組込みシステム設計のための正確かつ高速なキャッシュ構成シミュレーションに関する研究

Exact and Fast Cache Configuration Simulation for Embedded Systems

2015年2月

早稲田大学大学院 基幹理工学研究科

情報理工学専攻 情報システム設計研究

多和田　雅師

Masashi TAWADA

目 次

1 序論 **1**

1.1 本論文の背景 2

1.2 本論文の目的 6

1.3 本論文の概要 10

2 関連研究 **12**

2.1 本章の概要 13

2.2 キャッシュメモリとキャッシュ構成シミューション 13

2.3 Janapsatya らの手法 16

2.4 CRCB (Configuration Reduction approach by the Cache Behavior) 手法 . . . 23

2.5 CRCB-U 手法 23

2.6 Haque らの手法 26

2.7 本章のまとめ 27

3 柔軟なリプレースメントポリシをベースとするキャッシュの高速キャッシュ構成シミュレーション **28**

3.1 本章の概要 29

3.2 CRCB 手法が適用可能なアーキテクチャ 29

3.3 キャッシュセットの動作が同じキャッシュ構成を統合するデータ構造 33

3.3.1 FIFO をキャッシュリプレースメントポリシとする L1 キャッシュメモリの性質 33

3.3.2 PLRU をキャッシュリプレースメントポリシとする L1 キャッシュメモリの性質 36

3.4 高速化アルゴリズムの提案 38

3.5 提案手法の評価 40
3.6 本章のまとめ 41

4 FIFO ベースキャッシュの高速キャッシュ構成シミュレーション 44

4.1 本章の概要 45
4.2 データ構造の更新不要なキャッシュ操作を考慮した複数キャッシュ構成を統合するデータ構造 45
4.3 高速化アルゴリズムの提案 46
4.4 提案手法の評価 49
4.5 本章のまとめ 51

5 マルチコアプロセッサアーキテクチャにおける高速キャッシュ構成シミュレーション 52

5.1 本章の概要 53
5.2 キャッシュコヒーレンシ 53
5.3 マルチコアプロセッサアーキテクチャにおけるキャッシュ動作の分類 55
5.4 キャッシュコヒーレンシを考慮した複数キャッシュ構成を統合するデータ構造 58
5.5 高速化アルゴリズムの提案 61
5.6 提案手法の評価 70
5.7 本章のまとめ 70

6 結論 71

謝辞 75

参考文献 76

研究業績 84

第１章

序論

1.1 本論文の背景

近年，半導体市場の規模は大きくなっている．表1.1に年代と半導体市場の規模の関係を示す [1]．2013年で半導体市場は3000億ドルを超える規模である．2010年から2013年まで半導体市場は成長し，中でも特に光電子工学とセンサーの市場が成長している．また，半導体市場の統計機関WSTS (World Semiconductor Trade Statistics) は2016年まで半導体市場は成長を続けると予測している [2]．半導体市場は巨大市場でありながら成長期であり，実社会において重要な位置にある．

半導体は様々な回路の部品として使われるが，特に重要なのが組込みシステムと呼ばれる専用機器である．[3] では「パソコンとは比較にならないほどの膨大な数量および種類が存在するのが専用機器」とある．[4] では組込みシステムを「プログラミング可能なコンピュータを組み込みつつも，汎用目的のコンピュータとしての使用は想定していない装置」と定義している．具体例としては自動車や電子レンジ，室温調整システム，カメラ，テレビが挙げられる [4]．これらの組込みシステムでは動作対象となるアプリケーションは汎用的ではなく，限定的である．組込みシステムでは消費電力や面積に厳しい制約が要求されるため，アプリケーションに特化したアーキテクチャとして設計される．[4] では「組込みシステムのほとんどは小規模チームによって厳しい納期のもとに設計される」とある．組込みシステムの設計期間を短縮することが強く要求されている．

ITRS (International Technology Roadmap for Semiconductors; 半導体技術のロードマップ) によると，年代を経るにつれてDRAMとFlash memoryのハーフピッチは縮小している [5]．DRAMとFlash memoryのハーフピッチの推移を図1.1に示す．ここでハーフピッチとは最小配線幅の1/2を表し，DRAMのハーフピッチがテクノロジノードである．一般にテクノロジノードはプロセスの微細化の指標として使われる．図1.1によると，テクノロジノードは年代を経るにつれ小さくなりプロセスは微細化している．

また，プロセスの微細化によってより多くのトランジスタがLSI上に乗るようになる．ムーアの法則によれば18ヶ月ごとにLSI上のトランジスタ数は2倍となる [6]．単位面積あたりのトランジスタ数が増加するとスケーリング則により演算速度が向上する [7]．一方，DDR

表 1.1: 年別半導体市場の規模 [1].

暦年	2008	2009	2010	2011	2012	2013
半導体市場 [M$]	248603	226313	298315	299521	291562	305584
成長率 [%]	-2.8	-9.0	31.8	0.4	-2.7	4.8
単機能半導体 [M$]	16935	14175	19802	21387	19138	18201
成長率 [%]	0.7	-16.3	39.7	8.0	-10.5	-4.9
光電子工学 [M$]	17902	17043	21702	23092	26175	27571
成長率 [%]	12.6	-4.8	27.3	6.4	13.4	5.3
センサー [M$]	5111	4753	6903	7970	8009	8036
成長率 [%]	-0.3	-7.0	45.2	15.5	0.5	0.3
IC [M$]	208656	190342	249909	247073	238240	251776
成長率 [%]	-4.2	-8.8	31.3	-1.1	-3.6	5.7

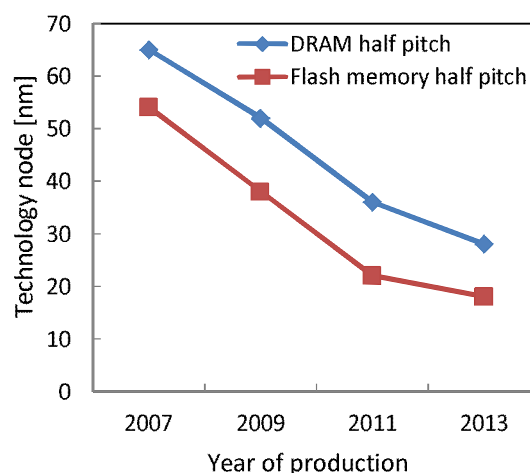

図 1.1: テクノロジノードの推移 (資料 [5] より作成).

(Double Data Rate) の技術によりメインメモリのアクセス速度も向上している [8]. しかし, メモリアクセス速度は十分ではなく, 南カルフォルニア大学の Pershin 氏はフォンノイマン型の計算機の欠点はプロセッサとメモリの間の通信速度がボトルネックになることであると述べている [9]. プロセッサの演算速度とメインメモリのアクセス速度を比較する. 例えば Intel の最新プロセッサ Haswell E5-1630 の動作周波数は 3.7GHz である [10]. 対応するメインメモリの規格 DDR4-2133 の定格動作周波数は 133MHz であると定められている [8]. 動作周波数は 27.8 倍も異なり, プロセッサの演算速度に対してメインメモリのアクセス速度は追い付いていない.

メインメモリのアクセス速度がボトルネックとなり演算全体の速度は向上が困難になって

おり，メインメモリへのアクセス速度の向上が重要である．なお，アクセス速度とメモリの容量の間にはトレードオフの関係があるため，大容量で高速アクセスのメモリを実現することは困難であり，大容量の低速なメモリと小容量の高速なメモリの組合せで実質高速大容量のメモリを実現することが課題である．

プロセッサとメインメモリに間にバッファを挿入することで実質のアクセス速度を向上する技術が存在する．バッファを挿入する技術にはスクラッチパッドメモリやキャッシュメモリが存在する [11]．スクラッチパッドメモリは明示的なソフトウェア管理によりデータを高速な小容量メモリに保持する．キャッシュメモリはハードウェア制御により自動的に必要と判断されたデータを高速小容量のメモリに保持する．一般にシステム作成時の負担軽減のためキャッシュメモリが使用される．例えばIntelのプロセッサHaswell E5-1630では10MBのキャッシュメモリが使用されている [10]．メインメモリは主にDRAMで構成され安価で大容量だが，低速でしか動作しない．キャッシュメモリは主にSRAMで構成され高価かつ小容量だが，高速に動作する．

なお，キャッシュメモリでも，容量が大きくなるに従い，アクセス速度が低下する．このため，メインメモリとの間に何段階かのサイズの異なるキャッシュメモリの階層を置き，メインメモリとプロセッサの間の速度差を埋めることが一般的である．メモリを階層的に構成することで効率的に演算全体の速度を向上させる．図1.2のようにプロセッサに近い順にL1キャッシュ，L2キャッシュ，L3キャッシュと表される．L1キャッシュが最も高速に動作するがデータ容量が小さい，L2キャッシュはL1キャッシュほど高速ではないがデータ容量が大きい．

プロセッサが必要とするデータがキャッシュメモリにあれば，アクセス速度は向上するが，なければメインメモリにアクセスするのと同じである．このため，アクセス時点で必要なデータがキャッシュにあるかどうか (ある場合をキャッシュヒット，ない場合をキャッシュミスと呼ぶ) が重要であり，全アクセス回数に対するヒットした回数の割合であるキャッシュヒット率により，アクセス速度は大きく異なる．キャッシュヒット率は，キャッシュサイズや，必要なデータの判定方法，必要とされるデータと現在キャッシュ上にあるデータのリプレース法に依存する．特に，キャッシュメモリが十分なデータ容量を持たない構成の場合に

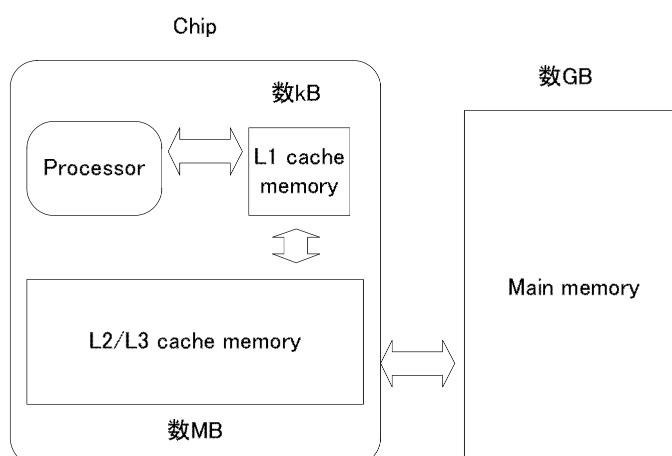

図 1.2: キャッシュメモリの位置.

はキャッシュヒット率が十分ではなく，プロセッサから見たメインメモリのアクセス速度が要求性能に達しない可能性がある.

いま，キャッシュヒット率によるメインメモリのアクセス時間の比較を考える．キャッシュヒットが起きた場合のメインメモリのアクセス時間を 1ns とする．キャッシュミスが起きた場合のメインメモリのアクセス時間を 10ns とする．10 回のキャッシュアクセスがあったとする．10 回キャッシュヒット，キャッシュヒット率 100%ならばメインメモリの全アクセス時間は 10ns である．9 回のキャッシュヒットと 1 回のキャッシュミス，キャッシュヒット率 90%ならばメインメモリの全アクセス時間は 19ns である．10%のキャッシュヒット率の差が全アクセス時間の差として大きく表れる．キャッシュメモリのサイズが大きくなればキャッシュヒット率は向上するが面積が大きくなる．キャッシュメモリが必要以上に規模の大きい構成の場合にはメインメモリのアクセス速度に対して消費電力や面積が余分にかかる可能性がある.

組込みシステムでは消費電力や面積に厳しい制約が要求されるため，これら制約と要求性能に対して適切なキャッシュ構成を設計する必要がある．特に組込みシステムでは事前にアプリケーションを与えられているのでキャッシュ構成をそのアプリケーションに特化することができる．組込みシステムが使われる製品は例えば自動車，テレビ，冷蔵庫がある．自動車のエンジンやブレーキの制御にはリアルタイム性と信頼性が要求されるため十分に性能が高いのキャッシュ構成が想定される．テレビはリアルタイム性があるが自動車ほど信頼性は要求されないので高い要求性能を持ち消費電力，面積が小さいキャッシュ構成が想定される.

冷蔵庫は自動車やテレビほどリアルタイム性，信頼性が要求されないので必要最低限の要求性能を持ち消費電力，面積が小さいキャッシュ構成が想定される．

キャッシュ構成の性能の良さはキャッシュヒット率を指標として測ることができる [12]．キャッシュヒット率はキャッシュヒット/ミス回数から計算できる．キャッシュヒット/ミス回数はキャッシュヒットとキャッシュミスが発生した回数である．あるキャッシュ構成でアプリケーションが動作したときのキャッシュヒット/ミス回数が推定できれば性能の指標にすることができる．つまり組込みシステムのプロセッサアーキテクチャは汎用のプロセッサアーキテクチャと異なり事前にキャッシュヒット/ミス回数を推定してキャッシュ構成を設計することができる．

キャッシュヒット/ミス回数を推定する手法は大きく分けて2つある．1つはキャッシュの性質やワーストケース解析によりキャッシュヒット/ミス回数を見積もる手法 [13–41] であり，もう1つは実際にメモリアクセスをシミュレーションしてキャッシュヒット/ミス回数を数え上げる手法 [42–49] である．前者の手法は見積もりにかかる時間は数秒と短いが誤差が大きく，後者の手法は正確だが数時間から数日と時間がかかる [50]．組込み用途では，正確な評価に基づく最適化が必要不可欠であり，本論文では正確性を重視し後者の手法を対象とする．ただし，設計期間を短縮することが強く要求される組込みシステムでは後者の手法は時間がかかりすぎるという問題があるため，その高速化を目指す．

1.2 本論文の目的

組込みシステムでは設計期間を短縮することが重要である．実際にメモリアクセスをシミュレーションしてキャッシュヒット/ミス回数を正確に数え上げる手法では時間がかかるという問題に対して，これを高速化することを目的とする．これにより，種々のキャッシュ構成の中から最適な構成を求めることができ，組込みシステムのトータル性能の向上につながる．

あるアプリケーションが実行されるとき，プロセッサからメインメモリへのアクセス要求の順序は，命令の実行順序を変更しないインオーダ実行ならばキャッシュ構成に依存しない．よってプロセッサとアプリケーションが与えられたとき，キャッシュ構成が隠蔽されていても

プロセッサ上でアプリケーションが動作した際のメモリアクセスの順序をキャッシュ構成と独立に得ることができる．このメモリアクセスの順序をメモリアクセストレースと呼ぶ．メモリアクセストレースをもとにパラメータを組み替えた複数のキャッシュ構成でシミュレーションしキャッシュヒット/ミス回数を数え上げることがトレースベースのキャッシュ構成シミュレーションである．設計時にキャッシュ構成シミュレーションするのは時間がかかり，設計期間が長くなるとそれだけ設計コストがかかる．組込みシステムの設計期間の短縮のためキャッシュ構成シミュレーションを高速化することを目的とする．

キャッシュアーキテクチャ設計時にアプリケーションに特化したキャッシュ構成をシミュレーションにより探索する方法について考える．最も簡単なトレースベースシミュレーションとしてキャッシュ構成それぞれに対して個別にシミュレーションを行う全探索手法が考えられる．しかし，これは数時間から数日と時間がかかるため現実的ではない [50]．既存のトレースベースのキャッシュ構成シミュレーションの高速化手法のうち，Janapsatya らの手法 [12] や CRCB 手法 [50]，CRCB-U 手法 [51] は LRU ベースキャッシュ構成シミュレーションの高速化手法である．LRU ベースキャッシュには *Inclusion Property* [52] と呼ばれる性質が存在する．Inclusion Property は小さなキャッシュ構成に含まれるデータは大きなキャッシュ構成にも含まれるという性質である．この性質を使い，Janapsatya らの手法や CRCB 手法，CRCB-U 手法はキャッシュ構成シミュレーションを高速化する．Janapsatya らの手法は LRU ベース L1 キャッシュを対象とする手法である．LRU ベース L1 キャッシュの複数の構成をひとつのデータ構造で表す．ひとつのデータ構造の探索と更新を省略することで高速にキャッシュ構成シミュレーションする．CRCB 手法は LRU ベース L1 キャッシュを対象とする手法である．あるメモリアクセスがひとつのキャッシュ構成でキャッシュヒットが発生した場合，別のキャッシュ構成においてもキャッシュヒットするという性質を使うことで探索を省略し高速にキャッシュ構成シミュレーションする．CRCB-U 手法ははLRU ベース L1 データキャッシュ，L1 命令キャッシュ，L2 ユニファイドキャッシュの 2 階層キャッシュアーキテクチャを対象とする手法である．L1 データキャッシュをひとつの構成に固定して考えると，L1 命令キャッシュと L2 ユニファイドキャッシュは構成を変えながら CRCB 手法を適用できる．L1 命令キャッシュと L2 ユニファイドキャッシュに CRCB 手法を適用することで高速にキャッシュ

構成シミュレーションする．Haqueらの手法 [43] はLRUベースL1キャッシュを対象とする手法である．これらの既存手法はLRUベースキャッシュのみを対象としてキャッシュ構成シミュレーションを高速化しており，組込み用途で用いられるPLRUやFIFOベースキャッシュには対応していない．

Haqueらの手法 [44–46] はFIFOベースL1キャッシュを対象とする手法である．手法 [43,44] ではデータ構造として木構造を用いて複数の構成の探索を容易にすることでシミュレーションを高速化している．手法 [45] では手法 [44] のデータ構造に加えてルックアップテーブルとリスト構造を用いてシミュレーションを高速化している．手法 [46] では手法 [44,45] のデータ構造に加えてFIFOキューに関する性質 *Intersection Property* [46] を用いてシミュレーションを高速化している．キャッシュ構成はパラメータによって決定される．Haqueらの手法はパラメータの1つを固定して探索しているため，シミュレーション可能なキャッシュ構成のパラメータ範囲が狭く，十分なキャッシュ構成のパラメータ範囲をシミュレーションするためには，手法を繰り返し適用する必要があるため，非常に時間がかかる．パラメータ範囲全体に対して高速化が達成されていないため，キャッシュ構成シミュレーションの高速化手法として不十分である．

そこで，本論文では，組込み向けのFIFOやPLRUベースのキャッシュに適用可能な正確かつ高速なキャッシュ構成シミュレーションの研究を行う．本論文ではキャッシュ構成のパラメータ範囲に制限を設けずにシミュレーションを高速化することを目標とする．

本論文ではトレースベースのキャッシュ構成シミュレーションの高速化を目的とする．キャッシュメモリにはLRUやFIFO，PLRUベースキャッシュがある．FIFO/PLRUベースキャッシュはLRUベースキャッシュよりもハードウェアコストが低いため組込みシステムのプロセッサアーキテクチャとの親和性が高い．Intel XScaleやARM9, ARM11，PowerPCなどのプロセッサではキャッシュリプレースメントポリシにFIFOやPLRUが使われている [21,53]．FIFO/PLRUベースキャッシュ構成シミュレーションは既存のシミュレーション高速化手法では適用不可能である．そこでFIFO/PLRUベースキャッシュを対象にキャッシュ構成シミュレーションを高速化する．また，マルチコアプロセッサアーキテクチャのキャッシュについてトレースベースのキャッシュ構成シミュレーションの高速化は研究例がない．そこでマル

表 1.2: 既存手法と本論文の手法の比較.

	キャッシュリプレースメントポリシ	[12]	[50]	[43]	[44–46]	本論文で提案する手法
シングルコア	LRU	可	可	可	不可	可 (第3章)
	FIFO	不可	不可	不可	一部可	可 (第3,4章)
	PLRU	不可	不可	不可	不可	可 (第3章)
マルチコア	LRU	不可	不可	不可	不可	可 (第5章)

チコアプロセッサキャッシュを対象にキャッシュ構成シミュレーションを高速化する．マルチコアプロセッサキャッシュは性質が明瞭で高速化しやすいLRUベースキャッシュを対象とする．これらキャッシュ構成シミュレーションを本論文では対象とし，シミュレーション高速化を実現する．表1.2に各キャッシュリプレースメントポリシに対して既存手法と本論文の手法が適用可能か不可能かの比較を示す.

