領域区分コード挿入位置を決定することができる．

ここでは，Omni OpenMP Exc compiler tool kit の Xobject 形式[5] を利用する．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から，Block, Statement などの構造に分解することが可能である．また，Xobject 形式は C 言語プログラムに復元可能である．

Statement は分岐を持たない文や条件式を表現しており，これらを複数個まとめて Block が形成される．複数の Block から BlockList を構築し，これによりネスト構造を表現している．また，データ構造に対して様々なメソッドが定義されている（**図 2**）．

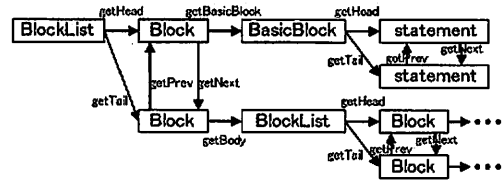

図 2　Block 構造と各メソッド

得られた構造に対して 3.2.3 節で得られた実行-電力データを付与する．実行-電力データと本節で行った解析結果により，“Block を 1 回処理するのに要する実行時間” を求めることが可能であり，この情報も付与する．

3.2.5　実行電力データとプログラム構造を用いたプログラムの領域分割

電力情報とプログラムの構造から分割可能な領域を決定する．消費電力の挙動が同じ領域をプログラムの構造に沿ってまとめ，かつ 1 領域の実行時間ができる限り大きくなるようにする．**図 3** にプログラムの領域分割アルゴリズムを示す．本アルゴリズムは以下に示す 3 種の状態を想定している．

(1)　**Block 内の電力データがすべて一定範囲内におさまる場合．**
この場合，Block はひとつの領域とみなすことができる．さらに上位の Block において他の Block と結合することができる可能性がある．このため，この Block 内の電力データを上位 Block に渡す．上位 Block に渡す電力データはその Block 内の平均電力である．

(2)　**Block 内で電力データが大幅に変動する場合．**
block 内で電力が大幅に変動した地点に領域区分コードを挿入する．これにより，Block 内に複数の領域を作成することができる．Block 内で大幅な電力変動が N 回観測された場合，その Block は (N+1) 個の領域に分割される．

(3)　**Block 内の処理を 1 度行うのに要する時間が非常に短い場合．**
このような状態の部分を領域に設定してしまうと，周波数切り替えが大量に発生し性能が低下する恐れがある．したがって Block の処理を 1

```
divideRegion(cycle)
{
   // for basic block
   if( ( block = getBasicBlock () ) == TRUE ){
      if ( power data is unchanged in this block){
           assume block to be one region
           return power _data;
      } else {
           create some regions using power_data.
      }
   )

   // for BlockList
   else if( ( block=getBody () ) == TRUE ){
      divideRegion (block.cycle);
      if( block.time / cycle >= Threshold ){
         if ( power data is unchanged in this block){
              assume block to be one region
              return power_data;
         } else {
              create some regions using power_data.
         }
      } else {
         return power _data;
      }
   }
}
```

図 3　領域分割アルゴリズム

回行うために要する実行時間が，あるしきい値以下の場合は領域としない．評価実験では周波数切り替えオーバヘッドの 100 倍をしきい値に設定した．一方，上位 Block においてこのような部分を何度も呼び出すことにより，1 回の実行時間は短いものの全実行時間に占める割合が大きくなる可能性もある．このような場合に対応するために，Block 内のデータを上位 Block に渡す必要がある．

この処理を下位 Block から上位 Block に向かって順次適用していくことにより，プログラムを複数の領域に分割することが可能である．これにより，ループや関数といった異なる階層での領域分割が可能となる．

3.3 最適周波数選択アルゴリズム

前節で示した手法により，プロファイル取得コードを挿入する位置を決定することができる．このように分割した領域に対して，周波数選択アルゴリズム[1] を用いて電力最適化を行うことができる．これは，動作可能なすべての周波数でプログラムを実行してプロファイルを取得し，各領域に対して最適な周波数を選択するものである．我々が提案した領域分割は，プロファイル取得のためのコード挿入による性能低下を考慮した領域分割を行っている．したがって，プロファイル取得のための挙動の変化は僅かである．

図 4 に最適周波数選択アルゴリズムを利用して電力最適化を行う時の処理を示す．

4. 評価環境

電力測定システム PowerWatch，周波数制御ライブラリなどを用いる．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，プログラム中の特定領域を指定した周波数で動作させる．本稿で提案したアルゴリズムを用いて挿入箇所を決定し，これにより周波数を変更しながらプログラムを実行する．

