システムソフトウェアと
オペレーティング・システム 77－24
マルチメディア通信と分散処理 87－24
（1998. 2. 27）

アプリケーションプログラム参加の省電力制御に必要な機能の考察

古市 実裕　相原 達
日本アイ・ビー・エム (株) 東京基礎研究所

ノートブック型パソコンでは，長時間バッテリ駆動させるために，消費電力を抑える技術が用いられる．しかし，OS や BIOS などのシステムソフトウェアが省電力制御の主導権を持っているため，作業内容の変化に応じて柔軟に制御方法を切替えることが困難であり，時には極端な性能の低下や，不必要な電力消費を招く．本稿では，アプリケーションプログラムが電力管理へ参加した場合，従来手法よりきめ細かく効果的な省電力制御が実現できることを実験により示し，そのために OS やアプリケーションプログラムに求められる機能やその実装方法について考察する．

Application-Oriented Power Management

Sanehiro Furuichi　Toru Aihara
IBM Research, Tokyo Research Laboratory, IBM Japan, Ltd.

Various technologies for reducing the power consumption of notebook computers have been developed to extend the battery life. However, the current power management is controlled by system software such as the operating system (OS) and the basic I/O system (BIOS), and therefore it not only cannot switch the control method dynamically in accordance with the workflow, but also causes many unexpected problems such as extreme performance degradation power wastage. The experimental results reported in this paper show that application programs are capable of more granular and more effective power management, and suggest new functions for current OSs and application programs.

1 はじめに

携帯用のノートブック型パソコンは，限られたバッテリ容量で長時間動作させる必要があるため，電力の消費量をできるだけ抑える必要がある．その一方で，近年は，ノートブック型パソコンにも，デスクトップ型と同等の機能を要求する声が高まり，デバイスの低消費電力技術の進歩にもかかわらず，システム全体の電力消費量はますます増大する傾向にある [1]．高度な機能を維持しながら，消費電力を抑制できる技術が強く求められている．

現在，多くのノートブック型パソコンに，様々な省電力化技術が実装されている [2][3]．しかし，処理する仕事の内容や性質によらず，常に画一的な制御をするため，必要な時に性能が落ちたり，不必要な時に電力を大きく浪費するなどの問題点が多い．

本稿では，具体的な実験により，アプリケーションプログラムが省電力制御へ参加できれば，作業内容に応じて動的に制御方針を切り替えることが可能となり，従来手法以上の省電力効果が期待できることを示す．その上で，アプリケーション主導の省電力制御を実現するために，OS やアプリケーションに求められる機能やその実装方法について考察する．

2 既存の省電力制御の問題点

ノートパソコンの BIOS は，ROM や NVRAM に記録された設定に基づき，機械的にデバイスの省電力制御をする．例えば，ハードディスクドライブ (HDD)，CD-ROM ドライブ，液晶ディスプレイ

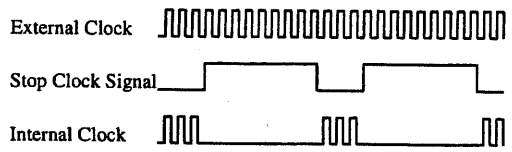


図 1: スロークロックエミュレーションによる，CPU 速度の調整．図は 1/4 クロックスピードの模式図．

(LCD) パネルなどは，一定時間未使用の場合にドライブモータを停止したり，バックライトを消すことができる．また，Intel Pentium プロセッサなどは，図 1に示すように，周期的に CPU の内部クロックを停止させ，見かけ上のクロック周波数を下げるスロークロックエミュレーションにより，CPU の電力を抑制できる [4]．アプリケーションプログラムのレベルでどんな作業が行なわれているか関知しないので，作業内容によっては，ユーザがその都度設定を変更しない限り，パフォーマンスやユーザビリティの悪化につながる場合がある．

一方，APM は，OS 主導の省電力制御である．OS 内のスケジューラが CPU アイドルを検出すると，APM BIOS インタフェース [5] を介してアイドル状態を APM BIOS に通知し，APM BIOS は CPU のクロックを停止する．ただし，スケジューラのレベルでのアイドル状態しか検出できないので，タスクの中に細かいアイドル状態が散在し，必ずしも CPU を最高速度で稼働させる必要がない場合でも，一律に最高速度で稼働させる．

