修 士 論 文

消費電力の削減を支援する組込みOSに関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

圖子 弘記

2009年3月

修 士 論 文

消費電力の削減を支援する組込みOSに関する研究

指導教官　田中清史 准教授

審査委員主査　田中清史 准教授
審査委員　日比野靖 教授
審査委員　篠田陽一 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0710040 圖子 弘記

提出年月: 2009 年 2 月

概 要

近年，携帯電話やディジタルカメラのような複雑な機能を持ったモバイル機器が広く普及している．これらのモバイル機器は，バッテリーを用いる場合に連続した長い駆動時間が要求されるため，消費電力の一層の削減が重要な課題になっている．

将来の組込み向けのプロセッサにおいて，消費する電力の大部分をキャッシュメモリが占めると予想される．これは，高性能化の要求からプロセッサ内で占めるキャッシュメモリの面積比率が増え，キャッシュメモリの消費電力がプロセッサ全体の消費電力の大部分を占めると考えられるためである．キャッシュメモリの消費電力削減法の一つとしてソフトウェア Self-Invalidation[3] がある．この手法は，今後再び参照されないキャッシュブロックを予測し，そのブロックを自発的に無効化させて電力供給を断つことにより消費電力を削減するものである．

本研究では，ソフトウェア Self-Invalidation を組込み OS に適用する．同時に，アプリケーション開発者による電力削減を支援する環境を整備する．最後に，シミュレーションにより評価を行う．

目 次

第1章　はじめに

1.1　研究の背景

近年，ウルトラモバイル PC（UMPC) やスマートフォンに代表される高性能なモバイルコンピュータが普及し，組込みシステムにおいてもプロセッサの高性能化と低消費電力化を同時に満たす要求が高まっている．一般的に，このようなモバイル機器には，制御プログラムとしてオペレーティングシステム（OS) が搭載されている場合がほとんどである．OS を搭載することにより，アプリケーションプログラムとそれを管理するプログラムの 2 つに分割され，プログラムの生産性や保守性を高めている．また，搭載される OS は，組込みシステムの特性であるリアルタイム性を向上させるために，リアルタイム OS (RTOS) である場合が多い．

一方，消費電力削減技法については，CPU パワーや搭載メモリサイズなどのハードウェア資源を制限するまでに至り，実質的な低消費電力化のサポートは行われていない．しかしながら，システムの高機能化・多機能化が進むにつれて，プログラムの規模が膨大し，特に複数のタスク間で大量のデータを共有している場合などには処理が複雑になり，電力消費も激しくなる．さらに，リアルタイム性が要求される組込みシステムでは，制限されたハードウェア資源で処理要求を満たすことも難しくなる．

以上より，組込みシステムにおいて，性能を維持したままプロセッサの消費電力を削減することは重要な課題といえる．

1.2　研究の目的

将来の組込み向けプロセッサは，近年のマイクロプロセッサ同様に面積の大部分をキャッシュメモリが占め，消費する電力の大部分もキャッシュメモリが消費すると考えられる．近年，キャッシュメモリの電力削減を目的とした研究が報告されている．

キャッシュメモリの消費電力を削減する手法の 1 つとして Gated-Vdd[1] が提案されている．Gated-Vdd は，キャッシュメモリを構成する SRAM セルと GND の間に閾値の高いトランジスタを設け，このトランジスタが電力供給を制御することによって，リーク電流を削減することができる．この手法は，キャッシュブロック単位で供給電圧を制御しているため，対象とするキャッシュブロックの供給電圧が遮断されると，そのブロックに保持されているデータは消滅し，キャッシュメモリの電力を削減する．

次に，Gated-Vdd を使用したキャッシュメモリの構成例として，Cache decay[2] が提案された．Cache decay ではキャッシュメモリの時間的局所性に着目し，あるキャッシュブロックが一定時間アクセスされなかった場合，そのキャッシュブロックは今後再利用されないと判断し，Gated-Vdd による電力供給制御を行うものである．

次に，ソフトウェア Self-Invalidation[3] が提案された．この手法は，あるキャッシュブロックが今後再び参照されないと予想される場合に，そのブロックを最後に参照する命令を Last-Touch 命令に置換する．そして，この Last-Touch 命令により，参照されるブロックが自発的に無効化されて電力供給が断たれ，結果として消費電力が削減される．

本研究では，μITRON4.0[4] に準拠した組込み OS をターゲットとしている．ソフトウェア Self-Invalidation を組込み OS のシステムコール内に適用し，電力に最適化された組込み OS を提供する．更に，アプリケーション開発者が容易に Last-Touch 命令を適用できる環境を提供する．以上により，OS とアプリケーションの双方で電力削減を達成することを目的とする．

1.3 本論の構成

本論文は全 6 章で構成される．

第 2 章では，従来のキャッシュメモリにおける電力削減手法と，関連研究ついて述べる．
第 3 章では，ソフトウェア Self-Invalidation について述べる．
第 4 章では，電力最適化組込み OS と電力削減支援環境について述べる．
第 5 章では，評価環境について述べた後，提案手法の評価を行う．
第 6 章では，まとめと今後の課題について述べる．

第2章　関連研究

本章では，キャッシュメモリの消費電力削減法として報告されている Gated-Vdd[1], Cache decay[2]，及びソフトウェア Self-Invalidation[3] について説明する．

2.1　gated-Vdd[1]

キャッシュメモリのリーク電流を削減するため，キャッシュメモリを構成する SRAM セルと GND の間に閾値の高いトランジスタを設ける (図 2.1)．そして，このトランジスタのスイッチングによりキャッシュメモリに対する供給電圧を制御する手法である．

しかしながら，キャッシュメモリを構成するすべての SRAM セルを図 2.1 の構成にすると，キャッシュメモリの面積があまりにも増大してしまう．文献 [1] において，各キャッシュブロックにつき 1 つの Gated-Vdd を設けるのが性能と面積のトレードオフが最も優れているとされている (図 2.2)．

また，この方式はキャッシュブロックへの電力供給を遮断するため，そこに保持されていたデータは消滅するという性質を持っている．そのため，キャッシュミスペナルティが増加する場合がある．

2.2　cache decay[2]

Gated-Vdd を使用したキャッシュメモリの消費電力削減手法の 1 つとして，Cache decay がある．

キャッシュメモリへのアクセスには局所性があり，アクセス傾向として，キャッシュブロックへデータが格納されてから最後にアクセスされるまでのアクセスが頻繁に発生する期間 (live time)，最後にアクセスされてからリプレースされるまでのアクセスが発生しない期間 (dead time) がある．dead time 中のキャッシュブロックは，アクセスされることなくデータを保持し続け，無駄な電力を消費していることになる．Cache decay では，dead time 中のキャッシュブロックに対する電力供給を，Gated-Vdd[1] により遮断することで電力削減を図っている．

