情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 4　1–14 (Dec. 2009)

バイナリ変換によるデータプリフェッチのためのハードウェア削減手法

請　園　智　玲[†1]　　田　中　清　史[†1]

プロセッサの動作速度とメモリアクセススピードの間には大きな速度差があり，その速度差は現在も増大を続け，プロセッサの計算速度向上の妨げとなっている．この問題はメモリウォール問題として広く知られている．これまで，メモリウォール問題を改善するための手法の 1 つとしてデータプリフェッチが研究されてきた．データプリフェッチはコンパイラによるソフトウェアプリフェッチと専用ハードウェアを用いたハードウェアプリフェッチに大きく分類することができる．一般的に，ソフトウェアプリフェッチは予測精度が悪く，ハードウェアプリフェッチは予測精度は高いがハードウェアコストが高い．本稿は，ネイティブバイナリに対する動的最適化手法を用いることで，予測精度が高くハードウェアコストの低いデータプリフェッチを実現する手法を提案する．シミュレーションにより提案手法が他のプリフェッチ手法に比べて，同等もしくはそれ以上の性能持つことを確認した．また，それぞれの必要メモリ量の見積りを行い，提案手法ハードウェア規模がきわめて小さいことを示した．

A Technique for Reducing Data Prefetch Hardware Using Binary Translation

Tomoaki Ukezono[†1] and Kiyofumi Tanaka[†1]

There is a significant gap between processor's operational speed and memory access speed, and currently the gap is increasing. This problem is widely known as memory wall problem. Data prefetch techniques have been studied as a way to alleviate the memory wall problem. Data prefetch techniques can be devided roughly into software prefetch by using a compiler and hardware prefetch by using dedicated hardware. In general, prefetch accuracy of software prefetch techniques is lower than that of hardware prefetch and hardware prefetch while hardware prefetch requires higher cost of dedicated hardware. In this paper, we propose a prefetch technique that can achive both low hardware cost and accurate prefetching by using a dynamic optimization technique for native binary. We confirmed that the proposed techniques can had similar or better performance than conventional prefetch techniques by simulation, and we showed that hardware scale of the proposed techniques was much small by estimating required memory for each technique.

1. はじめに

データプリフェッチは主記憶にあるメモリブロックが実際に使われる前に，主記憶に対して投機的にノンブロッキングリード要求を発行する手法である．プリフェッチにより主記憶から読み出されたメモリブロックは，プログラム実行のバックグラウンドでキャッシュメモリなどに格納される．これにより，計算に必要なメモリブロックが実際に使われる前にキャッシュメモリに到着した場合，メモリアクセス遅延を完全に隠蔽でき，プロセッサとメモリアクセス速度の差を吸収することができる．また，対象ブロックの到着が間に合わなかった場合でも，部分的にメモリアクセス遅延を隠蔽することができる．このことから，データプリフェッチの適用はキャッシュメモリの非効率利用や初期参照ミスの問題を解決する有効な手段となる．

代表的なプリフェッチ実現のアプローチは 2 つある．ソフトウェアプリフェッチとハードウェアプリフェッチである．

ソフトウェアによるアプローチはコンパイラによる最適化手法の 1 つとして実現される[2)–5)]．この手法を用いる際に，プログラム中のある段階で，将来必要になるであろうメモリブロックのアドレス（プリフェッチアドレス）の予測は，プログラムコードの構造やデータ構造などから解析される．このため，ソフトウェアプリフェッチは特別なハードウェアによるサポートを必要としない．ソフトウェアプリフェッチは単純な制御構造やデータ構造を持つプログラムコードに対し，メモリアクセスレイテンシを隠蔽するうえで大きな貢献がある一方で，解析の困難さに起因する複雑なプログラムコードに対する低いプリフェッチ精度や，プリフェッチ命令自身とプリフェッチアドレス計算のための命令数増加による命令フェッチオーバヘッドの問題がある．

ハードウェアによるプリフェッチ手法は，CPU やメモリコントローラなどに追加されたハードウェアがプログラム実行のバックグラウンドでプリフェッチリードを発行する方法である．この手法の特徴は，実際に CPU から発行されるアドレスを用いてプリフェッチアド

†1 北陸先端科学技術大学院大学情報科学研究科
JAPAN Advanced Institute of Science and Technology

© 2009 Information Processing Society of Japan

レスを予測するところにある[6)–8)]．これらの提案は，参照を解析するために特別なハードウェアを用いることで，高精度でプリフェッチ発行を行うことを可能にする．しかしながら，実行中に発行済みアドレスを記憶するための大規模なハードウェアが要求されるため，ハードウェア量の増大や遅延設計上，回路の動作周波数を低下させる要因となる可能性がある．