1.3 本論文の概要

本論文でははじめに記憶装置のアクセス速度を向上させる技術であるキャッシュメモリについて取り上げ，キャッシュメモリの設計技術であるキャッシュ構成シミュレーションを説明する．次にキャッシュ構成シミュレーションの高速化に関連する研究を紹介する．その中でメモリアクセストレースによりシミュレーションを行うトレースベースシミュレーションを使用したシングルコアプロセッサのFIFO/PLRUベースキャッシュ構成シミュレーションの高速化手法を提案する．さらにシングルコアプロセッサのFIFOベースキャッシュ構成シミュレーションの高速化手法を提案する．そしてマルチコアプロセッサのLRUベースキャッシュ構成シミュレーションの高速化手法を提案する．

以下に本論文の構成を示す．

第2章「関連研究」では関連研究を紹介する．最初にキャッシュメモリの説明とキャッシュメモリを構成する際のパラメータ，キャッシュ構成シミュレーションを説明する．次に関連研究としてキャッシュシミュレーション高速化手法を紹介する．シングルプロセッサアーキテクチャのキャッシュを対象とする手法としてJanapsatyaらの手法やCRCB手法，CRCB-U手法，Haqueらの手法がある． マルチコアプロセッサアーキテクチャのキャッシュを対象とする手法は存在しない．Janapsatyaらの手法やCRCB手法，CRCB-U手法はLRUベースキャッシュの性質Inclusion Propertyを利用してキャッシュ構成シミュレーションを高速化する手法である．Haqueらの手法はFIFOベースキャッシュの性質Intersection Propertyを利用してキャッシュ構成シミュレーションを高速化する手法である．

第3章「シングルコアプロセッサアーキテクチャにおける柔軟なリプレースメントポリシをベースとするキャッシュの高速キャッシュ構成シミュレーション」ではFIFO/PLRUベースキャッシュを対象にキャッシュ構成シミュレーションを高速化する．最初に第2章で紹介したLRUベースキャッシュを対象とするキャッシュ構成シミュレーション高速化手法CRCB手法がFIFO/PLRUベースキャッシュ構成シミュレーションで動作するか検証し，動作するために満たすべき条件を考察，証明する．次にFIFO/PLRUベースキャッシュの性質からキャッシュ構成シミュレーション高速化を考察する．FIFO/PLRUベースキャッシュについてキャッ

シュセット同士に成り立つ関係を定義し，これを用いてキャッシュ構成シミュレーション高速化のためのデータ構造とこれを扱うアルゴリズムを提案する．最後に提案手法の計算機実験を行う．

第 4 章「シングルコアプロセッサアーキテクチャにおける FIFO ベースキャッシュの高速キャッシュ構成シミュレーション」では FIFO ベースキャッシュを対象にキャッシュ構成シミュレーションを高速化する．最初に FIFO ベースキャッシュの性質からキャッシュ構成シミュレーション高速化手法を考察する．次に FIFO ベースキャッシュについてキャッシュセット同士に成り立つ関係を定義し，これを用いてキャッシュ構成シミュレーション高速化のためのデータ構造とこれを扱うアルゴリズムを提案する．最後に提案手法の計算機実験を行う．

第 5 章「マルチコアプロセッサアーキテクチャにおける高速キャッシュ構成シミュレーション」ではマルチコアプロセッサアーキテクチャのキャッシュを対象にキャッシュ構成シミュレーションを高速化する．最初にマルチコアプロセッサキャッシュが持つキャッシュコヒーレンシプロトコルを説明する．次にキャッシュコヒーレンシプロトコルの性質からキャッシュ構成シミュレーション高速化手法を考察する．キャッシュコヒーレンシプロトコルに関連したメタ状態を導入する．メタ状態により効率的にキャッシュ構成シミュレーションするデータ構造とこれを扱うアルゴリズムを提案する．最後に提案手法の計算機実験を行う．

第 6 章「結論」では本論文のまとめを述べる．

第2章

関連研究

2.1 本章の概要

本章では最初にキャッシュメモリの説明とキャッシュメモリを構成する際のパラメータ，キャッシュ構成シミュレーションを説明する．次に関連研究としてキャッシュシミュレーション高速化手法を紹介する．シングルコアプロセッサアーキテクチャのキャッシュを対象とする手法として Janapsatya らの手法や CRCB 手法，CRCB-U 手法，Haque らの手法があり，それらについて説明する．Janapsatya らの手法や CRCB 手法，CRCB-U 手法は LRU ベースキャッシュの性質 Inclusion Property を利用してキャッシュ構成シミュレーションを高速化する手法である．また Haque らの手法は FIFO ベースキャッシュの性質 Intersection Property を利用してキャッシュ構成シミュレーションを高速化する手法である．

2.2 キャッシュメモリとキャッシュ構成シミューション

本節ではキャッシュメモリの構成と本論文で対象とするキャッシュ構成シミュレーションを説明する．メインメモリは主に DRAM で構成される．DRAM は安価なため大容量のメモリを構成できるが速度が遅いという問題がある．これに対し高速で小容量なメモリをプロセッサとメインメモリの中間に挿入するキャッシュメモリがある．キャッシュメモリは主に SRAM で構成される．[54] では DRAM のアクセス時間は 10ns であり，SRAM のアクセス時間は 2ns とアクセス時間に数倍の差がある．SRAM は DRAM に比べて高速だが高価なため，キャッシュメモリは小容量な SRAM で構成される．さらに図 1.2 のようにキャッシュメモリを階層的に構成することで効率的に演算全体の速度を向上させる．読み書きを毎回プロセッサからメインメモリに対して行うと通信時間がかかってしまう．一方，読み書きをキャッシュメモリに対して行うと通信時間は短い．キャッシュメモリはプロセッサとメインメモリの中間に位置し，これを導入することでデータをバッファして速度を上げることができる．データをバッファするとプログラムやデータの時間的局所性や空間的局所性により，実質にメインメモリのアクセス速度が向上する．

キャッシュメモリは設定されたパラメータによってハードウェア制御で動作し，実装によっていくつかの方式がある．本論文ではキャッシュメモリの構成を定義する方式として，セッ

トアソシアティブ方式を対象とする [48]．セットアソシアティブ方式はキャッシュメモリをセット数，ブロックサイズ，連想度の3つのパラメータで管理する．キャッシュ上で管理する情報の最小単位をブロックと呼ぶ．ブロックサイズはブロックの容量である．キャッシュメモリはいくつかのブロックをまとめて管理する．ブロックのまとまりをキャッシュセットと呼ぶ．セット数はキャッシュを構成するキャッシュセットの数である．キャッシュセットに含まれるブロックの最大数を連想度と呼ぶ．キャッシュメモリはセットから構成され，キャッシュメモリを構成するセットの数をセット数と呼ぶ．セットはブロックから構成され，セットを構成するブロックの数を連想度と呼ぶ．また，キャッシュ内のデータを追い出すアルゴリズムをキャッシュリプレースメントポリシと呼ぶ．本論文ではキャッシュリプレースメントポリシは LRU (Least Recently Used)，FIFO (First In First Out)，PLRU (Pseudo LRU) のうちいずれかを前提とする [21, 53]．キャッシュセットはそれぞれがキャッシュリプレースメントポリシに沿って動作する優先度付きキューである．LRU によって動作するキャッシュセットは最もアクセスされていないブロックを優先して追い出す．FIFO によって動作するキャッシュセットは最も先に入ったブロックを優先して追い出す．PLRU は局所性を利用した LRU に似たアルゴリズム全般を指し，特定のアルゴリズムを指すものではない．本論部では木構造に基づく PLRU を想定する [55]．PLRU によって動作するキャッシュセットは疑似的に最も使われていないブロックを優先して追い出す．

LRU はハードウェアで実現する際にデータごとに優先度を表現するビットを用いて実装する [56]．LRU では要素数 A に対して $A \times \log_2 A$ のビットが必要である．これに対して PLRU では優先度を表現するビットは共有されている．PLRU はは要素数 A に対して $A-1$ のビットが必要である．LRU に比べて PLRU は必要とするビット数は少なく，ハードウェアコストが低い．

本論文ではセット数 s，ブロックサイズ b，連想度 a のキャッシュ構成 c を $c = (s, b, a)$ で表す．2階層キャッシュでは，L1 データキャッシュを $c_d = (s_d, b_d, a_d)$，L1 命令キャッシュを $c_i = (s_i, b_i, a_i)$，L2 ユニファイドキャッシュを $c_{L2} = (s_{L2}, b_{L2}, a_{L2})$ で表し，全体のキャッシュ構成を $c_U = ((s_d, b_d, a_d), (s_i, b_i, a_i), (s_{L2}, b_{L2}, a_{L2}))$ で表す．キャッシュ構成 (s, b, a) に対してメモリアクセスが発生したとき，アドレスはタグ，インデックス，オフセットに分割される．ア

ドレスの下位 $\lg b$ ビットはオフセット，続く下位 $\lg s$ ビットはインデックス，残りのビットはタグとなる．図2.1にメモリアドレスのタグとインデックス，オフセットの分割を示す．タグはセット内のブロックにどのアドレスのデータが入っているかを示す．インデックスはどのセットに該当データが含まれるかを示す．オフセットはブロックの何バイト目が該当データかを示す．キャッシュ構成 c のインデックス i のセットを $S(c,i)$ で表す．セットはキャッシュリプレースメントポリシに沿って動作する優先度付きキューとみなせるため，セット内のブロックに優先度を定義できる．優先度が大きい順に追い出されるとする．LRUでは最も使われていないデータの優先度が高く，FIFOでは先に入ったデータの優先度が高い．セット $S(c,i)$ の優先度 j のブロックを $S(c,i)_j$ で表す．キャッシュ構成 $c=(16,16,4)$，メモリアクセス $A = 1010,1010\,0000,0000$ とするとタグは $1010,1010$，インデックスは 0000 である $S(c,0000)$ と $S(c,0000)_1$ の例は図2.2となる．メモリアクセス A はインデックス 0000 のセットの優先度3のブロックにデータが存在する．

プログラムが動作するときプロセッサからメインメモリへのメモリアクセスはキャッシュメモリの存在を意識しない．あるアプリケーションが動作するときのメモリアクセスのリストを入手すれば，プログラムを再度実行することなくメモリアクセスをシミュレーションすることができる．このリストをメモリアクセストレースと呼ぶ．メモリアクセストレースを使い，特定の構成のキャッシュメモリでキャッシュヒット/ミス回数を数えるシミュレーションをキャッシュシミュレーションと呼ぶ．キャッシュ構成シミュレーションは複数のキャッシュ構成でキャッシュシミュレーションを行いキャッシュヒット/ミス回数を数え上げるシミュレーションである．

キャッシュセット数とブロックサイズはメモリアドレスのビットによって表現される．そのためキャッシュセット数とブロックサイズは2のべき乗の値が与えられるのが一般的である．本論文で対象とするキャッシュ構成は

$s = s_0, 2s_0, 4s_0, 8s_0, \dots, s_m$

$b = b_0, 2b_0, 4b_0, 8b_0, \dots, b_m$

$a = 1, 2, 3, \dots, a_m$

とする．ここで s_0，b_0 はセット数，ブロックサイズの最小値であり，s_m，b_m，a_m はセット

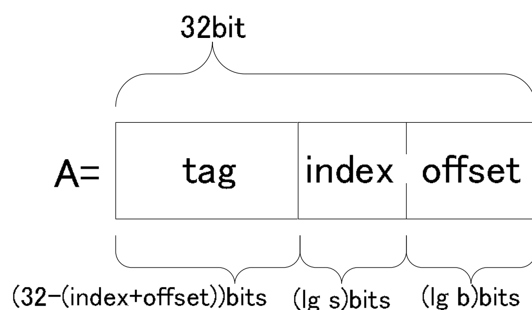

図 2.1: メモリアドレス 32bits，セット数 s，ブロックサイズ b Bytes，連想度 a の場合の tag, index, offset の各 bit 数 ([57] に掲載).

数，ブロックサイズ，連想度の最大値である．

今，1 つのキャッシュ構成 $c = (s, b, a)$ を考える．キャッシュシミュレーションの全探索アルゴリズムを示す.

1) キャッシュ構成 c に対しメモリアクセス A が発生したときインデックス i とタグ t を求める．インデックス i よりセット $S(c, i)$ を求める．

2) セット $S(c, i)$ の優先度付きキューにタグ t が存在しているか判定する．

3) もしステップ 2) でセット $S(c, i)$ に優先度 j でタグ t が存在していれば，メモリアクセス A はキャッシュ構成 c に対しキャッシュヒットとなる．またセット $S(c, i)_j$ の優先度ををキャッシュリプレースメントポリシに基づき更新する．

4) もしステップ 2) でセット $S(c, i)$ にタグ t が存在していなければ，メモリアクセス A はキャッシュ構成 c に対しキャッシュミスとなる．またセット $S(c, i)$ にキャッシュリプレースメントポリシに基づきタグ t のブロックを追加する．

キャッシュ構成シミュレーションはメモリアクセストレースに対し対象とする全てのキャッシュ構成のキャッシュヒット/ミス回数を数え上げるシミュレーションである．

2.3 Janapsatya らの手法

Janapsatya らの手法 [12] は LRU ベース L1 キャッシュのキャッシュを対象とする手法である．$a < a'$ のとき，LRU ベースキャッシュではキャッシュ構成 $c_0 = (s, b, a)$ とキャッシュ構成 $c_1 = (s, b, a')$ の間に Inclusion Property [12] と呼ばれる密接な関係が成り立つ．

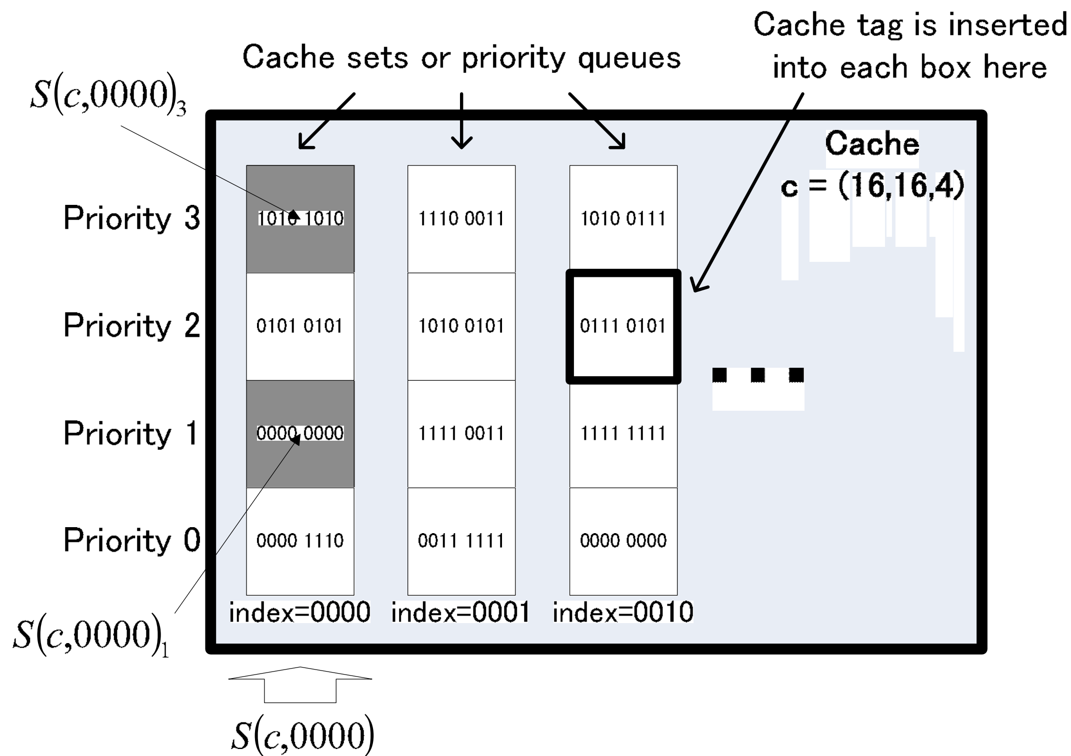

図 2.2: $S(c, 0000)$ と $S(c, 0000)_1$ の例 ([57] に掲載).

性質 1. $a < a'$ のとき，キャッシュ構成 $c_0 = (s, b, a)$，$c_1 = (s, b, a')$ に対しあるメモリアクセス A が発生したとする．もし c_1 で A がキャッシュミスしたならば，c_0 でもキャッシュミスとなる．もし c_0 で A がキャッシュヒットしたならば，c_1 でもキャッシヒットとなる．

性質 1 を図 2.3 に示す．Janapsatya らの手法は性質 1 を使い連想度の異なる複数のキャッシュ構成のデータをひとつのデータ構造で表し，探索と更新を省略することで高速にキャッシュ構成シミュレーションする．

Janapsatya らの手法は LRU ベースキャッシュを対象とした手法であり，キャッシュリプレースメントポリシが FIFO や PLRU であるキャッシュのキャッシュ構成シミュレーションには適用できない．FIFO ベースキャッシュのキャッシュ構成シミュレーションでは Janapsatya らの手法が適用できないことを証明する．

定理 1. *FIFO ベースキャッシュでは性質 1 は成立しない構成が存在する．*

Proof. FIFO ベースキャッシュのキャッシュ構成 $c_0 = (16, 16, 2)$，$c_1 = (16, 16, 4)$. に対して連続する 7 個のメモリアクセス Ai ($i = 1, 2, 3, \ldots, 7$) が発生したとする．

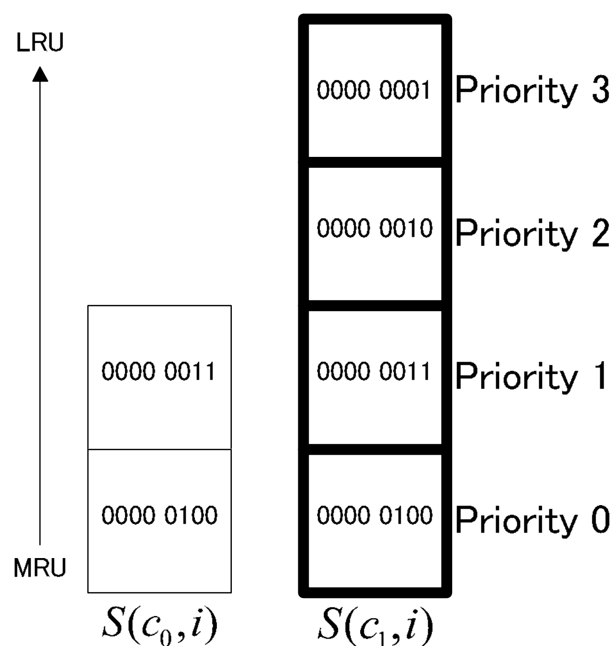

図 2.3: Janapsatya の手法：性質 1 の例 ([57] に掲載).

（1）$A_1 = 0000\ 0001\ 1000\ 0000$

（2）$A_2 = 0000\ 0010\ 1000\ 1000$

（3）$A_3 = 0000\ 0011\ 1000\ 0100$

（4）$A_4 = 0000\ 0100\ 1000\ 0010$

（5）$A_5 = 0000\ 0001\ 1000\ 0001$

（6）$A_6 = 0000\ 0101\ 1000\ 0001$

（7）$A_7 = 0000\ 0001\ 1000\ 0010$

メモリアクセス Ai のインデックスは c_0，c_1 でともに 1000 となる．すべてのメモリアクセスでそれぞれのキャッシュメモリの $S(c_0, 1000)$，$S(c_1, 1000)$ を更新することになる．図 2.4 にメモリアクセス Ai ($i = 1, 2, 3, \ldots, 7$) が発生したときの $S(c_0, 1000)$，$S(c_1, 1000)$ を示す．メモリアクセス A_7 の前，図 2.4(f) では $S(c_0, 1000)$ にはタグ 0000 0001，0000 0101 のブロックがある．$S(c_1, 1000)$ には 00000010， 00000011，00000100，00000101 のブロックがある．メモリアクセス A_7 は発生したとき $S(c_0, 1000)$ にはタグ 0000 0001 のブロックがあるが $S(c_1, 1000)$

には存在しない．よって c_0 でキャッシュヒットするが，c_1 でキャッシュミスとなる．したがって FIFO ベースキャッシュでは性質 1 は成立しない． □

同様に PLRU ベースキャッシュのキャッシュ構成シミュレーションでは Janapsatya らの手法が適用できないことを証明する．

定理 2. *PLRU ベースキャッシュでは性質 1 は成立しない構成が存在する．*

Proof. PLRU ベースキャッシュのキャッシュ構成 $c_0 = (16, 16, 4)$，$c_1 = (16, 16, 8)$. に対して連続する 14 個のメモリアクセス Bi ($i = 1, 2, 3, \ldots, 14$) が発生したとする．

(1) $B_1 = 0000\ 1000\ 1000\ 0000$

(2) $B_2 = 0000\ 0111\ 1000\ 1000$

(3) $B_3 = 0000\ 0110\ 1000\ 0100$

(4) $B_4 = 0000\ 0101\ 1000\ 0010$

(5) $B_5 = 0000\ 0100\ 1000\ 0001$

(6) $B_6 = 0000\ 0011\ 1000\ 0001$

(7) $B_7 = 0000\ 0010\ 1000\ 0010$

(8) $B_8 = 0000\ 0001\ 1000\ 0000$

(9) $B_9 = 0000\ 0101\ 1000\ 0000$

(10) $B_{10} = 0000\ 1001\ 1000\ 0000$

(11) $B_{11} = 0000\ 1010\ 1000\ 0000$

(12) $B_{12} = 0000\ 1001\ 1000\ 0000$

(13) $B_{13} = 0000\ 1011\ 1000\ 0000$

(14) $B_{14} = 0000\ 0001\ 1000\ 0000$

メモリアクセス Bi のインデックスは c_0，c_1 でともに 1000 となる．すべてのメモリアクセスでそれぞれのキャッシュメモリの $S(c_0, 1000)$，$S(c_1, 1000)$ を更新することになる．図 2.5 にメモリアクセス Bi ($i = 1, 2, 3, \ldots, 13$) が発生したときの $S(c_0, 1000)$，$S(c_1, 1000)$ を示す．メモリアクセス B_{14} は発生したとき，図 2.5(e) では $S(c_0, 1000)$ にはタグ 0000 0001 のブロックがある．$S(c_1, 1000)$ には 00000001 のブロックがない．よって c_0 でキャッシュヒットするが，c_1 でキャッシュミスとなる．したがって PLRU ベースキャッシュでは性質 1 は成立しない． □

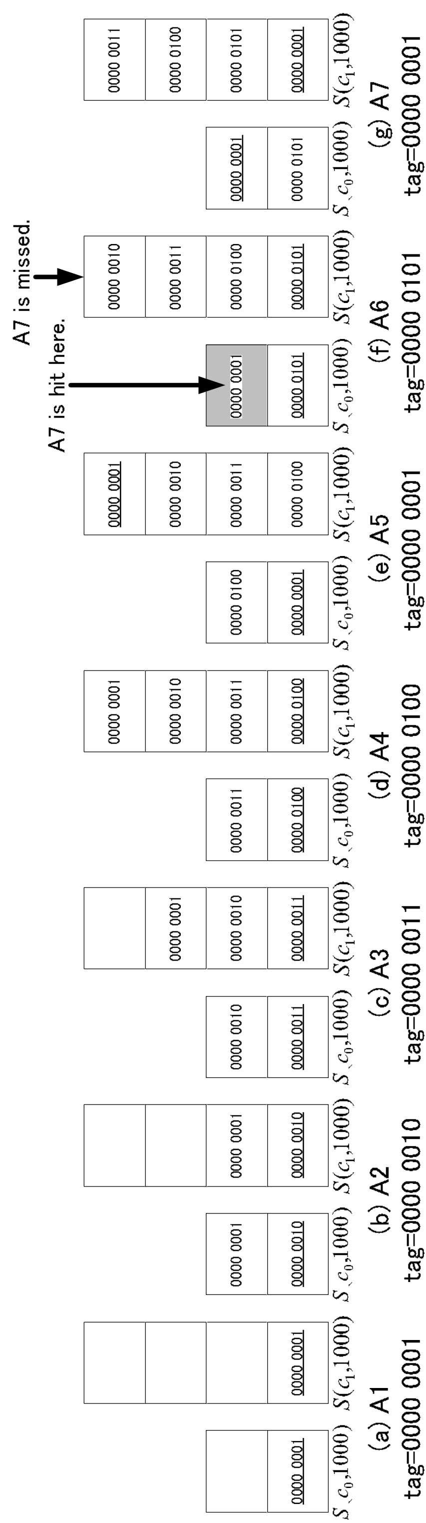

図 2.4: 定理 1 の証明 ([57] に掲載).

