JAIST Repository
https://dspace.jaist.ac.jp/

Title	スレッドレベル投機実行を支援するキャッシュ機構に関する研究
Author(s)	越前，智史
Citation	
Issue Date	2004-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1793
Rights	
Description	Supervisor:田中 清史，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

スレッドレベル投機実行を支援するキャッシュ機構に関する研究

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

越前 智史

2004 年 3 月

修 士 論 文

スレッドレベル投機実行を支援するキャッシュ機構に関する研究

指導教官　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 助教授

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

210036 越前 智史

提出年月: 2004 年 2 月

Copyright © 2004 by Koshimae Satoshi

概 要

スレッドレベル投機実行は，並列計算機上で逐次プログラムを並列に実行することを可能にする手法である．

本研究では，スレッドレベル投機実行における投機メモリアクセスに対するデータ依存違反検出機構を備えた分散キャッシュを検討し，アクセス履歴に基づくスレッド間の同期機構および冗長キャッシュを用いた効率化手法の提案，評価を行う．

スレッド間の同期機構では最大で約 18%，冗長キャッシュを用いた効率化手法では，約 12%の性能向上が見られた．

目 次

第1章　序論

1.1　研究の背景

近年のプロセッサ製造技術の進歩により，チップ上に集積可能なトランジスタの数は増加している．そのハードウェアの利用を最大限に利用するために，現在主要なアーキテクチャとして，スーパースカラプロセッサが多くのマイクロプロセッサに採用されている．

スーパースカラプロセッサは，コンパイル時の静的なコードスケジューリングと，実行時に動的に命令の並び替えを行うアウトオブオーダー機構により，プログラムから命令レベルの並列性を抽出し，複数の機能ユニットで異なる命令を並列に実行することで処理の高速化を図っている．しかし，プログラム内の命令間には様々な依存関係があり，命令レベルの並列を抽出するには限界がある．

そこで，プログラムからスレッドレベルの並列実行を行うアーキテクチャとして，チップマルチプロセッサ（CMP）の研究がなされている [1]．CMP は，並列化されたプログラムをチップ上の各プロセッサユニット（PU）で実行させることにより，スーパースカラプロセッサにはないスレッドレベルの並列性を引き出すことが可能となる．しかし，潜在的な並列性を持たない逐次実行を行うプログラムの並列化は，プログラムが複雑な制御構造を持っていることが多く，静的にデータの依存関係を解析することが難しいため容易ではない．

並列化が困難な逐次プログラムの並列実行を可能にし，より多くの並列性を抽出しようとする技術としてスレッドレベル投機実行がある．これは，依存関係が完全に解決しきれていないスレッドを投機的に実行し，依存関係を侵す処理が検出された場合は，そのスレッドの処理を取り消すことでプログラムの実行の正しさを保証する手法である．スレッドレベルの投機実行が可能であれば，コンパイル時に解析しきれない依存関係を含むプログラムでも，積極的にスレッドに分割することができ，プログラムが本来持つスレッドレベルの並列性をより多く抽出することが可能となる．

1.2 研究の目的

スレッドレベル投機実行を実現するためには，投機的なメモリアクセスによる依存関係の違反を検出する必要がある．本論文では，メモリ投機を支援するキャッシュ機構として，メモリ違反検出機構を備えた分散キャッシュを用い，効率の良いメモリシステムを提案し，その評価を行う．従来方式では，投機スレッドが違反を起こした場合，違反を起こしたスレッドとともに，それを親とする全ての子スレッドも違反を起こしたものとして破棄される．また，スレッドが終了する毎に，使用していた全てのキャッシュラインが無効化され，新たな投機スレッドがプロセッシングユニットに割り当てられる際，キャッシュは完全に空の状態から開始される．本論文では，投機メモリアクセスに対して同期をとる手法と新しい投機スレッドの開始時のキャッシュミス率を軽減する手法について提案，評価を行う．

1.3 本論文の構成

本論文は全 7 章からなる．第 2 章ではスレッドレベル投機実行について述べる．

第 3 章ではスレッドレベル投機実行を支援するキャッシュ機構の基本機構について述べる．第 4 章では提案手法であるメモリの投機的アクセスに対して同期を取る手法と，新しい投機スレッドの開始時のキャッシュミス率を軽減する手法について述べる．第 5 章でその有効性をシミュレーションにより評価する．第 6 章でスレッドレベル投機実行を支援するキャッシュ機構に関する関連研究を紹介する．最後に，第 7 章で本論文の結論を述べる．

第2章　スレッドレベル投機実行

2.1　スレッドレベル投機実行の基本モデル

プログラムから並列性を抽出するには，コンパイラ，あるいは並列プログラミングによって，並列に実行可能な処理を静的に切り分ける手法と，スーパースカラプロセッサのように実行時にプロセッサ内部で，並列実行可能な命令を探し並列に実行する動的な手法の2つがある．前者は粒度の大きな並列性（スレッド）を抽出し，後者は命令レベルでの並列性を抽出する [8]．しかし，静的な並列化では静的に依存関係を解析しなければならず，特にメモリアクセスに対する依存解析は困難なことからコンパイラによる自動並列化には限界がある．また，人の手による並列化プログラミングはプログラマへの負担が大きい．一方，実行時の命令レベルの並列性の抽出も，命令発行幅の増加等によりハードウェアの設計が困難であり，また，たとえ命令発行のサイズを拡大しても近傍命令間の依存関係が多く，実際に並列実行可能な命令を抽出するには限界がある．

これらの問題を解決し，静的に厳密な依存解析を必要とせずに並列実行を可能とする手法がスレッドレベル投機実行 [5, 6, 7] である．

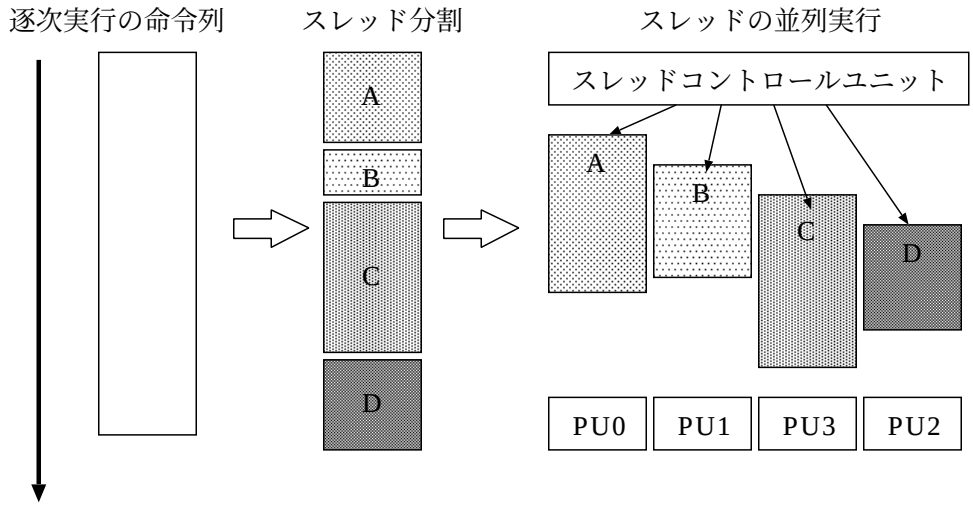


図 2.1: スレッドレベル投機実行

スレッドレベル投機実行とは，図 2.1 に示すようにプログラム実行時の逐次の命令列を連続したスレッドに分割し，本来逐次に実行されるそれぞれのスレッドを複数の実行ユニットに割り振り，各スレッドを投機的に並列実行する手法である．

スレッドレベル投機実行において並列に実行されるスレッドは，本来は先頭のスレッドから逐次に実行されるべきものである．したがってこれらのスレッド間には，元のプログラムで存在した制御依存やデータ依存がそのままスレッド間の依存関係 [8] として存在する．スレッドの間に制御依存やデータ依存が存在する場合，それらを解決しなければ並列に実行することができない．データ依存を無視して実行し続ければ，間違った値を用いて実行を行ってしまうことになる．また，制御依存に対しても本来実行されるべきでないコードを実行してしまう可能性がある．このような投機実行失敗は，誤っていた投機実行を巻き戻し，再実行を行わなければならない．

スレッドレベル投機実行のモデルは，それぞれのスレッドが順次後続する子スレッドを生成して実行が進むので，逐次実行のプログラムと同様に依存関係は常に実行が進む方向のみ存在する．そのためスレッドレベル投機実行におけるスレッド間の依存関係は，常に親スレッドから子スレッドの方向にのみ存在する．したがって，親スレッドがその子スレッドに影響を受けることが無い．またメモリ等への書き込みの実行結果は，逐次と同じ順序で処理される必要がある．スレッドの実行は，各スレッドが並列に実行されるため，メモリへの書き込み順序がスレッドの実行によって変化する．そのため，投機実行を行っているスレッドのメモリへの書き込みは，キャッシュなどローカルな一時領域へ書き込み，親スレッドの実行が完了し投機状態でなくなってから実際にメモリへの書き込み処理が行われる．

スレッドレベル投機実行を行うには，逐次プログラムをスレッドに分割した上で，スレッドの投機的な実行と実行結果の保存，スレッド間の依存関係の処理，投機実行の成功，失敗の判定，失敗時のスレッドの巻き戻し等を行う機構が必要になる．この機構を実現する手法についてはソフトウェア，ハードウェア様々な方式が提案されている [10, 12].

2.2 スレッド間の依存関係

2.2.1 制御依存

スレッドレベル投機実行を行うアーキテクチャでは，本来逐次実行されるプログラムを複数のスレッドに分割し，そのスレッドを投機的に並列実行することで処理の高速化を図る．したがって，スレッド投機を行うためには，何らかの方法を用いてプログラムをスレッドに分割する必要がある．

一般的に，スレッド分割法 [13] は，逐次のプログラムコードに埋め込まれた情報と，実行時の履歴情報などをもとにして，スレッド制御ユニットが次に実行されるべきスレッドの開始点のアドレスを予測し，プロセッシングユニットへそのアドレスを伝えることで実現する．

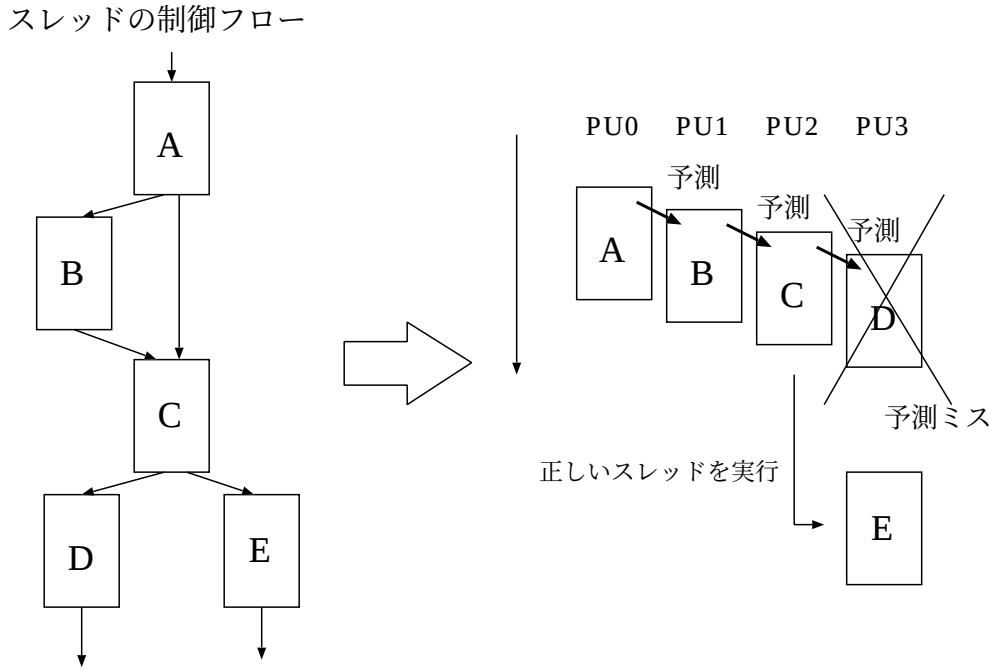


図 2.2: スレッドの制御依存違反

したがって，ある投機スレッドが実行されているとき，そのスレッドに含まれる分岐命令の実行結果が確定するまで，または，そのスレッドの実行が完了するまで次に実行されるスレッドは一意には決定できない．スレッドの制御投機は，図 2.2 のように次に実行されるスレッドを予測し，投機スレッドとして実行するものである．予測が外れていた場合は，間違っていた投機スレッドとともに，後続する投機スレッドも同様に破棄（Squash）して正しいスレッドで実行を再開する．

2.2.2 データ依存

スレッドレベル投機実行を行うアーキテクチャでは，データ依存は命令間に加え，スレッド間にも発生する．そして，投機スレッドは異なるプロセッシングユニットで処理されるため，スレッド間のデータの依存関係 [14, 15] はプロセッシングユニットを跨いだものになる．

スレッド間のデータ依存は，メモリを介したものとレジスタを介したものとに分けられる．

スレッド間のレジスタ依存は，レジスタ通信機構[1]を用いることによって解決可能である．これは，レジスタアクセスはコンパイラによるレジスタ割り当てによって決定するも

[1]親スレッドは子スレッドへ明示的なレジスタ送信を行い，一方子スレッドは利用時に暗黙な待ち合わせを行うことによってレジスタデータの通信を行う機構

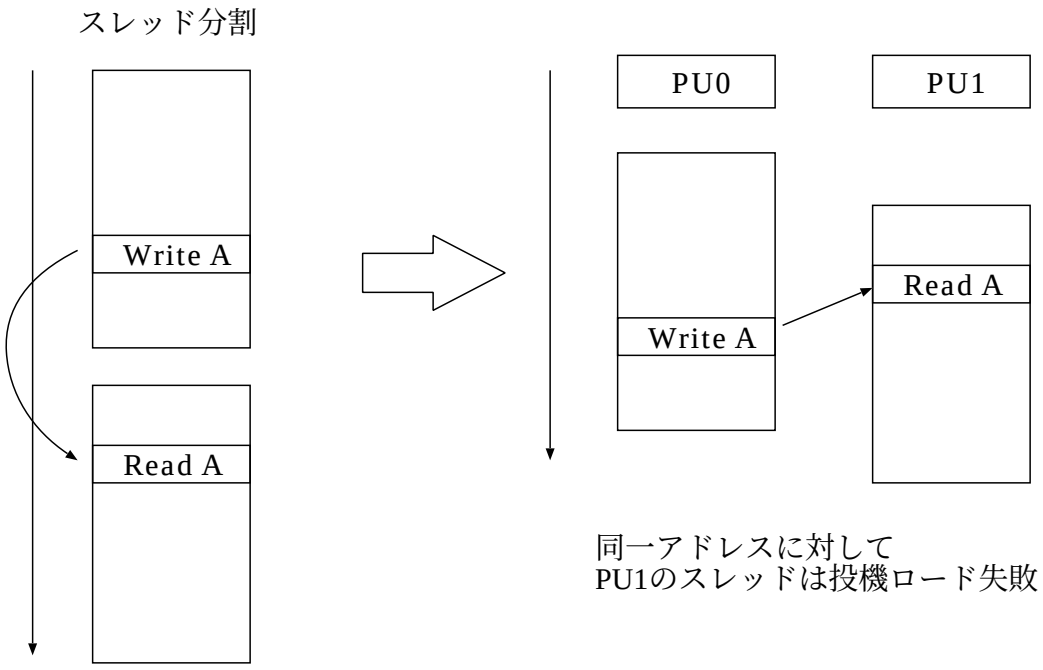


図 2.3: メモリ投機依存違反

のであり，コンパイル時にスレッド間のレジスタ依存は，その依存関係を解析することが可能であり，スレッド間で同期がとれる．

しかし，スレッド間のメモリ依存は，レジスタ通信のようには解決することができない．これは，高度なポインタ解析を用いたとしても，コンパイル時にメモリのアクセス範囲を完全に解析することが難しい．

スレッドレベル投機実行では，図 2.3 のように，メモリ依存に関して，スレッド間の依存関係が存在しないと仮定し，投機スレッドは実行を行う．投機スレッドにおいて，ロード命令より投機的に読み出したデータが，先行スレッドによって上書きされた場合，投機的に読み出した値は逐次実行を行っていた場合は古い値であり，結果として，その古い値に基づいて行った処理は誤りとなる．このようなメモリ投機失敗をメモリ投機依存違反（メモリバイオレーション）と呼ぶ．

プログラムの処理の正しさを保証するためにメモリ依存違反の発生を検出するための機構を用意し，メモリ依存違反を検出した場合には，投機スレッドによって行われた処理を取り消す必要がある．

第3章　スレッドレベル投機実行を支援するキャッシュ機構

スレッド投機アーキテクチャでは，メモリ投機の失敗であるメモリ依存違反を検出し，プログラムの実行の正しさを保証するための機構が必要である．そこで，階層化されたメモリシステムの最上位に位置し，プロセッシングユニットから直接かつ最も頻繁にアクセスされるキャッシュ機構に着目し，そのキャッシュ機構にメモリ投機を可能にするための機構を付加することを考える．

3.1　キャッシュの基本機構

スレッドレベル投機実行を支援するキャッシュ機構は，メモリアクセスの局所性を利用することでメモリアクセスを高速化するという本来の機能に加え，メモリ投機を可能にするために，投機的データの保持と投機的なメモリアクセスによる依存関係の違反の検出を行わなければならない．

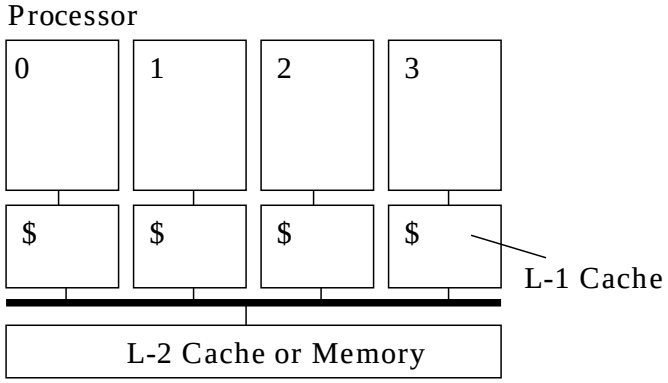


図 3.1: キャッシュの基本構成

3.1.1 キャッシュの構成

各プロセッシングユニットは図3.1のように，プライベートな1次キャッシュを持ち，投機的メモリアクセスはその状態がどのくらい投機的であるか，投機データの親子関係等，取りうる状態すべてをキャッシュに記録する．それらのプライベートなキャッシュを共有バスで接続し，そのバスをスヌープすることにより，他のキャッシュへのメモリアクセスを監視する．

通常の計算機では，メモリの階層化を高めるため2次，3次キャッシュを用意するが2次キャッシュ以降には投機的なデータを含まないものとする．

3.1.2 データの投機度

スレッドを処理するプロセッシングユニットは，それぞれスレッドの投機の程度に応じた投機度を持つ．投機度は実行中のスレッドおよびそれを処理するプロセッシングユニットの親子関係を示すものであり，その表現方法[1]は様々なものが提案されている．ここでは，確定スレッドの投機度を0とし，以後，より投機的なものほど大きくなる整数として表現する方法をとる．

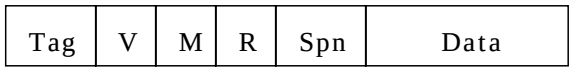
例えば，4つのプロセッシングユニットを持つモデルの場合では，通常，スレッドの処理が開始した時点では，それぞれ親スレッドから0，1，2，3の順番で表現される．親スレッドの処理が終了し，終了確定（コミット）が起こる度に各投機度はデクリメントされる．親のスレッドがコミットした場合，次に一番投機度の高いスレッドがそのプロセッシングユニットで実行されるため，そのプロセッシングユニットの投機度は3で表現され，以前投機度が1であったプロセッシングユニットは投機度0となり，不投機の状態（HEAD）となる．

一般的に，投機度が高いほど，スレッドの予測ミスおよびメモリ投機違反によるスレッドの破棄の可能性が大きくなる．

3.1.3 キャッシュラインの構成

スレッドレベル投機実行を支援するキャッシュのラインは，既存のキャッシュラインの構成に，データを読み出されたことを記録するRビットと，データの投機度を表すSpnが追加される．キャッシュラインに各プロセッシングユニットの読み書きの情報を残すことにより，データの順序関係を保持する．スレッドレベル投機実行を支援するキャッシュラインは図3.2のように構成される．

[1]SVC:Speculative Versioning Cache[2] ではキャッシュに同じデータを保持する親の pointer を記録し，VOL，VCL を用いて依存関係を決定する．



V : Valid
M : Modified
R : Read
Spn : Speculative Number

図 3.2: キャッシュラインの基本構成

- Valid(V)

  そのキャッシュラインが有効であることを示す.

- Read(R)

  そのキャッシュラインに対してロードが行われた場合にセットされる．既に Modified ビットがセットされているラインに対してはメモリ依存違反の原因とはならないためセットしない．[2]

- Speculative Numver(Spn)

  格納するデータの投機度を示す．データを他のキャッシュ投機ロードした場合，そのプロセッサの投機度をそのラインに記録しておく．下位のメモリから投機ロードした場合は，投機度 0，つまり一番投機度の低いプロセッサである HEAD からデータをロードしてきたのと同じ状態とする.

  一方，投機ストアの場合，データが書き込んだプロセッシングユニットの投機度をそのラインに記録する．ただし，Read ビットがセットされている場合，メモリ依存違反の検出が行えるように，たとえそのラインに対して書き込みを行っても Spn の値を変化させない.

- Modified(M)

  そのラインに対して書き込みを行った場合にセットされる.

メモリ依存違反の判定には，ストアの伝搬の際に Spn と Read が併せて用いられる．またストアの伝搬の際，該当ラインのアップデートを行うためキャッシュが付属するプロセッシングユニットの投機度とキャッシュ中のデータの投機度は必ずしも一致しない.

また，Spn は，セットされたときの投機度を保持し続けるものではなく，HEAD スレッドのコミットが起こり，各プロセッシングユニットで実行しているスレッドの投機度が減少するに従ってキャッシュ内の有効なラインに対してすべての Spn を減少させる必要がある.

[2]ただし，ラインサイズによって条件が異なる（後述）

3.2 キャッシュの基本動作

スレッドレベル投機実行を支援するキャッシュ機構では，メモリ依存違反の検出が必要になる．その検出機構を通常の並列計算機のキャッシュの一貫性プロトコルを拡張する事により実現する．キャッシュの一貫性プロトコルには，ライトインバリデート方式，ライトアップデート方式の 2 種類の方式があり，本研究では，ライトアップデート方式を拡張することにより，メモリ依存違反検出を行う．

3.2.1 ストアの伝搬

プロセッシングユニットからキャッシュに対する書き込みは，そのアドレスとデータ，投機度を共有バスに流すことで，他のキャッシュからスヌープ可能な状態にされる（図 3.3）．

キャッシュコントローラは，他のキャッシュからの書き込みをスヌープした場合に，スヌープしたデータを自分が格納しているかを調べ，格納している場合は，スヌープした投機度と自分自身の投機度を比較することで，メモリ依存違反の発生を検出する．一般的に，メモリ依存違反は，親スレッドによる書き込みより先に子スレッドが同じアドレスに対して読み込み，場合によっては，書き込みがあった場合に，発生するので親から子へのみ伝搬，アップデートを行う．

メモリ依存違反の発生を検出した場合は，スレッド制御ユニットに対してその発生を通知するとともに，自身も依存違反発生時の処理を行う．一方，依存違反を引き起こさなかった場合は，スヌープした情報を用いてデータの更新を行う．

このように共有バスを用い，ライトアップデート方式を拡張したストアの伝搬と投機度に基づいたデータの更新を行うことにより，親から子へのキャッシュラインの投機的アップデート，メモリ依存違反検出の実現を可能にする．

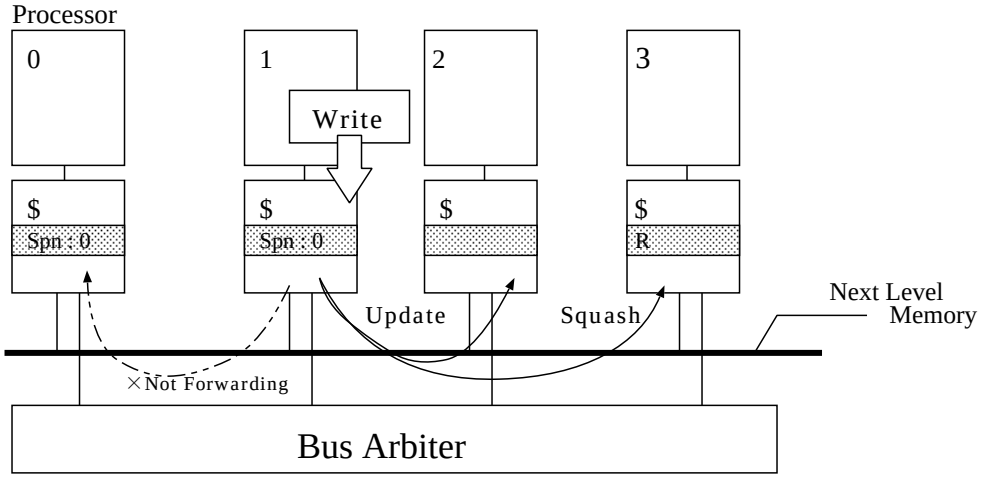


図 3.3: ストアの伝搬

3.2.2 キャッシュの充填

単一のプロセッサでは，キャッシュミスが発生した場合は，ひとつ下のメモリ階層からデータをロードしてくる．例えば，1 次キャッシュでミスした場合，2 次キャッシュからデータを充填する．しかし，キャッシュ間で投機的にデータのアップデートを行う場合は，より投機でない，すなわち，自分より投機度の低いキャッシュに，下位レベルのメモリよりも新しいデータを保持している可能性がある．従って，スレッドの親子関係を考慮し，逐次実行時のメモリのアクセス順序を守るためには，自分より投機度の低いキャッシュからデータを転送する必要がある．

実際のキャッシュの充填は以下の手順で行われる（図 3.4）．キャッシュミスを起こしたキャッシュは，共有バスに対して充填要求（1）を出し，アドレスと投機度を流す．キャッシュ要求をスヌープしたキャッシュは，自分自身の投機度と要求を出したキャッシュの投機度を比較する．自分自身の投機度が要求を出したものよりも低い場合は，格納するデータが有効であるかどうかを調べ，有効であった場合バスアービタに対して転送要求（2）を送る．バスアービタは，キャッシュからの転送要求を受信した場合に，最も投機度の高いモノに対して転送許可（3）を出す．キャッシュから転送要求がでなかった場合は，下位メモリに対して転送要求を出す．転送許可を受信したキャッシュは共有バスに自分が格納するデータをその状態を表す情報とともに流す（4）．充填要求を出したキャッシュは，共有バスに流れるデータで充填を行う．

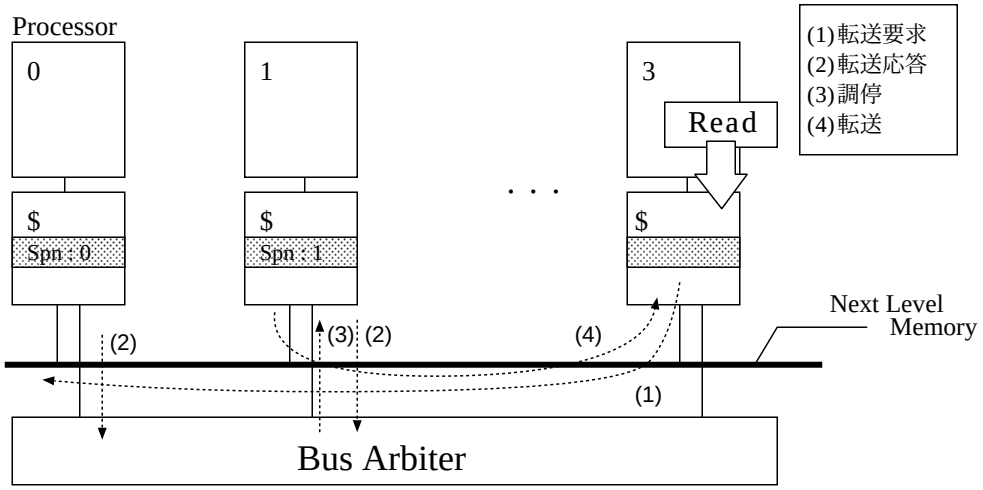


図 3.4: キャッシュの充填

3.2.3 投機データのライトバック

キャッシュの内容がストアによって変化した場合，階層間でメモリの一貫性を保つために，その変化を下位下層のメモリに反映しなければならない．投機実行を支援するキャッ

シュ機構では，ある特定の条件を満たした時にまとめて書き戻しを行うライトバック方式をとる．ただし，通常のライトバック方式とは異なる方法をとらなければならない．通常のキャッシュでは，ライン競合（コンフリクト）発生によりキャッシュラインを置換する際，ラインの内容が更新されていればライトバックを行う．スレッドレベル投機実行を支援するキャッシュ機構では，投機ストアの内容はプライベートなキャッシュに破棄可能な状態で保持し，下位メモリには，その投機データを含めないという条件があるため，ラインの置換を行ってはならない．

