修 士 論 文

リアルタイムデータアプリケーションに適した再構成可能キャッシュメモリに関する研究

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

高橋 礼彦

2005年3月

修 士 論 文

リアルタイムデータアプリケーションに適した再構成可能キャッシュメモリに関する研究

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

310061 高橋 礼彦

提出年月: 2005 年 2 月

概 要

本研究では，キャッシュメモリを有効に利用するために，動的に分割することが可能な再構成可能キャッシュを利用する．これにより、アプリケーションごとの要求に対し，可変にキャッシュを対応させ，メモリアクセスの性能を向上させる．再構成キャッシュとして以下の３つの機能を実現する．

1. リアルタイムタスクの優先度に基づいたパーティション分割
2. 大規模データを格納する FIFO バッファ
3. プリフェッチしたブロックを格納するプリフェッチバッファ

シミュレーションによりこれらの実行性能の評価を行う．

目 次

第1章　序論

1.1　研究の背景

近年プロセッサはトランジスタの集積度の進歩を背景に，動作周波数の向上，スーパースカラや VLIW に代表される命令レベル並列性の活用などにより高性能化が達成されてきたが，一方でメモリアクセスの性能はプロセッサほど改善されていない．よって，実行性能がメモリアクセスによって制限されることが問題となっている．また，科学，工学，データベース，最近ではメディアプロセスなどの様々なアプリケーションの要求に対応した性能を持つプロセッサが増えてきている．しかし，既存のメモリシステムは汎用性を重視した構造であるため，使用するアプリケーションによってはその性能は十分ではない．

1.2　研究の目的

本研究ではメモリシステムの中でも，アクセス速度の速いキャッシュメモリを有効に利用する．キャッシュメモリを有効に利用する為に動的にキャッシュメモリの構成を変える．つまり，使用するアプリケーションによって異なる要求に対応するために，アプリケーションに対応した専用のキャッシュ構成に変える．これによりキャッシュメモリに専用性を持たせ，アプリケーションが要求するデータをメモリ，ハードディスクではなく，キャッシュメモリの中に格納する．

また，キャッシュメモリに専用性だけでなく，従来の汎用性も持たせる．特定のアプリケーション以外のアクセスには，従来のキャッシュ構成のまま通常のキャッシュアクセスを行えるようにする．

再構成キャッシュメモリにより以下の３つの機能を実現する．

1. リアルタイムタスクの優先度に基づいたパーティション分割
2. 大規模データを格納する FIFO バッファ
3. プリフェッチしたブロックを格納するプリフェッチバッファ

リアルタイムアプリケーションを実行させる場合，多くのタスクに分けて処理を行うが，効率の良い実行のためにはマルチスレッド型のプロセッサを使用することが有効である[1]．この際，優先度の高いタスクに多くのキャッシュ領域を割り当て，高い処理速度を

[1]本研究で用いるタスクはスレッドと同じ意味である．

保証する．

メディアプロセッシングのような大規模な連続したデータを必要とするアプリケーションに対しては，FIFO パーティションを用意し，SDT などの DMA 機構によって転送される連続データを受け取る．これにより高速化を図る．

様々なアプリケーションでプリフェッチ機能が有効であるが，プリフェッチされたデータが容量性ミスにより実際に使用される前にリプレースされる，あるいはプリフェッチデータが頻繁にアクセスされるデータを追い出す問題がある．これに対し，再構成された一つのパーティションをプリフェッチバッファとして使用することにより解決する．

1.3 本論文の構成

本論文は全 6 章からなる．

第 2 章では再構成可能キャッシュメモリについて述べる．

第 3 章ではシミュレーション環境について述べる．

第 4 章では提案機構の有効性をシミュレーションにより評価し、考察する．

第 5 章では関連研究を紹介する．

最後に，第 6 章で本論文の結論を述べる．

第2章　再構成可能キャッシュメモリによる３つの機能の提供

2.1　再構成可能キャッシュメモリ

本研究では，キャッシュを動的に複数のパーティションに分割し，性能向上を図る reconfigurable キャッシュを利用する．これにより，アプリケーションごとの要求に対し，可変にキャッシュが対応し，メモリアクセスの性能が向上する．

キャッシュを動的に複数のパーティションに分割し，性能向上を図る reconfigurable キャッシュの研究がある [1]．例えば，一つのパーティションを FIFO バッファとして利用することにより，大規模なデータアクセスに対応させ，データアクセスの性能を向上させる提案がなされている [2]．また，連想キャッシュの一部のウェイをロックすることで，ハードウェアによるリプレースメントを制御し，必要なデータがキャッシュから追い出されるのを防ぐ機構が提案されている [3]．

しかし，これらは再構成の際に，セットアソシアティブ方式のウェイ単位で分割を行うため，パーティションのサイズが固定される問題がある．本研究では reconfigurable キャッシュのパーティション分割の際に，ウェイより細かい単位での分割を可能にする（図 2.1）．これを利用して，様々なアプリケーションに対して適切なサイズのパーティションを提供する．

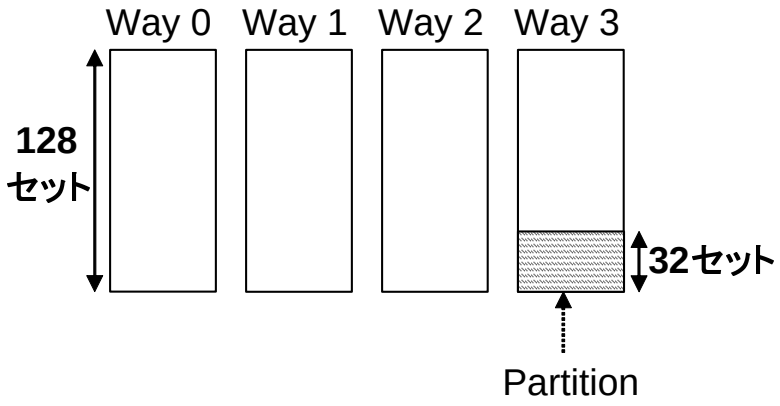

図 2.1: パーティション分割

またキャッシュを動的にパーティション分割するために，専用の設定レジスタを設ける．この設定レジスタへの書き込みはI/O空間へのストア命令を使用する[1]．専用の設定レジスタの値によって，キャッシュは再構成を行う．

2.1.1 タグの拡張

ウェイより細かい単位でキャッシュの再構成を行った際に，パーティションにメモリアドレス（タグ，インデックス）アクセスを行う場合，メモリアドレスのタグの値を拡張させ，インデックスの値を縮小してアクセスを行う．例えば，図2.2のように1ウェイが128セットある4ウェイセット・アソシアティブキャッシュを，32セットのパーティションに再構成を行う．この時パーティションにアクセスを行うには，インデックスが最大で32なのでメモリアドレスのインデックスを2ビット減らし，タグを2ビット増やす．

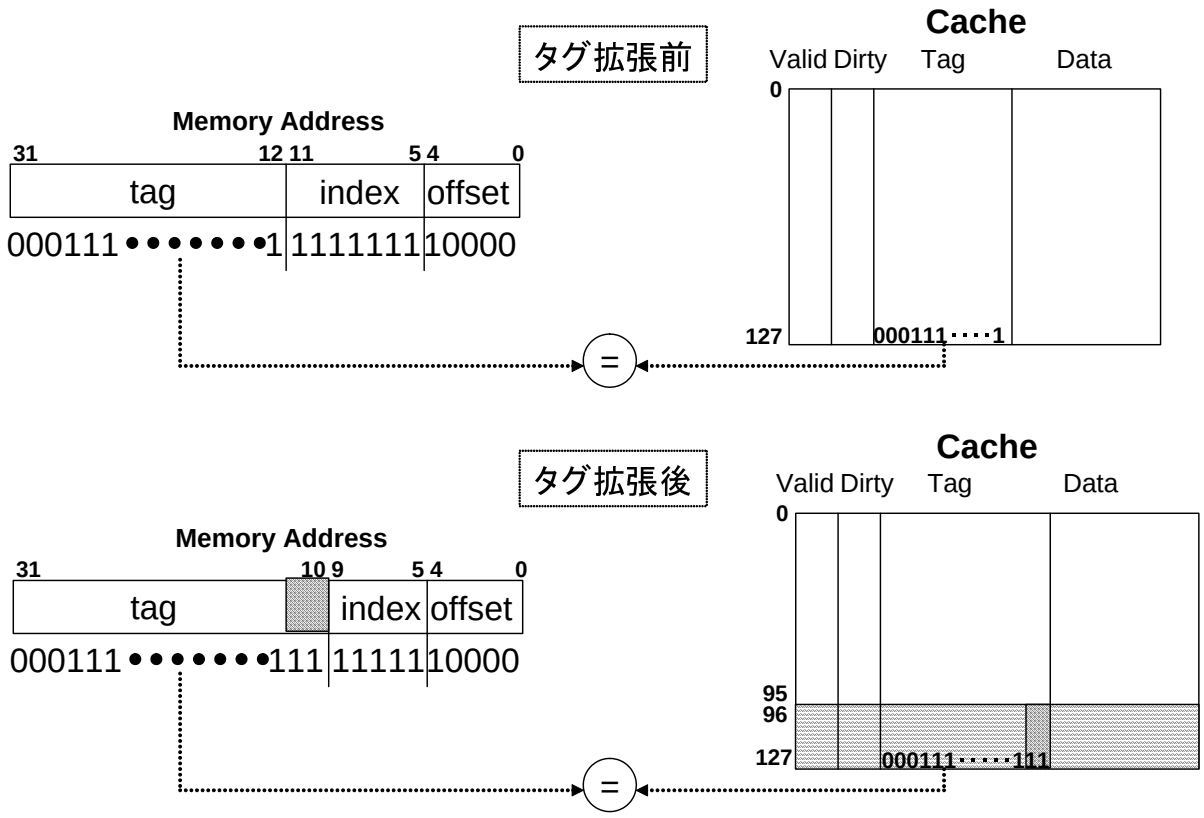

図 2.2: タグ拡張

[1]本研究ではSPARCアーキテクチャ[4, 5]をターゲットとし，ASI=0x18を指定したalternate spaceへのストア命令を使用する．

メモリアドレスのタグを拡張する際，タグメモリに格納されているタグの値も拡張する必要がある．図 2.2 のようにウェイの下の領域を再構成でパーティション分割を行った場合，タグを増やした部分に 1 を代入する．また，同様にウェイの上の領域を使って再構成を行った場合には，タグを増やした部分に 0 を代入する．パーティションのタグ拡張を行うことにより，分割前のメモリアドレスでパーティションにアクセスすることができ，パーティション内データ参照が可能となる．

またタグの拡張を行った際に，拡張したタグを格納できるだけのタグメモリ領域をあらかじめ用意しておく．

このようにタグの拡張を行うことにより，ウェイより細かい単位でキャッシュの再構成を行った場合でもパーティションにメモリアドレス（タグ，インデックス）でアクセスすることができる．

2.1.2 比較器の拡張

ウェイより細かい単位でキャッシュの再構成を行った際に，メモリアドレス，キャッシュ（パーティション）共にタグの拡張を行う．メモリアドレス，パーティションの拡張したタグを比較する為，比較器も拡張を行わなければならない．

比較器は単一ゲート XNOR，AND を組み合わせた論理回路である．タグを拡張しても比較できるようにするには図 2.3 のように XNOR の数を増やす．またタグを拡張しない時には，増やした XNOR の信号をネゲートする．このように比較器を拡張することによっ

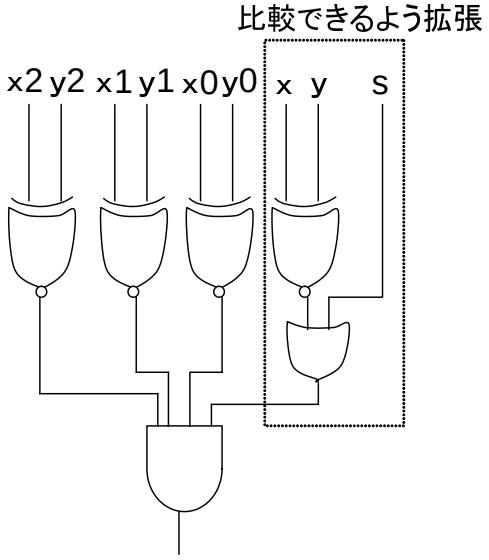

図 2.3: 比較器

て，再構成行った場合でもタグの比較を可能にする．

2.2 リアルタイムタスクの優先度に基づいたパーティション分割

リアルタイムアプリケーションを実行させる場合，多くのタスクに分けて処理を行うが，効率の良い実行のためにはマルチスレッド型のプロセッサを使用することが有効である．このマルチスレッド型のプロセッサを用いる際，タスク毎に固有なキャッシュを用意した方が高性能となる傾向が Tullsen らによって示されている [6]．本研究で用いる再構成キャッシュでは，タスク間でお互いの影響をなくす為に，タスク毎に使うキャッシュ領域を割り当てるパーティション分割を行う．

再構成を行った際には，図 2.4 のようにキャッシュのウェイ単位で一つのパーティションとし，タスク毎にキャッシュ領域を割り当てる．割り当てる方法として，実行中のタスク，パーティション毎にタスク識別子を格納する専用レジスタを用意する．そしてメモリからキャッシュに充填を行う際に，専用レジスタに格納されている実行中のタスクの識別子，パーティション毎の識別子の比較を行い，一致したパーティションに充填を行う．一致するパーティションが二つ以上あった場合には，通常の LRU 法と同じく使用されていない時間が最も長いブロックを置き換える．