キャッシュメモリへのアクセスには時間的局所性が存在し，これをあるキャッシュブロックが dead time であるかどうかの判断に利用できる．つまり，ある一定期間 (decay interval) アクセスされなかったキャッシュブロックは dead time 中であると判断し，リプレースさ

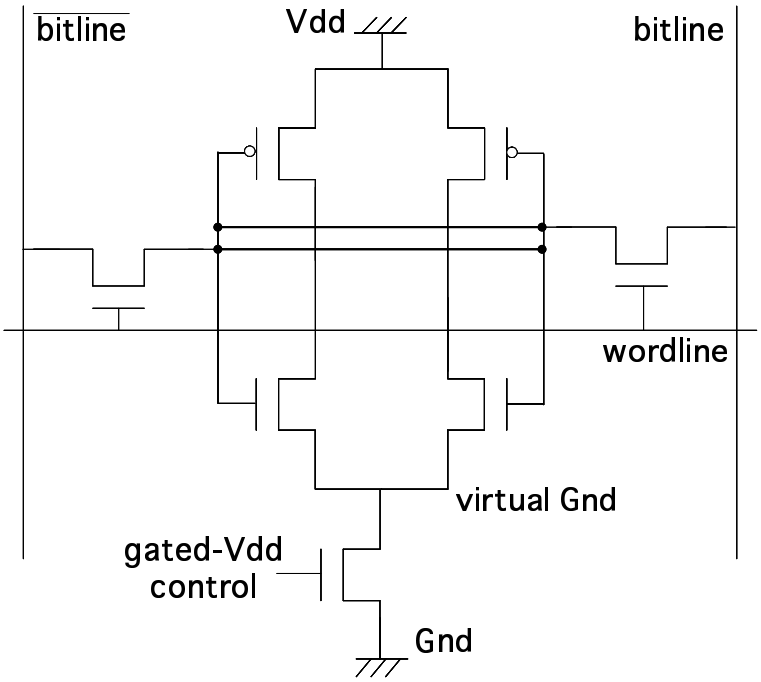

図 2.1: NMOS トランジスタによる SRAM セルの gated-Vdd 構成

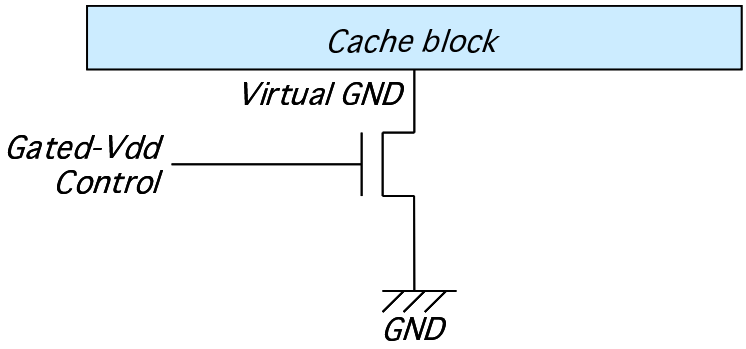

図 2.2: キャッシュブロック単位での電圧制御の例

れるまでの間，供給電力がカットされる．したがって，この decay interval の設定を適切に行うことが重要となる．decay intervel を同一キャッシュブロックへのアクセス間隔 (access interval) より短く設定した場合，キャッシュミスが増大し，プロセッサの処理速度が著しく低下する．逆に，dead time より長く設定した場合，処理速度の低下は起きないが，電力削減効果はほとんど期待できない．このため，decay interval の設定は，access interval より長く，dead time より短く値を設定する必要がある．

図 2.3 に，あるキャッシュブロックのアクセスパターンと cache decay での電力制御の概要を示す．

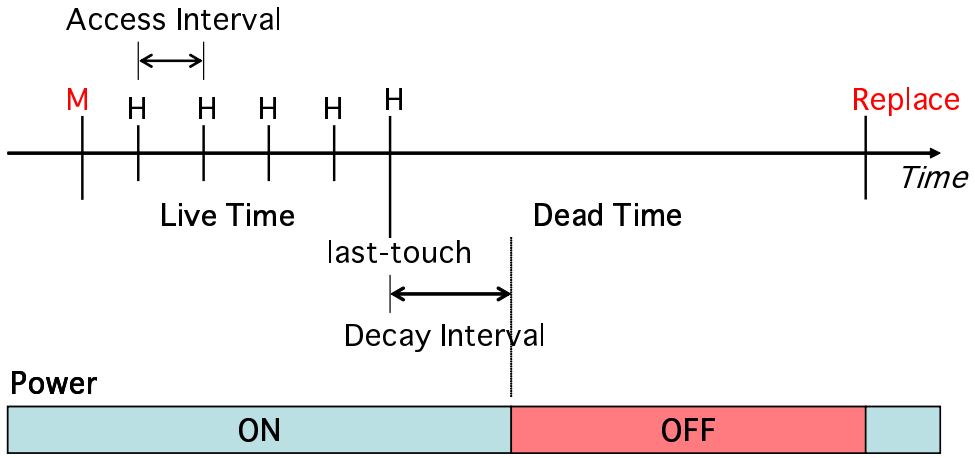

図 2.3: あるキャッシュブロックへのアクセスパターンと電力制御

2.3 ソフトウェア Self-Invalidation[3]

Gated-Vdd を使用したキャッシュメモリの消費電力削減手法の 1 つとして，ソフトウェア Self-Invalidation がある．

あるキャッシュブロックがメモリアクセス命令によって今後再び参照されないと予想される場合，キャッシュメモリ上に残されたデータは無駄に電力を消費し続けることになる．そこで，ソフトウェア Self-Invalidation[3] では，あるキャッシュブロックを最後に参照する命令を Last-Touch 命令に置換することによって，参照されるブロックが自発的に無効化される．無効になったキャッシュブロックは，Gated-Vdd によって制御され，電力供給が遮断される．これにより，キャッシュミスを増やすことなく消費電力が削減される．

ただし，ソフトウェア Self-Invalidation では，Last-Touch 命令に置換するための前提としてメモリアクセストレースが必要であるため，事前のプログラム実行が不可欠となり，実現性に乏しい．

図 2.4 に，ソフトウェア Self-Invalidation による電力制御の概要を示す．ソフトウェア Self-Invalidation についての詳細は，次章で詳しく述べる．

