修 士 論 文

キャッシュメモリの消費電力削減を目的とした自発的無効化命令の適用法に関する研究

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

山野 純嗣

2008 年 3 月

修 士 論 文

キャッシュメモリの消費電力削減を目的とした自発的無効化命令の適用法に関する研究

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　井口寧 准教授

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

0610089 山野 純嗣

提出年月: 2008 年 2 月

概 要

近年のマイクロプロセッサにおいて消費電力の増大が問題となっており，バッテリ駆動型の高性能なモバイル機器に大きな影響を与えている．これは，キャッシュメモリの容量が増大し，プロセッサ全体の大部分がキャッシュメモリによって占められていることに起因する．そのため，プロセッサのキャッシュメモリに注目し，消費電力削減を目的とした研究が行われている．

消費電力削減を目的としたソフトウェア Self-Invalidation[5] は，専用命令であるラストタッチメモリアクセス命令を用いてキャッシュメモリの電力供給を制御する．ラストタッチメモリアクセス命令は，通常のメモリアクセス命令に加えキャッシュメモリの電力供給を断つという機能を持つ．このラストタッチメモリアクセス命令を参照したキャッシュブロックが将来的に 1 度もアクセスされずに無効化されることが自明の場合に使用することで，消費電力を削減する．ソフトウェア Self-Invalidation は，プロセッサの性能低下を引き起こさずにキャッシュメモリの消費電力を削減することができるが，プログラムの事前実行が必要であるため実用性に乏しいという問題点がある．

本研究はこの問題点を解決するため，データの再利用情報 [6] を利用しコンパイル時にラストタッチメモリアクセス命令を埋め込むという手法を提案する．評価対象のプログラムとして SPLASH-2 ベンチマークを選択し，CPU シミュレータを用いて評価を行った結果，平均で 20.75%の消費電力が削減された．

目 次

第1章　はじめに

1.1　研究の背景

近年，半導体におけるプロセス技術の微細化が進み，プロセッサの動作周波数は飛躍的に向上した．しかしながら，トランジスタの微細化が進むにつれて，リーク電力が無視できないほどに大きくなり，プロセッサの消費電力は増大している．

一方で，バッテリで駆動するモバイル機器が増えている．特に，携帯電話やノートパソコンに代表される高性能なモバイル機器では，高い処理能力と長い駆動時間が要求されている．しかし，モバイル機器における処理能力や駆動時間はプロセッサの消費電力増大が問題となる．なぜなら，駆動時間は消費電力が単純に増えることによって短縮され，処理能力は電力の消費に伴う発熱が増えプロセッサの動作速度向上が困難になるからである．

よって，プロセッサの処理能力を落とさずに消費電力を削減することは重要な課題である．

1.2　研究の目的

近年では，プロセッサの速度向上に対して主記憶の速度向上はそれほど上がっていないため，プロセッサと主記憶の速度差が大きくなっている．この問題を解決するために，プロセッサと主記憶間にあるキャッシュメモリの容量が増大してきている．その結果，プロセッサにおけるキャッシュメモリの面積は増大し，キャッシュメモリにおける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ようになった．よって，キャッシュメモリにおける消費電力を削減することは重要な課題であり，近年，キャッシュメモリの電力削減を目的とした研究が行われている．

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 の間に高い閾値を持つトランジスタを設け，これをオフにすることで電力供給を断ち，リーク電流を削減している．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，対象キャッシュブロックに保持されているデータは消滅するという特徴を持っている．

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 では時間情報に基づいた gated-Vdd 制御を行っている．つまり，あるキャッシュブロックが使用されない時間が一定時間に達した場合，そのキャッシュブロックを今後使用しないと判断し gated-Vdd による電力供給制御を行うものである．

Dynamic Self-Invalidation[3] と Last-Touch Prediction[4] では，Self-Invalidation が提案された．Self-Invalidation とは，無効化されると予測されたキャッシュブロックをあらかじめ無効化するという考え方である．文献 [3] と [4] では Self-Invalidation を，マルチプロセッサにおけるキャッシュコヒーレント維持のためのオーバヘッド削減のために使用している．つまり，他のプロセッサからキャッシュブロックの無効化要求が来る前に，それらを予測しあらかじめ無効化するというものである．これらの制御は，ハードウェアによってテーブルなどを用意し実現する．

ソフトウェア Self-Invalidation[5] は，前述の Self-Invalidation をキャッシュメモリの消費電力削減のために応用したものである．しかし，文献 [3] や [4] で用いられるハードウェアを使用しての制御は消費電力削減目的には適さない．そのため，ソフトウェア Self-Invalidation において，Self-Invalidation を実現する制御はソフトウェアで行う．しかし，この手法では Self-Invalidation を適用する際に事前のプログラム実行が必要であり，また，その適用は手動であるため非効率的である．

本研究は，ソフトウェア Self-Invalidation を効率的に適用する手法を提案し，キャッシュメモリにおける消費電力を削減することを目的とする．

1.3 本論文の構成

本論文は全 6 章で構成される．

第 1 章 本章．

第 2 章 従来のキャッシュメモリにおける電力削減手法など，関連研究を紹介する．

第 3 章 ソフトウェア Self-Invalidation について説明する．

第 4 章 提案手法について説明する．

第 5 章 提案手法の評価を示す．

第 6 章 本研究の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，本研究の関連研究としてキャッシュメモリの消費電力削減法である gated-Vdd[1]，cache decay[2]，及びソフトウェア Self-Invalidation[5] 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 の間に高い閾値を持つトランジスタを設置する (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，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] においては，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．

キャッシュメモリの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ラストアクセスまでのアクセスが頻繁に発生する期間 (live time)，ラストアクセス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消費電力削減を図っている．

あるキャッシュブロックが dead time であるかどうかの判断は，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するということを利用している．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 中であるとみなされ，リプレースされるまでの間，供給電力がカットされる．したがって，この decay

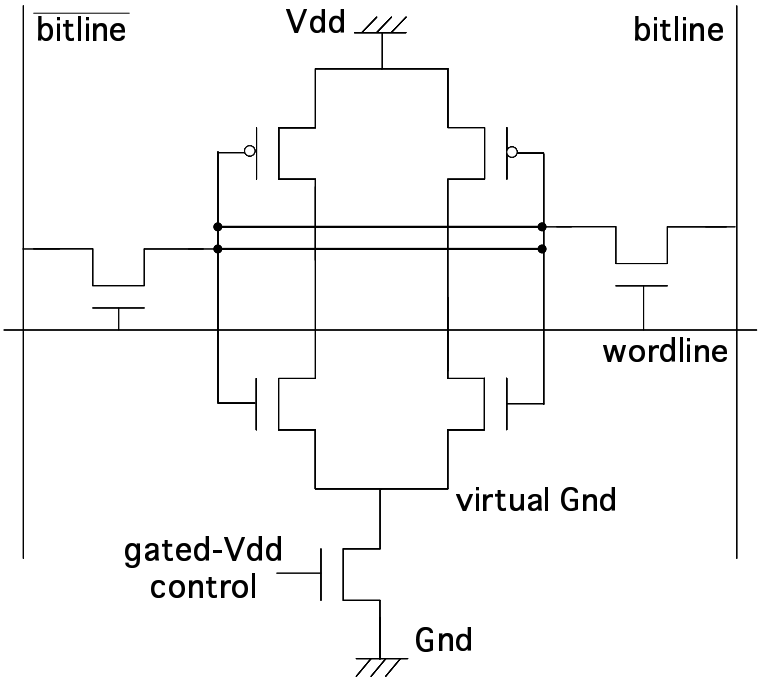

図 2.1: NMOS トランジスタを使用した SRAM セルの gated-Vdd 構成

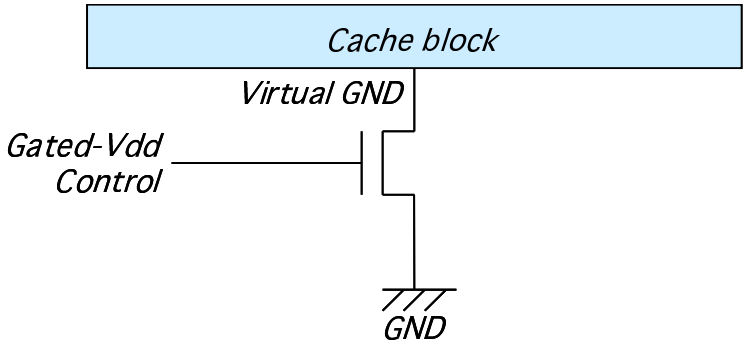

図 2.2: gated-Vdd を使用したキャッシュブロック単位での電圧制御

interval の設定を適切に行うことが重要となる．なぜなら，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．