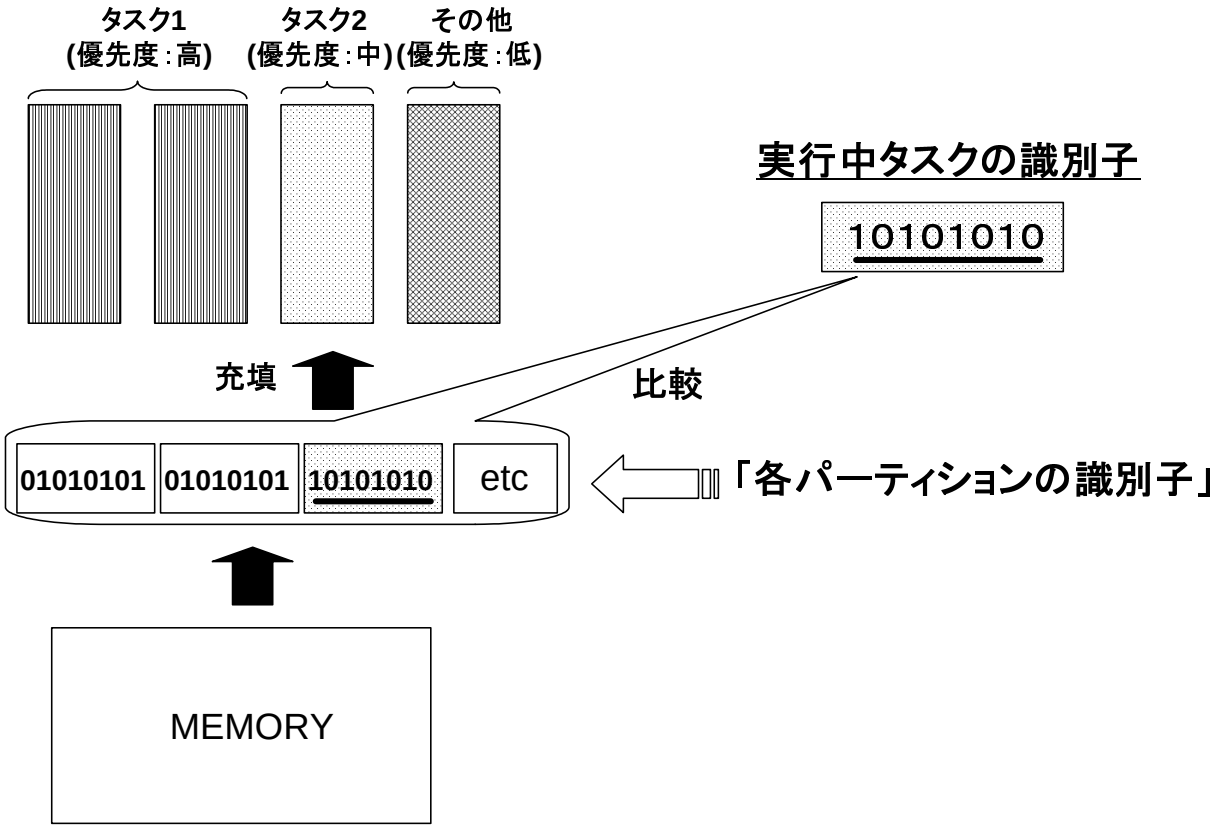

図 2.4: 優先度に基づいたパーティション分割

再構成キャッシュではタスク毎にパーティション分割を行うが，タスクが参照できるキャッシュ領域まで固定する必要はなく，キャッシュ全体を参照する．つまり，キャッシュ参照を行う際に他のタスクに割り当てられたパーティション領域にデータがあった場合には，キャッシュヒットとしてデータを読み書きする．

リアルタイムアプリケーションにおいて，速く実行を終わらせたいタスク順に優先度を与えて実行を行う．再構成を行う場合にも，この優先度を考慮したパーティション分割を行う．例えば図 2.4 のように優先度の一番高いタスクに二つのパーティション領域を割り当て，次に優先度の高いタスクに一つのパーティション領域，その他残りのタスクで一つのパーティションを使う．優先度の高いタスクの実行が終わった場合，パーティションの識別子を格納している専用レジスタを書き換え，新たにパーティション分割を行う．このようなパーティション分割を行うことによって，優先度の高いタスクに多くのキャッシュ領域を割り当て，高い処理速度を保障する．

またマルチスレッド型のプロセッサも優先度を考慮してタスク切り替えを行うことが有効である．マルチスレッド型のプロセッサではキャッシュミスが起きたらタスクを切り替えて，メモリからキャッシュにデータの充填を行う間，他のタスクの実行を少しでも進める．図 2.5 のようにキャッシュミスのタイミングで実行を切り替えるのに加えて，優先度の高いタスクがメモリからの充填が完了したときには実行をそのタスクに切り替える．

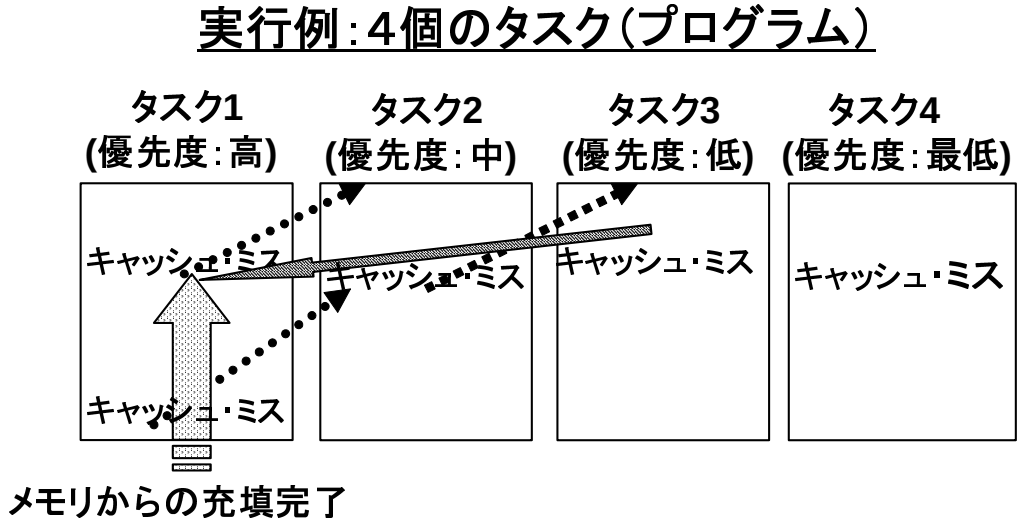

図 2.5: 優先度を考慮したタスク切り替え

本研究で用いるマルチスレッド型のプロセッサでは，タスクを切り替えるためにタスク固有のコンテキストセット｛汎用レジスタ，プログラムカウンタ，状態レジスタ（PSR）等｝をタスクの数だけ用意する．そうすることにより切り替えの時のオーバーヘッドを削減する．

このような優先度を考慮した再構成キャッシュとマルチスレッド型のプロセッサを用い

ることにより，優先度の高いタスクに多くのキャッシュ領域を割り当てることを可能にし、タスクの高い処理速度を保障する．

再構成キャッシュをリアルタイムアプリケーションに用いるもう一つの利点として，以下が挙げられる．

組み込み分野やマルチメディア処理などのリアルタイム処理では，単純なスループット，平均的な実行速度のみでなく，各タスクの実行がデッドライン時刻を越えずに終了することが重要である．このことを保障するために，タスクの最悪実行時間（WCET：Worst Case Execution Time）を用いたスケジューリング可能性の解析が行われることがある．その際，WCET が実際の実行時間と比較し過度に大きな場合は保障が困難となる．すなわち，実際には確実にスケジューリング可能なタスクセットに対して，スケジューリング不可能という解析結果となる．

WCET の決定に関して，実際にタスクを実行することにより求める場合と，タスクのプログラムを解析することにより見積もる場合がある．前者は最悪実行である保障が無いため適用が困難である．後者に関して，実際の実行ではプリエンプションにより中断して他のタスクへ実行が移り，キャッシュの内容が変化する可能性がある．このことは，キャッシュアクセスの解析において，全てをキャッシュミスとして見積もる必要があることを意味する．したがって，WCET は実際の実行時間と比較し，過度に大きくなる．

提案方式は，タスク間でのキャッシュ競合を削除することで，タスクの WCET を現実に近い値に見積もることを可能とし，スケジューリング可能性の保障を容易にする．

2.3 大規模データを格納するFIFOバッファ

メディアプロセッシングのような大規模な連続したデータを必要とするアプリケーションに対しては，FIFOパーティションを用意し，SDT[7]などのDMA機構によって転送される連続データを受け取る．これにより高速化を図る．

2.3.1 Stride Data Transfer（SDT）方式 [7]

本節では，メモリアクセス時間を抑える一手法として提案されている，DRAMの同一ページ内への連続アクセス方法（Stride Data Transfer:SDT）について述べる．詳しくは[7]を参照されたい．

SDTの概要

現在のDRAMのメモリアレイは複数のバンクから構成されており，バンクごとに独立した動作が可能になっている．DRAMアクセスはメモリアドレスのバンク（Bank）アドレスで一つのバンクを選択し，そのバンクに対し行（Row）アドレスを与え，該当するRowアドレスのデータ全体をセンス・アンプで増幅し，一旦ラッチする．そしてラッチされたデータに対して列（Col）アドレスを与えることでCASレイテンシ後にデータが読み出される（図2.6）．なお，Row，Colアドレスは信号線を共有しており，時分割でRowアドレス，Colアドレスの順に与えられる．

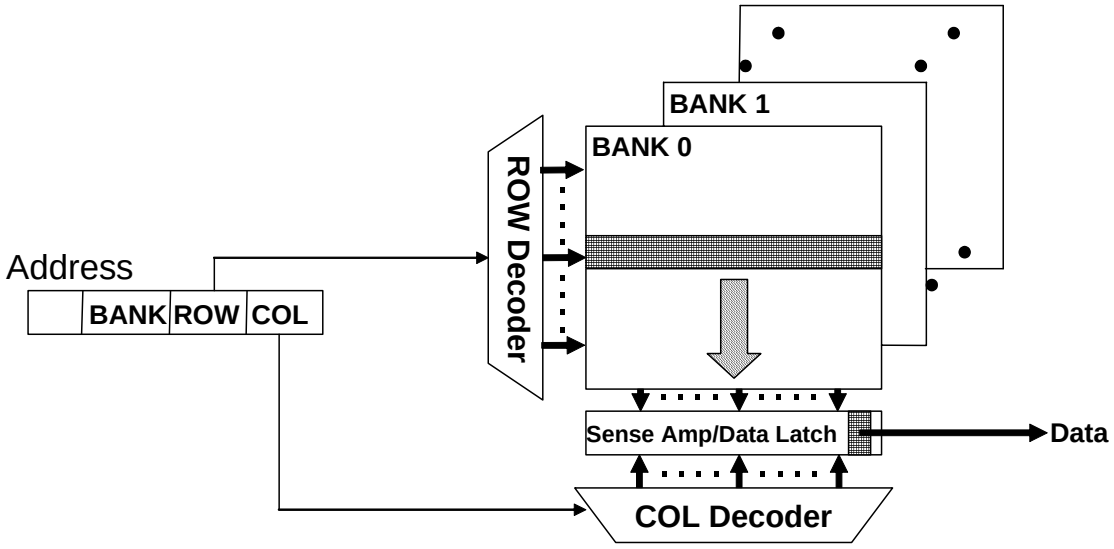

図 2.6: DRAMアクセス

しかし同一 Bank，Row アドレス内に一定間隔で存在する複数の不連続なデータをアクセスする際，従来のメモリコントローラ（MC）では各データを含むキャッシュブロック単位でアクセスし，それぞれのブロックに対して Bank，Row アドレスを毎回指定するため効率が悪く，データ転送までのレイテンシが増大する．

Stride Data Transfer（SDT）方式は同一の Bank，Row アドレス内に存在する複数の不連続なデータに対して初回のみ Bank，Row アドレスを指定し，その後は Col アドレスの連続指定でデータを転送する機構である．すなわち MC が一定間隔値を加算して Col アドレスを自動生成することにより，同一 Bank，Row アドレスに存在するデータに対して連続アクセスが可能になる．

従来の MC と SDT のデータ転送方式の比較を図 2.7 に示す．従来の MC では物理アドレスに一定間隔で存在する各データに対して Bank，Row，Col アドレスを毎回指定する為，データの読み出しのレイテンシが発生するが，SDT 方式により MC と DRAM 間での Bank，Row アドレスの再入力の除去によってデータの連続読み出しが可能になる．また図において，従来では三回のメモリリクエストを発行するのに対し，SDT 方式ではプロセッサと MC 間のメモリリクエストを一括化することでデータ転送効率が向上する．

