修 士 論 文

主記憶データベースを対象とした高機能メモリアーキテクチャに関する研究

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

府川 智治

2003 年 3 月

修 士 論 文

主記憶データベースを対象とした高機能メモリアーキテクチャに関する研究

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　堀口進 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

110110 府川 智治

提出年月: 2003 年 2 月

概 要

近年，プロセッサと主記憶間の性能格差の拡大およびデータの大規模化により，データベースの応答時間が増大し，質問処理の高速化が重要となっている．一方で半導体技術の発達により主記憶装置である DRAM が大容量化，低価格化してきており，主記憶内にデータベースを格納する主記憶データベースの実現が可能になってきた．本論文では主記憶データベースにおけるデータ構造と DRAM のハードウェア特性を利用した高速大規模データ転送方式を提案し，シミュレーションによりそれらの評価を行う．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景と目的

近年データの大規模化により，データベースの応答時間が増大してきており，質問処理の高速化が重要になっている．一方，半導体技術の発達により主記憶装置である DRAM が大容量化，低価格化し，従来はディスク上に格納していた大量のデータを主記憶内に格納する主記憶データベース（Main Memory Database System，MMDB）[1] の実現が可能になってきた．MMDB はディスク格納型データベースに比べて高速なデータアクセスが可能であり，高いリアルタイム性が要求されるデータベースシステムに使用される．

データベースシステムを高速化する要因として，

- データベースの質問処理時間
- 障害時のログファイルによるデータ回復時間
- トランザクションの安全性を確保するために定期的に取るバックアップ時間

などが挙げられるが，本研究ではデータベースの質問処理時間を短縮化する手法に着目する．

データベースの質問処理時間にはプロセッサのデータ参照時間とデータベース演算時間が含まれる．データ参照時間は与えられた質問に対してプロセッサがメモリから検索するタプルを読み出す時間であり，データベース演算時間は主に条件に一致したデータを抽出するための実行時間である．

大規模なリレーションに対して特定の属性を走査する場合，読み出す属性データのデータアクセス間隔が大きく，しかもアクセスする属性データは条件比較のための一度しか使用されない．そのためメモリ内では空間的局所性，および時間的局所性が共に存在せず，プロセッサが有するキャッシュを有効に機能させることが困難である．

また，MMDB 方式ではメモリ使用量を削減するためにプロセッサはメモリ上のデータをアクセスするときポインタを介したデータアクセスを行う．しかしこの際，データの実体だけでなくそのポインタもキャッシュに格納されるため，キャッシュのヒット率が低下する．そしてヒット率の低下はメモリアクセスのオーバヘッドを増大させ，質問処理時間に影響する．

本論文ではデータ参照時間を削減するために，メモリ空間に一定間隔で存在するデータをメモリからプロセッサに連続して転送する方式と，ポインタを介した二段階のアクセス

に対してプロセッサが見かけ上一回のメモリアクセスでデータを取得する方式の 2 つの方式を提案する．また，これらを実現する機構をメモリコントローラ（以下，MC）に組み込み，データ参照時間を最小限に抑えることをシミュレーションで示す．

1.2 本論文の構成

本論文の構成を以下に示す．

第 2 章 MMDB の特徴および MMDB におけるリレーショナルデータベースのリレーション構成について述べる．

第 3 章 本研究で提案する二つのデータ転送方式とその性能見積もり結果ついて述べる．

第 4 章 提案したデータ転送方式を実現する MC の仕様および動作を説明する．

第 5 章 質問処理を実行するシミュレーション環境を説明し，評価対象リレーションとそれを用いた質問処理の種類について述べる．

第 6 章 評価対象リレーションのデータ構造を決定し，各質問処理を実行したサイクル数を従来方式と比較した結果を考察する．

第 7 章 本研究の関連研究について報告する．

第 8 章 まとめと今後の課題について述べる．

第2章　主記憶データベース（MMDB）

従来のデータベースシステムはディクス格納型データベースであるのに対し，MMDB は主記憶内にリレーションを格納するため，従来のディスクを使用したデータベースシステムと比較し高速なデータアクセスが可能である．また，主記憶内にデータを格納することでランダムアクセスが発生した場合でもアクセス時間を低下させることなく，一定時間内に大量のデータを処理することができるため高速リアルタイム処理に適している．

ただし主記憶に使用される DRAM は揮発性であるため，障害によるデータやトランザクションの消失を防ぐために，定期的にディスクなどの補助記憶装置にバックアップを取る必要がある．

図 2.1 に MMDB 方式のリレーショナルデータベースのリレーション構成を示す．

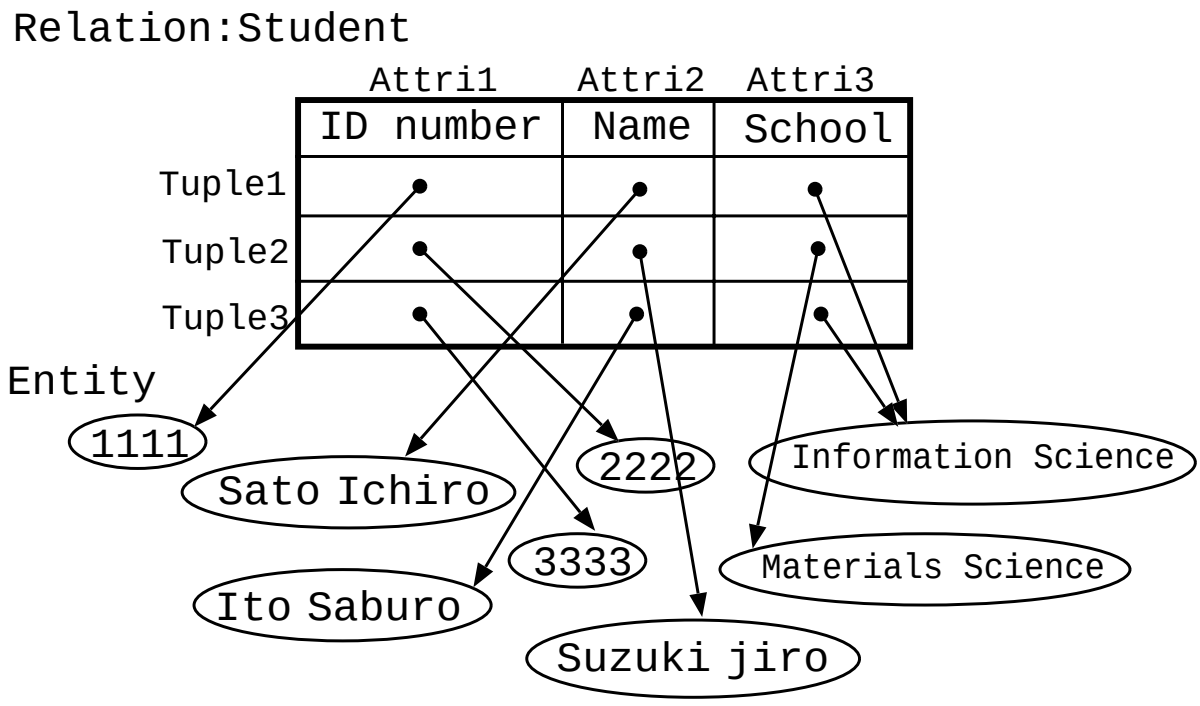

図 2.1: MMDB 方式のリレーション構成

メモリ内に展開されたリレーション内の各セルには実体へのアドレス（ポインタ）が格納され[1]，複数のセル間で共通する実体は一つのみ存在する（同図における Tuple1，3 の School）．これによりサイズの大きい共通データがリレーション内に頻出する場合，各セルにその共通アドレスを格納することでメモリ資源の効率化が可能である．なお，メモリ内で実体はリレーションとは別の領域に格納される．

このことから，質問処理を実行する場合，ポインタ比較によってデータの一致／不一致を判定することが可能であるが，データの大小関係などは実体同士の比較によってのみ得られるため，実体へのポインタを取得しそのポインタを使って実体へアクセスする必要がある．

[1]ただしポインタサイズ以下のデータはリレーションに直接データを格納することでメモリ資源を節約する．

第3章　データ転送方式

本章ではデータアクセス時間の短縮のためにDRAMのハードウェア特性を考慮した高速データ転送方式であるStride Data Tranfser方式と，MMDBによるポインタを介した二段階アクセスを高速化するTwo-Phase Data Transfer方式を提案する．

3.1　Stride Data Tranfser（SDT）方式

リレーションを格納する主記憶装置としてはDRAM（Dynamic RandomAccess Memory）の使用が主流となっている．

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3.1**）．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．

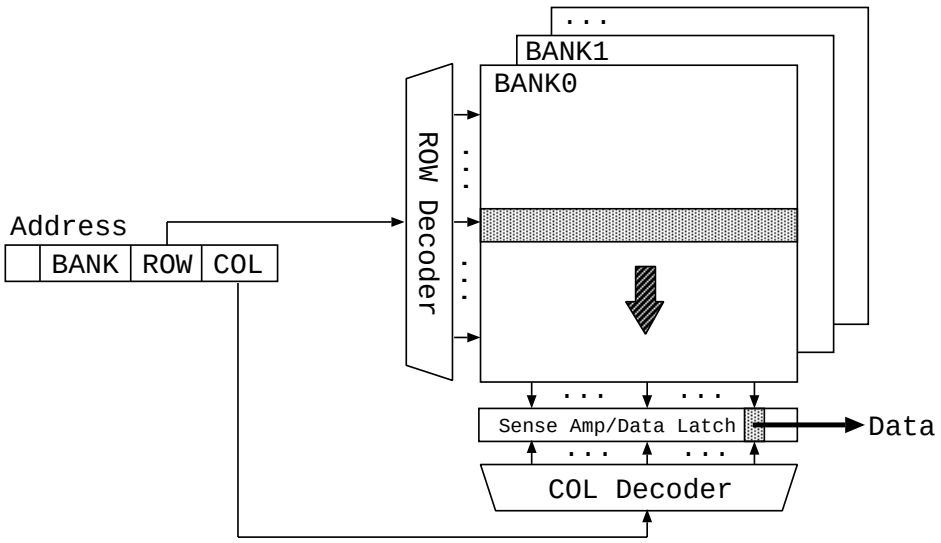

図 3.1: DRAMアクセス

ここで，Bank，Row アドレスを指定した状態，複数の Col アドレスを連続して与えることにより該当するデータが連続して出力される．しかし MMDB において同一 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 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．

本論文で提案する Stride Data Transfer（SDT）方式は同一 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．

３つの属性を持ったタプルが４つで構成するリレーションをタプルの集合で管理する場合，物理アドレス空間では**図 3.2** のようなメモリ配置になる．

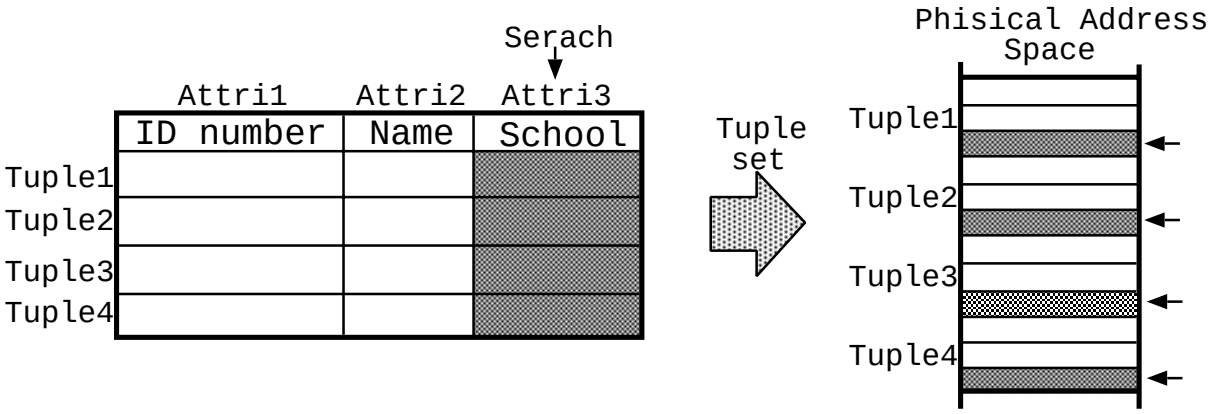

図 3.2: リレーションのメモリ配置

ここで属性 School で Tuple1 から Tuple4 を条件探索する場合，一つのタプルのデータサイズが一定間隔に相当し，その一定間隔毎のメモリアクセスが発生する．

このとき，MC は最初のタプルの属性 School データを Bank，Row および Col アドレスを与えることにより読み出すが，それ以降の同一 Bank，Row アドレスに存在する各タプルの属性 School データは対応する Col アドレスのみの指定で読み出す．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 3.3** 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ためデータ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を向上させる．

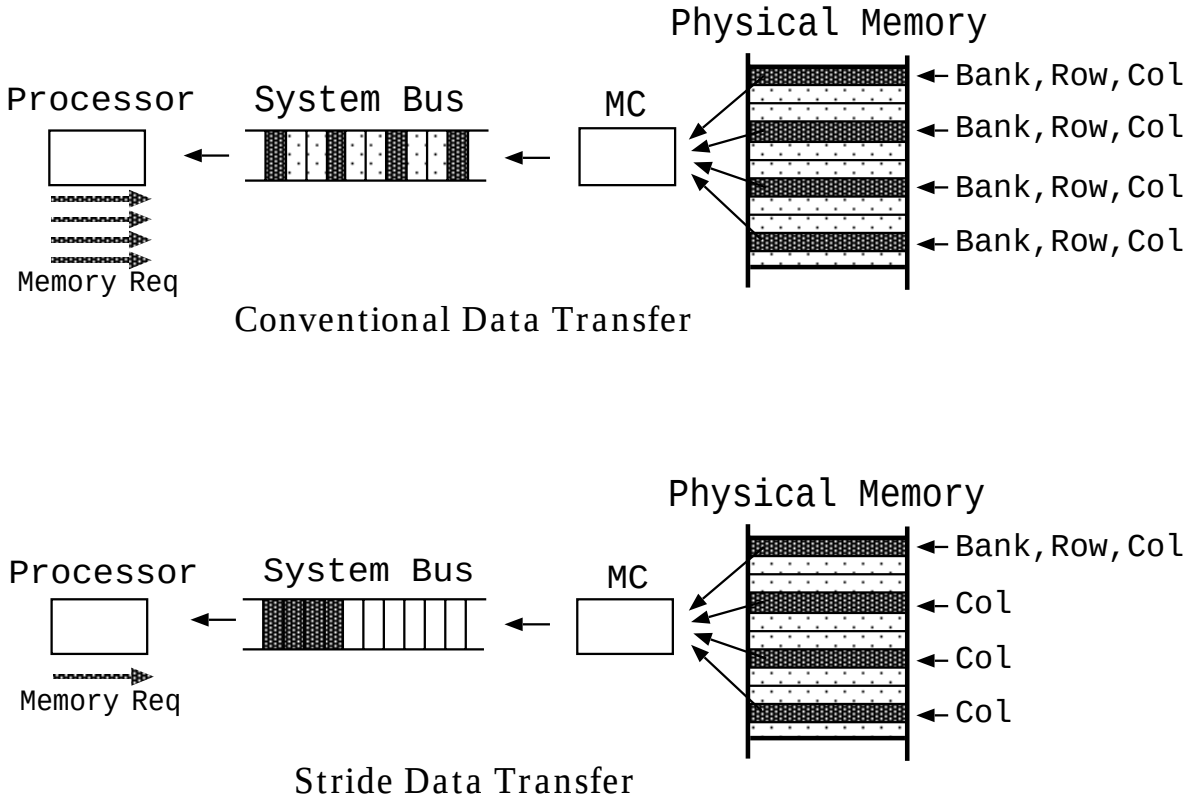

図 3.3: SDT 方式によるデータ転送

SDT 方式の性能見積もり

SDT 方式の性能見積もりとして同一 Bank， Row アドレス内に存在する複数データを読み出す場合のメモリアクセス時間を算出し，従来方式と SDT 方式でそのサイクル数を比較する．