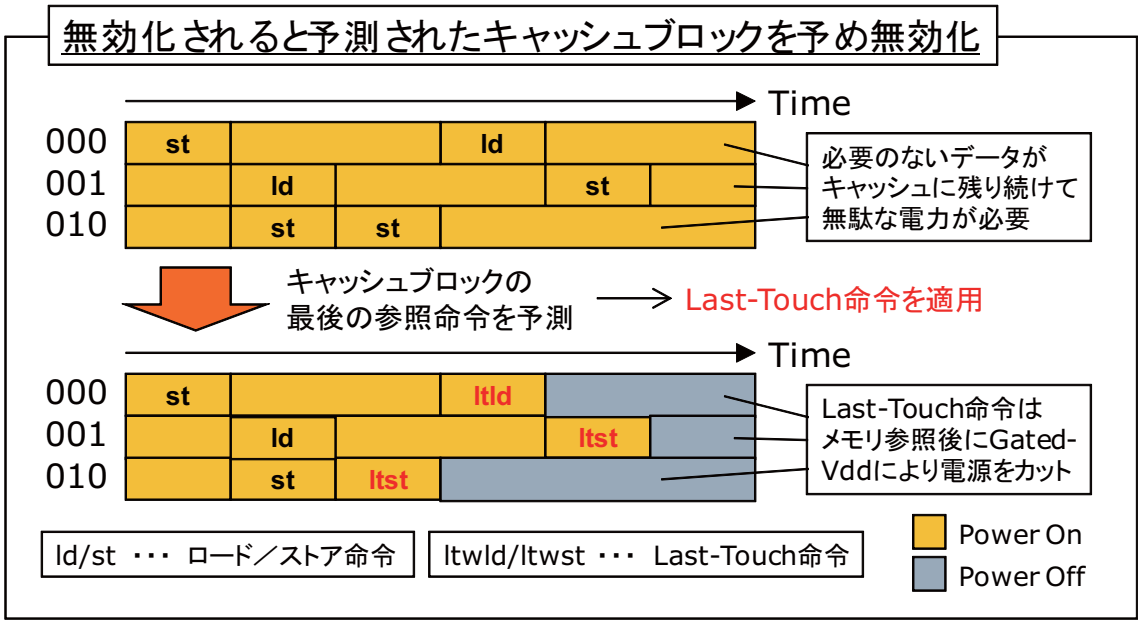

図 2.4: ソフトウェア Self-Invalidation による電力制御

第3章　ソフトウェアSelf-Invalidation

本章では，ソフトウェア Self-Invalidation について説明する．

3.1　ソフトウェアSelf-Invalidation

ソフトウェア Self-Invalidation では，Gated-Vdd を制御する専用命令として Last-Touch 命令 (last-touch load/store) を命令セットに導入している．そして，この Last-Touch 命令をメモリブロックを最後にアクセスする命令と置き換えて実行するという方式をとる．

3.1.1　Last-Touch 命令

Last-Touch 命令は，通常のロード・ストア命令と同様の振る舞いをするが，キャッシュメモリに対しては対象メモリブロックへのアクセスが完了すると同時に該当するキャッシュブロックを無効化するという機能を持つ．そして，無効化したキャッシュブロックの供給電力を Gated-Vdd によって遮断することで，キャッシュメモリにおける消費電力削減を実現している．

通常，キャッシュメモリのブロックサイズはマルチワードである．しかし，メモリアクセス命令によるキャッシュメモリへのロード・ストアは，ブロックサイズよりも小さいサイズで行われる．

例えば，ブロックサイズが 4 ワードのキャッシュブロックへのシーケンシャルアクセスを考える．単一のメモリアクセス命令がブロック内の４つのワードにそれぞれ一回ずつアクセスした場合，４回目のメモリアクセスが，このブロックにとっての最終アクセスとなる．このような場合に，メモリアクセスと同時にブロック全体を無効にする Last-Touch 命令を実行すると，すべてのアクセスでキャッシュミスが発生し，性能低下を引き起こす．これは，そのようなアクセス傾向があるキャッシュブロックから見ると，4 回に 1 度そのキャッシュブロックへのラストアクセスとなるためである．

そこで，Last-Touch 命令を以下の 2 種類用意することで，マルチワードブロックのキャッシュメモリに対応可能にする．

- Last-Touch Block **命令(last-touch block load/store)**
  対象メモリブロックへのアクセスが完了すると同時に，アクセスしたキャッシュブロック内のワード位置にかかわらず，該当するキャッシュブロックを無効化する．

- Last-Touch Word **命令(last-touch word load/store)**
  対象メモリブロックへのアクセスが完了すると同時に，アクセスしたキャッシュブロック内のどのワードにアクセスしたかを記憶する．キャッシュブロック内のすべてのワードがこの命令によってマークされると，そのキャッシュブロックを無効化する．

3.1.2 Last-Touch 命令への置換

置換対象となるメモリアクセス命令は，アクセスしたメモリブロックが将来的に１度もアクセスされることがない場合，つまりそのブロックを最後にアクセスする命令を Last-Touch 命令への置換対象とする．図 3.1 において，ループ内の配列 src を参照するメモリアクセス命令はブロック内ワード数ごとのアクセス回数に１度，該当メモリブロックへのラストアクセスとなる（配列 src は今後使用されないとする）．この場合，Last-Touch Word 命令に置換することで，キャッシュミスの増大を回避しながらキャッシュブロックを無効にすることが可能である．

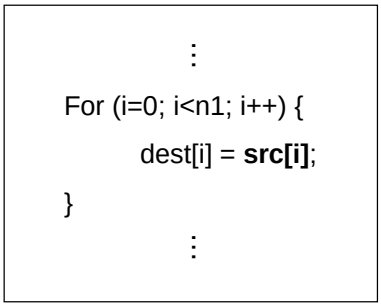

図 3.1: Last-Touch 命令の適用箇所

3.2 キャッシュメモリの構成

Last-Touch Word 命令の導入により，従来のキャッシュメモリの構造にやや変更を加える必要がある．以下に，付加される機構について述べる．

3.2.1 ラストタッチフラグビット

Last-Touch Word 命令を実装するには，この命令がキャッシュブロック内のどのワードに対してアクセスしたかという情報を記録するための領域（以降，ラストタッチフラグビットと呼ぶ.）が必要である．ラストタッチフラグビットは，キャッシュブロック内の１

ワードにつき１ビットを割り当て，対応するワードに対して Last-Touch Word 命令での参照があったかどうかを記録する．

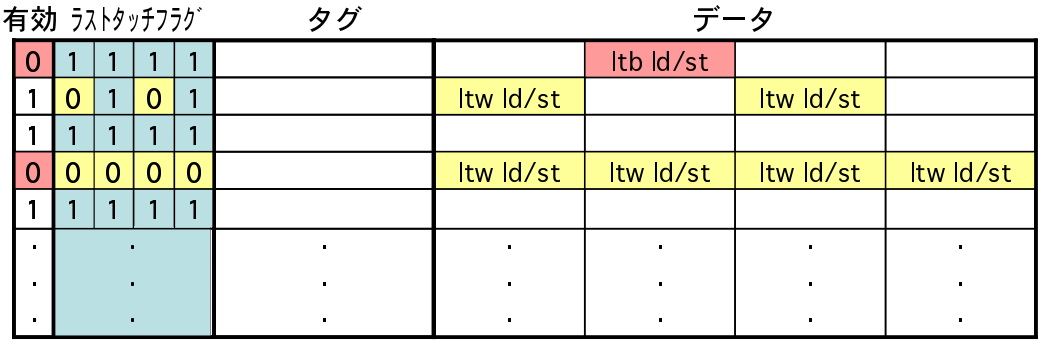

図 3.2: キャッシュメモリの構成

図 3.2 に Last-Touch Word 命令に対応したキャッシュメモリの構成を示す．また，このキャッシュメモリ構成における 2 種類の Last-Touch 命令の振る舞いは以下のようになる．