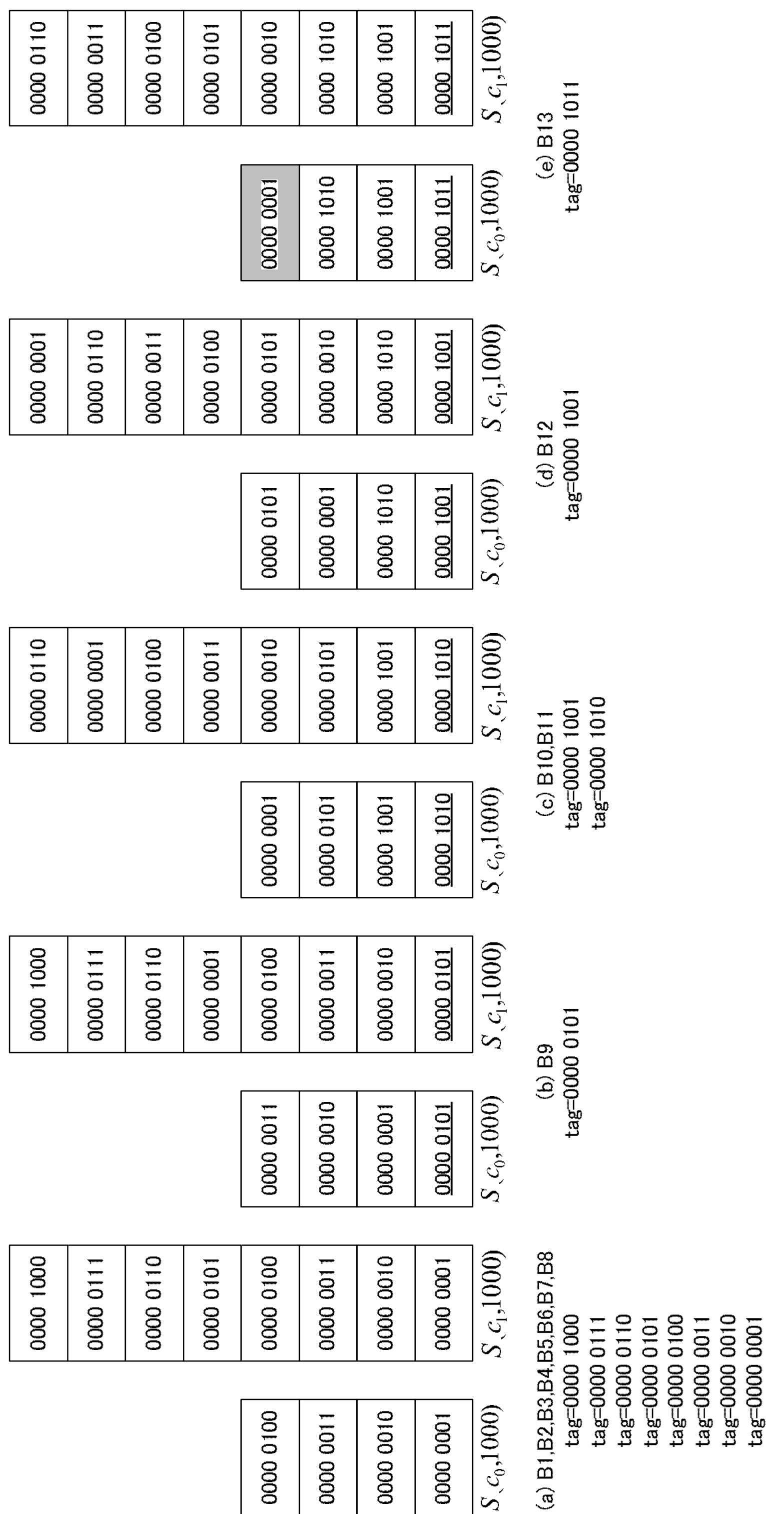

図 2.5: 定理 2 の証明.

2.4 CRCB (Configuration Reduction approach by the Cache Behavior) 手法

CRCB 手法は LRU をキャッシュリプレースメントポリシとして想定したキャッシュ構成シミュレーションの高速化手法である [50]．CRCB 手法は複数のキャッシュ構成同士の間に成り立つ関係を利用することで一部のキャッシュ構成をシミュレーションすることなくそのキャッシュ構成のヒット/ミス数を正確に判定する．CRCB 手法は CRCB1 手法と CRCB2 手法から成り立っている．CRCB1 手法はブロックサイズが同じでセット数，連想度が異なるキャッシュ構成の関係を利用する手法である．CRCB2 手法は連続したメモリアクセスからキャッシュヒットを判定する手法である．

アルゴリズムは以下のようになる．

CRCB1 もし，キャッシュ構成 $(s, b, 1)$ について，あるメモリアクセス A がキャッシュヒットしたとすれば，$s' = s, 2s, 4s, \ldots, s_m, a = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b, a) でも，キャッシュヒットするため，$s' = s, 2s, 4s, \ldots, s_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b, a') のヒット/ミス判定を省略する．

CRCB2 もし，キャッシュ構成 (s, b, a) で $n-1$ 回目のメモリアクセスと n 回目のメモリアクセスのインデックス i およびタグ t が等しい場合，n 回目のアクセスは $s' = s, 2s, 4s, \ldots, s_m, b' = b, 2b, 4b, \ldots, b_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b', a') で必ずキャッシュヒットするため，$s' = s, 2s, 4s, \ldots, s_m, b' = b, 2b, 4b, \ldots, b_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b', a') でのヒット/ミス判定を省略する．

キャッシュリプレースメントポリシとして LRU を前提に Janapsatya らの手法と CRCB 手法を用いれば，全探索するキャッシュ構成シミュレーションより，平均 205 倍高速化されることが報告されている [50]．

2.5 CRCB-U 手法

CRCB-U 手法は LRU をキャッシュリプレースメントポリシとする L1 データキャッシュ，L1 命令キャッシュ，L2 ユニファイドキャッシュの 2 階層キャッシュメモリを対象としたキャッ

シュ構成シミュレーション高速化手法である [51]．CRCB-U 手法はL1データキャッシュの構成を固定して考えることでL1命令キャッシュとL2ユニファイドキャッシュでJanapsatyaらの手法とCRCB手法を使い探索を省略する．以下にCRCB-U手法のアルゴリズムを示す．

CRCB-U 手法

(U-0) $s_d = s_0, b_d = b_0, a_d = 1$ に初期化する．

(U-1) メモリアクセス m_i を m_1 に初期化する．

(U-2) メモリアクセス m_i が命令メモリへのアクセスなら **L1命令キャッシュシミュレーション**の O-1 へ行く．

(U-3) メモリアクセス m_i の構成 (s_d, b_d, a_d) におけるインデックス i_d とタグ t_d を計算する．m_i が初期参照ミスか調べる．

(U-4) 構成 (s_d, b_d, a_d) のデータキャッシュの該当セットでデータが存在するか調べる．キャッシュヒットならば総構成 $((s_d, b_d, a_d), -, -)$ でキャッシュヒットとなる．

(U-5) キャッシュミスならば **L2キャッシュシミュレーション**を行う．

(U-6) m_i が m_n でなければ m_i に m_{i+1} を代入し U-2 へ行く．

(U-7) 構成 (s_d, b_d, a_d) のキャッシュミス数が初期参照ミス数でなければ s_d に $2s_d$ を代入する．構成が探索範囲内ならば U-1 へ行く．探索範囲外ならば次へ行く．初期参照ミス数と一致すれば $s'_d = s_d, \cdots, s_m$ となる $((s'_d, b_d, a_d), (s_i, b_i, a_i), (s_{L2}, b_{L2}, a_{L2}))$ のキャッシュミス回数は $((s_d, b_d, a_d), (s_i, b_i, a_i), (s_{L2}, b_{L2}, a_{L2}))$ と一致するので探索を省略する．

(U-8) s_d に s_0 を，a_d に $a_d + 1$ を代入する．構成が探索範囲内ならば U-1 へ行く．

(U-9) a_d に a_0 を，b_d に $2b_d$ を代入する．構成が探索範囲内ならば U-1 へ行く．探索範囲外ならば終了．

L1命令キャッシュシミュレーション

(O-1) m_i が初期参照ミスか調べる．

(O-2) $s_i = s_0, b_i = b_0, a_i = 1$ に初期化する．

(O-3) メモリアクセス m_i の構成 (s_i, b_i, a_i) におけるインデックス i_i とタグ t_i を計算する．

(O-4) m_i が初期参照ミスならばO-6へ行く．初期参照ミスでないならば構成 (s_i, b_i, a_m) にデータが存在するか探索する．

(O-5) セットの j 番目のブロックでキャッシュヒットしたとき，$a' = j+1, \cdots, a_m$ となる構成 $(-, (s_i, b_i, a'), -)$ でキャッシュヒット，$a'' = 1, 2, \cdots, j-1$ となる構成 $(, -(s_i, b_i, a''), -)$ でキャッシュミスする．キャッシュミスする構成で**L2キャッシュシミュレーション**を行い，O-7へ行く

(O-6) 構成 (s_i, b_i, a_m) でキャッシュミスならば $a''' = 1, 2, \cdots, a_m - 1$ となる構成 $(-, (s_i, b_i, a'''), -)$ でキャッシュミスする．キャッシュミスする構成で**L2キャッシュシミュレーション**を行い，O-9へ行く．

(O-7) O-5でキャッシュヒットしたとき，$j = 1$ ならば $s' = s_i, \cdots, s_m, a'''' = 1, \cdots, a_m$ となる $(-, (s', b_i, a''''), -)$ でもキャッシュヒットする．

(O-8) O-5で構成 (s_d, b_d, a_d) における m_i と m_{i-1} のタグとインデックスが等しい場合，$s' = s_i, \cdots, s_m, b' = b_i, \cdots, b_m, a'''' = 1, \cdots, a_m$ となる $(-, (s', b', a''''), -)$ でもキャッシュヒットする．

(O-9) b_i に $2b_i$ を代入する．構成が探索範囲内ならばO-3へ行く．

(O-10) b_i に b_0 を，s_i に $2s_i$ を代入する．構成が探索範囲内ならばO-3へ行く．範囲外ならばU-6へ行く．

L2キャッシュシミュレーション

(T-1) $s_{L2} = s_i, b_{L2} = b_i, a_{L2} = a_m$ に初期化する．

(T-2) メモリアクセス m_i の構成 (s_{L2}, b_{L2}, a_{L2}) におけるインデックス i_{L2} とタグ t_{L2} を計算する．

(T-3) m_i が初期参照ミスでないならば構成 (s_{L2}, b_{L2}, a_{L2}) にデータが存在するか探索する．初期参照ミスならばT-5へ行く．

(T-4) セットの l 番目のブロックでキャッシュヒットしたとき，$a' = l+1, \cdots, a_m$ となる構成 $(-, -, (s_{L2}, b_{L2}, a'))$ でキャッシュヒット，$a'' = 1, 2, \cdots, l-1$ となる構成

$(-, -, (s_{L2}, b_{L2}, a''))$ でキャッシュミスする．T-6へ行く

(T-5) 構成 (s_{L2}, b_{L2}, a_{L2}) でキャッシュミスならば $a''' = 1, 2, \cdots, a_m - 1$ となる構成 $(-, -, [(s_{L2}, b_{L2}, a'''))$ でキャッシュミスする．T-7へ行く．

(T-6) T-4でキャッシュヒットしたとき，$l = 1$ ならば $s' = s_i, \cdots, s_m, a'''' = 1, \cdots, a_m$ となる $(-, -, (s', b_{L2}, a''''))$ でもキャッシュヒットする．

(T-7) b_{L2} に $2b_{L2}$ を代入する．構成が探索範囲内ならばT-2へ行く．

(T-8) b_{L2} に b_i を，s_{L2} に $2s_{L2}$ を代入する．構成が探索範囲内ならばT-2へ行く．

(T-9) L2キャッシュシミュレーションを終了する．

CRCB-U手法はLRUをキャッシュリプレースメントポリシとするL1，L2キャッシュメモリを対象としており，FIFOをキャッシュリプレースメントポリシとするキャッシュメモリには対応していない．

2.6 Haqueらの手法

Haqueらの手法 [44–46] はFIFOベースL1キャッシュを対象とする手法である．手法 [44] ではデータ構造として木構造を用いて複数の構成の探索を容易にすることでシミュレーションを高速化している．手法 [45] では手法 [44] のデータ構造に加えてルックアップテーブルとリスト構造を用いてシミュレーションを高速化している．手法 [46] では手法 [44,45] のデータ構造に加えてFIFOキューに関する性質 *Intersection Property* [46] を用いてシミュレーションを高速化している．Intersection Propertyは性質2,3,4から成る．

性質 2. 連想度 A_x の *FIFO*キャッシュにデータ t がキャッシュインするとき，連想度 A_y のキャッシュで t が $2A_x - 3$ 番目以降にキャッシュアウトする優先度とする．連想度 A_x の *FIFO* キャッシュにデータ t が存在するならば連想度 A_y の *FIFO*キャッシュにデータ t が存在する．

性質 3. 連想度 *2,* セット数 s_i のキャッシュ構成 c_i において，すべてのセットの *MRI (Most Recently Inserted)* は，$a > 2$,$s_j > s_i$ となる連想度 a，セット数 s_j のキャッシュ構成 c_j に含まれる．

性質 4. 連想度 *2,* セット数 s_i のキャッシュ構成 c_i において，すべてのセットの *NMRI (Non Most Recently Inserted)* は，*MRI*のキャッシュイン以降にアクセスがあった場合，$a > 2$,$s_j > s_i$

となる連想度 a，セット数 s_j のキャッシュ構成 c_j に含まれる．

Intersection Property によりキャッシュ構成シミュレーションを行うデータ構造とこれを扱うアルゴリズムが提案されている．

2.7 本章のまとめ

本章ではキャッシュメモリの説明とキャッシュメモリを構成する際のパラメータ，キャッシュ構成シミュレーションを説明した．関連研究としてシングルプロセッサアーキテクチャのキャッシュを対象とする Janapsatya らの手法や CRCB 手法，CRCB-U 手法，Haque らの手法を紹介した．既存手法は Inclusion Property や Intersection Property などのキャッシュリプレースメントポリシに関わる性質を利用し，一部のキャッシュ構成でシミュレーションすることで他のキャッシュ構成も同時にシミュレーションしている．同時にシミュレーションできるデータ構造を作ることでキャッシュ構成シミュレーションを高速化できる．

第3章

シングルコアプロセッサアーキテクチャにおける柔軟なリプレースメントポリシをベースとするキャッシュの高速キャッシュ構成シミュレーション

3.1 本章の概要

本章はシングルコアプロセッサアーキテクチャにおける柔軟なリプレースメントポリシをベースとするキャッシュの高速キャッシュ構成シミュレーション手法を提案し評価する．特にFIFO/PLRU ベースキャッシュを対象とする．最初に既存の LRU ベースキャッシュを対象とするキャッシュ構成シミュレーション高速化手法 CRCB 手法がより柔軟なキャッシュリプレースメントポリシをもつキャッシュアーキテクチャのキャッシュ構成シミュレーションに適用できることを証明する．CRCB 手法が適用できるようなキャッシュセットデータの追い出し優先度を与える．次に FIFO ベースキャッシュについて異なるキャッシュ構成間のキャッシュセットに成立する関係性を考察する．同様に PLRU ベースキャッシュについて異なるキャッシュ構成間のキャッシュセットに成立する関係性を考察する．シングルコアプロセッサにおける FIFO/PLRU ベースキャッシュについて異なるキャッシュ構成間のキャッシュセットに成立する関係性により高速にキャッシュ構成シミュレーションを行えるデータ構造とこれを用いたアルゴリズムを提案する．最後に提案手法を計算機上で実装し評価する．

3.2 CRCB 手法が適用可能なアーキテクチャ

CRCB 手法が適用可能なアーキテクチャを考察する．CRCB 手法は LRU を前提に，次の 2 つの性質を利用している [50].

性質 5. あるメモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットしたとき，$s' = s, 2s, 4s, \ldots, s_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b, a') でもキャッシュヒットとなる．

性質 6. キャッシュ構成 (s, b, a) に対して連続したメモリアクセス，$n-1$ 番目のメモリアクセスと n 番目のメモリアクセスが発生し，*2* つのメモリアクセスのタグ t とインデックス i が等しい場合，$s' = s, 2s, 4s, \ldots, s_m, b' = b, 2b, 4b, \ldots, b_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b', a') で n 番目のメモリアクセスはキャッシュヒットする．

CRCB1 手法は性質 1 を利用するものであり，CRCB2 手法は性質 2 を利用するものである．

性質 5 は以下の 2 つの小性質に分割できる．

小性質 1. あるメモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットしたとき，$s' = s, 2s, 4s, \ldots, s_m$ のキャッシュ構成 $(s', b, 1)$ でもキャッシュヒットとなる．

小性質 2. あるメモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットしたとき，$a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s, b, a') でもキャッシュヒットとなる．

性質 6 は以下の 2 つの小性質に分割できる．

小性質 3. キャッシュ構成 (s, b, a) に対して連続したメモリアクセス，$n-1$ 番目のメモリアクセスと n 番目のメモリアクセスが発生し，*2* つのメモリアクセスのタグ t とインデックス i が等しい場合，$s' = s, 2s, 4s, \ldots, s_m, b' = b, 2b, 4b, \ldots, b_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b', a') で *2* つのメモリアクセスのタグ t' とインデックス i' が等しい．

小性質 4. キャッシュ構成 (s, b, a) に対して連続したメモリアクセス，$n-1$ 番目のメモリアクセスと n 番目のメモリアクセスが発生し，*2* つのメモリアクセスのタグ t とインデックス i が等しい場合，n 番目のメモリアクセスはキャッシュヒットする．

[50] は LRU を前提に上記の性質 5，性質 6 が成立することを証明している．

ここで，以下の性質 7 が成り立つキャッシュリプレースメントポリシに着目する．

性質 7. キャッシュ構成 $c = (s, b, a)$ に対する *2* つのメモリアクセス A_j，A_k がある．

i) キャッシュ構成 c で A_j，A_k のインデックスが等しく，両方 $S(c, i)$ にキャッシュアクセスされる．

ii) メモリアクセス A_k は A_j よりも後に発生する．

iii) *2* つのメモリアクセス A_j，A_k の間に，$S(c, i)$ に関わるメモリアクセスは発生しない．

上記の前提条件をもとに以下の状態を満足する．

a) メモリアクセス A_j の後，A_k が発生するまで $S(c, i)$ の内部データが保存される．

b) A_j と A_k のタグが一致するならば，A_k でキャッシュヒットし，キャッシュリプレースメントポリシによる $S(c, i)$ の更新の必要はない．

性質 7 が成り立つキャッシュリプレースメントポリシでは以下の定理が成り立つ．

定理 3. 性質 *7* が成り立つキャッシュリプレースメントポリシでは性質 *5*，性質 *6* が成り立つ．

まず，性質 7 が成り立つキャッシュリプレースメントポリシでは小性質 1，小性質 2，小性質 3，小性質 4 が成り立つことを証明する．

補題 1. 性質 *7* が成り立つキャッシュリプレースメントポリシでは，小性質 *1* が成り立つ．性質7が成り立つキャッシュリプレースメントポリシにおいて，キャッシュ構成 $c = (s, b, a)$ で以下の条件を満足するメモリアクセス A_j，A_k が発生．

i) キャッシュ構成 c で A_j，A_k のインデックスが等しく，両方 $S(c, i)$ にキャッシュアクセスされる．

ii) メモリアクセス A_k は A_j よりも後に発生する．

iii) 2つのメモリアクセス A_j，A_k の間に，$S(c, i)$ に関わるメモリアクセスは発生しない．

A_k が $S(c, i)$ でキャッシュヒットしたと仮定する．キャッシュ構成 c は連想度が1なので $S(c, i)$ のブロックは1個である．A_j，A_k のインデックスが等しいため，両方 $S(c, i)$ にキャッシュアクセスされる．そのブロックに A_k の直前にキャッシュインあるいはキャッシュヒットしたデータは A_j である．よって A_j と A_k のタグは一致する．キャッシュ構成 $(s, b, 1)$ において A_j，A_k のタグとインデックスがそれぞれ等しいとき，$s' = s, 2s, 4s, \ldots, s_m$ のキャッシュ構成 $c' = (s', b, 1)$ でも A_j，A_k のタグとインデックス i' がそれぞれ等しい．A_j が $S(c', i')$ にキャッシュアクセスしたあと，次に $S(c', i')$ に関わるメモリアクセスは A_k である．したがってキャッシュ構成 c' でも A_k がキャッシュヒットする．メモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットしたとき，$s' = s, 2s, 4s, \ldots, s_m$ のキャッシュ構成 $(s', b, 1)$ でもキャッシュヒットとなることが証明された． □

補題 2. 性質 *7* が成り立つキャッシュリプレースメントポリシでは，小性質 *2* が成り立つ．性質7が成り立つキャッシュリプレースメントポリシにおいて，キャッシュ構成 $c = (s, b, a)$ で以下の条件を満足するメモリアクセス A_j，A_k が発生．

i) キャッシュ構成 c で A_j，A_k のインデックスが等しく，両方 $S(c, i)$ にキャッシュアクセスされる．

ii) メモリアクセス A_k は A_j よりも後に発生する．

iii) 2つのメモリアクセス A_j，A_k の間に，$S(c, i)$ に関わるメモリアクセスは発生しない．

A_k が $S(c, i)$ でキャッシュヒットしたと仮定する．小性質1と同様に $a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 $c' = (s, b, a')$ で A_j と A_k のタグとインデックスがそれぞれ等しい．したがって

キャッシュ構成 c' でも A_k がキャッシュヒットする．メモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットしたとき，$a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s, b, a') でもキャッシュヒットとなることが証明された． □

補題 3. 性質 *7* が成り立つキャッシュリプレースメントポリシでは，小性質 *3* が成り立つ．性質7が成り立つキャッシュリプレースメントポリシにおいて，キャッシュ構成 $c = (s, b, a)$ で連続するメモリアクセス A_j，A_k が発生．キャッシュ構成 c で A_j と A_k のタグとインデックスがそれぞれ一致すると仮定する．A_j，A_k の下位 $\lg b$ ビット以外のビット列は一致する．よって $s' = s, 2s, 4s, \ldots, s_m, b' = b, 2b, 4b, \ldots, b_m, a' = 1, 2, 3, \ldots, a_m$ となるキャッシュ構成 (s', b', a') で2つのメモリアクセスのタグ t' とインデックス i' が等しい．キャッシュ構成 (s, b, a) に対して連続したメモリアクセス，$n-1$ 番目のメモリアクセスと n 番目のメモリアクセスが発生し，2つのメモリアクセスのタグ t とインデックス i が等しい場合，$s' = s, 2s, 4s, \ldots, s_m, b' = b, 2b, 4b, \ldots, b_m, a' = 1, 2, 3, \ldots, a_m$ のキャッシュ構成 (s', b', a') で2つのメモリアクセスのタグ t' とインデックス i' が等しいことが証明された． □