1 メモリアクセス時間 （バスクロックサイクル数）は，プロセッサのメモリリクエスト発行から該当するデータを読み出し，プロセッサで処理するまでの時間とする．このサイクル数は本研究で設計した MC に基づいた結果，バスクロックサイクルで 12 サイクルを必要とする．

同一 Bank，Row アドレス内の複数データを読み出す場合，従来の方式では各データに対して毎回 Bank， Row アドレスを指定するため 1 データ当たり 1 メモリアクセス時間が必要となる．SDT 方式では最初のデータに対して Bank，Row アドレスを与えるため 1 メモリアクセス時間が必要となるが，それ以降の各データは Col アドレスのみを指定するため 1 データにつき 1 バスクロックサイクルでデータを読み出すことができる．

以下の二つの例はリレーションの 1 タプルのデータサイズを 60byte として、その一定間隔に存在する属性データ（1 データ当たり 4byte）をタプルの条件探索のためにメモリから読み出してプロセッサに転送するまでのバスクロックサイクル数を従来方式と SDT 方式で比較する．

例 1） 1Row のデータサイズが 1024byte で 60byte 間隔に存在する合計 17 個のデータを読み出す場合のバスクロックサイクル数

表 3.1: SDT 方式の性能見積もり結果 -1-

従来方式	$12 \times 17 = 204$ cycle
SDT 方式	$12 + (17 - 1) = 28$ cycle
性能向上	7.3 倍

従来方式では 17 個のデータに対して各 1 回のメモリアクセス時間を要するため合計 17 回のメモリアクセス時間が必要となり，データを全て読み出す場合 204 バスクロックサイクルになる．

SDT 方式では最初のデータのみ 1 メモリアクセス時間を必要とするが，残り 16 個のデータに関しては 1 データにつき 1 バスクロックサイクルでデータの読み出しが可能になるためデータを全て読み出す場合 28 バスクロックサイクルになる．結果として SDT 方式は従来方式に比べ最大 7.3 倍のデータ読み出しが可能になる．

例 2）1Row のデータサイズが 2048byte の場合

表 3.2: SDT 方式の性能見積もり結果 -2-

従来方式	$12 \times 34 = 408$ cycle
SDT 方式	$12 + (34 - 1) = 45$ cycle
性能向上	9.1 倍

1Row のデータサイズを 2 倍にした場合読み出すデータ数が 34 になるため，結果として SDT 方式は従来方式に比べ 9.1 倍のデータ読み出しが可能になる．

1Row のデータサイズが増加した場合に SDT 方式の性能向上率が向上するが，現在の半導体技術では 1Row 当たり 2048byte のデータサイズが最大になっている．

3.2 Two-Phase Data Tranfser（TPDT）方式

MMDB ではプロセッサが主記憶上の実体データにアクセスするとき，メモリ資源の効率化を図るために実体へのポインタを介した二段階のメモリアクセスを実行する．プロセッサとメモリの速度差が大きくなっている現在の計算機システムにおいて，メモリアクセス時間を短縮することが重要課題である．またリレーションの属性を走査する場合，プロセッサは一回の条件比較で時間的・空間的局所性のない実体データとその実体へのポインタもキャッシュに格納する．そのためヒット率が低下しその結果メモリアクセスが増大する．

Two-Phase Data Transfer（TPDT）方式はこの二段階のメモリアクセスを回避するためにプロセッサからは見かけ上，一回のメモリアクセスでデータを取得するデータ転送方式である．この方式は実体データ（例えば，文字列データ）同士の大小比較を行う場合，あるいはポインタテーブルを介した条件探索を行う場合などで使用される．なおポインタテーブルによる条件探索は 6.7 節で述べる．

従来の MC を介する方式と TPDT 方式の比較を**図 3.4** に示す．TPDT を実現する MC は第一段階のメモリアクセスでポインタを取得し，その値を使用して第二段階のメモリアクセスを行い，読み出した実体データをプロセッサに転送する．この間ポインタはプロセッサに転送されない．これによりデータアクセスレイテンシが減少し，ポインタがキャッシュに格納されないためキャッシュの使用効率が向上する．

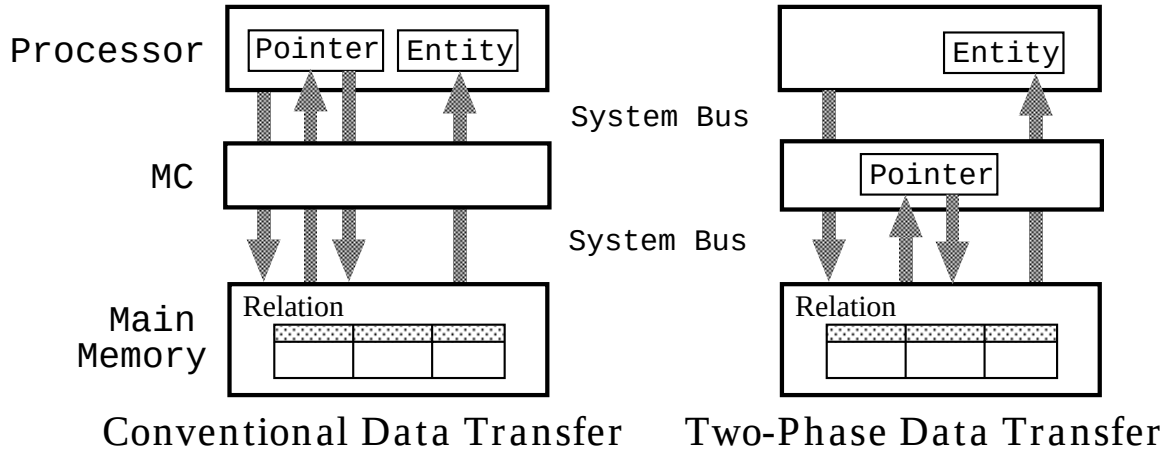

図 3.4: TPDT 方式によるデータ転送

TPDT 方式の性能見積もり

ここではプロセッサがメモリから実体データを読み出すまでのメモリアクセス時間を算出し，従来方式と TPDT 方式でそのサイクル数を比較する．1 メモリアクセス時間は 3.1 節で述べたバスクロックサイクル数とする．

表 3.3: TPDT 方式の性能見積もり結果

従来方式	$12 \times 2 = 24$ cycle
TPDT 方式	20 cycle
性能向上	1.2 倍

従来の方式ではプロセッサが実体データへのポインタを読み出すためのメモリアクセスで 12 バスクロックサイクルが必要になる．次に，そのポインタで実体データを取得するメモリアクセスでさらに 12 サイクルを要するため 24 バスクロックサイクルになる．

TPDT 方式では見かけ上一回のメモリアクセスで実体データを取得するため，MC は実体データへのポインタをプロセッサには転送せずに次のメモリアクセスを行う．そのため本研究の MC では 20 バスクロックサイクルで実体データを読み出し，結果として TPDT 方式は従来方式に比べ 1.2 倍のデータ読み出しが可能になる．

第4章　メモリコントローラの実装

本章では提案した転送方式を実現するMCの基本仕様とSDT方式およびTPDT方式によるMCの動作について述べる．

4.1　概要

本論文で提案したデータ転送方式を実現するMCを設計し，その基本仕様とブロック図を**図 4.1** に示す．

MCの基本仕様

- 32bitアドレスバス
- 32bitデータバス
- バースト長（連続して入力または出力可能なデータ数）· · · 1，2，4，8に設定可能
- CASレイテンシ（Colアドレス入力後，実際データが読み出されるまでの必要サイクル数）· · · 3バスクロックサイクル

同図においてMCは**Command Interface**，**Address Generator**，**Command Generator**，**Data Path** の4つのモジュールから構成される．

Command Interface，Command GeneratorおよびData Pathは従来のMCに存在するモジュールであり，Address Generatorモジュールの追加によってSDTおよびTPDTを実現する．各モジュールの機能について述べる．

- **Command Interface**
  プロセッサからRead，Writeなどのメモリアクセス要求（REQ）とアドレスストローブ信号（ADS）およびメモリアドレス（ADR）を受け取り，DRAMの動作コマンドとアクセスモード（通常・SDT・TPDTアクセス）を決定し，Command Generatorに送る．
- **Address Generator**
  Stride Address Generator（SDT AG）とPointer Address Generator（TPDT AG）

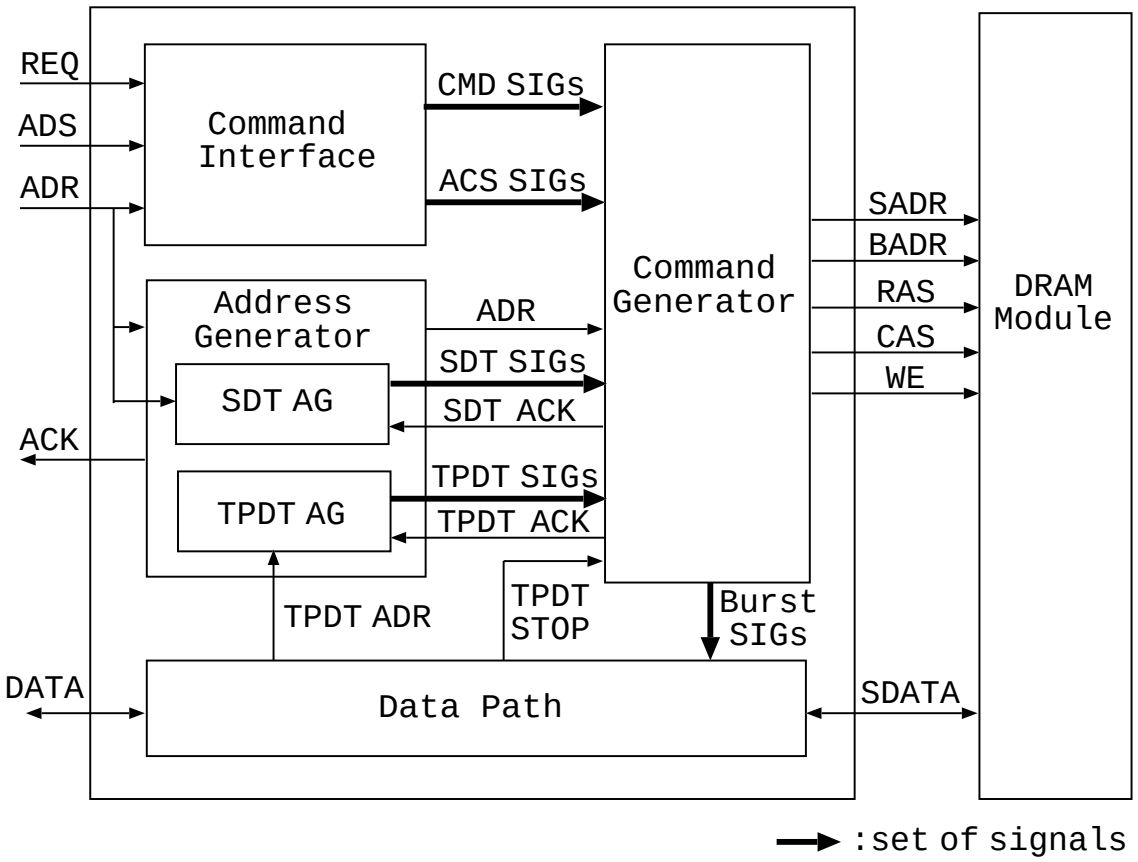

図 4.1: メモリコントローラのブロック図

モジュールを内蔵し，それぞれ SDT および TPDT アクセスを実現するメモリアドレスを生成する．詳細は 4.3 節で述べる．

- **Command Generator**
  Command Interface からのコマンドとアクセスモード，および Address Generator で生成されたアドレスを受け取り，DRAM への制御信号を生成し適切なタイミングで送る．メモリアドレス（SADR）は Row アドレスと Col アドレスで共用するため，それらのアドレスは RAS 信号と CAS 信号で区別する．また SDT 転送の終了判定はこのモジュールで行う．

- **Data Path**
  プロセッサと DRAM 間のバースト転送および TPDT 転送の終了判定を行う（4.3 節）．TPDT 転送は文字列型データの終りを示す NULL 文字を検出することで終了する．

4.2 メモリアドレス形式

SDT および TPDT 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 2bit をメモリアクセスモードの指定フィールド（AMODE）とする．**図 4.2** にメモリアドレス形式，**表 4.1** に AMODE フィールド情報を示す．AMODE フィールドの値はコンパイラあるいはリンカのアドレスリロケーションおよび，仮想記憶（アドレスの変換）機構が協調することにより指定される．

また AMODE フィールドはメモリアクセスの種類を判別するためのフィールドであり，実際の SDT 方式および TPDT 方式のメモリアクセスは AMODE フィールドを除いた Col，Row，Bank アドレスで DRAM アクセスを行う．

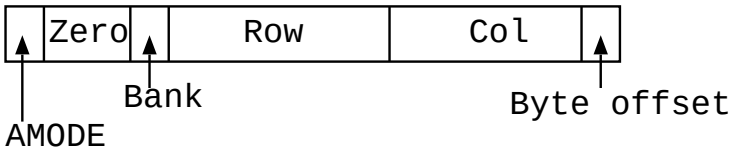

図 4.2: メモリアドレス形式

表 4.1: AMODE フィールド

アクセスの種類	AMODE
通常のメモリアクセス	00
SDT 方式のメモリアクセス	01
TPDT 方式のメモリアクセス	10
Reserved	11

4.3 プロセッサと MC の協調動作

本節では通常のリードリクエストによる MC の動作と SDT および TPDT 方式におけるプロセッサと MC の動作を述べる．

4.3.1 リードの動作

通常，プロセッサのリードリクエストは命令実行中に命令キャッシュミス，ロード・ストア命令によるデータキャッシュミスによってメモリアクセスが発生する．この時のプロ

セッサのリクエストからデータを読み出すまでのタイミング波形を**図 4.3** に示す.