そのため，投機スレッドを処理するプロセッシングユニットでは，投機スレッドがHeadになるまで，キャッシュは要求されたデータを返すことができず，その間の実行がストールすることがありうる．

3.3　メモリ依存違反

3.3.1　メモリ依存違反の検出

スレッドレベル投機実行を支援するキャッシュにおいて，投機ロードされたブロックに対して，投機度の低いプロセッシングユニットからのストアの伝搬によるアップデートが行われる際，同時にメモリ依存違反の検出を行う．メモリ依存違反は，あるメモリアドレスに対して，投機スレッドが値を読み出した後に，同じアドレスに対して親スレッドが値を書き換えた場合に発生する．

しかし，通常，キャッシュはその効率を得るため空間的局所性を利用し，1 つのキャッシュラインに対して複数のデータを収める方法をとる．本研究では，投機データの状態管理をライン毎に行うため，ラインに含まれるデータ数によって依存違反の取り扱いが異なる場合がある．

そこで，キャッシュライン中に含まれるデータ数が単一である場合と，複数である場合に分けて考える必要がある．

- 単一データの場合

  この場合，ストアによってラインののデータがすべて上書きされる．

  このため，投機ストアが行われたラインにはそのスレッドによって生成されたデータしか含まれておらず，以後そのラインに対してロードを行ったとしても，それは投機ロードにはならない．

  この場合，スレッド内で依存関係が完結しているので，そのスレッドが投機的であったとしても，投機ロードとする必要がないためである．したがって，投機ストアが行われたラインに対しては，ロードはその状態を変化させず，ストアの伝搬によるアップデートを行ってはならない．

  単一データの場合は，投機ロードのラインに対して，ストアの伝搬があればメモリ投機依存違反とし，投機ストアのラインに対しては，ストアの伝搬によるアップデートを行わない．

- 複数データを含む場合

  ラインに複数のデータが存在し，一回のストアによってライン内のデータが全て上書きされるということはない．

  このため，投機ストアが行われたラインには，更新されないデータが残り，以後そのラインに対してロードを行った場合は，その古いデータを読み出す可能性がある．

  これは，疑似共有（フォールスシェアリング，図 3.5) と同様の問題であり，ライン単位で状態を保存する限り，ロードが投機的であるかどうかを区別することができない．

また，投機ストアが行われたラインに対するストアの伝搬によるアップデートはライン全体をアップデートしてしまうため，投機ストアによって更新されたデータを古い値で上書きしてしまう可能性がある．

複数のデータを含む場合は，投機ロードと投機ストアのどちらの状態であっても，そのラインに対するストアの伝搬があれば依存違反としなければならない．

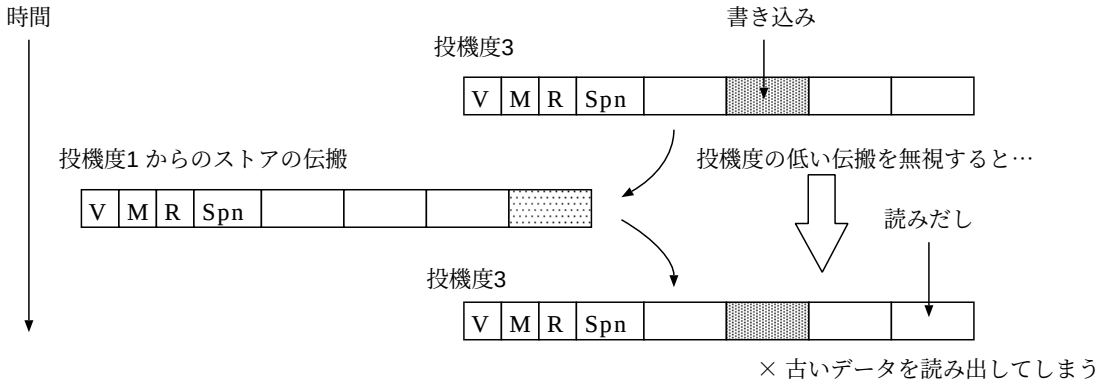


図 3.5: フォールスシェアリング問題

3.3.2 メモリ依存違反による無効化

メモリ依存違反が発生した場合，キャッシュ内にあるデータのうち，誤っている可能性のあるラインに対して無効化しなければならない．

誤っている可能性のあるデータは以下の条件のものである．

- 違反を起こしたスレッドの投機度以上のライン
- 違反を起こしたスレッドが使用するキャッシュ中で Modified ビットがセットされたライン

したがって，メモリ依存違反が発生した場合，上記の条件を満たすラインに対して無効化を行う．以上の条件にはずれたラインに対しても，Read ビットがセットされているラインに関しては，スレッドの再実行に備えて，Read ビットをクリアにしておかなければならない．

3.3.3 メモリ依存違反の連鎖

投機スレッドがメモリ依存違反を起こした場合，依存違反を起こしたスレッドとともに，それを親とする全ての子スレッドも依存違反を起こしたものとして破棄されなければならない．これは，子スレッドは親スレッドの生成した誤ったデータを用いて，誤った処理を行っている可能性があるためである．

本来，スレッドの予測ミスによる制御依存違反とは異なり，メモリ依存違反は依存違反を引き起こした親スレッドが生成したデータを利用していない限り，子スレッドは破棄する必要がない．しかし，この条件を満たす場合は少なく，依存違反の判定の条件から除外するには，それを判定する何らかの機構が新たに必要となる．

したがって，本研究では，投機スレッドが依存違反を引き起こした場合，それにつながる子スレッドを破棄するものとする．

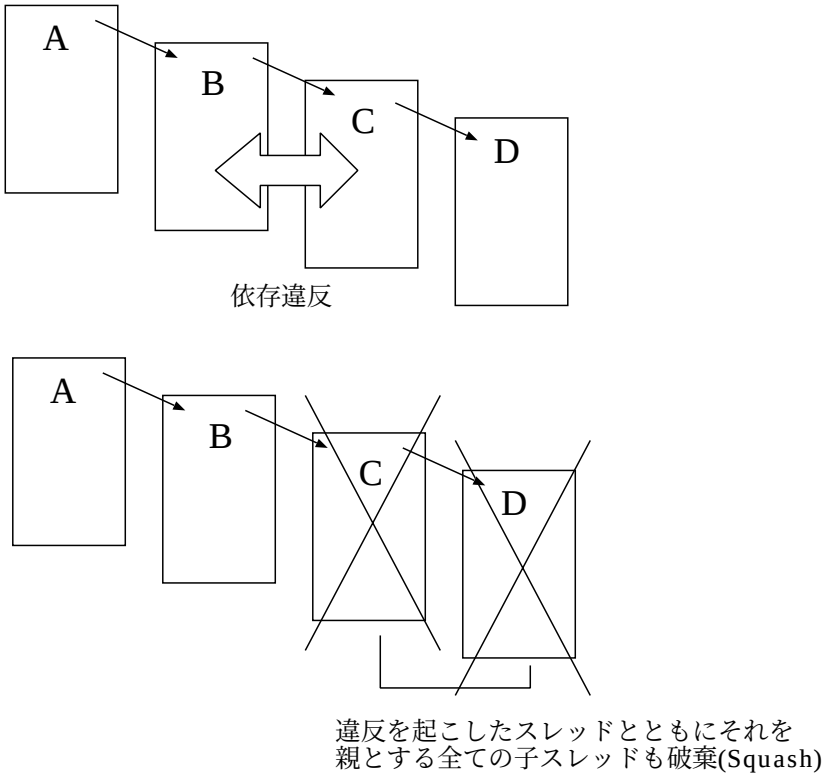


図 3.6: メモリ依存違反の連鎖

第4章　高機能キャッシュの設計

前章で述べた基本的なキャッシュ機構をもとに効率の良いメモリアクセスを実現する．その手法として，データ依存に対して同期をとりメモリ依存違反回数を削減する手法と冗長キャッシュを用いたキャッシュミス率軽減方法について述べる．

4.1　データ依存同期型キャッシュ

投機スレッドがメモリ依存違反を起こした場合，依存違反を起こしたスレッドとともに，それを親とする全ての子スレッドも依存違反を起こしたものとして破棄されなければならない．一度依存違反が発生した場合，該当する投機スレッドは，そのスレッドの最初から実行しなおさなければならず，かつ，そのスレッドが使用していたキャッシュは前章で述べたブロックに対して無効化しなければならない．このように依存違反を多く引き起こすことは，パフォーマンスのボトルネックとなる可能性がある．

スレッド間で投機メモリアクセスに対して同期操作をとりメモリ依存違反を削減する手法を提案する．図 4.1 で同期操作によりメモリ依存違反の削減が可能となる例を示す．子スレッドと孫スレッド間でメモリ投機違反が発生し，孫スレッドは依存違反となり，そのスレッドの実行が始めから再開される（図 4.1 上段）．その後，親スレッドと子スレッドの関係でメモリ依存違反が起きたとき，子スレッド，孫スレッドともに破棄（Squash）され，実行が始めから再開される（図 4.1 中段）．子スレッド，孫スレッドはともに制御の流れが変化しない限り，同じ実行が再び行われる．この場合，子スレッド，孫スレッド間で再び孫スレッドがメモリ依存違反を引き起こし破棄される可能性がある．そこで，キャッシュ内に特別な制御フィールドを用意し，過去のメモリ依存違反を引き起こしたデータの情報をキャッシュラインに保持し，依存違反の原因となるメモリアクセスを回避する方法をとる．

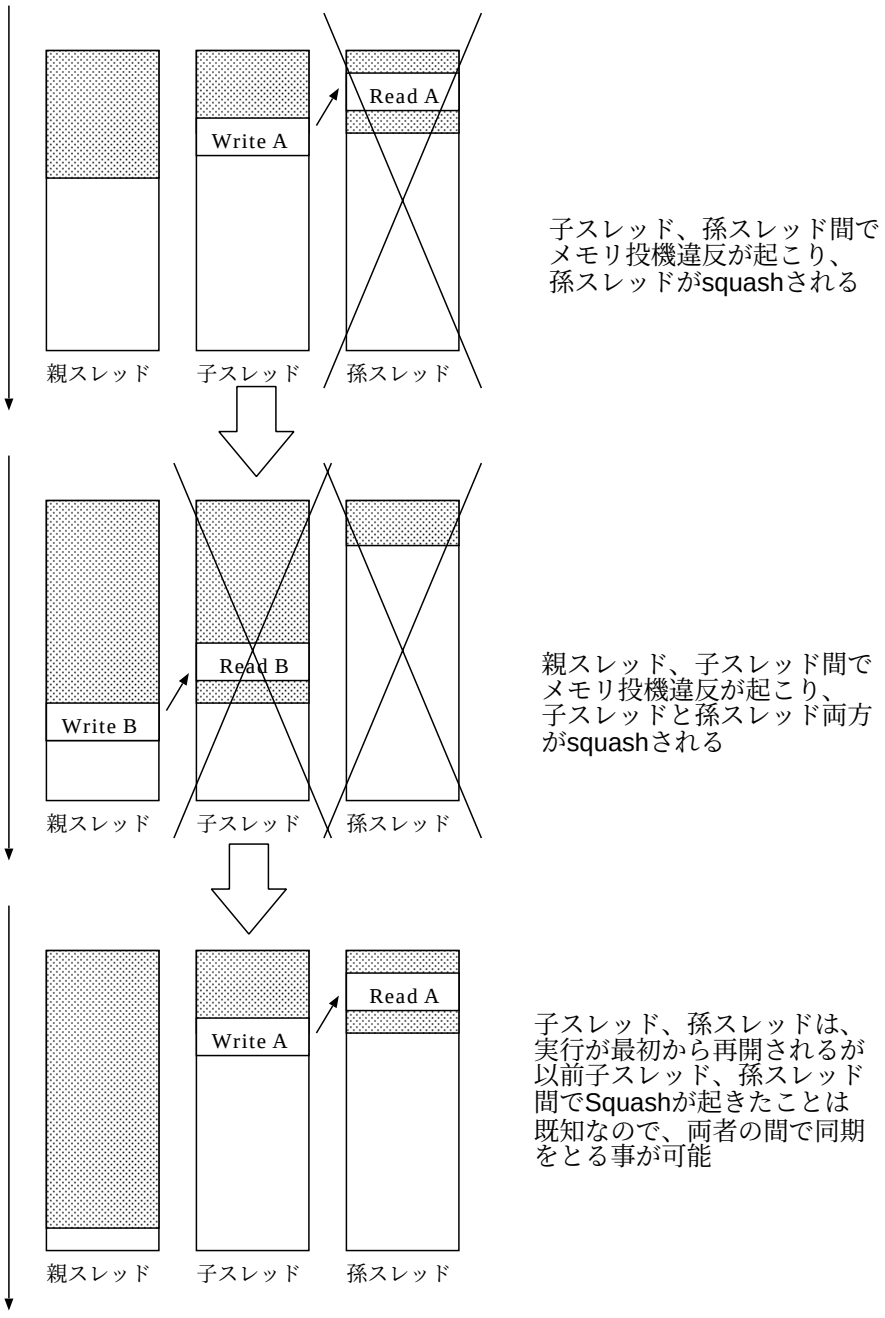


図 4.1: 非効率なスレッド実行遷移

キャッシュラインの構成

Tag	V	M	R	Spn	Sq	Prev	Curr	Data

Spn : Speculative Number
Sq : Squash
Curr : Current Store
Prev : Previous Store

図 4.2: キャッシュラインの構成

データ依存の同期操作を，メモリ依存違反検出を行う分散 1 次キャッシュを用いて行う．実際の同期操作は，キャッシュラインに Sq，Prev，Curr を設けることにより実現する．メモリ依存違反が発生した後，スレッドの親子間のストア伝搬履歴を使用して更なる子スレッドの破棄を抑制する方法をとる．

依存違反が発生する前，各キャッシュは，Curr（Current Store）によって，キャッシュが属するプロセッシングユニットのストア回数を記録する．依存違反が発生した後，依存違反を引き起こしたプロセッシングユニットおよびそれを親とするプロセッシングユニットは，Curr の値を Prev（Previous Store）にセットする．また，同時に依存違反を引き起こしたプロセッシングユニット以上の投機度のデータを持つすべてのキャッシュラインに対して Sq ビットをセットする．

メモリ依存違反の発生後の実行は，親スレッドは当該ラインへのストアの回数（Curr）が依存違反前の実行時のストア回数（Prev）より小さい場合，そのラインに対して同期不成立となりストアの伝搬を行わない．一方，子スレッドにおいて Sq ビットがセットされたラインにロードアクセスした場合，実行がストールする．親スレッドからのストアの伝搬があったとき Sq ビットはクリアされ，ストールが解除されて投機ロードが行われ，そのスレッドの実行が再開される．

この操作により，依存違反が発生した後の投機メモリアクセスに対して同期操作がとれる．以上まとめると，キャッシュラインの状態ビットは，以下のような条件で変化する．

- Squash（Sq）

  メモリ依存違反が発生し，Squash が起きた場合セットする．Sq ビットをセットされるのは，依存違反を引き起こしたスレッドの投機度以上のキャッシュラインに対してである．

  スレッド中の制御フローの変化により，同期失敗の可能性がある．これは，親スレッドのストア回数の変化によりストアの伝搬が行われない可能性があるためである．これにより，子スレッドがストールし続ける可能性がある．その場合，そのスレッドが不投機の状態（HEAD）に変化した場合，そのキャッシュ中で Sq ビットの立っているライン の Sq ビットは，すべてクリアにされる．

- Previous Store（Prev）

  メモリ依存違反の発生以前のすべてのストアのアクセスの回数を保持する．スレッド再開時に Curr のコピー (Squash 発生時までに実行していたストアの回数) がセットされる．ただし，Curr の値が Prev よりも小さい場合，前回の Prev の値を保持する．これは，同じスレッド実行中に複数回，メモリ依存違反の発生した場合，前回のメモリ依存違反発生時よりもスレッドの実行が進んでいない場合があるためである．

- Current Store（Curr）

  スレッドの実行開始時からメモリ依存違反が発生し，Squash が起きるまで，そのキャッシュラインに対して，すべてのストアのアクセスの回数を保持する．スレッド再開時にクリアされる．

4.2 冗長キャッシュを用いた効率化手法

分散キャッシュ型のアーキテクチャの場合，各プロセッシングユニットに属した固有のキャッシュを持ち，各プロセッシングユニットはそれぞれが持つキャッシュのみにアクセスする方式をとる．しかし，スレッドレベル投機実行を支援するキャッシュ機構では，HEADの実行終了時に様々な作業があり，固有のキャッシュを利用するだけでは効率の良い実行が行えない．

本研究では，各プロセッシングユニットが持つ固有のキャッシュ以外に，冗長なキャッシュを用意する．各プロセッシングユニットが状況に応じて各キャッシュと冗長なキャッシュを切り替えることで，効率の良い実行を行う．

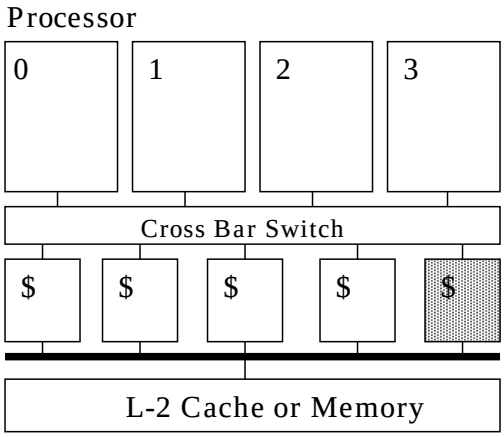


図 4.3: 冗長分散キャッシュの構成

4.2.1 スレッド終了時の処理

スレッドレベル投機実行は，逐次プログラムから積極的にスレッドに分割し，並列に実行する手法である．逐次プログラムを実行する場合，各スレッドが生成したデータの順序は保証されなければならない．そのため，プロセッシングユニットでHEADのスレッドの実行が終了し，次のスレッドが実行されるとき，正しいデータの生産と消費のルールを守るための以下のような注意が必要である．実際のスレッドの状態遷移を図4.4に示す．

キャッシュラインの無効化処理

スレッドレベル投機実行を支援するキャッシュ機構では，確定スレッドがコミットする毎に，全てのキャッシュラインが無効化（インバリデート）される．これは，ストアの伝搬によって投機度の高いスレッドへのアップデートを行うため，確定スレッドが使用し

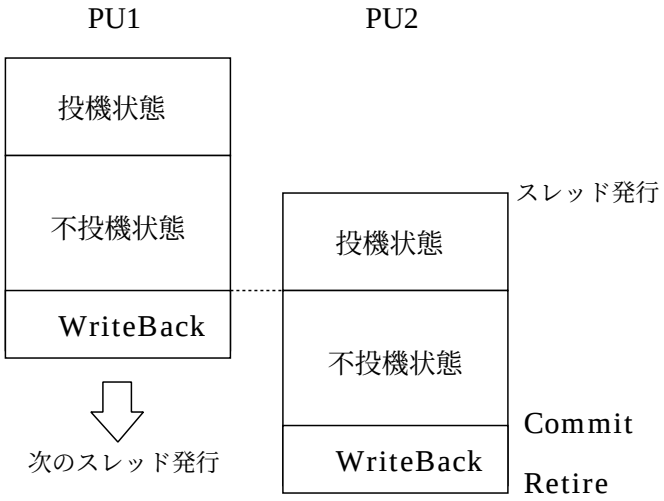


図 4.4: スレッドの遷移.

ていたキャッシュには古いデータが含まれている可能性があるためである．そのため，新たなスレッドが投機スレッドとしてプロセッシングユニットに割り当てられる際，キャッシュは完全に空の状態になっており，メモリアクセスを行う場合は，その都度キャッシュを充填しなおさなければならない．したがって，コミット時に全てのブロックを無効化していたのでは，スレッド間で時間的，空間的局所性が共有できず，十分なパフォーマンスを得られない可能性がある．

ライトバック処理

ライトバックは，確定スレッドの処理が終了した後に，スレッドのリタイア処理の一つとして行われる．

ライトバックを行う必要のあるキャッシュのブロックは，Modified がセットされているラインである．Modified がセットされているラインはメモリとキャッシュ間で一貫性が取れておらず，現在実行している投機スレッドがキャッシュミスによりメモリからデータを取ってくる際，古いデータを使用してしまう可能性がある．そのため，ライトバックが終了するまではスレッドはリタイアできず，そのプロセッシングユニットには新たなスレッドを割り付ける事ができない．

スレッド終了時のライトバックは可能な限り速く処理されることが望ましいが，キャッシュと下位レベルのメモリを接続するバスの処理能力には，限界がある．そのため，ライトバックを行う必要のあるラインが多数存在する場合は，次スレッドの実行が即座に行えず，パフォーマンスのボトルネックとなる可能性がある．

4.2.2 冗長キャッシュ

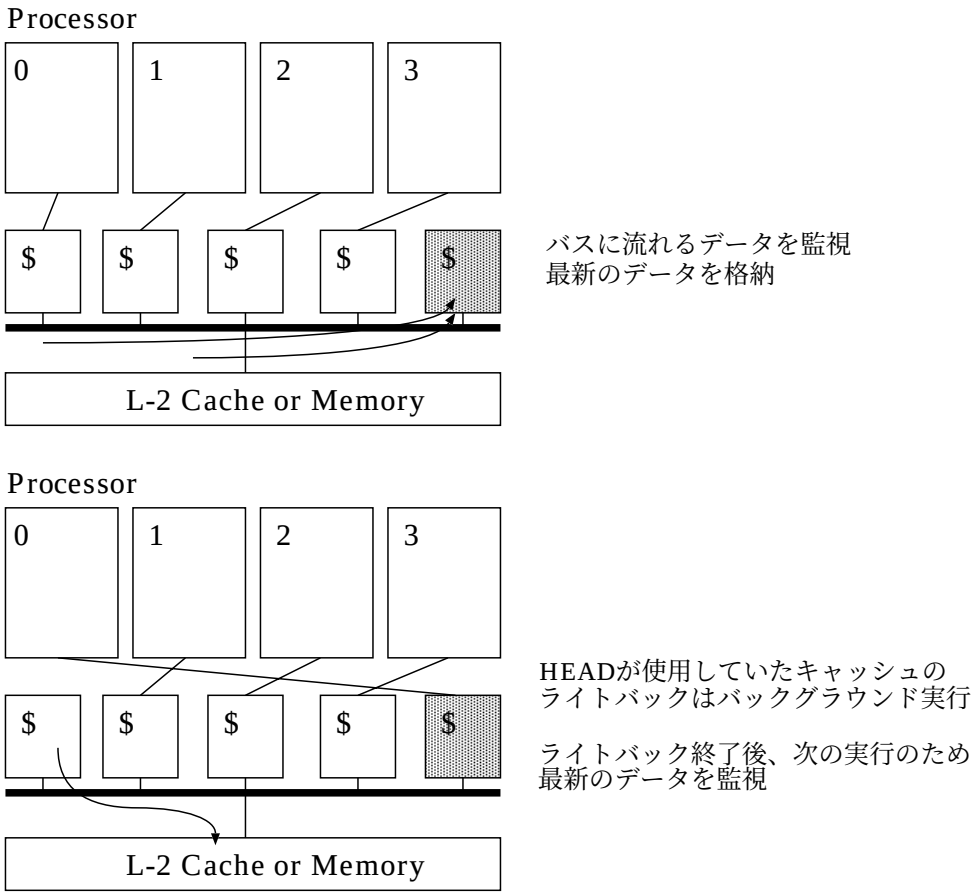


図 4.5: 冗長分散キャッシュの動作

前節で述べたスレッド実行の終了時の処理である無効化およびライトバック処理のオーバヘッドを削減するために，図 4.3 のような冗長な分散キャッシュを構成する．スレッド実行に割り当てられていないキャッシュは待機キャッシュとして振る舞い，ストアの伝搬による最新の情報をスヌープし，次に割り当てらるスレッドがすぐに使用できる状態にする．

HEAD が終了した際，次の新しいスレッドが直ちにそのキャッシュを利用してその実行を開始する（図 4.5）．キャッシュの中にはストアの伝搬による最新のデータが存在し，新たに無効化およびライトバックの操作を行う必要がない．

一方，終了した HEAD スレッドが使用したキャッシュは，通常行わなければならないスレッド実行終了コミット時の操作である無効化，ライトバックの操作をバックグラウンドで行う．

第5章　評価

5.1　評価環境

5.1.1　ベンチマークプログラム

シミュレーションの対象プログラムとして，Livermore Kernel[17] を用いる．これらのプログラムは，専用コンパイラによってコンパイル時にスレッド分割処理が行われ，実行バイナリ中の適切な位置にスレッドスタート命令が挿入されるアーキテクチャを仮定する．ベンチマークプログラムとそれぞれのプログラムをシングルプロセッサで逐次実行した場合の動的な命令数を表 5.2 に示す.

5.1.2　シミュレーションモデル

基本仕様

スレッドレベル投機実行を行うアーキテクチャとして，4 台のプロセッシングユニットを備えるチップマルチプロセッサを仮定する．コンパイラによって分割された各スレッドは，スレッドコントロールユニットによって各プロセッシングユニットにそのプログラムカウンタが渡され，実行される．スレッドの実行終了の際，スレッドコントロールユニットに終了通知をし，新たなスレッドが割り当てられる．レジスタ通信機構は考慮しない.

各プロセッシングユニットの実行ユニットで処理される命令の実行に必要なレイテンシは表 5.1 とする.

表 5.1: 命令実行レイテンシ

Class	Latency	Example
branch	3 cycles	beq,bne,bgnz
mul&div	4 cycles	mult,mul,dvi
others	1 cycle	add,sub,and,etc.