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．

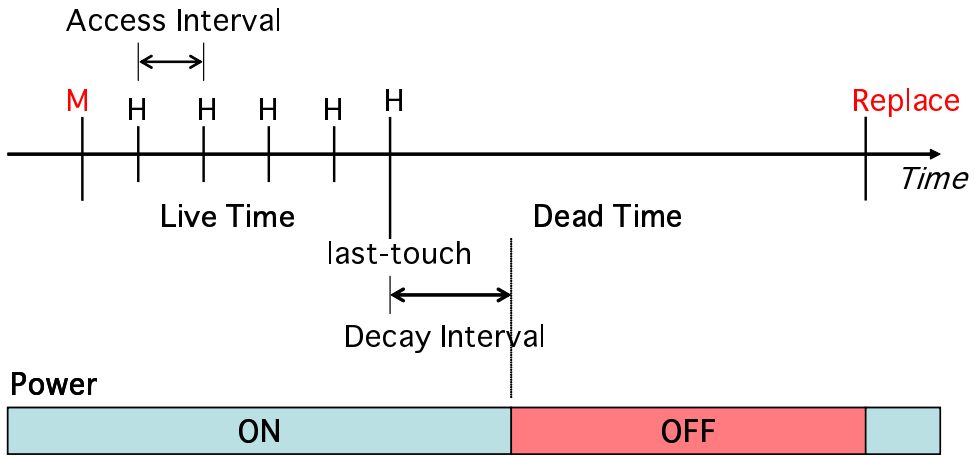

図 2.3: あるキャッシュブロックへのアクセスパターンと電力制御

2.3 ソフトウェア Self-Invalidation[5]

Dynamic Self-Invalidation[3] と Last-Touch Prediction[4] では，Self-Invalidation が提案された．

Self-Invalidation とは，無効化されると予測されたキャッシュブロックをあらかじめ無効化するという考え方である．文献 [3] と [4] では Self-Invalidation を，マルチプロセッサにおけるキャッシュコヒーレント維持のためのオーバヘッド削減のために使用している．つまり，ほかのプロセッサからキャッシュブロックへの無効化要求が来る前に，それを予測しあらかじめ該当キャッシュブロックを無効化する．これらの制御は，テーブルなどのハードウェアを用意することによって実現する．

ソフトウェア Self-Invalidation[5] は，Self-Invalidation をキャッシュメモリの消費電力削減のために応用した手法である．しかし，文献 [3] や [4] で用いられるハードウェアを使用しての制御は，消費電力削減を目的としたソフトウェア Self-Invalidation には適さない．なぜなら，追加するハードウェアが大きくなると，それ自体が消費する電力が増大するからである．よって，ソフトウェア Self-Invalidation において Self-Invalidation を実現する制御はソフトウェアで行う．これによって，追加するハードウェアはごくわずかに抑える

ことが可能となる．

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．

第3章　ソフトウェアSelf-Invalidation

本章では，Self-Invalidation をソフトウェアで行うソフトウェア Self-Invalidation について説明し，評価および問題点を述べる．

3.1　基本的な考え方

今後，使用されないと予測されたキャッシュブロックをあらかじめ無効化するという考え方が Self-Invalidation である．ソフトウェア Self-Invalidation は，Self-Invalidation をキャッシュメモリの消費電力削減を目的として応用したものである．例えば，将来的に無効化されるキャッシュブロックは，最後にアクセスされてから無効化されるまで必要ないにもかかわらずキャッシュメモリに残り続け，無駄な電力を消費することになる．このようなキャッシュブロックへの電力をソフトウェアによって制御し，消費電力を削減する．

具体的には，プログラム中のメモリアクセス命令のうち，メモリブロックのラストアクセスとなる命令を専用命令に置き換える．そして，その専用命令が実行時にキャッシュメモリの供給電力を gated-Vdd によって制御することで実現する．

3.2　ラストタッチメモリアクセス命令の導入

ソフトウェア Self-Invalidation では，gated-Vdd を制御する専用命令としてラストタッチメモリアクセス命令 (last-touch load/store) を命令セットに導入している．

3.2.1　ラストタッチメモリアクセス命令

ラストタッチメモリアクセス命令は，CPU に対しては通常のロード・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した際に，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．このラストタッチメモリアクセス命令をどのようにしてメモリアクセス命令と置き換えるかは，3.4 節にて説明する．

3.2.2 マルチワードブロックのキャッシュメモリへの対応

前述のラストタッチメモリアクセス命令だけでは，一般的なキャッシュメモリにおいては対応できない．なぜなら，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り，アクセス粒度はブロックサイズより小さいからである．

例えば，図 3.1 のようなブロックサイズが 4 ワードである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たとき，前述のラストタッチメモリアクセス命令を使用した場合，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．

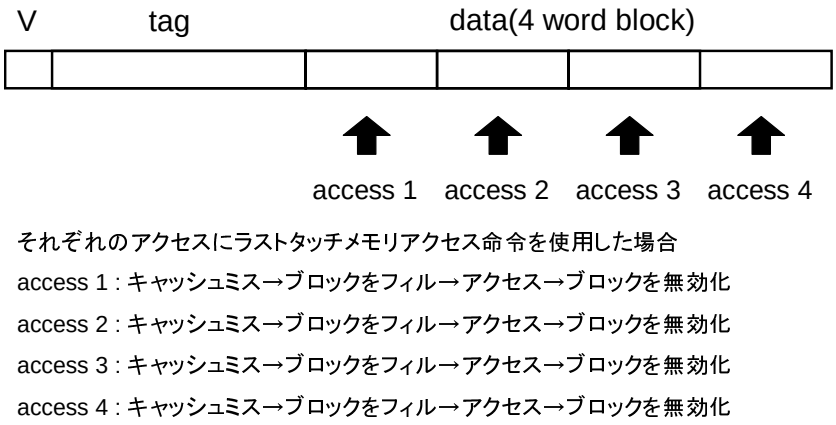

図 3.1: マルチワードブロックのキャッシュメモリでのラストタッチメモリアクセス命令

そこで，ラストタッチメモリアクセス命令を以下の 2 種類用意することで，マルチワードのキャッシュメモリに対応した柔軟な制御を可能とする．

- **ラストタッチブロックメモリアクセス命令(last-touch block load/store)**
  対象メモリブロックへのアクセスが完了した際に，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．

- **ラストタッチワードメモリアクセス命令(last-touch word load/store)**
  対象メモリブロックへのアクセスが完了した際に，アクセスしたキャッシュブロック内のワード位置にマークを付け，記録する．キャッシュブロック内のすべてのワード位置がこの命令によってマークされると，その該当するキャッシュブロックを無効化する．

図 3.1 における問題も，ラストタッチワードメモリアクセス命令を使用することで，過分なキャッシュミスを発生させることなく供給電力を遮断することができる．

3.3　キャッシュメモリの構成

キャッシュの電力供給は，2 種類のラストタッチメモリアクセス命令によって制御される．また，ラストタッチワードメモリアクセス命令の導入により，従来のキャッシュメモリの構造に少し変更を加える必要がある．

3.3.1　ラストタッチフラグビット

ラストタッチワードメモリアクセス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ことを記録するための領域が必要である．そのため，ラストタッチフラグビットをキャッシュメモリに用意する．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ラストタッチワードメモリアクセス命令での参照があったかどうかを記録する．

有効	ﾗｽﾄﾀｯﾁﾌﾗｸﾞ				タグ	データ			
0	1	1	1	1			ltb ld/st		
1	0	1	0	1		ltw ld/st		ltw ld/st	
1	1	1	1	1					
0	0	0	0	0		ltw ld/st	ltw ld/st	ltw ld/st	ltw ld/st
1	1	1	1	1					
.	.				.	.	.	.	.
.	.				.	.	.	.	.
.	.				.	.	.	.	.