リードのタイミング波形

（1）ではプロセッサからアドレスストローブ信号（ADS），メモリアドレス（ADR）およびリードリクエスト（REQ）がシステムバスを通じて MC に送られる.（2）では MC はこれらの信号をデコードして Bank アドレス（BADR），Row アドレス（SADR）および RAS 信号を生成し DRAM にアクセスする．DRAM への制御信号によって DRAM 内のメモリアレイの該当する ROW データがラッチされる.（3）では CAS 信号と共に Col アドレス（SADR）を与える.（4）では Col アドレスを与えてから CAS レイテンシ後に DRAM から該当するデータが SDATA を通じて読み出され，MC に送られる．この時バースト長（Burst Length）をキャッシュのブロック数に設定することで一回のメモリアクセスでキャッシュブロック分のデータ転送が可能になる．今回バースト長を 4 に設定したためバスクロックの 1 サイクル毎に 4 つのデータが MC に転送される.（5）では MC は読み出された 4 つ分のデータを DATA を通じてプロセッサに転送し、その後処理を終了する.

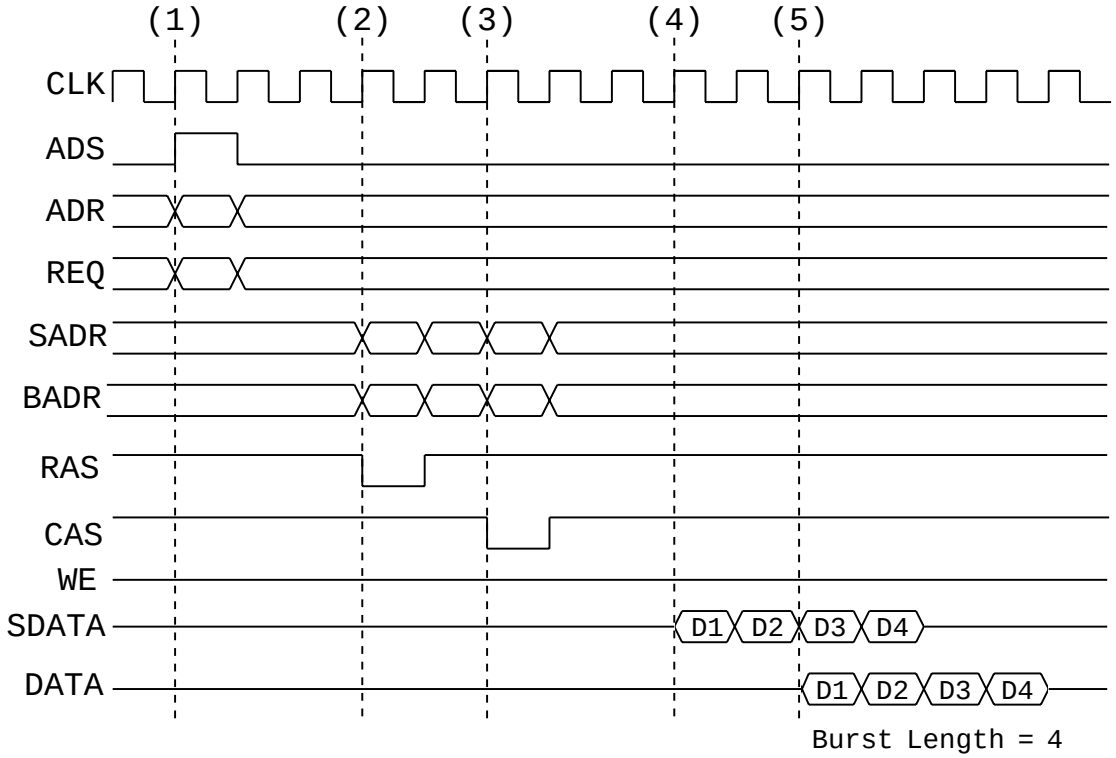

図 4.3: リードのタイミング波形

4.3.2 SDT 方式の動作

SDT 転送は一定間隔に存在するデータを連続転送するため，MC 内部の SDT AG でメモリアドレスを自動生成し DRAM アクセスを行う．このときのプロセッサと MC の動作を述べ，そのタイミング波形を**図 4.4** に示す．

まず，プロセッサはキャッシュスルー命令で SDT 開始前に SDT AG のレジスタにマップされたアドレスに書き込みを行うことにより，転送するデータ数およびデータ間隔値を格納する[1]．リレーションを走査する場合，転送するデータ数をリレーションの総タプル数，データ間隔値を 1 タプルのデータサイズとする．

そしてプロセッサは AMODE フィールド（4.2 節）を “01” 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 AMODE フィールドをデコードし，SDT の開始アドレスとして SDT AG 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，Data Path を通じてプロセッサへ転送する．プロセッサ内の再構成可能なキャッシュの一部を FIFO バッファとして利用し，転送されたデータが順次格納される [3]．プロセッサは FIFO バッファから転送されたデータ，すなわち属性データを読み出しタプルの条件比較する．データ転送の終了条件は，

（a）MC のレジスタにセットした転送データ数を越える場合

（b）データ間隔値の加算で Bank，Row アドレスが変化する場合

（c）仮想記憶によるページ境界を越える場合

である．(a）はリレーションの条件探索が終了する場合であり，(b）に関して，SDT は 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 アドレスから与え直すことで SDT を再開する．(c）のページ境界での終了は仮想アドレスに空間に対する物理アドレスの連続性が保証されないため，この場合もアドレスを与え直すことで SDT を再開する．

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．

プロセッサは FIFO がアンダーフロー状態でデータを読み出そうとする場合，データがまだ到着していないためストールする必要がある．このときそのデータに対するアドレスが MC に対して送られているため，MC 内部でそのアドレスを無効化する．一方，FIFO

[1]更にデータサイズを格納することにより任意のデータサイズの SDT が可能となるが，本設計では簡単化のためこれを省略し，SDT でアクセスするデータをポインタ（ワード）サイズに限定する．

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 内のデータが消費されるまで SDT は再開されない．

また FIFO としての使用範囲を広げるためキャッシュの次ブロックを利用する際にそのブロックへの書き込みがあった（dirty ビットがセットされている）場合，そのブロックをライトバックしてから使用する．このような場合も同様に SDT を一時中断する．

SDT 方式のタイミング波形

本 MC における SDT 方式のタイミング波形を下図に示す．MC 内のレジスタに転送するデータ数とデータ間隔値を格納する部分の波形は省略した．データ転送数を 8，データ間隔値を 60 として既にセットされているとする．まず，(1) プロセッサから ADS 信号，AMODE フィールドを “01” とするアドレス（ADR）およびリードリクエスト（REQ）が発行され，MC に送られる．(2) MC はそれらをデコードし BADR，SADR および RAS 信号を生成する．(3) バスクロックサイクル毎に CAS 信号と共に間隔値が加算された Col アドレス（SADR）の連続入力によって DRAM アクセスを行う．このとき転送したデータ数を記憶しておく．(4) DRAM から MC にデータが転送され（SDATA），(5) MC からプロセッサに転送される（DATA）．(6) MC にセットされたデータ数分の入力を終了する．(7) データ数分のデータを転送後に終了する．

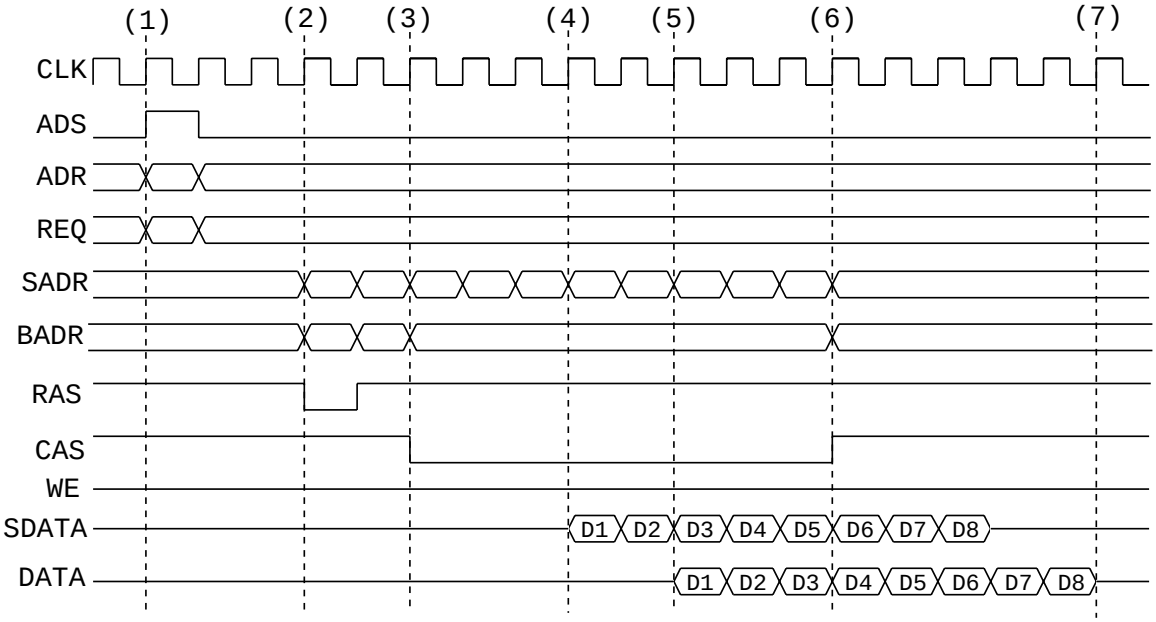

図 4.4: SDT 方式のタイミング波形

次に SDT 転送中に命令あるいはデータキャッシュミスによるメモリアクセスが発生し、SDT を一時停止してから再開するまでのタイミング波形を**図 4.5** に示す。

SDT 方式のタイミング波形（SDT の一時停止）

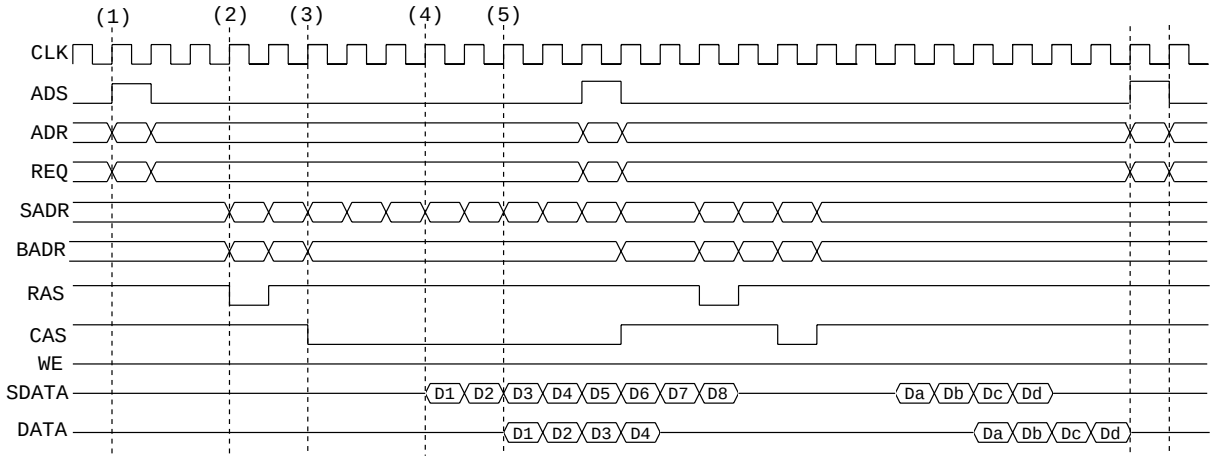

図 4.5: SDT 方式のタイミング波形（SDT の一時停止）

4.3.3 TPDT 方式の動作

本 MC では，実体として文字列型のデータを扱い，文字列型データの TPDT 転送におけるプロセッサと MC の動作を述べ，そのタイミング波形を**図 4.6** に示す.

まず，プロセッサから AMODE が “10” のアドレスでメモリリクエストを発行し，MC はメモリ内のリレーションから実体へのポインタを読み出す．読み出されたポインタは Data Path を通じて TPDT AG 内のレジスタに記憶される．次にポインタの値は仮想アドレスであるため，TPDT AG 内の MMU で物理アドレスに変換され，Command Generator を介して DRAM に送られ，文字列型の実体データが読み出される．このとき実体データは 32bit データバスを通じて先頭から 4 文字（1 文字当たり 1byte）が一度に転送される．さらに次の 4 文字分をひとまとまりとしたデータがバスクロックの 1 サイクル毎にバースト回数分連続して MC に転送される[2].

バースト長より大きい実体データを転送する場合，読み出されたポインタを TPDT AG 内のレジスタで保存するため，そのポインタにバースト長を加算したアドレスを生成し，DRAM にアクセスすることで実体データが連続して MC に転送される．TPDT 転送中は MC 内の TPDT ステータスレジスタを TPDT 転送状態として記憶し，実体データは Data Path を通じてプロセッサ内の FIFO バッファに転送される．TPDT 方式におけるデータ転送の終了条件は

[2]バースト長 4 の場合，バスクロックの 4 サイクルで計 16 文字の実体データが転送可能.