- Last-Touch Block Load/Store(ltb ld/st) **命令の動作**
  Last-Touch Block 命令によってアクセスされた場合，アクセスされたキャッシュブロックのワード位置にかかわらず，該当するブロックの有効ビットがクリアされ，無効状態になる．

- Last-Touch Word Load/Store(ltw 6:21 2009/02/05ld/st) **命令の動作**
  Last-Touch Word 命令によってアクセスされた場合，キャッシュタグ中の対応するワード位置のラストタッチフラグがクリアされる．ラストタッチフラグがすべてクリアされると，有効ビットもクリアされ，該当するブロックが無効状態になる．

3.2.2 キャッシュメモリの電力供給機構

キャッシュメモリの供給電力制御は，Gated-Vdd を用いる．キャッシュブロックごとに 1 つの Gated-Vdd を付加し，ブロック単位での電力制御を行う．Gated-Vdd の制御信号は，キャッシュの有効ビットを使用する．これらの構成を図 3.3 に示す．

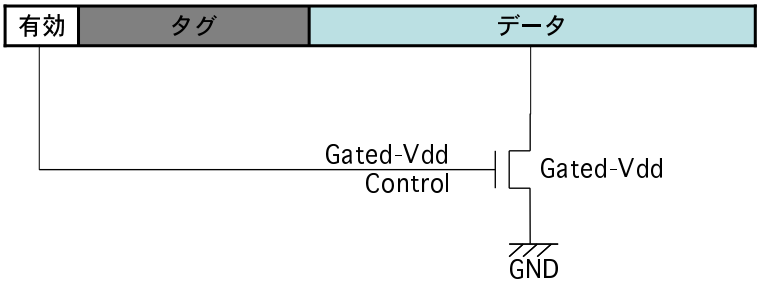

図 3.3: キャッシュメモリの供給電力制御

第4章　電力最適化組込みOSと電力削減支援環境

本章では，ソフトウェア Self-Invalidation を OS のシステムコール内部に適用した電力最適化組込み OS について述べる．また，アプリケーション開発者による電力削減支援環境について述べる．

4.1　リアルタイムオペレーティングシステム

本節では，提案手法について述べる前に実装ターゲットとなるリアルタイム OS である μITRON 仕様 OS ついて説明する．まず，本論で使用する ITRON 仕様に基づいた用語を説明し，次に OS の持つ基本的な機能について述べる．

4.1.1　用語の定義

ITRON 仕様では，並行処理するプログラムの単位をタスクと呼ぶ．マルチタスク機構により概念的に複数のタスクが同時に実行される．マルチタスクは，複数あるタスクが一定の規則により実行状態が時分割で切り替わることにより実現している．実行するタスクが切り替わることをディスパッチと呼び，OS 内でディスパッチを行う部分をディスパッチャと呼ぶ．

ITRON 仕様 OS は，優先度の高いタスクから優先的に実行される．優先順位を基準に次に実行するタスクを決定することをスケジューリングと呼ぶ．タスク間の優先順位は，スケジューラがタスクの優先度に基づいて決定する．

4.1.2　μITRON 仕様 OS

μITRON 仕様 OS の個々の機能について以下に示す．

- **タスク管理機能**
  タスクを管理する主な情報として，タスク ID（タスクの名前），タスクの状態，タスクの優先度などがある．これらの管理情報はタスクコントロールブロック（TCB）に格納される．タスクの管理はすべてこの TCB の情報に基づいて行われる．

ITRON 仕様において，タスクのとりうる状態として，実行状態（RUNNING），実行可能状態（READY），待ち状態（WAITING），強制待ち状態（SUSPENDED），二重待ち状態（WAITING-SUSPENDED），休止状態（DORMANT），未登録状態（NON－EXISTENT）の７つの状態が定義されている．

- **タスク付属同期機能**
  タスクの状態を直接的に操作することで同期を行うための機能である．具体的には，タスクを起床待ち状態にしたり，待ち状態から起床させる機能などがある．

- **同期・通信機能**
  タスク間の同期・通信の機能として，セマフォ，イベントフラグ，メールボックスなどの機能がある．

- **メモリプール管理機能**
  アプリケーションが動的にメモリを使用する場合に使用される．OS は要求されたメモリサイズを獲得してアプリケーションに通知する．アプリケーションが不要になったメモリは OS に対してメモリ解放を要求して解放が行われる．

- **時間管理機能**
  時間に依存した処理を行うための機能である．システム時刻を操作する機能としてシステム時刻を設定/参照/更新する機能がある．

- **システム状態管理機能**
  システムの状態を変更・参照するための機能である．具体的には，実行状態のタスク ID を参照する機能，CPU ロック状態への移行／解除，コンテキストやシステム状態を参照する機能がある．

- **割込み管理機能**
  割込みによって起動される処理として，割込みハンドラと割込みサービスルーチンがあり，いずれか一方あるいは両方の機能が提供される．

4.2 電力最適化組込み OS

電力最適化組込み OS では，OS のシステムコール内に Last-Touch 命令を埋め込むことにより，OS の利用者が消費電力削減の恩恵を受けることを目的とする．

4.2.1 Last-Touch 命令の適用箇所とタイミング

ソフトウェア Self-Invalidation を組込み OS に適用するために，まず Last-Touch 命令の適用箇所とそのタイミングについて検討する．

本研究では，ITRON 仕様 OS が提供する機能のうち，データ通信機能としての役割を担うメールボックスと固定長メモリプールの関係に着目し，Last-Touch 命令の適用を考える．一般に，メールボックスでは固定長メモリプールから獲得したメモリブロックに対してデータが格納される．そして，そのメモリブロックがタスク間で授受された後，受信側タスクで不要になった時点でそのメモリブロックがメモリプールに返却される．

カーネルでは，タスク終了時に，タスクが獲得したメモリブロックなどの資源を解放する処理を行わない．これらはすべてアプリケーション側の責任で行われるため，メモリプールを用いた処理では必然的にメモリブロックを返却するシステムコールが呼ばれることになる．

以上から，Last-Touch 命令の適用は，データの受け渡しを終えてメモリブロックがメモリプールへ返却されるタイミングが最適であると考えられる．OS の利用者は，受信側のタスクからメモリブロックを返却するシステムコールを呼び出すことで，OS レベルでの電力削減効果が得られる．