補題 4. 性質 *7* が成り立つキャッシュリプレースメントポリシでは，小性質 *4* が成り立つ．性質7が成り立つキャッシュリプレースメントポリシにおいて，キャッシュ構成 $c = (s, b, a)$ で連続するメモリアクセス A_j，A_k が発生．キャッシュ構成 c で A_j と A_k のタグとインデックスがそれぞれ一致すると仮定する．メモリアクセス A_j，A_k は性質7の前提条件を満たすのでメモリアクセス A_j の後，A_k が発生するまで $S(c, i)$ の内部データが保存される．よって A_k でキャッシュヒットする．キャッシュ構成 (s, b, a) に対して連続したメモリアクセス，$n-1$ 番目のメモリアクセスと n 番目のメモリアクセスが発生し，2つのメモリアクセスのタグ t とインデックス i が等しい場合，n 番目のメモリアクセスはキャッシュヒットすることが証明された． □

小性質1と小性質2から性質5が導出され，小性質3と小性質4から性質6が導出される．よって，性質7が成り立つキャッシュリプレースメントポリシでは性質5と性質6が成り立つのでCRCB手法が適用できる [50]．

LRUだけでなくFIFOやPLRUでも性質7が成立するのでFIFOやPLRUをキャッシュリ

プレースメントポリシとするキャッシュ構成シミュレーションでもCRCB手法の基本アルゴリズムが適用できる．CRCB手法を適用し拡張することでFIFO，PLRUベースキャッシュに対しても高速にキャッシュ構成シミュレーションが可能となる．

3.3 キャッシュセットの動作が同じキャッシュ構成を統合するデータ構造

3.3.1 FIFOをキャッシュリプレースメントポリシとするL1キャッシュメモリの性質

FIFOはFirst In First Outの略称であり，先に入ったデータから先に追い出されるアルゴリズムである．図3.1に要素数4のFIFO優先度付きキューにアクセスがあったときの動作の例を示す．FIFO優先度付きキューにデータA，B，C，Dが存在することを考える．最初にDのデータが入ったとき，そのデータはFIFO優先度付きキューのIn側に配置される．その後，C，B，Aのデータが入るたびにFIFO優先度付きキューの内部データはひとつずつOut側に移動する．つまり，データが新しく挿入される度に既に存在するデータの優先度が大きくなる．FIFOをキャッシュリプレースメントポリシとするキャッシュ構成において，セット$S(c,i)$を優先度付きキューとみなしたとき，ブロックの追い出し優先度を後にキャッシュインした順に$0,1,2,\ldots,a_0-1$と定義することで，常に最大の優先度をもつブロックから追い出される．図3.2に要素数4のFIFO優先度付きキューにおけるデータの優先度の例を示す．最も先に入ったデータであるDの追い出し優先度が最も高く，次に追い出されるデータはDである．

セット数とブロックサイズが同じで連想度が異なるキャッシュ構成$c_0=(s,b,2)$と$c_1=(s,b,4)$があるとする．インデックスiのセット$S(c_0,i)$のブロックが全て優先度がそのままで$S(c_1,i)$に含まれるとき，iでc_0はc_1に含まれると定義し，$c_0 \subsetneq_i c_1$と表現する．含まれていないとき$c_0 \not\subset_i c_1$と表現する．同じデータでも入っているブロックの優先度が異なる場合は含まれていない．図3.3にキャッシュ構成c_0とc_1のインデックス0101と1111のセットの関係を示す．図3.3(a)では$c_0 \subset_{0101} c_1$だが図3.3(b)では$c_0 \not\subset_{1111} c_1$である．

セット数とブロックサイズが同じで連想度が$a<a'$で異なるキャッシュ構成$c=(s,b,a)$と

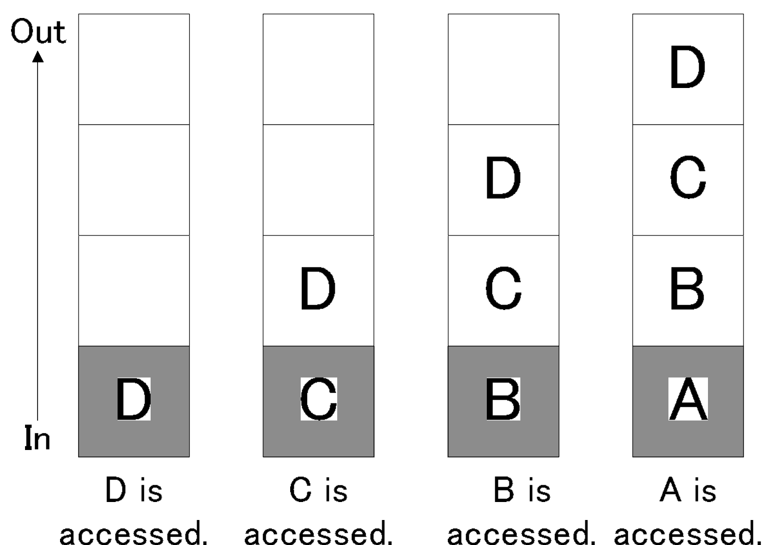

図 3.1: FIFO 優先度付きキューの動作 ([57] に掲載).

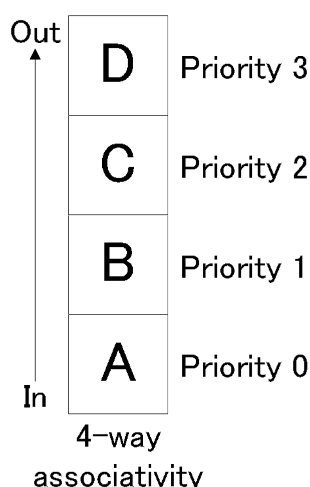

図 3.2: FIFO 優先度付きキューの優先度の例 ([57] に掲載).

$c' = (s, b, a')$ があるとする．関係 $c \underset{i}{\subseteq} c'$ が成り立つとき，以下の 2 つの性質が成立する．

性質 8. キャッシュ構成 c, c' に対してメモリアクセス A が発生したとする．$j < a$ となる c' のインデックス i のセットの優先度 j のブロック $S(c', i)_j$ でキャッシュヒットしたならば，c でもキャッシュヒットし関係 $c \underset{i}{\subseteq} c'$ は保存される．$a \leq j < a'$ となる c' のインデックス i のセットの優先度 j のブロック $S(c', i)_j$ でキャッシュヒットしたならば，c でもキャッシュミスし $c \underset{i}{\not\subseteq} c'$ となる．

Proof. $j < a$ となる $S(c', i)_j$ でキャッシュヒットしたならば，$c \underset{i}{\subseteq} c'$ より，$S(c, i)_j = S(c', i)_j$ な

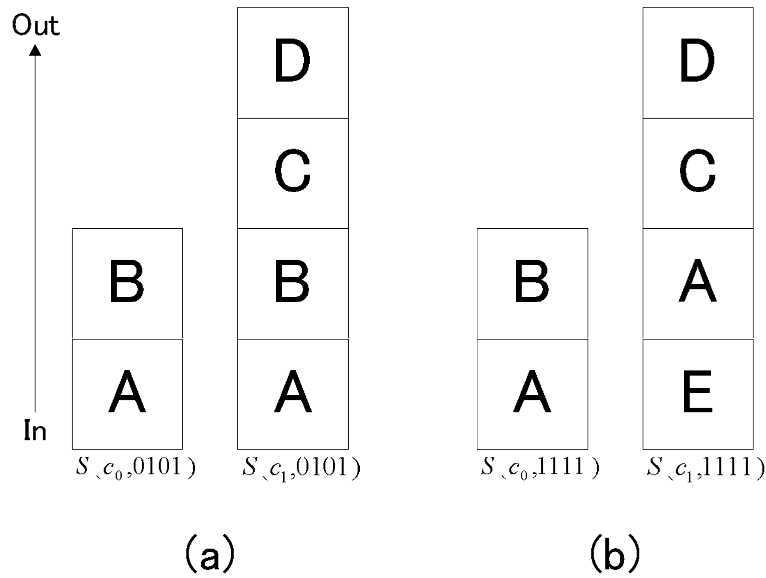

図 3.3: (a) $c_0 \subset_i c_1$ (b) $c_0 \not\subset_i c_1$ ([57] に掲載) .

ので c でキャッシュヒットする．FIFO ベースキャッシュはキャッシュヒットのときブロックの移動が行われないため $c \underset{i}{\subset} c'$ が保存される．$a \leq j < a'$ となる $S(c', i)_j$ でキャッシュヒットしたならば，該当ブロックは $0 \leq j' < j$ となる $S(c', i)'_j$ に存在しない．$c \underset{i}{\subset} c'$ より，$S(c, i)'_j = S(c', i)'_j$ なので c でキャッシュミスする．$S(c, i)_j$ でキャッシュミスとなりブロックの更新が行われるが，$S(c', i)_j$ ではキャッシュヒットするため更新されない．よって $c \underset{i}{\not\subset} c'$ となる． □

性質 9. キャッシュ構成 c，c' に対してメモリアクセス A が発生したとする．A が c' でキャッシュミスしたとき，c でもキャッシュミスし，関係 $c \underset{i}{\subset} c'$ は保存される．

Proof. c' でキャッシュミスしたならば，該当ブロックは $0 \leq j < a'$ となる $S(c', i)'_j$ に存在しない．$c \underset{i}{\subset} c'$ より，$0 \leq j < a$ で $S(c, i)_j = S(c', i)_j$ なので c でもキャッシュミスする．$S(c, i)$ と $S(c', i)$ でともにブロックの更新が行われるため，$c \underset{i}{\subset} c'$ は保存される． □

性質 8，性質 9 より $c \underset{i}{\subset} c'$ のとき，連想度の高い構成 c' のセット $S(c', i)$ を探索，更新すれば連想度が低い構成 c のセット $S(c, i)$ は探索，更新を省略できる．図 3.4 に $c \underset{i}{\subset} c'$ となるキャッシュ構成 c，c' にメモリアクセスが発生したときの動作の例を示す．c，c' の両方でキャッシュヒット，あるいは両方でキャッシュミスしたとき $c \underset{i}{\subset} c'$ は保存される．c でキャッシュミス，c' でキャッシュヒットしたとき $c \underset{i}{\not\subset} c'$ となる．

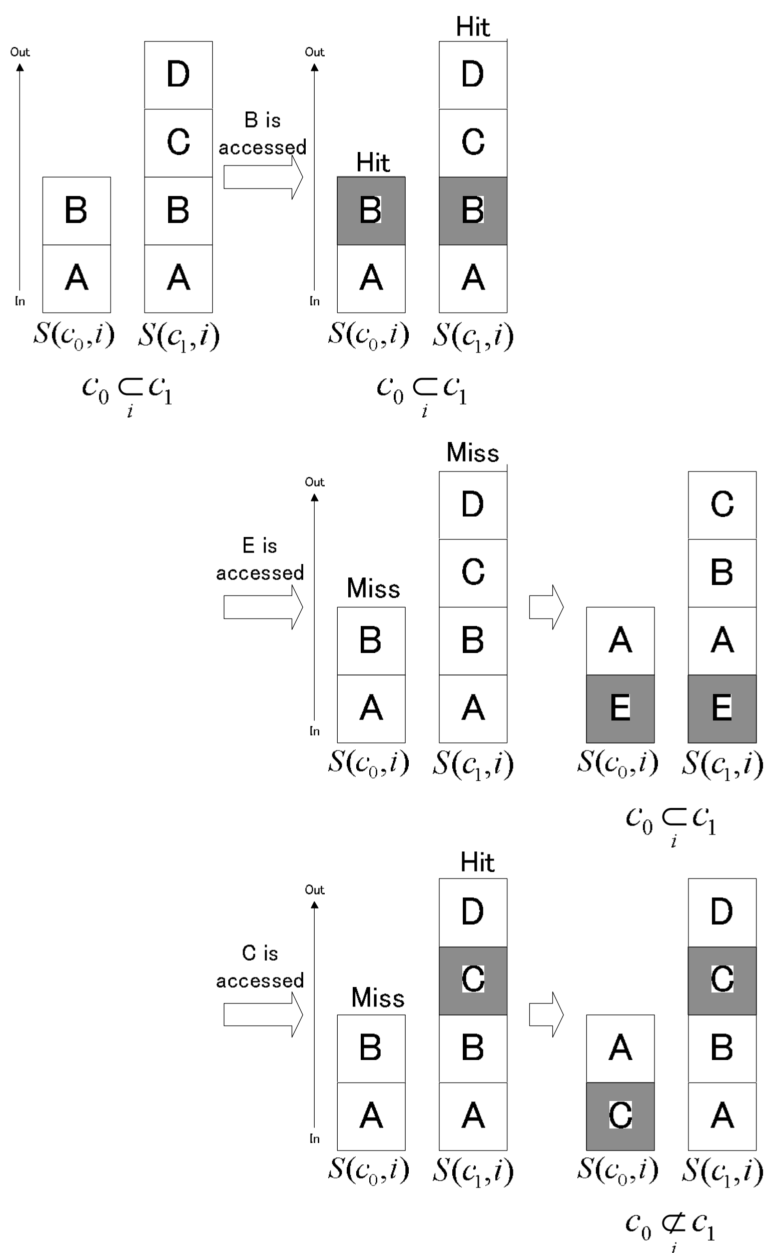

図 3.4: $c \subseteq_i c'$ のときの動作の例 ([57] に掲載).

3.3.2 PLRU をキャッシュリプレースメントポリシとする L1 キャッシュメモリの性質

PLRU は Pseudo LRU の略称であり，疑似 LRU と言われる．LRU はハードウェアで実装する場合，複雑な機構が必要になりコストがかかる．そのため LRU のように高いキャッシュヒット率を持ちつつ，実装が簡単なキャッシュリプレースメントポリシが考案された．PLRU は局所性を利用した LRU に似たアルゴリズム全般を指し，特定のアルゴリズムを指すもの

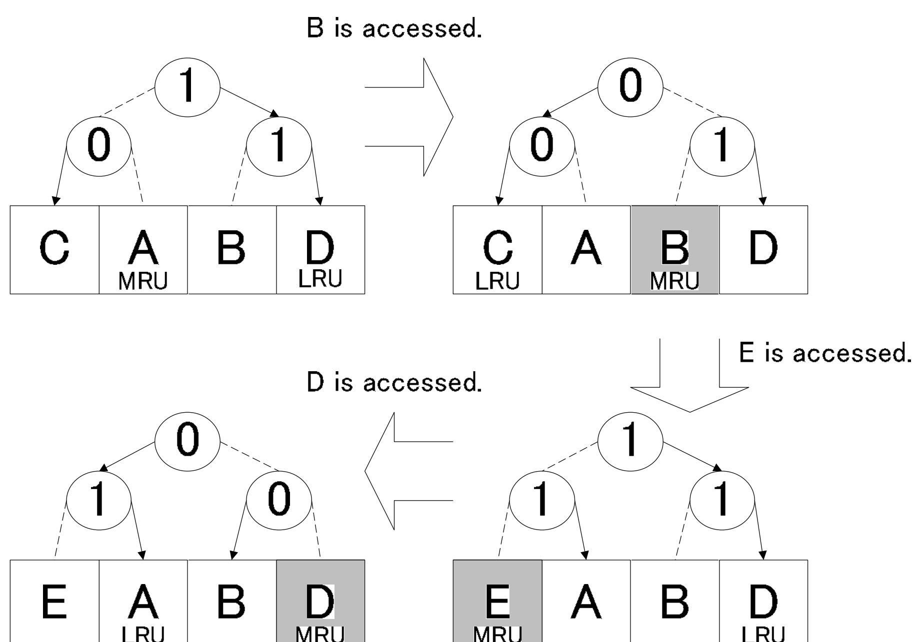

図 3.5: PLRUt の動作の例 ([57] に掲載).

ではない．本論文では管理方式が簡単で組込み用途でも一般的に用いられている PLRU として木構造を基にした PLRU を対象とする [55]．木構造を基にした PLRU を PLRUt と呼ぶ．PLRUt はブロックを葉とし，葉に至るまでのノードが 1bit を保持する完全二分木で表わされる．それぞれのノードが保持する bit は左右どちらの子が古い参照かを指し示し，bit が 0 ならば左の子，1 ならば右の子が古い参照であることを意味する．あるデータが参照されると根から葉に至るノードにおいて，これらの葉やノードが古い参照にならないように bit が設定される．キャッシュミスが起きた場合，各ノードの bit が示す古い参照側の子を参照していった LRU の葉のブロックを追い出し，それまでのノードの bit を反転する．図 3.5 に PLRUt の動作の例を示す．図において各ノードから出る矢印は古い参照を表す．根から出た矢印の先をたどると最も古い参照である LRU に到達する．初期状態で LRU は D である．このとき最も新しい参照は A である．B にメモリアクセスが発生したとき，根から B までの道程上のノードの矢印は B を指さない方向に切り替わる．よって LRU は C になる．次にキャッシュ上にない E へのメモリアクセスが発生したとき LRU の C がキャッシュアウトし E がキャッシュインする．根から E の道程上のノードが更新される．さらに，D にメモリアクセスが発生したとき根から D の道程上のノードが更新される．

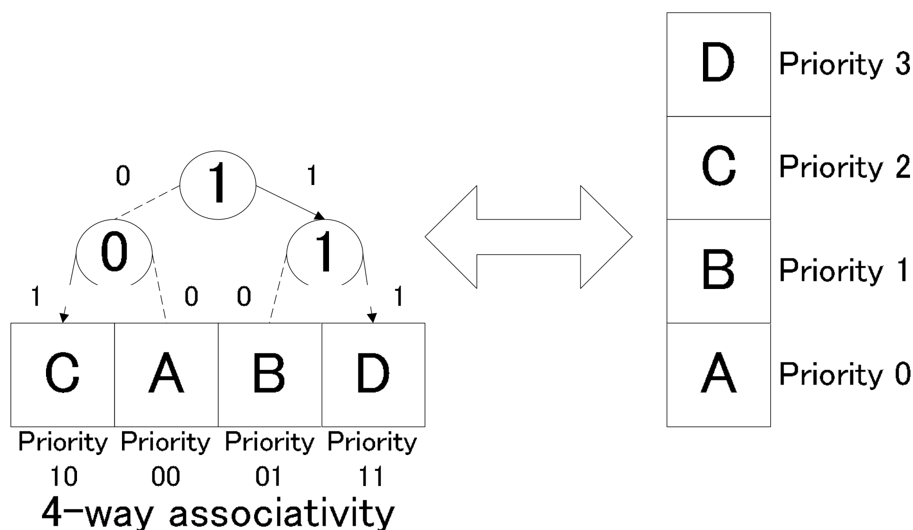

図 3.6: PLRUt の配列的表示と優先度 ([57] に掲載).

PLRUt の追い出し優先度を根から葉に至るまでの道程のノードの bit の値から決定する．今ある葉 t に着目する．t の優先度 p を次のように定義する．深さ n のノードが表す bit が葉 t の方向を示しているならば優先度 p の $n\,bit$ 目を 1 とする．この定義により，次に追い出されるブロックの追い出し優先度が最も高くなる．図 3.6 に PLRUt の木構造を配列的に変換し優先度を表したものを示す．キャッシュミスが発生した場合，D のブロックが次に追い出される．

定義した優先度によると小さい木構造は大きい木構造の部分木となるため，性質 8，性質 9 は FIFO と同様に PLRUt でも成立する．

FIFO と同様に PLRUt ベースキャッシュでもキャッシュ構成同士の関係 $c \underset{i}{\subseteq} c'$ を適用できる．

3.4 高速化アルゴリズムの提案

性質 7，性質 8，性質 9 を利用することで FIFO，PLRUt をキャッシュリプレースメントポリシとする L1 キャッシュのキャッシュ構成シミュレーションの高速化アルゴリズムを提案する．セット数，ブロックサイズが同じで連想度が異なるキャッシュ構成 $c = (s, b, a)$，$c' = (s, b, a')$ を考える．$c \underset{i}{\subseteq} c'$ が成り立つとき，c，c' はインデックス i で同じキャッシュセットグループに属すると定義する．図 3.7 にキャッシュセットグループの例を示す．$c_0 \underset{i}{\subseteq} c_1$ であり，$c_1 \underset{i}{\subseteq} c_2$ である．よって $S(c_0, i)$，$S(c_1, i)$，$S(c_2, i)$ は同じキャッシュセットグループに属する．メモ

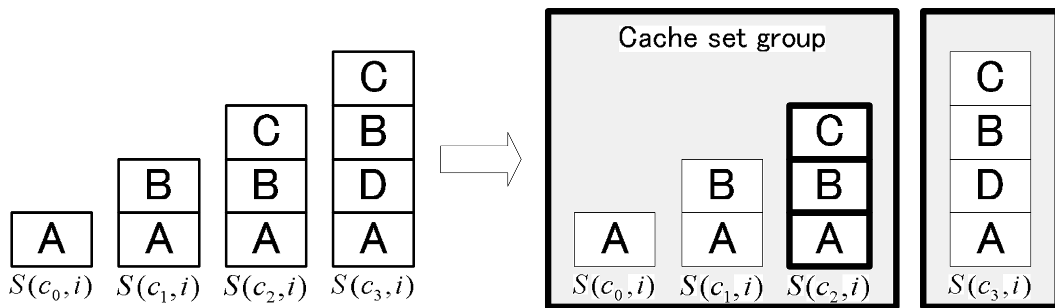

図 3.7: キャッシュセットグループの例 ([57] に掲載).

リアクセスに対し $S(c_2, i)$ を探索，更新すれば同じキャッシュセットグループ内の連想度が小さいキャッシュ構成で探索，更新を省略できる．

キャッシュセットグループを使ったキャッシュ構成シミュレーションの高速化手法を提案する．提案手法をCSG(Cache Set Group) 手法と呼称する．アルゴリズムを以下に示す.

CSG 手法

1) 初期値としてセット数とブロックサイズが同じで連想度が異なるキャッシュ構成について，インデックスが等しいセットを全て同じキャッシュセットグループに属するとする.

2) あるメモリアクセスが発生した場合，探索するキャッシュ構成のブロックサイズ b に b_0 を，セット数 s に s_0 を，連想度 a に 2 を代入する.