表 5.2: 各ベンチマークプログラムとその動的命令数

ベンチマークプログラム	動的命令数
Kernel 01 – hydro fragment	8,192
Kernel 02 – RCCG excerpt (Incomplete Cholesky Conjugate Gradient)	7,909
Kernel 03 – inner product	11,089
Kernel 04 – banded linear equations	15,811
Kernel 05 – tri-diagonal elimination, below diagonal	12,510
Kernel 06 – general linear recurrence equations	58,348
Kernel 07 – equation of state fragment	51,969
Kernel 08 – ADI integration	113,840
Kernel 09 – integrate predictors	56,170
Kernel 10 – difference predictors	124,389
Kernel 11 – first sum	7,310
Kernel 12 – first difference	8,610
Kernel 13 – 2-D PIC (Particle In Cell)	160,951
Kernel 14 – 1-D PIC (Particle In Cell)	234,709
Kernel 15 – Casual Fortran. Development version	242,312
Kernel 16 – Monte Carlo search loop	12,854
Kernel 17 – implicit, conditional computation	47,808
Kernel 18 – 2-D explicit hydrodynamics fragment	158,175
Kernel 19 – general linear recurrence equations	32,229
Kernel 20 – Discrete ordinates transport, conditional recurrence on xx	30,043
Kernel 21 – matrix*matrix product	423,724
Kernel 22 – Planckian distribution	10,201
Kernel 23 – 2-D implicit hydrodynamics fragment	453,053
Kernel 24 – find location of first minimum in array	10,917

各プロセッシングユニットが持つキャッシュは，容量 8KB，ダイレクトマップ方式とする．メモリアクセスは，キャッシュヒット時は 1 サイクルとし，ミス時のペナルティは 20 サイクルとした．2 次キャッシュを考慮し，2 次キャッシュは 100% ヒットを仮定する．また，1 次キャッシュ間のアクセスレイテンシ (ストアの伝搬，親スレッドからのロードの充填) をそれぞれ 6 サイクルとした．

スレッドレベル投機実行を支援するキャッシュ機構の評価を行うため，ラインサイズを変化させて行った．また，従来方式と今回提案したデータ依存同期型キャッシュ，冗長キャッシュを用いたキャッシュ，両方，それぞれ対応する機構をシミュレータに実装し，それらを切り替えてシミュレーションを行い評価を行う．

シミュレーションの理想化

シミュレーションを行う際，いくつかの機構を理想化し，そのパラメータを仮定した．スレッドレベル投機実行を支援するキャッシュ機構を評価する際に重要ではない部分，シミュレータの仕様上シミュレーションを行うのが困難な部分について理想化を行った．

- スレッド予測

  本研究はスレッドのメモリ投機のデータフローに着目するものであり，コントロールフローの変化によるデータフローの変化の影響を極力排除する．シミュレータでは，シングルプロセッサで実行した場合の実行トレースを再構成することにより実現した．

- 2 次キャッシュ

  2 次キャッシュは，100%ヒットを想定した．これは，シミュレーションにおいてベンチマークプログラムの扱うメモリ空間は，既存のアーキテクチャにおいても全て 2 次キャッシュに載る程度であり，現実性を失うものではない．

- レジスタ通信機構

  レジスタ通信機構は行わないこととする．実際に効率的なスレッドレベル投機実行を行うためには，生成したレジスタの値を直接，それを親とするスレッドに送る必要がある．レジスタの値を送信するには，専用のコンパイラによって明示的に受け渡しをする専用命令を挿入する方法や，ハードウェアで制御する方法が提案されている．しかし，各スレッドが生成するレジスタの値を受け渡しはその数が少なく，また，実際に専用コンパイラ等により，レジスタ通信を極力行わないよう最適化するのが一般的であるため，シミュレータの有効性を失うものではない．

  本研究では，スレッドが生成したレジスタの値は，常にメモリに格納するようにコンパイルを行い，すべてのスレッド間の値の受け渡しをメモリを介して行うこととした．

表 5.3: シミュレータの基本仕様

Instruction Set Architecture	MIPS
Processing Unit	4 Unit
Cache Hit Latency	1 cycle
Cache Miss Latency	20 cycles
Block Forward Latency	6 cycles
Block Load Latency	6 cycles
Integer Register	32 registers
Thread Start Latency	4 cycles
Thread Restart Latency	4 cycles
Thread Retire Latency	4 cycles

以上をまとめると，シミュレータの基本仕様は表 5.3 のようになる．

5.2 結果

シミュレーションで得られた結果を示し，それぞれの項目について考察する．処理速度の尺度として IPC を用いた．これは，全命令からスレッド開始，終了命令を除き，その命令数をプログラムの実行に要したサイクル数で割った，実効 IPC である．また，命令数は，有効実行命令数とし，Squash 等により破棄された命令はカウントしない．

5.2.1 全体評価

ラインサイズ

ラインサイズを 8，16，32，64 バイトと変化させた場合の各ベンチマークプログラムの IPC の結果を図 5.1 から図 5.24 に示す．それぞれ提案手法である同期型キャッシュ，冗長キャッシュ，それぞれ同時に行った結果を併せて示した．ラインサイズが，8 バイトの場合はシミュレータ上の最小の語長であるシングルワード単位である．したがって，16，32，64 バイトの複数語を含む場合とメモリ依存違反の取り扱いが異なる．

一般的に，ラインサイズを大きくすると，空間的局所性をより多く利用することができるようになる．しかし，スレッドレベル投機実行を行う場合は，ラインサイズを大きくなるとメモリ投機違反の発生が多くる．これは，第 3 章で述べたフォールスシェアリングの問題があるからである．今回ベンチマークプログラムで使用した Livermore Kernel の大半は，連続したメモリ領域に小刻に書き込みを行うプログラムであるため，その影響が結果に現れている．

以上の結果から，ラインサイズは8バイトが良い傾向がある．しかし，実際は，ラインサイズが小さくなると，キャッシュライン内のアドレスタグおよび，状態ビットの占める割合が大きくなるため，キャッシュの記憶容量とのバランスにより，一概に8バイトラインが良い結果であるとは言えない．

また，複数ワードラインの場合でも16，32，64バイトと増やすことにより，シミュレーションを行ったプログラムでは，空間的局所性が大きくとれ，キャッシュミス率の軽減が得られた．これは共有バスのトラフィック量の減少につながったためである．以上の状況は，Livermore Kernel 13，14，16，19などで顕著に表れている．

表5.4から表5.6で性能向上率を示す．スレッド間の同期機構で最大約18%，冗長キャッシュを用いた効率化手法で最大約12%の性能向上が見られた．

図5.25から図5.48でキャッシュミス率を示す．同期キャッシュを用いた場合，同期成功時のストールによりキャッシュミス率が軽減する．また，冗長キャッシュによるキャッシュミス率の軽減は，時間的局所性が大きく利用されたためである．

次節以降，それぞれの提案手法について実行結果と照らし合わせて考察を行う．

5.2.2 同期キャッシュの評価

ストアの伝搬削減

ストア伝搬の削減率を図5.49から図5.55に示す．これは，従来型キャッシュのストアの回数と同期型キャッシュ，または同期型キャッシュと冗長キャッシュを併せた手法を比較することで算出した．

本研究で同期を行う投機データは，キャッシュライン毎に管理されるため，キャッシュラインを大きくするとともにストアの伝搬削減率が高くなる傾向にある．また，Squashが一度も起こらない，またはそのラインへの書き込みが1回以下の場合は，ストアの伝搬回数の削減はできない．これは，シングルワードラインの場合に多く見られた．

ストアの伝搬の削減により共有バスのトラフィック量の減少が考えられるが，実際，表5.7から表5.9で示すようにメモリ投機違反回数がほとんど変化していない．例えば，Livermore Kernel 18では，ストアの伝搬の削減率が約50%あるにもかかわらず，約40%程度メモリ投機違反が増加している．このことから，ストアの伝搬の削減は可能であるが，それがメモリ投機違反の増大を引き起こし，結果として全体の性能向上には結び付いていないことがわかる．

ストアの伝搬が削減されると，他の投機スレッドの実行が進み，結果として投機メモリアクセスが増加し，結果としてメモリ投機違反が増大すると考えられる．

同期回数

同期回数とSquash回数を表5.7から表5.9に示す．同期回数は，実際にSqビットによって投機メモリアクセスの同期が成功した回数を示す．また，Squash回数は，直接依存違

反の原因となったSquash回数を直接Squash 回数，親スレッドからのメモリ依存違反の連鎖によるSquash回数を間接Squash回数として表現した．

同期回数もストアの伝搬の削減と同様，他の投機スレッドの実行が進み，他の投機メモリアクセスが増加し，メモリ投機違反が増大する場合がある．結果として全体の性能向上には結び付いていない．また，大半のプログラムでは，同期が成立したスレッドのその後の実行で再びメモリ依存違反を起こすため，全体の性能向上には結び付かない結果となった．

5.2.3 冗長キャッシュを用いた効率化手法の評価

冗長キャッシュを使用した場合の結果について検討する．図5.1から図5.24で示されるように時間的局所性を大きく利用することにより，最大で12%の性能向上が得られた．これは，待機中のキャッシュにストアの伝搬による最新のデータが次に割り当てらるスレッドでキャッシュヒットするためである．特に，シングルワードラインの場合，または，Squash回数が低い，もしくは，一回も起こさない場合，性能向上率が高い結果を示す傾向にある．

一方，Squash回数が高い場合でも，Squashされる可能性の高い投機スレッドのキャッシュヒットによる共有バスのトラフィック量の軽減により性能の向上が得られた．この場合の，共有バスのトラフィック量の軽減は，前に述べたストアの伝搬削減によるものとは性質が異なる．投機度が最も高く，最も依存違反の起こりやすいスレッド実行は，いずれメモリ投機違反を起こすため，共有バスのトラフィック量の軽減は意味がある．