図 3.2: キャッシュメモリの構成

図 3.2 にラストタッチワードメモリアクセス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 種類のラストタッチメモリアクセス命令の動作は以下のようになる．

- **ラストタッチブロックメモリアクセス命令の動作**
  ラストタッチブロックメモリアクセス命令によってアクセスされた場合，そのアクセスされたキャッシュブロックのどのワード位置であるかにかかわらず，そのキャッシュブロックの有効ビットはクリアされ，無効状態になる．

- **ラストタッチワードメモリアクセス命令の動作**
  ラストタッチワードメモリアクセス命令によってアクセスされた場合，キャッシュメモリの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，そのキャッシュブロックは無効状態になる．

3.3.2 供給電力制御

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．また，電力制御対象は，キャッシュメモリのデータ部のみである．これらの構成を図 3.3 に示す．

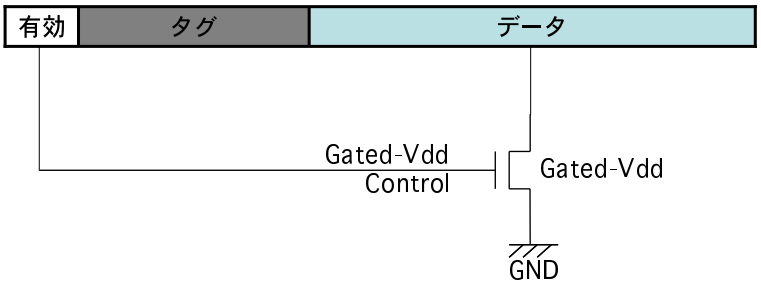

図 3.3: キャッシュメモリの供給電力制御

3.4 ラストタッチメモリアクセス命令の適用

ラストタッチメモリアクセス命令を適用するためには，メモリブロックへのラストアクセスとなるメモリアクセス命令を見つけ出さなくてはならない．もしそれが適切に行われなければ，キャッシュミスが増大し，性能低下を引き起こすことになる．また，用意した 2 種類のラストタッチメモリアクセス命令は状況によって使い分ける必要がある．

3.4.1 置換

あるメモリアクセス命令によって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なく無効化される場合，その命令をラストタッチメモリアクセス命令への置換対象とする．マルチプロセッサ環境において，この状況は 2 種類存在する．

- **他のプロセッサから無効化要求を受け取る場合**
  共有変数へのアクセスによってキャッシュメモリにコピーされたデータは，他のプロセッサが同じ領域に書き込みを行った場合に，そのプロセッサからの無効化要求を受け取ることになる (キャッシュコヒーレンシプロトコルはライトインバリデート方式であるとする)．このような場合，アクセス粒度に関係なくキャッシュブロック全体が無効化されるので，そのキャッシュブロックへの参照命令はラストタッチブロックメモリアクセス命令を使用する．例えば，図 3.4 の例において，共有変数 Globalid に対してのアクセスが条件を満たすならば，そのアクセス命令をラストタッチブロックメモリアクセス命令に置換する．

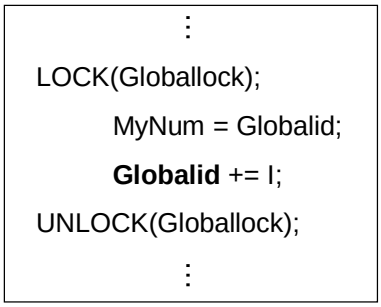

図 3.4: 他のプロセッサから無効化される例

- **自身が今後アクセスしないため無効化される場合**
  今後使用されないデータを最後に参照した場合，そのデータのコピーはキャッシュメモリに残り続けることになり無駄な電力を消費する．そこで，同一のプロセッサから今後アクセスされないデータへの参照はラストタッチメモリアクセス命令に置換する．しかし，図 3.5 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 度の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)．このような場合，ラストタッチブロックメモリアクセス命令を使用すると，ループ回数分のキャッシュミスが起こってしまう．よって，ラストタッチワードメモリアクセス命令を使用することでこの問題を回避し，Self-Invalidation を実現する．

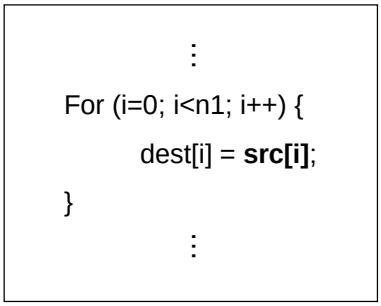

図 3.5: 自身が無効化する例

また，ラストタッチメモリアクセス命令への置換は，アセンブリコードを手動にて書き換えることによって実現する．

3.4.2 命令置換アルゴリズム

ソフトウェア Self-Invalidation では，メモリブロックへのラストアクセスとなるメモリアクセス命令を見つけるためにメモリアクセストレースを使用する．メモリアクセスト

レースとは，プログラムを 1 度実行し，プログラム中の全メモリアクセスを記録したものであり，以下の 2 種類がある．

- PC **ベーストレース**
  メモリ参照命令ごと (PC ごと) に，その命令が発行した全アドレス情報とアクセスした回数を記録したもの

- **アドレスベーストレース**
  参照されたメモリアドレスごとに，最後にアクセスした参照命令の PC アドレスを記録したもの

これら 2 種類のメモリアクセストレースを使用し，ラストアクセスとなる参照命令を探し出す．

3.5 評価

ここでは，ソフトウェア Self-Invalidation の評価環境，評価方法およびシミュレーション結果について述べる．

3.5.1 ベンチマークプログラム

評価は，マルチプロセッサ環境の性能を評価するために作成された SPLASH-2 ベンチマークプログラム [7] の中から以下の 5 つを使用して行う．

- RADIX
  基数ソート

- FFT
  高速フーリエ変換

- LU(contiguous block allocation)
  LU 分解 (隣接ブロック割当)

- LU(noncontiguous block allocation)
  LU 分解 (非隣接ブロック割当)

- CHOLESKY
  コレスキー分解

表 3.1: 各プログラムの入力データサイズ

プログラム	入力データサイズ
RADIX	262144 keys
FFT	65536 complex
LU(contiguous)	256×256 matrix
LU(noncontiguous)	256×256 matrix
CHOLESKY	wr10.O[1]

シミュレーションに使用したこれらの各プログラムの入力パラメータはCPUシミュレータに合わせ，プロセッサ数2，キャッシュメモリサイズ32KByteに設定した．また，並列計算の分割サイズはキャッシュブロックサイズである16Byteに設定している．各プログラムの入力データを表3.1に示す．

3.5.2 シミュレータ仕様

シミュレータの主な仕様は以下の通りである．

- SPARC V9命令セットアーキテクチャ[8]
- 2コアチップマルチプロセッサ
- キャッシュメモリ構成
  - 32KB L1命令キャッシュ
    - 16Byteブロック4ウェイセットアソシアティブ
  - 32KB L1データキャッシュ
    - 16Byteブロック4ウェイセットアソシアティブ

キャッシュメモリの置換アルゴリズムはLRUを使用する．また，このシミュレーションではL1キャッシュメモリより下位の階層を考慮せず，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．

キャッシュメモリへの書き込み方式はライトバック方式を使用する．また，十分に大きい容量を持つライトバッファを仮定することで，キャッシュメモリの制御を単純化している．

プロセッサ間のキャッシュメモリのコヒーレンシ維持には，スヌープ方式のライトインバリデートプロトコルを採用する．また，共有領域へのアクセスに伴う無効化要求は，1クロックサイクルで完了するものとする．

[1]CHOLESKYは入力ファイルを指定する必要があり，SPLASH-2に標準で用意されているものから選択した．

3.5.3 キャッシュメモリの消費電力計算

プログラム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 (3.1) で計算できる．

$$E_{leak} = E_{standby} \times \frac{t_s}{f_c} + E_{active} \times \frac{(t_e - t_s)}{f_c} \tag{3.1}$$

$E_{standby}$(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 3.2)．

表 3.2: 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 (3.2) で表現できる．

$$E_{block} = E_{leak} \times S_{block} \times 8 \tag{3.2}$$

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

3.5.4 シミュレーション結果

シミュレーションでは，以下の 3 種類の実行を比較することでソフトウェア Self-Invalidation における電力削減の評価を行う．

- **通常実行**
  キャッシュに対して電力制御を行わない実行
- **gated-Vdd 実行**
  有効ビットを gated-Vdd 制御に使用した実行