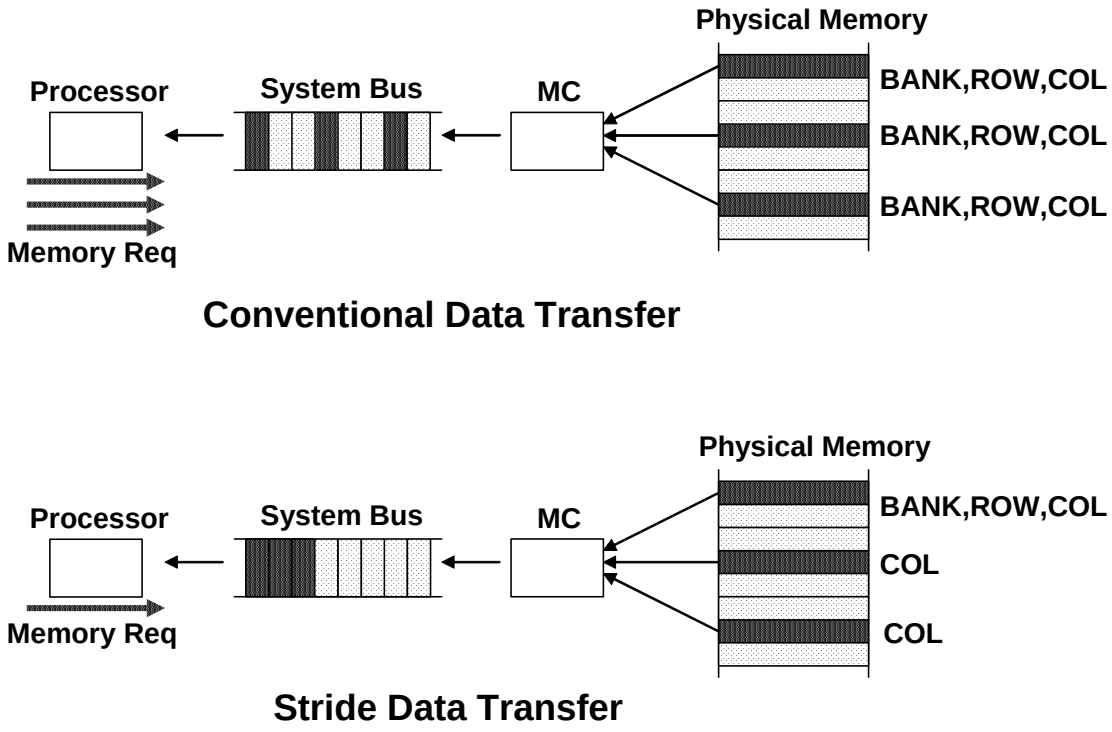

図 2.7: SDT 方式によるデータ転送

SDT 方式の動作

SDT 方式では MC が一定間隔値を加算して Col アドレスを自動生成する為，SDT 開始前に MC にある専用レジスタに転送するデータ数およびデータ間隔値を格納する．

SDT のメモリアクセス要求を検出するために，MC は物理アドレスのフィールドを使用する．通常メモリアドレスのフィールドは Byte offset を除いた下位ビットから Col，Row，Bank アドレスで構成され，残りの上位ビットはシステム依存である．ここではメモリアドレスの最上位 1 ビットをメモリアクセスモードの指定フィールドとする．図 2.8 にメモリアドレス形式を示す．この指定フィールドの値はプログラムからメモリアドレスの上位 1 ビットの値を変えることにより指定される．実際の SDT 方式のメモリアクセスは指定フィールドを除いた Bank，Row，Col アドレスで DRAM アクセスを行う．

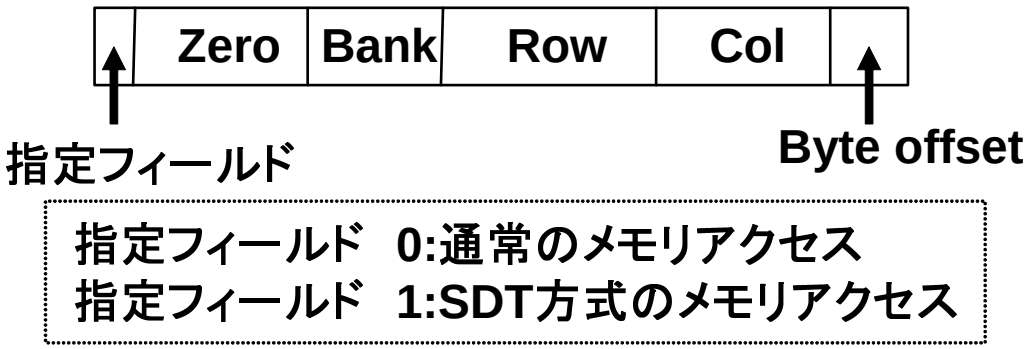

図 2.8: メモリアドレス形式

プロセッサは指定フィールドを“ 1 ”とするアドレスによりメモリリクエストを発行する．MC は受け取ったアドレスの指定フィールドをデコードし，SDT の開始アドレスを MC 内のレジスタに記憶し，DRAM アクセスを行いデータをプロセッサへ転送する．以下，記憶された開始アドレスにデータ間隔値を加算することにより次アドレスを生成し，Col アドレスの連続指定によって一定間隔毎のデータを読み出し，データバスを通じてプロセッサへ転送する．本研究では連続してアクセスするデータのサイズ（バースト長：Burst Length：BL）は 1 ワード（4byte）とする．

SDT 転送はプロセッサの命令実行のバックグラウンドで，プロセッサとメモリ間のシステムバスを占有してデータを連続転送する．プロセッサ内のキャッシュの一部を FIFO パーティションとして再構成を行い，転送されたデータを順次格納する．プロセッサは再構成キャッシュの FIFO パーティションから転送されたデータを読み出す．

データ転送の終了条件は以下の 5 つである．

(1) MC のレジスタにセットした転送データ数を超える場合
(2) データ間隔値の加算で Bank，Row アドレスが変化する場合
(3) 仮想記憶によるページ境界を越える場合
(4) プロセッサから SDT 転送以外のメモリリクエストが発行された場合
(5) 転送データ量が FIFO パーティションの容量を超える場合

(1) は転送が完了した時であり，(2) に関して，SDT は同一 Bank，Row アドレス内のデータ連続転送に限るため，どちらかのアドレスが変化した場合は Bank，Row アドレスを与え直すことで SDT を再開する．(3) のページ境界での終了は仮想アドレス空間に対する物理アドレスの連続性が保障されないため，この場合もアドレスを与え直すことで SDT を再開する．(4) では SDT 転送はプロセッサの命令実行のバックグラウンドでデータを連続転送しているため，プロセッサの命令キャッシュミス，ロード・ストア命令のデータキャッシュミスおよびライトバックによってメモリアクセスが発生する．このとき SDT 転送中の場合，プロセッサとメモリ間のシステムバスを占有しているため，SDT を中断してメモリアクセスを優先し応答する．その後プロセッサは FIFO 内のデータを使い続け，やがて FIFO が空になった時点で再リクエストが発行されて SDT が再開する．(5) は FIFO がオーバーフロー状態になる前にプロセッサは MC に SDT の一時中断を要求し，FIFO 内のデータを使い続け，やがて FIFO が空になった時点で再リクエストが発行されて SDT が再開する．

2.3.2 FIFO パーティション

メディアプロセッシングなどのアプリケーションでは大規模な連続したデータセットを扱い，そのデータは再利用されることが少ないため時間的および空間的局所性が存在せず，キャッシュメモリを有効利用できないことがある．このようなアプリケーションに対して本研究で用いる再構成キャッシュでは FIFO パーティションを用意し，メモリから転送されたデータを格納する．

FIFO パーティションに再構成を行う際に，専用レジスタに値をセットする．そのレジスタに値がセットされた時に，キャッシュの一部を FIFO パーティションとして再構成を行う（図 2.9）．再構成を行う際に，新たに FIFO パーティションとして使うキャッシュ領域の追い出しを行う．つまりパーティションとして使うキャッシュ領域の状態を調べ，データが更新された状態（“ dirty ”）ならば外部メモリを更新する．

再構成後には図 2.9 のように FIFO パーティションはカウンターによるアクセスとなり，通常のタグ，インデックスを用いたメモリアクセスを行わない．つまり，メモリアドレスの最上位ビットの指定フィールドを“ 1 ”とするメモリアクセス以外では，FIFO パーティションは参照されることはない．また，通常のアクセスでメモリからの充填を行う際にも FIFO パーティションの領域以外で充填を行う．

FIFO パーティションとして再構成を行う際には，タグ，比較器の拡張により，ウェイより細かい単位でのパーティション分割を行う．図 2.9 ではウェイ 3 の 8 セットの領域をパーティションとして用意している．ウェイより細かい単位での分割を行うことにより，SDT などの DMA 転送以外の通常のデータキャッシュとしての機能をできるだけ妨げないようにし，再構成の際に FIFO パーティションとして使うキャッシュ領域の追い出しを行う時にも，追い出しにかかるレイテンシを小さくする．

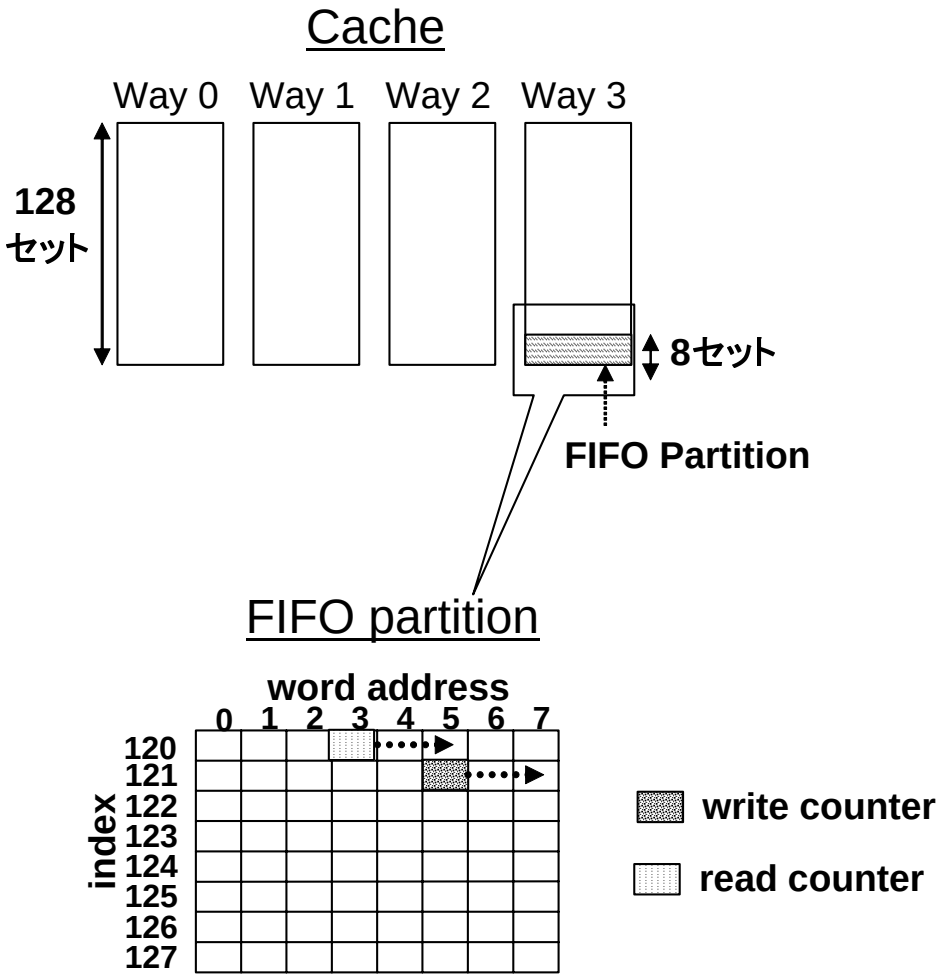

図 2.9: FIFO パーティション

FIFO パーティションをアクセスする際には，リードカウンタとライトカウンタを使用する（図 2.9）．リードカウンタはデータの読み出し位置を示し，ライトカウンタはデータの書き込み位置を示す．

専用のレジスタに値が書き込まれ，再構成が行われる時にはカウンタの値は 0 にリセットされる．

動作例を示すと，まずプロセッサがメモリにアクセスしFIFOバッファにデータが書き込まれライトカウンタが自動的にインクリメントされていく．その後，FIFOバッファのデータを読み出すとリードカウンタが自動的にインクリメントしていき，やがてリードカウンタがライトカウンタと同じ値を指す．カウンタ同士が同じ値を指す時，読み出しを要求した場合にはFIFO バッファに読み出すべき有効なデータは存在しないことを表し，ミスシグナルがアサートされる．

2.3.3 FIFOパーティションを用いたSDT動作

主記憶データベース（MMDB）等ではメモリ上で等間隔に並ぶデータにアクセスを行う．図2.10に通常キャッシュと，SDT転送方式を再構成キャッシュで行った場合のキャッシュ比較を示す．