3) ブロックサイズ b の値により CRCB2 手法によって判定省略可能ならステップ 10) に行く.

4) メモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットすれば CRCB1 手法により $s' = s, 2s, \ldots, s_m, a' = 1, 2, \ldots, a_m$ となるキャッシュ構成 (s', b, a') でもキャッシュヒットとなり判定省略可能なのでステップ 9) に行く，メモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュミスならば該当セットを更新する.

5) キャッシュ構成 (s, b, a) の該当セットと同じキャッシュセットグループにある連想度が最大のキャッシュ構成 (s, b, a_M) でキャッシュヒット/ミスを判定する．キャッシュヒットならステップ 6) に，キャッシュミスなら更新してステップ 7) に行く.

6) ステップ 5) で優先度 j でキャッシュヒットしたとする．同じキャッシュセットグループに存在する $j < a'$ となる (s, b, a') でキャッシュヒットとなり判定省略可能である．$a' \leq j$ と

なる (s,b,a') でキャッシュミスとなり更新を行う．$(s,b,a') \underset{i}{\not\subset} (s,b,a_M)$ となるのでキャッシュセットグループを分割する．

7) a に a_M+1 を代入する．(s,b,a) が探索範囲ならステップ 5) に行く．

8) a に 2 を s に $2s$ を代入する．(s,b,a) が探索範囲ならステップ 4) に行く．

9) s に s_0 を b に $2b$ を代入する．(s,b,a) が探索範囲ならステップ 3) に行く．

10) メモリアクセスが存在するならステップ 2) に行く．メモリアクセスが存在しないなら終了する．

3.5 提案手法の評価

FIFO ベース L1 キャッシュのキャッシュ構成シミュレーションと PLRU ベース L1 キャッシュのキャッシュ構成シミュレーションに対して計算機上で CSG 手法を実装し，従来手法と実行時間を比較した．FIFO ベースキャッシュでは，Dinero IV [58]，CRCB 手法を FIFO に適用した手法，CRCB 手法を FIFO に適用した手法にさらに CSG 手法を加えた手法を用いてキャッシュ構成シミュレーションを行った．入力データはメモリアクセストレースであり，出力はキャッシュヒット/ミス回数である．使用した計算機は CPU が AMD 1.3GHz であり，メインメモリが 16GB の P C である．提案アルゴリズムは C 言語で実装した．Dinero IV はキャッシュ構成を指定しメモリアクセストレースを入力することで単一のキャッシュ構成に対しキャッシュヒット/ミス回数をシミュレートするアプリケーションである．Dinero IV は指定できるキャッシュリプレースメントポリシとして LRU や FIFO に対応しており，キャッシュシミュレータとして一般的である．アプリケーションとして MediaBench [59] の複数のアプリケーションを使った．MediaBench は画像，音声，動画を扱うためのベンチマークである．またアプリケーションのメモリアクセストレースを得るのに SimpleScalar [60] を使った．

シミュレーションの対象とするキャッシュ構成はキャッシュサイズが 256,512,. . . ,4194304Bytes，セット数が 32,64,. . . ,524288，ブロックサイズが 8,16,. . . ,1024Bytes，連想度が 1,2,4,. . . ,16 の計 380 構成である．

表 3.1 に Dinero IV と CRCB 手法，CRCB 手法 +CSG 手法のキャッシュ構成シミュレー

ション時間の比較結果を示す．CRCB 手法 +CSG 手法では Dinero IV に比べ最大 208 倍の高速化となった．Dinero IV と CRCB 手法，CRCB 手法 +CSG 手法のキャッシュ構成シミュレーションの出力においてキャッシュヒット/ミス数は一致した．よってシミュレーションは正しく実行された．

全探索手法，CRCB 手法を PLRUt に適用した手法，CRCB 手法を PLRUt に適用した手法にさらに CSG 手法を加えた手法を用いてキャッシュ構成シミュレーションを行った．使用する計算機環境，入力データ，探索するキャッシュ構成は FIFO ベースキャッシュのキャッシュ構成シミュレーションにおける評価実験と同じものを用いる．

表 3.2 に全探索手法と CRCB 手法，CRCB 手法 +CSG 手法のキャッシュ構成シミュレーション時間の比較結果を示す．CRCB 手法 +CSG 手法では全探索手法に比べ最大 249 倍の高速化となった．全探索手法と CRCB 手法，CRCB 手法 +CSG 手法のキャッシュ構成シミュレーションの出力においてキャッシュヒット/ミス数は一致した．よってシミュレーションは正しく実行された．PLRU ベースキャッシュについて全探索手法と提案手法で malloc 関数による動的なメモリ確保の量を計測した．全探索手法では瞬間的に 14MB (14,680,064Bytes) を動的にメモリ確保した．提案手法では 132MB (138,853,376Bytes) を動的にメモリ確保した．malloc 関数による動的にメモリ確保はプログラム実行時に処理しており，入力するメモリアクセストレースのアプリケーションの種類に依存しない．提案手法は全探索手法に比べメモリサイズを 10 倍使用した．ただし，メモリ使用量が大きくなったとしても，最近の PC が数十 GB のメインメモリを持つのが一般的であることを考えると，十分に実用に耐えうる．

3.6　本章のまとめ

本章は既存の LRU ベースキャッシュを対象とするキャッシュ構成シミュレーション高速化手法 CRCB 手法がより柔軟なキャッシュリプレースメントポリシをもつキャッシュアーキテクチャのキャッシュ構成シミュレーションに適用できることを証明した．次に FIFO/PLRU ベースキャッシュについて異なるキャッシュ構成間に成立する関係性を考察した．FIFO/PLRU ベースキャッシュについて異なるキャッシュ構成間に成立する関係性により高速にキャッシュ構成シミュレーションを行えるデータ構造とこれを用いたアルゴリズムを提案した．提案手法

表 3.1: FIFO ベースキャッシュの Dinero IV，CRCB 手法，CRCB 手法+CSG 手法の実行時間比較.

	Dinero IV	CRCB	CRCB+CSG
	CPU time [sec]	CPU time [sec]	CPU time [sec]
ADPCM E	67 (1)	1.01 (0.015)	0.96 (0.014)
ADPCM D	68 (1)	1.04 (0.015)	0.98 (0.014)
ADPCM E (I)	669 (1)	4.83 (0.007)	4.30 (0.006)
JPEG E	666 (1)	9.28 (0.014)	8.92 (0.013)
JPEG D	156 (1)	3.51 (0.023)	3.23 (0.021)
EPIC E	1047 (1)	28.82 (0.028)	26.54 (0.025)
EPIC D	224 (1)	8.89 (0.040)	7.95 (0.035)
G721 E	5833 (1)	29.80 (0.005)	28.81 (0.005)
G721 D	5980 (1)	29.31 (0.005)	28.64 (0.005)

表 3.2: PLRUt ベースキャッシュの全探索手法，CRCB 手法，CRCB 手法+CSG 手法の実行時間比較.

	全探索手法	CRCB	CRCB+CSG
	CPU time [sec]	CPU time [sec]	CPU time [sec]
ADPCM E	82 (1)	1.45 (0.018)	1.03 (0.013)
ADPCM D	81 (1)	1.51 (0.019)	1.07 (0.013)
ADPCM E (I)	988 (1)	5.77 (0.006)	4.87 (0.005)
JPEG E	787 (1)	11.16 (0.014)	10.88 (0.014)
JPEG D	186 (1)	4.59 (0.025)	3.95 (0.021)
EPIC E	1232 (1)	35.01 (0.028)	36.32 (0.029)
EPIC D	264 (1)	10.89 (0.041)	10.53 (0.040)
G721 E	7073 (1)	30.92 (0.004)	29.30 (0.004)
G721 D	7243 (1)	30.25 (0.004)	29.04 (0.004)

を計算機上で実装し評価した．FIFO ベースキャッシュのキャッシュ構成シミュレーションでは提案手法は DineroIV に比べ最大 208 分の 1 に実行時間を削減できた．FIFO ベースキャッシュのキャッシュ構成シミュレーションでは提案手法は CRCB 手法を FIFO ベースキャッシュに適用させた手法に比べ最大 11%実行時間を削減できた．PLRU ベースキャッシュのキャッシュ構成シミュレーションでは全探索手法に比べ最大 249 分の 1 に実行時間を削減できた．PLRU ベースキャッシュのキャッシュ構成シミュレーションでは提案手法は CRCB 手法を PLRU ベースキャッシュに適用させた手法に比べ最大 29%実行時間を削減できた．

第4章

シングルコアプロセッサアーキテクチャにおけるFIFOベースキャッシュの高速キャッシュ構成シミュレーション

4.1 本章の概要

第 3 章では FIFO/PLRU ベースキャッシュを対象とするキャッシュ構成シミュレーションの高速化手法を提案し評価した．FIFO ベースキャッシュを対象とするキャッシュ構成シミュレーションにはさらなる高速化できる要因があるため，本章では FIFO ベースキャッシュを対象とするキャッシュ構成シミュレーションの高速化について述べる．最初に FIFO ベースキャッシュの性質からキャッシュ構成シミュレーション高速化手法を考察する．次に FIFO ベースキャッシュについて異なるキャッシュ構成間のキャッシュセットに成立する関係性を考察し，FIFO ベースキャッシュについて異なるキャッシュ構成間のキャッシュセットに成立する関係性により高速にキャッシュ構成シミュレーションを行えるデータ構造とこれを用いたアルゴリズムを提案する．最後に提案手法を計算機上で実装し評価する．

4.2 データ構造の更新不要なキャッシュ操作を考慮した複数キャッシュ構成を統合するデータ構造

LRU や PLRU のキャッシュリプレースメントポリシではキャッシュヒットしたとき，該当ブロックがセットのどの優先度に存在するかが重要な問題となった．LRU や PLRU の優先度付きキューではアクセスがあったとき，優先度の更新が行われる．複数のキャッシュ構成で同じ優先度をもつ，あるいは優先度が更新されないキャッシュヒットのときのみ更新を省略できる．FIFO 優先度付きキューではアクセスがあったとき，データが存在すれば優先度の更新が行われない．これは FIFO 特有の性質である．本節では FIFO ベースキャッシュに特有の性質を考察し，関係性を定義する．

FIFO の性質を利用することで 2 章で定義した関係 $\underset{i}{\subseteq}$ を**緩く**適用することを考える．セット数とブロックサイズが同じで連想度が $a_0 < a_1$ で異なる構成 $c_0 = (s, b, a_0)$ と $c_1 = (s, b, a_1)$ があるとする．インデックス i のセット $S(c_0, i)$ のブロックが全て $S(c_1, i)$ に含まれるとき，i で c_0 は c_1 に**緩く**含まれると定義し，$c_0 \underset{i}{\subsetsim} c_1$ と表現する．セット数とブロックサイズが同じで連想度が $a < a'$ で異なるキャッシュ構成 $c = (s, b, a)$ と $c' = (s, b, a')$ があるとする．関係 $c \underset{i}{\subsetsim} c'$ が成り立つとき，以下の性質が考えられる．

性質 10. キャッシュ構成 c，c' に対してメモリアクセス A が発生したとする．キャッシュ構成 c のセット $S(c,i)$ でキャッシュヒットしたならば，$S(c',i)$ でもキャッシュヒットし，関係 $c \underset{i}{\tilde{\subseteq}} c'$ は保存される．

Proof. $c \underset{i}{\tilde{\subseteq}} c'$ より，$S(c,i)$ のブロックはすべて $S(c',i)$ に含まれる．よって $S(c,i)$ でキャッシュヒットするならば，$S(c',i)$ でもキャッシュヒットする．キャッシュヒットしたとき FIFO では内容が更新されないので $c \underset{i}{\tilde{\subseteq}} c'$ は持続する． □

性質 11. キャッシュ構成 c，c' に対してメモリアクセス A が発生したとする．$S(c,i)$ でキャッシュミスし，$S(c',i)$ でキャッシュヒットしたとき，関係 $c \underset{i}{\tilde{\subseteq}} c'$ は保存される．

Proof. $c \underset{i}{\tilde{\subseteq}} c'$ より，キャッシュミスによりキャッシュインするブロックを除き $S(c,i)$ のブロックはすべて $S(c',i)$ に含まれる．キャッシュインするブロックは $S(c',i)$ でキャッシュヒットしたので $S(c',i)$ に含まれるため，$c \underset{i}{\tilde{\subseteq}} c'$ は保存される． □

性質 12. キャッシュ構成 c，c' に対してメモリアクセス A が発生したとする．$S(c,i)$，$S(c',i)$ でキャッシュミスしたとき，関係 $c \underset{i}{\tilde{\subseteq}} c'$ は保存されない可能性がある．

Proof. $S(c',i)$ でキャッシュミスによりキャッシュアウトしたブロックが $S(c,i)$ に更新後も存在するブロックである場合，$c \underset{i}{\tilde{\subseteq}} c'$ が成立しない．よって $S(c,i)$，$S(c',i)$ でキャッシュミスしたとき，関係 $c \underset{i}{\tilde{\subseteq}} c'$ は保存されない． □

性質 10 より，$c_0 \underset{i}{\tilde{\subseteq}} c_1$ のとき，c_0 でキャッシュヒットしたならば，c_1 でもキャッシュヒットするため探索を省略できる．

4.3 高速化アルゴリズムの提案

性質 10，性質 11，性質 12 を利用することで FIFO をキャッシュリプレースメントポリシとする L1 キャッシュのキャッシュ構成シミュレーションのさらなる高速化アルゴリズムを提案する．セット数，ブロックサイズが同じで連想度が異なるキャッシュ構成 $c = (s,b,a)$，$c' = (s,b,a')$ を考える．$c \underset{i}{\tilde{\subseteq}} c'$ が成り立つとき，c，c' はインデックス i で同じスーパーキャッ

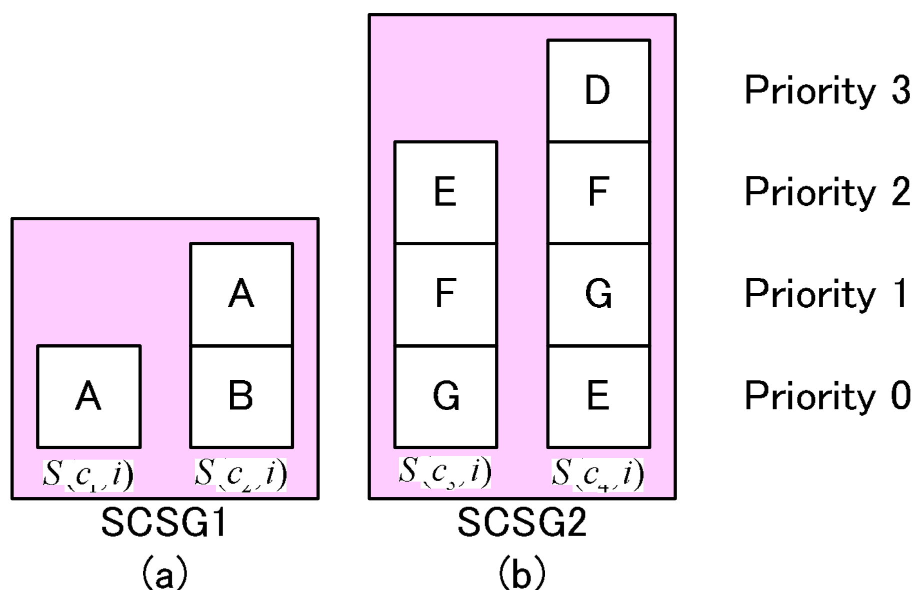

図 4.1: スーパーキャッシュセットグループの例 ([61] に掲載).

シュセットグループに属すると定義する．図 4.1 にスーパーキャッシュセットグループの例を示す.

$c_1 \underset{i}{\tilde{\subsetneq}} c_2$ であり，$c_3 \underset{i}{\tilde{\subsetneq}} c_4$ である．よって $S(c_1, i)$ と $S(c_2, i)$，$S(c_3, i)$ と $S(c_4, i)$ はそれぞれ同じスーパーキャッシュセットグループに属する.

スーパーキャッシュセットグループ内の全てのセットにおいてキャッシュヒット/ミス判定を行うとき，連想度が小さいセットでキャッシュヒットすればそれより大きい連想度を持つセットでもキャッシュヒットする．連想度が小さい順にセットを探索することでキャッシュヒット/ミス判定が省略できる可能性がある.

スーパーキャッシュセットグループを使ったキャッシュ構成シミュレーションの高速化手法を提案する．提案手法を SCSG(Super Cache Set Group) 手法と呼称する．アルゴリズムを以下に示す.

SCSG 手法

1) 初期値としてセット数とブロックサイズが同じで連想度が異なるキャッシュ構成について，インデックスが等しいセットを全て同じスーパーキャッシュセットグループに属するとする.

2) あるメモリアクセスが発生した場合，探索するキャッシュ構成のブロックサイズ b に b_0

を，セット数 s に s_0 を，連想度 a に 2 を代入する．

3) ブロックサイズ b の値により CRCB2 手法によって判定省略可能ならステップ 10) に行く．

4) メモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュヒットすれば CRCB1 手法により $s' = s, 2s, \ldots, s_m$, $a' = 1, 2, \ldots, a_m$ となるキャッシュ構成 (s', b, a') で判定省略可能なのでステップ 9) に行く，メモリアクセスがキャッシュ構成 $(s, b, 1)$ でキャッシュミスならば該当セットを更新する．

5) キャッシュ構成 (s, b, a) がキャッシュヒットならば，(s, b, a) の該当セットと同じスーパーキャッシュセットグループにある連想度が a より大きいセットでもキャッシュヒットする．スーパーキャッシュセットグループに含まれるセットの最大の連想度を a_M とする．a に a_M を代入してステップ 7) へ行く．

6) キャッシュ構成 (s, b, a) がキャッシュミスし，(s, b, a) の該当セットと同じスーパーキャッシュセットグループにある連想度が a より小さいセットでもキャッシュミスしていればスーパーキャッシュセットグループを分割する．

7) a に $a + 1$ を代入する．(s, b, a) が探索範囲ならステップ 5) に行く．

8) a に 2 を s に $2s$ を代入する．(s, b, a) が探索範囲ならステップ 4) に行く．

9) s に s_0 を b に $2b$ を代入する．(s, b, a) が探索範囲ならステップ 3) に行く．

10) メモリアクセスが存在するならステップ 2) に行く．メモリアクセスが存在しないなら終了する．

CSG 手法と SCSG 手法は複合して用いることができる．図 4.2 にキャッシュセットグループを用いた手法と提案手法の複合手法の例を示す．SCSG1 のキャッシュヒット/ミスを判定する場合，まず CSG1 のキャッシュヒット/ミスを判定する．CSG1 でキャッシュヒットするセットが存在すれば CSG2 の全てのセットでキャッシュヒットする．CSG1 のキャッシュヒット/ミスを判定する場合，まず $S(c_2, i)$ のキャッシュヒット/ミスを判定する．$S(c_1, i)$ と $S(c_2, i)$ は同じ優先度での内容が同じため，$S(c_2, i)$ のキャッシュヒットしたときの優先度を調べれば $S(c_1, i)$ のキャッシュヒット/ミス判定が省略できる．

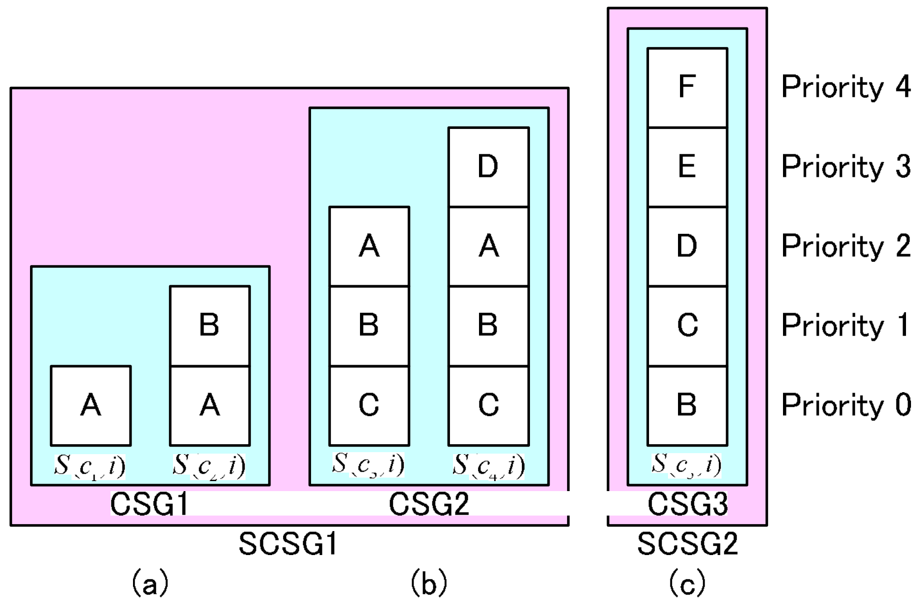

図 4.2: CSG 手法と SCSG 手法の複合手法 ([61] に掲載).

4.4 提案手法の評価

FIFO ベース L1 キャッシュのキャッシュ構成シミュレーションに対して計算機上で SCSG 手法を実装し，従来手法と実行時間を比較した．

Dinero IV,CRCB 手法，CRCB 手法に CSG 手法を加えた手法，CRCB 手法に SCSG 手法を加えた手法，CRCB 手法に CSG 手法と SCSG 手法を加えた手法を用いてキャッシュ構成シミュレーションを行った．入力データはメモリアクセストレースであり，出力はキャッシュヒット/ミス回数である．使用した計算機は CPU が Intel Xeon 3.0GHz であり，メインメモリが 4GB の PC である．アプリケーションとして MediaBench [59] の複数のアプリケーションを使った．MediaBench は画像，音声，動画を扱うためのベンチマークである．またメモリアクセス履歴データを得るのに SimpleScalar [60] を使った．シミュレーションの対象とするキャッシュ構成はキャッシュサイズが 256,512,. . . ,134217728Bytes，セット数が 32,64,. . . ,524288，ブロックサイズが 8,16,. . . ,1024Bytes，連想度が 1,2,4,. . . ,512 の計 925 構成である．

表 4.1 に FIFO に対する Dinero IV，CRCB 手法，CRCB 手法 +CSG 手法，CRCB 手法 +SCSG 手法，CRCB 手法 +CSG 手法 +SCSG 手法のキャッシュ構成シミュレーション時間の比較結果を示す．CRCB 手法に SCSG 手法を加えた手法は CRCB 手法に CSG 手法を加え

表 4.1: FIFOベースキャッシュのDinero IV，CRCB手法，CRCB手法 +CSG手法，CRCB手法 +SCSG手法，CRCB手法 +CSG手法 +SCSG手法の実行時間比較.