（a）転送された実体データに NULL 文字が含まれている場合

（b）文字列の比較途中で探索条件に一致しないとわかる場合

である．(a）は Data Path で転送された 4 つの実体データに文字列の終了を表す NULL 文字が含まれている場合である．(b）に関して，プロセッサは FIFO バッファから 1 文字ずつ実体データを読み出しタプルの条件探索を行うが，このとき文字列の比較途中で探索条件に一致しない場合，そのタプルに関してその後の文字列を探索する必要がないため転送を終了する．プロセッサは次タプルの実体データが転送されるまでに FIFO バッファをクリアし，MC では発行された前タプルのメモリアクセス要求を無効化する．(a）, (b）のいずれかの終了条件を満たした場合，TPDT ステータスレジスタに TPDT 転送終了状態として記憶する．

TPDT 転送中は文字列の連続転送でシステムバスを占有するため，SDT 転送と同様なメモリアクセスが発生する場合，あるいは MC 内の MMU でページフォルトが発生する場合に TPDT を中断し TPDT ステータスレジスタを TPDT 停止状態にする．プロセッサ内の FIFO バッファが空になり，TPDT 再開アドレスが MC に送られ，ステータスレジスタを TPDT 転送状態として再開する．

TPDT 方式のタイミング波形

本 MC における TPDT 方式のタイミング波形を下図に示す．MC のバースト長を 4 とし，文字列型の実体データは 32byte（NULL 文字含む）とする．(1）プロセッサから ADS 信号，AMODE フィールドを “10” とするアドレス（ADR）およびリードリクエスト（REQ）が発行され，MC に送られる．(2）MC はそれらをデコードして実体へのポインタを取得するため DRAM への制御信号（BADR，SADR および RAS 信号）を生成する．(3）CAS 信号と共に Col アドレスを DRAM に与える．(4）MC は DRAM から実体へのポインタを読み出し，その内部の MMU で仮想アドレスを物理アドレスに変換する．(5），(6）MC は実体データを読み出すためにその物理アドレスで再度メモリアクセスを行う．(7）で実体データは DRAM から MC に転送される．(8）実体データはバースト長より大きいため MC 内部で生成されたアドレスで Col アドレスを再入力する．(9）その実体データはプロセッサに転送され，(11）NULL 文字を検出し転送を終了する．(10）Col アドレスが入力されているため DRAM から該当する実体データに連続したデータが読み出されるがプロセッサには転送しない．

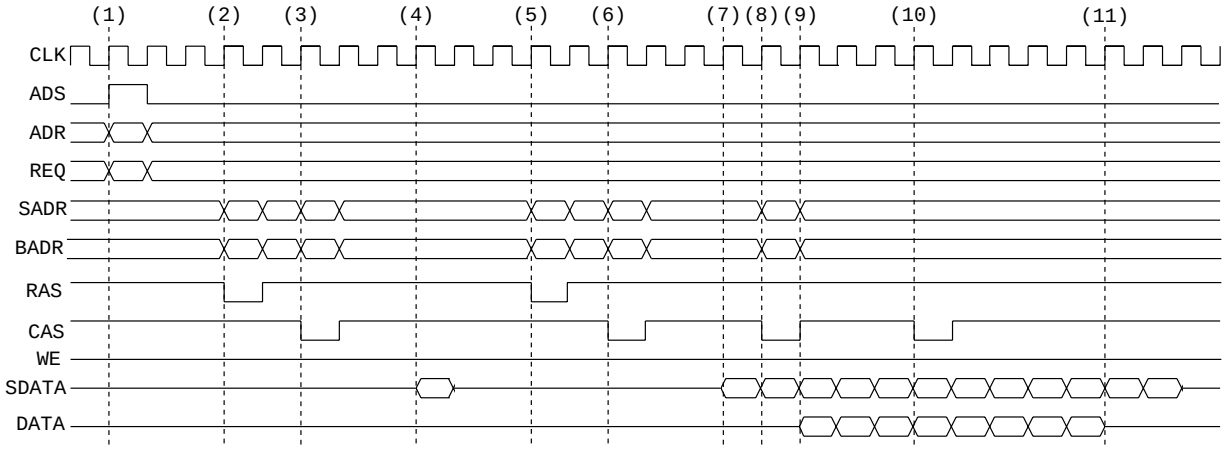

図 4.6: TPDT 方式のタイミング波形

第5章　性能評価

本章では質問処理を実行するためのシミュレータ環境と評価対象のリレーションおよび評価する質問処理の種類について述べる．

5.1　シミュレーションモデル

C 言語で記述された質問処理をコンパイルして生成されたアセンブリコード（SPARC 命令セット）を入力とするシミュレータを作成した．1 命令実行を 1 クロックサイクルとし，総実行サイクル数を求める．命令，データキャッシュはそれぞれ 8KB（2 ウェイセットアソシアティブ）である．キャッシュのブロックサイズを 32byte として 1 ブロックを 8 ワードで構成する．またメモリのバースト長を 8 に設定しキャッシュのブロックサイズに合わせることで 1 回のメモリアクセスでブロックを埋める[1]．キャッシュヒット時は 1 サイクルで読み書き可能であるが，ミス時のペナルティは 120 サイクルとした[2]．今回二次キャッシュは考慮していない．

SDT 処理および TPDT 処理に関する所要サイクル数として，実際に VHDL で設計した MC における動作サイクル数を使用した．これらを以下に示す．

（a）SDT 転送の前処理（転送数と間隔値の格納）・・・各 40 プロセッササイクル

（b）SDT 転送の開始または再開時における初回データの読み出し・・・120 プロセッササイクル

（c）TPDT 転送における実体データの読み出し・・・200 プロセッササイクル

（d）FIFO バッファ内データの読み出し（ヒット時）・・・1 プロセッササイクル

（e）FIFO バッファ内データの読み出し（ミス時）・・・1〜10 プロセッササイクル

（f）1 ブロック（8 ワード）のライトバック・・・140 プロセッササイクル

[1]ただし，バスクロックの 1 サイクル毎にデータがキャッシュに格納されるため計 8 バスクロックサイクル必要である．

[2]1 メモリアクセス時間が 12 バスクロックサイクルであり，これを 10 倍の動作周波数を仮定したプロセッサのクロックサイクルで換算した．

（a）に関して，MC 内部のレジスタにそれぞれの値を格納するまでのサイクル数で，プロセッサがアドレスおよび値を送出してから MC の書き込み終了を表す ACK 信号が返ってくるまでを仮定した．(b)，(c）はプロセッサがメモリアドレスを送出してメモリからデータを読み出し，そのデータを使用するまでのプロセッササイクル数とする．実際はこれらの値より少ないプロセッササイクル数でプロセッサに転送されるがデータを使用するまでにかかるオーバヘッドを考え過大に設定した．(d）はプロセッサの命令実行に MC から転送されたデータが間に合っている場合のプロセッササイクル数である．通常のキャッシュヒットと同じプロセッササイクル数である．(e）に関してはプロセッサの命令実行に転送されたデータが間に合っていない場合のサイクル数である．このときプロセッサはデータの到着までストールする．このサイクル数はデータアクセス命令の実行タイミングと SDT の動作状況で決まり，プロセッサのストールは最大でプロセッサとメモリバスの周波数比の 10 サイクルとなる．(f）はキャッシュブロックをライトバックするために必要なプロセッササイクル数で，ライトバック中に SDT および TPDT におけるメモリアクセスが発生する場合はライトバックが終了するまでプロセッサはストールする．

TPDT 方式における MC 内での仮想–物理アドレス変換のサイクル数は未実装のため考慮していない．

シミュレーション対象のメモリは SDRAM を想定し，1Row のデータサイズは 4KB とする．このデータサイズは OS が管理するページサイズと同じと仮定する．

5.2 FIFO バッファ[3]

本節では，提案したデータ転送方式によってメモリから転送されるデータを格納する FIFO バッファの概要とその動作について述べる．詳しくは [3] を参照されたい．

5.2.1 概要

通常のキャッシュは一般目的としたデータ処理において有効であるが，データベースや media-processing などのアプリケーションでは大規模なデータセットを扱い，そのデータは再使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しない場合に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キャッシュを複数のパーティションに分割し目的に合わせて利用するキャッシュを再構成可能なキャッシュと呼んでいる．

FIFO バッファはこの再構成可能なキャッシュのあるパーティションを FIFO として利用し，メモリから転送されたデータを格納する．再構成可能な単位はセットアソシアティブキャッシュのウェイとするため，FIFO バッファのサイズは最大で 1 ウェイ分になる．このとき転送されたデータは最上位ブロックから格納される．2 ウェイセットアソシアティブキャッシュのパーティション分割を**図 5.1** に示す．

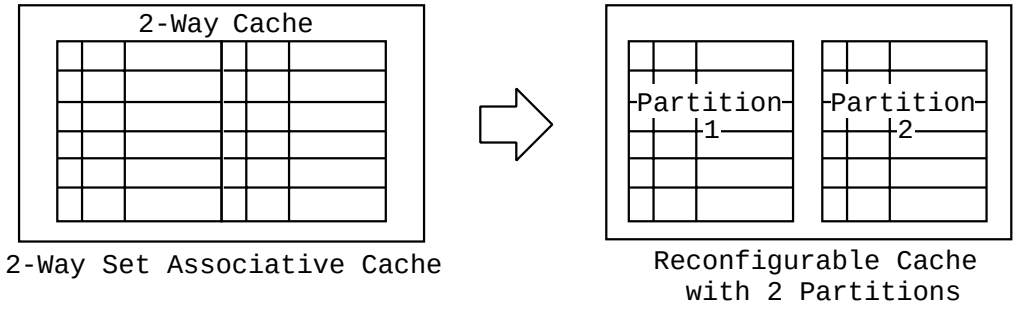

図 5.1: 2 ウェイキャッシュのパーティション分割

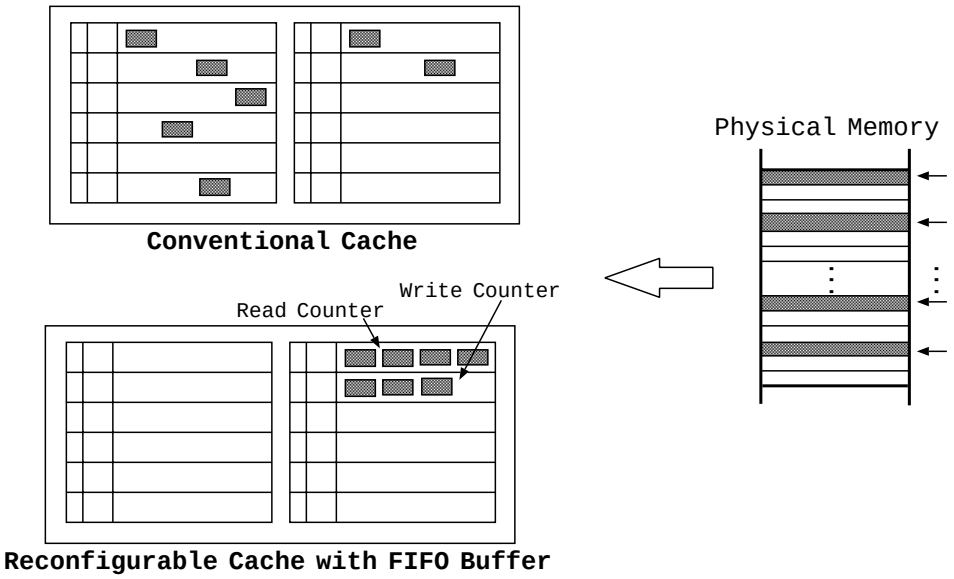

図 5.2: SDT 方式による FIFO バッファの使用

5.2.2 提案したデータ転送方式との協調動作

ここでは SDT 転送を例にプロセッサが FIFO に格納される一定間隔のデータを読み出す場合の処理動作を述べる．**図 5.2** に通常キャッシュと FIFO バッファを使用したキャッシュを比較する．

(1) プロセッサは SDT 転送の前処理として MC 内のレジスタに割り当てられたアドレスでデータ転送数とデータ間隔値を書き込む．

(2) プロセッサはタプルのデータに対してロード命令を発行するが，FIFO バッファは空であるため FIFO ミスとして上位ビットを “01” としたアドレスで MC にメモリアクセスを要求する．

（3）MC はプロセッサから要求されたアドレスから Col アドレスを生成し，DRAM から連続してデータを読み出す．間隔値の加算によって Bank，Row アドレスが変化した場合，あるいはページ境界を越えた場合に連続転送を終了する．

（4）読み出されたデータは書き込み位置を示すライトカウンタの場所に格納してカウントアップする．プロセッサは読み出し位置を示すリードカウンタの場所からデータを読み出しカウントアップする．両カウンタが同じ位置を示そうとした場合，それは FIFO バッファに転送されるデータが到着していないことを示しており，プロセッサはストールする．このとき MC に対してメモリアクセス要求が発行されているため，MC ではそのロード命令を無効化する．またライトカウンタがブロックの境界を越える場合，そのブロックのデータに書き込みがあった場合にライトバックが発生するため SDT 転送を中断する．

通常のロード・ストア命令によるメモリアクセスで SDT 転送が中断した場合，プロセッサは FIFO バッファ内のデータ読み出しを続け，やがて FIFO が空になり次第（2）以降の手順で MC にセットした転送数に達するまで継続する．

（1）～（4）の処理で FIFO バッファはロード命令で必要なデータをプリフェッチすることで実際のメモリアクセス時間を削減している．

5.3 評価対象リレーション

評価対象のリレーションを**表 5.1** に示す．これは Wisconsin Benchmark[2] の一部で，一つのタプルは 15 個の属性を持っており，13 個の属性（unique1 から oddOnePercent）は整数型の実体を，残りの 2 個の属性（string1 と string2）が文字列型の実体へのポインタを表す．

属性 “unique1”，“unique2”，“unique3” はそれぞれ 0 から MAXTUPLES-1 のユニークでランダムな値を割り当てる．各属性の情報として左から属性名，値の範囲，順番，選択率を表す．順番とは各属性の値（例えば属性 “four” において 0，1，2，3 の値）のそれぞれが出現する順番であり，random はランダム，sequential は昇順を意味する．また選択率とは各属性の値が出現する割合であり，後の評価において全タプル数に対して質問処理で選択抽出されるタプル数の割合を表す．つまり属性 “four” でリレーションを条件探索する場合，全タプル数の 25%が選択抽出される．なお MAXTUPLES は全タプル数を示す．

属性 “string1”，“string2” に関しては後の評価において適宜文字列のポインタを割り当てる．

5.4 リレーションの質問処理

リレーションの質問処理は単一または複数のリレーションに対して探索条件を指定し，その条件に一致するタプルを抽出して結果リレーションを作成することである．質問処理

表 5.1: リレーションの仕様

Attribute Name	Range of Values	Order	Selectivity
unique1	0-(MAXTUPLES-1)	random	unique
unique2	0-(MAXTUPLES-1)	sequential	unique
two	0-1	random	50%
four	0-3	random	25%
ten	0-9	random	10%
twenty	0-19	random	5%
onePercent	0-99	random	1%
tenPercent	0-9	random	10%
twentyPercent	0-4	random	20%
fiftyPercent	0-1	random	50%
unique3	0-(MAXTUPLES-1)	sequential	unique
evenOnePercent	0, 2, 4, ..., 198	random	1%
oddOnePercent	1, 3, 5, ..., 199	random	1%
string1	-	cyclic	20%
string2	-	cyclic	20%

は**図 5.3** のように分類できる[3].