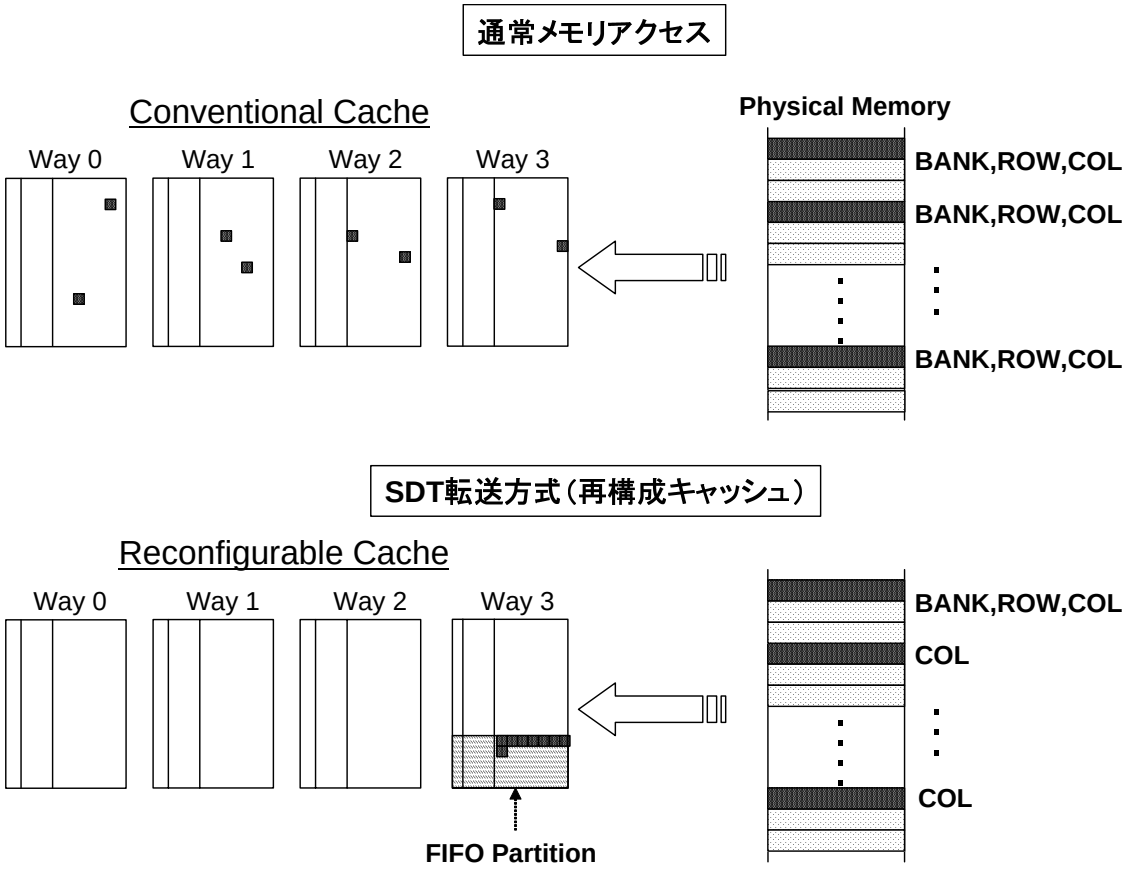

図 2.10: SDT方式によるキャッシュ比較

キャッシュを用いる目的として，再利用される可能性が高い時間的および空間的な局所性があるデータを格納することで，実行性能の向上を図ることが挙げられる．通常キャッシュでは，キャッシュ－メモリ間のデータの受け渡しはキャッシュブロックサイズで行われるが，MMDB のような必要なデータが不連続に存在する場合，ブロック内のすべてのデータが必要なデータとなることはなく，不必要なデータも含まれる．その為、必要なデータが断片的な状態でキャッシュに配置されることになる（図 2.10）．

このように不必要なデータがキャッシュに配置されることにより，不必要なデータの転送時間が発生するだけでなく，再利用性の高いデータを追い出すことにより，キャッシュとしての機能の低下を招く問題もある．

本研究で着目するメディアプロセッシングのような大規模な連続したデータを必要とするアプリケーション等では，必要なデータは連続してメモリ上に配置されている場合もある．

この場合，キャッシュ－メモリ間のデータの受け渡しを行うブロック内のデータに不必要なデータが含まれることは少なくなるが，必要なデータがすべてキャッシュに取り込まれるまで，ブロックサイズでのデータ転送を繰り返すことになる．また，再利用性の高いデータの追い出しによるキャッシュ機能の低下は回避できない．

SDT 方式を再構成キャッシュで用いた場合には，必要なデータのみを一括して連続転送することが可能である．これにより，不必要なデータの転送，ブロックサイズでのデータ転送といった無駄な転送時間を無くすことができる．また，メモリから転送される連続データを FIFO パーティションに格納することで，先に問題となっていた再利用性の高いデータの追い出しによるキャッシュ機能の低下を回避する．

2.4 プリフェッチしたブロックを格納するプリフェッチバッファ

2.4.1 プリフェッチ機構

近年，プロセッサとメモリの動作速度差を隠蔽するため，データプリフェッチ機構が使われており，様々な研究がなされている [8, 9, 10, 11].

本研究では，後に使用されるデータ・ブロックを洗い出し，特別な命令を使用して該当ブロックをキャッシュに取り組むように記憶階層に指示するソフトウェア・プリフェッチに着目する.

プロセッサの命令実行のバックグラウンドで，プリフェッチデータを転送し，キャッシュの一部をプリフェッチパーティションとして再構成を行い，転送されたデータを順次格納する．プロセッサは再構成キャッシュのプリフェッチパーティションから転送されたデータを読み出す.

プリフェッチ命令を発行する際，複数のプリフェッチ要求を多重に発行することを，再構成キャッシュとスプリットフェーズ式のメモリバス，高性能の MC を使用することにより可能とする．再構成キャッシュの一部をプリフェッチパーティションとして用意することにより，実行パイプラインをストールさせることなしに多重プリフェッチを可能にする.

図 2.11 に，プリフェッチを用いた実行効果の例を示す．ここで，キャッシュのブロックサイズを 32byte とする．図（a）のコードはプリフェッチなしの実行であり,（b）のコードは多重プリフェッチを伴う実行である．プリフェッチなしの実行は，多重プリフェッチを伴った実行に比べてサイクル数が多くなっている．これはプリフェッチなしの実行が，バスの使用に関して断片的になっているために 2 つのロード命令（ld）がストールしているのに対し，多重プリフェッチ実行では最初のロード命令でプリフェッチによるキャッシュへの充填が間に合っていないためにストールしているのみで，後続するその他の命令はストールなしで実行できるためである.

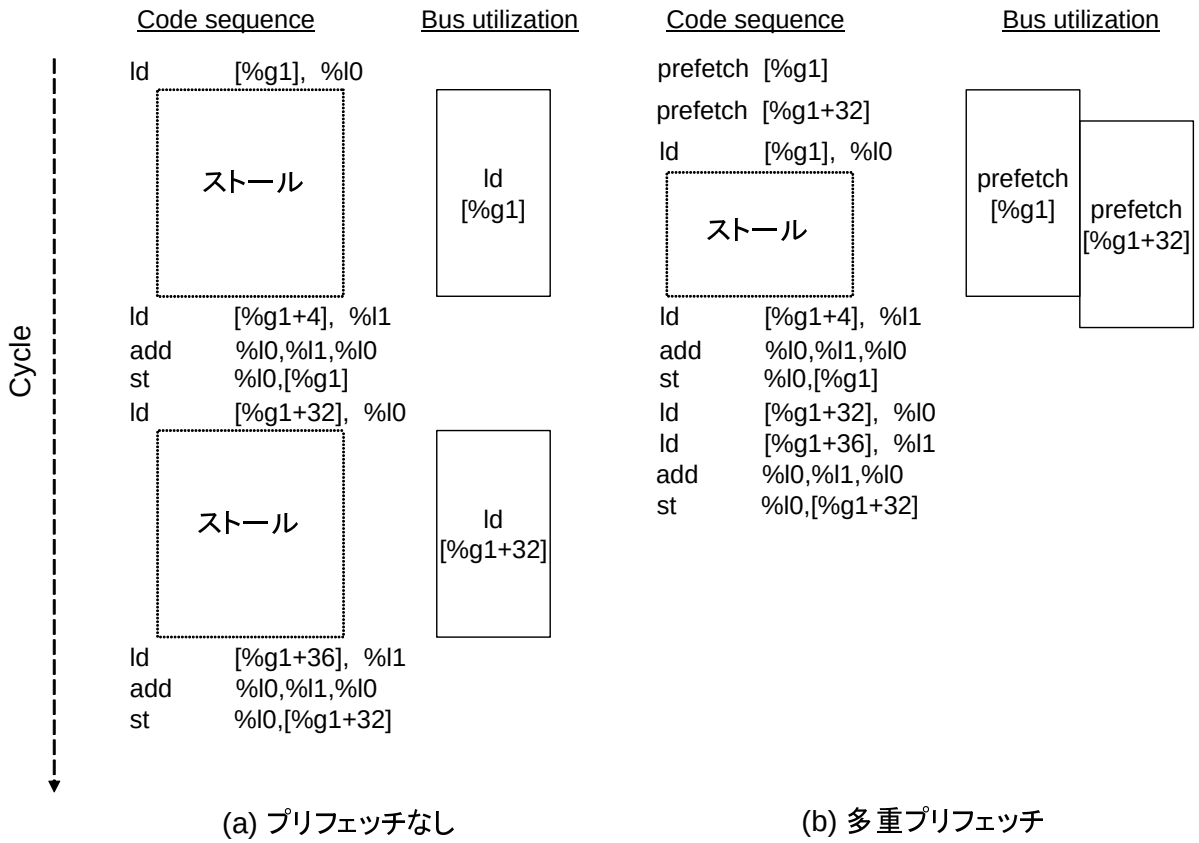

図 2.11: 多重プリフェッチの実行効果

SDT 転送方式を用いた場合でも，メモリからキャッシュにあらかじめ必要となるデータをプリフェッチできる．ブロックサイズを 32byte とした時，プリフェッチ機構ではデータサイズが 1 ブロックサイズ（32byte）でのデータ転送になるのに比べ，SDT 方式では 1 ワード単位（4byte）での転送が可能である．従って，必要なデータのみを一括して連続転送することにより，ブロック単位での転送に比べ，不必要なデータの転送といった無駄な転送時間を無くすことができる．しかし，プリフェッチ機構では転送されてきたデータに対して読み出し，書き込みの両方ができるのに比べ，SDT 方式では読み出すことしかできなく，書き込みを行うことができないという欠点がある．

このように SDT 転送方式，プリフェッチ機構は互いに異なる利点と欠点を持っている．本研究では両方の機能を備えることにより，用途に応じてこの二つの機能を使い分けて，お互いの欠点を補う．

2.4.2 プリフェッチパーティション

プリフェッチ命令で該当ブロックをキャッシュに取り込む際にはキャッシュの一部をプリフェッチパーティションとして再構成を行う（図 2.12）．再構成を行う際には，FIFO パーティションの時と同様に専用のレジスタを用いる．この専用レジスタに値がセットされたときにキャッシュは再構成を行う．再構成を行う際にはウェイより細かい単位での分割を行い，プリフェッチ転送以外の通常のデータキャッシュとしての機能をできるだけ妨げないようにする．

プリフェッチパーティションが FIFO パーティションと違うところは，図 2.12 のようにタグ，比較器の拡張により，パーティションの参照が可能となる．つまり，プリフェッチパーティション内にあるプリフェッチ転送されたデータを参照するときには，メモリアドレスを用いたキャッシュアクセスを行い，キャッシュヒットとしてプロセッサにデータを渡す．

通常のアクセスでメモリからの充填を行う際には，FIFO パーティションと同様にプリフェッチパーティションの領域以外で充填を行う．

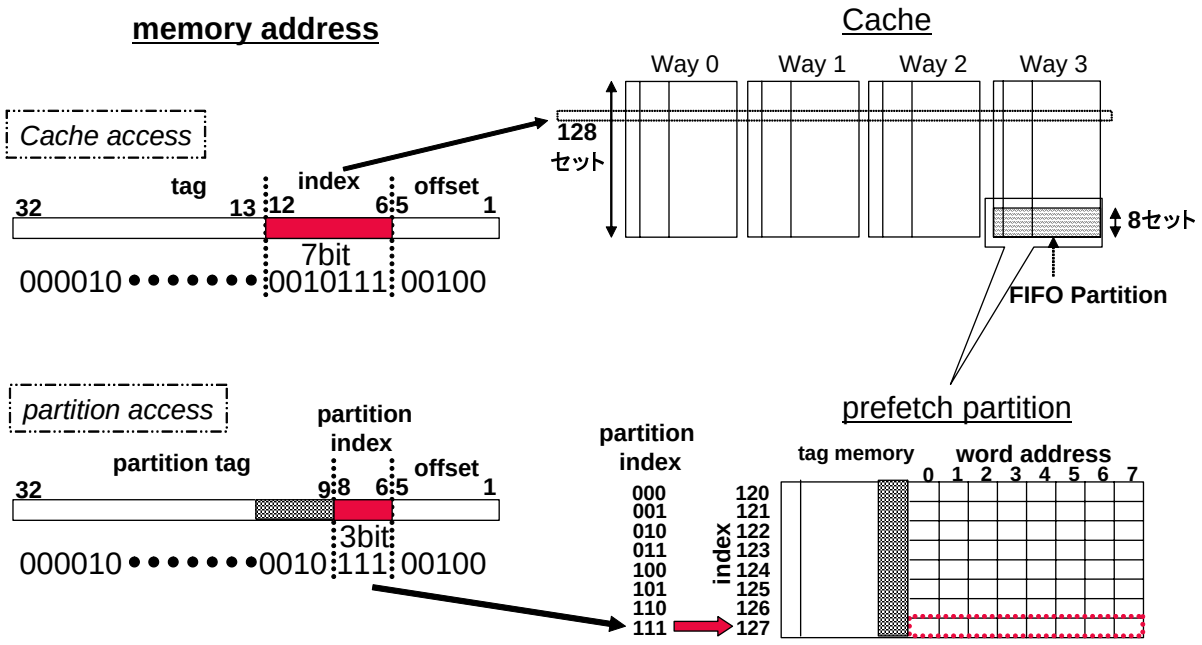

図 2.12: プリフェッチパーティション

様々なアプリケーションでプリフェッチ機能が有効であるが，通常のキャッシュでは，プリフェッチ転送してきたデータをキャッシュに格納した場合に，他のデータとの競合により実際に使用される前にリプレースされる，あるいはプリフェッチデータが頻繁にアクセスされるデータを追い出す問題がある（図 2.13）．これに対し，本研究では再構成キャッシュを用いて，再構成された一つのパーティションをプリフェッチバッファとして使用することにより解決する．