- **ソフトウェアSelf-Invalidation 実行**
  gated-Vdd 実行に加え，ソフトウェア Self-Invalidation による電力制御を行う実行

図 3.6 にこれらの実行における L1 データキャッシュメモリの消費電力をシミュレーションし，評価した結果を示す．これは，通常実行時の消費電力の割合を 100%とし，各プログラムごとにグラフ化したものである．

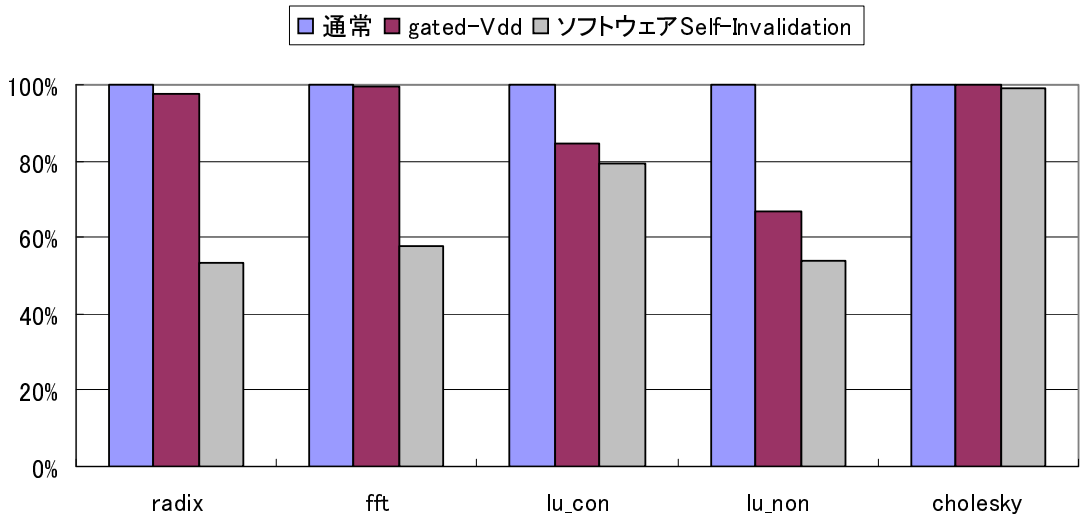

図 3.6: 各手法による消費電力

ソフトウェア Self-Invalidation 実行時には通常実行時の消費電力に比べ，平均 31.33%の消費電力を削減している．電力削減率が大きいプログラムでは，キャッシュブロックの無効化回数が多いという傾向が見られた．

消費電力以外の知見として，Self-Invalidation を行うことで，LRU のリプレース対象の選択が適切に行われ，キャッシュミスが減少するという副次的な効果があるということも得られた．また，キャッシュミスが減少することに伴い，実行クロックサイクルも減少している．

以上から，ソフトウェア Self-Invalidation は十分な消費電力削減効果が認められることがわかる．

3.6 問題点

ソフトウェア Self-Invalidation は，十分な消費電力削減を達成しているが，以下の問題点がある．

- **事前のプログラム実行が必要である**
  ラストタッチメモリアクセス命令への置換は，メモリアクセストレースを必要としているため，事前のプログラム実行が不可欠であり実用性に乏しい．

- **命令置換は手動である**
  ソフトウェア Self-Invalidation においての命令置換は，アセンブラコードの手動での書き換えであるため，非効率的であり生産性が低い．

以上の問題点を解決する手法として，ソフトウェア Self-Invalidation 適用法を提案する．これは，メモリアクセストレースを用いず，自動的にラストタッチメモリアクセス命令への置換を行うものである．ソフトウェア Self-Invalidation 適用法については次章で詳しく説明する．

第4章　提案手法

本章では，ソフトウェア Self-Invalidation の問題点を解消する手法であるソフトウェア Self-Invalidation 適用法について説明する．

4.1　ソフトウェア Self-Invalidation 適用法

ソフトウェア Self-Invalidation 適用法とは，ラストタッチメモリアクセス命令に置換することのできるメモリアクセス命令を予測し，自動的に置換する手法である．この手法によって，従来手法であるソフトウェア Self-Invalidation の非効率性および非実用性といった問題を解決できる．

さらに，どの段階において命令置換を行うかによって，ソフトウェア Self-Invalidation 適用法は以下の 2 つに類別して考えられる．

- **静的ソフトウェア Self-Invalidation 適用法**
  プログラムをコンパイルする際に置換することのできる命令を予測し置換する．
- **動的ソフトウェア Self-Invalidation 適用法**
  プログラムの実行時に置換することのできる命令を予測し置換する．

本研究では，静的ソフトウェア Self-Invalidation 適用法について評価を行い，従来手法であるソフトウェア Self-Invalidation の評価結果と同等の効果を得ることを目的としている．

4.2　命令置換予測

静的ソフトウェア Self-Invalidation 適用法において用いるラストタッチメモリアクセス命令に置換可能かどうかの予測は，データの再利用情報 [6] を有効に利用したものである．以下，その再利用情報を活用した EM ビットの使用方法や予測アルゴリズムを説明する．

4.2.1　EM ビット

文献 [6] では，セットアソシアティブキャッシュにおける LRU リプレースメントの選択を改善するために，EM(Evict Me) ビットをキャッシュメモリに導入している．これは，LRU

のリプレースメント対象を選択する際に優先的にリプレース対象になるフラグである．このフラグを参照の際にセットすることで，キャッシュメモリからその参照データが追い出される可能性が高くなる．つまり，今後再利用されないデータの参照の際にEM ビットをセットすることで，LRU の性能を改善しキャッシュヒット率を向上させることができる．また，このEM ビットによってLRU の性能が下がることはないことが文献 [6] にて証明されている．

4.2.2 データの再利用性

キャッシュメモリは，プログラムにおけるデータ参照の局所性を利用してデータの転送効率を向上している．したがって，その局所性に注目することでキャッシュメモリ上のデータが再利用されるかどうかという再利用性をある程度予測することが可能となる．局所性には時間的局所性と空間的局所性が存在するため，キャッシュメモリ上のデータの再利用性は，以下の 2 種類が存在する．

- **時間的局所性を利用したデータの再利用性**
  ある参照命令 A が参照するデータが，後にキャッシュメモリにおける時間的局所性を利用して別の参照命令によって参照される場合，参照命令 A は時間的局所性を利用したデータの再利用性を持つという．

- **空間的局所性を利用したデータの再利用性**
  ある参照命令 A が参照するデータが，後にキャッシュメモリにおける空間的局所性を利用して別の参照命令によって参照される場合，参照命令 A は空間的局所性を利用したデータの再利用性を持つという．

この再利用性の情報と EM ビットを組み合わせることで，文献 [6] ではLRU リプレースメント選択の性能改善を達成している．

4.2.3 予測アルゴリズム

文献 [6] では，プログラムコード中のループ内における配列型データ参照に注目し，その参照データが今後再利用されるかどうかを予測するという手法を提案している．つまり，再利用されないと予測されたデータへの参照には，その参照の際にEM ビットをセットする．また，そのデータの再利用性を予測できない場合にも，そのデータへの参照時にEM ビットをセットする．

このEM ビットをセットする予測アルゴリズムを図 4.1 に示す (以下このアルゴリズムを Wang の予測アルゴリズムとする)．この Wang の予測アルゴリズムは，まず，プログラム中のすべてのループに対してデータボリュームの見積もりを行う．データボリュームとは，そのループの実行開始から実行終了までに使用されるデータの総量であり，データ

セットとも呼ばれる．そしてそのデータボリュームの見積もりは，基本的には 1 ループで使用されるデータ量とループ回数の積で求められる．しかし，プログラムコード内のループにおいて，ループ回数が必ずしも把握できるわけではなく，正確なデータボリュームを求めることは困難な場合が多い．そこで，この Wang の予測アルゴリズムは，ループのデータボリュームが不明な場合は，キャッシュメモリの容量と同等のデータボリュームであると仮定する．こうすることで，データの再利用性が予測できない場合にも EM ビットをセットするということを実現している．