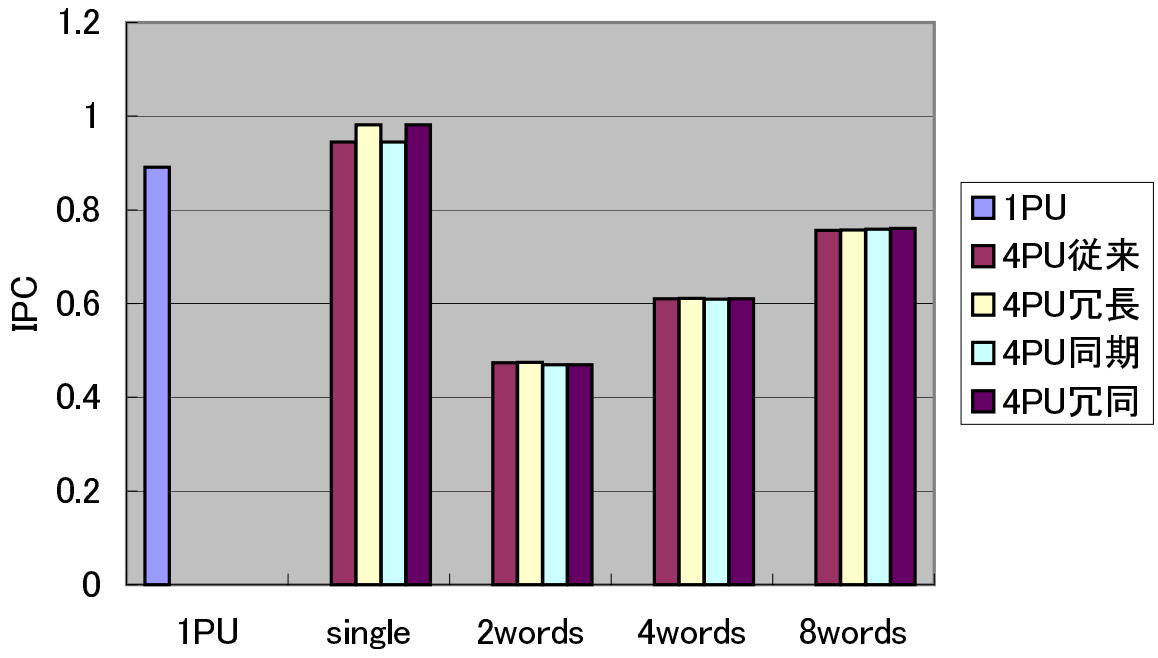


図 5.1: Livermore Kernel-01 IPC

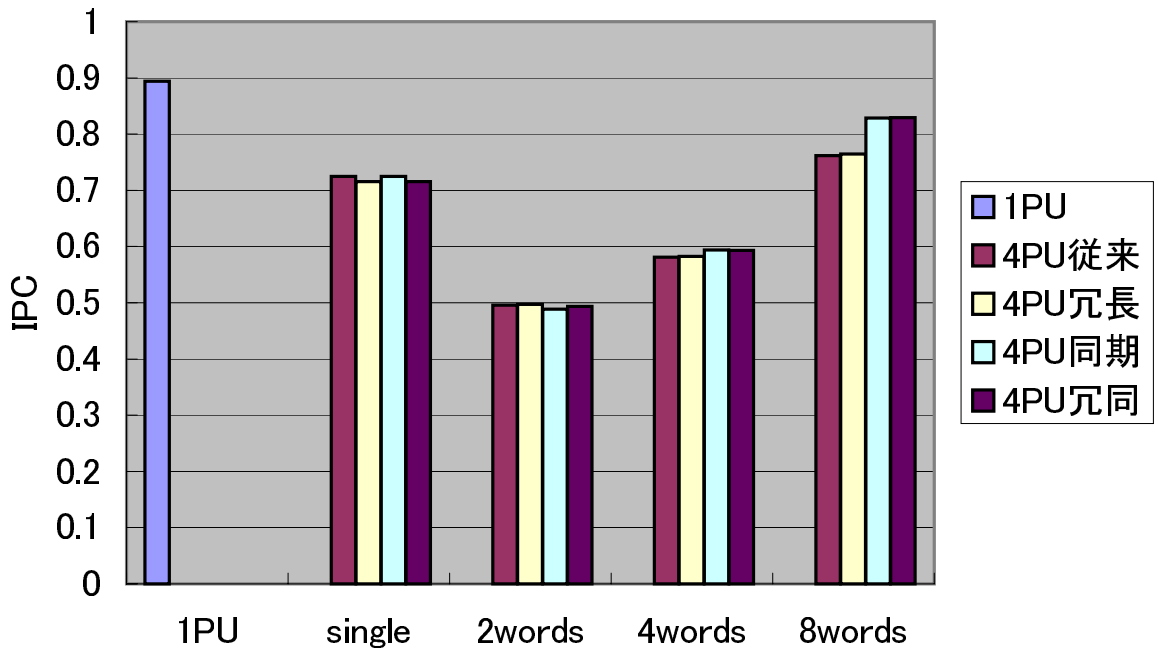


図 5.2: Livermore Kernel-02 IPC

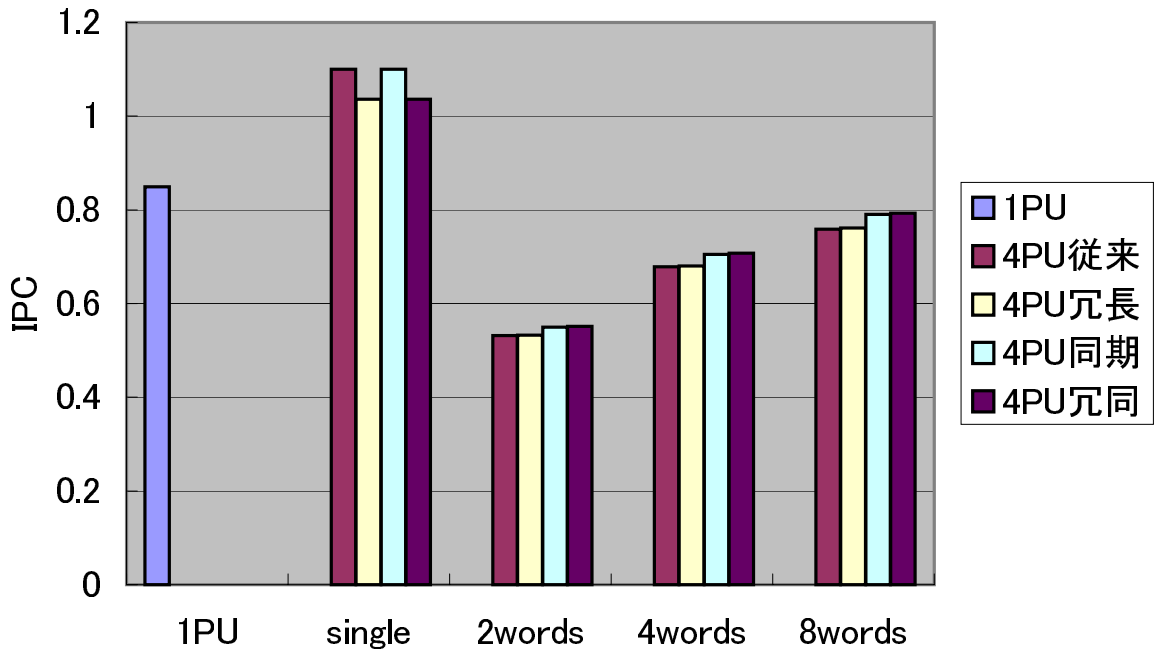


図 5.3: Livermore Kernel-03 IPC

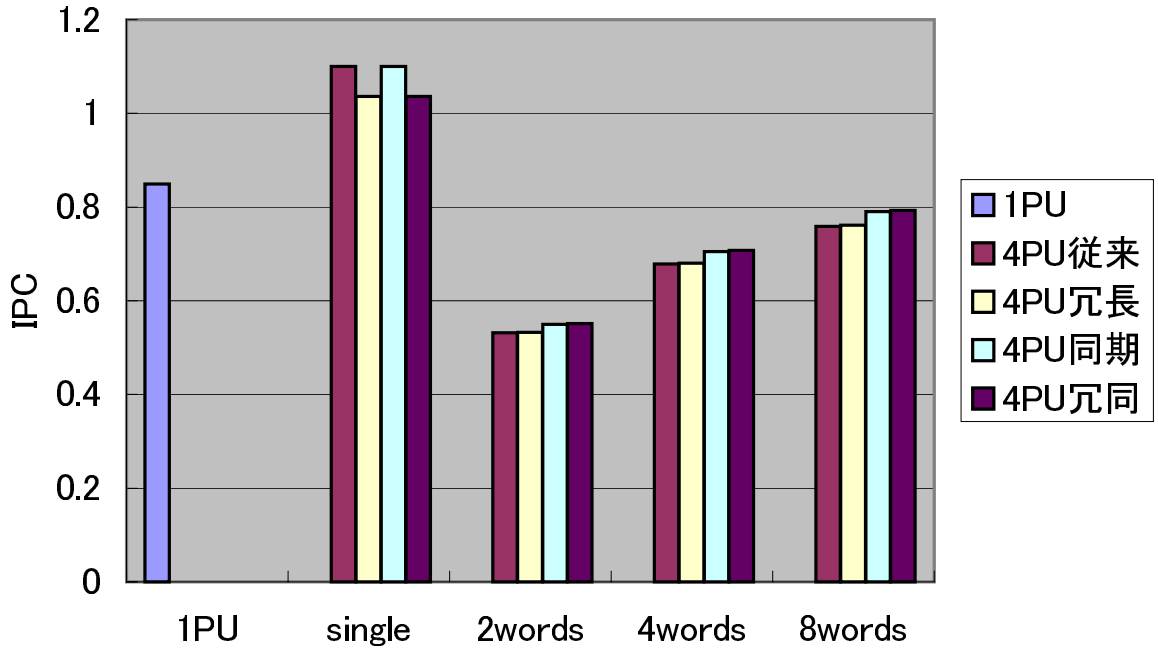


図 5.4: Livermore Kernel-04 IPC

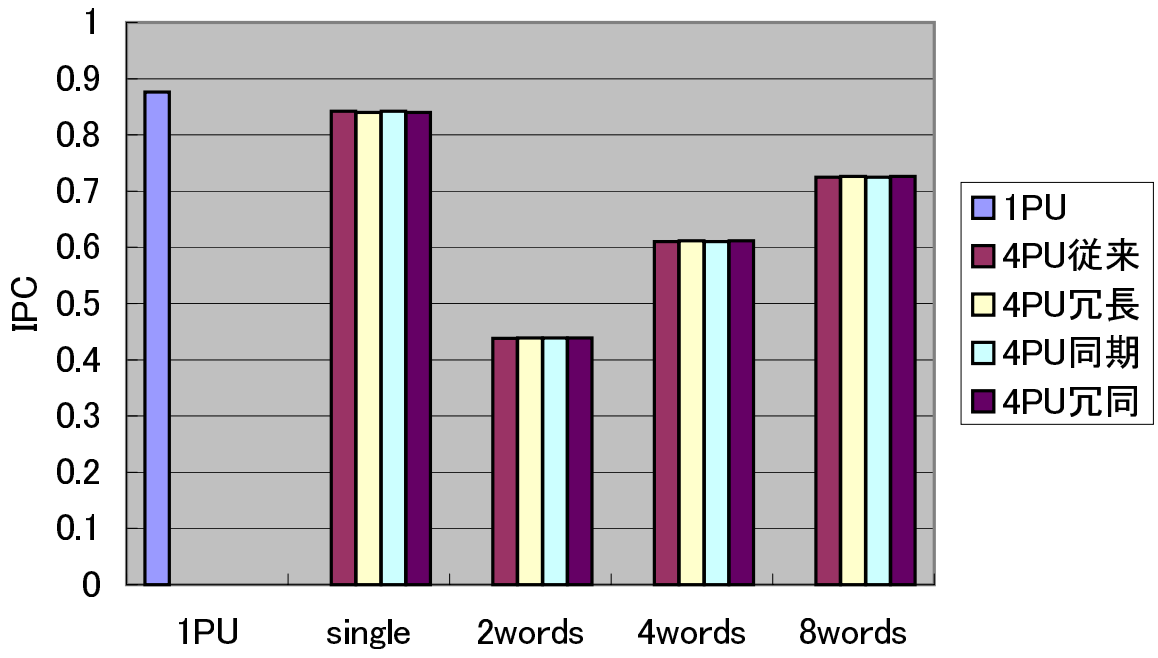


図 5.5: Livermore Kernel-05 IPC

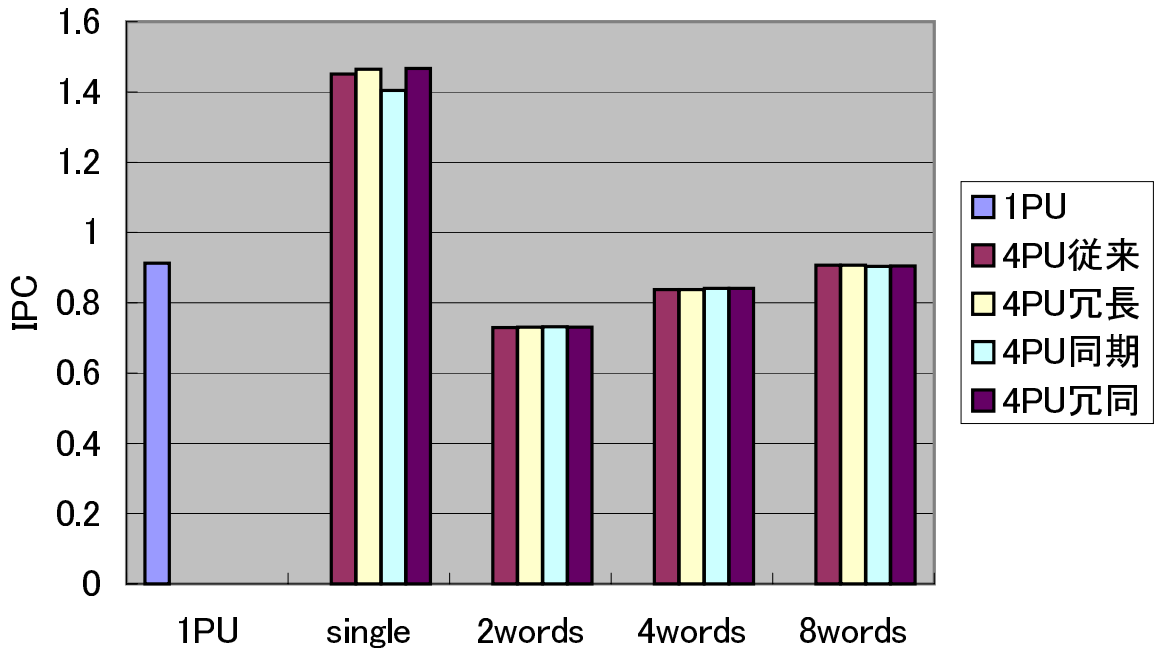


図 5.6: Livermore Kernel-06 IPC

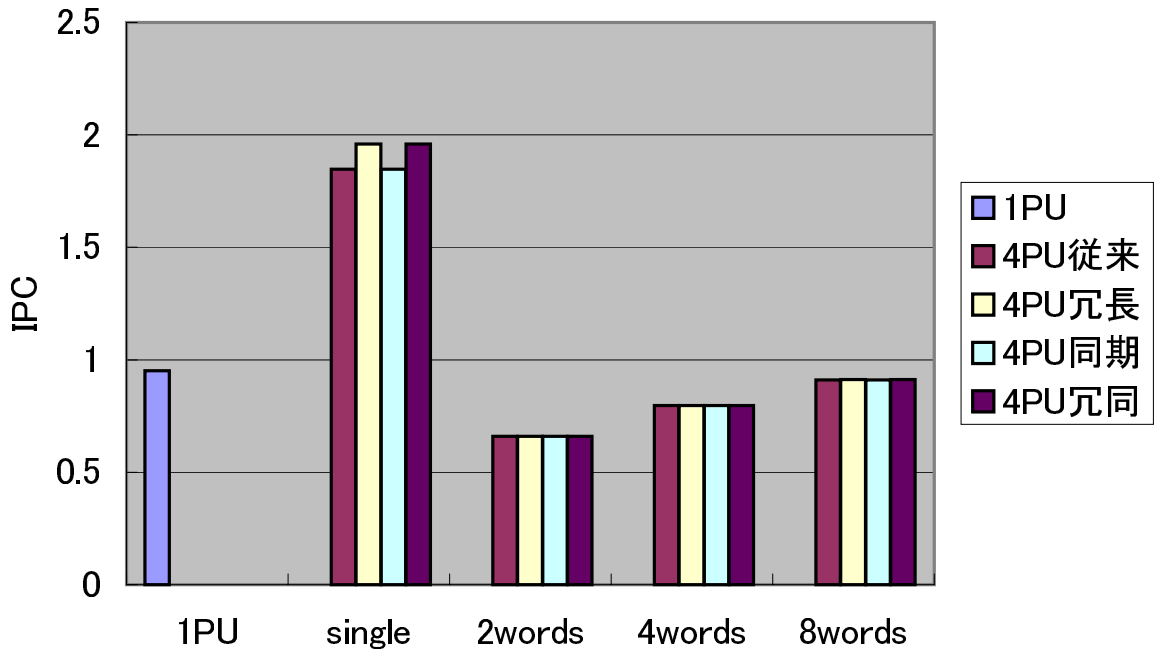


図 5.7: Livermore Kernel-07 IPC

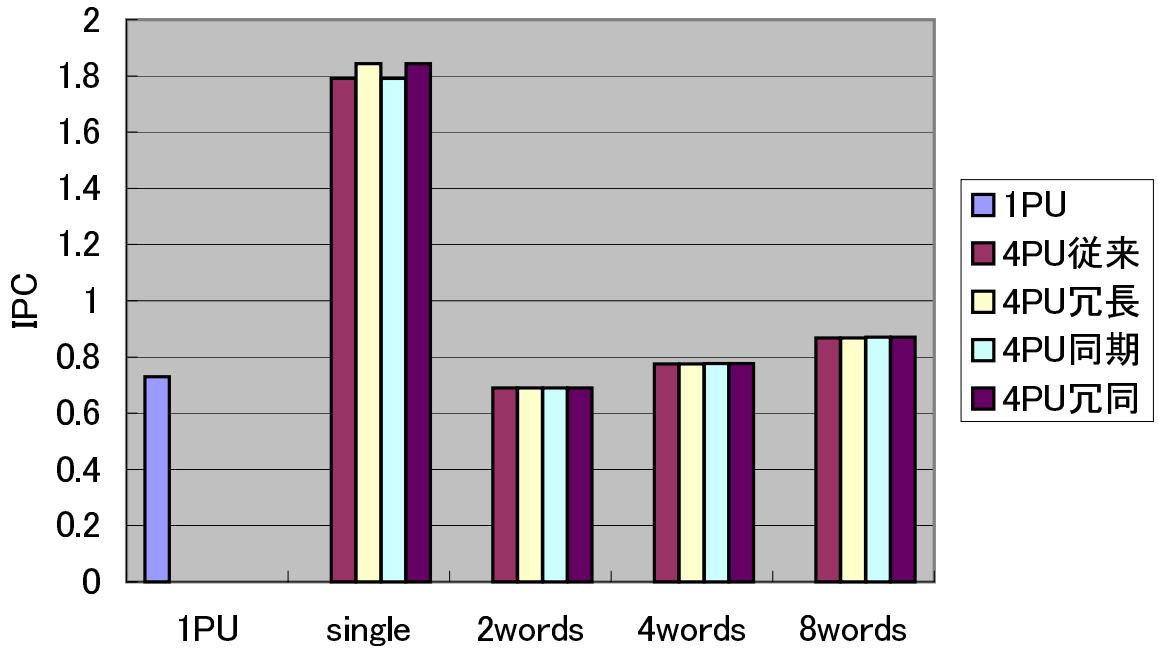


図 5.8: Livermore Kernel-08 IPC

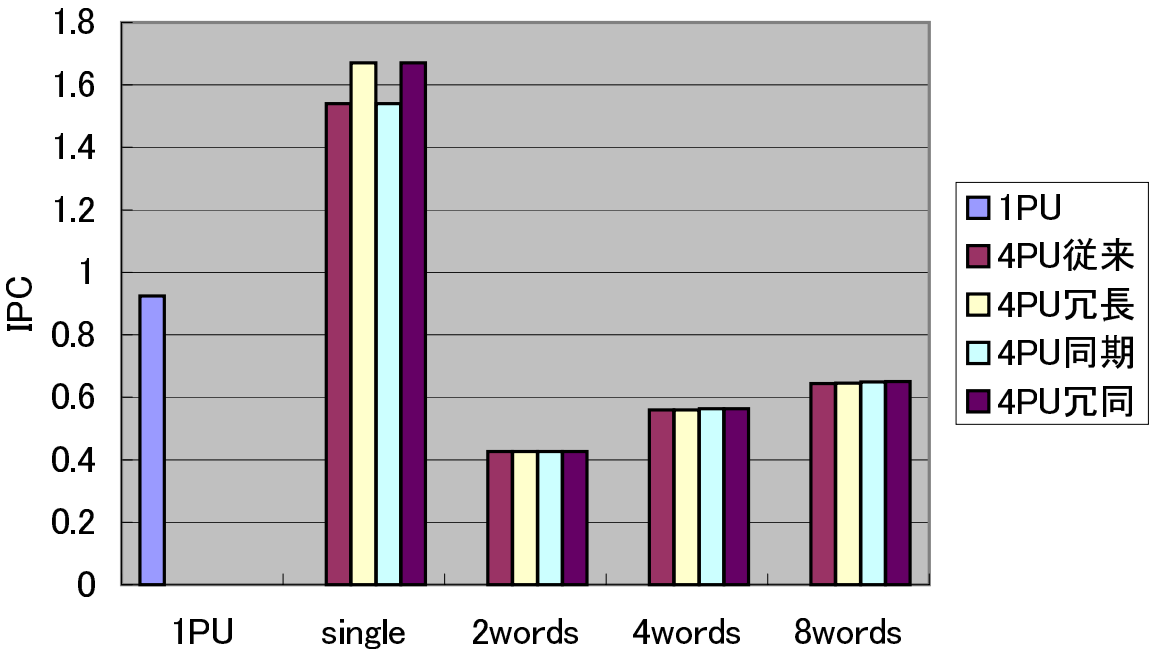


図 5.9: Livermore Kernel-09 IPC

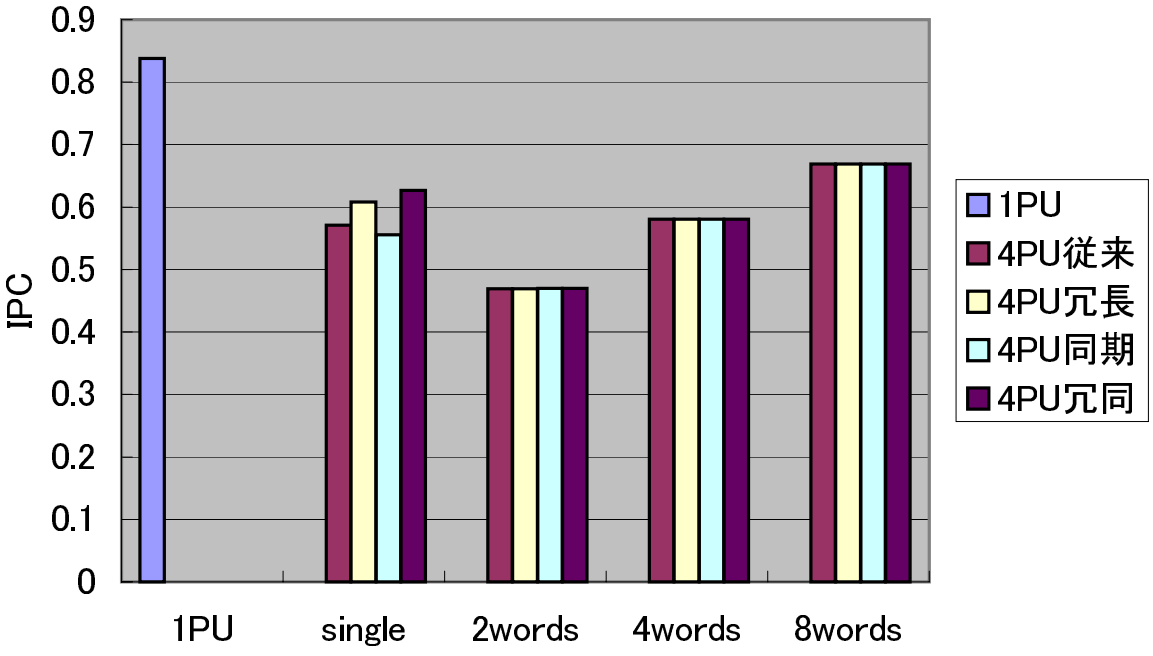


図 5.10: Livermore Kernel-10 IPC

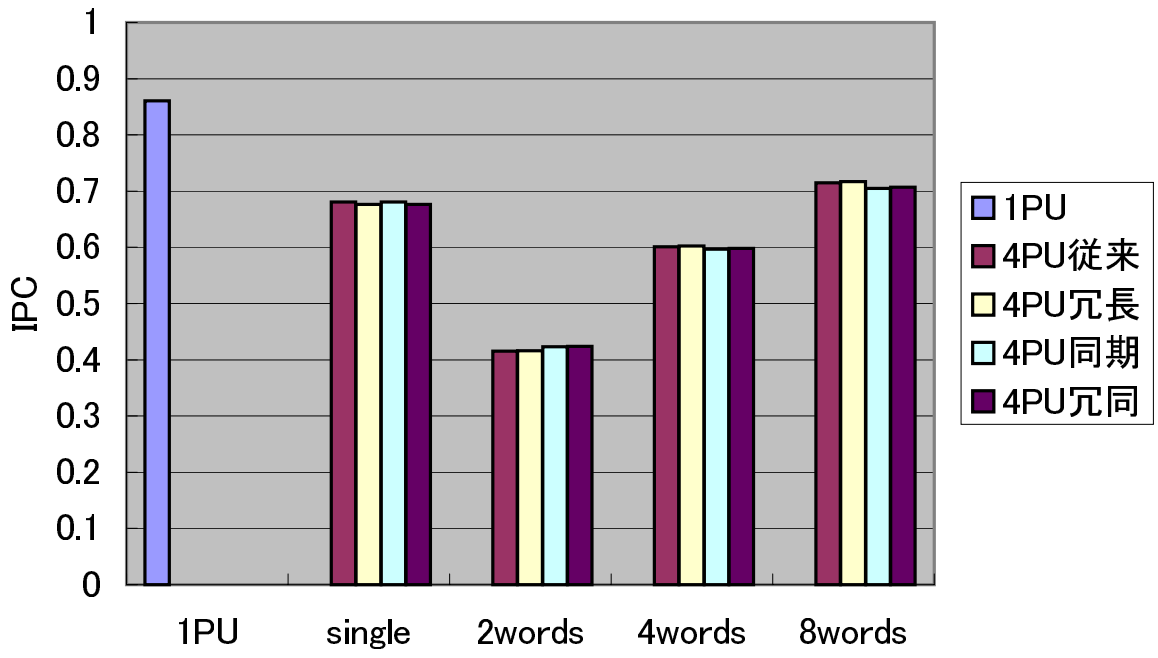


図 5.11: Livermore Kernel-11 IPC

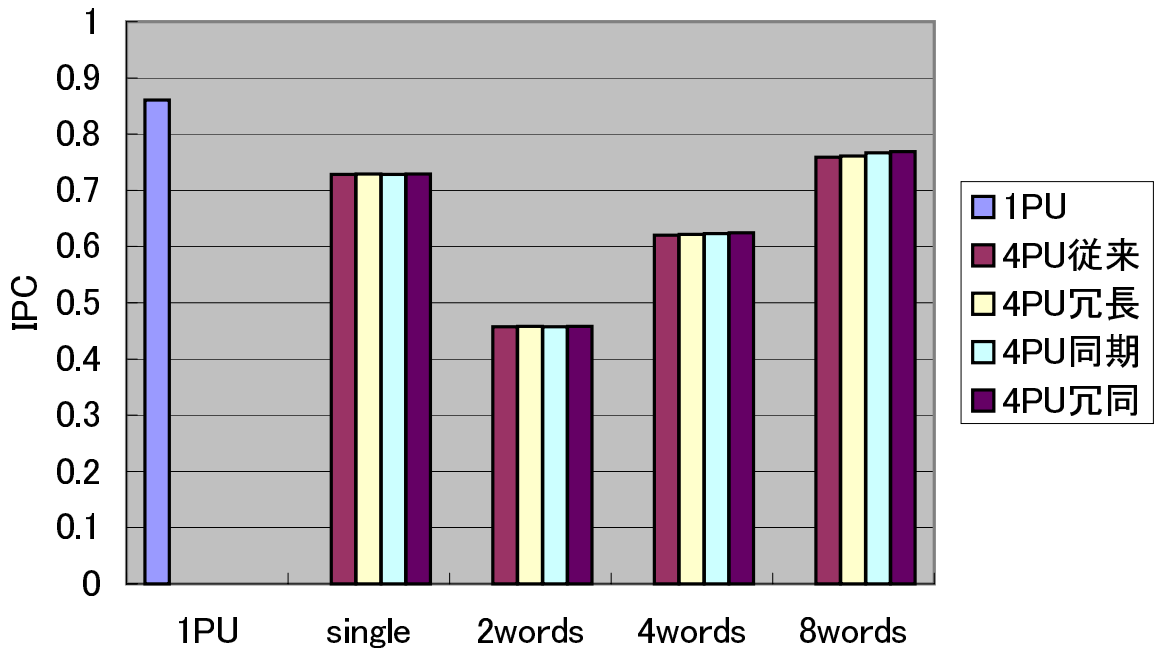


図 5.12: Livermore Kernel-12 IPC

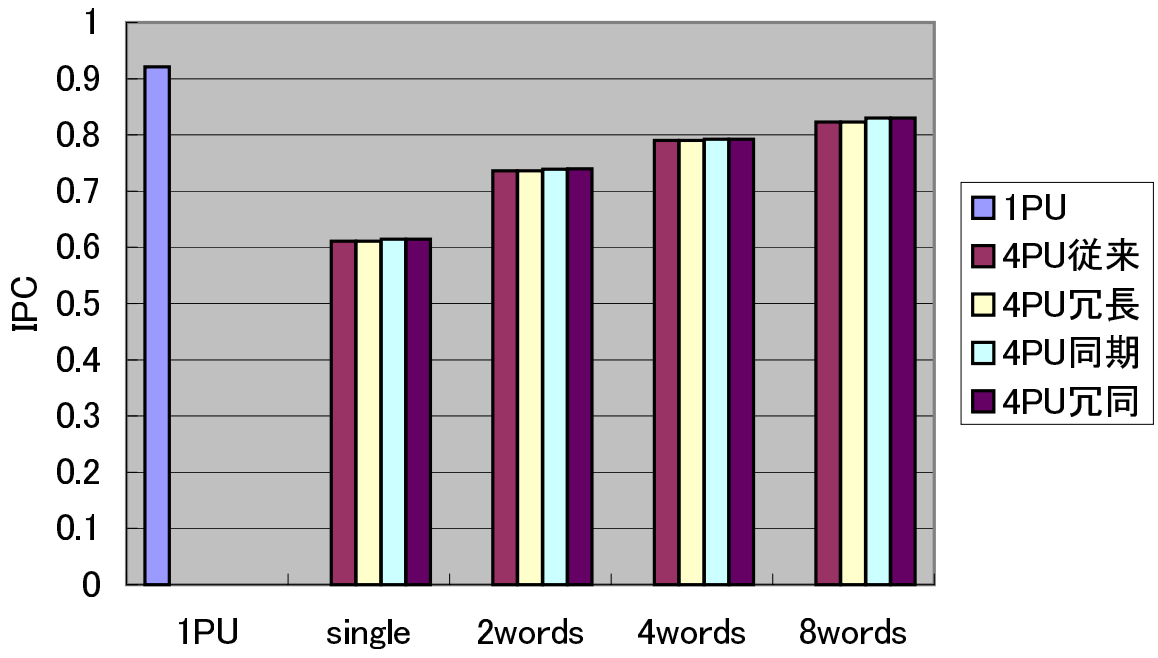


図 5.13: Livermore Kernel-13 IPC

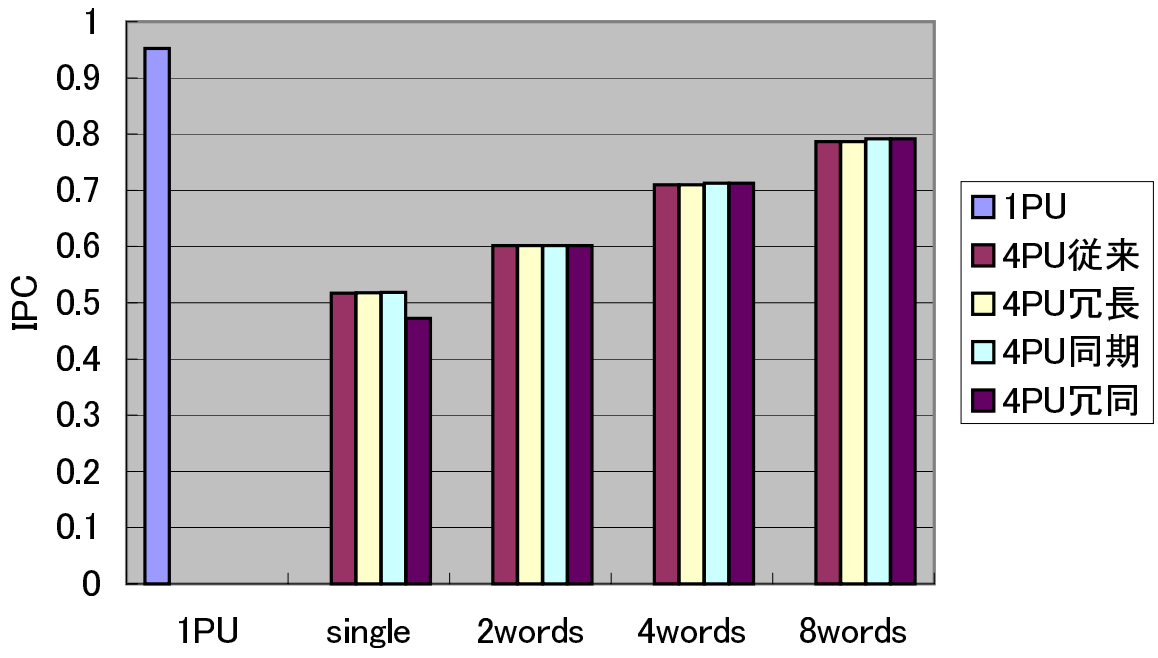


図 5.14: Livermore Kernel-14 IPC

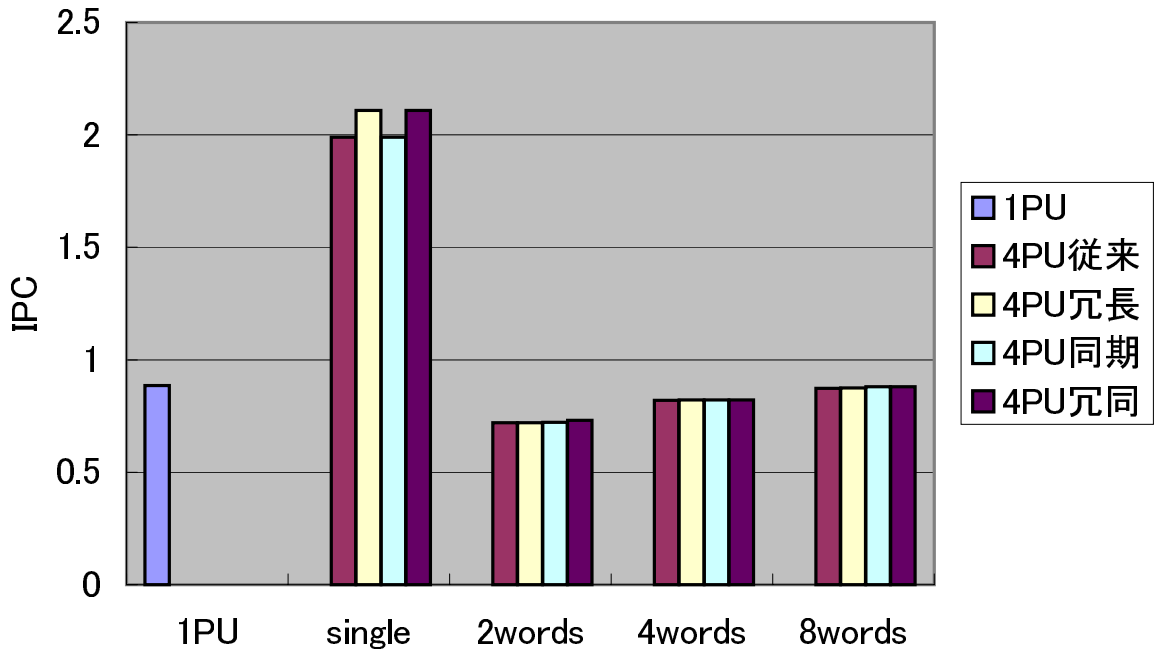


図 5.15: Livermore Kernel-15 IPC

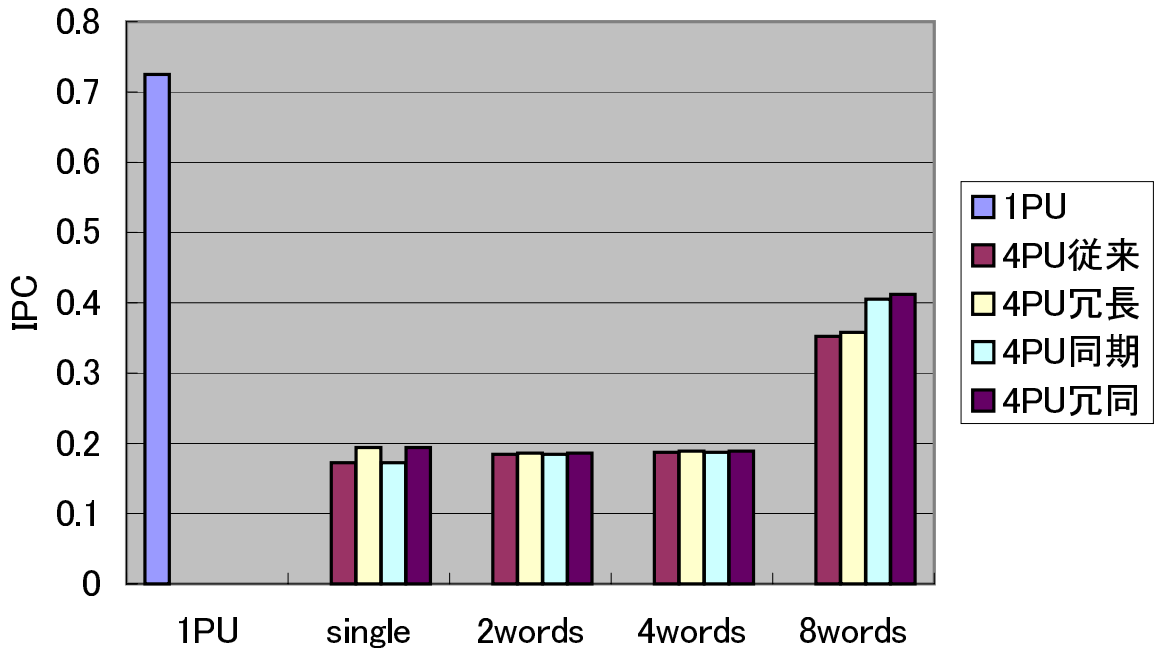


図 5.16: Livermore Kernel-16 IPC

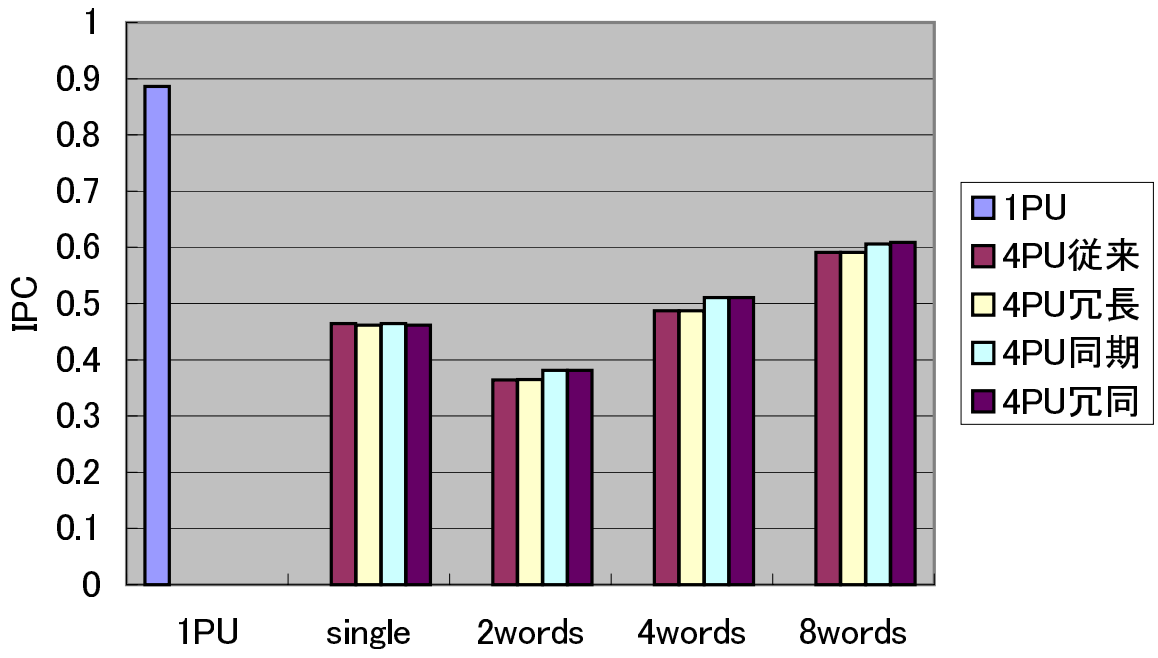


図 5.17: Livermore Kernel-17 IPC

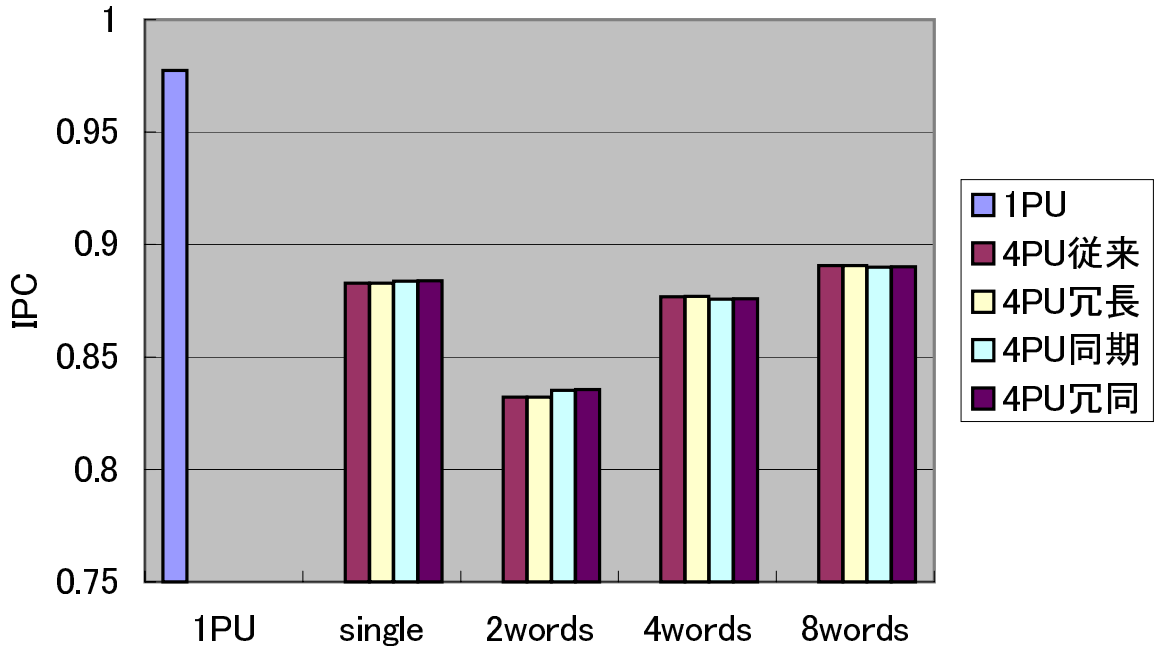


図 5.18: Livermore Kernel-18 IPC

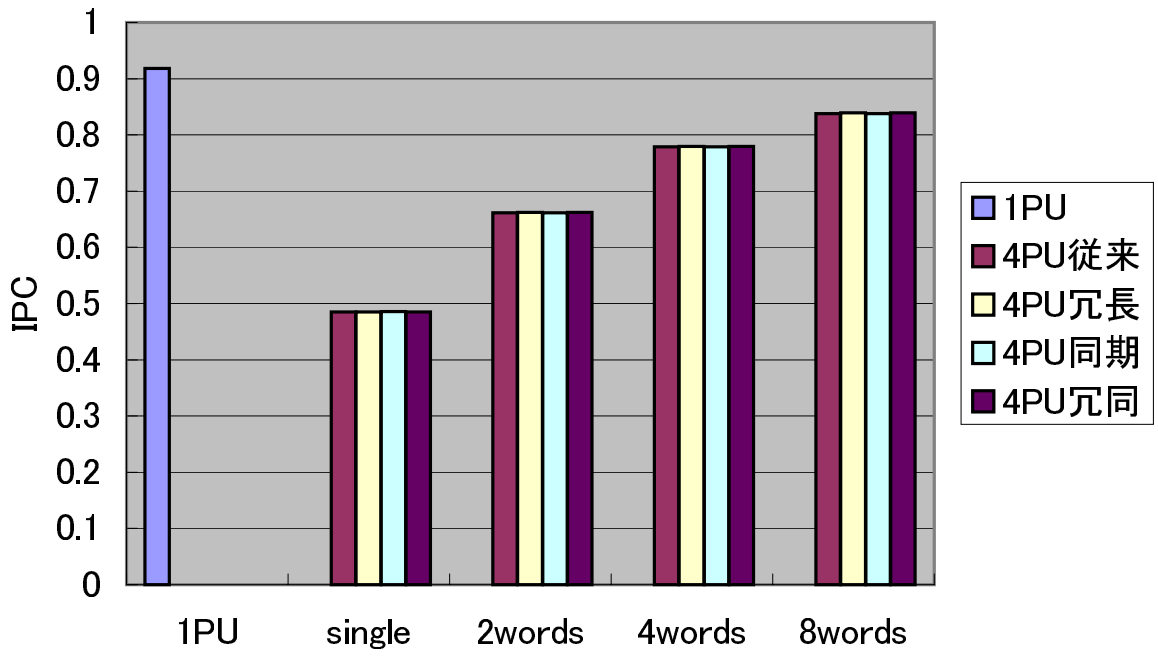


図 5.19: Livermore Kernel-19 IPC

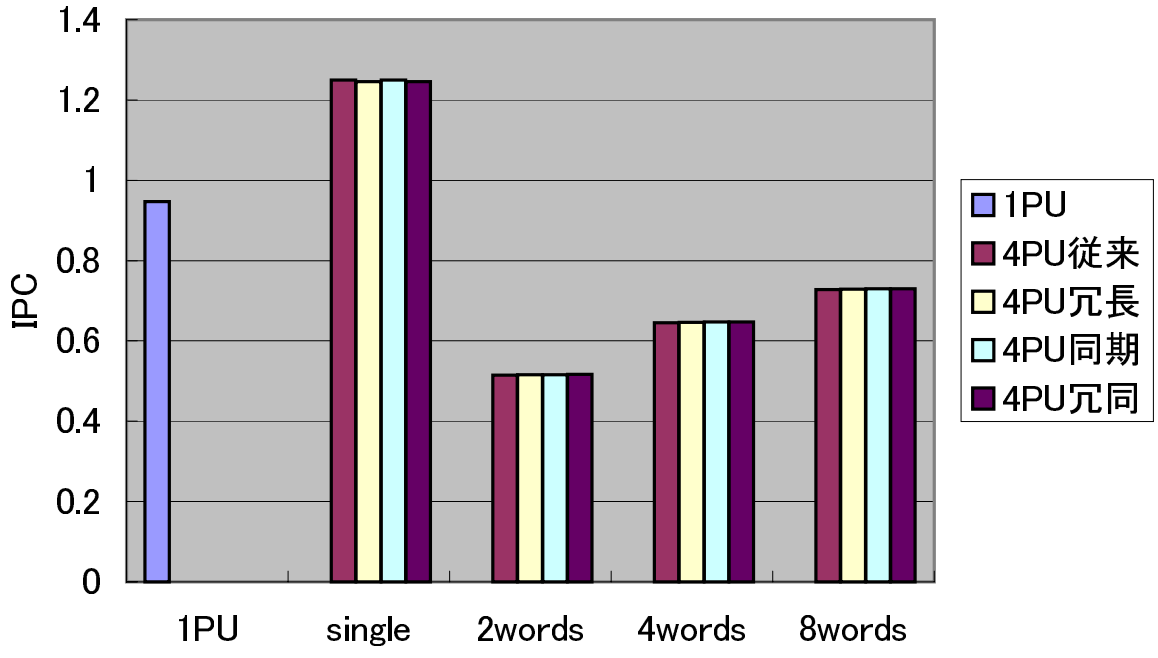


図 5.20: Livermore Kernel-20 IPC

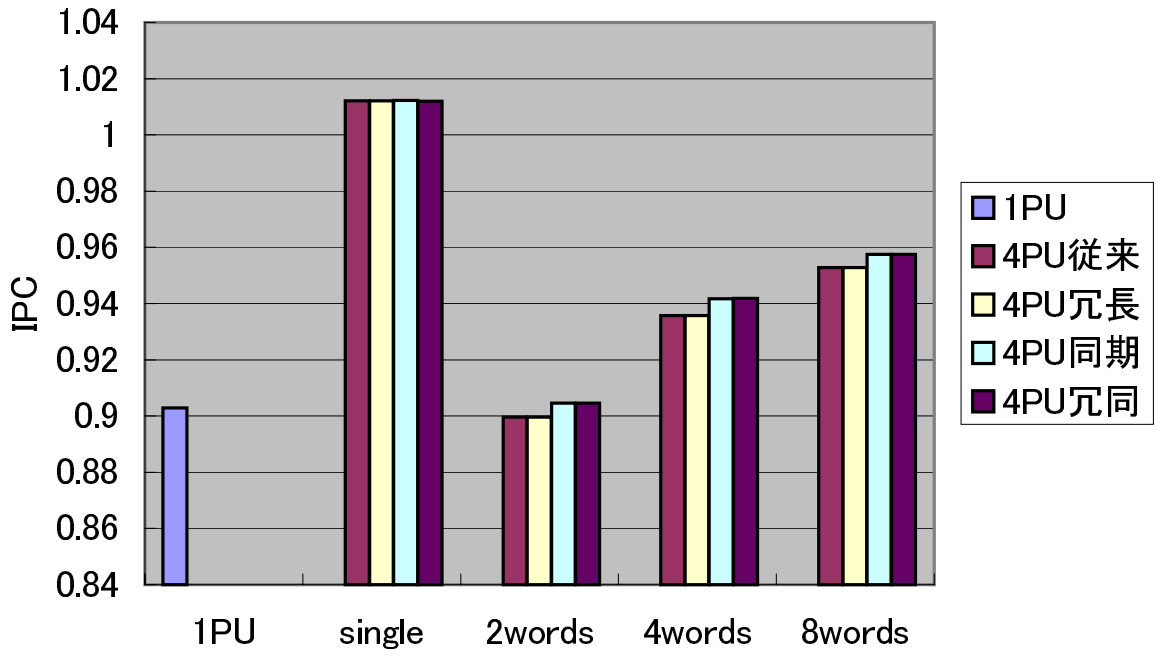


図 5.21: Livermore Kernel-21 IPC

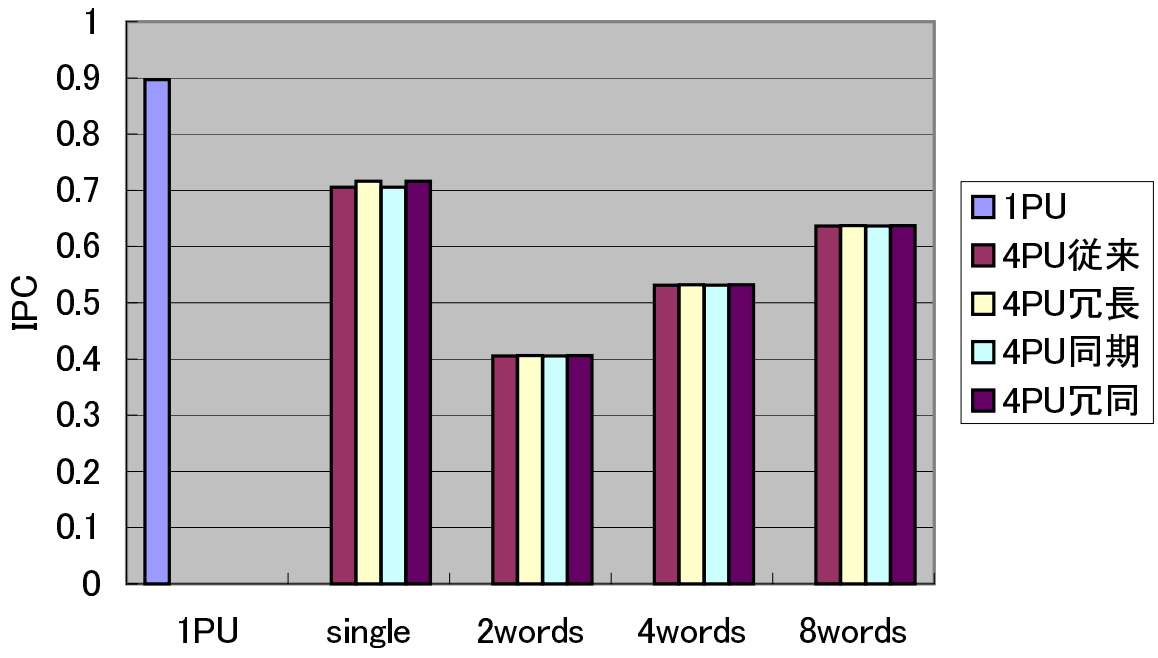


図 5.22: Livermore Kernel-22 IPC

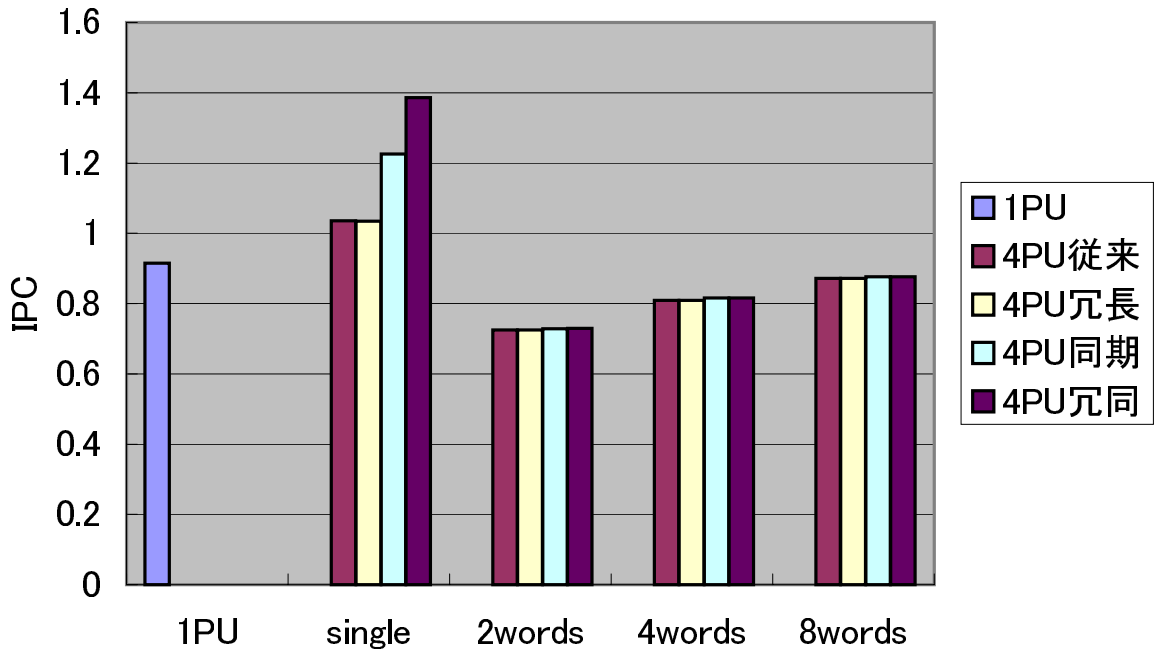


図 5.23: Livermore Kernel-23 IPC

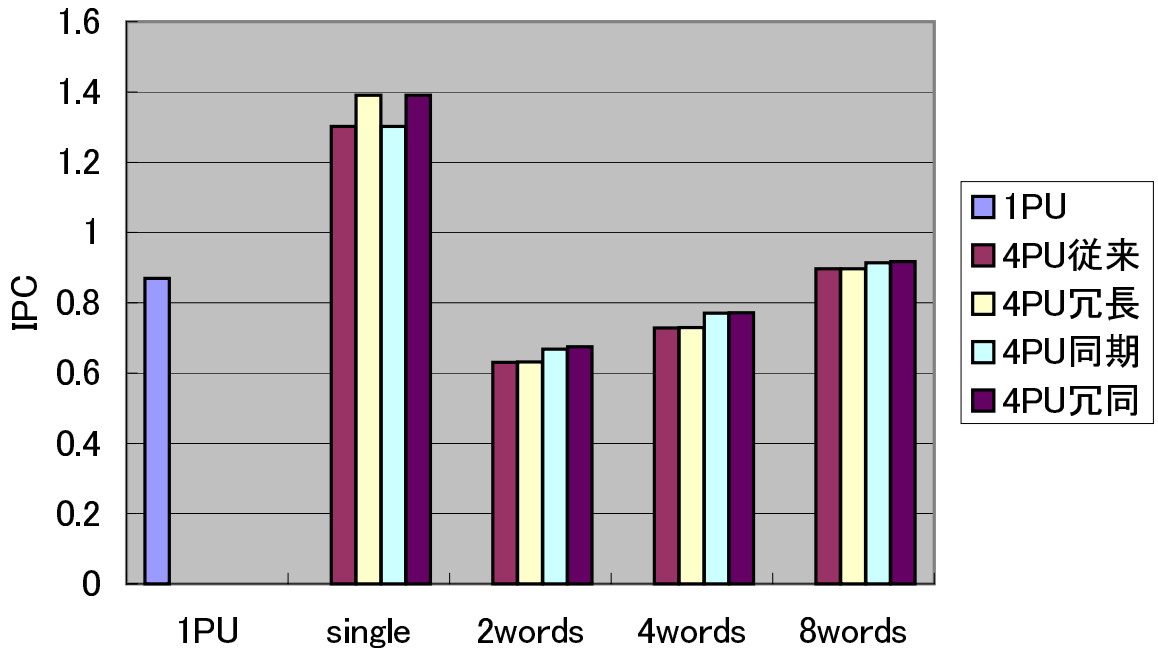


図 5.24: Livermore Kernel-24 IPC

表 5.4: 性能向上率-（1）

ベンチマークプログラム	ラインサイズ	性能向上率（%）		
		冗長キャッシュ	同期キャッシュ	冗長＋同期
Livermore Kernel 01	single	3.92	0	3.92
	2words	0.12	-1.02	-0.91
	4words	0.15	-0.07	0.07
	8words	0.19	0.39	0.58
Livermore Kernel 02	single	-1.32	0	-1.32
	2words	0.21	-1.51	-0.51
	4words	0.24	2.28	2.13
	8words	0.32	8.75	8.88
Livermore Kernel 03	single	-5.83	0	-5.83
	2words	0.21	3.45	3.67
	4words	0.27	3.97	4.26
	8words	0.3	4.18	4.51
Livermore Kernel 04	single	0.21	0.66	0.2
	2words	0.16	-0.55	-0.39
	4words	0.17	1.57	1.75
