サンプリングによる消費電力の計測手法

小野美由紀[†1]　山本昌生[†1]　中島耕太[†1]

概要：HPC システムやデータセンターではシステム規模の増大や処理量の増加に伴い，消費電力も増大しており，電力を削減することが重要な課題となってきている．省電力化に関しては，これまではハードウェア手法による省電力技術や，計算ノード単位や装置単位の大粒度での監視制御技術の研究開発が進んできた．しかし，ソフトウェアの挙動と電力消費の関係解明や省電力観点でのプログラム最適化の研究は，ハードウェア主体の省電力技術の進歩と比べるとまだ十分ではなく，発展の余地が残されている．そこで，我々はプログラムの電力最適化を行うことを目標とし，関数単位での電力ホットスポットを明らかにする消費電力プロファイル分析について研究を行っている．これまでに，電力データ採取のための専用計測器やソフトウェアの改変を不要とする電力データ採取方法として，タイムベースサンプリングと同時に CPU が備える消費電力監視機能 RAPL の消費電力値を採取する方法を提案した．しかし，タイムベースサンプリングでは時間がかかる箇所がサンプリングされ，必ずしも電力消費が多い箇所をサンプリングできるとは限らない．本稿では，電力消費に関係する CPU の性能イベントをベースとしたサンプリングを行うことにより，従来のタイムベースサンプリングよりも精度の高い電力サンプリングを可能とする手法を提案する．提案手法を検証するため，電力消費に関係する 4 つの性能イベントをベースとしたサンプリングを個別に行い，採取した各サンプルデータを合成して複数イベントサンプリングを模擬した．電力計測を行った 6 つのアプリケーションのうち 5 つで誤差が 1%以下に収まり，一定の消費電力量によるサンプリングが可能であることを確認した．

Sampling-based Power-Measurement Method

MIYUKI ONO[†1]　MASAO YAMAMOTO[†1]　KOHTA NAKASHIMA[†1]

Abstract: Power reduction has increasingly become an important problem in HPC systems and data centers. This is because power consumption is growing in HPC systems and data centers with the increasing systems and workloads. In former works, concerning the power saving, the methodologies with hardware equipment had been studied well, and hardware energy-efficiencies have been already well-established. Moreover, the monitoring and control technologies, by the computing node or the device, have advanced. However, the study is not enough yet; the study on the clarification of the relation between software behavior and power consumption, and the study on the program optimization from a viewpoint of power saving. Therefore, for the power optimization of programs, we have been working on the power consumption profile method that identify the power consumption hotspots on a program code in the function level. In previous our work, we proposed the method of analysis of software power consumption in the function level, based on time-based sampling which includes data collected from Intel's Running Average Power Limit (RAPL) interfaces. RAPL is the interface for power consumption monitoring system and is equipped in CPU. This method requires neither modification of software nor instrumentation for measuring power consumption. However, time-based sampling does not always sample a power hotspot in a program. That is, its spot represents for the part where it does not consume much power, but takes much time. In this paper, we propose the power-based sampling, by which accuracy is higher than time-based sampling, using the performance event of CPU concerning the power consumption as the base event of sampling. To verify our proposed method, first, we do the event-based sampling measurement four times, using one performance event at a time out of four performance events correlated to the power consumption. Then, we simulate the multiple-events-based sampling by combining the four sampling data together. Finally, our profiling results demonstrate the deviation between expected value and simulated one is lower than 1% on five of six applications. Thereby, we can confirm that power-based sampling works well.

1. はじめに

ICT システムにおける省電力化の要求は，コストや資源枯渇の面だけでなく地球環境問題の観点から高まっている．これに伴い，計算機でも，CPU のマルチコア化以降，電力が設計要件や性能指標の一部として重要視されるようになっている．

HPC システムやデータセンターではシステム規模の増大や処理量の増加に伴い，消費電力も増大しており，電力を削減することが重要な課題となってきている．例えば，スーパーコンピュータシステムに対する性能ベンチマーク Linpack で 33.9PFlop/s を達成した Tianhe-2 では約 17.8MW，11PFlop/s 達成の京では約 12.7MW という莫大な電力を必要とする[1]．したがって，省電力化が非常に重要である．また, HPC システムの電力効率のランキングである Green 500 [2] によると，2015 年 6 月時点では，最高の電力効率は 7.032GFlops/W であり，この電力効率で 1EFlops を達成しようとすると 142MW が必要となってしまう．このような膨大な消費電力は現実的ではないため，今後計算速度を向

†1 (株) 富士通研究所
Fujitsu Laboratories Ltd.

上させていくためにも，省電力化は非常に重要である．

計算機システムの省電力化に関しては，これまではハードウェア手法による省電力技術や，計算ノード単位や装置単位の大粒度での監視制御技術の研究開発が進んできた．しかし，ソフトウェアの挙動と電力消費の関係解明や省電力観点でのプログラム最適化の研究は，ハードウェアによる省電力技術の進歩と比べると，まだ十分でなく，発展の余地が残されている．省電力観点でのプログラム最適化を実現するためには，まずプログラムの消費電力のホットスポットを明らかにする必要がある．そこで，我々はプログラムの電力最適化を行うことを目標とし，関数単位での電力ホットスポットを明らかにする消費電力プロファイル分析について研究を行っている．これまでに，電力データ採取のための専用計測器やソフトウェアの改変を不要とする電力データ採取方法として，タイムベースサンプリングと同時にCPUが備える消費電力監視機能RAPLの消費電力値を採取する方法を提案した [8]．しかし，タイムベースサンプリングでは時間がかかる箇所がサンプリングされ，必ずしも電力消費が多い箇所をサンプリングできるとは限らない．本稿では，電力消費に関係するCPUの性能イベントをベースとしたサンプリングを行うことにより，従来のタイムベースサンプリングよりも精度の高い電力サンプリングを可能とする手法を提案する．提案手法を検証するため，電力消費に関係する4つの性能イベントをベースとしたサンプリングを個別に行い，採取した各サンプルデータを合成して複数イベントサンプリングを模擬した．電力計測を行った6つのアプリケーションのうち5つで誤差が1%以下に収まり，一定の消費電力量によるサンプリングが可能であることを確認した．