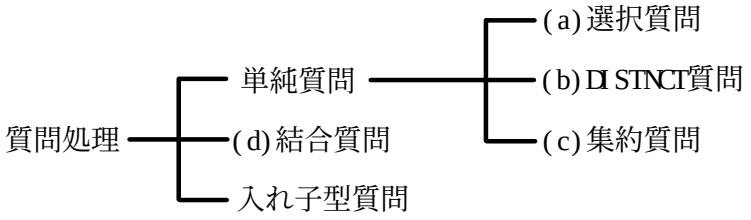

図 5.3: 質問処理の種類

単純質問は選択質問，DISTNCT 質問，集約質問などがあり，**SELECT** 文の **FROM** 句に一つのリレーションを指定する．結合質問は単純質問のリレーション指定を複数にした場合の質問である．入れ子型質問は **SELECT** 文の **WHERE** 句中に **SELECT** 文が入る質問である．

今回の評価は 5.3 節のリレーションを用いて選択質問，DISTNCT 質問，集約質問および結合質問を実行し，SDT 方式および TPDT 方式の効果を確認する．それぞれの質問の説明と適用する転送方式について述べる．なお今回は入れ子型質問は評価しない．

（a）選択質問（Selection Queries）

[3]図のようなタプルの検索以外にタプルの挿入・削除・修正処理があるが評価対象外とする．

(i) `SELECT * FROM R WHERE two = 1`

選択率 50%の属性 two でリレーション R 内の全タプルを条件探索する．つまり 10 万のタプル数とし「属性 two=1」で条件探索した場合タプルが 5 万件選択され，結果リレーションが作成される．この結果リレーションは一致したタプルの先頭アドレスのみを格納し，結果を出力する際には該当の属性名を先頭アドレスから求める．属性 two はリレーション内に整数型の実体を直接格納しているため SDT 方式を適用する．後の評価においても同様に属性 four，ten，twenty に対して SDT 方式を適用する．

(ii) `SELECT * FROM R WHERE string1 = 'xxxxx'`

任意の文字列型データ 'xxxxx' でリレーション R 内の全タプルを条件探索する．このときの結果リレーションにはこの文字列に一致するタプルの先頭アドレスが格納される．本来文字列の一致／不一致はポインタ比較で判定するが，敢えて大小比較で判定する[4]．属性 string1 に対して TPDT 方式を適用する．

（b）DISTNCT 質問（DISTINCT Queries）

```
SELECT DISTINCT four FROM R
```

この質問処理はリレーション R から属性 four の値（すなわち 0，1，2，3）が重複しているタプルを除去し，結果リレーションにはその属性の値 0，1，2，3 が格納される．なおこの値の順番で出現するとは限らないため格納順はリレーションの探索順になる．この属性 four はリレーション内に整数型の実体を格納しているため SDT 方式を適用する．

（c）集約質問（Projection Queries）

```
SELECT four, COUNT(*) FROM R
GROUP BY four
```

GROUP BY 句の指定で属性 four の値をグループごとに分け，その出現回数をカウントする COUNT 関数を用いた集約を実行する．つまり 10 万のタプル数で実行した場合，各値はランダムに出現するがその出現回数はそれぞれ 2.5 万回になる．結果リレーションには属性 four の値とその出現回数を格納する．この属性 four はリレーション内に整数型の実体を格納しているため SDT 方式を適用する．

（d）結合質問（Join Queries）

(i)
```
SELECT * FROM R, S
WHERE R.unique1 = S.unique1
```

[4]後の評価において文字列の種類や長さを変更して実験した際に，一致したタプル数が明確になり実行サイクル数が比較しやすいため．

リレーション R，S の属性 unique1 の値が一致するタプルをそれぞれ抽出する結合処理を行う．リレーション S のタプル数を R の 1/10 とした場合，結合属性 unique1 はユニークであるため結果のタプル数は S と同じ数になる．また結果リレーションには両リレーションで一致したタプルの先頭アドレスをそれぞれ格納する．属性 unique1 の値は整数型の実体であるため SDT 方式を適用する．

(ii)
```
SELECT * FROM R, S
WHERE  R.unique2 = S.unique2
AND    R.two = 1
AND    S.two = 1
```

この質問処理は結合質問に複数の条件を指定する場合である．まずリレーション R，S に対してそれぞれ属性 two で条件探索し，一時的な結果リレーション（以下，ポインタテーブル）を作成する．このポインタテーブルは一致したタプルの先頭アドレスを格納し，結合処理をするために使用する．そして各ポインタテーブルを辿り結合属性 unique2 で条件探索し，最終的な結果リレーションを作成する．ポインタテーブルを使用する結合処理は一致したタプルの先頭をアクセスして結合属性を読み出すという 2 段階のメモリアクセスが必要になるため TPDT 方式を適用する[5]．

[5]ポインタテーブルの作成は各リレーションの属性 two に対して SDT 方式を適用できるが，ここでは TPDT 方式のみの効果を確認する．

第6章　評価結果

本章では評価するリレーションのデータ構造を決定し，各質問処理で提案した転送方式を適用して実行サイクル数を求め，従来方式と比較する．

6.1　リレーションのデータ構造

後の評価においてリレーションのデータ構造を決定することは重要である．ここではリレーションをタプルの集合で管理し，タプルおよびタプルの属性は連続領域に確保される必要があるため，リレーション（relationR，relationS）は属性をメンバとする構造体で表し，その構造体配列で定義する（**図 6.1**）．リレーションのサイズは配列の大きさ（MAXTUPLES_R，MAXTUPLES_S）で決定する．タプルの各属性は**表 5.1** リレーションの仕様から，属性 unique1 から oddOnePercent までは属性値を直接格納し，属性 string1，string2 は文字列型の実体へのポインタを格納する．文字列の実体はリレーションと別の領域で確保する．一つのタプルのデータサイズは 15(属性数) $\times$ 4($byte$) = 60$byte$ になる．

6.2　SDT 方式を適用する選択質問

本節で評価する質問処理は直接リレーションに格納されている属性値に対して条件探索する選択質問を対象とする．SELECT 文の WHERE 句で指定する属性で選択率を変更し，従来型 MC 方式と SDT 方式による実行サイクル数を比較する．評価する選択質問を以下に挙げる．

（a）`SELECT * FROM R WHERE two = 1`

（b）`SELECT * FROM R WHERE four = 1`

（c）`SELECT * FROM R WHERE ten = 1`

（d）`SELECT * FROM R WHERE twenty = 1`

（a）の選択質問に関して従来型 MC 方式と SDT 方式で実行する場合の質問処理コードを比較する．

従来型 MC の質問処理コード

```
/* リレーションの定義 */
typedef struct {
  int unique1;
  int unique2;
  int two;
  int four;
  int ten;
  int twenty;
  int onePercent;
  int tenPercent;
  int twentyPercent;
  int fiftyPercent;
  int unique3;
  int evenOnePercent;
  int oddOnePercent;
  char *string1;        /* 文字列型の実体へのポインタ */
  char *string2;        /* 文字列型の実体へのポインタ */
} Tuple;

Tuple relationR[MAXTUPLES_R]; /* リレーション R の構成 */
Tuple relationS[MAXTUPLES_S]; /* リレーション S の構成 */
```

図 6.1: リレーションのデータ構造

```
j = 0;
for( i_R = 0; i_R < MAXTUPLES_R; i_R++ ){  /* タプルの繰り返し処理    */
  if(relationR[i_R].two == 1){             /* タプルの条件探索        */
    rslt_ptr[j++] = &relationR[i_R];       /* 結果リレーションの作成 */
  }
}
```

この質問処理コードは**図 6.1** で定義したリレーション（relationR）を全タプル数（MAX-TUPLES）回繰り返すループで走査する．構造体で表現したタプルのメンバである属性 two が探索条件を満たす場合そのタプルの先頭アドレスを結果リレーション（rslt_ptr）に書き込む．従来方式では 1 タプルのデータサイズがキャッシュのブロックサイズより大きいためタプルの条件探索毎にデータキャッシュミスによるメモリアクセスが発生する．

SDT 方式の質問処理コード

```
int *sdt_addr;        /* SDT 方式で参照する属性のアドレス */
```

```
SET_TIMES(100);       /* 転送数をレジスタに格納          */
SET_STRIDE(60);       /* 間隔値をレジスタに格納          */
j = 0;
/* relationR の繰り返し処理 */
for( i_R = 0; i_R < MAXTUPLES_R; i_R++ ){
  /* relationR の属性読み出し */
  sdt_addr = (unsigned int)(&tplarryR[i_R].two) | 0x40000000;
  if( *std_addr == 1 ){                    /* タプルの条件探索       */
    rslt_ptr[j++] = &relationR[i];         /* 結果リレーションの作成 */
  }
}
```

この質問処理コードはリレーションのタプル数を 100 に設定して SDT 方式で実行する場合である．前処理として MC の内部レジスタに割り当てられたアドレスでデータ転送数（タプル数の 100）および間隔値（タプルサイズの 60byte）を書き込む．SDT 転送を属性 two への参照で使用するために，上位 2 ビットに “01” を加えた仮想アドレス sdt_addr（ここでは 32 ビット）を指定する．これによりタプルの条件探索では属性 two の値がメモリから連続して転送され FIFO バッファに格納される．

実行サイクル数

表 6.1 に従来型 MC 方式と SDT 方式で（a）〜（d）の選択質問の WHERE 句で指定する属性とリレーションのタプル数を変更して実行したときの総サイクル数を示す．SDT 方式では従来型 MC 方式と比較し実行サイクル数の 76.0〜90.0%を削減することができた．

従来方式では総サイクル数の 82〜93%がキャッシュミスに伴うメモリアクセスに費やされていた．これは 1 タプルのサイズがキャッシュのブロックサイズよりも大きいため空間的局所性が活かされず，タプル数分のメモリアクセスが発生していることに起因する．これに対し SDT 方式では従来方式と同数のメモリアクセスが発生するが，プロセッサの命令実行と並行してメモリから読み出された属性値が FIFO バッファに格納されるため，見かけ上のメモリリクエスト回数およびメモリアクセス時間を削減している．

タプル数が 100 で属性 two で条件探索した場合，100 個の属性値を FIFO バッファから読み出した時のデータ数とそのサイクル数の内訳は，

- SDT 転送を開始した最初のデータ · · · 1 個（120 サイクル）
- キャッシュミスによって SDT 転送が中断されたデータ数 · · · 8 個（960 サイクル）
- FIFO バッファでヒットしたデータ数 · · · 56 個（56 サイクル）
- FIFO バッファでミスしたデータ数 · · · 35 個（89 サイクル）

表 6.1: 選択質問の実行サイクル数（SDT） -1-

タプル数	方式	(a) 属性 two	(b) 属性 four	(c) 属性 ten	(d) 属性 twenty
100	normal	14,058	13,623	13,338	13,202
	sdt	3,368	2,636	2,273	2,035
	性能向上率 (%)	76.0	80.7	83.0	84.6
1K	normal	145,195	136,256	130,955	129,396
	sdt	26,939	19,840	15,309	13,983
	性能向上率 (%)	81.4	85.4	88.3	89.2
10K	normal	1,465,628	1,368,372	1,309,764	1,290,556
	sdt	337,036	218,622	146,662	132,670
	性能向上率 (%)	77.0	84.0	88.8	89.7
100K	normal	14,671,896	13,688,616	13,096,228	12,903,928
	sdt	3,540,907	2,343,212	1,630,325	1,392,444
	性能向上率 (%)	75.9	82.9	87.6	89.2

となっており，SDT 方式で 100 個のデータを読み出す場合に 1 データ当たりの平均読み出しサイクル数は,

$$(120 + 960 + 56 + 89)/100 = 12.25\,(\text{サイクル})$$

となる．つまり従来方式で 1 データ当たり 120 サイクル必要であるのに対し，SDT 方式では 12.25 サイクルで読み出していた.

次にタプル数を固定して（a）の選択質問に複数の探索条件を指定し読み出す属性数を変更した場合の実行サイクル数を比較する．評価する選択質問を以下に挙げる．ここでは（h）において最大 4 つの属性値を読み出す.

（e）`SELECT * FROM R WHERE two = 1`

（f）`SELECT * FROM R WHERE two = 1 OR four = 1`

（g）`SELECT * FROM R WHERE two = 1 OR four = 1 OR ten = 1`

（h）`SELECT * FROM R WHERE two = 1 OR four = 1 OR ten = 1 OR twenty = 1`

WHERE 句中の **OR** 条件は指定した属性のうちどれか一つ条件を満たす場合にタプルが選択されるため，これらの探索条件を複数指定した選択質問において，最初の属性が条件を満たさない場合に次の属性で条件探索する，という処理をする.