	8words	0.22	3.15	3.38
Livermore Kernel 05	single	-0.27	0	-0.27
	2words	0.14	0.03	0.17
	4words	0.19	0	0.19
	8words	0.22	0	0.22
Livermore Kernel 06	single	0.99	-3.2	1.1
	2words	0.04	0.21	0.08
	4words	0.05	0.36	0.41
	8words	0.05	-0.34	-0.29
Livermore Kernel 07	single	6.04	0	6.04
	2words	0.06	0	0.06
	4words	0.08	0	0.08
	8words	0.09	0.03	0.11
Livermore Kernel 08	single	2.87	0	2.87
	2words	0.05	0.12	0.13
	4words	0.03	0.07	0.11
	8words	0.04	0.39	0.42

表 5.5: 性能向上率-（2）

ベンチマークプログラム	ラインサイズ	性能向上率（%）		
		冗長キャッシュ	同期キャッシュ	冗長＋同期
Livermore Kernel 09	single	8.43	0	8.43
	2words	0.03	0	0.03
	4words	0.04	0.65	0.69
	8words	0.05	0.73	0.98
Livermore Kernel 10	single	6.58	-2.62	9.79
	2words	0.02	0.22	0.24
	4words	0.03	0	0.03
	8words	0.03	0	0.03
Livermore Kernel 11	single	-0.54	0	-0.54
	2words	0.16	1.88	2.05
	4words	0.24	-0.68	-0.4
	8words	0.28	-1.34	-1.08
Livermore Kernel 12	single	0.16	0	0.16
	2words	0.16	0	0.16
	4words	0.22	0.49	0.72
	8words	0.27	1.06	1.34
Livermore Kernel 13	single	0	0.58	0.51
	2words	0	0.43	0.53
	4words	0	0.27	0.27
	8words	0	0.9	0.9
Livermore Kernel 14	single	0.09	0.28	-8.74
	2words	0.02	0.01	0.03
	4words	0.03	0.43	0.46
	8words	0.03	0.61	0.64
Livermore Kernel 15	single	5.96	0	5.96
	2words	0.01	0.12	1.51
	4words	0.15	0.09	0.28
	8words	0.16	0.78	0.88
Livermore Kernel 16	single	12.53	0	12.53
	2words	0.8	0	0.8
	4words	0.81	0	0.81
	8words	1.54	14.94	16.98

表 5.6: 性能向上率-（3）

ベンチマークプログラム	ラインサイズ	性能向上率（%）		
		冗長キャッシュ	同期キャッシュ	冗長＋同期
Livermore Kernel 17	single	-0.71	0.04	-0.68
	2words	0.06	4.56	4.62
	4words	0.08	4.84	4.85
	8words	0.09	2.53	3.02
Livermore Kernel 18	single	0.01	0.11	0.12
	2words	0.01	0.38	0.41
	4words	0.01	-0.13	-0.1
	8words	0.01	-0.08	-0.06
Livermore Kernel 19	single	0.05	0.15	0.09
	2words	0.07	0	0.07
	4words	0.09	0	0.09
	8words	0.09	0	0.09
Livermore Kernel 20	single	-0.34	0	-0.34
	single	0.04	0.19	0.23
	single	0.05	0.26	0.31
	single	0.05	0.25	0.3
Livermore Kernel 21	single	0.01	0.01	-0.01
	2words	0	0.55	0.56
	4words	0	0.65	0.65
	8words	0	0.5	0.5
Livermore Kernel 22	single	1.53	0	1.53
	2words	0.08	0	0.08
	4words	0.1	0	0.1
	8words	0.13	0	0.13
Livermore Kernel 23	single	-0.03	18.44	33.92
	2words	0.01	0.52	0.53
	4words	0.01	0.88	0.89
	8words	0.01	0.44	0.45
Livermore Kernel 24	single	6.86	0	6.86
	2words	0.12	5.94	6.98
	4words	0.13	5.73	5.88
	8words	0.12	1.92	2.39