メールボックスと固定長メモリプールを併用した処理での Last-Touch 命令の適用例を図 4.1 に示す．

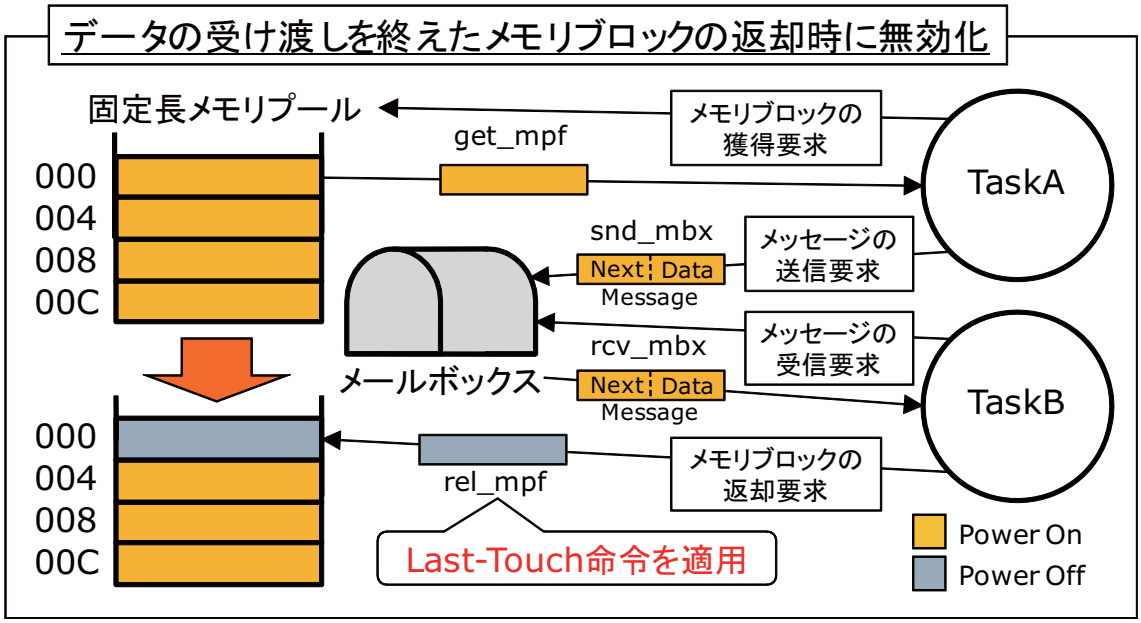

図 4.1: メールボックスと固定長メモリプールの処理での Last-Touch 命令の適用例

4.2.2 Last-Touch 命令の適用方法

システムコールへの Last-Touch 命令の適用は，システムコールの処理に直接アセンブリコードを埋め込むことにより実現する．本研究では，専用命令である Last-Touch 命令

としてSPARCV8[5] 命令セットのASI(Address Space Identifer) フィールドを持ったロード命令を使用する．そして，その命令を”asm volatile”によってコード上に埋め込み，gccを使用して実行バイナリを生成する．

4.3 電力削減支援環境

本節では，アプリケーション開発者が容易にLast-Touch 命令を適用できる環境を提案する．そして，アプリケーション開発者による電力制御を支援することを目的とする．

4.3.1 アプリケーション開発者による電力制御

一般に，組込みシステムではある特定の用途向けにハードウェアとソフトウェアが開発されるため，ソフトウェアを開発するにあたって，プログラマはハードウェアの構造を熟知した上でプログラムのチューニングを行っている．

よって，電力削減の最適化をプログラマサイドで行うことが期待できる．以降，Last-Touch 命令をプログラム内に埋め込むためのインターフェースを提供し，アプリケーション開発者を支援する方法について述べる．

4.3.2 Last-Touch 命令適用指示子と適用支援ツールの導入

まず，Last-Touch 命令に置換するための指示子を導入する．この指示子は，プログラマによってソースコード内にコメント文の形で挿入される．コメント文として挿入することにより，プログラム本来の意味を変えずにLast-Touch 命令を適用することができる．また，Last-Touch 命令を適用しない場合においても，そのままの形でコンパイル可能である．

指示子の挿入の際には，Last-Touch 命令で無効化したい変数や配列の要素などを具体的なターゲットとして指定する．ただし，指定可能なターゲットは，同一行にある代入式のうちラストアクセスとなるターゲットのみである．プログラマは自らの采配でソースコード内にこの指示子を埋め込む．このような指示子を基にLast-Touch 命令を適用するツールを用いることにより，プログラマは容易に電力を制御することが可能となる．図4.2 にプログラマによるLast-Touch 命令の適用例を示す．

4.3.3 Last-Touch 命令置換方法

Last-Touch 命令への置換は，4.2.2 節と同様にコードに直接アセンブリを埋め込むことで行う．ただし，適用するターゲットの参照がロードとストアのどちらであるかを判断しなければ命令を埋め込むことはできない．これを解決するために，ターゲットへの参照は