本稿では，2章でタイムベースサンプリングと同時にCPUが備える消費電力監視機能RAPL(Running Average Power Limit)の消費電力値を採取する従来手法について述べる．3章では電力消費に関係するCPUの性能イベントをベースとしたサンプリングによる電力計測手法について述べ，4章ではイベントベースとタイムベースでサンプリングする電力量を比較する．5章では関連研究，6章でまとめと今後について述べる．

2. タイムベースサンプリング

2.1 RAPLによる電力計測

RAPLは，ソフトウェアから消費電力を監視制御機能するための機能で，Sandy Bridge以降のIntel CPUに搭載されている．本機能により，CPUやメモリの消費電力を監視し，予め設定した電力に達した場合に消費電力を制限することができる [3]．

RAPLにより， CPU全体（package）やCPU内のコア全体（Core）の消費電力を監視できる．さらに，サーバ機ではメモリ（DRAM），クライアント機では，グラフィクスの消費電力が監視対象となる．これらの監視単位毎に，電力制御や消費電力記録などのためのレジスタが用意されており，消費電力値（Joule：以降Jと表記）は約1ms毎に更新される．

電力プロファイルの実現において，RAPLには十分な精度と利便性があると考えている．RAPLで得られる消費電力は計算値である[4]が，RAPLと外部電力計の消費電力傾向は非常に良く一致するという報告がある[5]．これは，電力プロファイルで消費電力のホットスポットを発見するために，消費電力の傾向を捉える用途には十分な精度である．また，専用ハードウェアなしに，CPUに組み込まれた機能を利用して，電力情報を採取可能であるという利点もある．

RAPLはあくまで積算電力のカウンタであり，これと走行するプログラムの電力とを関連付けるための機能はない．したがって，たとえば測定対象のプログラム内部にRAPLによって計測される電力を記録する仕組みを埋め込む機構や，定期的にRAPLの電力を計測し，その瞬間に動作していたプログラムと関連付ける機構のような仕組みが必要である．また，RAPLで採取できるのはCPU全体の消費電力であり，CPU内の各コアが消費した電力はわからない．CPU内の各コア上で別々のプログラムが走行する場合，個々のプログラムの消費電力を把握することができない．

2.2 タイムベースサンプリングによる電力計測

RAPLでは関数レベルの電力プロファイルの実現が難しいという課題に対し，図1のような電力サンプリング機構による電力データ採取と，CPU単位の電力値をCPU内の各コアに分配する手法を提案した．

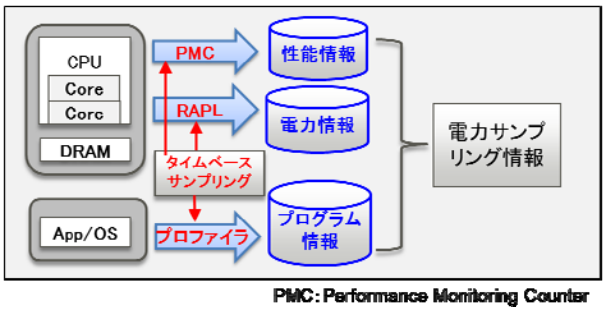

図1　タイムベースの電力サンプリング機構
Figure 1　Time-based sampling mechanism for power.

RAPLでは積算電力ベースのサンプリングが実現できないため，プロファイラで一定間隔毎にタイムベースサンプリングを行う際に，RAPL機能を利用して消費電力量を採取することにより電力サンプリングを実現する．このとき，コアの性能情報も採取する．この性能情報をもとにコアに消費電力を分配する．さらに，各サンプルデータの動作プログラム情報を利用して，コアに分配した消費電力を関数

に分配する．各サンプルデータには，通常のプロファイラと同様に動作プロセスや関数を特定する情報が採取される．

図 2 を用いて，CPU 全体電力の各コアへの分配方法を説明する．サンプリング時に，Core と DRAM の電力を分配するための性能情報（電力指標値）を採取しておく．各コアの電力指標値の割合に応じて，RAPL で得られた CPU 全体で消費した電力を各コアに分配する．図 2 では RAPL によりコア全体の電力 20J とメモリ全体の電力 10J が得られている．CPU に搭載されたコア 0 とコア 1 にこれらの電力を分配する．コアの電力指標値はコア 0 が 70，コア 1 は 30 であり，コア全体電力 20J を 7 対 3 の割合で分配すると，コア 0 は 14J，コア 1 は 6J となる．同様に，メモリの電力指標値はコア 0 が 20，コア 1 は 80 であり，メモリ全体の電力 10J を 2 対 8 の割合で分配する．このように，CPU 全体の消費電力を各コアに分配することにより，各コアで動作したアプリケーションの消費電力が把握可能となる．

なお，本プロファイラでは時刻に相当するイベント CPU_CYCLES を使用して一定時間間隔のサンプリングを実現している．従って，イベントベースサンプリングの一種であるが，他のイベントベースサンプリングと区別するために，タイムベースサンプリングと記述する．

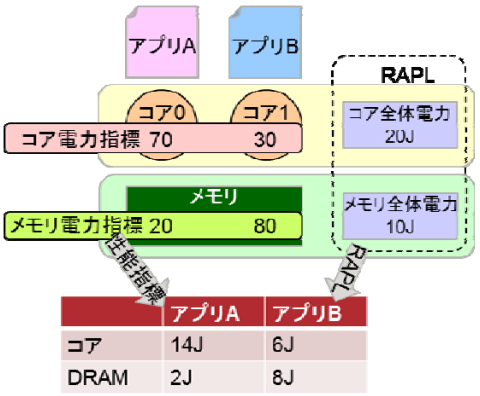

図 2　コアへの電力分配

Figure 2　Distribution power consumption to each core.