またプリフェッチパーティションを用いることにより，パーティションサイズをソフトウェアから見切ることでき，計画的にプリフェッチデータを溜めておく，またはプリフェッチデータを追い出すことが可能となる．

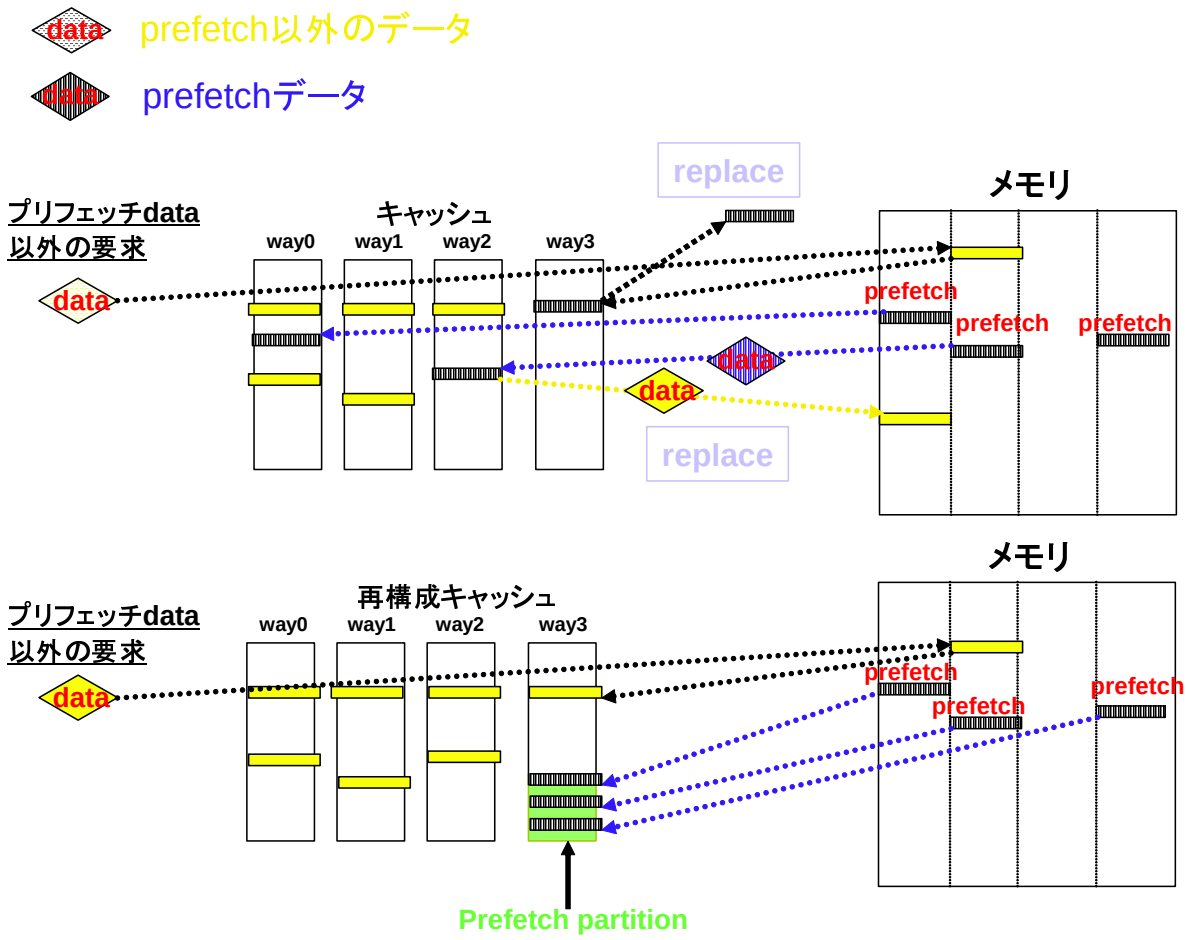

図 2.13: プリフェッチ機構での通常キャッシュと再構成キャッシュの比較

第3章　シミュレーション環境

3.1　CPU の仕様

本研究で作成したシミュレータは，命令セットとして SPARC Architecture Version 8[4] の整数命令群を使用する．本シミュレータは SPARC 特有のレジスタウィンドウを持たず，ウィンドウを明示的に回転する命令（save/restore）を使用しないことを前提とする．GCC[12] にはレジスタウィンドウを回さないコードを生成するためのオプション（-mflat）を持つバージョンがあり，レジスタウィンドウを持たないことはプログラム開発において支障はない．

命令実行に必要な基本レイテンシは乗算には 3 クロックサイクル，除算には 15 クロックサイクル，その他には 1 クロックサイクルとする（表 3.1）．

作成したシミュレータは，シングルスレッド型のプロセッサによる実行，マルチスレッド型のプロセッサによる実行を可能としている．マルチスレッド型のプロセッサで実行を行う際には，プロセッサ内に実行タスクと同数の数のタスクコンテクストセットを備えることにより，メモリアクセスを行わずに実行タスクを切り替えることができる．

その際，パイプラインフラッシュによるタスク切り替えのオーバヘッドは 5 クロックサイクルとする．

リアルタイムアプリケーションにおいて，速く実行を終わらせるべきタスク順に優先度を与えて実行を行う．マルチスレッド型のプロセッサを実行する場合にも，優先度を考慮してタスク切り替えを行うことが有効である．マルチスレッド型のプロセッサではキャッシュミスが起きた場合，実行タスクを切り替えて，メモリからキャッシュにデータの充填を行う間，有効実行を行う．作成したシミュレータではキャッシュミスのタイミングで実行を切り替えるのに加えて，優先度の高いタスクのキャッシュミスに起因するメモリからの充填が完了したときには実行をそのタスクに切り替える動作も可能とする．

表 3.1: 命令実行レイテンシ

Class	Latency
乗算	3cycles
除算	15cycles
others	1cycles

表 3.2: ASI=0x18 フィールドのレジスタ

register number	special register
0	way0 タスク識別子
4	way1 タスク識別子
8	way2 タスク識別子
12	way3 タスク識別子
16	プリフェッチパーティション専用レジスタ
20	実行中タスク識別子
24	SDT 転送時のデータ間隔値（MC）
28	SDT 転送時のデータ転送量（MC）
32	FIFO パーティション専用レジスタ

キャッシュを動的にパーティション分割するために，専用の設定レジスタを設ける．この設定レジスタへの書き込みは I/O 空間へのストア命令を使用する．シミュレータでは，ASI=0x18 を指定した alternate space へのストア命令[1]を使用する．

ASI=0x18 で指定される専用レジスタを表 3.2 に示す．

3.2 キャッシュ構成

本研究で作成したシミュレータではデータキャッシュは 4 ウェイセットアソシアティブキャッシュを用いる．

キャッシュサイズは 16KB，一つのウェイの領域は 4KB であり，1 ブロックサイズは 32byte とする（図 3.1）．

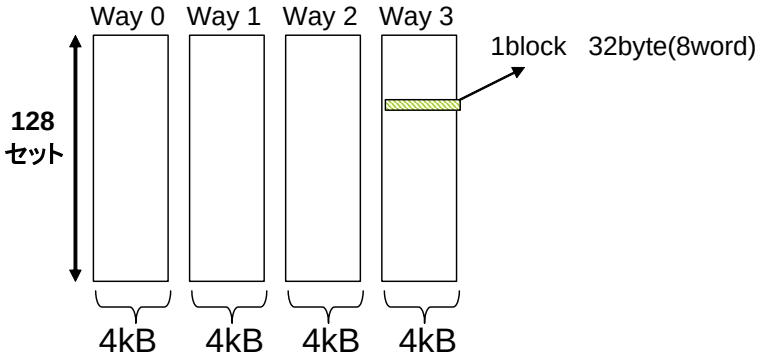

図 3.1: キャッシュ構成

[1]ASI は SPARC Version 8 における，アドレス空間識別子である．

メモリアクセスはキャッシュヒットした時には 1 クロックサイクルとし，キャッシュミスした時には，ペナルティは 100 クロックサイクルとする．なお，命令キャッシュ，二次キャッシュは本シミュレータには存在しない．

3.3 優先度パーティション分割

作成したシミュレータは，メモリからキャッシュへ充填を行う際には，LRU 法と本研究の提案手法である優先度パーティション分割を可能にする．LRU 法は該当ブロックの中で，使用されていない時間が最も長いブロックを置き換える．優先度パーティション分割を行うために，実行中のタスク，パーティション毎にタスク識別子を格納する専用レジスタを用意する．メモリからキャッシュに充填を行う際に，専用レジスタに格納されている実行中のタスクの識別子，パーティション毎の識別子の比較を行い，一致したパーティションに充填を行う．一致したパーティションが二つ以上あった場合には，LRU 法と同じく使用されてない時間が最も長いブロックを置き換える．

優先度パーティション分割を行うときには，まず専用レジスタへ識別子をセットする．LRU 法を用いる場合には，値をセットしないため，実行中のタスク，パーティションの専用レジスタには 0 が格納されており，4 つのパーティションの中から使用されていない時間が最も長いブロックを置き換える．

3.4 SDT 動作

再構成キャッシュが FIFO パーティションに再構成を行うために，専用レジスタを設けている．この専用レジスタに値をセットした場合，再構成キャッシュは FIFO パーティションを一部に備えたキャッシュへと再構成を行う．同時にリードカウンタ，ライトカウンタの値をリセットする．

再構成を行う際に，新たに FIFO パーティションとして使うキャッシュ領域追い出しを行う．すなわちパーティションとして使うキャッシュ領域の状態を調べ，データが更新された状態（“ dirty ”）ならば外部メモリを更新する．作成したシミュレータでは，キャッシュの 1 ブロックの状態を調べるレイテンシを 7 クロックサイクルとする．また，調べたデータが更新された状態の時には，100 クロックサイクルで外部メモリを更新する．

また，メモリコントローラー（MC）にある専用レジスタに，データの間隔値，データの転送量を書き込む．この専用レジスタは MC にある事をを考慮して，書き込みに掛るペナルティを 40 クロックサイクルとする．

メモリアドレスの最上位 1 ビットの指定フィールドを 1 としたメモリアクセスを行うことにより，SDT 転送を開始する．最初のアクセスではキャッシュミスと同様に 100 クロックサイクル掛けて，メモリへアクセスを行う．その後 MC 側が，専用レジスタに格納されている間隔値を加算したメモリアドレスのデータを，プログラム実行のバックグラウンド

で 10 クロックサイクル毎に転送する．

3.5 プリフェッチ動作

再構成キャッシュがプリフェッチパーティションとして再構成を行うために，専用レジスタを設けている．この専用レジスタに値をセットすることによって，プリフェッチパーティションを一部に備えたキャッシュへと再構成を行う．パーティションとなる領域が，ウェイの下の領域ならば，タグを拡張した部分に 1 を代入する．また，同様にウェイの上の領域を使って再構成を行った場合には，タグを拡張した部分に 0 を代入する．パーティションのタグ拡張を行うことにより，分割前のメモリアドレスでパーティションにアクセスすることができ，パーティション内データ参照が可能となる．

プリフェッチを行う際には，0 番レジスタへのロード命令[2]をプリフェッチ命令として発行する．このプリフェッチ命令はノンブロッキングで実行され，ミスの場合の該当ブロックの転送はプロセッサの命令実行のバックグラウンドで行い，100 クロックサイクル掛けて，キャッシュへの充填が行われる．

作成したシミュレータでは通常キャッシュでのプリフェッチ，プリフェッチパーティションを用いたプリフェッチを可能にする．また，最高 8 個までの多重プリフェッチを可能としている．

[2]SPARC アーキテクチャでは，0 番レジスタは常に固定値（0）である．

第4章　評価と考察

4.1　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，リアルタイムタスクの優先度に基づいたパーティション分割を評価する際には単純なソートプログラム，行列計算プログラムを用いる．FIFO パーティションおよびプリフェッチパーティションの評価には Livermore Kernel[13] を用いる．

4.2　結果と考察

4.2.1　リアルタイムタスクの優先度に基づいたパーティション分割

リアルタイムアプリケーションを実行させる場合，多くのタスクに分けて処理を行うが，効率の良い実行のためにマルチスレッド型のプロセッサを使用する．

評価を行うにあたって，以下の３つの比較を行う．

(1) シングルスレッド型のプロセッサ（LRU 法）
(2) マルチスレッド型のプロセッサ（LRU 法）
(3) マルチスレッド型のプロセッサ（優先度パーティション分割）