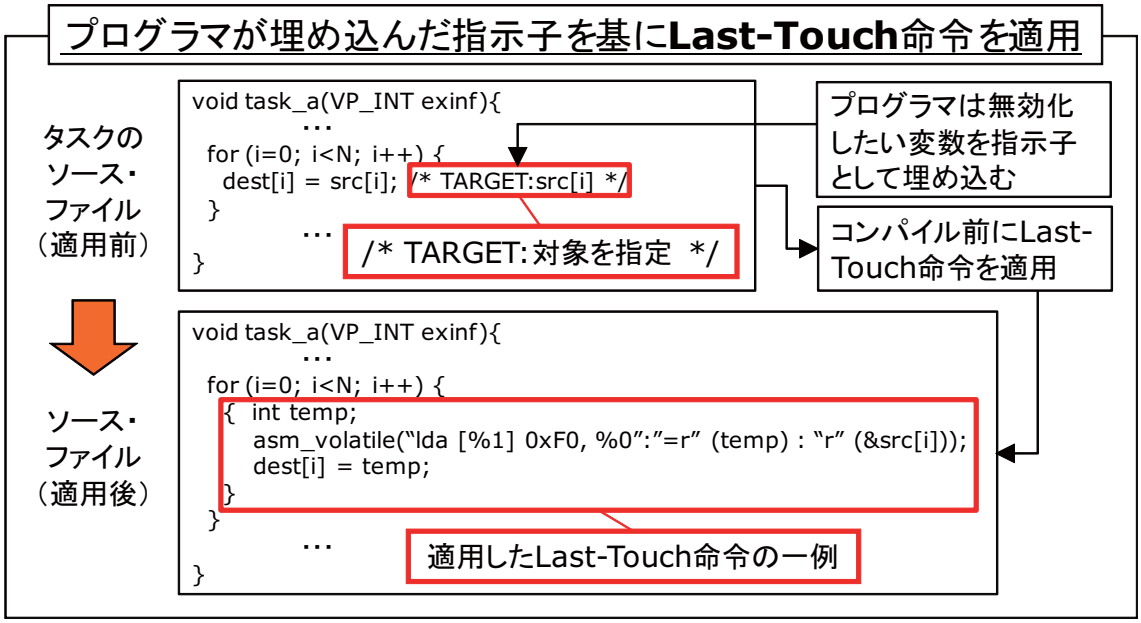

図 4.2: プログラマによる Last-Touch 命令の適用例

代入文によって実行されることに注目する．そのターゲットへの参照が代入文の左式に存在するか右式に存在するかで，ロードかストアかを判断することができる．もし，左式に置換対象のターゲットが存在するならば，ストアを実行する Last-Touch 命令に置換する．置換対象のターゲットが左式に存在する場合の Last-Touch 命令の置換例を図 4.3 に示す．

逆に，右式に置換対象のターゲットが存在するならば，ロードを実行する Last-Touch 命令に置換する．この場合，ロードしただけではその代入文全体を実行することができないので，そのロードした値を一時的に保持する領域が必要となる．これには参照配列と同じ型宣言の一時変数を用意することで問題を解決する．置換対象の配列型参照が右式に存在する場合の Last-Touch 命令の置換例を図 4.4 に示す．

代入式の左式，右式共に置換対象の配列型参照が存在した場合であっても，これらの方法を組み合わせることで適用可能である．

```
for (i=0; i<N; i++) {
  dest[i] = src[i]; /* TARGET:dest[i] */
}
```

dest[i]にLast-Touch命令を適用

```
for (i=0; i<N; i++) {
  asm_volatile("sta %0, [%1] 0xF0"::"r" (src[i]), "r" (&dest[i]));
}
```

図 4.3: 置換対象のターゲットが左式に存在する場合の置換例

```
for (i=0; i<N; i++) {
  dest[i] = src[i]; /* TARGET:src[i] */
}
```

src[i]にLast-Touch命令を適用

```
for (i=0; i<N; i++) {
  {
    int temp;
    asm_volatile("lda [%1] 0xF0, %0":"=r" (temp) : "r" (&src[i]));
    dest[i] = temp;
  }
}
```

図 4.4: 置換対象のターゲットが右式に存在する場合の置換例

第5章　評価

本章では，シミュレーション環境について述べた後，シミュレーションによる提案手法の評価と考察について述べる．

5.1　シミュレーション環境

評価に用いるCPUシミュレータの主な仕様は以下の通りである．

- SPARC V8命令セットアーキテクチャ[5]
- 16KB L1命令キャッシュ
- 16KB L1データキャッシュ

命令，データキャッシュは，それぞれ32バイトブロック4ウェイセットアソシアティブ方式キャッシュで，1ブロック8ワードで構成されている．キャッシュメモリの置換アルゴリズムはLRUを使用している．キャッシュへの書き込みはライト・バック方式である．また，今回の計測ではL2キャッシュは用意していない．キャッシュヒット時は1クロックサイクルで読み書き可能であるが，キャッシュミス時のCPUストール時間を10クロックサイクルとしている．

5.2　キャッシュメモリの消費電力計算

プログラムの実行開始から終了までのキャッシュメモリ1cellあたりの消費リーク電力E_{leak}は，式(5.1)で計算できる．

$$E_{leak} = E_{standby} \times \frac{t_s}{f_c} + E_{active} \times \frac{(t_e - t_s)}{f_c} \tag{5.1}$$

$E_{standby}$(Standby Leakage Energy[nJ])は，スタンバイ状態のSRAM 1cellあたりのリーク電力である．同様に，E_{active}(Active Leakage Energy[nJ])は，アクティブ状態のSRAM 1cellあたりのリーク電力である．これらの見積もりには，文献[1]にて採用されている値を使用した(表5.1)．

表 5.1: SRAM 1cell あたりのリーク電力

Technique	Active Leakage Energy[nJ]	Standby Leakage Energy[nJ]
no gated-Vdd	1740	N/A
gated-Vdd	1740	53

そして，t_s はスタンバイ時間 [cycle]，t_e はプログラム実行時間 [cycle]，f_c はクロック周波数 [Hz] である．また，スタンバイ状態からアクティブ状態になる際のレイテンシについては，キャッシュミスのレイテンシよりも小さいため，特に考慮しない．

ソフトウェア Self-Invalidation では，キャッシュブロック単位での電力制御を行う．よって，1 ブロックあたりの消費リーク電力はブロックサイズを S_{block}[Byte] とすると，式 (5.2) で表現できる．

$$E_{block} = E_{leak} \times S_{block} \times 8 \tag{5.2}$$

この式によって，各ブロックごとの消費リーク電力を計算し，集計することでキャッシュメモリ全体の消費電力を見積もる．

5.3 評価用タスクセット

組込みアプリケーションにおいて，一般的に扱われると想定されるデータ処理を取り上げて，タスクの実行時間から消費電力量を計測する．評価対象として，1〜1000 までの値を配列に格納し，これらの要素を以下の計測方法に従って加算を行うタスクセットを作成した．

- **計測１：**
  タスクを１つ実行し，タスク内で 1〜1000 の値を保持する配列要素を順に加算する．
- **計測２：**
  タスクを２つ実行し，一方のタスクが 1〜1000 までの配列要素を１要素ずつ 1000 回送信し，他方のタスクがこれらを受信し加算する．メッセージの受け渡し領域として固定長メモリプールを用いる．

5.4 シミュレーション結果

本節では，5.3 節のタスクセットを用いてシミュレーションを行った結果を示す．シミュレーション結果はそれぞれの計測パターンに対して，通常実行した場合（Normal），Gated-Vdd により無効状態のキャッシュブロックの電力を制御して実行した場合（Gated-Vdd），

ソフトウェア Self-Invalidation を OS のみ適用した場合（OS Only），ソフトウェア Self-Invalidation をアプリケーションのみ適用した場合（Application Only），ソフトウェア Self-Invalidation を OS とアプリケーション双方で適用した場合（OS and Application）の計 5 通りの比較結果を示す．

また，計測パターンによっては，キャッシュブロックを有効に利用しない場合がある．この場合，本手法を適用した部分とは無関係に電力が削減されてしまうため，公平でない．そこで，すべての計測パターンに対して計測開始直後にすべてのキャッシュブロックをアクティブ状態にする処理を施している．また，全体の実行サイクルが短いと，本手法を適用する場合とそうでない場合で電力削減効果に差が生じにくくなる．そこで，実行サイクルが短い計測パターンに対しては，計測終了時間を一定時間遅延させる処理を施している．