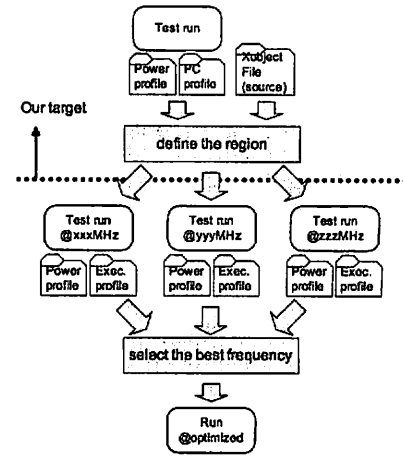

図 4　最適周波数選択アルゴリズムとの関連

表 1　周波数と電圧の組

frequency [MHz]	voltage [mV]
2200	1400
2000	1350
1800	1300
1600	1250
1400	1200
1200	1150
1000	1100

Opteron148 を搭載した測定対象マシンを用いて評価を行った．CPU が動作可能な周波数と電圧の組を表 1 に示す．メモリは DDR-SDRAM 1GB であり，カーネルは Linux kernel2.6.15 (with perfctr patch)，コンパイラは gcc4.0.0 である．評価ベンチマークは NPB3.2 であり，これを逐次プログラムとして実行した．電力測定システム PowerWatch[1] を用いて電力測定を行った．

5. 評価結果

5.1 領域分割アルゴリズム適用結果

標準周波数である 2200MHz で IS(CLASS=A) を実行した時の消費電力遷移を図 5 に示す．

提案アルゴリズムを電力プロファイルに適用する．Block 内の電力が 1%以上変動した場合，これを別領域とみなした．これは，測定対象として用いたマシンの構成と電力測定装置の精度を考慮して決定した．

領域分割アルゴリズムを適用した結果，

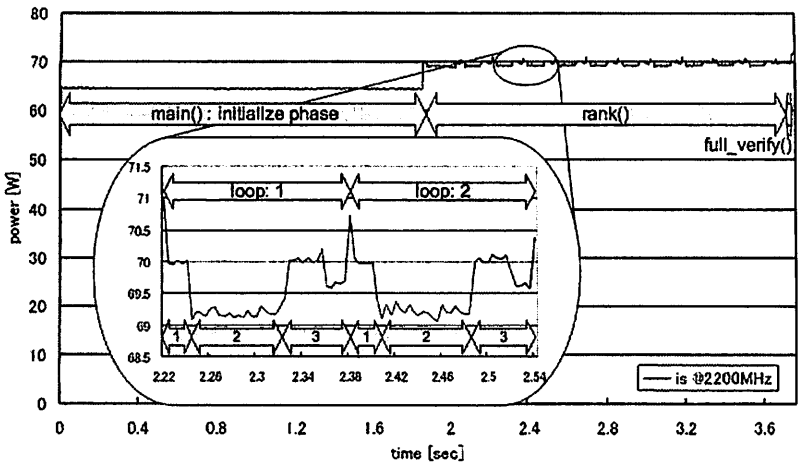

図 5 標準周波数実行時の消費電力遷移と領域分割

- main() の初期化部分
- rank() の前半部（拡大部 1 の部分）
  bucket_size[key_array[i]>>shift]++; のループ
- rank() の中盤部（拡大部 2 の部分）
  key_buff1[bucket_ptrs[key>>shift]++]=key; のループ
- rank() の後半部（拡大部 3 の部分）
  key_buff_ptr[key_buff2[i]]++; のループ
- full_verify()

の 5 つの領域に分割し，領域区分コードの実行回数は全部で 68 回であった．一方で，プログラム中にユーザが定義した関数をそれぞれ領域とみなした場合，領域区分コードを 3300 万回以上呼び出す．このような場合，領域区分コードの呼び出しオーバヘッドによりプログラムの挙動が変化し，プロファイル取得に要する時間が大幅に増加する．我々の手法は，プログラムの構造のみに頼った単純な分割方法と比較して領域区分コードの実行回数を大幅に削減することができた．

5.2 最適周波数選択アルゴリズム適用結果

5.1 節の方法によって領域を決定し，プロファイルを作成するとともに電力最適化を行う．周波数選択アルゴリズムを適用した結果，rank() の中盤部と full_verify() で 2000MHz を選択し，それ以外については標準周波数である 2200MHz を選択した．図 6 に最適周波数でプログラムを実行した時の電力遷移を示す．

最適周波数選択により，0.86%のエネルギーを削減した．削減率が低い理由については 6.2 節で考察する．

IS 以外の結果も領域区分コードの実行回数は大幅に削減するが，エネルギー削減率は低いという傾向であった．

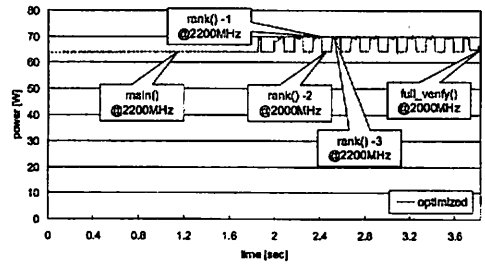

図 6 電力最適化時の消費電力遷移

6. 考　　察

6.1 電力データを利用することによる利点

従来の研究との最大の差異はプロファイル取得のためのオーバヘッドを考慮している点にあるが，領域分割のために電力データを使用している点も大きく異なる．従来手法の多くは，パフォーマンスカウンタ値を指標としてプログラム領域分割を行ってきた．我々が電力データを用いて領域分割した主な理由は以下に示す 2 つである．