（1）はマルチスレッドではなくシングルスレッドの実行を行う．つまり，タスクを実行中にキャッシュミスが起きた場合にはタスクを切り替える事なく，ストールしてメモリからキャッシュに充填されるのを待つ．(2) , (3）はマルチスレッド型のプロセッサでの実行であり，キャッシュミスが起きた場合にはタスクを切り替え，他のタスクの実行を行う．また，マルチスレッド型のプロセッサは優先度に基づくスレッド切り替えを行うものとする．(1) , (2）はメモリからキャッシュに充填を行う際に，複数のブロックの中から使用されずにいた時間が最も長いブロックを置き換える LRU 法を用いる．(3）は本研究の提案手法である再構成キャッシュを用いてタスク毎にキャッシュ領域を割り当てる方法である．メモリからキャッシュに充填を行う際に，専用レジスタに格納されている実行中のタスクの識別子，パーティション毎の識別子の比較を行い，一致したパーティションに充填を行う．

評価にあたって，４つのタスクを実行する．また，実行を速く終了させたい順番に従って，タスクに優先度を与える．(3）では優先度に従って，図 4.1 のようにパーティション

分割を行う．一番優先度の高いタスク 1 に二つのパーティションを割り当てる．二番目に優先度の高いタスク 2 に一つ割り当て，残ったタスク 3，タスク 4 で一つのパーティションを共有して使う．そして，図 4.1 のように一つのタスクの実行が終了したとき残ったタスクで，優先度，今まで使っていたパーティション領域を考慮して，新たにパーティション分割を行う．

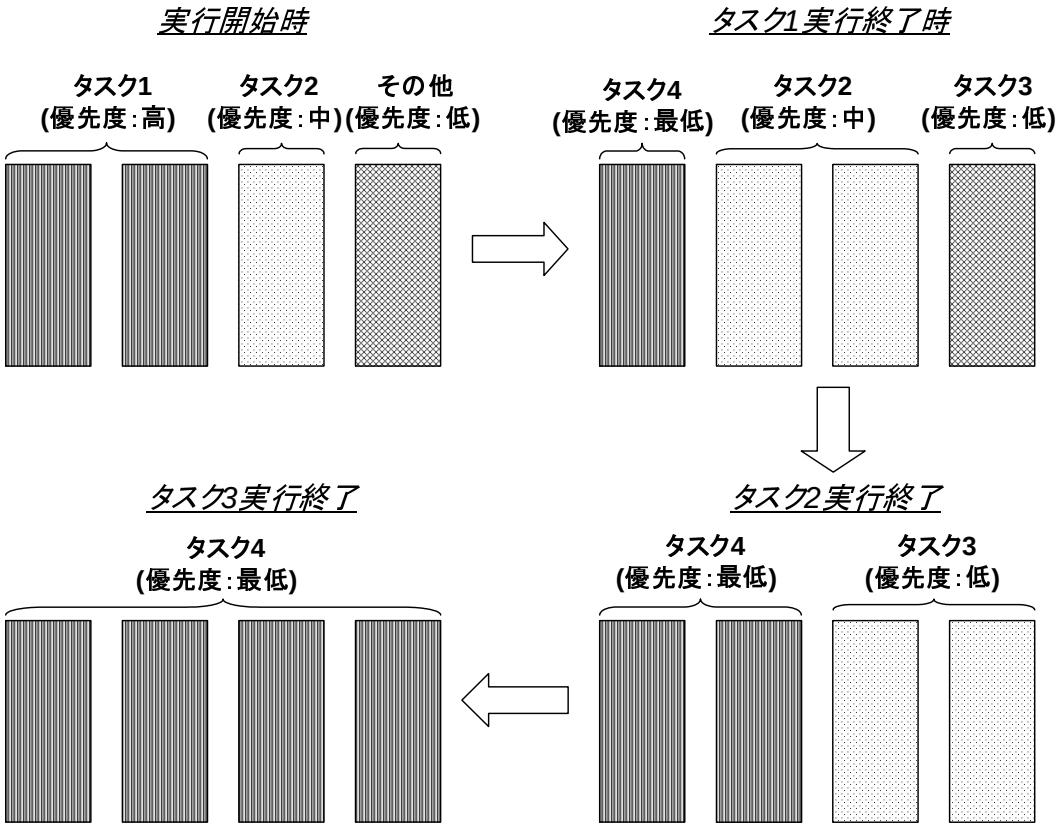

図 4.1: タスク優先度によるパーティション分割

図 4.2 にソートプログラムを４つのタスクとして実行を行った結果を示す．図 4.2 に示される「3-1」は，再構成キャッシュの最高優先度タスクへのパーティション割り当て領域を 3 つのパーティションとして，実行した場合である．

		タスク1	タスク2	タスク3	タスク4	全タスク
Single (LRU)	実行終了clock	2,355,749,246	4,711,498,393	7,067,247,540	9,422,996,687	9,422,996,687
	キャッシュミス数	2,441,840	2,441,839	2,441,839	2,441,839	9,767,357
Multi (LRU)	実行終了clock	2,401,757,853	4,577,003,210	7,955,543,271	8,467,275,929	8,467,275,929
	キャッシュミス数	2,873,032	2,942,676	3,644,715	3,317,816	12,778,239
Multi (task)	実行終了clock	2,504,899,229	4,765,656,469	8,254,478,272	8,603,403,693	8,603,403,693
	キャッシュミス数	3,870,686	4,714,921	6,079,181	5,373,448	20,038,236
3-1 (task)	実行終了clock	2,453,265,654	4,864,270,526	7,430,767,410	9,066,048,144	9,066,048,144
	キャッシュミス数	3,422,092	6,830,397	10,173,062	11,663,966	32,089,517

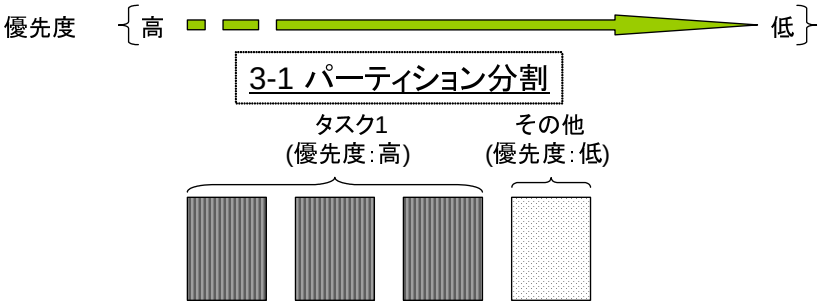

図 4.2: ソートタスク実行結果

図 4.3 に，行列計算を行うプログラムを４つのタスクとして実行を行った結果を示す．

		タスク1	タスク2	タスク3	タスク4	全タスク
Single (LRU)	実行終了clock	2,565,045,797	5,130,091,693	7,695,137,589	10,260,183,485	10,260,183,485
	キャッシュミス数	16,891,330	16,891,331	16,891,331	16,891,331	67,565,323
Multi (LRU)	実行終了clock	5,802,740,549	5,806,093,242	5,810,687,660	5,811,457,837	5,811,457,837
	キャッシュミス数	18,320,443	18,322,075	18,341,379	18,326,771	73,310,668
Multi (task)	実行終了clock	4,949,318,809	5,490,321,021	6,827,950,583	6,827,950,579	6,827,950,583
	キャッシュミス数	17,263,831	21,380,705	34,204,839	34,204,838	107,054,213

図 4.3: 行列計算タスク実行結果

両方の結果での全タスクの実行終了クロックが示すように，複数のタスクを実行する際にはシングルスレッド型のプロセッサを使用するより，マルチスレッド型のプロセッサを使用するほうがスループットが向上した．キャッシュアクセスの観点から見ると，キャッシュミスの多い行列計算のプログラムを実行させた場合の方がマルチスレッドを用いた時の効果は大きく，総実行クロック数を大幅に減らすことができた（図 4.3）．

キャッシュミスが多いほど，メモリから充填するストールする時間が大きくなるため，シングルスレッドで実行した場合には総クロック数が増える．マルチスレッドで実行した場合にはストールしている間，他のタスクの実行を行うため，ストールによるレイテンシを隠蔽することができる．その為，キャッシュミスが多いほどマルチスレッドはより多くのレイテンシを隠蔽することができる．しかし，マルチスレッドではキャッシュミスが起きた時にタスクを切り替えるため，キャッシュミスの多いタスクを実行した場合にはタスクは頻繁に切り替わる．その為，一つのキャッシュをタスクが競合して使う状態となり，キャッシュミスがさらに増える傾向がある．

キャッシュミスの多いタスクを実行した場合には，タスク切り替えが頻繁に起きるため，優先度を考慮してタスク切り替えを行っているにもかかわらず，図 4.3 のようにタスク 4 つそれぞれの実行終了クロックがほとんど同じである．逆にキャッシュミスの少ないタスクを実効した場合には，タスク切り替えがほとんど起きないため，図 4.2 のようにシングルスレッドで実行した時とほとんど変わらない結果となる．例えば，優先度の高いタスク 1 を実行している時に，キャッシュミスが少ないタスクではタスク 2 に実行が切り替わることはあっても，タスク 3，タスク 4 に切り替わることはほとんどない．その為シングルスレッドで実行した時とほとんど変わらない結果となる．

お互いの総実行クロック数を比べてみると，行列計算のタスクを実行した時にはキャッシュミスが多い分，ストールによる多くのレイテンシを隠蔽することができている．タスクが頻繁に切り替わることによりキャッシュミスの数は増えるが，それでもシングルスレッドと比較した時の実行性能の差は大きく，全タスク実行クロック数を半分程度まで減

らす結果となった．ソートのタスクを実行した時には，キャッシュミスによるストールが少ないため，レイテンシの隠蔽による効果が小さく，シングルスレッドとほとんど変わらない結果となる．

また，再構成キャッシュによるパーティション分割に関して，ソートのタスクのようなキャッシュミスが少ないタスクの場合には，タスク切り替えが起こる頻度が低いため，タスク間でキャッシュを競合して使わないため，逆に優先度の高いタスクのキャッシュ領域を限定する．つまり，LRU 法では優先度の高い一つのタスクのキャッシュ領域は最大で 4 つのパーティション領域を使えるのに対し，再構成キャッシュでは最大 2 つのパーティション領域であるために，優先度の高いタスク一つの実行性能を比較した場合劣る．そこで，優先度の高い一つのタスクの領域を 3 つのパーティションにして実行した結果，2 つのパーティション領域の時より優先度の高いタスク一つの実行性能は改善されたが，LRU 法より良くなることはなく，4 つのタスクの実行終了クロックは遅くなった．行列計算のタスクのようなキャッシュミスが多いタスクの場合には，タスクの切り替えが頻繁におきるため，キャッシュを同時にいくつかのタスクがアクセスする．そうした場合，優先度パーティション分割を行うと優先度の高いタスクは，パーティション分割したキャッシュ領域を利用できることが保障される為，優先度の高いタスクの実行を速くすることができた．

様々なタスクを実行させてみた結果，キャッシュのアクセス頻度の低いタスクの場合にはソートのタスクの場合と同様の結果であり，アクセス頻度の高いタスクの場合には行列計算のタスクの場合と同様の結果であった．また，同じタスクを 4 つ実行するのではなく，それぞれ違う 4 つのタスクで実行してみた．例えば，キャッシュアクセス頻度の低いタスクに一番高い優先度を与えた場合，二番目，三番目に高い優先度にキャッシュアクセス頻度の高いタスクを入れることにより，タスク 3，タスク 4 にも実行を切り替えるのを目的として実行した．しかし，結果として優先度の高いタスクの実行が速くなることはなかった．また，全タスクの実行終了クロックではマルチスレッド型のプロセッサ（LRU 法）を用いたものが一番良い結果となった．複数のタスクを実行させる場合には，シングルスレッド型の CPU を用いるより，マルチスレッド型の CPU を用いる方が性能は良くなる．

4.2.2 FIFO パーティションの評価

FIFO パーティションのサイズによって，実行性能にどれだけの違いがあるかを調べるために，Livermore Kernel#21（matrix*matrix product）（図 4.4）と Livermore Kernel#3（inner product）（図 4.5）の実行をパーティションのサイズを変えて行った.

```
#include<stdio.h>

#define Loop 10
#define N 64
int px[64][64],vy[64][64],cx[64][64];

int main(void)
{
  int l,i,j,k;
  for(l=0;l<Loop;l++){
    for(k=0;k<25;k++){
      for(i=0;i<25;i++){
        for(j=0;j<N;j++){
          px[j][i] += vy[k][i] * cx[j][k];
        }
      }
    }
  }
  return 0;
}
```

図 4.4: Livermore Kernel#21 コード

```
#include<stdio.h>

#define Loop 10
#define N 4095

int z[4096],x[4096]

int main(void)
{
  int l;
  int k;
  int q;
  for(l=1;l<=Loop;l++){
    q = 0;
    for(k=1;k<=N;k++){
      q += z[k] * x[k];
    }
  }
  return 0;
}
```

図 4.5: Livermore Kernel#3 コード

図 4.6 に Livermore Kernel#21 の結果を，図 4.7 に Livermore Kernel#3 の結果を示す.