5.4.1 計測 1 の結果

計測 1 では，あるタスクにおける局所的な配列アクセスに対してプログラマが Last-Touch 命令を適用した場合のソフトウェア Self-Invalidation の性能を評価する．本計測は，タスク間通信を行う必要がないためアプリケーション側のみ適用した場合の結果を示し，OS 側の計測は行わない．計測 1 の消費電力の結果を図 5.1 に示す．また，Last-Touch 命令の実行回数やキャッシュミス数，実行サイクル数などの結果については表 5.2 に示す．

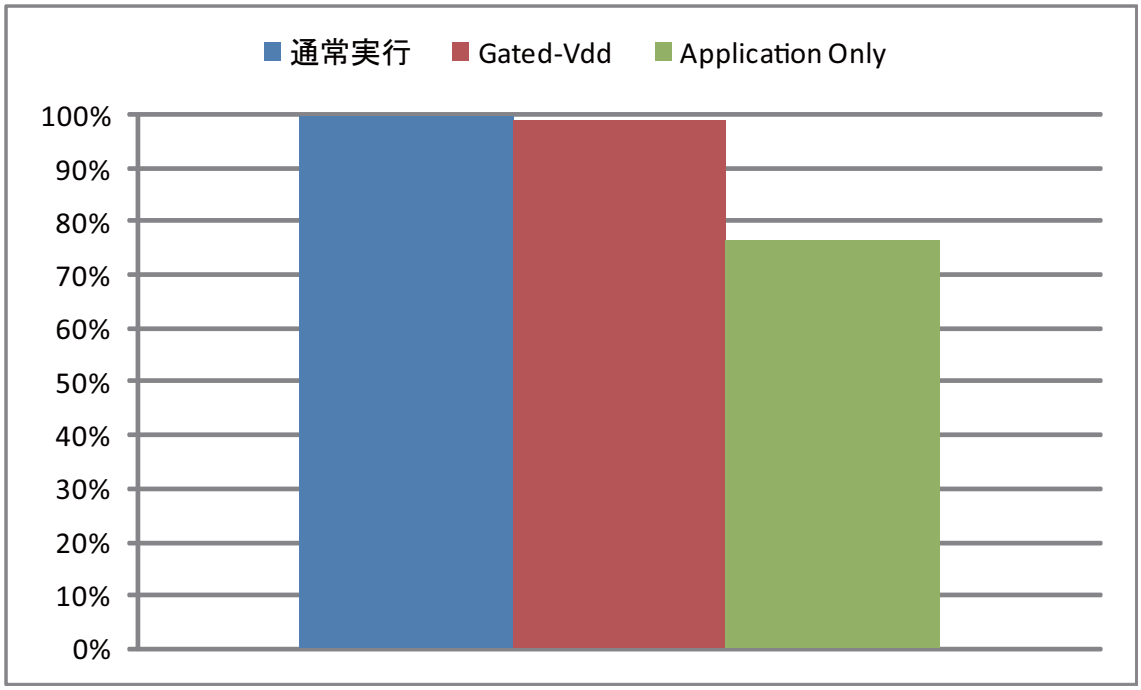

図 5.1: 計測 1 における 3 つの実行状態の消費電力

表 5.2: 計測 1 における 3 つの実行状態の全結果

	Normal	Gated-Vdd	Application Only
Last-Touch 命令実行数	0	0	1000
キャッシュミス数	1812	1812	1813
実行サイクル数	2091139	2091139	2090148
消費電力削減率	0%	1%	23%

図 5.1 では，通常実行時のキャッシュメモリの消費電力を 100%とし，それに対し Gated-Vdd 実行時やソフトウェア Self-Invalidation 実行時 (アプリケーションのみ適用時) の消費電力が何%であるかを示している．Gated-Vdd においては 1%とほとんど電力が削減できていないが，アプリケーションのみ適用時において 23%の電力が削減されている．

表 5.2 より，アプリケーションのみ適用時において Last-Touch 命令が 1000 回実行されている．これは，プログラマがループ内の演算部分に対して Last-Touch 命令を埋め込むことによって，1000 個の配列要素に対して Last-Touch 命令が適用されていることを示している．Last-Touch 命令の実行回数は，実質的には Last-Touch Word 命令が実行された総数であり 8 回に 1 回キャッシュブロックが無効化される．Last-Touch 命令の実行回数が直接消費電力の削減効果と比例するわけではない．消費電力の削減効果は，無効状態のキャッシュブロックが存在した時間に比例する．今回の計測においては，キャッシュブロックが Self-Invalidation されてから計測を終えるまでの間，ブロックの状態が変わらずにある程度の時間差が存在したために充分な電力削減効果が得られている．実際にはプログラムの実行時間とキャッシュブロックの状態によって，この電力削減効果は大きく異なる．

以上より，プログラマが Last-Touch 命令を適用することで電力が削減されることを確認した．

5.4.2 計測 2 の結果

計測 2 では，タスク間におけるメッセージの送受信に OS が Last-Touch 命令を適用した場合のソフトウェア Self-Invalidation の性能を評価する．また，送信する配列要素をメッセージ領域にコピーする際の局所的な配列アクセスに対してプログラマが Last-Touch 命令を適用した場合のアプリケーション側からの観点も含めて評価する．計測 2 の消費電力の結果を図 5.2 に示す．また，Last-Touch 命令の実行回数やキャッシュミス数，実行サイクル数などの結果については表 5.3 に示す．

図 5.2 では，通常実行時のキャッシュメモリの消費電力を 100%とし，それに対し Gated-Vdd 実行時や 3 つの実行形態（OS Only，Application Only，Os and Application）のソフトウェア Self-Invalidation の実行時の消費電力が何%であるかを示している．Gated-Vdd ではほとんど削減効果が得られていない．OS のみ適用時とアプリケーションのみ適用時