近い将来，BIOS や x86 CPU 依存からの脱却を目指して，OS が電力管理のすべての制御権を持つ ACPI [6] に移行する．機種に依存しない統一的な制御が実現できるが，CPU のアイドル状態の検出精度は APM 同様限界がある．また，パワーフレンドリーなアプリケーションプログラムを実現するために，いくつかの API [7] が用意されているが，OS が制御方針を決定する際の補助的な情報を提供する程度で，積極的にデバイスの電力制御をすることはできない．

3 アプリケーション主導型制御

3.1 デバイス制御の有効性

IBM ThinkPad 760ED において，代表的なノートブック型パソコン用ベンチマークである，ZD Battery Mark 1.0 [8] を実行した時の，全体の消費電力の内訳を図 2に示す．また，ThinkPad 760ED の仕様を，表 1に記す．

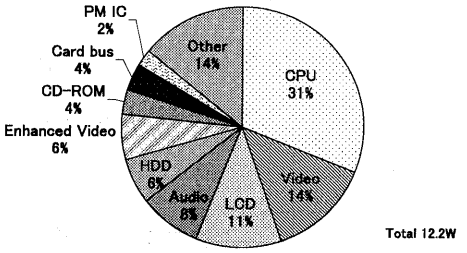


図 2: IBM ThinkPad 760ED において ZD Battery Mark 1.0 を実行した時の消費電力の内訳．

表 1: IBM ThinkPad 760ED の仕様．

CPU	Intel Pentium 133MHz
キャッシュメモリ	256KB 外部
主記憶	16MB(EDO)
VRAM	2MB
HDD	1.2GB
ディスプレイ	12.1 インチ TFT 液晶
拡張ビデオ	MPEG2

CPU は平均で約 4W 消費しているが，ベンチマーク全体の 7 割がアイドル状態なので，実際の稼働時は 6W 近く消費する．クロック周波数を下げると電力を約 2.5W まで抑えられるが，パフォーマンスの悪化につながるので，兼ね合いが難しい．

LCD の電力は，以前に比べて格段に低くなったが，依然全体に占める比率は無視できない．輝度を変えることで，電力を 4W から 1W 近くまで幅広く調節できるが，画面が暗いとユーザの操作性が悪くなる．

HDD や CD-ROM，オーディオなどは，未使用時にきめ細かく停止させると，電力を節約できる．しかし，停止中にアクセスすると，遅延が生じパフォーマンスが悪くなる．

これらのデバイスは，設定を変更することで消費電力を下げることができるが，すべてのアプリケーションに共通して使える効果的な省電力設定は存在しない．作業内容によって，制御対象デバイスや

設定パラメータを切替えないと，パフォーマンスやユーザビリティの悪化につながる．

3.2 アプリケーション主導型制御の利点

アプリケーションプログラムは，作業内容に適したCPU速度や，必要なI/Oデバイス，対話的操作の有無など，自分自身の稼働状況や使用している資源内容を最も詳細に把握している部分であり，将来の動向をある程度予測することも可能である．OSやBIOSでは知ることのできない情報を，タイムリーに省電力制御に反映させれば，作業の進捗状況に合わせてきめ細かな動的制御が実現できる．アプリケーションを省電力制御に参加させるために，OSとアプリケーションの間に省電力制御に関する強力なAPIが要求される．

3.3 省電力効果を評価する実験

アプリケーションプログラムが省電力制御に参加した場合，どの程度省電力効果が得られるかを評価する実験を行なった．

3.3.1 実験方法

市販のアプリケーションプログラムを実行し，タスクの性質が切り替わるごとに，個々のデバイスの設定を変更する．その間の消費電力を測定し，従来の固定的な省電力制御方法と比較する．

ただし，市販のアプリケーションからデバイスの設定を切替えることはできないので，実験では負荷の軽い別プログラムから仮想的に設定を変更した．

測定機種としてIBM ThinkPad 760ED，OSにはMicrosoft Windows 95を用いた．実行するアプリケーションプログラムには，プログラム開発環境のMicrosoft Visual C++ 4.0 Developer Studioを用いた．