(1) ソフトウェアでパフォーマンスカウンタ値を取得する場合，データ取得の際にプログラムに影響を与える可能性がある．

(2) 電力データを用いた場合，パフォーマンスカウンタ取得時と同等の分割を行うことができる．

パフォーマンスカウンタプロファイルを作成する場合，setitimer システムコールによるシグナル発生時にパフォーマンスカウンタ値を読み取る必要がある．よって，我々が提案した手法と比較して 1 回のシグナ

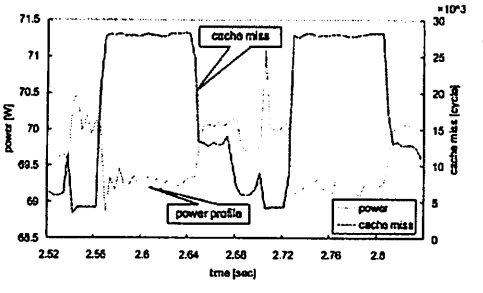

図 7 L2 キャッシュミスと消費電力（一部抜粋）

ル発生時に行う処理が増える．これにより，プログラムの挙動が変化する可能性がある．電力データは図 1 に示すように外部のマシンから取得可能であり，プログラムの挙動に与える影響は少ない．

電力データを用いた場合，パフォーマンスカウンタと同等の精度でプログラムを分割できる．図 7 に，NBB3.2-IS（CLASS=A）実行時の L2 キャッシュミス回数と消費電力遷移の抜粋を示す．キャッシュミス回数の取得には PAPI[6] を用いた．図 7 よりキャッシュミスが頻発している部分では低い電力，キャッシュミスの少ない部分では高い電力で処理する傾向が分かる．電力値はキャッシュミス回数に比べて同一領域内の変動が大きい傾向にあるが，適切なしきい値を用いることで対処可能である．

6.2 エネルギー削減効果

今回は 1 ノードで評価を行った．したがって，メモリアクセスが頻繁に発生する領域で低い周波数を選択できたのみである．従来の研究の多くは，通信のための待ちが発生する際に低い周波数を選択することでエネルギーや EDP を削減してきた．今回はこれが無いため，エネルギーの削減幅は少なかった．

また，周波数を選択する際に高い周波数を選択する傾向にあった．システム内部の電力を測定すると CPU が消費する電力はシステム全体の約半分であり，DVFS 制御による消費電力削減よりも実行時間の増加が影響したと考えられる．

6.3 今後の課題

今回は，並列環境ではなく 1 ノードでの評価を行った．これは，作成したプログラムの制限によるものである．作成したプログラムでは一定時間間隔でシグナルを発生させ，プログラムカウンタ値を取得した．しかし，通信時にシグナル発生による割り込みを行うことができず，データが欠落し実行データを作成できないという問題が生じた．この問題を解決するために，

(1) プログラムカウンタ値を取得する際に時刻を取得し，どの地点で通信が発生したかを調べる．

(2) 電力プロファイルと対応付ける際に，消費電力の変動から通信発生時刻を予測する．

といった方法が考えられる．しかし前者は割り込み時の処理が複雑化し，後者は精度の面で問題がある．

また，割り込みによって実装したため，測定間隔に限界がある．現在の Linux 環境下では 250Hz 間隔での割り込み処理が一般的である．これ以上の精度で測定を行う場合はカーネルレベルでの実装が必要となる．

7. 結　　論

本稿では，実行データと電力データ，プログラム構造を静的に解析したものを利用し，プログラムを複数の領域に分割する手法を提案した．提案手法は，“プロファイルを取得するためのオーバヘッドが少ない領域分割” を行うことができ，従来手法に多く見られたパフォーマンスカウンタなどのデータを必要としない．

提案手法を使用してプログラムを複数の領域に分割した結果，プログラムの構造のみに頼った単純な分割方法と比較して領域区分のためのコードの実行回数を大幅に削減することができた．提案手法によって分割したプログラムに対し，すでに提案されている周波数選択アルゴリズムを適用することでエネルギーを削減できることを確認した．

謝辞 本研究の一部は科学技術振興機構・戦略的創造研究推進事業（CREST）-情報社会を支える新しい高性能情報処理技術-「低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」による．

参 考 文 献

1) Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power-performance by using dynamic voltage scaling on a pc cluster. In *IPDPS Workshop on HPPAC*, 2006.

2) Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO*, pp. 271–282, 2005.

3) VincentW. Freeh and DavidK. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *PPoPP*, pp. 164–173, 2005.

4) Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI*, pp. 38–48, 2003.

5) Mitsuhisa Sato, Sigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of openmp compiler for an smp cluster. In *EWOMP*, pp. 32–39, 1999.

6) PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.