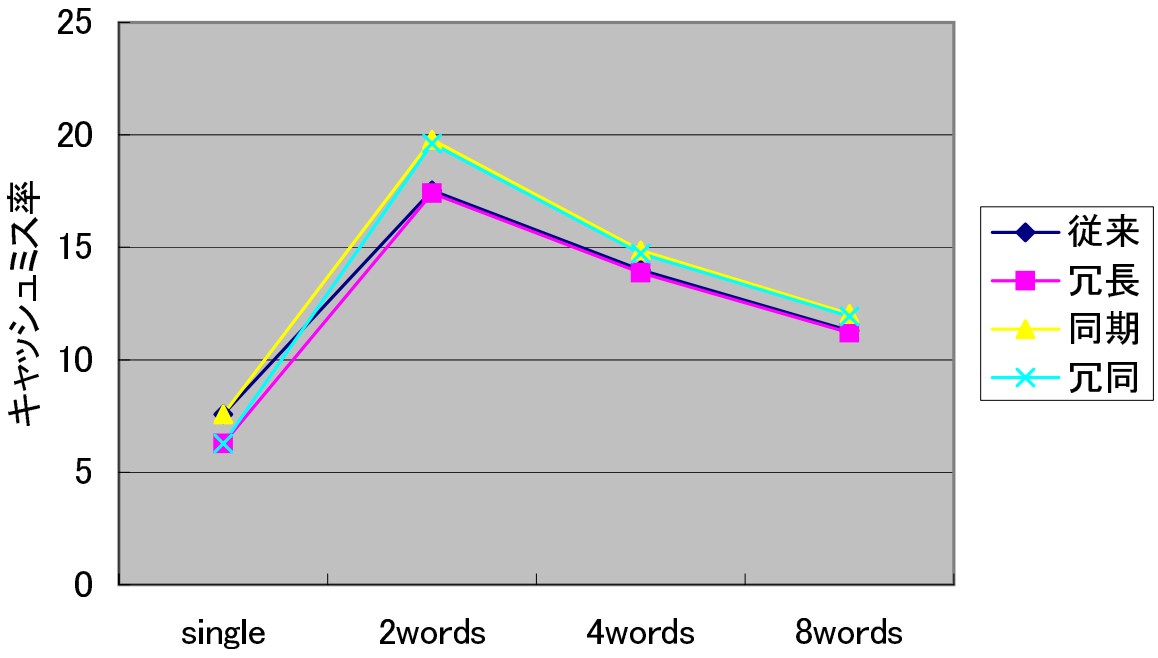


図 5.25: Livermore Kernel-01 キャッシュミス率

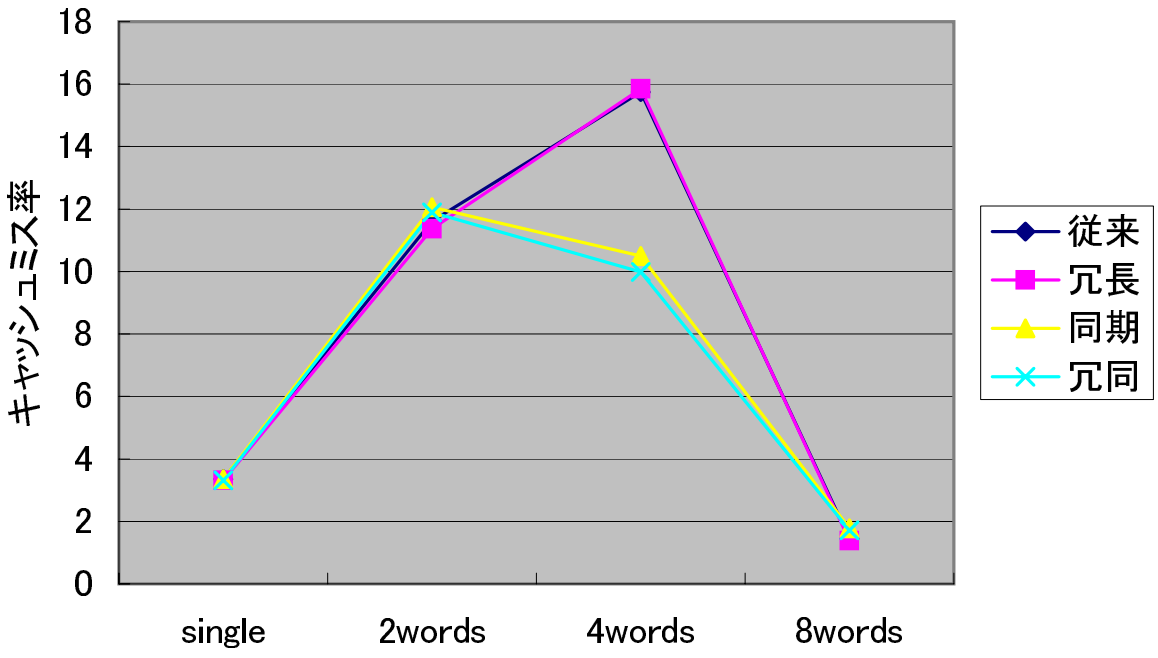


図 5.26: Livermore Kernel-02 キャッシュミス率

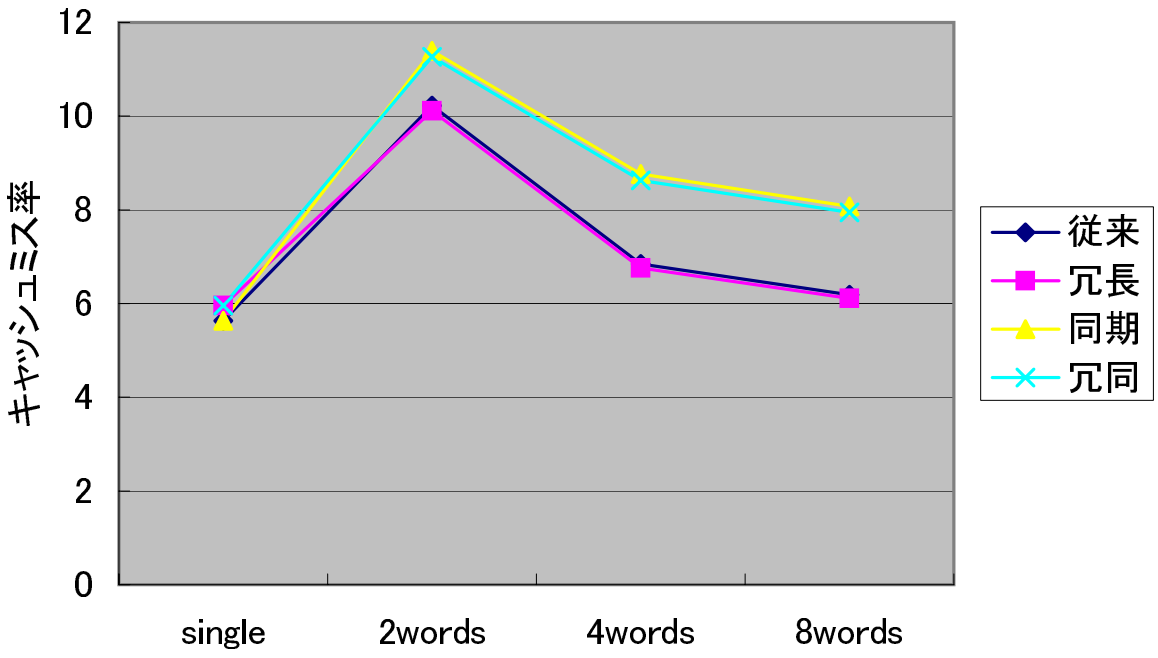


図 5.27: Livermore Kernel-03 キャッシュミス率

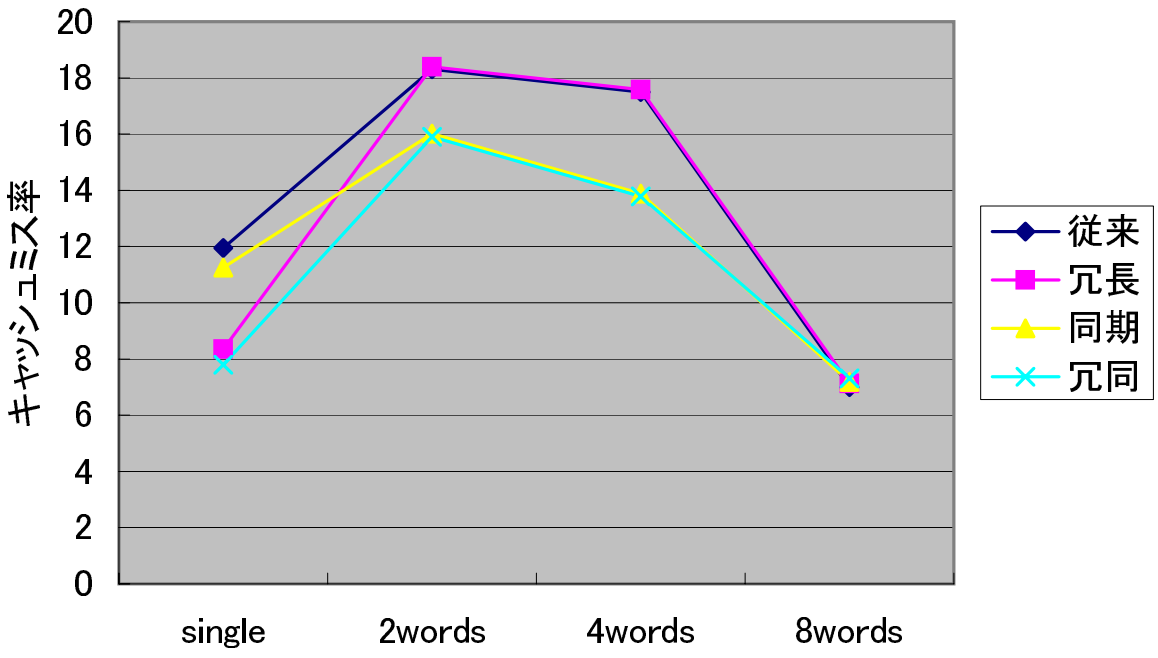


図 5.28: Livermore Kernel-04 キャッシュミス率

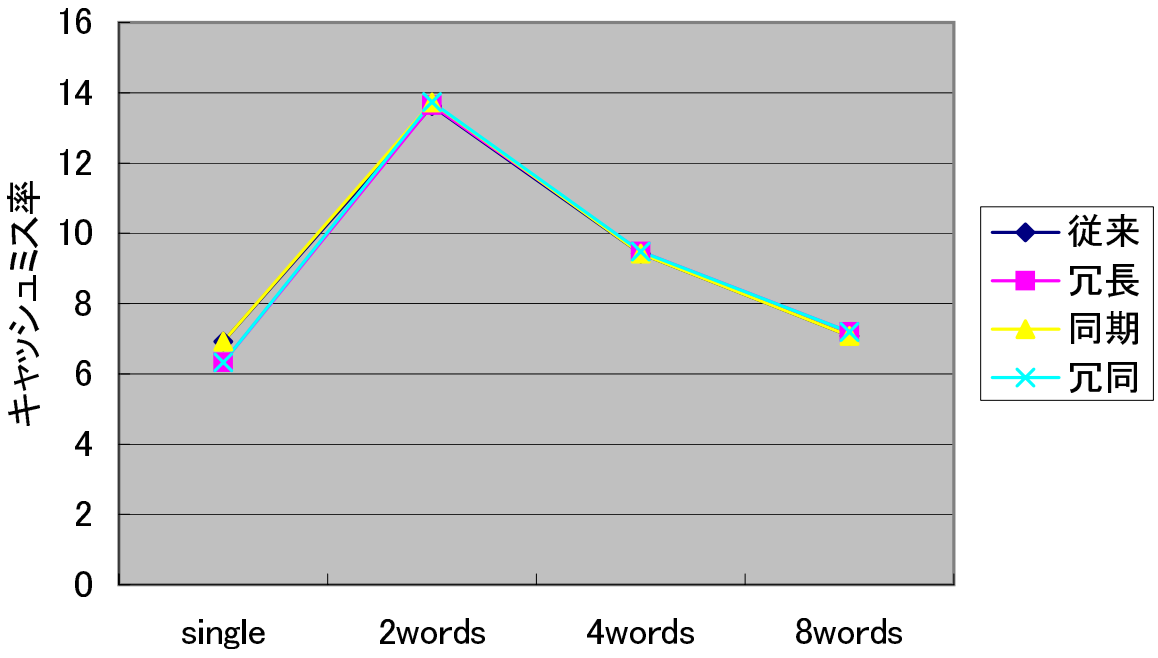


図 5.29: Livermore Kernel-05 キャッシュミス率

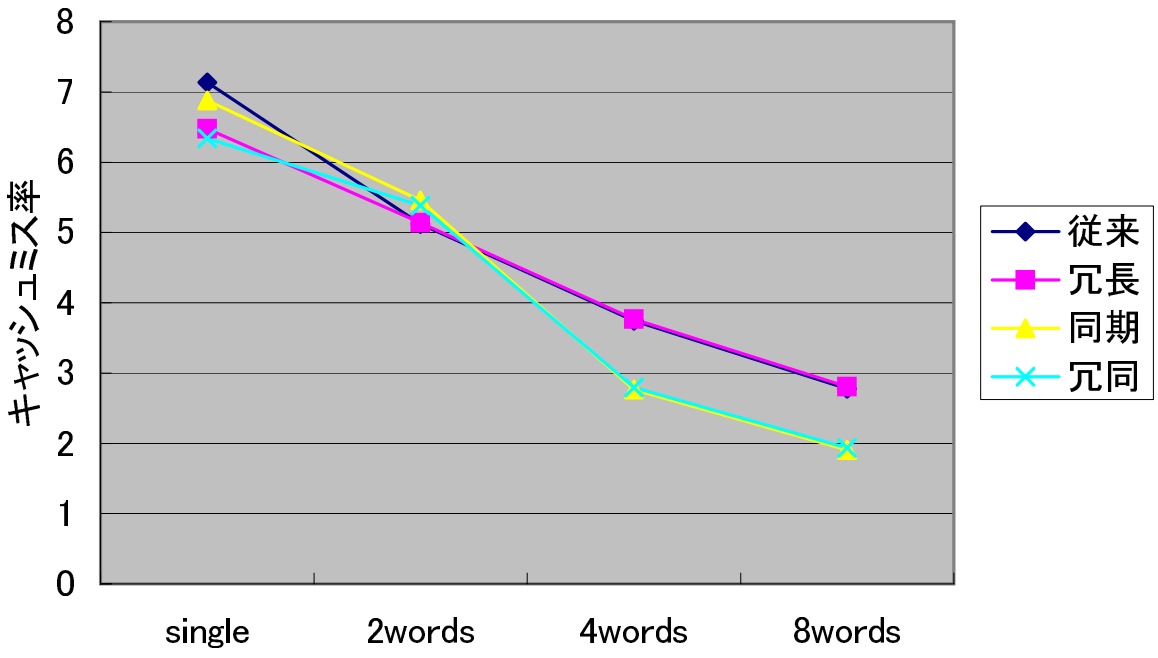


図 5.30: Livermore Kernel-06 キャッシュミス率

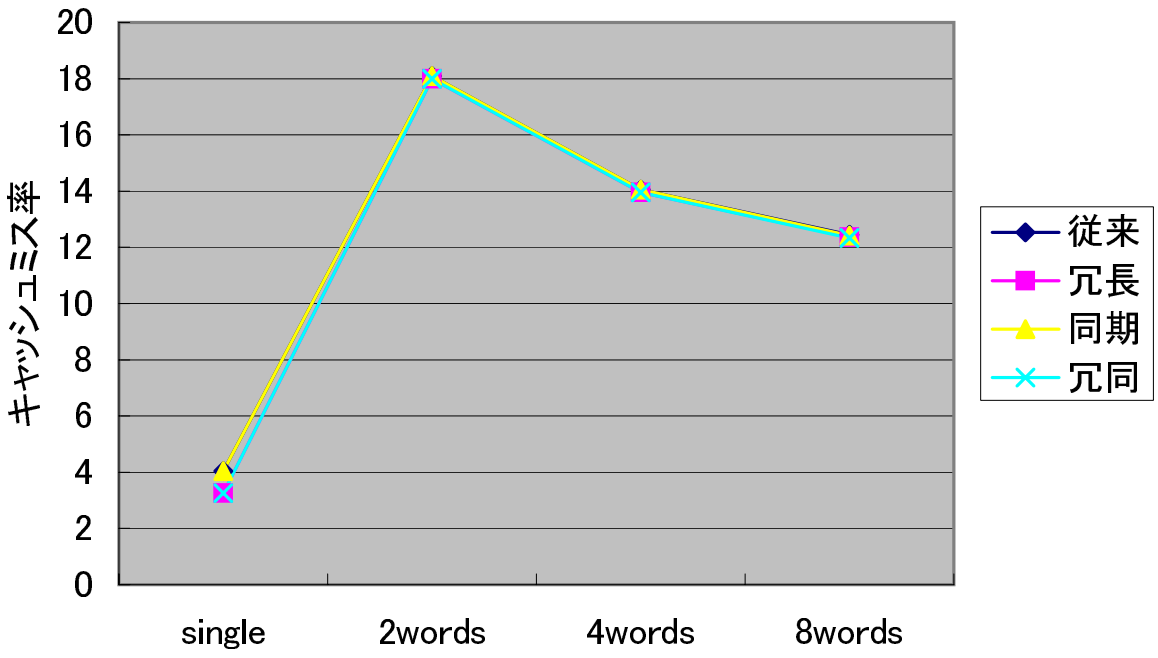


図 5.31: Livermore Kernel-07 キャッシュミス率

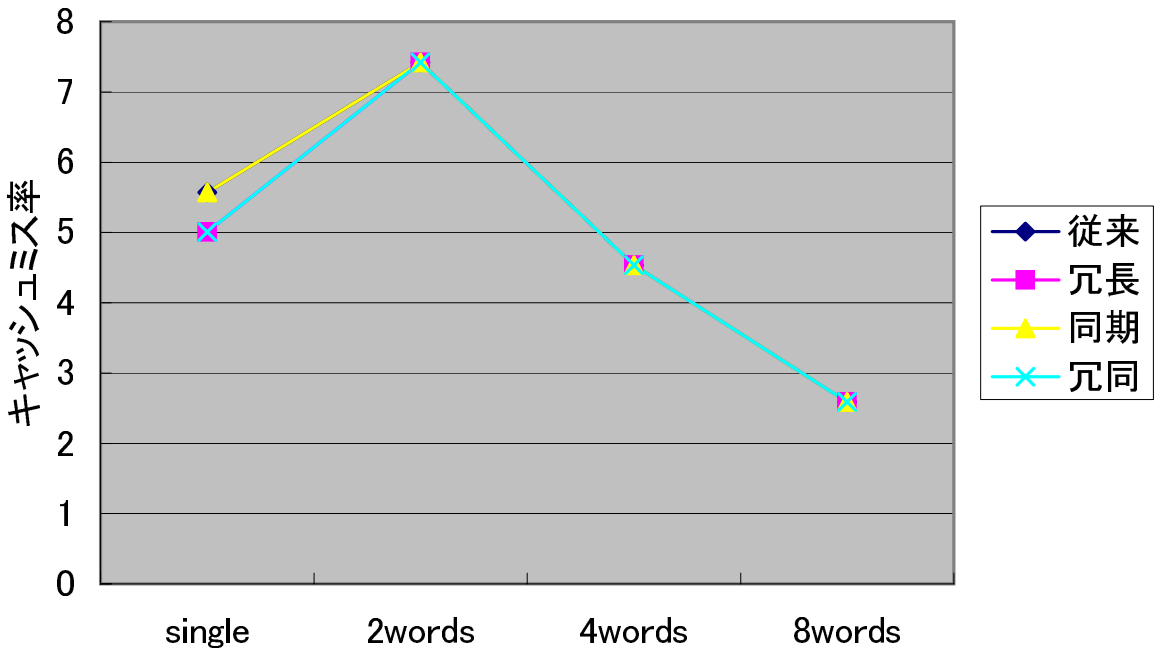


図 5.32: Livermore Kernel-08 キャッシュミス率

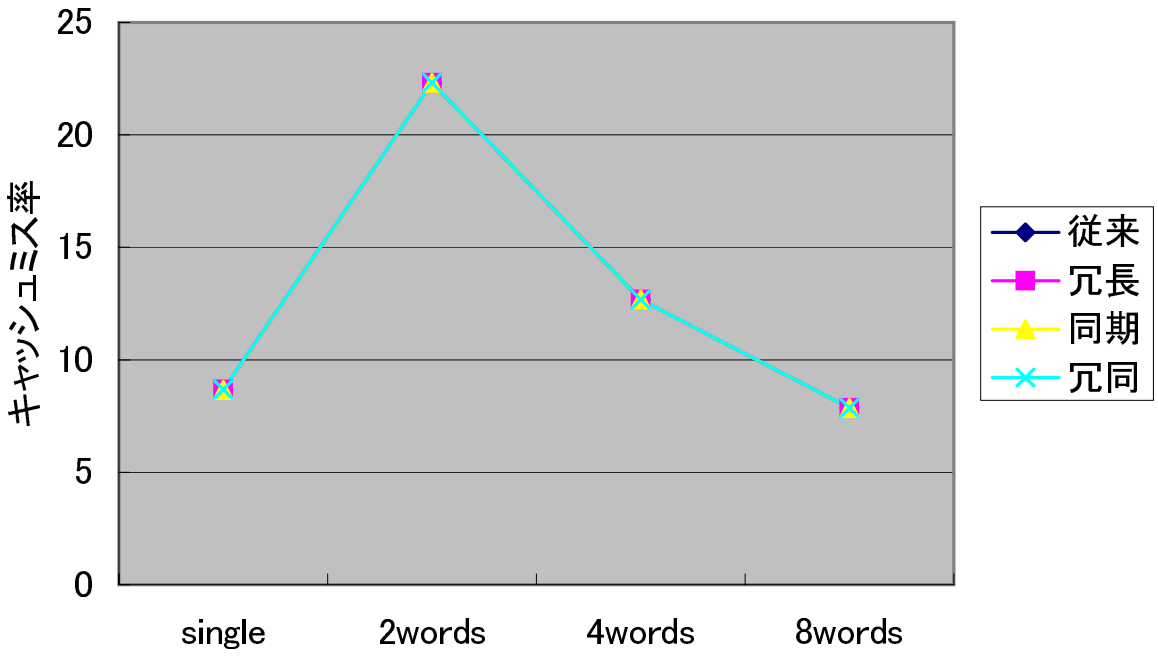


図 5.33: Livermore Kernel-09 キャッシュミス率

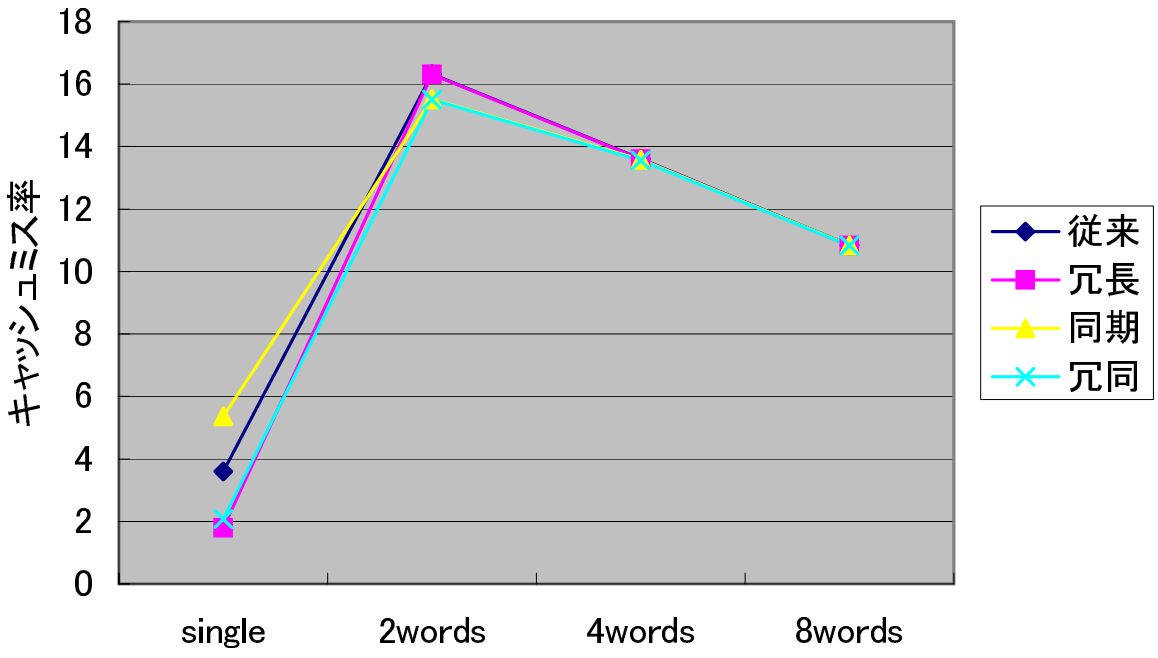


図 5.34: Livermore Kernel-10 キャッシュミス率

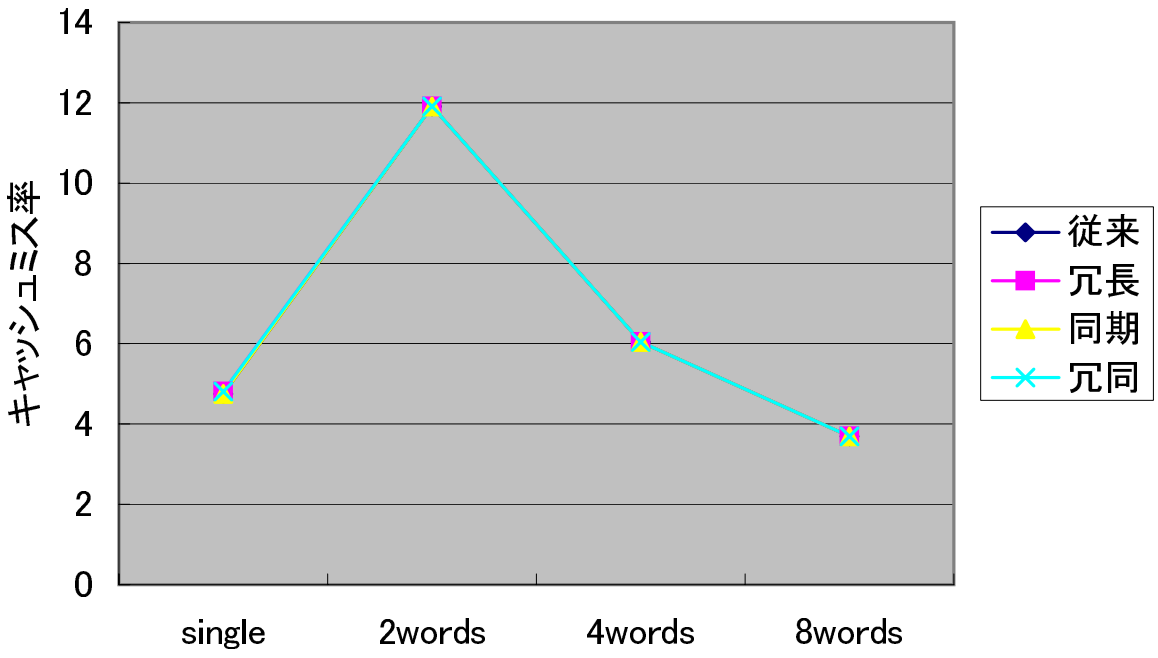


図 5.35: Livermore Kernel-11 キャッシュミス率

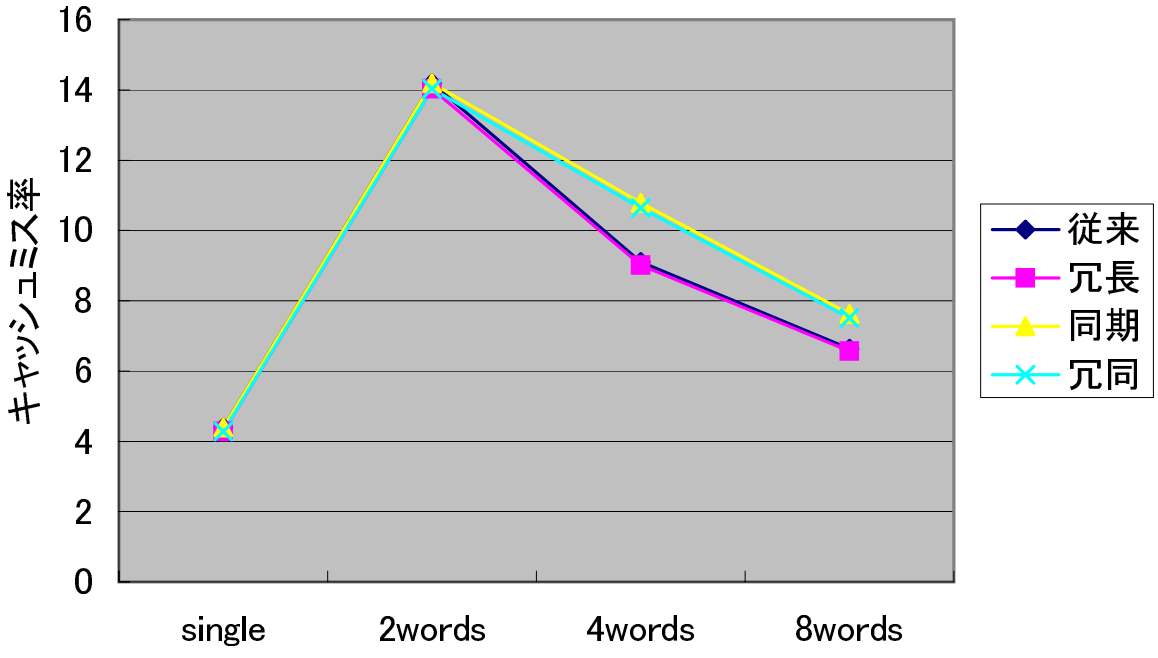


図 5.36: Livermore Kernel-12 キャッシュミス率

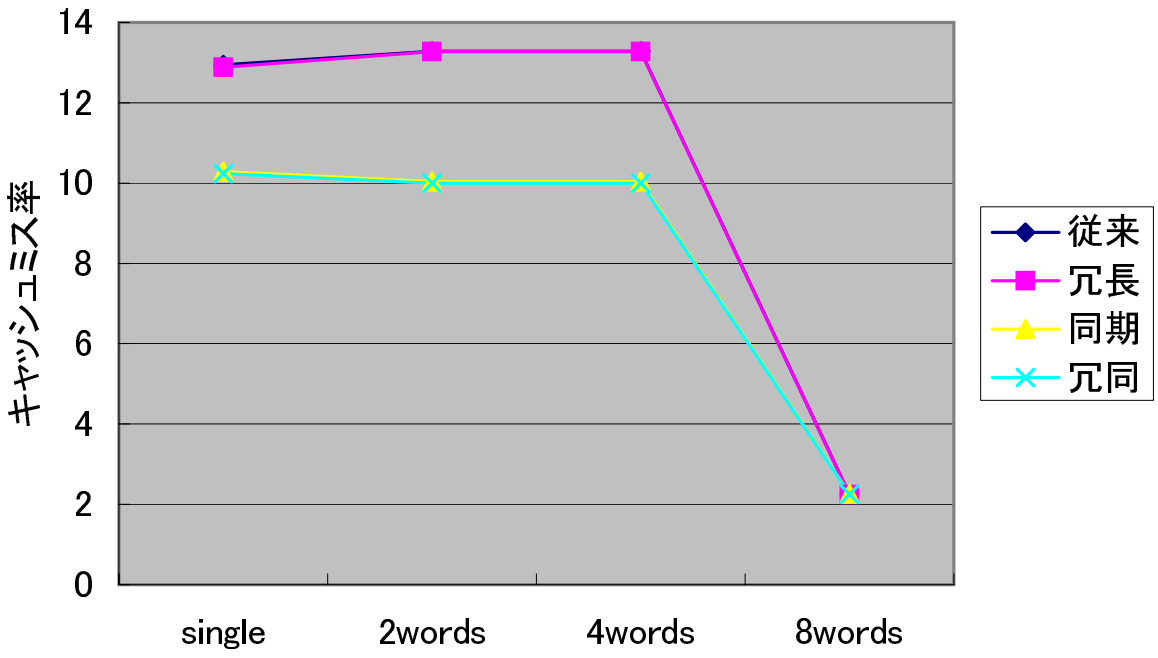


図 5.37: Livermore Kernel-13 キャッシュミス率

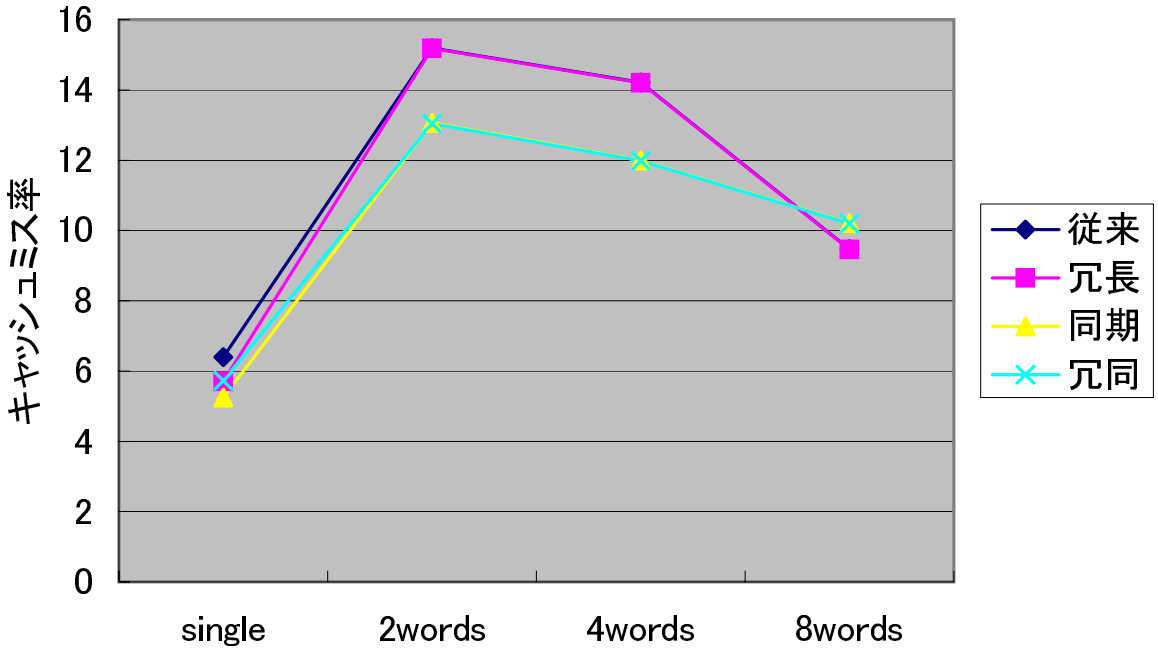


図 5.38: Livermore Kernel-14 キャッシュミス率

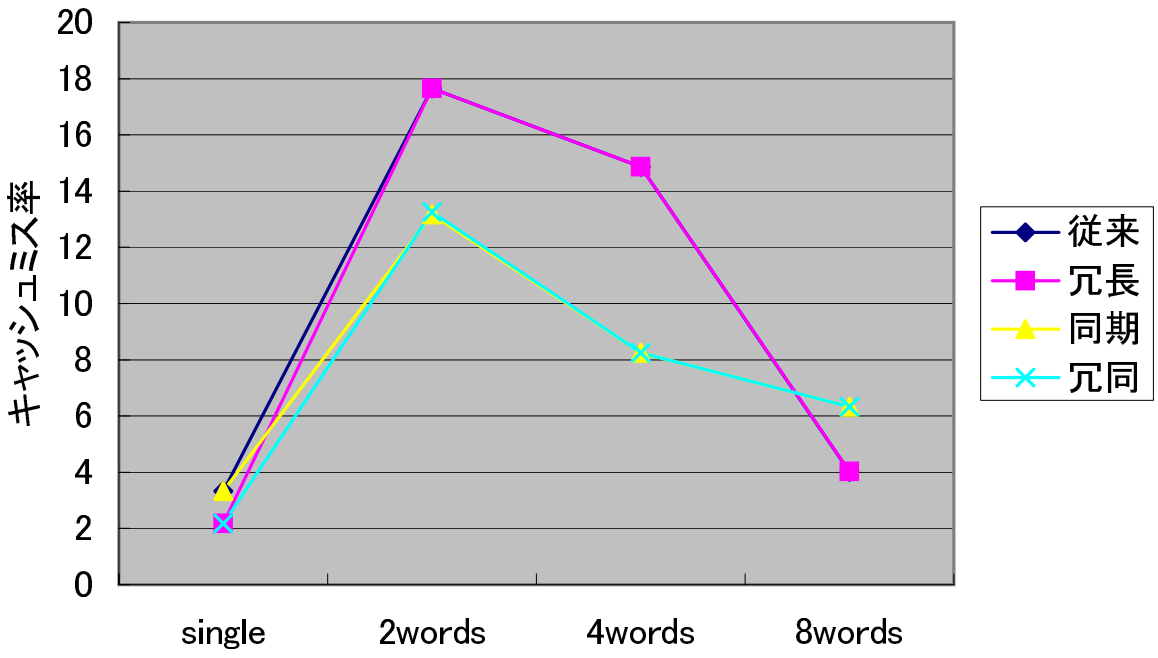


図 5.39: Livermore Kernel-15 キャッシュミス率

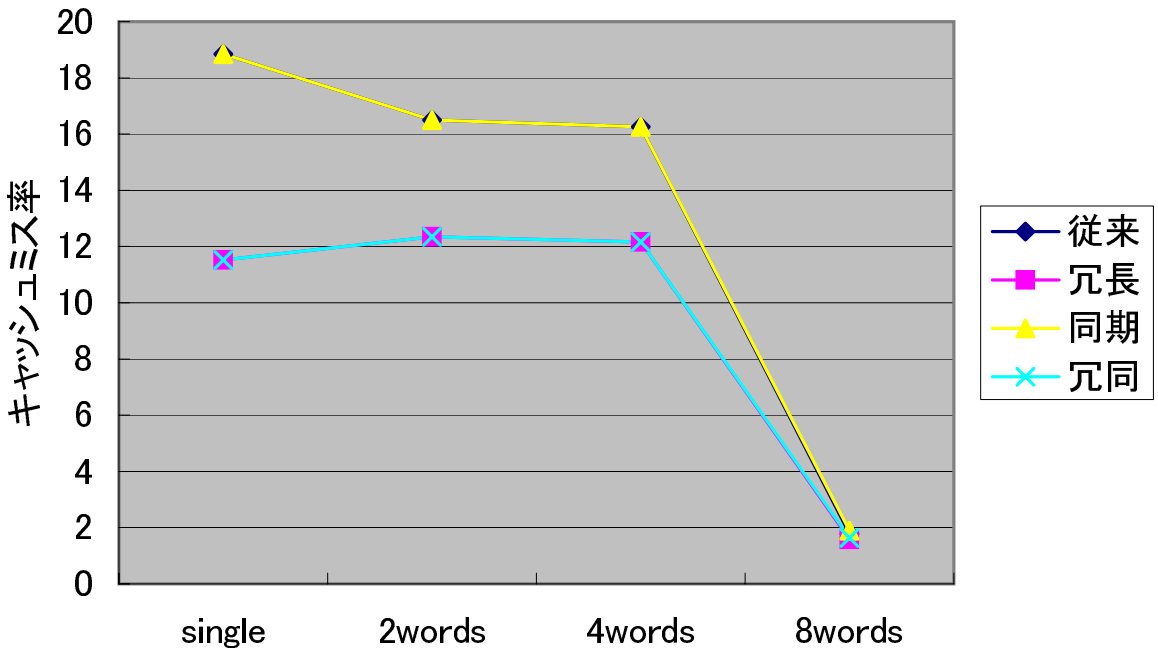


図 5.40: Livermore Kernel-16 キャッシュミス率

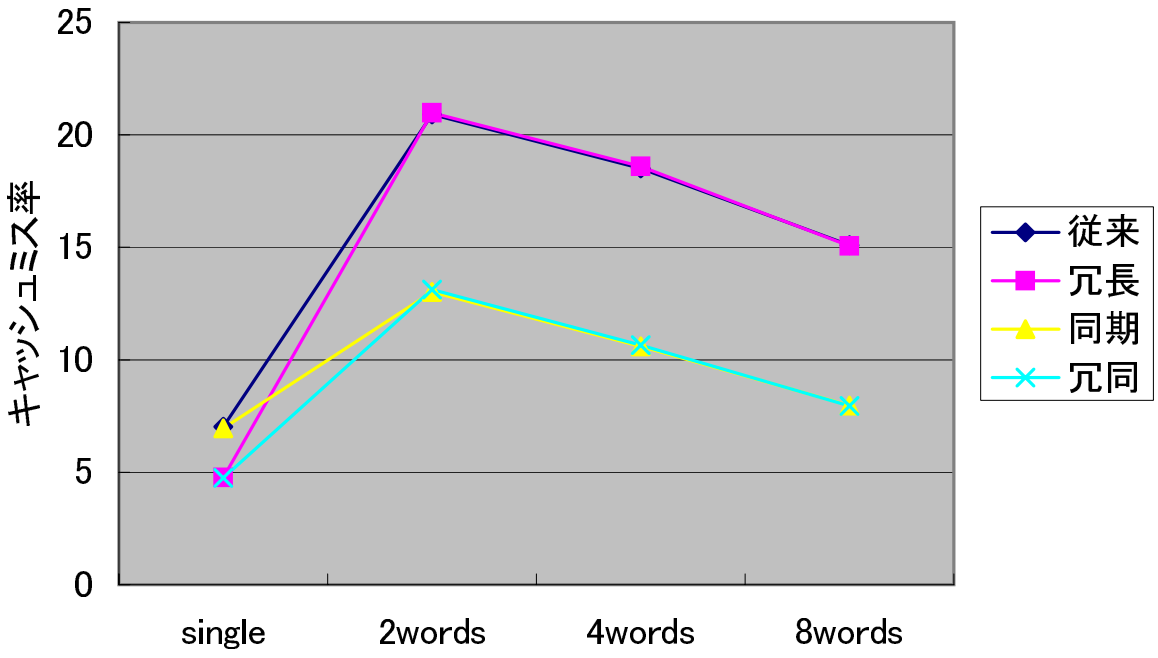


図 5.41: Livermore Kernel-17 キャッシュミス率

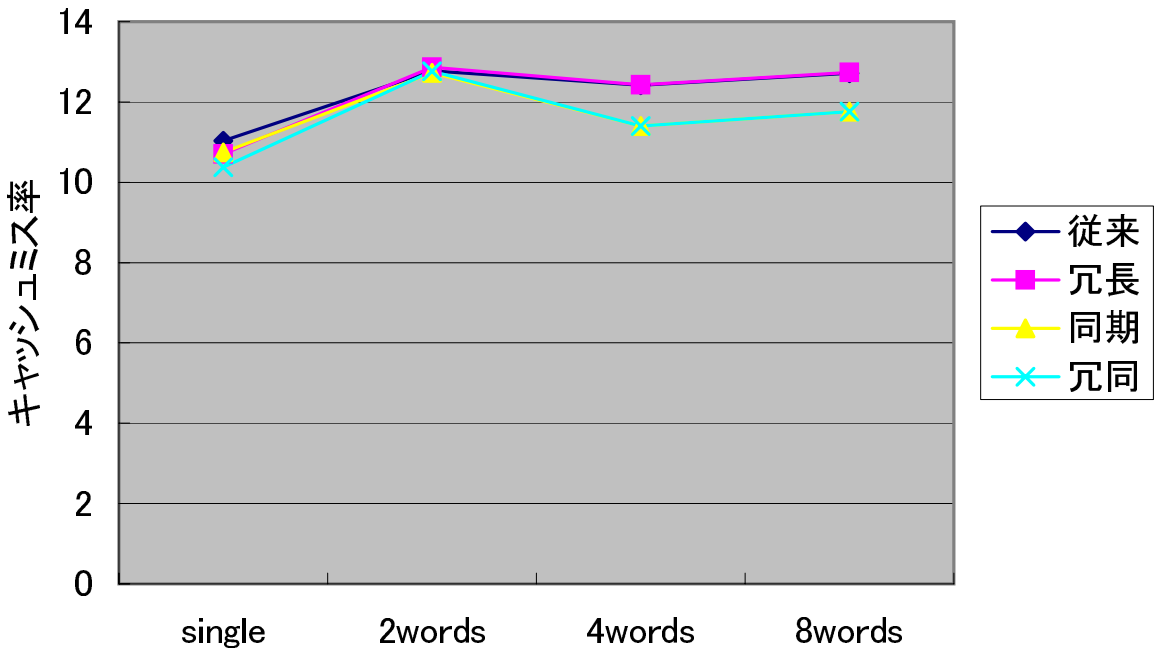


図 5.42: Livermore Kernel-18 キャッシュミス率

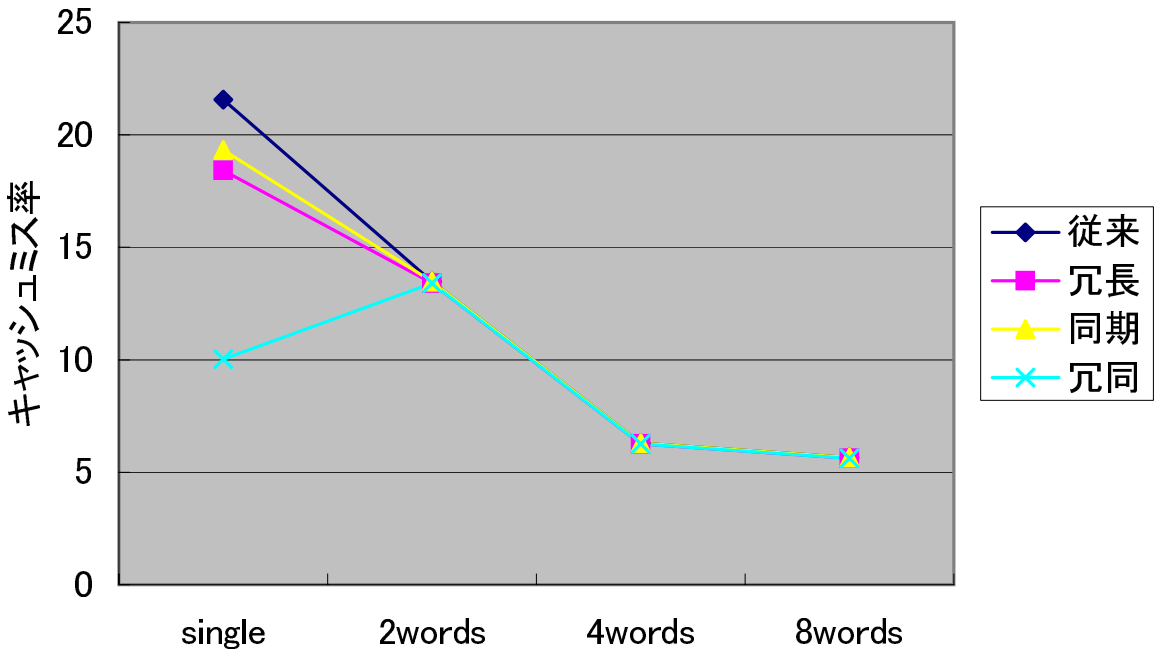


図 5.43: Livermore Kernel-19 キャッシュミス率

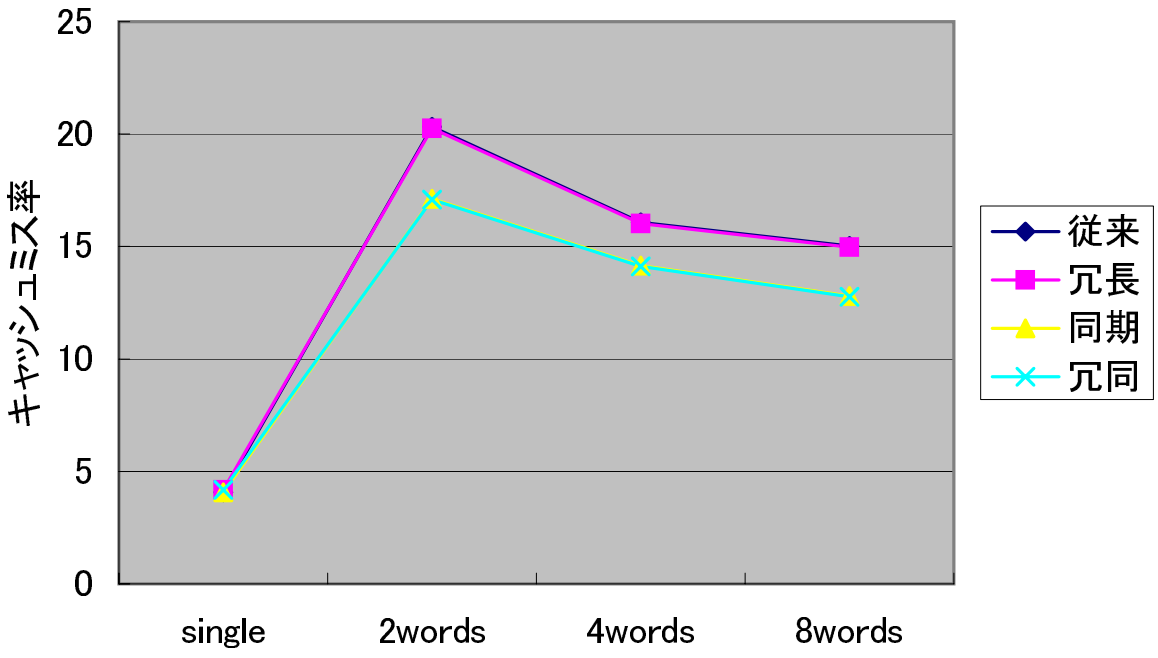


図 5.44: Livermore Kernel-20 キャッシュミス率

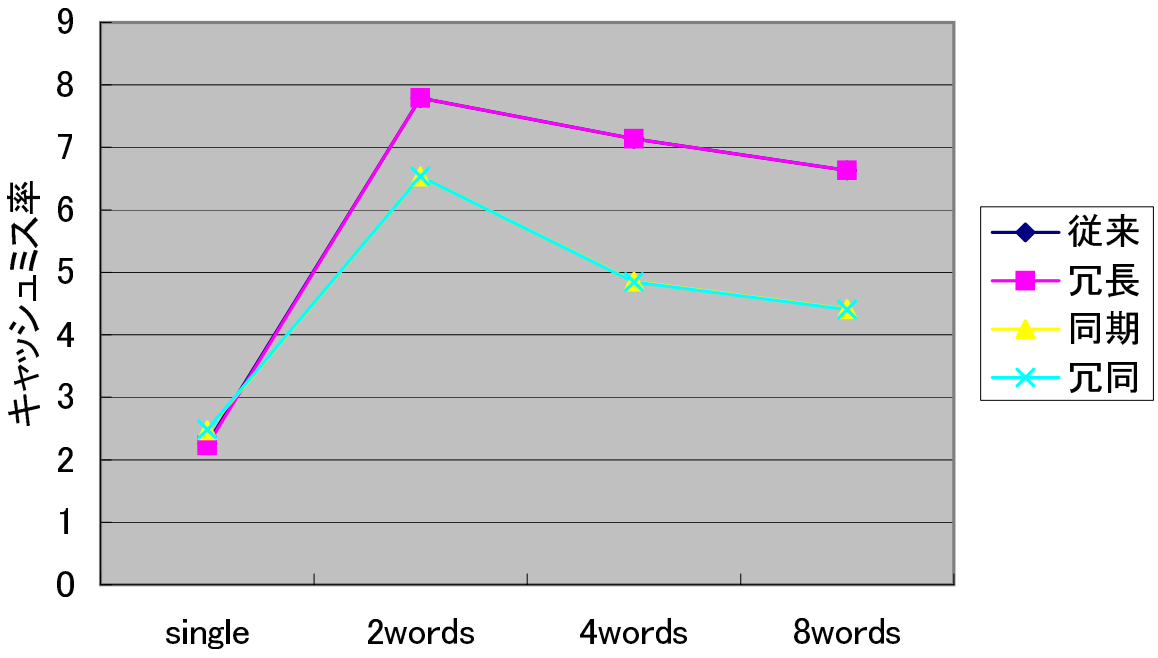


図 5.45: Livermore Kernel-21 キャッシュミス率

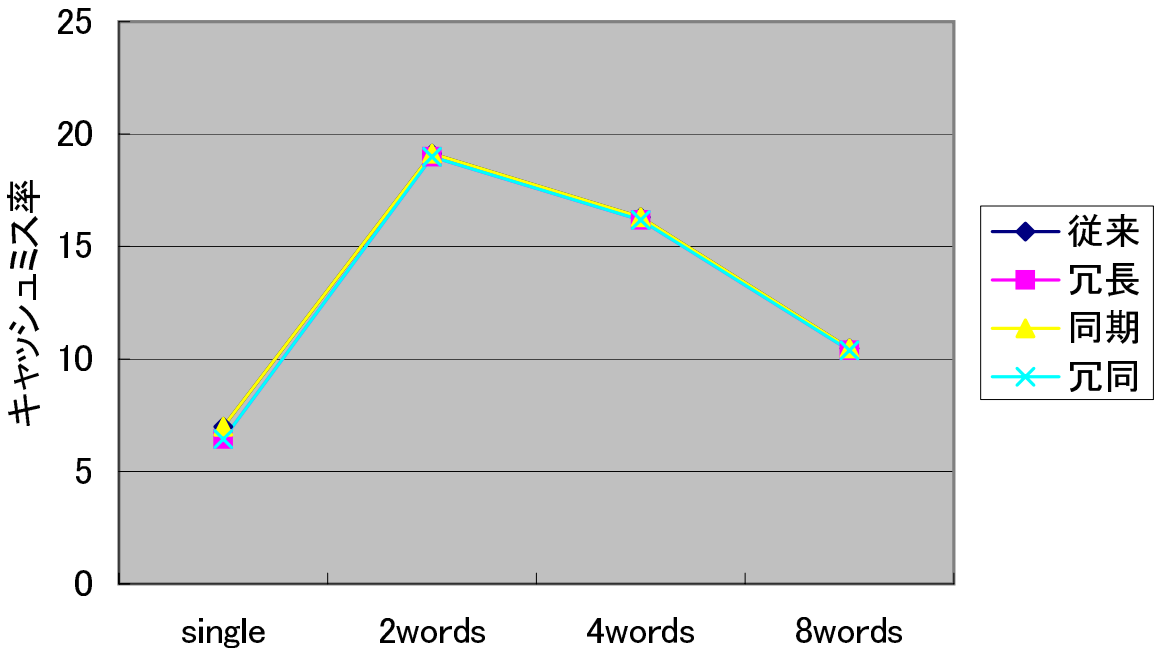


図 5.46: Livermore Kernel-22 キャッシュミス率

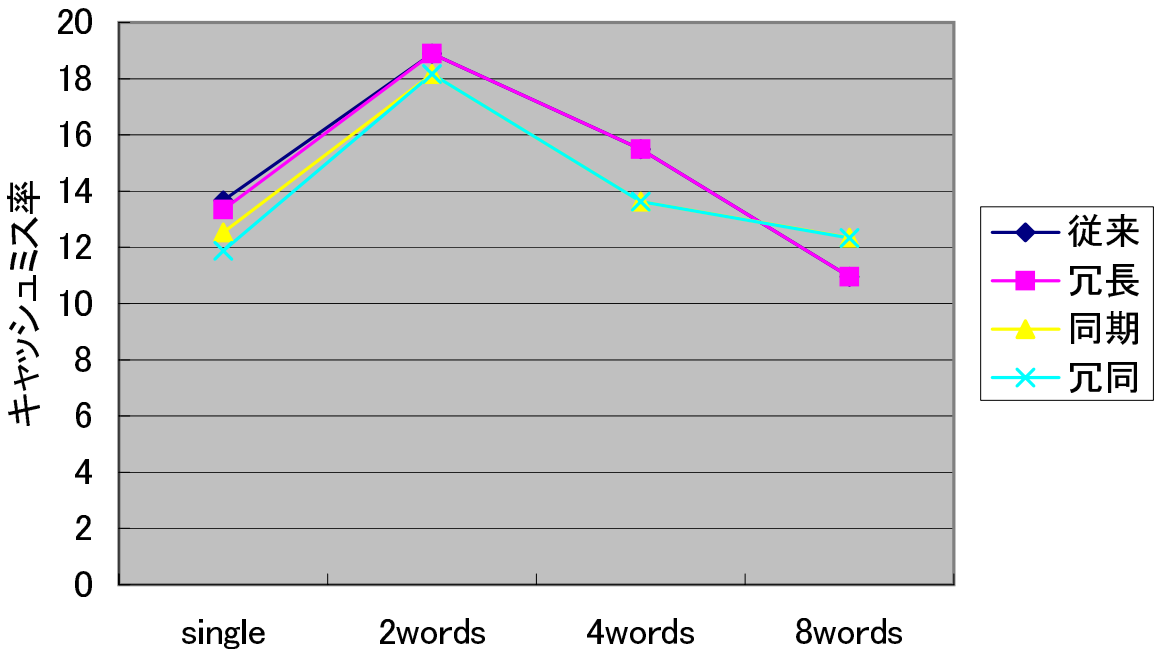


図 5.47: Livermore Kernel-23 キャッシュミス率

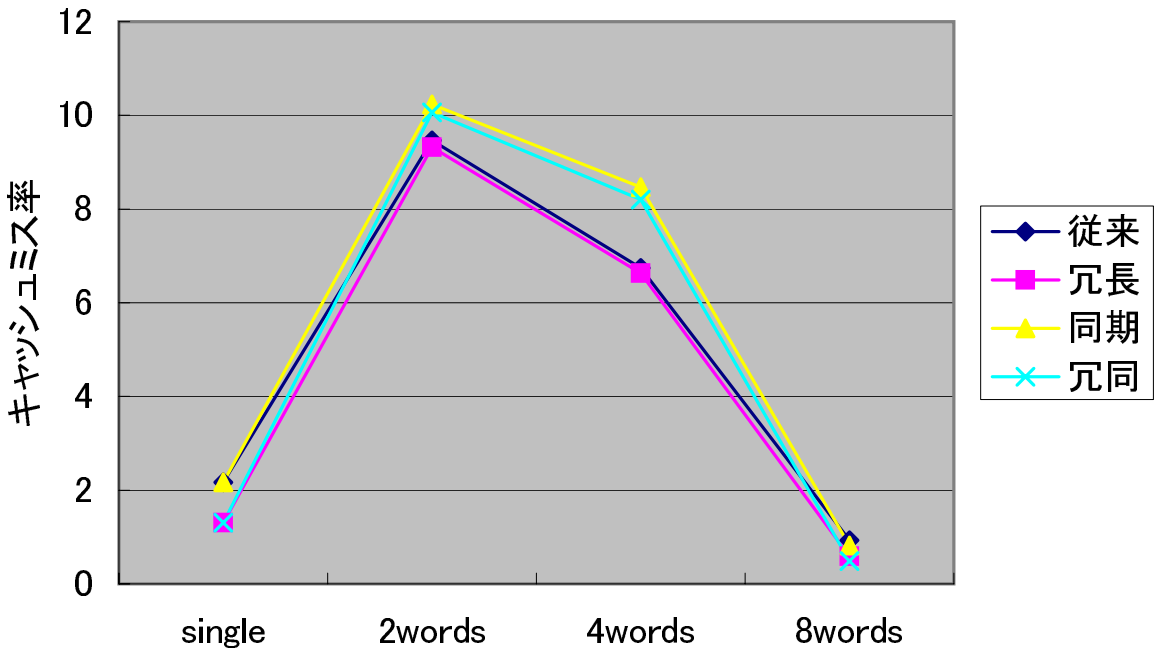


図 5.48: Livermore Kernel-24 キャッシュミス率

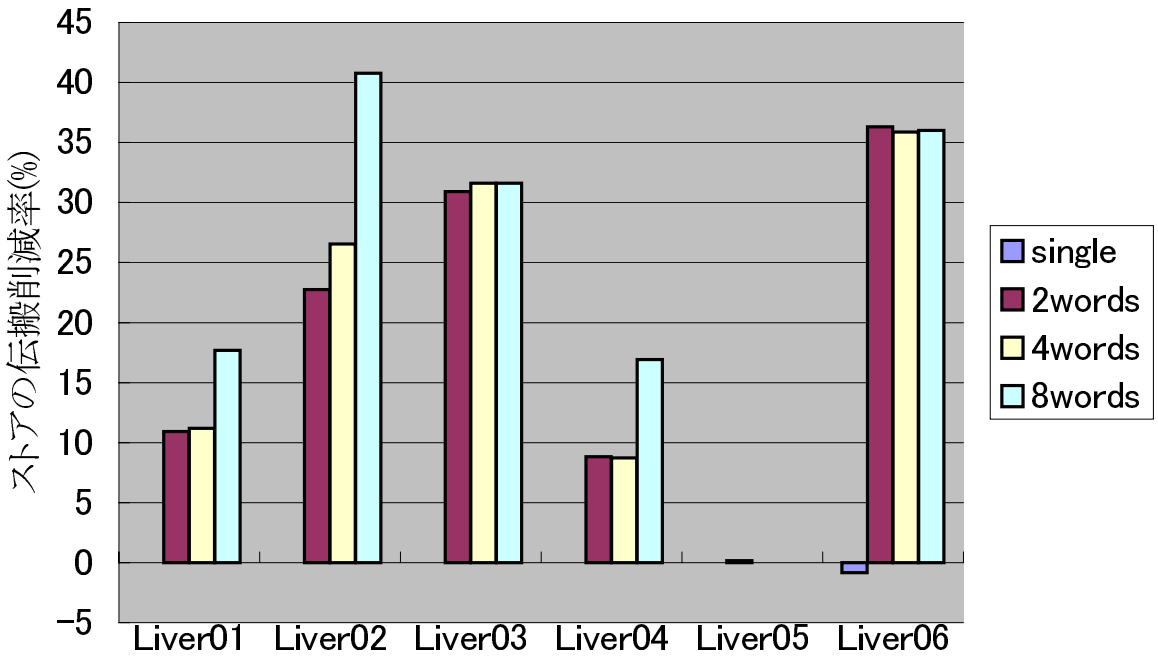


図 5.49: ストアの伝搬削減率 -同期-（1）

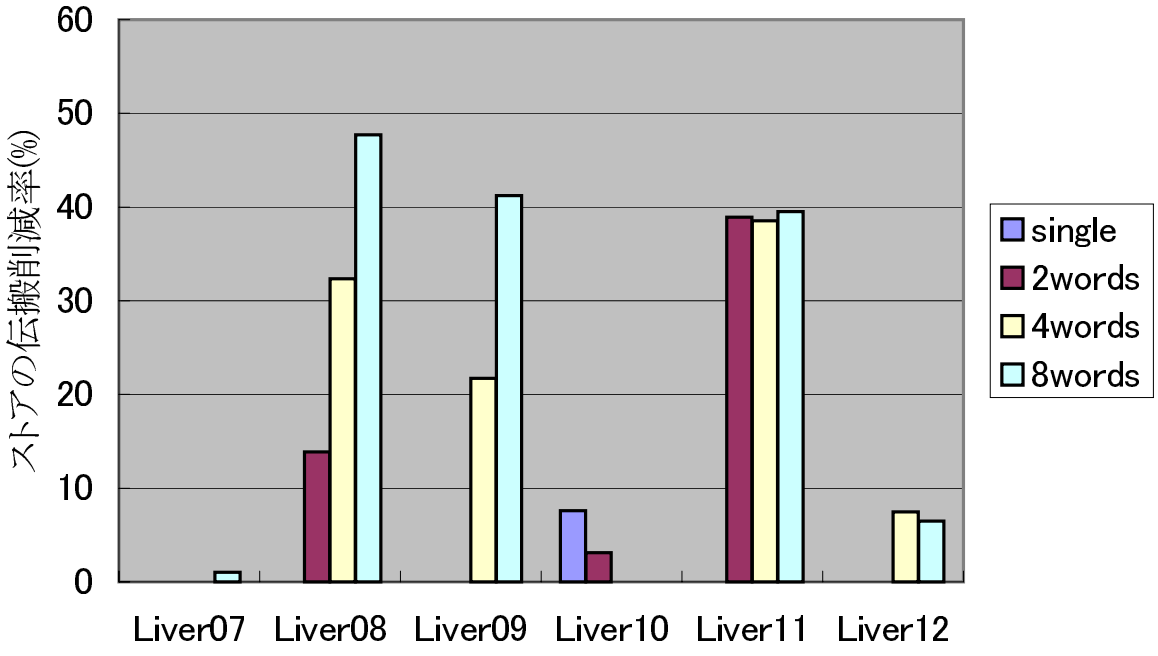


図 5.50: ストアの伝搬削減率 -同期-（2）

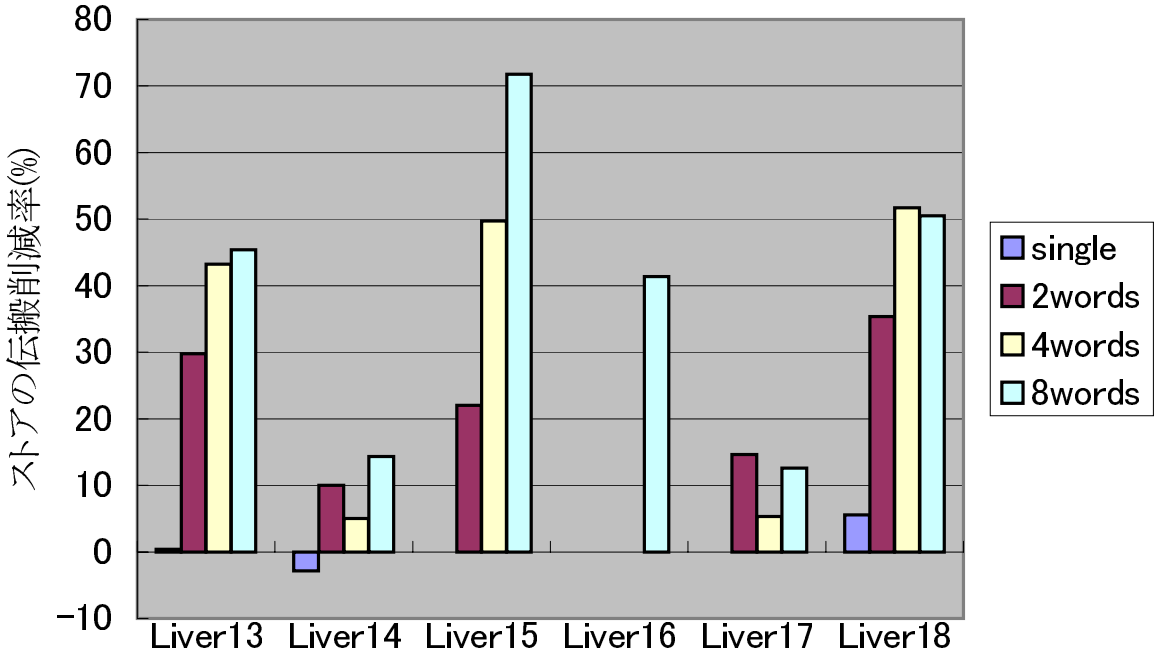


図 5.51: ストアの伝搬削減率 -同期-（3）

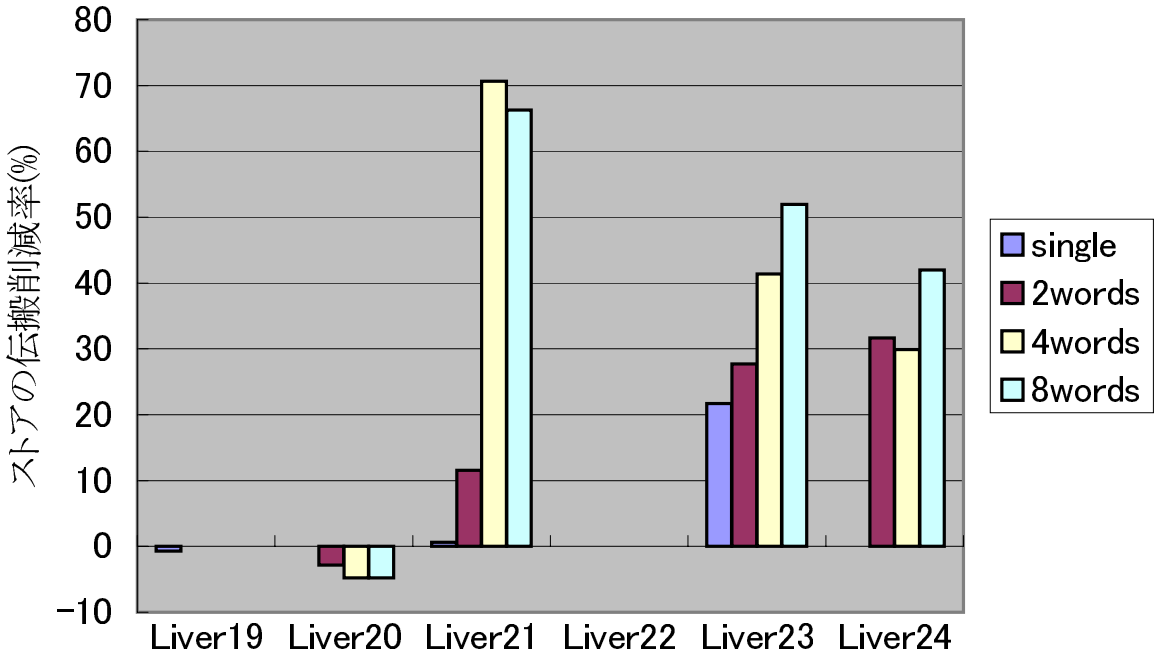


図 5.52: ストアの伝搬削減率 -同期 -（4）

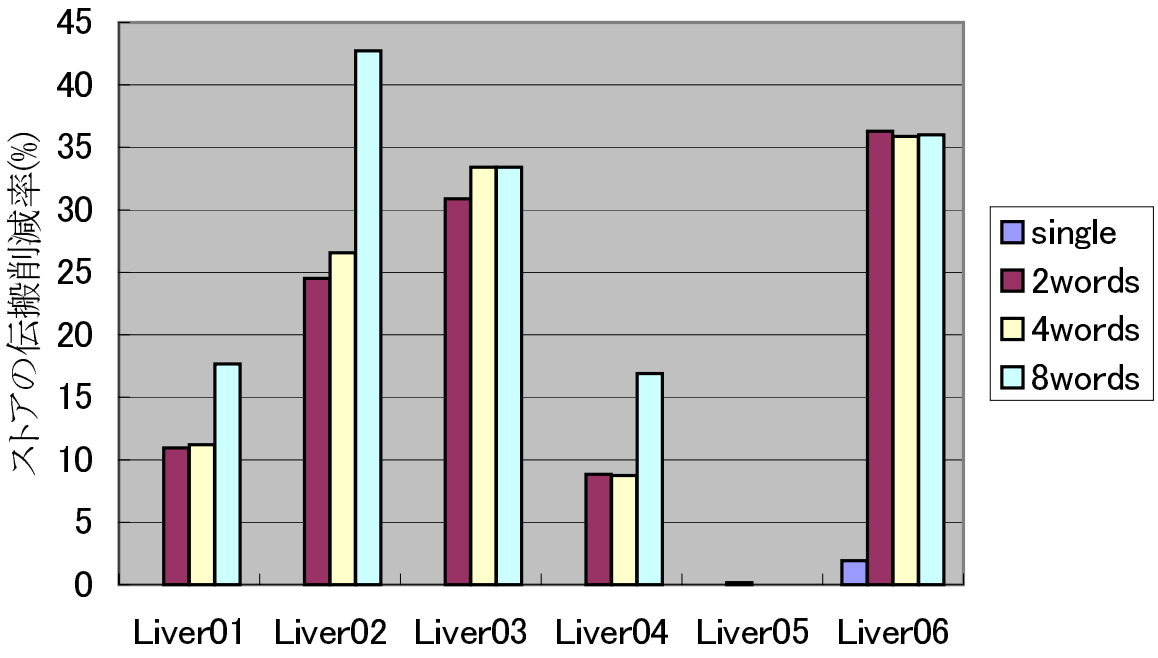


図 5.53: ストアの伝搬削減率 -同期 & 冗長- (1)

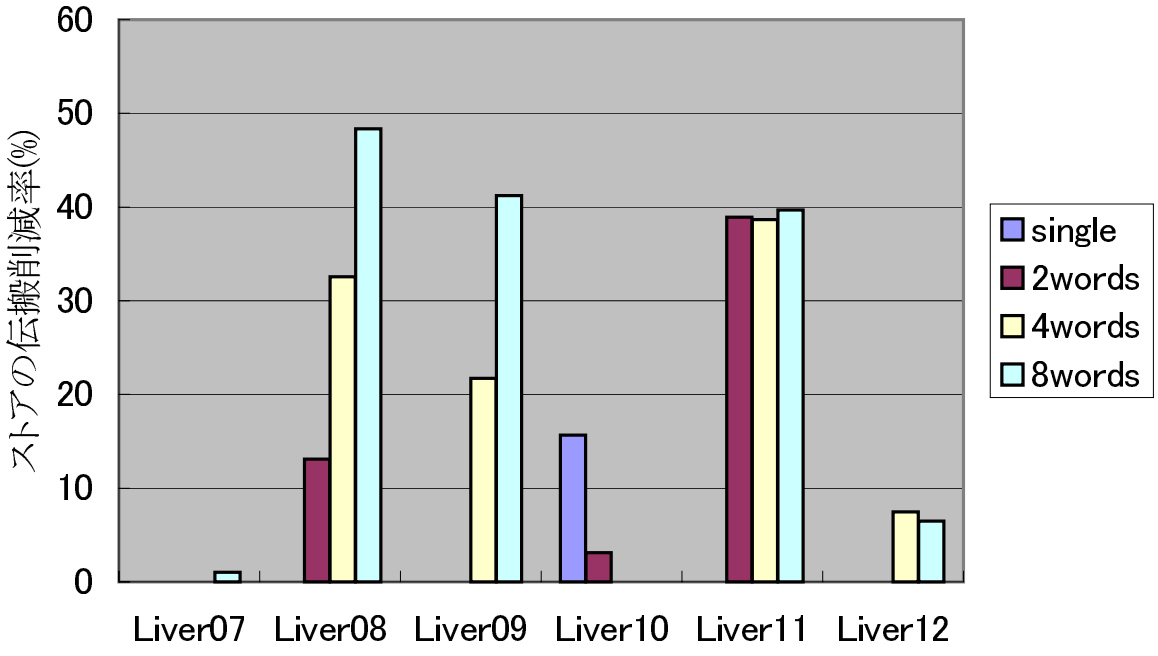


図 5.54: ストアの伝搬削減率 -同期 & 冗長- (2)

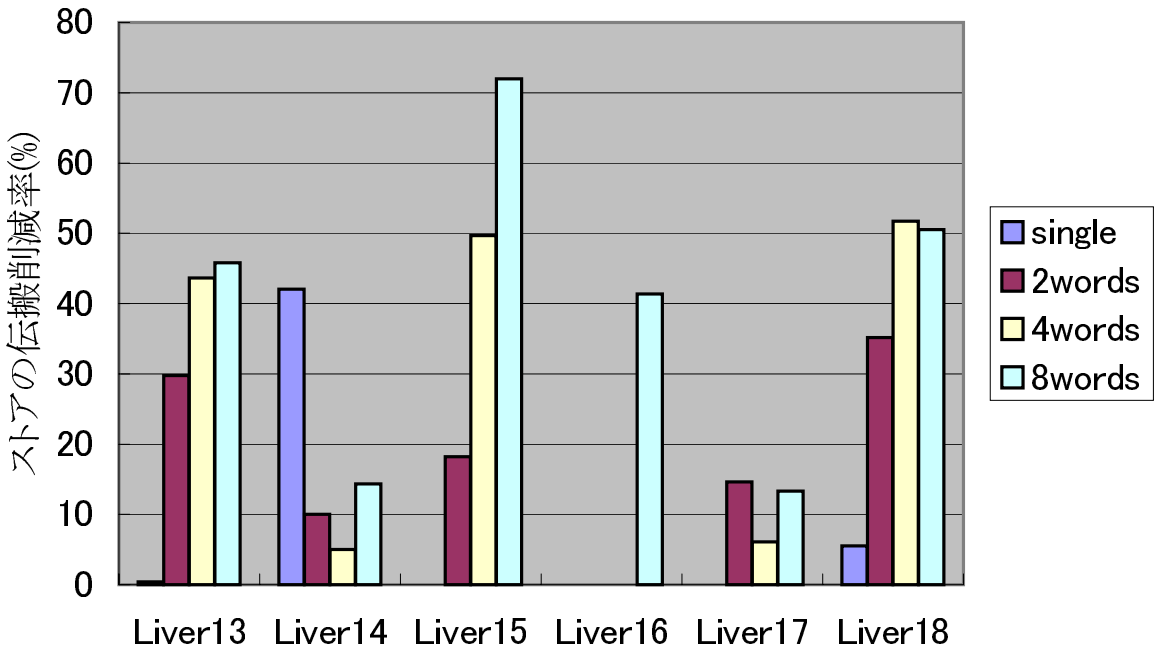


図 5.55: ストアの伝搬削減率 -同期 & 冗長- (3)

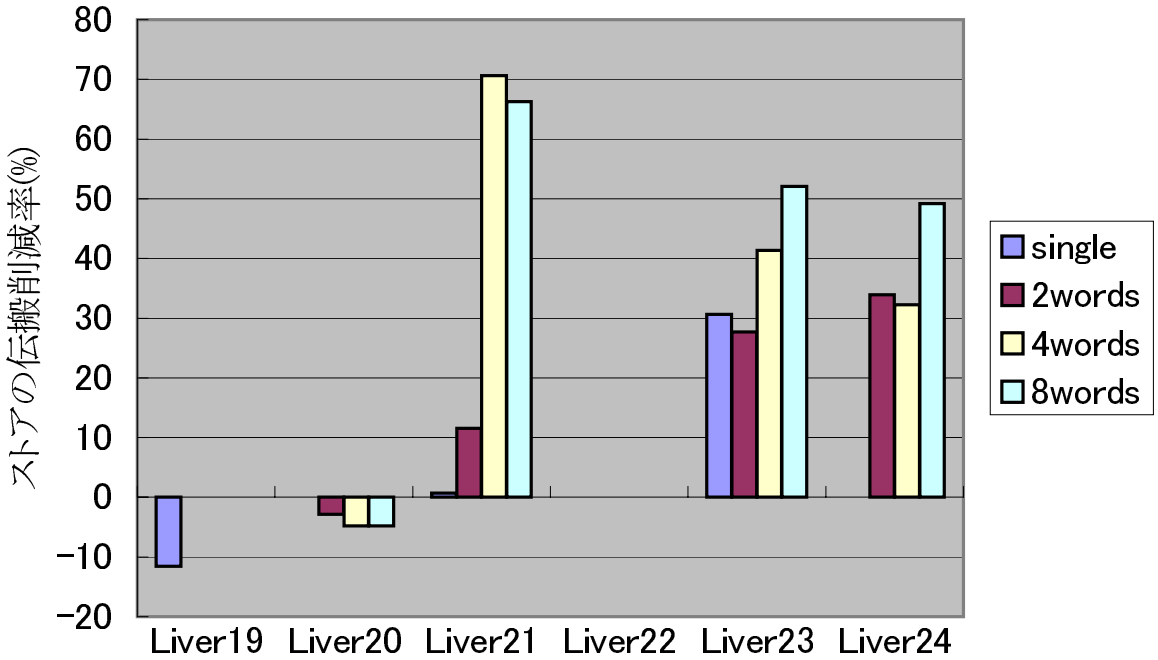


図 5.56: ストアの伝搬削減率 -同期 & 冗長- (4)

表 5.7: Squash 回数と同期回数（1）

		直接 Squash 回数				間接 Squash 回数				同期回数	
bench	W	従来	冗長	同期	冗同	従来	冗長	同期	冗同	同期	冗同
Liver01	1	0	0	0	0	0	0	0	0	0	0
	2	680	684	478	482	504	500	361	357	0	0
	4	680	684	522	526	504	500	389	385	0	0
	8	680	684	456	460	504	500	347	343	0	0
Liver02	1	0	0	0	0	0	0	0	0	0	0
	2	38	118	167	169	74	74	167	167	71	211
	4	509	513	300	304	513	509	326	322	86	250
	8	558	562	353	358	565	561	392	388	0	0
Liver03	1	0	0	0	0	0	0	0	0	0	0
	2	624	628	504	508	600	596	508	504	0	0
	4	596	600	443	447	564	560	379	375	0	0
	8	604	608	443	447	560	556	379	375	0	0
Liver04	1	38	54	38	54	37	29	37	29	35	98
	2	419	423	425	429	540	536	475	471	140	412
	4	419	423	444	448	540	536	489	485	174	510
	8	673	677	565	569	770	766	701	697	0	0
Liver05	1	0	0	0	0	0	0	0	0	0	0
	2	531	535	525	529	457	453	449	445	0	0
	4	549	553	549	553	465	461	465	461	0	0
	8	598	602	598	602	457	453	457	453	0	0
Liver06	1	44	44	44	48	11	10	12	12	10	56
	2	2597	2601	1898	1918	1992	1988	1558	1554	0	0
	4	2592	2596	2597	2601	1999	1995	2027	2023	0	0
	8	2589	2593	2614	2618	1996	1992	2027	2023	0	0
Liver07	1	0	0	0	0	0	0	0	0	0	0
	2	680	684	680	684	504	500	504	500	0	0
	4	680	684	680	684	504	500	504	500	0	0
	8	680	684	669	673	504	500	494	490	0	0
Liver08	1	0	0	0	0	0	0	0	0	0	0
	2	110	104	259	265	120	120	191	191	199	341
	4	262	262	343	343	218	218	200	200	219	357
	8	811	813	297	299	388	386	238	236	2	6

表 5.8: Squash 回数と同期回数（2）

		直接 Squash 回数				間接 Squash 回数				同期回数	
bench	W	従来	冗長	同期	冗同	従来	冗長	同期	冗同	同期	冗同
Liver09	1	0	0	0	0	0	0	0	0	0	0
	2	680	684	680	84	504	500	504	500	0	0
	4	680	684	456	460	504	500	347	343	0	0
	8	765	769	305	321	557	553	331	331	0	0
Liver10	1	16	22	20	28	13	8	12	6	12	16
	2	2543	2547	2551	2555	2646	2642	2642	2638	385	1133