次に，すべてのループ内の配列参照 a に対して以下の処理を行う．配列参照 a がそのループ内に時間的・空間的局所性を利用したデータの再利用性を持つかどうかを調べる．もし，再利用性を持っている場合，配列参照 a にて参照されるデータは今後も必要であるということなので EM ビットをセットする参照にはならない．再利用性を持っていない場合は，以降のループで同じ配列参照 a が存在するかどうかを調べる．存在しなければ，配列参照 a は今後再利用されないと判断され，参照の際に EM ビットをセットする．存在する場合にも，配列参照 a が存在するループと次の配列参照 a が存在するループ間のデータボリュームがキャッシュメモリサイズより大きければ，EM ビットをセットする．これはつまり，配列参照 a が参照したデータのキャッシュコピーは，次の配列参照 a が使用するときにはキャッシュメモリから追い出されていると考えられるからである．

本研究では，この Wang の予測アルゴリズムを応用し，ラストタッチメモリアクセス命令の置換に使用する．

4.3 命令置換方法

静的ソフトウェア Self-Invalidation 適用法は，ラストタッチメモリアクセス命令の置換をコンパイル時に行う．プログラム構造から参照傾向を解析し，ラストタッチメモリアクセス命令を埋め込んだ実行バイナリを生成する．

4.3.1 本手法において使用する置換アルゴリズム

本研究では，ラストタッチメモリアクセス命令の置換には前述の Wang の予測アルゴリズム (図 4.1) を用いる．つまり，再利用されないと予測されたキャッシュブロックの電力供給をラストタッチメモリアクセス命令を使用し，カットする．

しかし，この Wang の予測アルゴリズムをそのまま使用すると問題が生じる．それは，対象とするキャッシュブロックを無効化するタイミングが，EM ビットとラストタッチメモリアクセス命令では違うからである．以下にそれぞれの手法による無効化タイミングの差異を示す．

- **EM ビット**
  該当キャッシュブロック参照後，キャッシュメモリの容量性ミスあるいは競合性ミスにより，LRU によってリプレースが行われる際にその該当ブロックを無効化する．

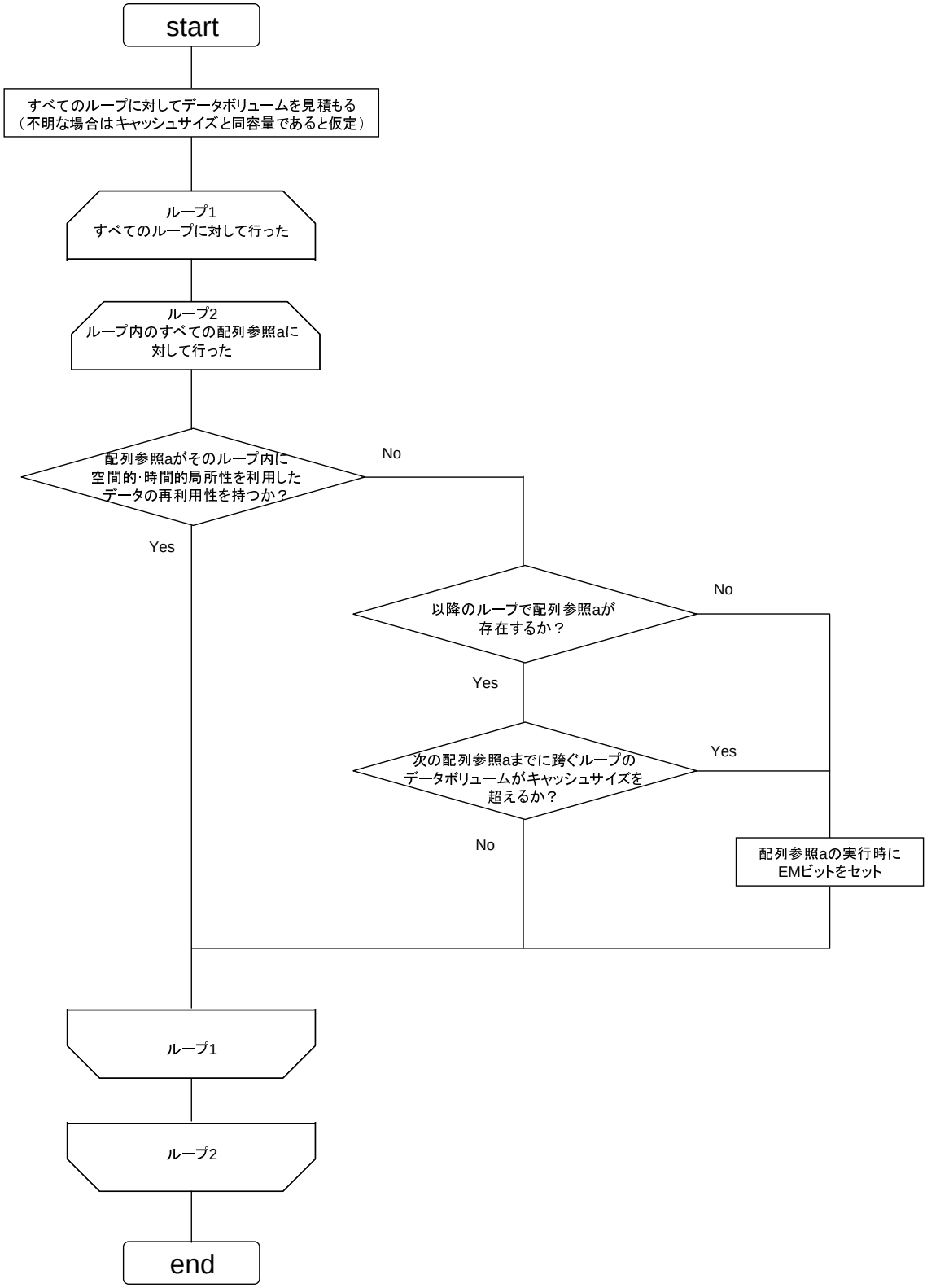

図 4.1: EM ビットをセットするアルゴリズム

- **ラストタッチメモリアクセス命令**
  該当キャッシュブロックのLast-Touch flagが成立する参照が実行された直後，その該当ブロックを無効化する．

EMビットは，LRUリプレース選択の性能改善のための手法であり，EMビットがセットされたからといって即座にデータがキャッシュメモリ上から追い出されるわけではない．そのため，Wangの予測アルゴリズムによって本来は再利用されるデータへの参照が再利用されないと間違った予測がされEMビットがセットされた場合においても，そのデータが局所性を利用して再利用される際にキャッシュメモリ上に残っている可能性が高い．

一方，ラストタッチメモリアクセス命令は消費電力削減を目的としているため，参照が行われた直後にデータを無効化し，キャッシュメモリ上から消去する．よって，Wangの予測アルゴリズムが外れ，再利用性がある参照に対してラストタッチメモリアクセス命令を適用した場合，キャッシュミスが発生し性能低下を引き起こすことになる．

つまり，ラストタッチメモリアクセス命令を用いた本手法においては，Wangの予測アルゴリズムの曖昧性を許容することはできないので，より厳密な予測が必要になる．したがって，Wangの予測アルゴリズムを適用する際に，再利用されないということが確定的なデータの参照のみに限定してラストタッチメモリアクセス命令に置換することが必要である．

具体的には，Wangの予測アルゴリズムの曖昧性の原因であるループのデータボリュームが不明な場合の見積もりを変更する．Wangの予測アルゴリズムにおいては，データボリュームが不明な場合はキャッシュメモリサイズと同容量であると仮定していたが，これをサイズ0であると仮定することで曖昧性を排除し，この問題を解決する．

また，本手法におけるラストタッチメモリアクセス命令は，ラストタッチワードメモリアクセス命令に限定して適用する．これによって，Wangの予測アルゴリズムにおける空間的局所性を利用した再利用性を考慮する必要がなくなり，置換アルゴリズムの簡素化が図れる．

最終的に使用する置換アルゴリズムを図4.2に示す．

4.3.2 置換手法

本研究では，ラストタッチメモリアクセス命令の置換アルゴリズムをコンパイラに組み込むことで置換を行う．これによって，従来手法における非効率な手動での置換を自動化し，効率の向上を実現する．

4.4 実装

提案手法は，4.3節にて説明した置換アルゴリズム (図4.2) を組み込んだコンパイラを作成することで実装する．なお，対象プログラムコードはC言語とし，コンパイラはyaccとlex，およびgccを使用して作成する．