2.3 タイムベースサンプリングの課題

タイムベースサンプリングでは一定時間間隔毎にサンプリングを行うため，時間がかかる部分が多くサンプリングされる傾向がある．時間はかからないが，電力は多く消費するような部分があった場合，少なくサンプリングされることになる．このような部分は実際よりも消費電力が少なく見えてしまうという問題が発生する．

図 3 を用いて，タイムベースサンプリングと積算電力ベースサンプリングの違いを説明する．図 3 の 2 つのグラフは横軸が経過時間，縦軸が消費電力量であり，各時間にどの関数がどのくらい電力を消費したかを表すものである．ここでは関数 A（水色）と B（ピンク色）がそれぞれ 12J ずつ電力を消費しているものとする．この状況で，タイムベース（左のグラフ）と積算電力ベース（右のグラフ）でどうサンプリングされるかをみてみる．タイムベースでは一定時間毎に割込みを発生し，その時に動作しているプログラムを特定するための情報を採取する．従って，時間がかかるプログラムが採取されやすい．一方，積算電力ベースでは一定電力毎に割込みを発生させるため，消費電力量が多いプログラムが採取されやすい．タイムベースでは関数 A が 3 回，関数 B が 1 回サンプリングされ，消費電力量は関数 A がそれぞれ 7J，8J，3J で合計 18J，関数 B は 6J である．一方，積算電力ベースでは 6J 毎にサンプリングを行い，関数 A と B がそれぞれ 2 回サンプリングされ，消費電力量も 12J ずつとなる．この例では，積算電力ベースでは正しく消費電力をサンプリングできるが，タイムベースでは正しく消費電力をサンプリングできていない．

このように，タイムベースサンプリングによる電力計測では，サンプリングされる電力量は一定とは限らず，必ずしも電力消費が多い箇所をサンプリングできるとは限らない．そのため，より正確に電力を計測できるサンプリング方式が必要である．

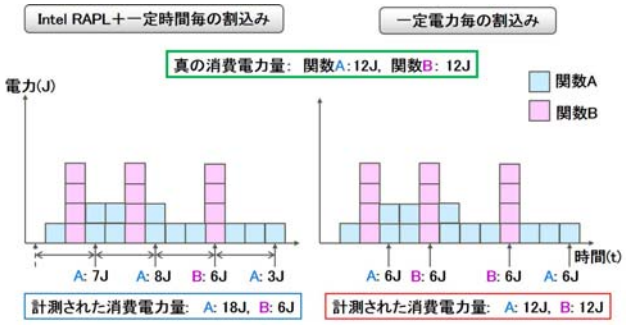

図 3　サンプリングの比較

Figure 3　Time-based sampling vs power-based sampling.

3. 電力モデルベースサンプリング

タイムベースサンプリングでは正確な電力サンプリングを行えないという課題に対して，時間ではなく，電力消費に関係する CPU の性能イベントをベースとしたサンプリングを行う手法を提案する．これにより，従来のタイムベースサンプリングよりも精度の高い電力サンプリングを可能とすることを目指す．

サンプリングのベースとするイベントとしては，CPU が備える PMU(Performance Monitoring Unit)という性能監視機構で測定監視できる CPU の性能(Performance Monitoring Counter: PMC)イベントを利用する．具体的には，3.2 で述べる消費電力モデルで使用するイベントをベースにしたサンプリングを行う．

3.1 イベントベースサンプリング

ここでは，CPU の PMU を利用したイベントベースサンプリング[11]の実現方法について説明する．PMU にはイベント設定レジスタとイベントカウンタレジスタがある．カウンタレジスタのカウンタオーバーフロー割込みをサンプリング契機に利用して行うサンプリングをイベントベースサンプリングと呼ぶ．これにより，設定イベントの発生回数が多いプログラムを把握できる．PMU でカウントできる値としては，実行命令数やキャッシュミス数などの性能に関係するイベントの発生回数がある．

図 4 に，PMU を利用したイベントベースサンプリング機構のイメージを示す．まず，PMU のイベント設定レジスタに使用イベント種と割込み発生の有効･無効の設定を行う．サンプリングベースにしたいイベントの場合は割込み有効と設定する．そして，カウンタレジスタには初期値を設定する．次に，サンプリング測定を開始し，対象アプリケーションを実行する．測定中，割込み発生有効に設定されたイベントに対し，カウンタレジスタに設定された回数 N のイベントが発生すると，PMU から割込みが発生する．N をサンプリング間隔と呼ぶ．割込みを受けたサンプリングドライバにおいて，その時動作していたプログラムの情報を採取する．最後に，測定終了後，動作プログラムのオブジェクトファイルやプロセス情報を収集する．収集したオブジェクトファイルやプロセス情報と，サンプルデータとして採取したプログラム情報（プロセスＩＤや命令アドレスなど）を突き合わせて，動作していたプロセスや関数を特定する．プロセスや関数毎にイベント発生回数を集計し，プロファイル結果をまとめる．この集計結果から対象イベント発生のホットスポットを見つけることができる．プロファイルでは，大量の一定回数毎のサンプリングデータを解析に用いることにより，解析結果の妥当性が統計的に担保される．また，サンプリング方式により，低オーバーヘッドに抑えることができる．

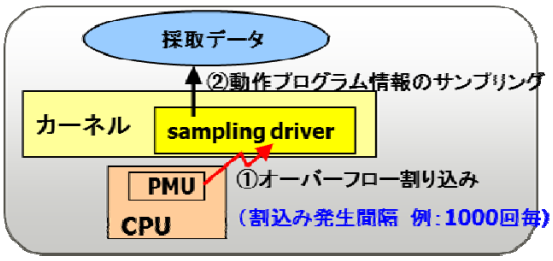

図 4　イベントベースサンプリング機構
Figure 4　Event-based sampling mechanism.

3.2 消費電力モデル

サンプリングのベースとなるイベントを決定するために，消費電力量と相関の高いイベントや CPU 情報をベースに，消費電力を表すモデルを作成する．まず，いくつかのアプリケーション実行時に消費電力量，PMC イベント値，CPU 情報を採取する．採取したデータから各消費電力値を目的変数，消費電力と相関が高い PMC イベント値やコア情報を説明変数として，重回帰分析により消費電力モデルを作成する．一度に採取可能な PMC イベント数には制限があり，例えば Sandy Bridge や Haswell では 4 つである．従って，電力測定を 1 度で済ませるためには，モデルに使用するイベント数を 4 つ以下に絞り込むことが必要である．

表 1　実験環境
Table 1　Experimental environment.