本稿では，それぞれのプリフェッチ実現手法の特徴を活かすために，新しいアプローチのデータプリフェッチ手法を提案する．本手法はソフトウェアプリフェッチと同様に，ソフトウェア側からプリフェッチ命令を実行する方式を採用するが，コンパイラによるプログラムコード解析ではなく，動的最適化システムによりプログラム実行環境で実際に発行されたメモリアクセスを解析し，プリフェッチのためのコード最適化を行う．本提案は手段としてソフトウェアの修正を用いるが，本質的には CPU にハードウェアを追加して発行済みアドレスの解析によるプリフェッチ発行を目的とすることから，ハードウェアプリフェッチにより近い．これまでの研究で，我々はネイティブバイナリに対する動的最適化を行うシステムを提案した．このシステムは Hybrid and Dynamic Optimization System （HDOS）と呼ばれる[1)]．本提案は HDOS を用いてデータプリフェッチを実現する．HDOS は実行中にコードの最適化を行うので，ユーザに対して透過的であるとともに，生成されたバイナリは，プログラム実行終了時に実行ファイルとして保存することが可能で，次回以降に同一の実行が行われる場合，最適化処理なしにプリフェッチの効果のあるバイナリを実行できる利点がある．また，HDOS は軽いハードウェアで実装できることから，少ないハードウェア量でハードウェアプリフェッチと同じ性能が実現できれば，CPU 設計においてハードウェア量の節約になる．加えて，本提案は従来のソフトウェアプリフェッチの欠点であったプリフェッチアドレス計算のための命令数増加を，通常のロード/ストア命令にプリフェッチ機能を付加した命令を用いることで削除している．

2. HDOS

HDOS は CPU 内部の補助的なトラップハードウェアとオペレーティングシステムに組み込まれたトラップハンドリングソフトウェアで構成される．最適化ルーチンはトラップハンドラとして実装される．本章では補助ハードウェアの詳細とハードウェア—ハンドラ間のインタフェースの詳細を説明する．

2.1 トラップハードウェア

以前の研究において，動的最適化システムのためにトラップハードウェアを提案した[1)]．提案ハードウェアはソフトウェアによりコントロールされ，必要に応じトラップイベントを

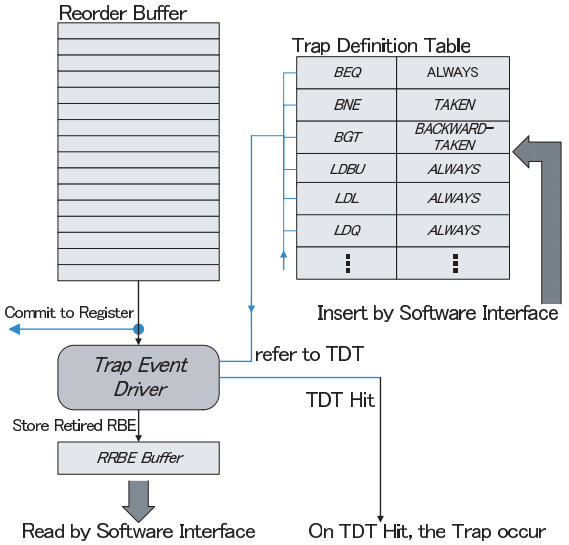


図 1　User Definable Trap（UDT）ハードウェア
Fig. 1　User Definable Trap (UDT) hardware.

発生させる．提案トラップハードウェアは本稿において User Definable Trap（UDT）と呼ばれる．UDT のハードウェアブロックを**図 1** に示す．図中では，たとえば分岐方向が taken のとき，BNE の実行後トラップを発生させる．従来の CPU デザインでは，CPU が提供するトラップの機能は例外や外部割込み，システムコールなど，CPU の設計者によって定められた原因の場合のみ，トラップハンドラへ制御の遷移を許す．これに対し，UDT はユーザによって指定されたトラップ条件が満たされた場合に，ユーザによって指定されたトラップハンドラを起動する．

UDT はリオーダバッファがリタイアエントリをレジスタファイルにコミットした後にトラップを発生させる機能を持つ．図 1 の Trap Event Driver（TED）が UDT の処理の中心となる回路である．TED は 2 つの処理を同時に行う．1 つ目の処理はリオーダバッファのリタイアエントリを Retired Reorder Buffer Entry（RRBE）バッファ内に格納することである．2 つ目の処理は Trap Definition Table（TDT）の参照である．TDT エントリは命令を識別するオペコードとその命令の振舞いが記述されている．RRBE は命令完了時の状態が記述されており，TED は RRBE の情報から TDT を参照する．TDT 内のエントリ