アプリケーションでの作業内容は，

1. 起動後，10秒間放置．
2. 30秒間プログラム編集．
3. ファイル保存後，10秒間放置．
4. その後，コンパイル．

というシナリオに従う．

各作業内容でのデバイスの設定パラメータを表2に示す．特に明記していない場合はディフォルト設定を用いる．各タスクの特徴は以下の通りである．

- ディフォルト…作業の進行過程で適宜パフォーマンスを切替えるので，ディフォルトは最低レベルのパフォーマンスで構わない．CPU速度，LCD輝度とも最低レベルまで下げる．サスペンドタイマ，HDD停止タイマも短くする．
- プログラム編集…軽負荷なのでCPU速度は最低でも充分である．スペルチェックなどの機能を使うときのみ一時的にCPU速度を上げる．ユーザインタラクションが主要な作業要素なのでLCD輝度は上げる．
- コンパイル…同一仕事量の場合，短時間に処理を済ませる方が，全体の電力消費量が少ないので，CPU速度は最高にする．ユーザインタラクションの必要がないので，LCDバックライトを消灯する．サスペンド，HDD停止機能は使わない．

一方，比較の対象として，2種類の従来手法を用意する．

1. パフォーマンス重視…CPU速度を最高速度に固定．HDD停止機能は用いない．
2. 省電力重視…CPU速度を最低速度に固定．サスペンド，HDD停止機能を利用．タイマ設定値は出荷時ディフォルト設定．

3.3.2 実験結果

図3には，動的にデバイス設定を切替えた場合の消費電力，図4には，デバイス設定を固定したままの従来手法1と2の消費電力の推移を示す．各タスクにおける平均消費電力を表3に示す．提案手法は，従来手法と比較して，以下のような効果が得られる．

1. 従来手法1との比較
   ディフォルト…LCD輝度が低い分，低電力．
   編集…CPUアイドル状態が多く，差はない．
   コンパイル…LCD消灯の効果の分、低電力．
2. 従来手法2との比較
   ディフォルト…LCD輝度が低い分，低電力
   編集…電力に差はない．
   コンパイル…LCD消灯の効果をCPU高速化により相殺するが，処理時間短縮により消費電力量は少ない．

表 2: Visual C++ 4.0 を実行中の各作業内容におけるデバイスの設定パラメータ.

	CPU 速度	LCD 輝度	サスペンドタイマ	HDD 停止タイマ
起動後ディフォルト	最低速度 (12.5%)	10%	20 秒	20 秒
プログラム編集	最低速度 (12.5%)	50%	20 秒	60 秒
コンパイル	最高速度 (100%)	0(消灯)	0 (常時動作)	0 (常時動作)
従来手法 1	最高速度 (100%)	50%	300 秒	0 (常時動作)
従来手法 2	最低速度 (12.5%)	50%	300 秒	180 秒

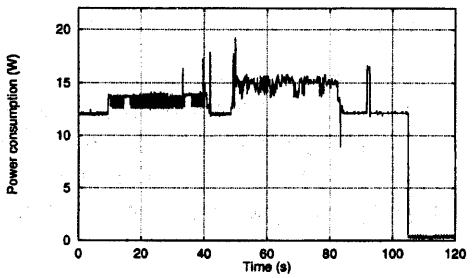
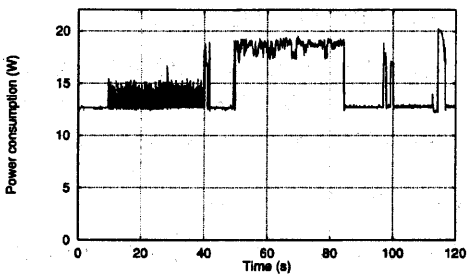
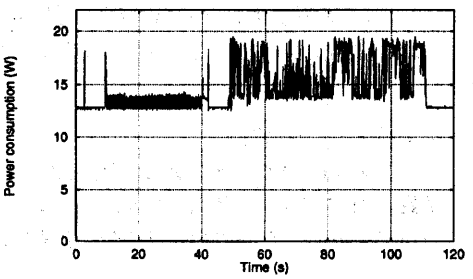


図 3: 動的にデバイスの設定を変更した場合の消費電力の推移.



(a) 従来手法 1



(b) 従来手法 2

図 4: デバイスをあらかじめ決められた設定に固定した場合の消費電力の推移.

表 3: アプリケーション実行中の平均消費電力.

	動的制御	従来手法 1	従来手法 2
ディフォルト	12.3W	12.7W	12.8W
編集	13.4W	13.3W	13.4W
コンパイル (所要時間)	14.9W (32 秒)	18.6W (32 秒)	15.8W (61 秒)

3.3.3 実験結果に対する考察

従来手法 2 のように, 省電力化のために, あらゆるデバイスのパフォーマンスを一律に下げるのは, 作業効率の悪化につながり, ユーザには受け入れられ難い. また, パフォーマンスの低下を嫌って, 従来手法 1 のように高いパフォーマンスに固定すると, 電力の浪費が著しい.