実行サイクル数

表 6.2: 選択質問の実行サイクル数（SDT） -2-

タプル数	方 式	(e) 属性数:1	(f) 属性数:2	(g) 属性数:3	(h) 属性数:4
10K	normal	1,465,628	1,561,101	1,655,193	1,750,708
	sdt	337,036	1,159,992	1,255,497	1,353,489
	性能向上率 (%)	77.0	25.7	24.1	22.7

表 6.2 は従来 MC 方式と SDT 方式で（e）～（h）の選択質問においてタプル数を 10K に固定して WHERE 句で指定する探索条件の属性数を増やしたときの実行結果である．

1 つの属性に対してタプルを検索する場合に SDT 転送の効果は大きいが，2 つ以上の属性に対しては効果が小さくなる．これは SDT 転送のメモリアクセスは 1 つの属性に対してのみ適用可能であり，残りの属性に対しては通常のメモリアクセスを行う必要があるためである[1]．しかし 4 つの属性検索の場合に 1 つの属性に対してのみ SDT 転送を適用することで実行サイクル数の 22.7%を削減する結果が得られた．

6.3 SDT 方式を適用する DISTINCT 質問

本節で評価する質問処理は SDT 方式を適用する DISTINCT 質問を対象とし，直接リレーションに格納されている属性値に対して SDT 方式を適用した場合の実行サイクル数を従来方式と比較する．評価する DISTINCT 質問を以下に挙げる．

```
SELECT DISTINCT four FROM R
```

この質問処理はリレーション R から属性 four の値が重複しているタプルを除去する．結果リレーションには属性 four 値（0，1，2，3 の値）が格納される．タプルの属性値を参照する毎に結果リレーションを走査してユニークな属性値のみを格納する．

次に DISTNCT 質問における従来型 MC の質問処理コードを以下に示す．

従来型 MC の質問処理コード

```
j = 0;
/* relationR の繰り返し処理 */
for( i_R = 0; i_R < MAXTUPLES_R; i_R++ ){
  tmp = tplarryR[i_R].four;                /* relationR の属性読み出し */
  /* 結果リレーションの走査 */
  for( i = 0; i <= j; i++ ){
    /* 属性値が重複する場合 */
```

[1]探索条件数が少ない場合においては，各探索条件でリレーションを走査しそれぞれの結果をまとめるという方針で実行した場合に従来方式より性能が向上する可能性はあるが，今回評価していない．

```
    if( tmp == rslt_val[i] )              /* タプルの条件探索          */
      break;
    /* 結果リレーションの作成 */
    else if( rslt_val[i] == NULL ){
      rslt_val[j++] = tmp;
      break;
    }
  }
}
```

この質問処理コードはリレーション R を MAXTUPLES 回繰り返す．属性値の読み出し毎に結果リレーション（rslt_val）を走査してその属性値がユニークであるかを判定し，結果リレーションを作成する．結果リレーションは NULL で初期化しておく．また読み出した属性は一時領域（tmp）に格納し不要なメモリアクセスを減らしている．

SDT 方式の質問処理コード

前処理として MC 内のレジスタにデータ転送数と間隔値を書き込む（コードは省略）．SDT 転送を適用するために従来の処理コードの “relationR の属性読み出し” 部分を

```
int *sdt_addr;        /* SDT 方式で参照する属性のアドレス */
sdt_addr = (unsigned int)(&tplarryR[i_R].four) | 0x40000000;
tmp = *sdt_addr;
```

に変更し，アドレスの上位 2 ビットを “01” とした仮想アドレス sdt_addr で参照する．これによりリレーション R の属性 four の値が FIFO バッファに格納される．

実行サイクル数

表 6.3 に従来型 MC 方式と SDT 方式でリレーションのタプル数を変更して実行したときの総サイクル数を示す．SDT 方式では従来型 MC 方式と比較して実行サイクル数の 73.0〜79.4%を削減することができた．

従来方式では実行サイクル数の 77〜81%がキャッシュミスに伴うメモリアクセスに費やされていた．これは選択質問を従来方式で実行した場合と同様に 1 タプルのデータサイズがキャッシュのブロックサイズより大きいためタプルの属性読み出し毎にデータキャッシュミスによるメモリアクセスが発生してたことに起因する．

SDT 方式では従来方式と同様にタプル数分のメモリアクセスが発生しているが，SDT 転送中においてメモリアクセスによる中断がない限り，プロセッサが属性値を読み出し，条件判定をしている間に次の属性値が FIFO バッファに格納されている．つまりプロセッサの命令実行に並行した属性読み出しを行い，FIFO バッファに格納することで実際のメモリアクセス時間を隠している．しかし，SDT 転送の中断を考慮すると FIFO バッファか

表 6.3: DISTINCT 質問の実行サイクル数（SDT）

タプル数	アクセス方式	総サイクル数
100	normal	15,522
	sdt	4,184
	性能向上率 (%)	73.0
1K	normal	147,254
	sdt	30,363
	性能向上率 (%)	79.4
10K	normal	1,486,127
	sdt	313,824
	性能向上率 (%)	78.9

ら属性値を読み出すときの 1 データ当たりの平均読み出しサイクル数はそれぞれ 4.6，2.9，2.8 サイクルであった（タプル数 100，1K，10K の順）．

今回の評価では属性 four の値（0，1，2，3 の値）は 4 つであるためキャッシュの 1 ブロックに収まり，結果リレーションの走査によるメモリアクセスはほとんど発生せず，SDT 転送中断によるストールサイクル数の影響を受けなかった．追加評価としては属性値の持つ値の範囲が大きい属性 unique1 および onePercent などで実行した場合との比較が考えられる．

6.4 SDT 方式を適用する集約質問

本節で評価する質問処理は SDT 方式を適用する集約質問を対象とし，直接リレーションに格納されている属性値に対して SDT 方式を適用した場合の実行サイクル数を従来方式と比較する．評価する集約質問を以下に示す．

```
SELECT four, COUNT(*) FROM R
GROUP BY four
```

この集約質問はリレーションから GROUP BY 句で指定した属性 four の値を読み出し，その値を各グループに分けて COUNT 関数によってその出現回数をカウントする．結果リレーションはタプルの属性読み出し毎に走査され，属性値の各値とその出現回数を格納する．

次に集約質問における従来型 MC の質問処理コードを以下に示す．

従来型 MC の質問処理コード

```
j = 0;
```

```
/* relationRの繰り返し処理 */
for( i_R = 0; i_R < MAXTUPLES_R; i_R++ ){
  tmp = tplarryR[i_R].four;                /* relationRの属性読み出し */
  /* 結果リレーションの走査 */
  for( i = 0; i < j; i++ ){
    /* 既出の属性値をカウント */
    if( tmp == rslt_val[i] ){
      rslt_cnt[i]++;
      break;
    }
    /* 結果リレーションの作成 */
    else if( rslt_val[i] == NULL ){
      rslt_val[j++] = tmp;
      rslt_cnt[j]++;
      break;
    }
  }
}
```

この質問処理コードはリレーション R を MAXTUPLES 回繰り返して属性値を読み出し，結果リレーションを走査して既に格納されている属性値ならば出現回数（rslt_cnt）をインクリメントする．属性値が未出ならば，その値（rslt_val）を結果リレーションに格納し，出現回数を 1 にセットする．読み出した属性は一時領域（tmp）に格納し結果リレーション走査時の不要なメモリアクセスを減らす．

SDT 方式の質問処理コード

選択・DISTNCT 質問と同様に前処理として MC 内のレジスタにデータ転送数と間隔値を書き込み，アドレスの上位 2 ビットを “01” とした仮想アドレス sdt_addr でリレーション R の属性値を読み出すことで SDT 転送を実現する．これによりリレーション R の属性 four の値が FIFO バッファに格納される．従来の処理コードの “relationR の属性読み出し” 部分を

```
int *sdt_addr;        /* SDT方式で参照する属性のアドレス */
sdt_addr = (unsigned int)(&tplarryR[i_R].four) | 0x40000000;
tmp = *sdt_addr;
```

に変更する．ほかの部分は従来の処理コードと同様である．

実行サイクル数

表 6.4: 集約質問の実行サイクル数（SDT）

タプル数	アクセス方式	総サイクル数
100	normal	16,529
	sdt	5,071
	性能向上率 (%)	69.3
1K	normal	154,236
	sdt	37,465
	性能向上率 (%)	75.7
10K	normal	1,531,484
	sdt	359,301
	性能向上率 (%)	76.5

表 6.4 に従来型 MC 方式と SDT 方式でリレーションのタプル数を変更して実行したときの総サイクル数を示す．SDT 方式では従来型 MC 方式と比較して実行サイクル数の 69.3〜76.5%を削減することができた．

従来方式では実行サイクル数の 72〜78%がリレーションの属性読み出しにおけるキャッシュミスに伴うメモリアクセスに費やされていた．これは選択・DISTNCT 質問と同様に 1 タプルのデータサイズがキャッシュブロックより大きいためである．

SDT 方式では DISTNCT 質問と同様にプロセッサの命令実行に並行した属性読み出しを行い，FIFO バッファに格納して実際のメモリアクセス時間を隠している．データキャッシュミスによるメモリアクセスで SDT 転送中断のストールサイクル数を考慮した FIFO バッファ内の 1 データ当たりの平均読み出しサイクル数はそれぞれ 4.6，2.9，2.8 サイクルであった（タプル数 100，1K，10K の順）．

今回の評価では DISTNCT 質問と同様の状況が発生する．属性 four の各値とその出現回数を格納する結果リレーション（rslt_val および rslt_cnt）はそれぞれキャッシュの 1 ブロックに収まるため結果リレーションの走査によるメモリアクセスがほとんど発生せず，SDT 転送中断のストールサイクルの影響を受けなかった．追加評価で属性値の持つ値の範囲が大きい属性で実行した場合の比較が考えられる．

6.5 SDT 方式を適用する結合質問

本節で評価する質問処理は SDT 方式を適用する結合質問を対象とする．ここでは二つのリレーションによる条件探索を行う．評価する結合質問を以下に示す．

```
SELECT * FROM R, S
WHERE R.unique1 = S.unique1
```

それぞれの探索条件の属性 unique1 はユニークであるため，リレーション R に対して選択されるリレーション S のタプルは一つ以下になる．結果としてサイズの小さいリレーションのタプル数分の結果リレーションが作成される．結果リレーションにはリレーション R および S のタプルの先頭アドレスが格納される．

この結合質問における従来型 MC の質問処理コードを示す．

従来型 MC の質問処理コード

```
/* relationR の繰り返し処理 */
for( i_R = 0; i_R < MAXTUPLES_R; i_R++ ){
  tmp = relationR[i_R].unique1;              /* relationR の属性読み出し */
  /* relationS の繰り返し処理 */
  for( i_S = 0; i_S < MAXTUPLES_S; i_S++ ){
    /* タプルの条件探索 */
    if( tmp == relationS[i_S].unique1 ){    /* relationS の属性読み出し */
      /* 結果リレーションの作成 */
      rslt_ptr[j].R = &relationR[i_R];
      rslt_ptr[j].S = &relationS[i_S];
      j++;
    }
  }
}
```

結合質問処理は 2 つのリレーション（relationR，S）を同時に参照するため二重ループ構造になり，外側ループで読み出された属性に対して内側ループで条件探索する．その探索回数は両リレーションのタプル数を掛け合わせたものになる．内側ループで検索されるリレーションは外側ループのリレーションより参照回数が多くキャッシュの効果を得やすいため，タプル数の少ないリレーションを適用する．読み出した属性を一時領域（tmp）に格納することで不要なメモリアクセスを減らしている．

SDT 方式の質問処理コード

これまでと同様に前処理で転送数と間隔値を MC に格納し，属性読み出し部分で SDT を適用するため仮想アドレスの上位 2 ビットを “01” としたアドレス sdt_addr を指定する．ここでは外側に適用した場合，従来の処理コードの “relationR の属性読み出し” 部分を

```
int *sdt_addr;        /* SDT 方式で参照する属性のアドレス */
sdt_addr = (unsigned int)(&tplarryR[i_R].unique1) | 0x40000000;
tmp = *sdt_addr;
```

表 6.5: 結合質問の実行サイクル数（SDT） -1-

タプル数	アクセス方式	リードミスのサイクル数	総サイクル数	性能向上率(%)
R=1K S=100	normal	160,080(18.3)	872,451	-
	sdt(S に適用)	120,000(8.0)	1,495,395	-71.4
	sdt(R に適用)	3,854,520(83.7)	4,602,621	-427.6

表 6.6: 結合質問の実行サイクル数（SDT） -2-

タプル数	アクセス方式	リードミスのサイクル数	総サイクル数	性能向上率(%)
R=1K S=1K	normal	120,120,000(94.3)	127,425,320	-
	sdt(S に適用)	120,000(1.0)	12,116,695	90.5
	sdt(R に適用)	120,000,000(94.2)	127,336,630	0.1

に変更する．残りの部分は従来方式のコードと同様である．SDT 方式は一つの属性に対して適用可能であるため，内側または外側のどちらかに適用して評価する．

実行サイクル数

表 6.5 にリレーション R と S のタプル数が 1K，100 の場合，**表 6.6** にそれぞれ 1K の場合で従来型 MC 方式と SDT 方式で結合質問を実行したときの総サイクル数である．表中のリードミスのサイクル数はタプルの属性読み出しで発生したデータキャッシュアクセスのミスであり，括弧内は総サイクル数に対する比率を示す．また性能向上率は従来型 MC 方式を基準とする．

表 6.5 に関して従来方式ではキャッシュの効果を得るためタプル数の少ないリレーション S を内側ループに適用した．SDT 方式では同様にリレーション S を内側ループとして，リレーション R，S のどちらか一方に対して SDT 転送を適用した．

従来方式において外側での属性読み出しは全てミスとなるが，内側の読み出しではそのほとんどがキャッシュに収まっており，キャッシュヒット率は 99.7%であった．これにより再び内側ループに入った際に探索するリレーション S の属性値がキャッシュに残っているためリードミスのサイクル数の割合は総サイクル数の 18.3%に抑えられている．

一方，SDT 方式を内側ループであるリレーション S に適用した場合，リードミスのサイクル数は外側での属性読み出しのみに起因しており，その割合は 8.0%になっている．しかし 1K 回の SDT 起動による初回データの待ち時間および，参照回数の多い内側ループでのキャッシュミスによる SDT 転送の中断から再開までのストールサイクルによって性能が低下している．

SDT 方式を外側ループのリレーション R に適用した場合，リードミスのサイクル数は内側での属性読み出しのみに起因している．従来方式ではキャッシュの両ウェイを使用したため高いキャッシュヒット率で内側ループの属性値が得られた．しかし SDT 転送はどちらかのウェイを FIFO バッファとして利用するため通常使えるキャッシュは一つのウェイに限られる[2]．そのため通常使えるキャッシュにリレーション S 全体を格納できないため内側ループの属性読み出しでキャッシュミスが多発し，キャッシュヒット率は 67.9%であった．よって従来方式より性能が低下したと考えられる．

表 6.6 はリレーションのタプル数が同じで，従来方式および SDT 方式においてリレーション R を外側ループ，リレーション S を内側ループに適用した結果である．SDT 方式ではどちらかのリレーションに対して SDT 転送を適用する．

従来方式において外側および内側の属性読み出しは全てミスになるため，リードミスのサイクル数の比率は総サイクル数のほとんどを占めている．これは本来ならばキャッシュを有効利用できる内側ループのリレーションサイズがキャッシュサイズよりも大きく，再度内側ループの属性値を参照する場合に探索する最初の属性値がキャッシュから追い出されているためである．

SDT 方式を内側ループのリレーション S に適用した場合，リードミスのサイクル数において外側の属性読み出しは全てミスになるが，内側の属性読み出しでは FIFO バッファによって見かけ上のメモリアクセス時間を削減している．FIFO バッファから属性値を読み出すときの 1 データ当たりの平均読み出しサイクル数は 4.9 サイクルであった．これにより総サイクル数の 90.5%を削減する結果が得られた．

SDT 方式を外側ループのリレーション R に適用した場合，内側でのデータ読み出しは全てミスになり，リードミスによるサイクル数の比率は全実行サイクル数のほとんどを占めている．これは従来方式と同様にリレーション S がキャッシュサイズよりも大きいことに起因する．