CPU	Sandy Bridge (XeonE5)
CPU 周波数	2.90GHz
CPU ソケット数	2
コア数／ソケット	8
キャッシュサイズ	L1d 32K, L1i 32K, L2 256K, L3(LLC) 20MB

表 2　データ採取パターン
Table 2　Target Patterns.

状態	プログラム	備考
アイドル状態	Sleep コマンド	
CPU 負荷	Stress コマンド	--cpu オプション（sqrt 関数を実行）
	自作プログラム	命令パイプラインが埋まるようアセンブリ記述
メモリ負荷	Stream ベンチマーク（使用データサイズは右の 4 パターン）	L1：12K
		L2：144K
		L3 (LLC)：12M
		メモリ：240M

今回，表 1 の実験環境で表 2 のようなデータを採取して消費電力モデルの作成を試みた．アイドル状態，CPU 負荷をかけるアプリケーション，メモリ負荷をかけるアプリケーション実行の消費電力と PMC イベント値，CPU 情報を採取した．CPU 負荷アプリとしては stress コマンドと自作プログラム，メモリ負荷アプリとしては stream ベンチマークを使用した．stress コマンドでは，CPU 負荷をかける worker を 1 つ動作させた．具体的には平方根を求める関数 sqrt を実行する．Stream ベンチマークはアプリケーションで使用するメモリ量を変え，L1 キャッシュサイズ内，L2 キャッシュサイズ内，L3 キャッシュサイズ内，メモリを使用する 4 つのパターンで実行した．また，1 ソケット 8 コア構成で，1 コアから 8 コアまでアプリを実行するコア数を 1 つずつ増やして 8 パターンを採取した．なお，今回は

2 ソケットのうちの 1 ソケットのみを使用し，残りの 1 ソケットはほぼアイドル状態だった．また，周波数は固定している．

PMC イベント約 80 個，コアがアクティブに動作した割合を示す C0 など採取し，Core と DRAM の消費電力との相関を調べたところ，Core の消費電力と相関が高かったのは，C0 や Halt でない状態でのサイクル数 CPU_CLK_UNHALTED.THREAD_P などで相関係数は 0.90 である．DRAM の消費電力と最も相関が高かったのは last level cache に対するキャッシュミスをカウントするイベント LONGEST_LAT_CACHE.MISS であり，相関係数は 0.99 である．C0 は各コアの値（0～1）の合計値を，他のイベントは発生回数を使用した．

これらをベースに重回帰分析を行い，Core と DRAM のモデルを作成した．消費電力モデルの重決定係数 R2 はいずれも 0.99 である．重決定係数は，目的変数の変動のうち説明変数によって説明される割合を表すものであり，0≦R2≦1 の値を取り，1 に近いほどよい．従って，作成したモデルは消費電力をうまく表せていると言える．

今回は Core の消費電力のみを対象とするため，以下に今回採用した Core のモデルを示す．Core のモデルは，CPU の使用状況を表す C0 と実行命令数 INST_RETIRED.ANY_P（INST），リソースに関係したストールサイクル数 RESOURCE_STALLS.RS（RESOURCE），L2 キャッシュアクセスに関するイベント L2_TRANS.ALL_PF（L2）を使用することにした．

Core の消費電力[W]

$$12.41+6.288\times C0$$
$$+2.06\times10^{-11}\times INST\_RETIRED.ANY\_P$$
$$+3.161\times10^{-10}\times RESOURCE\_STALLS.RS$$
$$+6.058\times10^{-10}\times L2\_TRANS.ALL\_PF$$

3.3 イベントのサンプリング間隔

次に，消費電力モデルに使用するイベントに対するサンプリング間隔を決定する．

今回は，Core の消費電力を対象とし，積算電力サンプリングする電力量 P にモデル式の各イベント E_i の係数 Ci の逆数を掛けて求めた回数をサンプリング間隔 R_i とする．

$$R_i = P / C_i$$

関数プロファイルを目的とした電力サンプリングのために，1 J 毎にサンプリングすることを想定する．表 3 に各イベントのサンプリング間隔を示す．例えば，INST イベントでは，1 秒間で約 4,854,000,000 回に 1 回サンプリングすることになる．関数プロファイルを考えると 1 秒間では荒いので，10ms 間相当とし，48,540,000 回に 1 回サンプリングする．なお，モデル式の C0 はレジスタから計算することを想定していたが，Halt でない状態でのサイクル数 CPU_CLK_UNHALTED.THREAD_P（CLK）から計算することができるため，イベントとしてこれを使用する．モデル式の C0 は各コアの値の合計から求めているため，係数をコア数で割った値を使用する．

表 3　各イベントのサンプリング間隔
Table 3　Sampling interval of each event.

イベント	回数／秒	回数／10ms
INST	4,854,368,932	48,540,000
RESOURCE	316,355,584	3,160,000
L2	165,070,981	1,650,000
CLK	3,689,567,430	36,900,000

3.4 電力サンプリング間隔

各イベントに対して計算した回数毎にサンプリングを行い，サンプリング時の電力値を確認する．サンプリング時の電力値は，1 つ前のサンプルデータの RAPL 値と現サンプルデータの RAPL 値との差分とする．理想は，これが常に目的とする消費電力量になることである．