動的制御では, 必要なデバイスのパフォーマンスを高くする一方, 使わないデバイスは積極的に停止することで, メリハリの効いた効果的な省電力制御を実現する. アプリケーションは, 自身のタスク内容を把握できるので, 適切なタイミングで適切な設定に切替えられる. 電力制御のための専用 API を用意し, アプリケーションを制御に参加させる意義は充分ある.

4 動的な省電力制御に必要な機能

4.1 新たな API の必要性

上で述べた実験では, アプリケーションプログラムがデバイスドライバを用いて, 独自にデバイスの設定を切替えた. しかし, 一般には複数のアプリケーションが同時に稼働するので, 個々のアプリケーションが勝手に設定を変更すると, 予期せぬ性能低下を招き問題である. すべてのプロセスを把握する OS が, 省電力制御のための API を提供し, 各

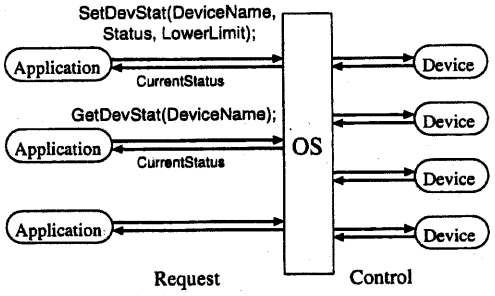


図 5: API の一例.

アプリケーションからの要求を調停する必要がある.

具体的には, 図5のように, デバイスの設定変更を要求する API(**SetDevStat**) や, 現在のデバイスの設定を知るための API(**GetDevStat**) を提供する.

各アプリケーションは, ウィンドウ状態の変更, 自分自身または子プロセスの起動, 特殊機能の実行など, タスクの状態変化に応じてデバイスを適切な設定とするように API を呼び出す.

子プロセスには親プロセスの設定が継承されることにする. 子プロセスが改めて設定変更 API を呼ばない限り, その設定が維持される. 子プロセスによる設定変更は, そのプロセスが終了するまで有効である.

4.2 OS 内部の調停機能

ここでは一例として, ユーザが主として操作対象としているプロセス (例えばウィンドウがフォアグランド状態のアプリケーション) の設定を優先しつつ, 他のプロセスの動作をも保証し, できる限りの省電力を図る調停方法を示す.

4.2.1 調停方法

図5に示すように, 各アプリケーションは, 制御対象デバイス名 $DeviceName$, 要求設定値 $Status$, バックグランド状態で OS に調停される際の許容下限値 $LowerLimit$ とを指定している. ここでは例として, アプリケーション A がフォアグランド状態, アプリケーション $B_i(i = 1, 2, \cdots)$ がバックグランド状態とし, それぞれの設定要求値を $StatA, StatB_i$, バックグランド状態での許容下限値を $LowA, LowB_i$ とする. 値が大きいほど, パフォーマンスが高く, 消費電力も大きいとする.

LCD やオーディオのように, フォアグランド状態のアプリケーション A が, ほぼ独占的に占有するデバイスであれば, OS は A の要求を満足するように配慮すればよい. 例えば, すべての B_i に対して, $StatA > StatB_i$ ならば $StatA$ を, また, $LowB_i > StatA$ となる B_i が存在すれば, それらのの中で最大の $LowB_i$ を設定する. アプリケーション B_i のパフォーマンスは, $StatB_i > StatA$ ならば, 設定値は保証されず, 許容下限値までの範囲で下げられる.

一方, CPU のように, 時分割により複数アプリケーションが共有するデバイスの場合, バックグランド状態のタスクも資源を利用するので, OS はそれらのパフォーマンスを加算して見積もる必要がある. $StatA$ とすべての $StatB_i$ の和が最高性能を越えた場合は, 各 B_i のスケジューリングの優先度を下げるなどして, バックグランド状態タスクのパフォーマンスを抑制する. この場合, 各 B_i にメッセージを通知し, 動作の変更を要求する.

すべての B_i が下限値 $LowB_i$ で動作しても, 全体の要求性能が最高性能を越える場合は, フォアグランド状態のタスク A のパフォーマンスを下げる. この場合も OS からアプリケーション A へメッセージを送り, アプリケーション側に動作の変更を要求する.