	メモリアクセス回数	Dinero IV CPU time [sec]	CRCB CPU time [sec]	CRCB+CSG CPU time [sec]	CRCB+SCSG CPU time [sec]	CRCB+CSG+SCSG CPU time [sec]
ADPCM (E)	522151	244.9 (32.1)	8.04 (1.05)	7.63 (1)	8.04 (1.05)	8.02 (1.05)
ADPCM (D)	521795	251.7 (32.7)	8.11 (1.05)	7.69 (1)	8.06 (1.05)	7.98 (1.04)
JPEG (E)	5183724	2223.2 (81.7)	35.7 (1.31)	27.2 (1)	25.5 (0.94)	27.9 (1.03)
JPEG (D)	1216590	546.2 (41.7)	16.2 (1.24)	13.1 (1)	12.4 (0.95)	12.9 (0.98)
EPIC (E)	8000017	3300.8 (36.2)	111.6 (1.23)	91.1 (1)	89.9 (0.99)	88.0 (0.97)
EPIC (D)	2314113	1011.6 (29.7)	49.4 (1.45)	34.1 (1)	40.9 (1.20)	35.1 (1.03)
G721 (E)	47898002	19415.9 (295.1)	53.8 (0.82)	65.8 (1)	36.7 (0.56)	42.0 (0.64)
G721 (D)	48998888	19921.6 (495.6)	48.0 (1.19)	40.2 (1)	35.7 (0.89)	39.4 (0.98)

た手法に比べ最大44%の高速化となった.

Dinero IVはメモリアクセス1回ごとにキャッシュシミュレーションを実行するため，メモリアクセス回数とDinero IVの実行時間はおおよそ比例すると考えられる．ADPCM (E)とADPCM (D)は比較的メモリアクセス回数が少なく実行時間が短い．G721 (E)とG721 (D)は比較的メモリアクセス回数が多く実行時間が長い．表4.1よりCRCB手法+CSG手法とCRCB手法+SCSG手法の実行時間を比較すると，アプリケーションがADPCM (E)とADPCM (D)などメモリアクセス回数が少ない場合は，CRCB手法+CSG手法の実行時間はCRCB手法+SCSG手法の実行時間より短い．一方，アプリケーションがG721 (E)とG721 (D)などメモリアクセス回数が多い場合は，CRCB手法+SCSG手法の実行時間はCRCB手法+CSG手法の実行時間より短い．図4.2のようにキャッシュセット$S(c_1, i)$, $S(c_2, i)$において$c_1 \subseteq_i c_2$ならば$c_1 \tilde{\subseteq}_i c_2$である．1つのキャッシュセットグループに属するキャッシュセットは1つのスーパーキャッシュセットに属することができる．図2.4で示したメモリアクセスが発生したときに図2.4 (e)ではキャッシュセット$S(c_0, 1000)$, $S(c_1, 1000)$において$c_0 \not\subseteq_{1000} c_1$であるが，$c_0 \tilde{\subseteq}_{1000} c_1$である．キャッシュセット$S(c_0, 1000)$, $S(c_1, 1000)$は1つのキャッシュセットグループに属することはできないが，1つのスーパーキャッシュセットグループに属することはできる．グループの数が等しい場合，データ構造としてシミュレーションするキャッシュセットが少ないためCSG手法の方がSCSG手法より高速に動作する．メモリアクセス回数が少ない場合，キャッシュセットグループの数がスーパーキャッシュセットグループの数とそれほど変わらないため，CSG手法の方が高速に動作することが期待できる．メモリ

アクセス数が多い場合，図 2.4 のようなメモリアクセスによりキャッシュセットグループの数がスーパーキャッシュセットグループの数より非常に多くなるため，SCSG 手法の方が高速に動作することが期待できる．よってメモリアクセス回数が少ないアプリケーションに対してはCRCB 手法+CSG 手法を用いると高速にキャッシュ構成シミュレーションが実行できると考えられる．同様にメモリアクセス回数が多いアプリケーションに対してはCRCB 手法+SCSG 手法を用いると高速にキャッシュ構成シミュレーションが実行できると考えられる．CRCB 手法+CSG 手法と CRCB 手法+SCSG 手法，CRCB 手法+CSG 手法+SCSG 手法の実行時間を比較すると，CRCB 手法+CSG 手法+SCSG 手法ではオーバヘッドが大きくなり CRCB 手法+CSG 手法や CRCB 手法+SCSG 手法より実行時間がかかるが，まれに EPIC (E) のように CRCB 手法+CSG 手法＋ SCSG 手法の方が CRCB 手法+CSG 手法や CRCB 手法+SCSG 手法より実行時間が短くなる．実際にこれらの手法を用いてキャッシュ構成シミュレーションを行う際には入力するアプリケーションのメモリアクセス回数に応じて実行するアルゴリズムを適応させると効果的にキャッシュ構成シミュレーションが高速化できると考えられる．

4.5　本章のまとめ

本章では FIFO ベースキャッシュについて異なるキャッシュ構成間に成立する関係性を考察した．FIFO ベースキャッシュについて異なるキャッシュ構成間に成立する関係性により高速にキャッシュ構成シミュレーションを行えるデータ構造とこれを用いたアルゴリズムを提案した．提案手法を計算機上で実装し評価した．CRCB 手法に提案した SCSG 手法を加えた手法は DineroIV に比べ最大 558 分の 1 に実行時間を削減できた．CRCB 手法に提案した SCSG 手法を加えた手法は CRCB 手法に CSG 手法を加えた手法に比べ最大 44%実行時間を削減できた．

第5章

マルチコアプロセッサアーキテクチャにおける高速キャッシュ構成シミュレーション

5.1　本章の概要

第3章，第4章ではシングルコアプロセッサアーキテクチャのキャッシュを対象にキャッシュ構成シミュレーションの高速化手法を提案，評価した．マルチコアプロセッサアーキテクチャのキャッシュにはキャッシュコヒーレンシプロトコルがあり，第3章，第4章の手法は適用できない．本章ではマルチコアプロセッサアーキテクチャのキャッシュを対象にキャッシュ構成シミュレーションの高速化手法を提案し評価する．最初にマルチコアプロセッサキャッシュが持つキャッシュコヒーレンシプロトコルを説明する．キャッシュコヒーレンシプロトコルが定める状態について，メモリアクセスが発生した場合の状態遷移を説明する．次にマルチコアプロセッサアーキテクチャにおけるキャッシュの動作を分類する．分類したキャッシュの動作回数を数え上げることをマルチコアプロセッサのキャッシュ構成シミュレーションと定める．マルチコアプロセッサのキャッシュ構成シミュレーションでは，キャッシュセットに入ってるデータが同じでも状態が異なる可能性がある．異なるキャッシュ構成のキャッシュセットが2つあるとき，データが同じでも状態が異なる場合は同じように扱えない．異なる複数の状態を表す拡張状態を導入する．拡張状態によりデータが同じキャッシュセットならば状態が異なっても統合的に情報を保持できるデータ構造とこれを扱うアルゴリズムを提案する．最後に提案手法を実装し，全探索手法と実行時間を比較する．

5.2　キャッシュコヒーレンシ

マルチコアプロセッサのそれぞれのコアには個別にプライベートキャッシュメモリが存在している．プライベートキャッシュメモリで同じデータに読み書きする場合は矛盾が発生する可能性がある．2つのコア，コア0とコア1があるとする．それぞれのコアが同じデータのインクリメントをするとき，インクリメントは2回行われることが期待される．ここでコア0とコア1が連続してデータを読み込み，その後連続してデータを書き込むとする．最終的に書き込まれたデータは1回しかインクリメントが行われていない．このようにして矛盾が発生する．矛盾が発生しないようにする機構としてキャッシュコヒーレンシプロトコルが存在する．キャッシュコヒーレンシプロトコルはキャッシュ内のデータに状態を結びつけて一

表 5.1: MESI プロトコルの持つ状態の性質.

	Modified	Exclusive	Shared	Invalid
データの有効性	有効	有効	有効	無効
他のキャッシュとの関係性	排他	排他	共有	-
メインメモリとの整合性	変更有り	変更無し	変更無し	-

貫性を保つように管理するプロトコルである．キャッシュコヒーレンシプロトコルには，ライト・インバリデート型とライト・アップデート型がある [62].

本章ではキャッシュコヒーレンシプロトコルとして最も標準的な MESI プロトコル [63] を採用する．MESI プロトコルはライト・インバリデート型プロトコルである．ライト・インバリデートはあるプロセッサからライト命令があったとき，他のプロセッサのキャッシュ上のデータを無効化することで一貫性を保つ．ライト・インバリデートを用いるとキャッシュヒット率が低くなるが，トラフィックは少なくなる．MESI プロトコルでキャッシュ上の各データは Modified, Exclusive, Shared, Invalid の 4 つの状態に遷移する．表 5.1 に各状態が満たすべき性質を示す．ライト命令により変更があったデータは，他のプロセッサでリード命令があったとき必ずメインメモリと整合性をとってから他のプロセッサのキャッシュに入ることになる．よって MESI プロトコルでは他のプロセッサのキャッシュと整合性がとれていてメインメモリとの整合性がとれていない状態は存在しない．

マルチコアプロセッサアーキテクチャでは他のプロセッサのデータアクセスを監視してキャッシュコヒーレンシを保つ機構がある．この機構により該当キャッシュのデータと同じデータが他のプロセッサに発見されることをスヌープヒットと呼ぶ．リード命令やライト命令があったときの MESI プロトコルの状態遷移を図 5.1 に示す．

リード命令がキャッシュヒット 状態に変更なし (図 5.1(a)).

リード命令がキャッシュミス

- **スヌープヒット** スヌープヒットしたプロセッサからデータを受け取ると同時に，該当データの状態を Shared にする．他のプロセッサはスヌープによりデータの状態を Shared に変更する (図 5.1(b)).
- **スヌープミス** 該当データの状態を Exclusive にする (図 5.1(c)).

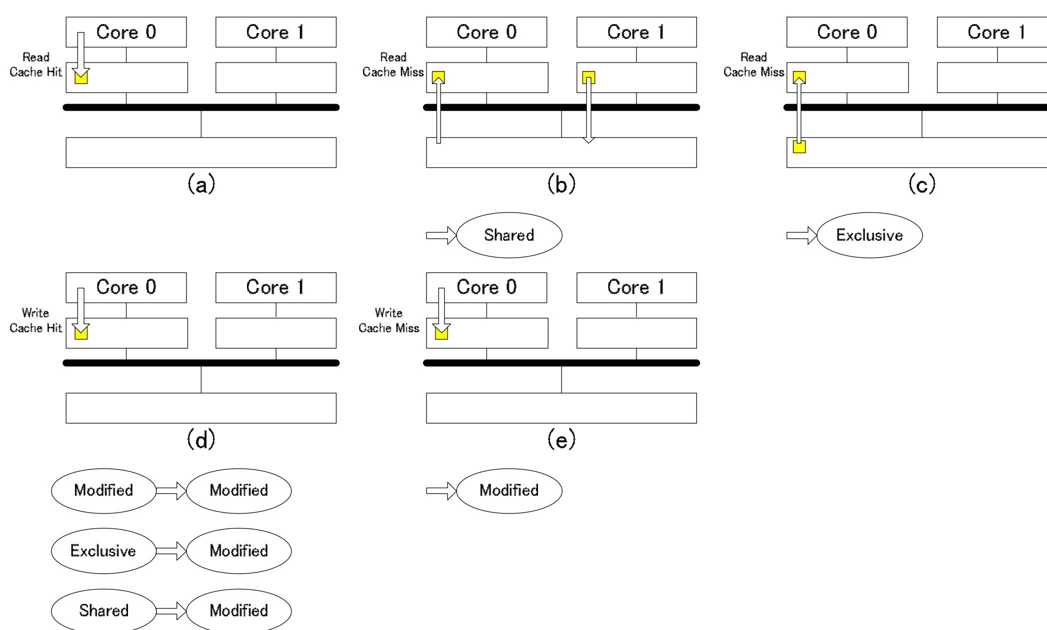

図 5.1: MESI プロトコルの状態遷移.

ライト命令がキャッシュヒット (図 5.1(d))

Modified 状態に変更なし.

Exclusive 該当データの状態を Modified にする.

Shared 該当データの状態を Modified にする. 他のプロセッサのキャッシュのデータの状態を Invalid にする.

ライト命令がキャッシュミス (図 5.1(e))

スヌープヒット 該当データの状態を Modified にする. 他のプロセッサのキャッシュのデータの状態を Invalid にする.

スヌープミス 該当データの状態を Modified にする.

5.3 マルチコアプロセッサアーキテクチャにおけるキャッシュ動作の分類

本章の対象アーキテクチャをキャッシュコヒーレンシプロトコルが MESI プロトコル, キャッシュリプレースメントポリシが LRU の 2 コアプロセッサプライベート L1 キャッシュのアーキテクチャとする. 対象とするアーキテクチャを図 5.2 に示す. Core 0 と Core 1 の各プロ

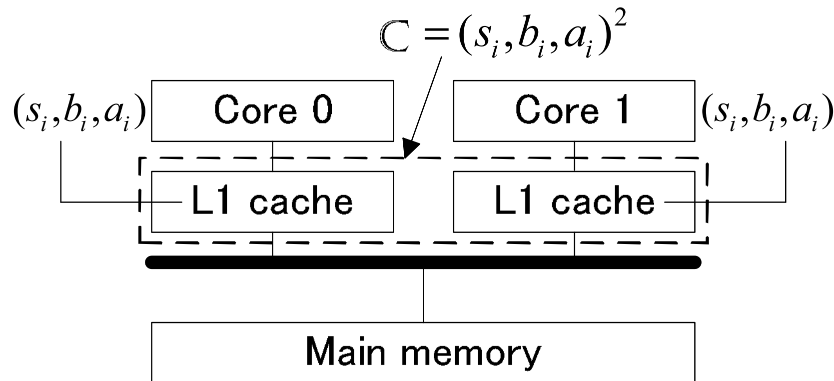

図 5.2: 対象アーキテクチャ ([64] に掲載).

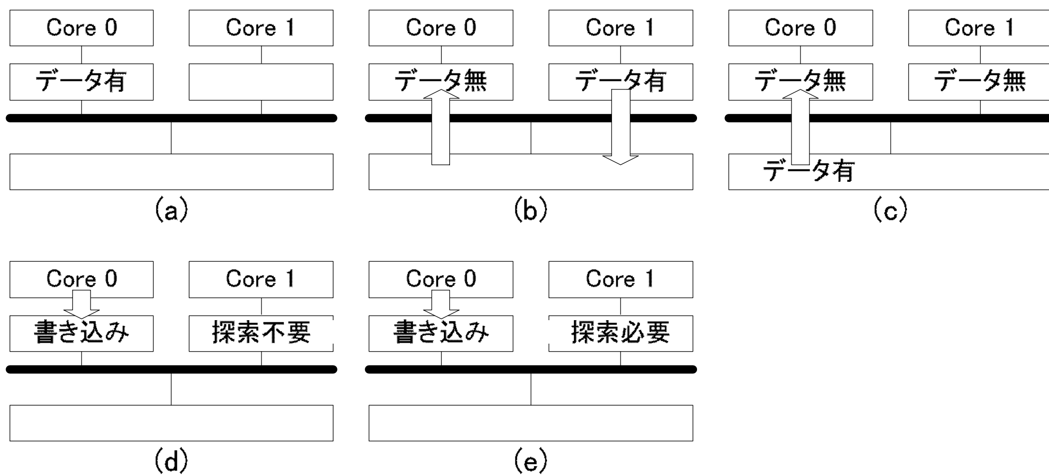

図 5.3: キャッシュ構成シミュレーションで回数を測定するべき状況.

セッサそれぞれのキャッシュメモリの構成は同じとする．Core 0 と Core 1 のキャッシュメモリのそれぞれのキャッシュ構成が共に (s, b, a) であるとき，アーキテクチャ全体のキャッシュ構成は $\mathbb{C} = (s, b, a)^2$ と表現する．

キャッシュシミュレーションの目的はキャッシュヒット/ミス回数を測定して消費エネルギーや速度を計算することである．マルチコアプロセッサのキャッシュシミュレーションでは，メモリアクセスの発生したコアのキャッシュメモリのキャッシュヒット/ミス回数だけでなく，他のコアのキャッシュメモリのスヌープヒット/ミス回数も含めて測定する必要がある．マルチコアプロセッサのキャッシュ構成シミュレーションで回数を数えるべき状況を以下に示す．

リード命令がキャッシュヒットした場合 Core 0 でリード命令が発生し，このメモリアクセスが Core 0 の L1 キャッシュでキャッシュヒットした場合を考える．該当データのキャッ

シュコヒーレンシプロトコルの状態に関わらずCore 1のL1キャッシュを探索する必要はない．キャッシュメモリ，メインメモリ間でデータの通信は発生しない．図5.3(a)にこの状況を示す．

リード命令がキャッシュミスした場合 Core 0でリード命令が発生し，このメモリアクセスがCore 0のL1キャッシュでキャッシュミスした場合を考える．該当データがCore 1のL1キャッシュに存在するかどうか調べるためCore 1のキャッシュを探索する必要がある．Core 1でスヌープヒットしたとき，キャッシュメモリ間でデータの通信が発生する．図5.3(b)にこの状況を示す．Core 1でスヌープミスしたとき，キャッシュメモリとメインメモリ間でデータの通信が発生する．図5.3(c)にこの状況を示す．

ライト命令がキャッシュヒットした場合 Core 0でライト命令が発生し，このメモリアクセスがCore 0のL1キャッシュでキャッシュヒットした場合を考える．該当データのキャッシュコヒーレンシプロトコルの状態がModifiedまたはExclusiveのとき，Core 1のキャッシュに同じデータは存在しないため，Core 1のキャッシュを探索する必要はない．図5.3(d)にこの状況を示す．該当データのキャッシュコヒーレンシプロトコルの状態がSharedのとき，Core 1のキャッシュに同じデータは存在する可能性があるため，Core 1のキャッシュを探索をする必要がある．図5.3(e)にこの状況を示す．

ライト命令がキャッシュミスした場合 Core 0でライト命令が発生し，このメモリアクセスがCore 0のL1キャッシュでキャッシュミスした場合を考える．該当データのキャッシュコヒーレンシプロトコルの状態に関わらずCore 1のL1キャッシュを探索する必要がある．図5.3(e)にこの状況を示す．

マルチコアプロセッサのメモリアクセストレースとはメモリアドレスのシーケンスであり，各メモリアドレスはどのプロセッサからのアクセス命令か，リード命令かライト命令かを付加情報として持つ．

キャッシュ構成を変えながら図5.3(a),(b),(c),(d),(e)の5つの状況が発生した回数をそれぞれ数えることで，各キャッシュ構成でのキャッシュメモリ間でのデータの通信回数，メインメモリとキャッシュメモリでのデータの通信回数がわかる．データの通信回数からキャッシュメモリ動作時の遅延時間や消費エネルギーを計算できる．

また，エネルギーを計算するためには，ライト命令でキャッシュメモリに書き込まれたデータがメインメモリに書き戻される回数を数える必要がある．キャッシュメモリからメインメモリに書き戻されるタイミングは「状態 Modified のブロックがキャッシュアウトするとき」と「リード命令がキャッシュミスし，異なるコアの状態 Modified のブロックでスヌープヒットするとき」である．ライト命令の発生した回数は全キャッシュ構成で一致するため，書き戻し回数が減るようなライト命令の発生回数を数えれば書き戻しの回数が計算できる．書き戻し回数の減るライト命令とは「ライト命令が状態 Modified のブロックにキャッシュヒットするとき」と「ライト命令がキャッシュミスし，異なるコアの状態 Modified のブロックでスヌープヒットするとき」である．このような状況を数えればよい．

5.4 キャッシュコヒーレンシを考慮した複数キャッシュ構成を統合するデータ構造

キャッシュ構成シミュレーションは，パラメータを変化させながら単一構成のキャッシュシミュレーションを複数回行うことで実現できる．しかし，この手法は現実的でない時間がかかる可能性がある．複数のキャッシュ構成をまとめて同時にシミュレーションすることができれば実行時間を短縮できる．複数のキャッシュ構成をまとめて同時にシミュレーションするためには，同時に複数のキャッシュ構成を表現するデータ構造が必要となる．そして，ひとつのデータ構造を探索，更新することで複数のキャッシュ構成で探索，更新が行われるようなデータ構造を構築することができれば高速なキャッシュ構成シミュレーションを実現できる可能性がある．本章では連想度に着目し，連想度の異なる複数のキャッシュ構成をひとつのデータ構造で表現する手法に焦点を当てる．

図 5.2 の 2 コアプロセッサの Core 0 と Core 1 のキャッシュについてセット数，ブロックサイズが等しく，連想度の異なる 2 つのキャッシュ構成 $\mathbb{C} = (4, 8, 1)^2$ と $\mathbb{C}' = (4, 8, 2)^2$ を考える．これらのキャッシュ構成を対象に同じメモリアクセストレースを与えれば，キャッシュ構成 $\mathbb{C}$ と $\mathbb{C}'$ では同じインデックスのセットを比べたとき，Inclusion Property [50] によって連想度が小さい構成 $\mathbb{C}$ に属するデータは連想度が大きい構成 $\mathbb{C}'$ に属するデータにすべて含まれる．

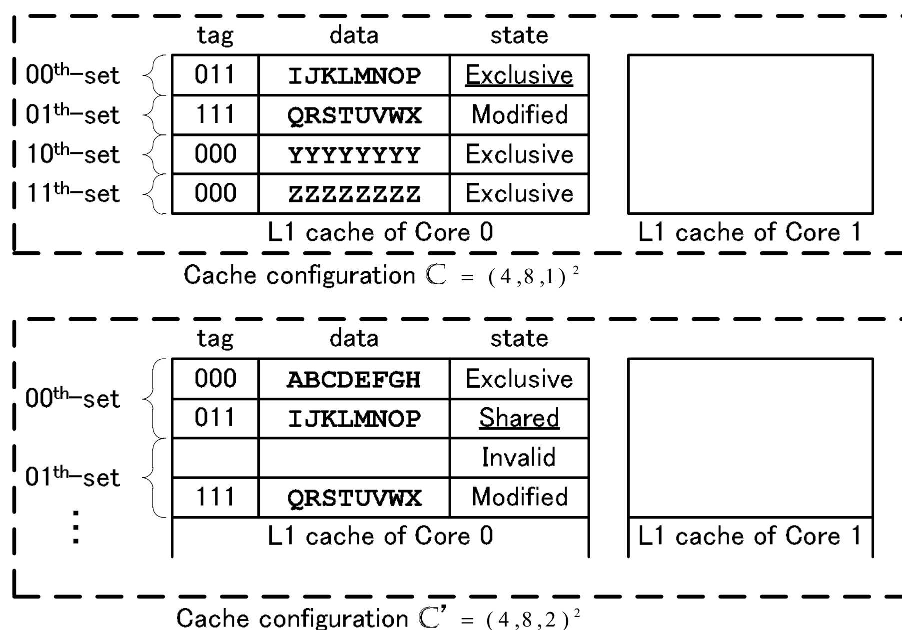

図 5.4: キャッシュメモリの内部状態 ([65] に掲載).