表 1 の実験環境で，消費電力モデルを作成する際にデータを採取したアプリケーションを実行し，各イベントベースでサンプリングを行った．

ソケット内の全 8 コアで同じアプリケーションを実行した．Stress コマンドは taskset コマンドで CPU（コア）を固定して 8 並列で実行し，他のアプリケーションは OpenMP により 8 並列で実行した．各アプリケーションは全コアでほぼ同じ動きをするため，1 コアのサンプルデータを抽出し，各サンプルデータの RAPL 値の差分を求め，この値の 8 分の 1 の値を Joule に換算したものを，Core 消費電力量とした．なお，OpenMP を利用した並列実行においてもコア間で完全に同期して実行が進むとは限らないし，各コアで異なる関数が実行される場合もある．そのような場合，CPU 単位の RAPL 値をコアに分配する必要がある．我々は，CPU の性能イベント情報を利用して RAPL 値をコアに分配する方法 [8] をすでに提案しており，このような方法を適用することが考えられる．本稿では，電力データの採取方法を焦点としており，採取した値の分配方法についてはこれ以上議論しない．

イベントでサンプリングした平均消費電力量は，表 4 のようになった．CLK イベントは比較的アプリケーションによる違いは大きくないが，他のイベントではアプリケーションによる消費電力量の違いが大きい．また，stress コマンドでは L2 イベントはサンプルデータが採取されず，RESOURCE イベントは 3 個しか採取されていない．アプリケーションによって採取されるイベントやイベントの発生

回数は異なる．いずれのイベントも今回の目標である消費電力量 1J にはなっていない．

表 4　各イベントの平均消費電力量（J）
Table 4　Average of power consumption each event.

	INST	RESOURCE	L2	CLK	10ms
cpu-stress	0.08	16.85	-	0.11	0.09
cpu-自作	0.12	0.03	1.94	0.17	0.14
stream-L1	0.48	0.14	0.65	0.11	0.09
stream-L2	0.19	0.13	0.63	0.11	0.09
stream-L3	0.12	0.07	0.06	0.16	0.13
stream-mem	0.55	0.06	0.27	0.13	0.10

タイムベースの例として, 10ms 間隔サンプリング時の平均消費電力量を一番右の欄に挙げている．CLK と同様に，比較的アプリケーションによる違いは大きくない．

１つのイベントでは，アプリケーションにより消費電力量が異なり，目的とする電力量をサンプリングすることができない．電力量を測定するアプリケーションが予め決まっており,そのアプリケーションしか動作しない場合には，サンプリング間隔を調整することにより，一定電力量でのサンプリングが可能かもしれない．しかし，アプリケーションが決まっていても，測定中に複数のアプリケーションが実行される場合には 1 つのイベントのみのサンプリングでは一定電力量によるサンプリングは難しい．

3.5 複数イベントによるサンプリング

前節で示したように，１つのイベントで目的とする電力量をサンプリングできるものがない．今回使用する消費電力モデルは 4 つのイベントの組み合わせで消費電力を表すものであり，サンプリングも複数イベントで行うことが理想的である．しかし，現在サンプリングに使用しているプロファイラでは複数イベントで同時にサンプリングできない．そこで，各イベントのサンプリング結果を合成することにより，複数イベントサンプリングを模擬してみる．

サンプルデータとして， CPU（コア）識別子，動作プロセス，実行アドレス，時刻情報，RAPL 値，各イベントの値を採取する．動作プロセスと実行アドレスから動作した関数，時刻情報から測定開始からの経過時間を求めることができる．

各イベントでサンプリングを行い，採取したサンプルデータをマージし，経過時間でソートする．経過時間順に，目的の消費電力量（1J）に近い値になるように複数データを集計して，１つの電力データを作成する．こうして，作成した電力データの電力値が目的の消費電力量に近ければ，正確にサンプリングできると考えられる．

複数イベントのサンプリング結果を合成して模擬した結果を表 5 にまとめる．基本は 4 イベントのサンプリング結果の合成だが，stress コマンドは L2 イベントや RESOURCE イベントの発生回数が少なく，使用できない．そこで，INST イベントと CLK イベントの 2 イベントの結果の合成と，2 イベントに RESOURCE イベントを加えた 3 イベントの結果の合成も行った．表 5 では，各アプリケーションに対して最も目的の電力量に近い値を色づけしている.アプリケーションにより最適なイベント数は異なるが，stream ベンチマークのメモリ使用以外は 1%以下の誤差に収まっている．この結果から，消費電力量の集計に 4 イベントすべてを使用することが最善とは限らないことがわかる．

表 5　複数イベントの平均消費電力量（J）
Table 5　Average of power consumption in multi events.

	4 イベント	3 イベント	2 イベント
cpu-stress	-	1.44	1.01
cpu-自作	1.07	1.02	1.00
stream-L1	1.01	1.00	
stream-L2	1.01	1.26	
stream-L3	1.01	1.00	
stream-mem	1.04	1.05	

4. 検証

4.1 平均消費電力量の比較

複数イベントのサンプリング結果を合成して模擬した結果とタイムベースサンプリングの集計結果を表 6 にまとめる．前節と同様に，タイムベースは 10ms 間隔でサンプリングを行った．タイムベースサンプリングの結果は，アプリケーション毎に平均消費電力量に近くなる個数を求め，その個数毎に集計した 10ms（最適）と，全アプリケーションで同じ個数（10 個）毎で集計した 10ms（10）の 2 つを示す．複数イベントサンプリングの結果は，各アプリケーションの最適値を使用している．

複数イベントサンプリングでは，stream ベンチマークのメモリ使用以外は 1%以下の誤差に収まっている．一方，10ms（最適）も stream ベンチマークの L1 と L3 使用以外は 1%程度の誤差に収まっている．しかし，10ms（10）では stream ベンチマークのメモリ使用以外は誤差が大きく，自作の CPU 負荷プログラムで 44%, stream ベンチマークの L3 使用で 31%となっている. この結果からタイムベースサンプリングでは，消費電力量が異なる，複数のアプリケーションを同時に測定することは難しいと思われる．

表 6 では，各アプリケーションにおいて最も目標とする電力量に近い値に色付けしており，複数イベントは stream ベンチマークのメモリ使用以外でタイムベースと同等あるいはそれ以上に平均的に電力量をサンプリングできている．

表 6　複数イベントとタイムベースの平均消費電力量（J）
Table 6　Average of power consumption (time and multi events)

	複数イベント	10ms (最適)	10ms (10)
cpu-stress	1.01	0.99	0.90
cpu-自作	1.00	1.01	1.44
stream-L1	1.00	0.96	0.88
stream-L2	1.01	1.01	0.92
stream-L3	1.00	1.05	1.31
stream-mem	1.04	1.00	1.00