	4	2433	2437	2441	2445	2236	2232	2232	2228	35	103
	8	4842	4846	4842	4846	4775	4771	4775	4771	0	0
Liver11	1	0	0	0	0	0	0	0	0	0	0
	2	171	171	189	189	333	333	331	331	161	481
	4	171	171	236	235	333	333	325	324	0	0
	8	171	171	225	226	333	333	311	310	0	0
Liver12	1	0	0	0	0	0	0	0	0	0	0
	2	511	515	511	515	504	500	504	500	0	0
	4	520	524	445	449	508	504	360	356	0	0
	8	533	537	493	497	512	508	416	412	0	0
Liver13	1	630	662	550	550	429	429	320	320	663	1959
	2	2951	2955	1564	1564	2662	2658	1861	1861	913	2711
	4	5404	5404	2548	2548	4083	4083	2731	2731	339	1021
	8	5857	5861	3014	3018	4717	4713	3233	3229	301	885
Liver14	1	38	52	38	20	37	30	37	17	347	719
	2	2025	2029	1767	1771	1582	1578	1338	1334	1347	3963
	4	1977	1981	1803	1807	1602	1598	1445	1441	1347	3963
	8	3071	3075	1835	1839	2116	2112	1420	1416	954	2730
Liver15	1	0	0	0	0	0	0	0	0	0	0
	2	5946	5946	4639	4818	5298	5298	4611	4635	9227	25221
	4	11007	11011	6388	6400	7572	7568	4920	4916	6036	17476
	8	16855	16859	5321	5325	9431	9427	4782	4778	378	1083
Liver16	1	0	0	0	0	0	0	0	0	0	0
	2	23	27	23	27	37	33	37	33	0	0
	4	23	27	23	27	37	33	37	33	0	0
	8	50	54	26	30	70	66	40	36	0	0

表 5.9: Squash 回数と同期回数（3）

		直接 Squash 回数				間接 Squash 回数				同期回数	
bench	W	従来	冗長	同期	冗同	従来	冗長	同期	冗同	同期	冗同
Liver17	1	16	28	16	28	8	3	8	3	12	14
	2	443	447	543	547	626	622	751	747	1172	3448
	4	475	479	633	637	746	742	822	818	560	1648
	8	571	575	682	686	932	928	905	901	595	1751
Liver18	1	332	332	320	320	229	229	214	214	510	1506
	2	882	878	2189	2185	1104	1108	1277	1281	2708	7936
	4	961	929	1500	1500	1321	1321	1211	1211	1657	4873
	8	959	927	1623	1623	1407	1407	1267	1267	1574	4630
Liver19	1	752	766	758	344	735	728	733	329	366	1032
	2	1492	1496	1492	1496	1416	1412	1416	1412	0	0
	4	1620	1624	1620	1624	1335	1331	1335	1331	0	0
	8	1620	1624	1620	1624	1335	1331	1335	1331	0	0
Liver20	1	0	0	0	0	0	0	0	0	0	0
	2	896	900	1125	1129	1083	1079	1034	1030	521	1529
	4	934	938	1176	1180	1087	1083	1051	1047	537	1573
	8	923	927	1152	1156	1049	1045	1013	1009	502	1470
Liver21	1	268	268	408	440	244	244	227	227	409	1341
	2	8236	8240	22404	22408	10151	10147	16427	16423	15287	44683
	4	8892	8896	6430	6434	10591	10587	11170	11166	9804	28810
	8	10128	10132	7354	7358	11371	11367	11058	11054	8712	25534
Liver22	1	0	0	0	0	0	0	0	0	0	0
	2	683	687	683	687	507	503	507	503	0	0
	4	683	687	683	687	507	503	507	503	0	0
	8	683	687	683	687	507	503	507	503	0	0
Liver23	1	272	296	136	104	74	74	39	31	112	335
	2	3379	3383	3090	3094	2285	2281	2285	2281	17	53
	4	4731	4735	5058	5062	3800	3796	3786	3782	1575	4635
	8	6170	6174	4084	4088	4396	4392	3119	3115	0	0
Liver24	1	0	0	0	0	0	0	0	0	0	0
	2	1477	1489	1070	1082	1385	1373	860	848	0	0
	4	1372	1384	1096	1108	1278	1266	936	924	0	0
	8	1274	1288	631	676	1230	1218	704	703	0	0

第6章　関連研究

本章では，スレッドレベル投機実行に関する関連研究を紹介する．特にメモリ投機を支援する機構に関する研究を 2 つ紹介する．

メモリ投機を支援する機構にはメモリ依存違反の検出，投機データの保持などが必要とする．ARB[3] のような投機データのアドレスをバッファリングする方法と，SVC[2] のようなそのメモリ投機を個々が持つキャッシュに記録しておく方法がある．

- ARB(Address Resolution Buffer)

  全てのプロセッサの投機的なアクセスをメモリアドレス毎に ARB と呼ばれるバッファに格納し，依存違反の検出を行う．

  投機ストアは，すべて ARB 中に格納される．また，投機ロードは，ARB を参照し，投機データをロードする．もし ARB にデータが無い場合，メモリからデータをロードする．その際，ARB 中にそのデータを格納する．

  これにより，データの正しい順序関係の保持，依存違反の検出を行う．しかし，投機データをロードする際，全てのプロセッサのエントリをたどり検索するため，プロセッサ数が増加した場合，アクセス要求を処理しきれなくなる問題がある．

- SVC(Speculative Versioning Cache)

  各プロセッサが持つキャッシュに投機データを格納し，キャッシュライン毎に依存関係の情報を保持することで，依存違反を検出する．投機ロードする場合，そのキャッシュラインにロードした先のプロセッサ番号（pointer）を保持することで，各スレッドが持つデータのバージョン（投機の程度）を表現する．

  各投機データの依存関係は VOL（Version Ordering List）によりプロセッサ毎にどの程度の投機スレッドを実行しているかを管理し，VCL（Version Control Logic）によって制御する．しかし，依存関係の管理を VCL が集中的に行うため，VCL の処理能力の問題などがある．

また，キャッシュシステムを拡張することでメモリ投機を支援する研究は，スケーラブルに行う研究 [4, 16] などが行われている．

第7章　結論

7.1　まとめ

本研究では，まずスレッドレベル投機実行の有効性について述べた．次に，スレッドレベル投機実行の基本的な原理を述べ，スレッド間の依存関係である制御依存，データ依存について述べた．そして，メモリ投機依存違反検出機能を備えたキャッシュ機構の基本構成，動作を述べ，スレッドレベル投機実行を支援するキャッシュ機構の拡張として，データ依存同期型キャッシュ，冗長キャッシュを用いた効率化手法を提案し，評価を行った．最後に，シミュレーションによる評価と提案手法の有効性を示し，それに対する考察を述べた．

本研究で述べたスレッドレベル投機実行が，将来の半導体技術で製造されるマルチプロセッサにおいて，最適な技術になることを期待する．

7.2　今後の課題

今後の課題として以下の点を挙げる．

(1) シミュレータの高精度化

(a) プロセッシングユニットの検討
(b) キャッシュラインの検討

(2) 大規模ベンチマークプログラムによる評価
(3) ライブラリ関数

（1）-（a）では，今回作成したシミュレータは，1命令1クロックサイクルで実行するものであった．本来，スレッドレベル投機実行を行うアーキテクチャでは，各プロセッシングユニットはスーパースカラプロセッサのような命令レベルの投機実行も行う．本研究は，提案するキャッシュ機構の性能を評価するものであり，各プロセッシングユニットが生成するデータフローを考慮するものであり，命令トレースベースのシミュレーションでも十分意味のある評価と言えるが，より厳密なシミュレーションを行うには，実際のデータフローに近いものを選ぶべきである．