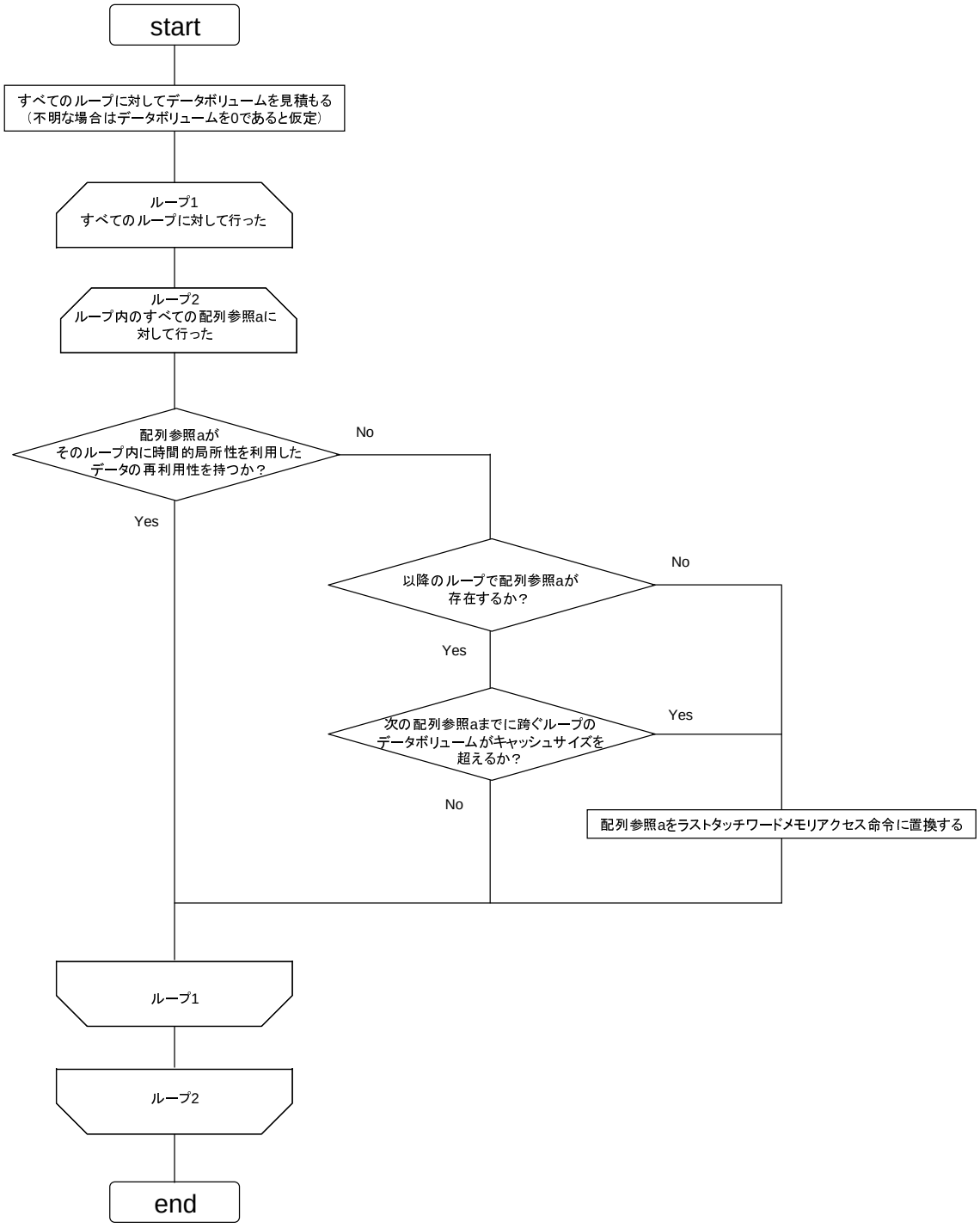

図 4.2: ラストタッチメモリアクセス命令への置換アルゴリズム

4.4.1 ループのデータボリュームの見積もり

プログラムコードにおけるループのデータボリュームの見積もりを説明する．データボリュームの見積もりの際に必要になるのは，誘導変数の初期値，継続条件，インクリメント値，ループ内の配列型参照の数，およびその型宣言である．しかし，実際のプログラムコードではそれらが明確に特定できることは少なく，ループのデータボリュームが正確に計算できないことが多い．例えばfor文によるループでは，誘導変数の初期値として変数が設定されていたり，継続条件の比較対象が変数だった場合，その変数の中にどのような値が代入されているかわからず，ループ回数が不明となるからである．

そこで，そのような場合には対象ループからプログラムの先頭に向かって，その変数にどのような値が入っているかを探索する．値探索は単純な四則演算による代入にのみ限定し，その変数にどのような値が入っているかを特定する．もし，関数の返り値が代入されているなどの複雑な場合は，探索を中止しループのデータボリュームは不明とする．図4.3にfor文のデータボリュームの計算例を示す．

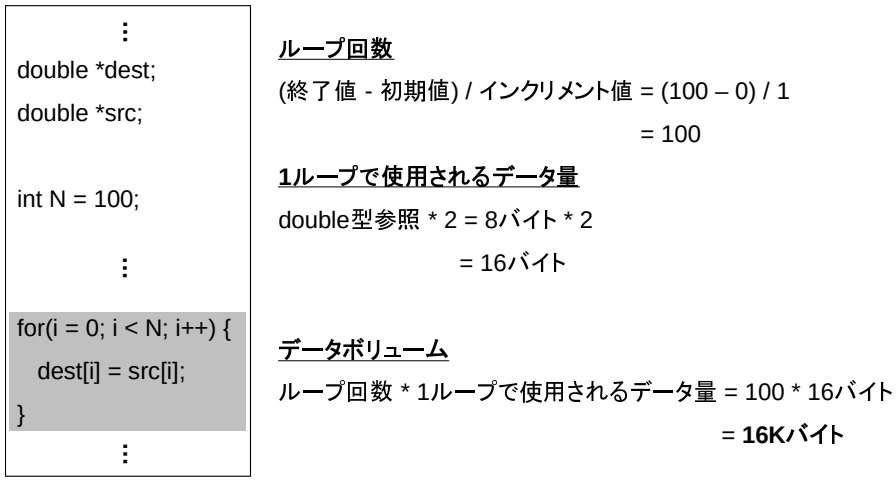

図 4.3: for文によるループのデータボリュームの計算例

4.4.2 ラストタッチメモリアクセス命令への置換方法

ラストタッチメモリアクセス命令への置換は，コードに直接アセンブリを埋め込むことで行う．本研究では，専用命令であるラストタッチメモリアクセス命令としてSPARC V9 命令セットのASI(Address Space Identifier)フィールドを持ったロード・ストア命令を使用する．そして，yaccとlexによってその命令を”asm volatile”によってコード上に埋め込み，gccを使用し実行バイナリを生成する．

しかし，この方法では適用する参照がロードであるかストア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，配列型参照は代入文によって実

行されることに注目する．その配列型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，左式に置換対象の配列型参照が存在するならば，ストアを実行するラストタッチメモリアクセス命令に置換する．置換対象の配列型参照が左式に存在する場合のラストタッチメモリアクセス命令の置換例を図 4.4 に示す．

```
for(i = 0; i < N; i++) {
   target_array[i] = src[i];
}
```

target_array[i]にラストタッチメモリアクセス命令を適用

```
for(i = 0; i < N; i++) {
   asm volatile("sta   %0, [%1] 0x3c" : : "r" (src[i]), "r" (&target_array[i]));
}
```

図 4.4: 置換対象の配列型参照が左式に存在する場合の置換例

逆に右式に置換対象の配列型参照が存在するならばロードを実行するラストタッチメモリアクセス命令に置換する．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ラストタッチメモリアクセス命令の置換例を図 4.5 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] = target_array[i];
}
```

target_array[i]にラストタッチメモリアクセス命令を適用