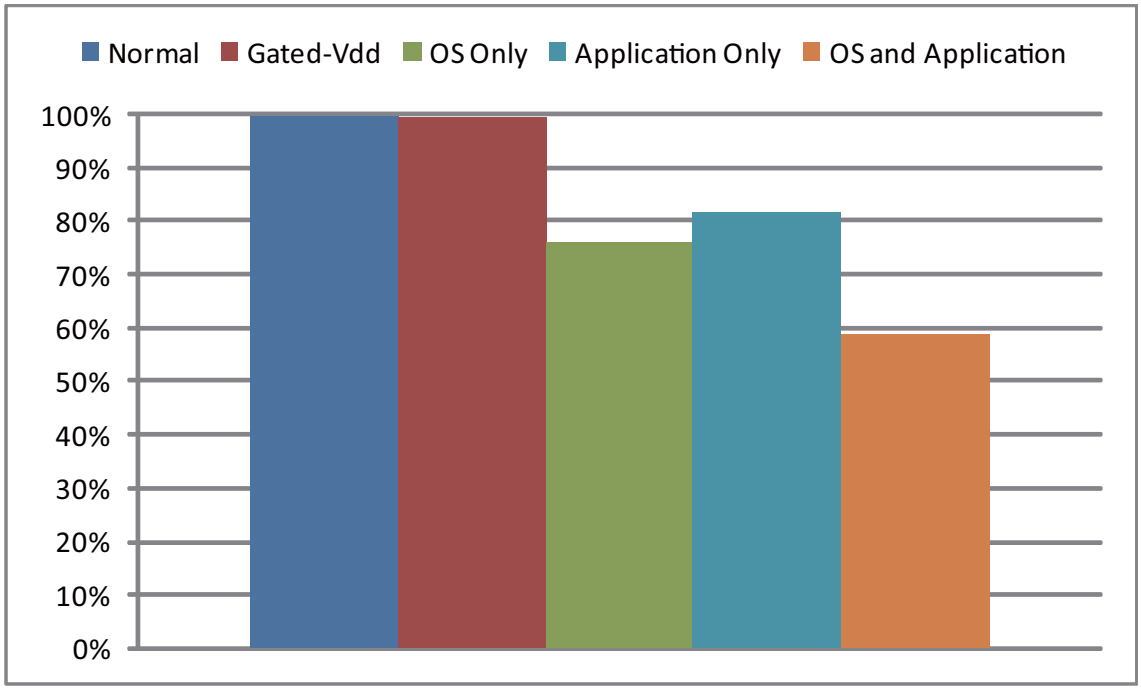

図 5.2: 計測 2 における 5 つの実行状態の消費電力

表 5.3: 計測 2 における 5 つの実行状態の全結果

	Normal	Gated-Vdd	OS Only	Application Only	OS and Application
Last-Touch 命令実行数	0	0	2000	1000	3000
キャッシュミス数	2657	2657	2662	2657	2689
実行サイクル数	8184773	8184773	8201823	8183773	8201093
消費電力削減率	0%	0%	24%	18%	41%

でそれぞれ 24%と 18%の電力が削減されている．また，OS とアプリケーション双方で適用した場合には 41%の電力が削減されている．

OS のみ適用した場合とアプリケーションのみ適用した場合について比較する．Last-Touch 命令の実行回数は OS 側の方が 2 倍多く実行されている．これは，メッセージ領域として確保された固定長メモリブロックが 8 バイト (2 ワード) であるため，OS 側において，メモリブロックの返却時に Last-Touch Word 命令が 2 回実行されているためである．

しかしながら，OS 側で Last-Touch 命令が 2 倍多く実行されているにもかかわらず，消費電力の面ではアプリケーション側の 2 倍もしくはそれに近い電力削減効果が得られていない．通常，アプリケーション側での Self-Invalidation は，あるメモリブロックを最後に参照する命令が参照を終えた時点で適用される．それに対し OS 側では，メモリブロックを参照後，システムコールによるメモリブロックの返却時に Self-Invalidation を行う．このため，Last-Touch 命令を適用するタイミングがアプリケーション側よりも遅くなるため，本計測において，アプリケーション側と比較した場合に 2 倍の電力削減効果を得ることはできなかった．ただし，プログラムの実行時間が長く，Self-Invalidation された無効状態のキャッシュブロックが長く存在する場合には，この差が顕著となり充分な電力削減効果が得られるものと思われる．

また，実行サイクル数においても全体と比べて OS 側がやや増えている．これは，システムコールに Last-Touch 命令を実装する際に命令を余分に追加する形となり，結果として実行命令数が増えたためである．

ソフトウェア Self-Invalidation を OS に適用した場合において，実行サイクルを維持したまま電力を削減することはできなかったが，OS とアプリケーションの両方で電力削減効果が得られることが示された．また，OS とアプリケーション双方を組み合わせることにより，さらに電力削減効果が得られることが示された．

第6章　まとめ

最後に本研究におけるまとめを行い，今後の課題について述べる．

6.0.3　まとめ

本論文では，将来の組込みプロセッサが消費する電力の大部分を占めると想定されるキャッシュメモリに焦点を当て，キャッシュメモリの消費電力削減手法の一つであるソフトウェア Self-Invalidation を組込み OS に適用した．また，アプリケーション開発者による電力削減を支援する環境を提案した．

組込みアプリケーションにおいて一般に扱われると想定されるデータ処理を用いたタスクセットを作成し，提案手法を適用して評価を行った結果，L1 データキャッシュメモリにおいて，OS とアプリケーションの両方において消費電力が削減されていることを確認した．また，OS とアプリケーション双方を同時に適用して実行した場合においても充分に削減されていることを確認した．

6.0.4　今後の課題

今後の課題として以下の点を挙げる．

- **組込みアプリケーションでの検証**
  今回の計測では，組込みアプリケーションで一般に扱われるデータ処理を想定して作成したタスクセットを用いており，実際の組込みアプリケーションでの評価を行っていない．データ処理が複雑な実際のアプリケーション（例えば，組込み用データベースマネジメントシステムなど）に対して，本手法を適用した場合の消費電力について評価する必要がある．

- **キャッシュメモリに追加したタグ情報のハードウェア量の測定**
  本研究では，各キャッシュブロックのタグ情報に数ビットのラストタッチフラグを設けている．このハードウェア量を正確に測定し，システム全体の性能や消費電力について検証する必要がある．

- **プロセッサ全体の消費電力**
  今回は，L1 データキャッシュメモリに対する電力評価しか行っていない．提案手法を適用した場合のプロセッサ全体の消費電力について評価する必要がある．

謝辞

本研究を遂行するにあたり，終始熱心にご指導を頂いた田中清史准教授に心から深く感謝致しますと共に，ここに御礼申し上げます.

貴重な御意見，御助言を頂きました日比野靖教授，篠田陽一教授に深く感謝致します.

そして，本学における研究，生活において常に御意見，御助言を頂いた計算機アーキテクチャ講座の皆様に厚く御礼申し上げます.

最後に，日頃から温かく見守ってくださった家族，友人たちに深く感謝致します.

参考文献

[1] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T.N.Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In the Proceedings of the 2000 International Symposium on Low Power Electronics and Design, pp. 90-95, 2000.

[2] Stefanos Kaxiras and Zhigang Hu, Margaret Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In the Proceedings of the 28th Annual International Symposium on Computer Architecture, pp. 240-251, 2001.

[3] 藤田 剛憲. 自発的無効化によるキャッシュメモリの低消費電力化に関する研究, 北陸先端科学技術大学院大学修士論文 2007.

[4] (社) トロン協会. μITRON4.0 仕様 Ver.4.02.00. http://www.ertl.jp/ITRON/home-j.html.

[5] SPARC International,Inc. The SPARC Architecture Manual Version 8. Prentice-Hall,Inc.(1992).