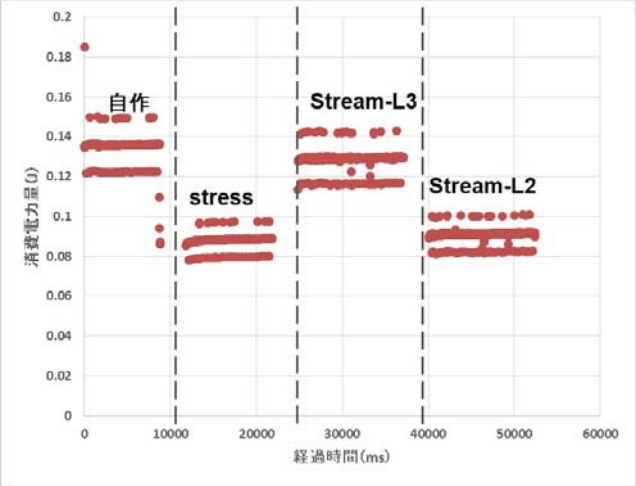

図 5　タイムベースの消費電力量の分布
Figure 5　Map of power consumption in time-based sampling.

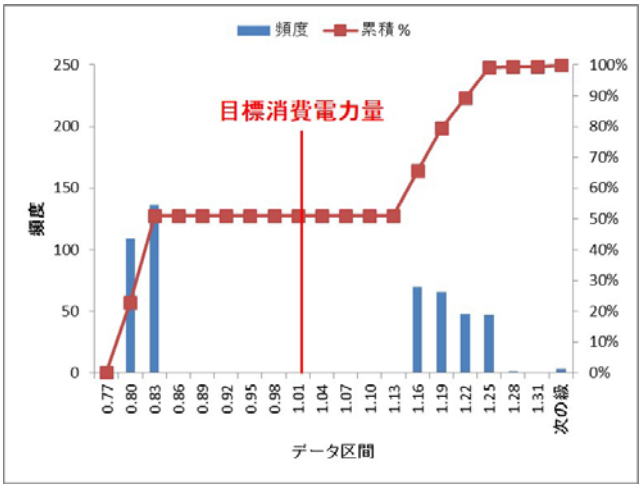

図 6　タイムベースの消費電力量ヒストグラム
Figure 6　Histogram of power consumption in time-based sampling.

次に，複数のアプリケーションを連続実行し，消費電力量が一定になるかを確認する．各アプリケーション実行の間には 3 秒間の sleep を入れ，CPU 負荷をかける自作プログラム，stress コマンド，stream ベンチマークの L3 使用，L2 使用を順番に実行した．アプリケーションの実行時間は自作プログラムが 9 秒，stress コマンドが 10 秒，L3 使用が 12 秒，L2 使用が 12 秒程度である．

図 5 にアプリケーション実行時に 10ms 間隔のタイムベースサンプリングにより採取したサンプルデータの消費電力量（J）の分布を示す．横軸は経過時間（ms），縦軸は Core 消費電力量（J）であり，左側から 4 つのアプリケーションが順番に実行されている．４つのアプリケーションは微妙に消費電力量が異なり，それぞれ 3 つの帯になっている．

このデータを 9 個ずつ集計し，電力データを作成した結果の消費電力量のヒストグラムを図 6 に示す．0.83J までで約 51%，1.16～1.25 に約 49%が分布している．平均値は約 1.00J であり，ほぼ目標の 1 J だが，１J 付近でサンプリングできないことがわかる．

同様に，データを 8 個ずつ集計した場合の消費電力量の分布は 0.73J まで約 51%，1.02～1.12J に残り 49%が分布しており，平均値は 0.89J である．データを 10 個ずつ集計すると，0.91～0.97J までに 51%，1.29～1.39J に残り 49%が分布し，平均値は 1.11J となる．従って，いずれも目標の消費電力量毎にサンプリングすることはできていない．

同様に，同じアプリケーション列の実行時に，4 つのイベントベースサンプリングを行った．各イベントベースサンプリングにより採取したサンプルデータの消費電力量（J）の分布を図 7 に示す．グラフは図 5 と同じく，横軸は経過時間（ms），縦軸は Core 消費電力量（J）である．上から CLK イベント, INST イベント, L2 イベント, RESOURCE イベントの結果である. CLK イベントはおおよそ 0.1～0.18 J の間，INST イベントは 0.07 ～0.2J の間であり，4 つのアプリケーションに大きな違いはない．しかし，RESOURCE イベントは stress コマンド実行の消費電力量が約 3.5J と極端に多い．これはサンプル数が少なく，1 サンプルあたりの消費電力量が大きくなっているためである. L2 イベントでは stress コマンド実行時にサンプルは採取されていない．そこで，今回は CLK イベントと INST イベントの 2 イベントの結果を合成する．合成したサンプルデータから作成した電力データの消費電力量のヒストグラムを図 8 に示す．0.96～1.08J の間に約 84%が分布しており，平均値は 1.00J である．従って，10ms 間隔のタイムベースサンプリングよりも CLK イベントと INST イベントをベースにしたサンプリング結果を合成した場合のほうが，より正確に一定の消費電力量でサンプリングできていると言える．我々は許容誤差・目標誤差を（経験的に）10%と設定しているが，タイムベース（図 6）は 0.9～1.1 の範囲で全くサンプリングできていないので，正しい電力ベースポイントの代替にはなっていない．それに対して，イベントベース（図 8）は目標範囲に 100%入っており，正しく電力ベースを実現できているとみなす．また，許容誤差を 5%としても，0.95~1.05 に約 76%が入る．