```
for(i = 0; i < N; i++) {
   {
   int temp;
   asm volatile("lda   [%1] 0x3c, %0" : "=r" (temp) : "r" (&target_array[i]));
   dest[i] = temp;
   }
}
```

図 4.5: 置換対象の配列型参照が右式に存在する場合の置換例

第5章　評価

本章では，提案手法である静的ソフトウェア Self-Invalidation 適用法のシミュレーションによる評価について述べる．

5.1　評価環境

従来手法であるソフトウェア Self-Invalidation と提案手法の評価の比較を容易にするため，評価環境は従来手法にて使用している環境と同等のものを使用する．

5.1.1　ベンチマークプログラム

3.5.1節と同様に，SPLASH-2ベンチマークプログラムから，RADIX，FFT，LU(contiguous)，LU(noncontiguous)，CHOLESKY の 5 つのプログラムを使用する．

5.1.2　シミュレータ仕様

3.5.2 節と同様の仕様にて作成したシミュレータによって提案手法の消費電力削減効果を評価する．

5.1.3　キャッシュメモリの消費電力計算

3.5.3 節と同様の方法にて，L1 データキャッシュの消費電力量を見積もる．

5.2　シミュレーション結果

シミュレーションでは，以下の 4 種類の実行を評価する．

- **通常実行**
  キャッシュに対して電力制御を行わない実行
- gated-Vdd **実行**
  有効ビットを gated-Vdd 制御に使用した実行

- **ソフトウェアSelf-Invalidation 実行**
  gated-Vdd 実行に加え，ソフトウェア Self-Invalidation による電力制御を行う実行

- **静的ソフトウェアSelf-Invalidation 実行**
  gated-Vdd 実行に加え，静的ソフトウェア Self-Invalidation 適用法による電力制御を行う実行

これらの実行を消費電力，キャッシュミス数，Self-Invalidation 回数，実行時間，命令置換数の観点から比較し，提案手法を評価する．

5.2.1 消費電力

図 5.1 にシミュレーションによって L1 データキャッシュメモリの消費電力を評価した結果を示す．これは，通常実行時の消費電力の割合を 100%として，各プログラムごとのそれぞれの手法における消費電力をグラフ化したものである．

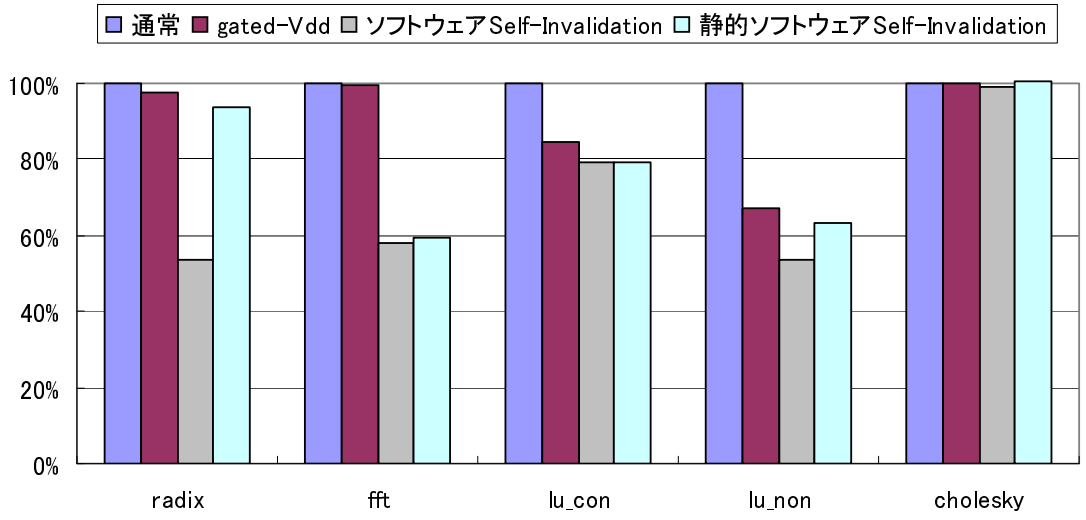

図 5.1: 各手法による消費電力

以下，提案手法に関してベンチマークプログラムごとの考察を示す．

- RADIX
  従来手法における電力削減効果に及んでいないが，gated-Vdd 実行時に比べ電力削減率が約 5%高いという結果が得られた．

- FFT
  従来手法における電力削減効果とほぼ同等の結果が得られた．

- LU(contiguous)
  従来手法における電力削減効果とほぼ同等の結果が得られた.

- LU(noncontiguous)
  従来手法における電力削減効果に及んでいないが，gated-Vdd 実行時に比べ電力削減率が約 6%高いという結果が得られた.

- CHOLESKY
  従来手法における電力削減効果に及ばず，gated-Vdd 実行時とほぼ同等の消費電力評価が得られた.

評価に使用した 5 つのプログラム中，FFT と LU(contiguous) の 2 つのプログラムで従来手法とほぼ同じ電力削減効果が得られた．この 2 つのプログラムに関しては，ラストタッチメモリアクセス命令が効率的に適用され理想的な消費電力削減効果を発揮していることがわかる.

また，RADIX と LU(noncontiguous) の 2 つのプログラムでは従来手法の電力削減効果に及ばないものの，gated-Vdd 実行の電力削減効果を上回っており，提案手法が電力削減に効果を持っていることを示している.

最後に，CHOLESKY は従来手法においてもっとも電力削減が難しいプログラムである．これは，提案手法においても同様であり，さらに，gated-Vdd 実行時より電力削減率が悪いという評価が得られた．しかし，これは無視できる範囲の値であり，gated-Vdd 実行や従来手法と同等の電力削減率であるといえる.

従来手法におけるこれら 5 つのベンチマークプログラムの平均電力削減率は 31.33%である．提案手法における平均電力削減率は 20.75%であり，これは従来手法の平均電力削減率に及ばないものの十分な効果が認められる.

5.2.2 命令置換数

ここでは，従来手法と提案手法においてラストタッチメモリアクセス命令に置換するメモリアクセス命令の数を比較する．表 5.1 に，各プログラムにおける従来手法，提案手法のラストタッチメモリアクセス命令置換数を示す.

表 5.1: 命令置換数比較

プログラム	従来手法	提案手法
RADIX	33	8
FFT	34	4
LU(contiguous)	31	3
LU(noncontiguous)	33	3
CHOLESKY	118	12

5つのプログラムすべてにおいて，提案手法は従来手法に比べラストタッチメモリアクセス命令置換数が減少している．

しかし，命令置換数が減少しているにもかかわらず，消費電力の観点から見ると，特にFFT，LU(contiguous) に関しては電力削減率がほぼ同じであり，直感的に矛盾する．これは，従来手法ではループ内の参照とループ外の参照を混合したメモリアクセストレースを用いているため，冗長な命令置換が行われているからである．例えば，ループ外に存在するある参照命令に対してラストタッチメモリアクセス命令を適用した場合，その参照命令はプログラム実行時に1度しか実行されないことが多い．そのようなデータ参照に対してラストタッチメモリアクセス命令を適用しても，効果がほとんど見られないということは明らかである．

また，従来手法においてあるループ外の参照命令が存在するために，それに対応するループ内の参照命令をラストタッチメモリアクセス命令に置換できないという場合も発生する．一般的にループ外の参照命令は1度だけしか参照を行わないということに対して，ループ内の参照命令は複数の参照を行う．そのため，ループ内の参照命令は複数の参照を行うと同時に，大量のデータのキャッシュコピーを生成する．従来手法では，ループ内参照命令の発行する参照群の中に後から実行されるループ外参照命令の発行する参照が重複していた場合，ループ内の参照命令はラストタッチメモリアクセス命令に置換しない．本来，ループ内の参照命令にラストタッチメモリアクセス命令を適用し，データを消去しても，そのデータを参照するのはループ外の参照命令，つまり1度だけの参照であり，この参照によって発生するのはキャッシュミス1回と，キャッシュフィルの少しのオーバヘッドだけである．よって，この場合，消費電力削減効果の観点からループ内参照命令をラストタッチメモリアクセス命令に置換すべきであるということは自明である．

提案手法においては，消費電力に支配的なループ内の配列型参照に注目し，ラストタッチメモリアクセス命令を適用する．そのため，効率的な命令置換を行うことができ，ラストタッチメモリアクセス命令数が少なくとも高い消費電力削減効果を得ることができる．

5.2.3 Self-Invalidation 回数

ここでは，キャッシュブロックがラストタッチメモリアクセス命令によって無効化された回数，つまり Self-Invalidation 回数について比較する．表 5.2 に，各プログラムにおける従来手法，提案手法の Self-Invalidation 回数を示す．

提案手法の RADIX における Self-Invalidation 回数は，従来手法における Self-Invalidation 回数とほぼ同じである．しかし，消費電力の削減という視点から見ると，提案手法の消費電力削減効果は従来手法に及んでいない．また，LU(contiguous) では，提案手法の Self-Invalidation 回数が従来手法の約 2 倍になっているが，消費電力削減効果はほぼ同じである．

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．よって，Self-Invalidation 実行回数が消費電力効果と比例するわけではない．例えば，Self-

表 5.2: Self-Invalidation 回数比較

プログラム	従来手法	提案手法
RADIX	272420	265706
FFT	131580	65536
LU(contiguous)	82150	163688
LU(noncontiguous)	157156	68340
CHOLESKY	14839	10105

Invalidation が成立し実行されたとしても，その直後に他のデータがそのキャッシュブロックを使用した場合，消費電力削減効果は低くなる．しかし，その場合はキャッシュメモリを有効に利用することができる可能性がある．

5.2.4 キャッシュミス数

表 5.3 に，通常実行，従来手法及び提案手法におけるキャッシュミス数を示す．

表 5.3: キャッシュミス数比較

プログラム	通常実行	従来手法	提案手法
RADIX	523157	498436	397376
FFT	920109	920097	920105
LU(contiguous)	577608	577489	577640
LU(noncontiguous)	2077916	2015857	2075325