以上の結果から，SDT 方式を適用した結合質問において内側ループのリレーションサイズがキャッシュ容量に収まる場合ではキャッシュの効果が得られるため SDT 転送は性能を低下させる．しかしキャッシュ容量より大きなリレーションを探索する場合，再度内側ループに入ったときの最初の属性値はキャッシュから追い出されているためキャッシュが有効利用されていない．このような状況では SDT 転送による性能向上が得られた．

6.6 TPDT 方式を適用する選択質問

本節で評価する質問処理はポインタを介した二段階アクセスが必要な文字列型データに対して条件探索する選択質問を対象とする．SLECT 文の WHERE 句で指定する文字列の種類または長さなどを変更して従来型 MC 方式と TPDT 方式による実行サイクル数の比較を行う．評価する選択質問を以下に挙げる．

[2]本来はどちらかのウェイの最上位ブロックから FIFO バッファとして使用し，残りのブロックは通常のキャッシュして使用できるが今回の評価ではウェイ全てを FIFO バッファと仮定した場合である．

```
SELECT * FROM R WHERE string1 = ‘xxxxx’
```

この質問処理はリレーション R から属性 string1 が任意の文字列 ‘xxxxx’ と比較し一致したタプルが選択され，結果リレーションには一致したタプルの先頭アドレスが格納される．属性 string1 への参照で TPDT 方式を適用する．

選択質問における従来型 MC の質問処理コードを以下に示す．

従来型 MC の質問処理コード

```
j = 0;
/* relationR の繰り返し処理 */
for( i_R = 0; i_R < MAXTUPLES; i_R++ ){
  srch =“ xxxxx ”;
  tmp = relationR[i_R].string1;              /* relationR の属性読み出し */
  /* 読み出した属性の実体比較 */
  for( ; *tmp ==  *srch ; tmp++, srch++ ){
    if( *srch == ’\0’ ){
      rslt_ptr[j++] = &relationR[i_R];
      break;
    }
  }
}
```

この質問処理コードは MAXTUPLES 回繰り返すループ内で探索する文字列へのポインタを srch に格納し，リレーション R から属性 string1 の値（文字列型の実体へのポインタ）を読み出す．それらのポインタを辿って実体同士を NULL 文字（\0）まで比較し，一致する場合に結果リレーションにタプルの先頭アドレスを格納する．

TPDT 方式の質問処理コード

```
char *tpdt_addr;             /* TPDT 方式で参照する属性のアドレス */

j = 0;
/* relationR の繰り返し処理 */
for( i_R = 0; i_R < MAXTUPLES; i_R++ ){
  srch =“ xxxxx ”;
  tpdt_addr = (unsigned int)(&relationR[i].string1) | 0x80000000;
  /* 読み出した属性の実体比較 */
  for( ; *tpdt_addr == *srch; srch++ ){
    if( *srch == ’\0’ ){
```

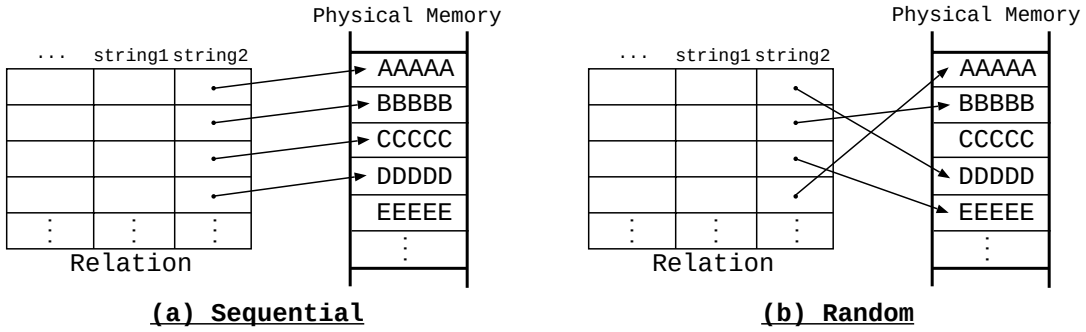

図 6.2: ポインタの格納順

```
      rslt_ptr[j++] = &relationR[i];
      break;
    }
  }
}
```

このコードは MAXTUPLES 回繰り返すループ内で探索する文字列へのポインタを srch に格納し，TPDT 転送を属性 string1 の実体データへの参照に適用するため仮想アドレスの上位 2 ビットを “10” としたアドレス tpdt_addr を指定する．これにより属性 string1 の文字列データがメモリから連続して転送され FIFO バッファに格納される．

プロセッサが転送中の文字列データに対するロード命令を発行した場合，MC ではそのアドレスに対するメモリアクセスを無効化する．

また文字列の比較途中で条件に一致しない場合，次タプルの文字列データが転送されてくるまでに FIFO 内の前タプルのデータをクリアする．

実行サイクル数

TPDT 方式を適用する選択質問では実体データが直接リレーションに格納されていないため，文字列の選び方やポインタの格納順で実行サイクル数が変わる．ここでは以下の項目について実行サイクル数を求め評価を行う．

- 文字列の種類
- 文字列の長さ
- ポインタの格納順

ポインタの格納順とは，各タプルの属性 string および string2 に文字列へのポインタを格納する順番のことであり，その順番がメモリ上に配置された順番（**図 6.2** の（a））, あるいはランダム（同図の（b））であるかを示す．

6.6.1 文字列の種類によるサイクル数の比較

表 6.7 に文字列の種類を変更して従来型 MC 方式と TPDT で選択質問を実行したときの総サイクル数である．表中の抽出タプルは，条件探索結果で得られたタプル数である[3]．従来方式における ld ミス回数は，二段階目のアクセスすなわち属性 string1 に格納されたポインタで実体データをアクセスする際に発生するロードミス回数である．また今回の評価は文字列の長さを 30 文字とし，最大で 500 種類の文字列が必要であり，以下の文字列から生成する．

```
AAAAA-AAAAA-BBBBB-CCCCC-DDDDD-EEEEE
AAAAA-BBBBB-CCCCC-DDDDD-EEEEE-FFFFF
...
AAAAA-SSSSS-TTTTT-UUUUU-VVVVV-WWWWW
AAAAA-TTTTT-UUUUU-VVVVV-WWWWW-XXXXX  （ハイフンは含まない）
```

これらは 20 種類存在し，先頭の 5 文字を B〜Y，a〜y に変更することで合計 1000 種類の文字列が生成される．これらの中から順番に 50，100，200，500 種類を選択し，実体データとして使用する．

表 6.7 において従来方式ではタプル数に対する文字列の種類が少ない場合，ポインタと実体の両方がキャッシュで保持されるため，再度同じ文字列を探索する場合にキャッシュミスによるメモリアクセスは発生しにくくなっている．一方，タプル数に対する文字列の種類が多い場合，ld ミス回数が多く，実行サイクル数が増加している．これはタプルの検索で文字列の種類が周回したときに，最初に検索した実体データが新しい実体データによって追い出されるためである．タプル数を増やした場合，このほかにポインタ読み出しによる実体データの追い出しがさらに発生するため ld ミス回数が多くなる．

一方，TPDT 方式ではタプルに対する文字列の種類が少ない場合，実体データの読み出し毎に 200 サイクル必要で，読み出された実体データはキャッシュに保存しないため，キャッシュが有効利用されている従来方式より性能向上は見られない．しかしタプルに対する文字列の種類が多い場合，タプル条件探索毎にポインタ読み出しと実体読み出しで 2 回のメモリアクセス（240 サイクル）が必要であるのに対し，TPDT 方式では 200 サイクルで実体データを読み出す．これによって性能向上が得られる．

以上のことから，タプルに対数文字列の種類が少ない場合，すなわち一つの文字列を複数のタプルで共有する場合にキャッシュを有効利用するため TPDT による効果は得られないことがわかる．

[3]例えばタプル数が 1K で文字列の種類が 50 の場合，リレーション内で 50 種類のポインタが周期的に繰り返されるため 50 種類中に 1 タプルが条件に一致すると合計 20 タプルが抽出される．

表 6.7: 選択質問の実行サイクル数（TPDT） -1-

タプル数	文字列の種類	抽出タプル数	norml		tpdt サイクル数	性能向上率 (%)
			ld ミス回数	サイクル数		
100	10	10	10	22,208	28,253	-27.2
	50	2	50	22,326	24,085	-7.9
	100	1	100	27,182	23,079	15.1
1K	50	20	59	164,455	232,453	-41.4
	100	10	217	170,323	222,153	-30.4
	200	5	895	245,366	217,123	11.5
	500	2	1,000	253,622	213,985	15.6
10K	50	200	127	1,582,350	2,315,173	-46.3
	100	100	1,389	1,612,052	2,213,133	-37.3
	200	50	8,755	2,433,623	2,161,993	11.2
	500	20	1,000	2,539,638	2,131,453	16.2

6.6.2 文字列の長さによるサイクル数の比較

表 6.8 では**表 6.7** で性能向上が得られたタプル数–文字列の種類の組を選び，探索する文字列の長さを変更して得られた結果である．文字列の長さは 15，30，60 としてそれぞれ最低 10，20，40 文字を比較するように設定した．

表 6.8 において従来方式ではこれまでのポインタ読み出しによるメモリアクセスのほかにプロセッサの文字列比較途中にも発生する．これはキャッシュの 1 ブロックに収まらない実体データはその比較途中で再度メモリアクセスし，文字列の続きを比較する必要があるためである．例えば，文字列の長さが 60 文字では最低 40 文字の比較が必要になるため，比較途中で最低 1 回のメモリアクセスが発生する．さらに読み出した実体データによってポインタが追い出されるという状況も発生する．一方，TPDT 方式では FIFO を使うことでこれらの問題を回避し，文字列データを連続して転送する．

表 6.7 および**表 6.8** は**図 6.2** の（a）のように属性 string1 のポインタはメモリ上に配置された順番で格納されている．**表 6.8** の文字列の長さが 15 のときキャッシュの 1 ブロックには 2 つの実体データが入り，それらはタプルの実体比較で順番に参照されるため，ld ミス回数はカウントされない．そこで**図 6.2** の（b）のようにランダムでポインタを格納する方針[4]を採用し，その実行結果が**表 6.9** である．

[4] このようにランダムでポインタが格納されるのは MMDB において自然であると考えたため．

表 6.8: 選択質問の実行サイクル数（TPDT） -2-

タプル数	文字列の種類	抽出タプル数	文字列の長さ	norml		tpdt	性能向上率 (%)
				ld ミス回数	サイクル数	サイクル数	
100	100	1	15	51	29,231	30,119	-3.0
			30	101	44,294	38,175	13.8
			60	201	74,398	54,375	26.9
1K	200	5	15	330	263,015	292,443	-11.2
			30	896	420,940	372,723	11.5
			60	2,001	734,234	533,163	27.4
	500	2	15	499	282,314	292,113	-3.5
			30	999	431,990	371,985	13.9
			60	1,999	732,114	531,913	27.3
10K	200	50	15	2,891	2,581,975	2,915,193	-12.9
			30	8,745	4,187,301	3,717,993	11.2
			60	20,001	7,342,093	5,321,313	27.5
	500	20	15	4,981	2,824,205	2,912,733	-3.1
			30	9,982	4,322,961	3,711,453	14.1
			60	19,963	7,319,249	5,309,653	27.5

6.6.3 ポインタの格納順によるサイクル数の比較

表 6.9 はポインタの格納順を**図 6.2** の（b）のようにランダムで格納した場合の総実行サイクル数である．タプル数，文字列の種類および文字列の長さはそのほかは**表 6.7** と同じ条件とする．

表 6.9 と**表 6.9** の比較すると，ポインタの格納順がメモリ上に配置された順番では，キャッシュの 1 ブロック内に格納された最初の実体データは ld ミスとしてカウントされるが，次の実体データはカウントされない．ポインタの格納順をランダムにするとブロック内に格納された複数の実体データが連続して参照されることが少なくなる．さらにブロック内の連続した実体データの読み出しまでにそのブロックに対するポインタあるいは別の実体データの読み出しでそのデータがキャッシュから追い出される状況がある．これらによって従来方式では ld ミス回数分のメモリアクセスによる実行サイクル数が増加するのに対し，TPDT 方式ではサイクル数の変化は見られない．TPDT 方式を適用することで実行サイクル数の 36.6%を削減する結果が得られた．

表 6.9: 選択質問の実行サイクル数（TPDT） -3-

タプル数	文字列の種類	抽出タプル数	文字列の長さ	norml		tpdt	性能向上率 (%)
				ld ミス回数	サイクル数	サイクル数	
100	100	1	15	136	39,520	30,135	23.7
			30	190	54,850	38,159	30.4
			60	292	85,300	54,359	36.3
1K	200	5	15	1,099	355,252	292,443	17.7
			30	1,731	520,963	372,523	28.5
			60	2,834	834,678	533,243	36.1
	500	2	15	1,206	367,420	292,305	20.4
			30	1,830	532,168	372,289	30.0
			60	2,891	840,004	532,553	36.6
10K	200	50	15	10,713	3,523,114	2,915,193	17.3
			30	17,197	5,200,541	3,715,993	28.5
			60	28,827	8,390,774	5,316,053	36.6
	500	20	15	11,809	3,648,046	2,914,653	20.1
			30	18,241	5,318,416	3,714,493	30.2
			60	28,827	8,390,774	5,316,053	36.6

6.7 TPDT 方式を適用する結合質問

本節で評価する質問処理はポインタテーブルを用いた結合質問を対象とし，結合属性への参照として TPDT 方式を適用する．ここでは二つのリレーションによる条件探索を行い，それぞれので作成されるポインタテーブルのサイズと結合属性を変更して実行する．評価する結合質問を以下に示す．

(a)
```
SELECT * FROM R, S
WHERE R.unique2 = S.unique2 AND R.two = 1 AND S.two = 1
```

(b)
```
SELECT * FROM R, S
WHERE R.unique2 = S.unique2 AND R.two = 1 AND S.four = 1
```

(c)
```
SELECT * FROM R, S
WHERE R.unique2 = S.unique2 AND R.two = 1 AND S.ten = 1
```

(d)
```
SELECT * FROM R, S
WHERE R.unique2 = S.unique2 AND R.two = 1 AND S.twenty = 1
```