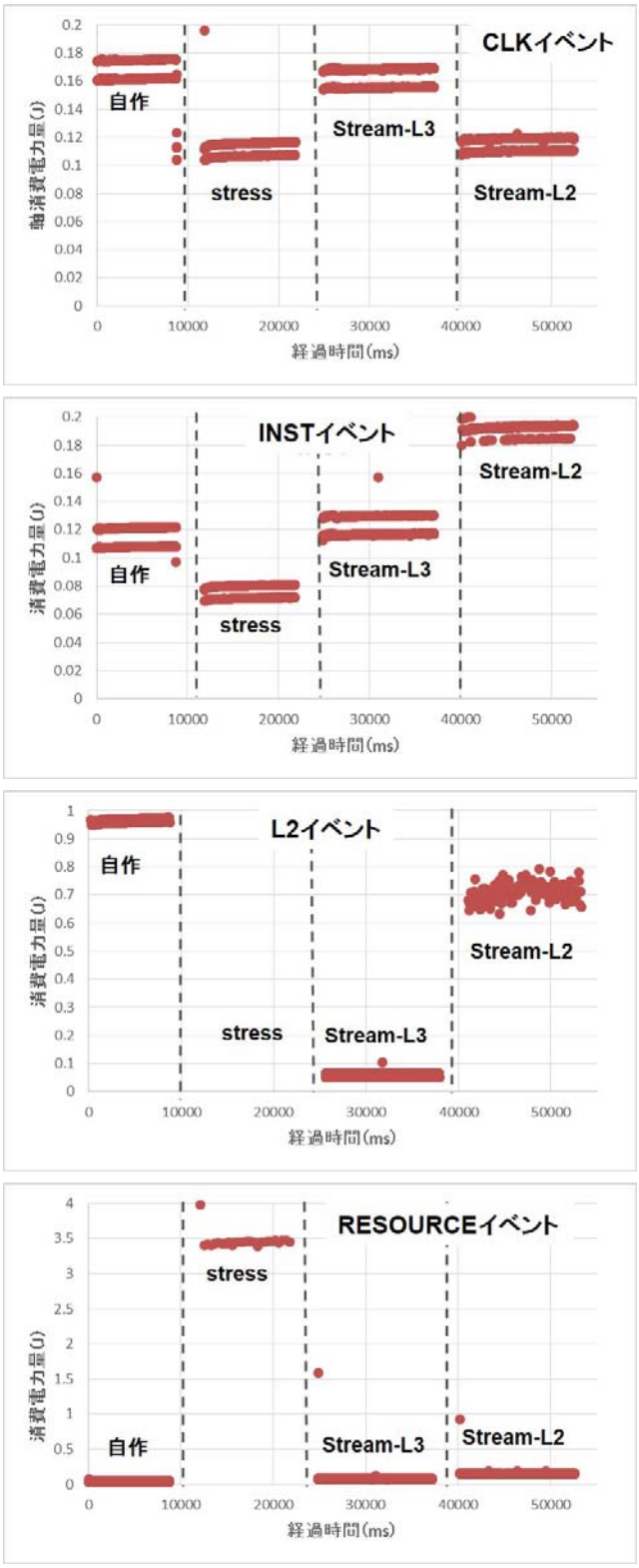

図 7　4 イベントの消費電力量の分布

Figure 7　Map of power consumption in event-based sampling.

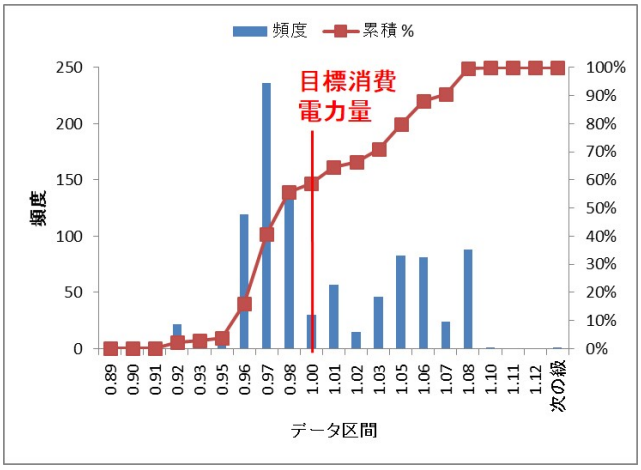

図 8　2 イベントの消費電力量ヒストグラム

Figure 8　Histogram of power consumption in two event sampling.

4.2 複数イベントサンプリングのオーバーヘッド

今回使用したプロファイラでは複数イベントサンプリングを行うインタフェースが用意されていないため，1 イベントずつサンプリングし，その結果を合成することにより複数イベントサンプリングを模擬した．複数イベントサンプリングを実際に行う場合のオーバーヘッドを検証するために，今回検証するイベントのみを扱えるようにプロファイラを修正した．この修正版を使用し，前節 4.1 で検証した 4 つのアプリケーションについて，タイムベースサンプリングと複数イベントサンプリング実行時の時間と Core 消費電力量を測定し，比較した．

まず，正しく複数イベントサンプリングを実行できることを確認するために，採取したサンプル数を調べる．表 7 は各イベントサンプリングで採取されたサンプル数をまとめたものであり，1 行目の数字は同時に割込みを発生させるイベントの数である．イベント 10ms は 10ms 間隔でタイムベースサンプリングした場合のサンプル数を示す．

INST イベントと CLK イベントの 2 イベントを同時にサンプリングした場合のサンプル数の合計は，2 つのイベントを別々にサンプリングした場合の和とほぼ等しい．同様に，2 イベントに RESOURCE イベントを追加した 3 イベントと，L2 イベントを追加した 4 イベントのサンプル数の各合計はそれぞれ 1 イベントサンプリング時の採取サンプル数の和とほぼ等しい．従って，正しく複数イベントサンプリングを実装できていると言える．なお，複数イベントの割込みが同時に発生した数は数十個程度であった．

サンプリングを行った場合のアプリケーションの実行時間（秒）を図 9 に，アプリケーション実行による Core 消費電力量の増加量（J）を図 10 にまとめる．サンプリングそのもののオーバーヘッドも見るために，サンプリングを行わずにアプリケーションを実行した場合（実行）についても測定を行った．Stress コマンドは指定時間で測定を終了

するため，実行時間は常に一定であり，実行時間については割愛する．タイムベースサンプリングと比較して，複数イベントサンプリングは実行時間で最大 1%程度の増加，Core 消費電力はほとんど変わらない．また，サンプリングを行わない実行と比較しても，実行時間と Core 消費電力いずれも最大 1%程度の増加に抑えられている．同時に割込みを発生させるイベント数による違いもほとんどない．従って，複数イベントサンプリングによるオーバーヘッドは問題にならないと考えられる．

表 7　サンプル数
Table 7　Number of samples.