© 2009 Information Processing Society of Japan

と RRBE の情報が一致した場合，UDT はトラップを発生させる．なお，RRBE と TDT は，専用命令を使用してトラップハンドラからアクセスを行うことが可能である．このインタフェースを利用してトラップハンドラ（最適化ルーチン）はトラップ条件の設定を行い，プログラム実行状態の把握を行うことができる．

また，TDT の付加機能としてトラップマスク機能がある．トラップマスク機能は TDT の 1 エントリを使って実現される．トラップマスク機能とは，基本的に TDT に登録する条件はトラップを発生させる条件であるが，その条件の例外でトラップを発生させない部分的な条件を設定する機能である．たとえば，「すべての後方分岐でトラップが発生する」条件の TDT の 1 エントリを追加して，「関数呼び出し（Jump and Link），関数戻り（Jump Register）の後方分岐ではトラップを発生させない」条件のトラップマスクを TDT に 1 エントリとして追加した場合，「関数呼び出し，関数戻り以外のすべての後方分岐でトラップを発生」という細かい条件設定を可能にする機能である．これはトラップ発生数を可能な限り抑えるための機能である．

2.2　命令セット変更

本稿では，ソフトウェアオーバヘッドのないデータプリフェッチ最適化のために，命令セットアーキテクチャに 3 種類の命令を加えた．本節ではその詳細を説明する．

2.2.1　逐次ロード/ストア命令

配列はプログラム言語を問わず，多くのプログラムで使用されるデータ構造である．配列の要素は仮想アドレス空間上，等間隔に配置される特性があり，このデータ構造へのアクセスは等間隔アドレスを発行する可能性が高い．このデータ構造へのアクセスに対してデータプリフェッチ最適化を施すために，新しいプリフェッチ命令 *Sequential Load or Store Instructions*（*SLSI*）を加えた．5 章で性能評価を行う際に Alpha 命令セット[18] に追加した SLSI の中で，整数ロードに関するもの中からいくつかを**表 1** に示す．

SLSI は既存のロード/ストア機能と逐次プリフェッチ機能の 2 つの機能を 1 つの命令で行う．SLSI は拡張元となる Alpha 既存命令に対し末尾に_SEQ_POS と_SEQ_NEG で示される 2 つのプリフェッチ方向を示すニモニックを追加した形で表現される．表 1 の説明で示される順方向とはアドレス空間上，上位番地に向けての逐次プリフェッチ機能が付加される意味で，逆方向とは下位番地に向けての逐次プリフェッチ機能が付加されるという意味である．拡張元命令の列は追加命令に対応する Alpha の持つ既存のロード命令を示す．説明の符有，整有の表記は既存命令の持つ符号拡張機能，アライン機能の有無を示している．実際には整数ロードに関する命令の追加は表 1 で示される命令数だけでなく，Alpha に既存の整数ロード命令数の 2 倍の数（POS と NEG）が追加される（18 命令）．また，SLSI はロード命令だけでなく，ストア命令も存在することから，これに加え Alpha に既存の整数ストア命令数の 2 倍（14 命令），さらに，浮動小数ロード/ストア命令に対しても既存命令の 2 倍の数の命令（16 命令）が追加される．このように，Alpha で提供されるすべてのデータ転送命令に対応する逐次プリフェッチ機能を備えた命令（計 48 命令）を用意することで，最適化はプログラムコード内の既存のロード/ストア命令と SLSI を置換することのみで行うことができる．SLSI の命令数を追加することにより，Alpha のデータ転送命令数は 3 倍になるが，すべての SLSI は Alpha 命令セットエンコーディングの分類 *miscellaneous instructions* 内で提供することが可能である．

図 2 に SLSI の実行時の振舞いを示す．図中の LDQU_SEQ_POS は SLSI 命令の 1 つで

表 1　ALPHA 命令セットに追加した整数ロードに関する SLSI の命令
Table 1　SLSIs for integer loads added to Alpha ISA.