	Normal	128entry (4kB)	64entry (2kB)	32entry (1kB)	8entry (256B)	4entry (128B)
総clock数	45,233,299	53,117,037	34,543,364	29,481,189	25,781,193	24,556,036
キャッシュ・ミス	274,163	216,585	116,869	67,036	29,664	17,288
instruction	17,291,162	18,491,168	18,491,168	18,491,168	18,491,168	18,491,168
キャッシュ・アクセス	6,032,562	6,032,562	6,032,562	6,032,562	6,032,562	6,032,562
partition over		390	781	1,562	6,249	12,499
counter clash		0	0	0	0	0
FIFOミス		116,831	36,844	35,698	35,698	35,699

図 4.6: Livermore Kernel#21 実行結果

	Normal	128entry (4kB)	64entry (2kB)	32entry (1kB)	8entry (256B)	4entry (128B)
総clock数	2,038,085	1,627,028	1,388,224	1,331,372	1,285,301	1,313,391
キャッシュ・ミス	10,244	2,963	1,747	1,139	683	607
instruction	942,029	1,064,885	1,064,885	1,064,885	1,064,885	1,064,885
キャッシュ・アクセス	368,618	368,618	368,618	368,618	368,618	368,618
partition over		39	79	159	639	1,279
counter clash		0	36	72	68	172
FIFOミス		1,779	653	689	697	1,057

図 4.7: Livermore Kernel#3 実行結果

結果の中で示される“ partition over ”は図 4.8 が示すように，カウンターがパーティション内を一周した回数を示すものであり，FIFO パーティションがどれだけ使われているかの尺度となる．“ counter clash ”はライトカウンタが一周して，リードカウンタに追いついた回数を示すものであり，SDT の転送データ量が FIFO パーティションの容量を超える場合の尺度となる．“ FIFO ミス ”はカウンタ同士が同じ値を指す回数であり，読み出し要求に対して FIFO パーティション内に読み出すべき有効なデータが存在しないため，キャッシュミスと同様にメモリからの充填を行う．

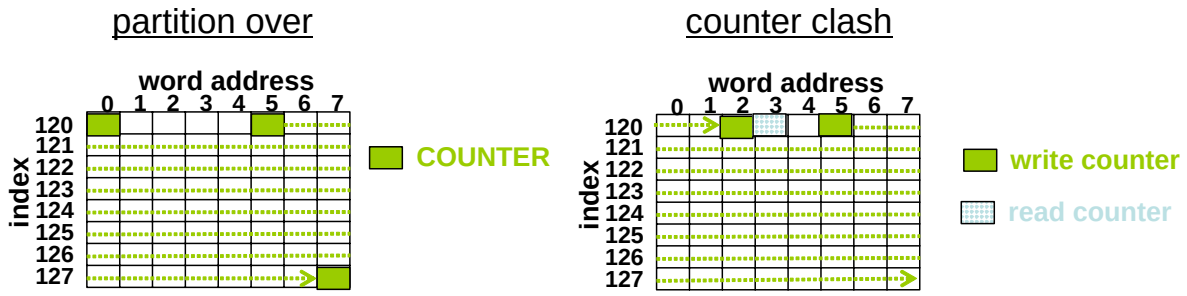

図 4.8: 「partition over」と「counter clash」の図

図 4.6 では，ウェイ単位（128entry）で再構成を行った場合には，Nomal で示す SDT 転送を行わない場合よりも実行が遅くなった．パーティションのサイズを小さくするほど実行が速くなり，4entry にした時には約半分のクロック数で実行を終えた．この場合ウェイより細かい単位での分割を行うことにより，SDT 転送以外の通常のデータキャッシュとしての機能をできるだけ妨げないため，大きいサイズでパーティション分割を行った時よりもキャッシュミス数が小さくなり，実行が速くなる．また，再構成の際に FIFO パーティションとして使うキャッシュ領域の追い出しを行う時に，追い出しにかかるレイテンシが小さくなることも実行が速くなった原因として挙げられる．

4entry よりもさらに小さくした場合でもクロック数，キャッシュミス数は変わらなかった．これは SDT 転送以外のアクセスがパーティション領域（ウェイ 3 の 124 セット以上）をアクセスすることがないためである．

図 4.7 では，図 4.6 に比べ SDT の転送データ量が FIFO パーティションの容量を超える“ counter clash ”の値が 0 ではない．“ partition over ”の数は図 4.6 の方が多く，パーティションを使うことは多いが，転送データ量が FIFO パーティションの容量を越えることはない．これは Livermore Kernel#21 が Livermore Kernel#3 に比べ，キャッシュミスの数が多く，キャッシュミスが原因で SDT 転送が中断することが多いためである．一方，Livermore Kernel#3 では，キャッシュミスの数が少なく SDT 転送が中断することが少ない．その為，FIFO パーティション領域が一杯になるまで転送を続ける．領域が一杯になったとき転送を終了し，FIFO 内のデータを使い続け，データを使いきったとき，FIFO ミスとして再リクエストが発行されて SDT を再開する．パーティション領域が小さくなるほ

ど，転送データで領域が一杯になることが多く，その分 FIFO ミスが増える．そのため，4entry では FIFO ミスが増え，8entry に比べて総クロック数が増えた．

Livermore Kernel#21 のようなキャッシュミスの多いプログラムでは，FIFO パーティションの領域を小さくするほど，データキャッシュとパーティションとの競合を避けられる，または追い出しにかかるレイテンシを小さくすることができるため，実行を速くすることができた．しかし，Livermore Kernel#3 のようなキャッシュミスの少ないプログラムで，転送量の多い SDT 転送を実行する場合にはパーティション領域一杯まで転送を行う．そのためパーティションを小さくするほど，実行が速くなるということはなく，SDT 転送が十分に行える領域（“ counter clash ”による FIFO ミスを少なくする領域）を用意する必要がある．このように SDT 転送を行う場合には，データキャッシュとパーティション領域の競合（キャッシュミス削減），SDT 転送を十分に行える領域（FIFO ミス削減）を考慮して，再構成キャッシュのパーティション分割を行うことにより，アプリケーションの実行性能を改善可能である．

4.2.3 プリフェッチパーティションの評価

再構成キャッシュと通常キャッシュでプリフェッチを行った場合の比較を行う.

Livermore Kernel#12（first difference）（図 4.9），Livermore Kernel#3（inner product）をプリフェッチを行わない通常キャッシュ（Nomal），プリフェッチを行う通常キャッシュ（conventional prefetch），再構成キャッシュでプリフェッチを行う場合（partition prefetch）で実行を行った結果を図 4.10，図 4.11 に示す．今回の実行では 8 個のブロックを同時プリフェッチできるように，8 セットのプリフェッチパーティション（256byte）を用意した.

```
#include<stdio.h>

#define Loop 10
#define N 4096

int x[4096],y[4096]

int main(void)
{
  int l,k;
  for(l=1;l<=Loop;l++){
    for(k=0;k<=N-1;k++){
      x[k] += y[k+1] - y[k];
    }
  }
  return 0;
}
```

図 4.9: Livermore Kernel#12 コード

	Normal	conventional prefetch	partition prefetch
総clock数	2,078,916	1,811,458	1,373,781
キャッシュ・ミス	10,243	5,124	703
instruction	1,064,859	1,304,182	1,304,184
キャッシュ・アクセス	368,608	468,439	468,439

図 4.10: Livermore Kernel#12 の実行結果

	Normal	conventional prefetch	partition prefetch
総clock数	2,038,085	1,770,538	1,332,663
キャッシュ・ミス	10,244	5,124	701
instruction	942,029	1,181,362	1,181,364
キャッシュ・アクセス	368,618	468,449	468,449

図 4.11: Livermore Kernel#3 の実行結果

二つの結果からもわかるように，プリフェッチを用いることによりキャッシュミスを削減でき，実行性能が改善された．また，再構成キャッシュを用いたプリフェッチを行うことにより，更にキャッシュミスを削減でき，実行性能が改善された．

通常のキャッシュでは，プリフェッチデータをキャッシュに充填する際に，頻繁にアクセスされるデータをリプレースする問題がある．また，充填したプリフェッチデータが，参照される前に他のデータアクセスによってリプレースされる問題がある．一方，本研究で用いる再構成キャッシュでは，キャッシュの一部をプリフェッチパーティションとして再構成することにより，プリフェッチデータによる他のデータの追い出し，またはプリフェッチデータの追い出しが発生しない．その結果，再構成キャッシュを用いた場合では

プリフェッチデータと他のデータアクセスがキャッシュ内で競合しないため，キャッシュミスを大幅に減らすことができ，実行性能が改善された．

また，プリフェッチパーティションでは，パーティションのサイズを小さくするほど，データキャッシュとしての機能を妨げることはなく，キャッシュミスを減らすことができる．そのため，実行プログラムにより，プリフェッチパーティションがプリフェッチ転送を行うのに十分なサイズであると判断できるならば，可能な限りパーティションを小さくすることが有効である．

4.2.4 SDT 転送方式とプリフェッチ機構の比較１

プリフェッチ機構の研究が数多くなされている [8, 9, 10, 11]．しかし，プリフェッチ機構ではブロックサイズでのデータ転送となるため，不必要なデータを転送する可能性がある．

本研究で用いる再構成キャッシュでは SDT 転送方式のデータを受け取ることができるため，この機構を用いて，メモリからキャッシュに必要となるデータのみをプリフェッチすることが可能である．必要なデータのみを一括して連続転送することにより，ブロック単位での転送に比べ，不必要なデータの転送時間を無くすことができる．

SDT 転送方式とプリフェッチ機構を比較するため，同領域のパーティション（256byte）を用意して，Livermore Kernel#3（inner product）を実行した結果を図 4.12 に示す．

	Normal	prefetch partition	FIFO partition (SDT転送)
総clock数	2,038,085	1,332,663	1,285,301
キャッシュ･ミス	10,244	701	683
instruction	942,029	1,181,364	1,064,885
キャッシュ･アクセス	368,618	468,449	368,618
FIFOミス			697

図 4.12: Livermore Kernel#3 の実行結果

Livermore Kernel#21(matrix*matrix product) を実行した結果を図 4.13 に示す．ここで「nomal-partition」はキャッシュからパーティションとなる領域を除いて実行した結果であり，パーティション領域が，データキャッシュとしての機能をどれだけ妨げるかを示す尺度となる．

	Normal	normal -partition 4entry(128B)	normal -partition 8entry(256B)	normal -partition 32entry(1kB)	prefetch機構 32entry(1kB)	SDT機構 4entry(128B)
総clock数	45,233,299	45,233,299	46,904,520	51,869,469	40,739,836	24,556,036
キャッシュ・ミス	274,163	274,163	291,044	341,195	195,378	17,288
instruction	17,291,162	17,291,162	17,291,162	17,291,162	20,597,414	18,491,168
キャッシュ・アクセス	6,032,562	6,032,562	6,032,562	6,032,562	7,588,812	6,032,562
FIFOミス						35,699

図 4.13: Livermore Kernel#21 の実行結果

Livermore Kernel#3 のような連続する配列データをアクセスする場合，再構成キャッシュによるプリフェッチ機構と SDT 機構を比較した結果，図 4.12 のように，SDT 機構の方が少ない実行クロック数で実行を終えた．これはプリフェッチ機構では必要なデータがすべてキャッシュに取り込まれるまで，ブロックサイズでのデータ転送を繰り返すことになるため，ブロックサイズに限定されないデータ転送が行える SDT 機構に比べて，効率が悪いからである．

Livermore Kernel#21 のようなメモリに一定間隔で格納される配列をアクセスする場合，プリフェッチ機構によるブロック単位でのデータ転送を行うと，ブロック内のすべてのデータが必要なデータとなることはなく，時間的および空間的局所性のない不必要なデータも含まれる．また，不必要なデータの転送に要する転送時間が発生するだけでなく，不必要なデータをパーティションに格納する事によってパーティションの有効利用が達成されない問題がある．

さらにプリフェッチパーティションを用いてプリフェッチを行った場合には，パーティションをメモリアドレスのタグ，インデックスを用いてアクセスするため，複数のプリフェッチデータを格納する時には，パーティションのセットが競合する事がある．それにより，プリフェッチしたデータの競合により，データの使用前に追い出される問題がある (図 4.14)．一方，FIFO パーティションはカウンタによるアクセスであるため，パーティションのセットの競合を考える必要がない．したがって，データ同士の追い出しは発生し

ない.