	1	2	3	4
10ms	34821			
INST	32839	32829	32849	32856
CLK	27277	27290	27355	27378
RESOURCE	57324		57254	57198
L2	24658			23916
合計		60119	117458	141348

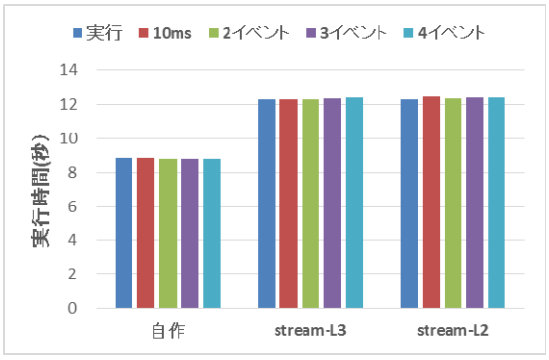

図 9　実行時間（秒）
Figure 9　Execution time.

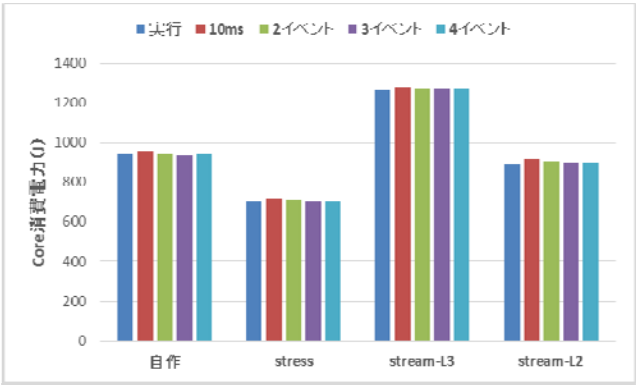

図 10　Core 消費電力量（J）
Figure 10　Core Power consumption.

5. 関連研究

これまで，専用ハードウェアによる電力ベースサンプリングシステム[9][10]が提案されてきた．[9] は PC サーバの消費電力を累積カウンタにより計測する装置及び制御ソフトウェアで構成され，OS やアプリの関数単位の電力消費を時系列で高精度に分析可能である．しかし，専用ハードを用いずに電力ベースサンプリングを行う研究は報告されていない．

電力制約適応型システムの実現のため，性能解析ツール TAU[6]でアプリケーションの性能情報を，RAPL で消費電力情報を採取し，2 つの情報を統合する手法[7]が提案されている．TAU から PAPI(Performance Application Programming Interface)と連携させることで RAPL の電力情報も同時に取得可能だが，実行時間のオーバーヘッドとそれによって生じる消費電力の誤差が大きいため，別々に採取して統合する方法を採用している．TAU からの電力測定はアプリケーションに計測の開始と終了のための関数を追加することが必要となる．また，TAU の計測間隔は秒単位であり，関数プロファイルとしては荒い．この手法では，RAPL の値を一定時間毎にサンプリングしており，イベントベースサンプリングは行われていない．

提案手法では，専用ハードウェアを使用せずに，CPU の RAPL 機能を用いることにより，電力値を獲得する．電力値の計測にはタイムベースではなく，電力消費に関係する CPU の性能イベントをベースとしたサンプリングを行う．こうすることにより，時間がかかる箇所を採取するのに適したタイムベースサンプリングと比べて，より正確な消費電力量のサンプリングを可能にする．

6. おわりに

本稿では，電力消費に関係する CPU の性能イベントをベースとしたサンプリングを行うことにより，従来のタイムベースサンプリングよりも精度の高い電力サンプリングを可能とする手法を提案した．

電力消費に関係する 4 つの性能イベントをベースとしたサンプリングを個別に行い，採取した各サンプルデータを合成して複数イベントサンプリングを模擬した．電力計測を行った 6 つのアプリケーションのうち 5 つで誤差が 1%以下に収まり，一定の消費電力量によるサンプリングが可能であることを確認した．また，消費電力量が異なる複数のアプリケーションを連続実行した場合，タイムベースサンプリングでは，サンプリングされる消費電力量が一定ではなくなる．これに対して，複数イベントサンプリングでは，一定の消費電力量でのサンプリングが可能であることを示した．

今後の課題として，より多くのアプリケーションを用いた本手法の検証，さらに本手法の省電力プログラム開発手法への適用がある．

参考文献

[1] Top500 : http://www.top500.org/
[2] Green500 : http://www.green500.org/
[3] Intel: CHAPTER 14.9 PLATFORM SPECIFIC POWER MANAGEMENT SUPPORT, Intel Software Developer's Manual, Volume 3B, (April 2015)
[4] Srinivas Pandruvada : Running Average Power Limit, https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl
[5] 近藤正章 他：ミニアプリを用いた HPC システムの電力解析，SDHPC10「ミニアプリセッション」(2013)
[6] Sameer S. Shende, Allen D. Malony : THE TAU PARALLEL PERFORMANCE SYSTEM, International Journal of High Performance Computing Applications (Summer 2006)
[7] 大坂隼平 他 : HPC アプリケーションの消費電力最適化に向けた性能・消費電力情報の統合手法,HPCS2015
[8] 小野美由紀 他：ソフトウェアの消費電力分析手法，情報処理学会研究報告，2015-HPC-150 N0.29
[9] 平井 聡 他 : 電力ベースサンプリングシステム PARITS の提案，情処第 72 回全国大会
[10] 三輪 真弘 他：電力ベースサンプリングシステム PARITS の評価，情処第 72 回全国大会
[11] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel: Establishing a Base of Trust with Performance Counters for Enterprise Workloads, 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 541-548, (2015).