追加命令	拡張元命令	説明
LDBU_SEQ_POS	LDBU	順方向 1B 逐次ロード/符無
LDWU_SEQ_POS	LDWU	順方向 2B 逐次ロード/符無
LDL_SEQ_POS	LDL	順方向 4B 逐次ロード/符有
LDQ_SEQ_POS	LDQ	順方向 8B 逐次ロード/符有
LDQU_SEQ_POS	LDQU	順方向 8B 逐次ロード/整無
LDBU_SEQ_NEG	LDBU	逆方向 1B 逐次ロード/符無
LDL_SEQ_NEG	LDL	逆方向 4B 逐次ロード/符有
LDQ_SEQ_NEG	LDQ	逆方向 8B 逐次ロード/符有
LDQU_SEQ_NEG	LDQU	逆方向 8B 逐次ロード/整有

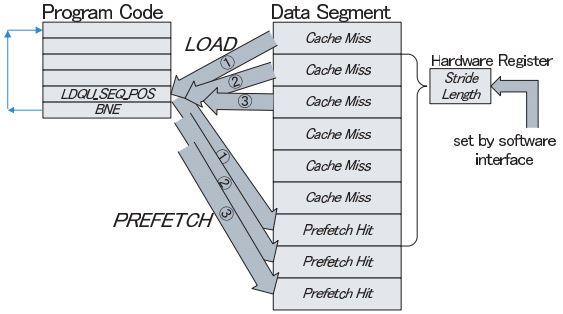


図 2　SLSI の動作例
Fig. 2　Example of behavior of SLSI.

© 2009 Information Processing Society of Japan

ある（アラインなし順方向 8B 逐次ロード）．この命令は以下の 3 ステップを実行する．最初のステップとして，関係するロード命令（LDQU）と同様の機能を実行する．第 2 のステップとして，LDQU 機能で生成した仮想アドレスに定数値を加算することによってプリフェッチするアドレスを計算する．第 3 のステップとして，LDQU_SEQ_POS は計算したプリフェッチアドレスを主記憶に発行する．図中の *Stride Length* はキャッシュブロック数で表現される．図の例では，LDQU_SEQ_POS の通常ロード機能が最初のメモリブロックにアクセスしている間，同命令のプリフェッチ機能が *Stride Length* 数分のキャッシュブロックサイズ先のアドレスに対しプリフェッチを行う．その後，LDQU_SEQ_POS の通常ロード機能が 2 番目のメモリブロックに対しロードを行うときに，同命令のプリフェッチ機能は 2 番目のプリフェッチ対象ブロックの発行を行う．このように，LDQU_SEQ_POS はつねに *Stride Length* 分先のブロックを先見してプリフェッチする．この *Stride Length* の例では，LDQU_SEQ_POS の持つ通常ロード機能は最初に 6 回のキャッシュミスを起こすが，6 回のキャッシュミスが起きた後，キャッシュミスを発生させない．

図 3 に SLSI を実現するためのハードウェア機構を示す．プリフェッチ発行のメカニズムは CPU 実装者の設計依存であるが，図 3 は実装する際のメモリシステムモデルの一例を示している．通常のスーパスカラ・アウトオブオーダプロセッサはストアのインオーダ性の保証とロードのオーバラップおよびストア値のフォワードのため，Load/Store Queue を

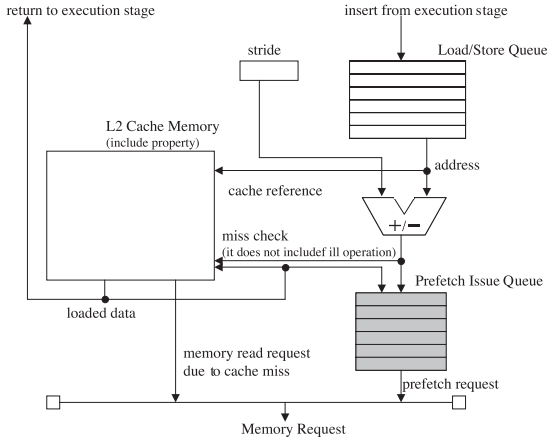


図 3　SLSI/ILI/SILI を実現するメモリシステムモデル
Fig. 3　Memory system model for SLSI/ILI/SILI.

備えている．SLSI は命令実行パイプライン上では，Load/Store Queue に挿入されるまで拡張元の命令とまったく同じ動作をする．通常のロード/ストア命令は Load/Store Queue の出口からキャッシュ参照するためのアドレスを発行するが，SLSI の場合は，それと同時にプリフェッチアドレスを計算するための加算器に参照アドレスを入力する．図 3 に示す stride 値は命令セットエンコーディングの空きの都合上，定数をセットするレジスタであったり，命令フィールドにセットされていたストライド値であったりする．図 3 では，便宜的に固定レジスタから取得する場合を示している．プリフェッチアドレスを計算後，計算結果はキャッシュ参照し，キャッシュが該当アドレスのデータを持たない場合のみ，Prefetch Issue Queue にプリフェッチアドレスが渡される．Prefetch Issue Queue は Queue がオーバフローしない状況で，かつ，挿入されるプリフェッチアドレスがすでに Queue 内に存在していないかを確認し，エンキューする．エンキューしない場合は，計算したプリフェッチアドレスは棄てられる．Prefetch Issue Queue は Queue に入れられた順にメモリに対して読み出し要求を行うが，同時にキャッシュミスが発生し，キャッシュから読み出し要求が出ている場合は，それが解決するまで要求の発行を待つ．なお，4.1 節と 5 章で示される評価のための Simple Scalar への修正はこのメモリシステムモデルに基づいている．