（1）-（b）では，今回作成したシミュレータは，キャッシュのラインをすべてワード単位で取り扱った．本来，プログラムの最小単位は，バイト単位であり，様々なプログラム

でバイト単位のアクセスが存在する．バイト単位でアクセスするには，キャッシュラインの各制御ビットは，バイト単位ごとに必要となる．しかし，バイト単位で制御ビットを追加すると，キャッシュ１ラインに対してタグ，制御ビットの割合がデータよりも大きくなるので，キャッシュの面積効率が悪化する．今回は，制御ビットをワード単位で管理する中間的な方法を採用した．今後は，以上のことを踏まえ，厳密なシミュレーションを行う必要がある．

（2）では，今回用いたベンチマークプログラムである Livermore Kernel は，様々なループをベンチマークプログラムとして扱ったものである．各プログラムは，使用するデータが少なく，キャッシュの容量を変化させたときのシミュレーションを行うことができなかった．今後は，大規模ベンチマークプログラムを用い，今回提案したキャッシュ機構の検討を行う．

（3）では，今回シミュレーションを行う際，ライブラリ関数を除外して行った．スレッドレベル投機実行は，一般的な逐次プログラムをターゲットとするため，ライブラリ関数中の特権命令の取り扱いについて検討する必要がある．

謝辞

本研究を行うにあたり御指導，御鞭撻を頂いた北陸先端科学技術大学院大学情報科学研究科 田中清史 助教授に深く感謝するとともに，ここに御礼申し上げます.

適切な御意見，御助言を頂きました本学の日比野 靖 教授，井口 寧 助教授に誠に深く感謝致します.

多種多様な疑問や問題に対して親切なアドバイスをして頂いた田中研究室学友の大崎哲弥氏，馬場久氏，山本達也氏，吉兼寛氏，荻野雅氏，その他貴重なご意見，討論を頂いきました田中研究室の皆様をはじめ多くの方々に対して厚く御礼申し上げます．また，研究に行き詰まった折りに心の支えとなってくれた Bump of Chicken に深く感謝致します.

最後に日頃よりあたたかく見守ってくださった両親に深く感謝致します.

参考文献

[1] Kunle Olukotn, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Cang., The Case for a Single Chip Multiprocessor. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.2-11, October 1996.

[2] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi Speculative Versioning Cache, In Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, pp. 195 - 205, June 1998.

[3] Manoj Franklin and Gurindar S. Sohi, Arb: A Hardware Mechanism for Dynamic Reordering of Memory References, IEEE Transactions on Computer, Vol. 45, No. 5,pp.195-205, May 1996.

[4] J.Greggory Steffan, Chrisopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A Scalable Approach to Thread-Level Speculation. In Proceedings of the 27th International Symposium on Computer Architecture, pp.1-24,June 2000.

[5] Haitham Akkary and Michael A.Driscol. A Dynamic Multithreading Processor. In Proceedings of the 31st Annual International Symposium on Microarchitecture, pp.226-236, November 1998.

[6] Gurindar S.Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors In Proceedings of the 25th International Symposium on Computer Architecture, pp.521-532,June 1998.

[7] Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen and Kunle Olukotun. The Stanford Hydra CMP. IEEE MICRO Magazine, March 2000.

[8] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register Renaming and Dynamic Speculation: an Alternative Approach. In Proceedings of the 26th International Symposimu on Microarchitecture, pp.202-213,December 1993.

[9] J.Oplinger, D. Heine, M. Lam, and K. Olukotun, In Search of Speculative Tread-Level Parallelism, Stanford University, Computer Systems Laboratory Technical Report CSL-TR-98-765, July 1998.

[10] J. Gregory Steffan and Todd C. Mowry, The Potentioal for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization, Proceedings of the 4th International Symposium on High-Performance Computer Architecture, February 1998.

[11] C. Brownhill, A. Nicolau, S. Novack, C. Polychronopoulos, Achieving Multi-level Parallelization, Proc. International Symposium on High Performance Computing, LNCS 1336, pp. 183-194,1997.

[12] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelinign With Run-Time Data Dependence Checking and Control Speculation. PACT, October,1996.

[13] Y.Zhang, L. RauchWerger, and J. Torrelas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In Proc. 5th Intl. Symp. on High-Performance Computer Architecture, pp. 135-139, January 1999.

[14] P.Rundberg and P.Stenstrom. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In 4th Workshop on Multithreaded Execution, Architecture and Compilation , December 2000.

[15] L.Hammond, M. Willey, and K.Olukotun. Data Speculation Suport for a Chip Multiprocessor. In 8th Intl. Conf. on Arch, Support for Prog.Lang. and Oper.Systems, pp. 58-69, October 1998.

[16] M.Cintra, J.F.Martinez, and J. Torrellas. Architural Suppout for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In Proc. 27th Annual Intl. Symp. on Computer Architecture, pp. 13-24, june 2000.

[17] F.McMohan, The Livermore Fortran kernels, A computer test of the mumerical performance range, Lawrence Livermore National Laboratory, Livermore, CA, Tech.Rep.UCRL-53745, Dec. 1986.