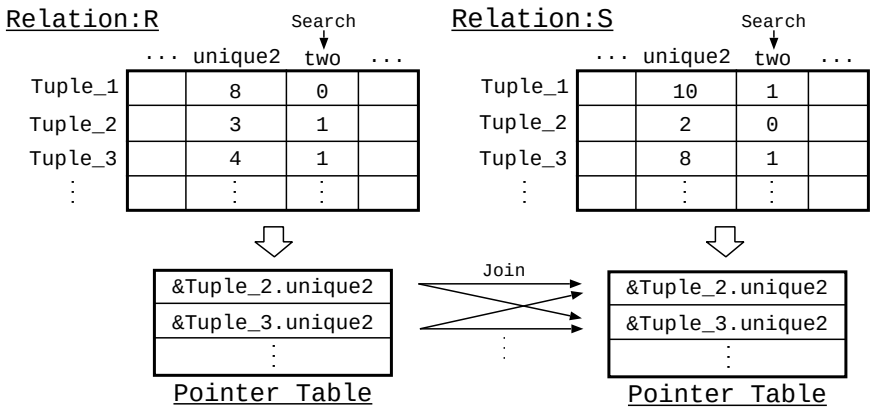

図 6.3: TPDT 方式のポインタテーブルを使った条件探索

（a）の結合質問において TPDT 方式によるポインタテーブルを使用した条件探索を**図 6.3** に示す．リレーション R および S からそれぞれ属性 two で探索条件に一致したポインタテーブルを作成する．ポインタテーブルには次の処理に必要な属性へのポインタ（つまり，属性 unique2 へのアドレス）を格納する．そしてポインタテーブル同士の比較で TPDT 方式を適用し，結合処理を行う．

一方，従来方式ではそれぞれ属性 two で一致したタプルの先頭アドレスをポインタテーブルに格納する．そしてポインタテーブルからそれぞれの先頭アドレスを辿って結合属性 unique2 の属性値を読み出し，比較する．

結合質問における従来型 MC の処理コードを以下に示す．

従来型 MC の質問処理コード

```
j = 0;
for( i_R = 0; i_R < MAXTUPLES_R; i_R++ ){
  if( relationR[i_R].two == 1 )
    ptr_tbl_R[j++] = &relationR[i_R];     /* ポインタテーブルの作成 (R)     */
}
stp_R = j;

j = 0;
for( i_S = 0; i_S < MAXTUPLES_S; i_S++ ){
  if( relationS[i_S].two == 1 )
    ptr_tbl_S[j++] = &relationS[i_S];     /* ポインタテーブルの作成 (S)     */
}
stp_S = j;
```

```
j = 0;
/* ポインタテーブルの繰り返し処理 */
for( i_R = 0; i_R < stp_R; i_R++ ){
  tmp_addr = ptr_tbl_R[i_R];            /* ポインタテーブルの読み出し（R） */
  tmp_val = tmp_addr->unique2;          /* 結合属性の読み出し（R）         */
  /* ポインタテーブルの繰り返し処理 */
  for( i_S = 0; i_S < stp_S; i_S++ ){
    /* タプルの条件探索 */
    if( tmp_val == ptr_tbl_S[i_S]->unique2 ){
      /* 結果リレーションの作成 */
      rslt_ptr[j].R = tmp_addr;
      rslt_ptr[j].S = ptr_tbl_S[i_S];
      j++;
    }
  }
}
```

この質問コードはそれぞれのリレーション（R，S）から属性 two で一致するタプルの先頭アドレスとポインタテーブル（ptr_tbl_R，ptr_tbl_S）に格納する．ポインタテーブル R で読み出したタプルの先頭アドレスおよび結合属性を一時領域（tmp_addr，tmp_val）に格納し，ポインタテーブル S を介した属性 unique2 で結合処理を行う．結果リレーションには条件に一致したそれぞれのタプルの先頭アドレスを格納する．

TPDT 方式の質問処理コード

まずリレーション R からポインタテーブルを作成するために属性 two で条件探索し，

```
ptr_tbl_R[j++] = &relationR[i_R].unique2;
```

で一致したタプルの属性 unique2 へのポインタを格納する．リレーション S についても同様である．次にこれらのポインタテーブルはメモリ内で常に最新の値で保持される必要があるため，キャッシュをライトバックしてポインタテーブルの値を更新する．この更新処理はキャッシュブロックを走査して書き込みビットがセットされたブロックをライトバックし，その必要がないブロックの場合は何も処理せず，パイプライン中にその命令が流れるだけである[5]．ポインタテーブル R の属性 unique2 への参照で TPDT を適用する場合，従来の処理コードの “ポインタテーブルの読み出し（R）” 部分と “結合属性の読み出し（R）” 部分をまとめて，

```
int *tpdt_addr;  /* TPDT 方式で参照する属性のアドレス */
tpdt_addr = (unsigned int)ptr_tbl_R[i_R] | 0x80000000;
tmp_val = *tpdt_addr;
```

[5]プロセッサの命令セットにこのような命令を用意する必要がある．

表 6.10: 結合質問の実行サイクル数（TPDT）

タプル数	R の探索属性	S の探索属性	normal	tpdt (R のみ)	性能向上率 (%)	tpdt (R と S)
R=10K S=1K	two	twenty	45.7	46.7	-2.3	544.2
		ten	135.1	132.8	1.7	1,061.7
		four	1261.7	1255.2	0.5	2,614.3
		two	3483.1	3479.6	0.1	5,201.9

（単位:10^5 サイクル）

のように記述し，仮想アドレスの上位 2 ビットを “10” とした tpdt_addr で TPDT 転送を実現する．ポインタテーブル S に適用する場合も同様である．最後に結合処理でタプルが一致した場合，

```
rslt_ptr[j].R = tpdt_addr - OFFSET_R;
rslt_ptr[j].S = ptr_tbl_S[i_S] - OFFSET_S;
```

のようにそれぞれの結合属性 unique2 から先頭のオフセット (OFFSET_R および OFFSET_S) 分を差し引いて先頭アドレスを求めた後で結果リレーション rslt_ptr に格納する．

実行サイクル数

表 6.10 は従来型 MC 方式と TPDT 方式で（a）〜（d）を実行したときの総サイクル数を示す．表中の R の探索条件および S の探索条件は，そリレーション R およびリレーション S からポインタテーブルを作成するときに指定する属性で，リレーション R では属性 two で固定しリレーション S ではその属性を変更する．タプル数をそれぞれ 10K，1K で固定しポインタテーブル R のタプル数は 5K，ポインタテーブル S はそれぞれ 50，100，250，500（属性 twnety，ten，four，two の順）になる．両方式においてポインタテーブルのサイズが小さいポインタテーブル S を内側ループとしてキャッシュを利用する．

S の探索条件を変更することによるサイクル数の増加は，ポインタテーブル S のサイズが大きくなるに従い，結合処理で内側ループでの参照回数が多くなるためである．

従来方式ではポインタテーブル S のサイズが小さい場合，外側ループでのポインタテーブルの読み出しは最初の要素（タプルの先頭アドレス）に関してキャッシュミスになるが同じキャッシュブロック内に格納される要素はヒットする．しかし，ポインタテーブル S のサイズが大きい場合，キャッシュがポインタテーブル S で埋められヒットしていた要素が追い出されやすくなる．

一方，TPDT 方式をポインタテーブル R に適用すると，ポインタテーブル S のサイズに関係なく結合属性 unique2 への参照毎に TPDT 転送によるメモリアクセス（200 サイクル）が必要になる．TPDT 方式の効果はキャッシュがポインタテーブル S で埋められ，ヒットしていた要素が追い出される状況になって現れる．

しかし実行結果の表からポインタテーブル S のサイズが大きくなるに従い（属性 twenty は除く），性能向上率が低くなっているのは，TPDT の効果分のサイクル数の比率が全実行サイクル数に比べて低くなっいるためである．

さらに TPDT をポインタテーブル R と S に適用した場合，それぞれのポインタテーブルでキャッシュヒットしていた要素が全て TPDT のメモリアクセスに置き換わったため実行サイクル数が増大した．

全実行サイクル数のほとんどは内側ループによるポインタテーブル S の参照時間であり，外側ループでの参照時間の比率は小さい．そのため今回の評価ではポインタテーブル R のサイズを大きくしてその比率を高め，TPDT の性能向上が得られた．しかし，ポインタテーブル S のサイズをさらに大きくした場合では性能が低下する．

6.8 考察

本節では SDT 方式および TPDT 方式を各質問処理に適用したときの評価結果から従来方式と比較し，考察する．

従来方式

通常リレーション内のタプルを条件探索すると，読み出すタプルはキャッシュのブロックサイズよりも大きいために空間的局所性が存在せず，タプルの探索毎にメモリアクセスが発生する．またそのようなタプルは比較のための一度しか使用されず，時間的局所性も存在しない．このような性質を持つデータアクセスでは通常のキャッシュは有効に利用されない．しかし，結合質問などでリレーションあるいはポインタテーブル同士を比較する場合では，二重ループ構造の内側で読み出すデータは再度利用される．このときリレーションがキャッシュ内に収まる場合ではキャッシュはより有効に機能する．

SDT 方式

SDT 方式では必要なデータを予めメモリから連続して読み出しており，FIFO バッファはプリフェッチ機構の役割を果たしている．メモリからの連続読み出しはプロセッサの命令実行のバックグラウンドで行われており，プロセッサはキャッシュミスで発生するメモリアクセスのペナルティをストールせずに済む．従来方式ではメモリからのデータ読み出しで 120 サイクルのストールが必要であるのに対し，SDT 方式では最大で 2.8 サイクルでデータを読み出していた．これはプロセッサとメモリ間の性能差を吸収していると言える．

TPDT 方式

TPDT 方式は主にタプルの条件探索で文字列型の実体データを比較する場合で適用する．文字列を検索する際に文字列の種類や長さによって実行サイクル数は変わってくる．例えばタプル数に対して文字列の種類が少ない場合，一つの文字列は複数のタプルで共

有しており，ポインタと実体データがキャッシュヒットする確率は高くなる．このような場合では TPDT 方式による性能向上は見込めない．しかし，従来方式では文字列が 1 ブロック内に収まらない場合には文字その比較途中でメモリアクセスが発生するが，TPDT 方式では TPDT 中断が割り込まない限り，プロセッサは FIFO バッファによって文字列を連続して読み出し可能になる．

第7章　関連研究

本章ではプロセッサとメモリ間の性能格差問題に取り組んだメモリアーキテクチャに関する研究を二つ報告する．

従来の MC に新たな機能を付け加えることで高速データ転送を実現する方式として Impulse[4] と SMC[5] がある．

- **Impulse**
  Impulse は空間的局所性のないデータをプログラム中に用意した配列の変数別名（エイリアス）を用いて必要なデータのみをその配列に格納し，配列ごとプロセッサに転送する．この方式によってメモリバスの効率化を図ると共にキャッシュ内に無駄なデータが格納されることを防いでいる．しかし，これはプログラマへの負担が大きく，MC 内でデータアクセス毎の多段なアドレス変換をするためオーバヘッドが大きい．

- **SMC（Stream Memory Controller）**
  SMC は DRAM の同一 Row 内のデータを Col アドレスの連続指定して MC 内のバッファに格納する．そのバッファは複数存在し，メモリアクセス要求で読み出されたストリーミングデータは MC 内のメモリアクセス管理機構で各バッファに振り分けられる．しかし，MC–DRAM 間のアクセスは高速化されるが，データアクセス毎に発生するプロセッサ–MC 間の転送がボトルネックになっている．

これらの研究では空間的局所性のないデータに対して有効であり，時間的局所性の保証されないデータがキャッシュに格納されている．本研究では FIFO バッファを使うことでその問題を解消し，さらに必要なデータのみをプリフェッチしてメモリアクセスのオーバヘッドを削減している．

第8章　おわりに

8.1　まとめ

本論では主記憶データベースにおけるデータ参照時間削減手法として，メモリ空間に一定間隔で存在するデータをメモリからプロセッサに連続して転送する SDT 方式と，ポインタを介した二段階のアクセスに対してプロセッサが見かけ上一回のメモリアクセスでデータを取得する TPDT 方式を提案した．これらの方式と従来方式に関してシミュレーションにより質問処理時間を評価した．評価結果から単純な質問処理と結合質問処理では，これらの転送方式を適用することでデータ参照時間を削減し，質問処理時間を短縮したと言える．

8.2　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

（2）複雑な質問処理文の評価

（3）データベース演算機構の追加

(1）では今回作成シミュレータは二次キャッシュを考慮していなかったため，ミスペナルティが全てメモリアクセスのペナルティになっている．また，TPDT 方式で MMU の仮想–物理アドレス変換機構を MC 内で実装していないため，所要サイクル数を考慮していなかった．今後は MC 内に MMU を実装する．

（2）では，今回実行した質問文は一つあるいは二つのリレーションに対する単純質問および結合質問であるが，今後はさらに複数のリレーションに対する実行や入れ子型質問などを評価する．

（3）では，MC にデータベース演算機構を追加することでプロセッサの演算負荷を分散し，スループットの向上を図る．

謝辞

本研究を遂行するにあたり，終始熱心にかつ懇切丁寧にご指導を賜りました，田中 清史助教授に心から深く感謝するとともに，ここに御礼申し上げます.

貴重な御助言をしていただきました日比野 靖 教授，堀口 進教授，宮崎 純　助手に深く感謝致します.

その他，貴重な御意見，御討論をいただきました田中・日比野研究室の皆様をはじめとする多くの方々の御助言に対して厚く御礼申し上げます.

本研究で使用した Synopsys 社と Model Technology 社の UniversityProgram．に深く感謝します.

最後に，日頃から暖かく支援して下さいました両親に感謝致します.

参考文献

[1] H. Garcia-Molina, K. Salem, “Main Memory Database Systems: An Overview.” IEEE Trans. on Knowledge and Data Engineering, Vol.4, No.6, pp.509–516, 1992.

[2] DeWitt. D. J, “The Wisconsin Benchmark:Past, Present, and Future.” The Benchmark Handbook, pp.269–316, J.Gray ed., Morgan Kaufmann, 1993.

[3] 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.

[4] J. Carter, W. Hsieh et al. “Impulse: Building a smarter memory controller.” In Proc. of the 5th HPCA, pp. 70–79, 1999.

[5] S. McKee et al. “Design and evaluation of dynamic access ordering hardware.” In Proc. of the 10th ICS, pp.125–132, 1996.

[6] Vinodh Cuppu, Bruce Jacob. “A Performance Comparison of Contemporary DRAM Architecture.” In Proc. of the 26th ISCA, pp.222–233, 1999.

[7] Altera, Inc “SDR SDRAM Controller White Pager”, 1999.

[8] HITACHI, Inc “HM5225165B/HM5225805B/HM5225405B-75/A6/B6”, 1999.

[9] Elpida Memory, Inc “ユーザーズマニュアル SDRAM の使い方”, 1999.

[10] 増永 良文, “リレーショナルデータベース入門”, サイエンス社 1991.