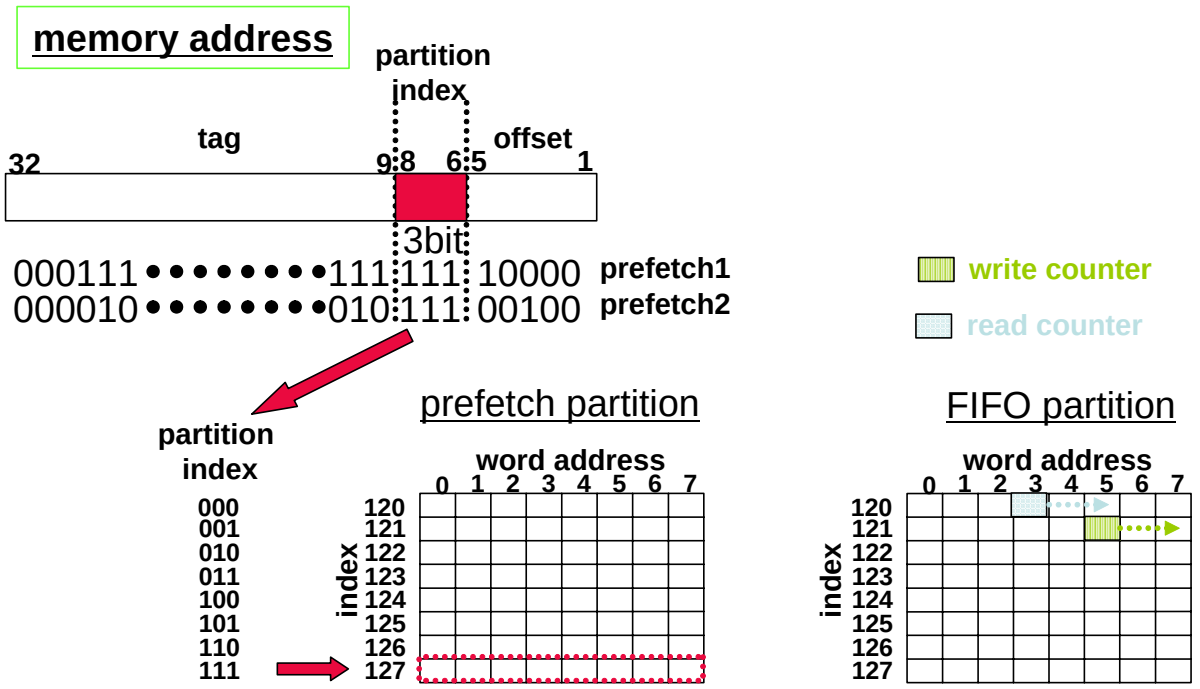

図 4.14: プリフェッチパーティションと FIFO パーティションの比較

Livermore Kernel#21 の実行において，SDT 転送方式では小さいパーティション領域でも十分（128byte）なのに対して，プリフェッチ機構はセットの競合を防ぐためにパーティション領域を大きくしなければならない（1Kbyte）．しかし，図 4.13 の「nomal-partition 4entry」と「nomal-partition 32entry」から，SDT 機構の FIFO パーティションに比べ，プリフェッチパーティションは大きくデータキャッシュとして機能を妨げる．

セットの競合によるパーティションの領域問題，不必要なデータの転送に要する転送時間の発生により，一定間隔値で格納される配列をアクセスする場合には，プリフェッチ機構は SDT 機構と比較し性能が低い．

4.2.5 SDT 転送方式とプリフェッチ機構の比較２

FIFO バッファ，再構成キャッシュを用いて SDT 転送方式を実現する研究がなされている [2, 7]．しかし，この機構は一定のデータ間隔値を持つデータアクセスの場合にのみ有効な転送方式である．また，SDT は同一 Bank，Row アドレス内のデータを連続転送するものであり，どちらかのアドレスが変化した場合には中断する．仮想記憶によるページ境界を超える場合にも，仮想アドレス空間に対する物理アドレスの連続性が保障されないため，転送が中断する問題，プリフェッチ機構では転送されたデータに対して読み出し，書き込みの両方ができるのに比べ，SDT 方式では読み出すのみで，書き込みを行うことができない問題がある．

SDT 転送方式とプリフェッチ機構の比較するため，Livermore Kernel#12（first difference）を実行した結果を図 4.15 に示す．パーティション領域は共に 256byte（8entry）である．

	Normal	FIFO partition (SDT転送)	prefetch partition
総clock数	2,078,916	2,479,664	1,373,781
キャッシュ･ミス	10,243	10,243	703
instruction	1,064,859	1,187,715	1,304,184
キャッシュ･アクセス	368,608	368,608	468,439
FIFOミス		2,602	

図 4.15: Livermore Kernel#12 の実行結果

SDT 転送方式では一つの連続するデータ列に対してのみ転送を行うのに対して，プリフェッチ機構では複数のデータを同時にプリフェッチできる利点がある．すなわち，Livermore Kernel#12 のような計算に必要なオペランドが複数あった場合，同時に複数のオペランドをプリフェッチできる．また，Livermore Kernel#12 では計算に必要なオペランドが，k の値によって変化する連続した配列 y[k]，y[k+1] のため，一回のブロック転送で二つのオ

ペランドのプリフェッチが可能である．しかし，SDT 転送の場合には一つのオペランドの転送のみが行われる．

よって図 4.15 に示す結果のように，SDT 機構はプリフェッチ機構に比べて性能が低い．また SDT 機構は，“ Nomal ”で示される通常の実行よりも遅くなった．“ Nomal ”で示される通常の実行の場合でも，ブロック転送になるため，二つのオペランドのデータを同時にアクセスできる．逆に，SDT 転送の場合は一つのオペランドをワード単位で転送を行いパーティションに格納し，同一メモリブロックに存在するもう一つのオペランドを通常アクセス（ブロック転送）でキャッシュに格納する．そのため，SDT 転送を行う事により，転送時間が増え，実行クロックを増やす結果となった．

このように SDT 転送方式，プリフェッチ機構を比較した結果，互いに異なる利点と欠点を持っている．本研究で用いる再構成キャッシュでは，両方の機能を備えることにより，用途に応じてこの二つの機能を使い分けて高速化を達成する．

第5章　関連研究

キャッシュを動的に複数のパーティションに分割させて，性能向上を図る reconfigurable キャッシュの研究がある [1]．例えば，一つのパーティションを FIFO バッファとして利用することにより，大規模なデータアクセスに対応させ，データアクセスの性能を向上させる提案がなされている [2]．また，連想キャッシュの一部のウェイをロックすることで，ハードウェアによるリプレースメントを制御し，必要なデータがキャッシュから追い出されるのを防ぐ機構が提案されている [3]．

しかし，これらは再構成の際に，セットアソシアティブ方式のウェイ単位で分割を行うため，パーティションのサイズが固定される問題がある．本研究では reconfigurable キャッシュのパーティション分割の際に，ウェイより細かい単位での分割を可能にする．これを利用して，様々なアプリケーションに対して適切なサイズのパーティションを提供する．

Khairuddin ら [2] は 2 ウェイセット・アソシアティブキャッシュを用い，ウェイ単位で FIFO パーティションに再構成を行うキャッシュを構築している．すなわち再構成を行った際には，一つのパーティション（1 ウェイ）を通常のデータキャッシュに，一つのパーティション（1 ウェイ）を FIFO パーティションとして使用している．しかし，この方法で実行を行うと，データキャッシュとして使用可能なのは半分の領域のみとなるため，FIFO の利用以外の通常のデータアクセスを妨げ，実行性能を低下させる可能性がある．また，FIFO パーティションとして使うキャッシュ領域の追い出しを行う際にも，半分のキャッシュ領域を調べ，更新されたデータの追い出しを行うために大きなオーバヘッドがかかる．

本研究では SDT 転送方式をウェイより細かい領域で実行を行った（4.2.2 節）結果，実行性能が向上することが示された．

再構成キャッシュの FIFO パーティション分割と同様に，連続するデータをターゲットとした stream buffer[14, 15] の研究がある．連続するデータを，データキャッシュとは別のバッファ（stream buffer）に転送し，プロセッサとメモリ間の動作速度差を隠蔽する．しかし，この stream buffer の機構では，ブロックサイズでのデータ転送となるため，ストライドデータの使用時には不必要なデータを転送する可能性がある．本研究で用いる再構成キャッシュでは SDT 転送方式のデータを受け取ることができるため，この機構を用いて，メモリからキャッシュに必要となるデータのみを転送することが可能である．必要なデータのみを一括して連続転送することにより，ブロック単位での転送に比べ，データの転送時間が短縮可能である．また，不必要なデータをバッファに格納することによってバッファの有効利用が達成されない問題も，本研究で用いる再構成キャッシュでは解決している．

近年，様々なプリフェッチに関する研究がなされている [8, 10, 11]．その中でも，データキャッシュとは別にプリフェッチバッファを用いて，プロセッサとメモリ間の動作速度差を隠蔽する手法が提案されている [9]．プリフェッチバッファを備えることにより，本論文の 4.2.3 節で示したプリフェッチパーティションと同様に，プリフェッチデータと他のデータアクセス間でのキャッシュ内競合を回避することが可能となる．

しかし，プリフェッチバッファでは，プリフェッチ転送が行われた時のみバッファを利用するため，プリフェッチが行われないときにはバッファは使用されない．本研究で用いる再構成キャッシュでは，プリフェッチ転送を行う時のみ再構成を行い，パーティションを用意する．プリフェッチバッファと異なり，パーティションが使用されない時は再構成を行わず，パーティション領域は通常キャッシュとして振る舞う．そのため，プリフェッチバッファのような使用されない領域が存在しない．したがって効率の良いハードウェア構成となる．

第6章　結論

本論文では，キャッシュを動的に複数のパーティションに分割させて，性能向上を図るreconfigurable キャッシュを利用し，アプリケーションごとの要求に対し，可変にキャッシュを対応させ，メモリアクセスの性能を向上させる方式を提案した．

reconfigurable キャッシュのパーティション分割の際に，タグ，比較器の拡張によりウェイより細かい単位での分割を可能にし，パーティションのサイズが固定されることなく，様々なアプリケーションに対して適切なサイズのパーティションを提供できた．

リアルタイムアプリケーションを実行する際に，複数のタスクの実行をマルチスレッド型のプロセッサを用いることにより，スループットを向上させることができた．また優先度を考慮したパーティション分割では，複数のタスクをマルチスレッドで実行した場合に，優先度の高いタスクが使えるキャッシュ領域を保障することにより，優先度の高いタスクの実行性能を向上させた．

SDT 転送を行う場合には，データキャッシュとパーティション領域の競合（キャッシュミス削減），SDT 転送に十分な領域（FIFO ミス削減）を考慮して，パーティション分割を行うことにより，アプリケーションの実行性能を改善できた．

データプリフェッチ機構では通常キャッシュを用いた場合，プリフェッチデータをキャッシュに充填を行う際に，頻繁にアクセスされるデータをリプレースする問題がある．また，プリフェッチ転送したデータが参照される前に他のデータアクセスによってリプレースされる問題がある．これを解決するために，本研究ではキャッシュの一部をプリフェッチパーティションとして再構成することにより他のデータの追い出し，プリフェッチデータの追い出しを制御する．それにより，再構成キャッシュを用いた場合ではプリフェッチデータと他のデータアクセスがキャッシュ内で競合しないため，キャッシュミスを大幅に減らすことができ，実行性能が改善された．

SDT 転送方式，プリフェッチ機構共にメモリからキャッシュへのデータプリフェッチの機能を備えている．SDT 転送方式，プリフェッチ機構を比較した結果，互いに異なる利点と欠点を持っている．本研究で用いる再構成キャッシュでは，両方の機能を備えることにより，用途に応じてこの二つの機能を使い分けて，お互いの欠点を補い，実行性能が改善できた．

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.

その他貴重なご意見，討論を頂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.

最後に日頃よりあたたかく見守ってくださった丹研の岩田君に深く感謝致します.

参考文献

[1] P.Ranganathan, S.Adve and N.P.Jouppi, “Reconfigurable Caches and their Application to Media Processing.” In Proc. of ISCA, pp.214-224, 2000.

[2] Khairuddin bin Khalid and Kiyofumi Tanaka, “Implementation of FIFO Buffer Using Cache Memory.” IPSJ SIG Notes, ARC, Vol.2002, No.112, pp.83-88, 2002.

[3] D.Chiou, P.Jain, S.Devadas, and L.Rudolph, “Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches” Technical Report CGS-Memo 427 , MIT, 1999.

[4] SPARC International, Inc.:The SPARC Architecture Manual Version 8, Prentice-Hall, Inc, 1992.

[5] SPARC International, Inc.:The SPARC Architecture Manual Version 9, Prentice-Hall, Inc, 1994.

[6] D.M.Tullsen et al., “Simultaneous Multithreading:Maximizing On-Chip Parallelism.” In Proc. ISCA-22, pp. 392-403, June, 1995.

[7] 府川智治, 田中清史, 宮崎純, “主記憶データベース向け高機能メモリコントローラの実現方式” 情報処理学会研究報告 ARC, Vol.2002, No.112, pp.77-82.

[8] E.Gornish, E.Granston, and A.Veidenbaum. “Compiler-directed data prefetching in multiprocessors with memory hierarchies.” In Int. Conf. Supercomputing, pages 354-368,1990.

[9] R.L.Lee. “The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors.” PhD thesis, University of Illinois at Urbana-Champaign, May 1987.

[10] T.Mowry and A.Gupta. “Tolerating latency through software-controlled prefetching in shared-memory multiprocessors.” J.Paral.Distr.Computing,to appear in June 1991.

[11] A.K.Porterfield. “Software Methods for Improvement of Cache Performance on Supercomputer Applications.” PhD thesis, Rice University, May 1989.

[12] http://gcc.gnu.org/

[13] 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.

[14] N.P.Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." In Proc. of the 17th ISCA, pages 364-373, May 1990.

[15] K.I.Farkas, N.P.Jouppi and P.Chow. "How Useful Are Nonblocking Loads, Stream Buffers,and Speculative Execution in Multiple Issue Processors?" In Proc. of the 1st HPCA,pages 78-89,Jan. 1994.