図 5.4 に $\mathbb{C}$ と $\mathbb{C}'$ のそれぞれの Core 0 のキャッシュの内部状態を示す．$\mathbb{C}$ と $\mathbb{C}'$ のそれぞれの Core 0 のキャッシュに対し，インデックス 00 のセットのタグ 011 のブロックにアクセスがあり図 5.4 の内部状態になったとする．アクセスがある前の内部状態では $\mathbb{C}$ のインデックス 00 のセットにタグ 011 のブロックはなく，$\mathbb{C}'$ では状態 Shared でブロックが存在していたとする．$\mathbb{C}$ のインデックス 00 のセットのタグ 011 のブロックのデータは `IJKLMNOP` であり，$\mathbb{C}'$ のインデックス 00 のタグ 011 のブロックのデータも `IJKLMNOP` である．異なるキャッシュ構成間で同じインデックスのセットかつ同じタグならばデータは一致する．一方，$\mathbb{C}$ のインデックス 00 のセットのタグ 011 のブロックの状態は Exclusive であり，$\mathbb{C}'$ のインデックス 00 のタグ 011 のブロックの状態は Shared である．キャッシュコヒーレンシプロトコルにより決定される状態は異なることがある．

まとめると，$\mathbb{C}$ と $\mathbb{C}'$ についてセットを優先度付きキューとみなしたときの各ブロックのデータと優先度は一致するが，キャッシュコヒーレンシプロトコルにより決定される状態は異なることがある．

$\mathbb{C}$ と $\mathbb{C}'$ をひとつのデータ構造で表現する．ここではより連想度が高いキャッシュ構成を表すデータ構造により，それより連想度が低いキャッシュ構成を同時に表す．つまり，$\mathbb{C}'$ のデータ構造のみを保持し $\mathbb{C}$ と $\mathbb{C}'$ をひとつのデータ構造で表現する．ところが前述のように

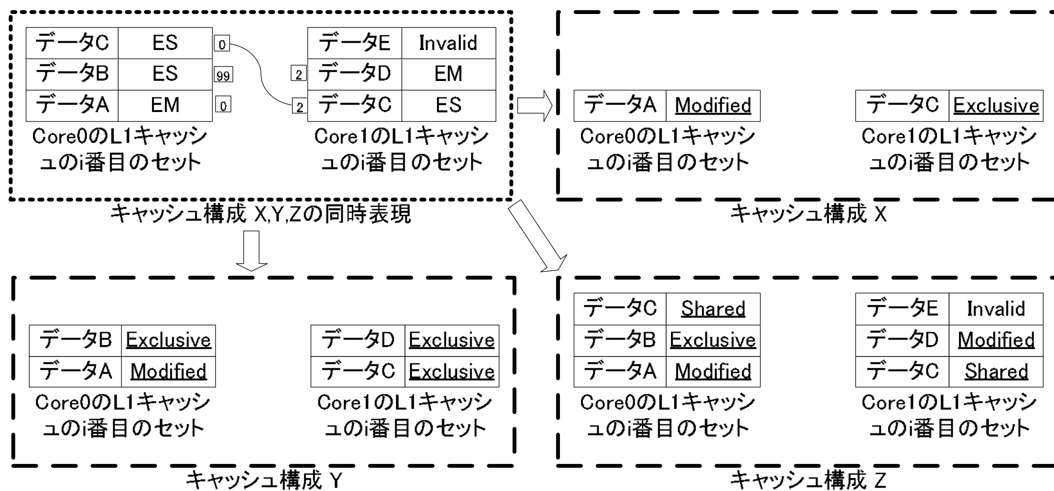

図 5.5: 提案データ構造 ([65] に掲載).

複数のキャッシュ構成に対して同じメモリアクセスが発生してもキャッシュ内のキャッシュコヒーレンシプロトコルの状態が Exclusive と Shared で異なることがある．この問題を解決するために Exclusive と Shared の状態のどちらかを表す拡張状態（super state）ES を導入する．あるキャッシュ構成で拡張状態 ES のデータが Exclusive なのか Shared なのかを判定するため，データごとに境界値（boundary）をつける．表現したいキャッシュ構成の連想度が該当データの境界値よりも大きい場合はそのデータは Shared, 境界値以下の場合は Exclusive とする．同様の問題が状態 Exclusive と Modified でも発生するため，拡張状態 EM，境界値を導入する．図 5.5 に提案データ構造を示す．表現したいキャッシュ構成の連想度が該当データの境界値よりも大きい場合はそのデータは Modified, 境界値以下の場合は Exclusive とする．また，拡張状態 ES のデータが同じキャッシュ構成の別のコアのキャッシュに同じデータが存在する場合にどの位置にデータが存在するかをポインタで示す．

なお，2 つのキャッシュ構成 $\mathbb{C}$ と $\mathbb{C}'$ において，MESI プロトコルの性質により，ライト命令の後に Shared であったり，ライト命令以外で Shared や Exclusive の状態が Modified になることはない．したがって，Shared と Modified を同時に表現する拡張状態は必要ない．

また，以下の定理 4 と定理 5 に示すように，図 5.5 のデータ構造に 5.5 節のアルゴリズムを用いて，正しくキャッシュメモリの動作をシミュレーションできる．

5.5 高速化アルゴリズムの提案

さらにこのデータ構造を用いてキャッシュメモリの状態を判定，更新するアルゴリズムを提案する．対象アーキテクチャは2コアプロセッサプライベートL1キャッシュである．パラメータの範囲はセット数$s = s_0, 2s_0, \ldots, s_m$，ブロックサイズ$b = b_0, 2b_0, \ldots, b_m$，連想度$a = 1, 2, 3, \ldots, a_m$である．セット数，ブロックサイズが同じで連想度が異なるキャッシュ構成$(s_i, b_i, 1)^2$，$(s_i, b_i, 2)^2$，…，$(s_i, b_i, a_m)^2$に対してひとつデータ構造を用意する．提案アルゴリズムはリード命令が発生したときのアルゴリズムとライト命令が発生したときのアルゴリズムを部分アルゴリズムとする．このアルゴリズムでは各キャッシュ構成における図5.3(a),(b),(c),(d),(e)のそれぞれの状況が発生した回数と書き戻しのないライト命令が発生した回数を数える．キャッシュ構成$\mathbb{C} = (s, b, a)^2$において図5.3(a),(b),(c),(d),(e)の状況が発生した回数をそれぞれ$N_a((s, b, a))$,$N_b((s, b, a))$,$N_c((s, b, a))$,$N_d((s, b, a))$,$N_e((s, b, a))$とする．書き戻しのないライト命令が発生した回数を$N_w((s, b, a))$とする．キャッシュメモリ上のあるブロックをblkとしたとき，その優先度を$p(blk)$，拡張状態を$ss(blk)$，境界値を$bdry(blk)$，ポインタ先のブロックを$ptr(blk)$とする．セット数s，ブロックサイズbのキャッシュ構成に対するのメモリアクセスAのインデックスをi，タグをtとする．

提案アルゴリズムを図5.6に示す．部分アルゴリズムとしてアルゴリズムRとアルゴリズムWを図5.7，図5.8に示す．部分アルゴリズムではセット数s_i，ブロックサイズb_i，連想度$a = 1, 2, 3, \ldots, a_m$のキャッシュ構成を同時に表現するデータ構造Dに対して，Core 0からメモリアクセスAがあったときのアルゴリズムを示している．セット数,ブロックサイズが固定で異なる連想度をもつキャッシュ構成での同時シミュレーションが可能である．

```
while コア j にメモリアクセス A が発生 do
  セット数 s, ブロックサイズ b からタグ t, インデックス i を計算;
  提案データ構造のコア j のキャッシュの i 番目のセットに注目;
  if メモリアクセス A がリード命令 then
    goto アルゴリズム R;
  else if メモリアクセス A がライト命令 then
    goto アルゴリズム W;
  end if
end while
```

図 5.6: 提案アルゴリズム ([65] に掲載).

if there exists a block blk which includes the tag t at the index i on Core 0 **then**
  if $ss(blk)$ == EM **then**
    $N_a((s,b,a))$++, $(p(blk) < a \leq a_m)$;
    $N_c((s,b,a))$++, $(1 \leq a \leq p(blk))$;
    $bdry(blk) \leftarrow \max(bdry(blk), p(blk))$;
  else
    $blk' \leftarrow ptr(blk)$;
    $p' \leftarrow a_m$;
    if blk' != NULL **then**
      $p' \leftarrow p(blk')$;
      $bdry(blk') \leftarrow 0$;
    end if
    $N_a((s,b,a))$++, $(p(blk) < a \leq a_m)$;
    $N_b((s,b,a))$++, $(p' < a \leq p(blk))$;
    $N_c((s,b,a))$++, $(1 \leq a \leq \min(p(blk), p'))$;
    $bdry(blk) \leftarrow \max(bdry(blk), p)$;
    $bdry(blk) \leftarrow \min(bdry(blk), p')$;
  end if
else
  $p' \leftarrow a_m$;
  if there exists a block blk' which includes the tag t at the index i on Core 1 **then**
    $p' \leftarrow p(blk')$;$ss(blk') \leftarrow$ ES; $bdry(blk') \leftarrow 0$;
    $ptr(blk) \leftarrow blk'$;$ptr(blk') \leftarrow blk$;
  end if
  $N_b((s,b,a))$++, $(p' < a \leq a_m)$;
  $N_c((s,b,a))$++, $(1 \leq a \leq p')$;
  Prepare a block blk with tag t;
  $ss(blk) \leftarrow$ ES; $bdry(blk) \leftarrow p'$;
  Put the block blk at index i on Core 0;
end if

図 5.7: アルゴリズム R ([64] に掲載).

if there exists a block blk which includes the tag t at the index i on Core 0 **then**
  if $ss(blk)$ == EM **then**
    $N_d((s, b, a))$++, $(p(blk) < a \leq a_m)$;
    $N_e((s, b, a))$++, $(1 \leq a \leq p(blk))$;
    $N_w((s, b, a))$++, $(\max(p(blk), bdry(blk)) < a \leq a_m)$;
    $bdry(blk) \leftarrow 0$;
  else
    $blk' \leftarrow ptr(blk)$;
    $N_d((s, b, a))$++, $(p(blk) < a \leq bdry(blk))$;
    $N_e((s, b, a))$++, $(\max(p(blk), bdry(blk)) < a \leq a_m)$;
    $N_e((s, b, a))$++, $(1 \leq a \leq p(blk))$;
    $ss(blk) \leftarrow$ EM;$bdry(blk) \leftarrow 0$;
    if blk' != NULL **then**
      $ss(blk') \leftarrow$ Invalid;
    end if
  end if
else
  $N_e((s, b, a))$++, $(1 \leq a \leq a_m)$;
  Prepare a block blk with the tag t;
  $ss(blk) \leftarrow$ EM;$bdry(blk) \leftarrow 0$;
  Put the block blk at index i on Core 0;
  if there exists a block blk' which includes the tag t at the index i on Core 1 **then**
    if $ss(blk')$ ==EM **then**
      $N_w((s, b, a))$++, $(\max(p(blk'), bdry(blk')) < a \leq a_m)$;
      $ss(blk') \leftarrow$ Invalid;
    end if
  end if
end if

図 5.8: アルゴリズム W ([64] に掲載).

表 5.2: 全探索手法と提案手法の出力，図 5.3(a),(b),(c),(d),(e) の 5 つの状況が発生した回数.

		全探索手法					提案手法				
		(a)	(b)	(c)	(d)	(e)	(a)	(b)	(c)	(d)	(e)
FFT	$(8,8,1)^2$	175343	48277	2714903	59096	1329635	175343	48277	2714903	59096	1329635
	$(16,16,4)^2$	858849	199137	1880537	328474	1060257	858849	199137	1880537	328474	1060257
	$(32,32,16)^2$	1850405	300963	787155	635702	753029	1850405	300963	787155	635702	753029
Simple	$(8,16,2)^2$	1588162	243963	9714057	394665	3492847	1588162	243963	9714057	394665	3492847
	$(16,32,4)^2$	1952807	305588	9287787	529772	3357740	1952807	305588	9287787	529772	3357740
	$(32,8,8)^2$	2085191	386515	9074476	658930	3228582	2085191	386515	9074476	658930	3228582
WETHER	$(8,32,4)^2$	2788771	473924	12316609	313372	2232782	2788771	473924	12316609	313372	2232782
	$(16,8,1)^2$	2317138	231861	13030305	122990	2423164	2317138	231861	13030305	122990	2423164
	$(32,16,2)^2$	2946829	472434	12160041	257337	2288817	2946829	472434	12160041	257337	2288817

提案したデータ構造に対し，以下の定理が成立する．

定理 4. 拡張状態 EM，ES を持つ優先度 p のブロックにリード命令またはライト命令がキャッシュヒットしたときこのデータ構造の整合性は保たれる．

Proof. （定理4）

拡張状態EMにリード命令がキャッシュヒットしたときにデータ構造の整合性が保たれることを証明する．

Core 0のキャッシュで状態Exclusive，Modifiedのブロックにリード命令がキャッシュヒットした場合は，どちらもCore 1のキャッシュを探索することがない．よって優先度 p より，連想度 a が大きければ図5.3(a)の状態となる．Core 0でキャッシュミスした場合はCore 1の探索を行う必要がある．拡張状態EMであるのでCore 1に該当ブロックが存在しないことがわかっているので，優先度 p が連想度 a 以下ならば図5.3(c)の状態となる．

データ構造で該当ブロックが拡張状態EMであるとき，連想度 a が境界値 $bdry$ より大きいキャッシュ構成では状態Modifiedである．連想度 a が境界値 $bdry$ 以下であるキャッシュ構成では状態Exclusiveである．

優先度 p が境界値 $bdry$ より大きいとき，$a > p$ となる連想度 a のキャッシュ構成では状態Modifiedでキャッシュヒットし，$a \leq p$ となる連想度のキャッシュ構成ではキャッシュミスする．よって，$bdry$ に p の値を代入すればキャッシュ更新後もデータ構造の整合性は保たれる．

優先度 p が境界値 $bdry$ 以下であるとき，$a > bdry$ となる連想度 a のキャッシュ構成では状態Modifiedでキャッシュヒットし，$bdry \geq a > p$ となる連想度 a のキャッシュ構成では状態Exclusiveでキャッシュヒットし，$a \leq p$ となる連想度のキャッシュ構成ではキャッシュミスする．よって，キャッシュ更新後もデータ構造の整合性は保たれる．

拡張状態ESにリード命令がキャッシュヒットした場合や，拡張状態ES，EMにライト命令がキャッシュヒットした場合も同様に証明できる．

拡張状態ESにリード命令がキャッシュヒットしたときにデータ構造の整合性が保たれることを証明する．

Core 0のL1キャッシュでリード命令がキャッシュヒットするとき，該当ブロックの状態ExclusiveならばCore 1を探索する必要がない状況である．該当ブロックの状態がSharedな

らばCore 1を探索する必要がある状況である．Core 0に該当ブロックが存在したとき，優先度をpとする．Core 1に該当ブロックが存在したとき，優先度をp' とする．

該当ブロックにポインタがつながっておらず境界値が連想度最大値のとき

該当ブロックはすべての連想度の構成で状態Exclusiveである．

更新後も状態Exclusiveであるのでデータ構造の整合性は保たれる．

該当ブロックにポインタがつながっておらず連想度最大値より小さい境界値jが存在するとき

このとき連想度aが$a > j$となるキャッシュ構成で該当ブロックは状態Shared，$a \leq j$となるキャッシュ構成で該当ブロックは状態Exclusiveと判定される．

キャッシュリプレースメントポリシがLRUなので該当ブロックの優先度は0になる．

ポインタがつながっていないのでCore 1には該当ブロックは存在しない．

$a > j$となる構成で状態Shared，$a \leq j$となる構成で状態Exclusiveとなるため，境界値は継続される．

しかし，$j < p$のとき，$j < a \leq p$となる構成では該当ブロックがすでにキャッシュアウトしているため状態Exclusiveにならなくてはならない．

提案アルゴリズムにより$j < p$ならばjにpが代入されるため，データ構造の整合性は保たれる．

$j < p$**のとき**

連想度aに対し，$a > p$でキャッシュヒットし状態Shared，$a \leq p$でキャッシュミスとなる．

更新後は$a > p$で状態Shared，$a \leq p$で状態Exclusiveとなるので，jにpが代入されればデータ構造の整合性は保たれる．

$j \leq p$**のとき**

連想度 a に対し，$a > j$ でキャッシュヒットし状態 Shared，$j \leq a > p$ でキャッシュヒットし状態 Exclusive，$a \leq p$ でキャッシュミスとなる．

更新後は $a > j$ で状態 Shared，$j \leq a$ で状態 Exclusive となるので，データ構造の整合性は保たれる．

該当ブロックにポインタがつながっており境界値 j が存在し，ポインタ先の境界値 j' のとき

このとき Core 0 のキャッシュで，連想度 a が $a > j$ となるキャッシュ構成で該当ブロックは状態 Shared

$a \leq j$ となるキャッシュ構成で該当ブロックは状態 Exclusive である．

Core 1 のキャッシュで連想度 a が $a > j'$ となるキャッシュ構成で該当ブロックは状態 Shared，$a \leq j'$ となるキャッシュ構成で該当ブロックは状態 Exclusive である．

$j < p$ かつ $j > p'$ のとき

Core 0 では連想度 a に対し，$a > p$ でキャッシュヒットし状態 Shared，$a \leq p$ でキャッシュミスとなる．

Core 1 では $a > j'$ で状態 Shared，$a \leq j'$ で状態 Exclusive となる．

更新後は Core 0 は $a > p$ または $a > p'$ で状態 Shared，$a \leq p$ かつ $a \leq p'$ で状態 Exclusive となるので，j に p' の値が代入されれば整合性は保たれる．

Core 1 では該当ブロックは状態 Shared となるので，j' に 0 が代入されればデータ構造の整合性は保たれる．

$j < p$ かつ $j \leq p'$ のとき

更新後は Core 0 は $a > p$ で状態 Shared，$a \leq p$ で状態 Exclusive となるので，j に p の値が代入されればデータ構造の整合性は保たれる．

Core 1 では該当ブロックは状態 Shared となるので，j' に 0 が代入されればデータ構造の整合性は保たれる．

$j \geq p$ かつ $j > p'$ のとき

更新後はCore 0は$a > p'$で状態Shared，$a \leq p'$で状態Exclusiveとなるので，jにp'の値が代入されればデータ構造の整合性は保たれる．

Core 1では該当ブロックは状態Sharedとなるので，j'に0が代入されればデータ構造の整合性は保たれる．

$j \geq p$かつ$j \leq p'$のとき

更新後はCore 0は$a > j$で状態Shared，$a \leq j$で状態Exclusiveとなるので，データ構造の整合性は保たれる．

Core 1では該当ブロックは状態Sharedとなるので，j'に0が代入されれば整合性は保たれる．

拡張状態EMにリード命令がキャッシュヒットした場合や，拡張状態ES，EMにライト命令がキャッシュヒットした場合も同様に証明できる．

□

定理 5. あるセットにリード命令またはライト命令がキャッシュミスしたときこのデータ構造の整合性は保たれる．

Proof. （定理5）

あるセットにライト命令がキャッシュミスしたときにデータ構造の整合性が保たれることを証明する．

Core 0のL1キャッシュでライト命令がキャッシュミスするとき，図5.3(e)の状況となる．該当ブロックはModifiedとしてキャッシュインするため，拡張状態EMかつ境界値0のブロックをデータ構造に追加すればキャッシュ更新後もデータ構造の整合性は保たれる．Core 1のキャッシュ上ではスヌープミスならばなにも行わず，スヌープヒットならば状態をInvalidに変えることでデータ構造の整合性は保たれる．

リード命令がキャッシュミスする場合も同様に証明できる． □

系 1. 定理*4*，定理*5*より各キャッシュ構成でブロックの状態が正確にシミュレーションできるため，正しく図*5.3(a), (b), (c), (d), (e)*の状況と書き戻しの発生した回数を測定できる．

さらに，前述したアルゴリズムにはCRCB手法[50]を適用できる．

表 5.3: 全探索手法と提案手法の実行時間の比較.