パフォーマンス不足のメッセージを受けたアプリケーションは, 例えばワープロのスペルチェックのように, 稼働させなくても全体の動作に影響を及ぼさないオプション機能などを停止し, 低いパフォーマンスに対処する.

4.2.2 調停の具体例

具体的な例として, 先ほどの実験シナリオを考える. 調停の様子を図 6にまとめる. 実験例では, Developer Studio (アプリケーション A とする) は, プログラム編集プロセス開始時に, LCD 輝度を 50% に, またコンパイルプロセス開始時は, LCD 輝度を 0, CPU 速度を要求値 100%, 許容下限値 50%に設定する.

ここで, コンパイル中に, Point Cast のような画面表示を主たる仕事とする別のアプリケーション B が起動され, 起動時に LCD 輝度を要求値 75%, 許容下限値 25%と設定, また, CPU 速度を要求値

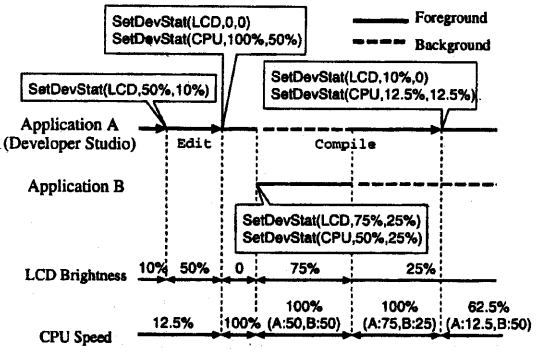


図 6: 2 つのアプリケーション間の調停の様子.

50%，許容下限値 25%と設定したとする．

B がフォアグランド状態の場合，LCD 輝度は 75%に保たれる．CPU は，A，B ともに 50%ずつ配分し，最高速度で動作する．ここで，コンパイル作業中の A をフォアグランドに持ってくると，LCD は消灯されずに，25%の輝度になる．これは，アプリケーション B が，LCD 表示を望んだため，調停された結果である．一方，CPU は，A の要求 100%と B の許容下限値 25%の和が最高性能を越えるので，A に対して要求が満たされないという通知を送り，100%で動作する．この時，B は A より低い優先度を与えられる．A のコンパイルが終了すると，A には 12.5%，B には 50%を供給し，CPU は 62.5%の速度で動作する．

4.3 課題

調停方法の一例を示したが，次のような課題が残されている．

- 本稿では一律にバックグランド状態のタスクを不利に扱う方法を例に示したが，調停の方法についてはさらに検討する必要がある．
- CPU 速度や LCD 輝度のように，機種に依存しない標準的な設定パラメータの提供が必要である．
- 高いパフォーマンスを長時間要求し続けるアプリケーションは，システム全体に影響を及ぼすため，一定パフォーマンスを一定時間以上要求することを禁じるなどの対策が必要である．

5 まとめ

アプリケーションプログラムを省電力制御に参加させ，個々のアプリケーションが持つ情報を生かしながら，柔軟で動的な制御を行なうことで，従来の汎用的制御を大幅に改善する手法とその効果について論じた．

今後は，複数アプリケーション間の調停機能や，個々のデバイスごとの設定値の標準化について検討し，最適な調停アルゴリズムの開発や，API に必要な機能の増強を行ないたい．多くのアプリケーションプログラムが，省電力化を念頭に置いて開発されることを期待する．

参考文献

[1] Cade Metz: Portable Property, PC MAGAZINE, Vol. 15, No. 14, pp. 100-195 (1996)

[2] 下遠野 享: ノートブック PC のパワーマネージメント，情報処理, Vol. 38, No. 2, pp. 135–142 (1997)

[3] 相原 達, 関家 一雄, 黒田 明裕: ノートブック PC におけるローパワーシステム技術, 電子情報通信学会誌, Vol. 80, No. 4, pp. 370–376 (1997)

[4] Intel Corp.: Pentium Processor Family Developer's Manual (1995)

[5] Intel Corp. and Microsoft Corp.: Advanced Power Management BIOS Interface Specification 1.2 (1996)

[6] Intel Corp., Microsoft Corp. and Toshiba Corp.: Advanced Configuration and Power Interface Specification 1.0 (1996)

[7] Microsoft Corp.: OnNow Power Management Architecture for Applications (1997)

[8] Ziff-Davis Inc.: BatteryMark (1996) (http://www.zdnet.com/zdbop/battmark/Home.htm)