CHOLESKY	482915	481376	492571

提案手法のキャッシュミス数について，大きな差異が認められるのは RADIX である．従来手法が通常実行と比べ，約 5%のキャッシュミス数を減らしているのに対して，提案手法は約 25%のキャッシュミス数を削減している．

キャッシュミス数が減少する理由としては，以下のことが考えられる．キャッシュメモリのリプレースアルゴリズムである LRU は，アクセス間隔を尺度として，最も長時間使用されなかったデータをリプレース対象として選択する．そのため，今後再利用されるデータがリプレース対象として選択され，追い出されてしてしまう可能性がある．しかし，提案手法ではラストタッチメモリアクセス命令を使用し，再利用されないデータを無効化することでその可能性を下げることができる．したがって，提案手法においてはキャッシュメモリを有効に使用することができ，キャッシュミス数が減少する．

RADIX 以外のプログラムでは，提案手法のキャッシュミスは通常実行と比べ増えているものもあるが，その増大率は最大でも約 2.0%程度である．これは無視できる範囲の値であり，これらのプログラムにおける提案手法のキャッシュミス数は，通常実行とほぼ同じであるといえる．

よって，提案手法では消費電力を削減するだけでなくキャッシュミス数を減らすという効果も認められる．

5.2.5 実行時間

表5.4に，通常実行，従来手法及び提案手法におけるプログラムの実行終了までにかかったクロックサイクル数を示す．

表 5.4: 実行時間比較

プログラム	通常実行	従来手法	提案手法
RADIX	122541690	122418679	121121148
FFT	118641649	118707110	119067652
LU(contiguous)	100084316	100084040	100281836
LU(noncontiguous)	98001681	97693117	97989311
CHOLESKY	39266223	39258305	39486068

RADIXに関しては，キャッシュミスの大幅な減少に伴って通常実行に比べて実行時間が約1%ほど減少している．FFT，LU(contiguous)，CHOLESKYでは提案手法の実行時間が通常実行より増えている．

提案手法の実行時間が通常実行に比べ増えているのは，本研究における提案手法の実装方法によるものである．本研究では，ラストタッチメモリアクセス命令としてSPARC V9命令セットからASIフィールドを持ったロード・ストア命令を使用している．そのため，通常のロード・ストア命令よりもラストタッチメモリアクセス命令の命令フィールドの制限が多い．例えば，ASIフィールドを持つロード・ストア命令は，対象メモリアドレスをレジスタでしか指定できない．そのため，ラストタッチメモリアクセス命令を適用すると通常よりも命令数が数命令増加する．さらに，提案手法ではループに注目し，ラストタッチメモリアクセス命令を適用するため，その増加した命令が多く実行されるため顕在化している．これが，提案手法の実行時間が通常実行より増加する理由である．しかし，その増大率は最大でも0.56%と無視できる大きさである．

よって，提案手法の実行時間は通常実行の実行時間と同じであると言え，提案手法によって発生するオーバヘッドはほとんどないと言える．

第6章　まとめ

最後に本研究における提案手法のまとめと，今後の課題について述べる．

6.1　まとめ

本論文では，プロセッサの大部分を占めるキャッシュメモリをターゲットとし，プロセッサの消費電力削減手法であるソフトウェア Self-Invalidation の適用に関する研究について述べた．従来手法であるソフトウェア Self-Invalidation について説明し，その問題点を説明した．従来手法の問題点を解決し，かつ同等の消費電力削減効果を得ることを目的とした静的ソフトウェア Self-Invalidation 適用法を提案した．静的ソフトウェア Self-Invalidation 適用法にて使用するデータの再利用情報を活用した置換アルゴリズムについて説明した．

SPLASH-2 ベンチマークプログラムの RADIX, FFT, LU(contiguous), LU(noncontiguous), CHOLESKY の 5 つのプログラムを使用し提案手法を評価した結果，L1 データキャッシュメモリの消費電力を平均で 20.75%削減することができた．これは，従来手法の平均消費電力削減率の 31.33%には及ばないものの，十分な消費電力削減効果が得られた．また，キャッシュミスを削減し，プログラム実行時間が短縮されるという副次的な効果も得られた．

6.2　今後の課題

今後の課題として以下の点を挙げる．

- **シングルプロセッサ環境での提案手法の適用とその効果の検証**
  本研究で提案したソフトウェア Self-Invlaidation 適用法は，マルチプロセッサ環境だけでなく，シングルプロセッサ環境にも応用可能である．その場合，cache decay[2] に代表されるほかのキャッシュメモリ低消費電力化技法との比較，組み合わせなどを検証することで，最適なキャッシュメモリの消費電力削減法が実現可能であると考えられる．

- **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**
  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測定し，コストや消費電力などを考慮したシステム全体の検証が必要である．

- **プロセッサ全体の消費電力**
  今回は，提案手法を適用したシステムの L1 データキャッシュメモリに対する消費電力評価しか行っていない．したがって，提案手法を適用したプロセッサ全体の消費電力について厳密に評価する必要がある．

- **動的ソフトウェア** Self-Invalidation **適用法の提案と検証**
  ソフトウェア Self-Invalidation 適用法には，静的ソフトウェア Self-Invalidation 適用法と動的ソフトウェア Self-Invalidation 適用法がある．今回はコンパイル時に命令を置換する静的ソフトウェア Self-Invalidation 適用法の消費電力の検証を行った．プログラムの実行時の情報を利用する動的 Self-Invalidation 適用法は，静的ソフトウェア Self-Invalidation 適用法に比べ，命令置換を予測する際の判断材料が多いと考えられ，より精度の高い命令置換が可能であると考えられる．また，静的ソフトウェア Self-Invalidation 適用法と動的ソフトウェア Self-Invalidation 適用法は，適用段階が独立しているため共存することが可能である．よって，動的ソフトウェア Self-Invalidation 適用法の提案と検証および静的ソフトウェア Self-Invalidation 適用法と動的ソフトウェア Self-Invalidation 適用法を同時に使用した場合の検証が必要である．

謝辞

本研究を遂行するにあたり，終始熱心にご指導してくださった田中清史准教授に心から深く感謝すると共に，ここに御礼申し上げます.

適切なご意見，ご助言をいただいた日比野靖教授，井口准教授に深く感謝いたします.

その他，貴重なご意見，討論をいただきました田中研究室の皆様をはじめとする多くの方に対して厚く御礼申し上げます.

最後に，日ごろから温かく見守ってくださった両親，友人たちに深く感謝いたします.

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] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In the Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 48-59, 1995.

[4] An-Chow Lai and Babak Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In the Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 139-148, 2000.

[5] 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.

[6] Z.Wang, K.S.McKinley, A.L.Rosenberg. Improving Replacement Decisions in Set-Associative Caches. Proc. of MASPLAS'01, The Mid-Atlantic Student Workshop on Programming Languages and Systems, Hawthorne, NY, Apr. 2001.

[7] Stevan Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In the Proceedings of the 22nd Annual International Symposium on Computer Architecturethe, pp. 24-36, 1995.

[8] SPARC International, Inc. The SPARC Architecture Mannual Version 9. July, 2003.