	全探索手法 [sec]	提案手法 [sec]
FFT	96.56 (1)	18.65 (0.19)
Simple	348.02 (1)	70.34 (0.20)
WETHER	403.53 (1)	80.06 (0.20)

5.6 提案手法の評価

提案手法をC言語で実装し, 全探索手法と提案手法で実行時間を比較した. 使用した計算機はプロセッサがQuad-Core AMD Opteron(tm) Processor 2360 SE 1.3GHzであり，メインメモリが16GBのPCである．探索対象とするキャッシュ構成はセット数が8から32, ブロックサイズが8Bytesから32Bytes, 連想度が1から16の計45構成である．Windows用GUIキャッシュシミュレータのSMPCache [66] のベンチマークアプリケーションとして2コアFFT，2コアSimple，2コアWETHERのメモリアクセストレースを入力とする．各構成でのエネルギーと遅延速度を計算するための状況数を出力とする．全探索手法と提案手法の実行時間の比較を表5.3に示す．全探索手法と提案手法の出力結果の一部を表5.2に示す．提案手法は従来の手法に比べ約5倍高速化となった．全探索手法と提案手法で出力結果が一致しているため，正しくシミュレーションしていることがわかる．

5.7 本章のまとめ

本章ではマルチコアプロセッサキャッシュのシミュレーションを高速化した．特に2コアプロセッサL1キャッシュについてキャッシュコヒーレンシプロトコルの動作を考察した．2コアプロセッサL1キャッシュのキャッシュ構成シミュレーション高速化手法を提案し，計算機上で実装し，評価した．提案手法を全探索手法と比較した結果，最大80%のシミュレーション時間を削減できた．

第6章

結論

本論文ではトレースベースキャッシュ構成シミュレーション高速化に関して研究した成果をまとめた．組込みシステムのキャッシュ設計ではシミュレーションを行いキャッシュヒット/ミス回数を数え上げ最適な構成のキャッシュを得ることが重要である．キャッシュ構成シミュレーションは単一のキャッシュにおけるシミュレーションではなく，複数のキャッシュにおけるシミュレーションである．複数のキャッシュにおけるシミュレーションでは省略可能なシミュレーションが存在するため，キャッシュ同士の関係性を用いたアルゴリズムにより，シミュレーション高速化アルゴリズムを提案した．シングルコアプロセッサアーキテクチャを対象にFIFO/PLRUベースキャッシュ構成シミュレーションを高速化した．FIFOベースキャッシュのキャッシュ構成シミュレーションにおいて第3章の手法はDinero IVに比べて実行速度が最大208倍となっている．PLRUベースキャッシュのキャッシュ構成シミュレーションにおいて第3章の手法は全探索の手法に比べて実行速度が最大249倍となっている．また，特にFIFOベースキャッシュについてキャッシュ構成シミュレーションをさらに高速化し，第4章の手法はDinero IVに比べて実行速度が最大558倍となっている．第4章の手法は第3章の手法に比べて実行速度が最大44%高速化された．また，マルチコアプロセッサキャッシュを対象にLRUベースキャッシュ構成シミュレーションを高速化した．全探索手法に比べ提案手法は実行速度が最大5倍に高速化された．

第2章「関連研究」では既存のキャッシュシミュレーションの研究について紹介した．最初にキャッシュメモリの説明とキャッシュメモリを構成する際のパラメータ，キャッシュ構成シミュレーションを説明した．次に関連研究としてシングルプロセッサアーキテクチャのキャッシュを対象とするキャッシュシミュレーション高速化手法Janapsatyaらの手法やCRCB手法，CRCB-U手法，Haqueらの手法を紹介した．

第3章「シングルコアプロセッサアーキテクチャにおける柔軟なリプレースメントポリシをベースとするキャッシュの高速キャッシュ構成シミュレーション」ではFIFO/PLRUベースキャッシュを対象にキャッシュ構成シミュレーションを高速化した．最初に第2章で紹介したLRUベースキャッシュを対象とするキャッシュ構成シミュレーション高速化手法CRCB手法がFIFO/PLRUベースキャッシュ構成シミュレーションで動作するか検証し，動作するために満たすべき条件を考察，証明した．次にFIFO/PLRUベースキャッシュの性質からキャッ

シュ構成シミュレーション高速化を考察した．FIFO/PLRUベースキャッシュについてキャッシュセット同士に成り立つ関係を定義し，これを用いてキャッシュ構成シミュレーション高速化のためのデータ構造とこれを扱うアルゴリズムを提案した．最後に提案手法を計算機実験した．全探索手法に比べ提案手法は，FIFOベースキャッシュのキャッシュ構成シミュレーションでは実行速度が最大208倍，PLRUベースキャッシュのキャッシュ構成シミュレーションでは最大249倍高速化された．

第4章「シングルコアプロセッサアーキテクチャにおけるFIFOベースキャッシュの高速キャッシュ構成シミュレーション」ではFIFOベースキャッシュを対象にキャッシュ構成シミュレーションを高速化した．最初にFIFOベースキャッシュの性質からキャッシュ構成シミュレーション高速化手法を考察した．次にFIFOベースキャッシュについてキャッシュセット同士に成り立つ関係を定義し，これを用いてキャッシュ構成シミュレーション高速化のためのデータ構造とこれを扱うアルゴリズムを提案した．最後に提案手法を計算機実験した．第3章の手法に比べ提案手法は，FIFOベースキャッシュのキャッシュ構成シミュレーションで実行速度が最大44%高速化された．

第5章「マルチコアプロセッサアーキテクチャにおける高速キャッシュ構成シミュレーション」ではマルチコアプロセッサアーキテクチャのキャッシュを対象にキャッシュ構成シミュレーションを高速化した．最初にマルチコアプロセッサキャッシュが持つキャッシュコヒーレンシプロトコルを説明した．次にキャッシュコヒーレンシプロトコルの性質からキャッシュ構成シミュレーション高速化手法を考察した．キャッシュコヒーレンシプロトコルに関連したメタ状態を導入した．メタ状態により効率的にキャッシュ構成シミュレーションするデータ構造とこれを扱うアルゴリズムを提案した．最後に提案手法を計算機実験した．既存手法に比べ提案手法は実行速度が最大5倍に高速化された．

本論文では第3章，第4章，第5章の3つの手法を提案した点で貢献がある．特に第3章の手法，第4章の手法は組込みシステムで使われるFIFO/PLRUベースキャッシュのキャッシュ構成シミュレーションを高速化した点で貢献がある．第5章の手法は既存手法では適用できないマルチコアプロセッサアーキテクチャのキャッシュ構成シミュレーションを高速化した点で貢献がある．

今後の課題として本論文で達成されていない問題を挙げる．

- 本論文で達成したマルチコアプロセッサキャッシュ構成シミュレーションよりコア数の多いキャッシュアーキテクチャにおいてキャッシュ構成シミュレーションを高速化
- スクラッチパッドメモリなど，本論文の手法と異なるアーキテクチャについての構成シミュレーション
- ワーストケース解析とトレースベースシミュレーションの精度比較

謝辞

本研究は，筆者が早稲田大学大学院基幹理工学研究科博士後期課程在学中に，同大学基幹理工学部戸川望教授の指導のもとに行ったものである．

本論文を結ぶにあたり博士後期課程進学以前から7年間の歳月に亘り日々の研究生活に多大なるご指導を賜りました戸川望教授に心より感謝いたします．ご指導頂きました研究活動における数々の技術は今後の研究者としての糧になります．同じく常日頃から適切なご指導を頂きました同大学基幹理工学部柳澤政生教授に深く感謝いたします．研究の方針に関する鋭いご意見を頂くことで多くの課題を発見することができました．本論文全般に亘り研究指導ならびに貴重な御助言を頂きました同大学情報生産システム研究科木村晋二教授に深く感謝いたします．多岐に亘るご指摘を頂き，本論文を詳細に修正することができました．加えて本論文の執筆にあたり熱心で的確なご教示を下さいました同大学基幹理工学部山名早人教授に深く感謝いたします．お忙しい中数多くのご指摘を頂き，本論文の質を高めることができました．

本研究の各段階では同大学大附辰夫名誉教授，小林優太氏，渡辺信太氏をはじめとする数多くの方々のご指導ご鞭撻を頂きましたことを深く感謝いたします．

本研究の一部は，日本学術振興会科学研究費補助金 (特別研究員奨励費) により行われた．

最後に，本論文に関する研究活動全般に亘り支援していただいた戸川研究室，大附研究室，柳澤研究室の皆様に深く感謝いたします．

参考文献

[1] JEITA, “一般社団法人 電子情報技術産業協会（jeita）,” http://www.jeita.or.jp/.

[2] W. GmbH, “Home - wsts (world semiconductor trade statistics),” https://www.wsts.org/.

[3] 藤広哲也, よくわかる最新組み込みシステムの基本と仕組み．株式会社 秀和システム, 2012.

[4] W. Wolf, 組込みシステム設計の基礎．日経 BP 社, 2009.

[5] International Technology Roadmap for Semiconductors, “Itrs home,” http://www.itrs.net/.

[6] G. E. Moore, “Cramming more components onto integrated circuits,” *Electric Magazine*, vol. 38, no. 8, April, 1965.

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet's with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[8] JEDEC, “Home — jedec,” http://www.jedec.org/.

[9] 日経テクノロジー online, “「計算はメモリーで実行」，ビッグデータ時代の超高効率な演算手法を議論,” http://techon.nikkeibp.co.jp/article/EVENT/20140329/342981/.

[10] Z. Yeraswork, “ビジネスニュース 業界動向：2013 年半導体売上高ランキング,” http://eetimes.jp/ee/articles/1312/11/news041.html.

[11] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad memory: Design alternative for cache on-chip memory in embedded systems,” in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002, pp. 73–78.

[12] A. Janapsatya, A. lgnjatovic, and S. Parameswaran, "Finding optimal l1 cache configuration for embedded systems," in *Proceedings of the 11th Asia and South Pacific Design Automation Conference*, 2006, pp. 796–801.

[13] S. Andalam, A. Girault, R. Sinha, P. Roop, and J. Reineke, "Precise timing analysis for direct-mapped caches," in *Proceedings of the 50th Design Automation Conference*, 2013.

[14] C. Ballabriga, H. Casse, and P. Sainrat, "An improved approach for set-associative instruction cache partial analysis," in *Proceedings of the ACM Symposium on Applied Computing*, 2008, pp. 360–367.

[15] S. Chattopadhyay and A. Roychoudhury, "Unified cache modeling for wcet analysis and layout optimizations," in *Proceedings of the Real-Time Systems Symposium*, 2009, pp. 47–56.

[16] L. Diaz, H. Posadas, and E. Villar, "Obtaining memory address traces from native co-simulation for data cache modeling in systemc," in *Proceedings of the 25th Conference on Design of Circuits and Integrated Systems*, 2010.

[17] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria, "A design framework to efficiently explore energy-delay tradeoffs," in *Proceedings of the 9th International Symposium on Hardware/Software Codesign*, 2001, pp. 260–265.

[18] L. Gao, K. Kaemer, and R. Leupers, "Multiprocessor performance estimation using hybrid simulation," in *Proceedings of the 45th Design Automation Conference*, 2008, pp. 325–330.

[19] A. Ghosh and T. Givargis, "Analytical design space exploration of caches for embedded systems," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 650–655.

[20] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: a compiler framework for analyzing and tuning memory behavior," *the ACM Transactions on Programming*

Languages and Systems, pp. 703–746, 1999.

[21] D. Grund and J. Reineke, "Toward precise plru cache analysis," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010, pp. 23–35.

[22] N. Guan, X. Yang, M. Lv, and W. Yi, "FIFO cache analysis for WCET estimation: a quantitative approach," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 2013, pp. 296–301.

[23] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim, "High level cache simulation for heterogeneous multiprocessors," in *Proceedings of the 41st Design Automation Conference*, 2004, pp. 287–292.

[24] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *Proceedings of Real-Time Systems Symposium*, 2008, pp. 456–466.

[25] D. Hardy and I. Puaut, "Estimation of cache related migration delays for multi-core processors with shared instruction caches," in *Proceedings of 17th International Conference on Real-Time and Network Systems*, 2009, pp. 45–54.

[26] R. Hassan, A. Harris, and A. Eftymiou, "Synthetic trace-driven simulation of cache memory," in *Proceedings of Advanced Information Networking and Applications Workshops*, 2007, pp. 764–771.

[27] P. Heidelberger and H. S. Stone, "Parallel trace-driven cache simulation by time partitioning," in *Proceedings of the 22nd conference on Winter simulation. IEEE Press*, 1990, pp. 734–737.

[28] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for WCET estimation," in *Proceedings of Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 203–212.

[29] M. Moeng, S. Cho, and R. Melhem, "Scalable multi-cache simulation using gpus," in *Proceedings of Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011, pp. 159–167.

[30] J. Schneider, J. Peddersen, and S. Parameswaran, "Mash fifo: a hardware-based multiple cache simulator for rapid fifo cache analysis," in *Proceedings of the the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[31] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003, pp. 36–47.

[32] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg, "Fast: frequency-aware static timing analysis," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 1, pp. 200–224, 2006.

[33] S. M. M. Shwe, H. Javaid, and S. Parameswaran, "Rexcache: Rapid exploration of unified last-level cache," in *Proceedings of 18th Asia and South Pacific Design Automation Conference*, 2013, pp. 582–587.

[34] J. Staschulat and R. Ernst, "Worst case timing analysis of input dependent data cache behavior," in *Proceedinfs of Real-Time Systems*, 2006, pp. 227–236.

[35] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel, "Hybrid source-level simulation of data caches using abstract cache models," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 376–381.

[36] J. Tao and J. Weidendorfer, "Cache simulation based on runtime instrumentation for openmp applications," in *Proceedings of Simulation Symposium 2004*, 2004, pp. 97–103.

[37] W. H. Wang and J. L. Baer, "Efficient trace-driven simulation method for cache performance analysis," in *Proceedings of ACM SIGMETRICS*, 1990, pp. 27–36.

[38] M. Watson and J. Flanagan, "Simulating l3 caches in real time using hardware accelerated cache simulation (hacs): A case study with specint 2000," in *Proceedings of Computer Architecture and High Performance Computing*, 2002, pp. 108–114.

[39] C. Xiangrong and Z. Xiaolin, "An efficient search algorithm for cache simulation platform," in *Proceedings of Software Engineering and Service Science*, 2013, pp. 1055–1058.

[40] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared l2 instruction caches," in *Proceedings of Real-Time and Embedded Technology and Applications Symposium 2008*, 2008, pp. 80–89.

[41] L. Yan, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Proceedings of Real-Time Systems Symposium*, 2009, pp. 57–67.

[42] Y. Chen, J. Cong, and G. Reinman, "Hc-sim: a fast and exact l1 cache simulator with scratchpad memory co-simulation support," in *Proceedings of the Proceedings of Hardware/Software Codesign and System Synthesis*, 2011, pp. 295–304.

[43] M. S. Haque, A. Janapsatya, and S. Parameswaran, "SuSeSim: a fast simulation strategy to find optimal l1 cache configuration for embedded systems," in *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 295–304.

[44] M. S. Haque, A. Janapsatya, and S. Parameswaran, "DEW: a fast level 1 cache simulation approach for embedded processors with fifo replacement policy," in *Proceedings of the 13th Design Automation and Test in Europe*, 2010, pp. 496–501.

[45] M. S. Haque, A. Janapsatya, and S. Parameswaran, "SCUD: A fast single-pass l1 cache simulation approach for embedded processors with round-robin replacement policy," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 356–361.

[46] M. S. Haque, A. Janapsatya, and S. Parameswaran, "CIPARSim: Cache intersection property assisted rapid single-pass fifo cache simulation technique," in *Proceedings of*

the International Conference on Computer-Aided Design, 2011, pp. 126–133.

[47] M. D. Hill and A. J. Smith, “Evaluating associativity in cpu caches,” *IEEE transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[48] R. A. Sugumar, “Set-associative cache simulation using generalized binomial trees,” *ACM Transactions Computer Systems*, vol. 13, no. 1, pp. 32–56, 1995.

[49] W. Zang and A. Gordon-Ross, “T-SPaCS– a two-level single-pass cache simulation methodology,” *IEEE transactions on Computers*, vol. 62, no. 2, pp. 390–403, 2013.

[50] N. Tojo, N. Togawa, M. Yanagisawa, and T. Ohtsuki, “Exact and fast l1 cache simulation for embedded systems,” in *Proceedings of the 14th Asia and South Pacific Design Automation Conference*, 2009, pp. 817–822.

[51] 小林優太, 戸川望, 柳澤政生, 大附辰夫, “組み込みアプリケーションを対象とした 2 階層ユニファイドキャッシュのシミュレーション手法,” 信学技報 ,VLD2009-47, vol. 109, no. 315, pp. 37–42, 2009.

[52] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” *IBM System J.*, vol. 9, no. 2, pp. 78–117, 1970.

[53] D. Grund and J. Reineke, “Abstract interpretation of fifo replacement,” in *Proceedings of the 16th International Sysmposium, SAS 2009*, 2009, pp. 120–136.

[54] 角南英夫, 半導体メモリ． 株式会社コロナ社, 2008.

[55] C. Berg, “PLRU cache domino effects,” in *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, vol. 4, 2006.

[56] K. Kedzierski, M. Moreto, F. Cazorla, and M. Valero, “Adapting cache partitioning algorithms to pseudo-LRU replacement policies,” in *Proceedings of Parallel and Distributed Processing (IPDPS)*, April, 2010, pp. 1–12.

[57] M. Tawada, M. Yanagisawa, T. Ohtsuki, and N. Togawa, “Exact, fast and flexible l1 cache configuration simulation for embedded systems,” *IPSJ Transactions on System*

LSI Design Methodology, vol. 4, pp. 166–181, 2011.

[58] J. Edler and M. D. Hill, "Dinero iv trace-driven uniprocessor cache simulator," http://www.cs.wisc.edu/~markhill/DineroIV/.

[59] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997, pp. 330–335.

[60] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE transactions on Computers*, vol. 35, no. 2, pp. 59–67, 2002.

[61] M. Tawada, M. Yanagisawa, and N. Togawa, "Speeding-up exact and fast FIFO-based cache configuration simulation," *IEICE Electronics Express*, vol. 8, no. 14, pp. 1161–1167, 2011.

[62] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *IEEE transactions on Computers*, vol. 23, no. 6, pp. 12–24, 1990.

[63] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of International Symposium on Circuits and Systems 84*, 1984, pp. 348–354.

[64] M. Tawada, M. Yanagisawa, and N. Togawa, "A high-speed trace-driven cache configuration simulator for dual-core processor l1 caches," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E96.A, no. 6, pp. 1283–1292, 2013.

[65] 多和田雅師, 柳澤政生, 戸川望, "2 コアプロセッサを対象とする正確で高速なヘテロ l1 キャッシュシミュレーション," 情報処理学会 DA シンポジウム 2012, vol. 2012, no. 5, pp. 85–90, 2012.

[66] M. Vega, J. Sanchez, R. Montafia, and F. Zarallo, "Simulation of cache memory systems on symmetric multiprocessors with educational purposes," in *Proceedings of the*

I International Congress in Quality and in Technical Education Inovation, 2000, pp. 47–59.

研究業績

論文

1. M. Tawada, M. Yanagisawa, and N. Togawa, "A high-speed trace-driven cache configuration simulator for dual-core processor l1 caches," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* E96.A, no. 6, pp. 1283–1292, Jun. 2013.

2. M. Tawada, M. Yanagisawa, and N. Togawa, "Speeding-up exact and fast FIFO-based cache configuration simulation," *IEICE Electronics Express*, vol. 8, no. 14, pp. 1161–1167, Jul. 2011.

3. M. Tawada, M. Yanagisawa, T. Ohtsuki, and N. Togawa, "Exact, fast and flexible l1 cache configuration simulation for embedded systems," *IPSJ Transactions on System LSI Design Methodology*, vol. 4, pp. 166–181, Aug. 2011.

4. S. Matsuno, M. Tawada, M. Yanagisawa, S. Kimura, T. Sugibayashi, and N. Togawa, "Energy Consumption Evaluation for Two-Level Cache with Non-Volatile Memory Targeting Mobile Processors," *IEIE Transactions on Smart Processing and Computing*, vol. 2, no. 4, pp. 226–239, Aug. 2013.

国際会議（査読付き）

1. （口頭・ポスター発表）M. Tawada, M. Yanagisawa, and N. Togawa, "A fast trace-driven heterogeneous l1 cache configuration simulator for dual-core processors," in *Proceedings of the 18th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2013),* R4-10s, pp. 259–260, Sapporo, Japan, Oct. 2013.

2. M. Tawada, M. Yanagisawa, T. Ohtsuki, and N. Togawa, "Exact and fast l1 cache configuration simulation for embedded systems with FIFO/PLRU cache replacement

policies", in *Proceedings of the 2011 International Symposium on VLSI Design, Automation and Test (VLSI-DAT 2011),* pp. 247–250, Hsinchu, Taiwan, Apr. 2011.

3. （ポスター発表）M. Tawada, M. Yanagisawa, T. Ohtsuki, and N. Togawa, "Exact and fast l1 cache configuration simulation for embedded systems for FIFO/PLRU-based caches", in *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011) Student Forum*, CF02, Makuhari, Japan, Jan. 2011.

国内学会（査読付き）

1. 多和田雅師, 柳澤政生, 戸川望, "2 コアプロセッサを対象とする正確で高速なヘテロ L1 キャッシュシミュレーション," 情報処理学会 DA シンポジウム 2012, vol. 2012, no. 5, pp. 85–90, Aug. 2012
2. 多和田雅師, 柳澤政生, 戸川望, "複数のキャッシュ構成を同時に表現するデータ構造とこれを用いた高速で正確な 2 コアキャッシュシミュレーション," 第 25 回 回路とシステムのワークショップ, pp. 414–419, Jul. 2012.
3. 多和田雅師, 柳澤政生, 大附辰夫, 戸川望, "FIFO と PLRU をキャッシュ置換ポリシとする高速なキャッシュ構成シミュレーション手法", 情報処理学会 DA シンポジウム 2010, vol. 2010, no. 7, pp. 63–68, Sep. 2010

国内学会（査読無し）

1. 多和田雅師, 柳澤政生, 戸川望, "2 コアアーキテクチャを対象とするトレースベースキャッシュシミュレーションの精度評価", 情報処理学会研究報告, vol. 2013-SLDM-160, no. 15, Mar. 2013.
2. 多和田雅師, 柳澤政生, 戸川望, "2 コアプロセッサ L1 キャッシュ構成の正確で高速なシミュレーション手法", 情報処理学会研究報告, vol. 2012-SLDM-155, no. 3, Mar. 2012.
3. 多和田雅師, 柳澤政生, 戸川望, "2 コアプロセッサアーキテクチャを対象とする正確なキャッシュ構成シミュレーションの高速化に対する一考察", 電子情報通信学会ソサイエティ大会講演論文集 2011 年 基礎・境界, p. 85, Aug. 2011.

4. 多和田雅師, 柳澤政生, 大附辰夫, 戸川望, “柔軟な置換ポリシをもつ2階層キャッシュの正確で高速なシミュレーション手法”, 信学技報, VLD2010-118, vol. 110, no. 432, pp. 13–18 , May 2010.
5. （口頭，ポスター発表）多和田雅師, 柳澤政生, 大附辰夫, 戸川望, “FIFOをキャッシュ置換ポリシとする正確なキャッシュ構成シミュレーションの高速化”, 信学技報, VLD2010-64, vol. 110, no. 316, pp. 55–60 , Nov. 2010.

受賞

1. 優秀発表学生賞, 情報処理学会 SLDM 研究会, Aug. 2013.
2. デザインガイア・ポスター賞, 情報処理学会，電子情報通信学会, Nov. 2013.

日本学術振興会科学研究費補助金

1. 日本学術振興会特別研究員奨励費, “ 世界最速を実現するメニーコアプロセッサの正確なキャッシュ構成シミュレーション技術, ” 2012–2014 年度, 総額 270 万円 (2012 年度:90 万円, 2013 年度:90 万円, 2014 年度:90 万円).