2.2.2　間接参照プリフェッチ命令

C 言語などのポインタを持つ高級言語では，リストやツリー構造など，検索や操作に要する時間を短縮するためにポインタを多用する．ポインタ変数をプログラム内に定義し，アクセスした場合，キャッシュには間接参照先の仮想アドレスが格納される．本稿はこのプログラム特性を活用するため，間接参照プリフェッチ命令 *Indirect Load Instruction*（*ILI*）を対象命令セットに加えた．ポインタをロードする命令として，たとえば，64 ビットアドレス空間を扱うアーキテチャでは，8 バイトロード命令が多用される．Alpha 命令セットでは LDQ/LDQU 命令がそれにあたる．本稿における実装ではアンアライン状態で主記憶上にポインタが配置される状況は想定せず，LDQ のみを対象とし LDQ の機能に間接参照プリフェッチの機能を備えた IND_LDQ の 1 命令を追加した．ILI(IND_LDQ) は通常のロード機能（8 バイトロード）と間接参照先のプリフェッチを行う機能を 1 つの命令で持つ．また，SLSI と同様に，置換操作だけで最適化を施行することが可能であり，SLSI とあわせて *miscellaneous instructions* で提供することが可能である．

ILI は通常のロード命令と特殊な制御フラグを持つキャッシュのフラグ操作を行う命令である．特殊な制御フラグとは，キャッシュブロック内のどの場所にポインタ値を持つかを示すフラグビット（P ビット）である．たとえば，64 ビットアドレスを扱うアーキテクチャ

© 2009 Information Processing Society of Japan

で，キャッシュブロックサイズが 64 バイトの場合，1 つのポインタは 8 バイトなので，1 ブロック内に 8 バイトアラインで最大 8 個のポインタ値を持つことができる．この場合，P ビットは 1 ブロックにつき 8 個あり，それぞれのビットがキャッシュブロック内の位置を示している．ILI は拡張元命令のロード操作を行うためにキャッシュタグ参照を行うが，それと同時に P ビットを立てる．ILI の拡張元命令との動作の違いはこの機能の有無のみである．P ビットを備えたキャッシュは 8 バイトロードでキャッシュが参照されたときに要求されるデータの該当 P ビットが立っているかを判断し，もし立っていれば，ロードしたデータをプリフェッチアドレスとして，まず，キャッシュ内にプリフェッチアドレスが示すデータ含まれないかを確認する．その後，含まれないと判断した場合に，図 3 の Prefetch Issue Queue に渡す．ILI は，まず，P ビットを立ててから，拡張元命令の機能であるロードを行う．このことから，ILI でデータロードを行うと，P ビットの立っていないキャッシュブロックへの初期参照の場合であっても，必ずロードしたデータをプリフェッチアドレスとして発行しようとする．また，1 度でも P ビットが立ったら，ブロックのリプレースが発生するまで消えることがないことから，ILI 以外の命令で P ビットの指す 8 バイトデータをロードした場合でも，同様にプリフェッチ発行を行おうとする．SLSI と同様に，4.1 節と 5 章で示される評価のための Simple Scalar への修正は，このメモリシステムモデルに基づいている．

2.2.3　逐次間接参照プリフェッチ命令

本稿では，SLSI と ILI を組み合わせた *Sequential Indirect Load Instruction*（*SILI*）を提案する．SILI は SLSI と同様の動作をするが，異なる点として SILI は逐次プリフェッチ機能で主記憶からリードして，キャッシュ内にデータ格納する際に，そのキャッシュブロックのすべての P ビットを有効にする機能を有している．これにより，ポインタ配列への配列アクセスであり，かつ，間接参照であるアクセスに対応することが可能となる．Alpha 命令セットにおいて，SILI は ILI と同様に 8 バイトロードに限定されるため，拡張元命令に LDQ を持つ SEQ_IND_LDQ と呼ばれる命令 1 つが追加される．SILI は SLSI，ILI とあわせて *miscellaneous instructions* で提供することが可能である．

3. 最適化アルゴリズム

本章では，メモリ参照解析と最適化を行うソフトウェアの詳細を述べる．最適化ソフトウェアは 2.2 節で述べた 3 種の命令を使い，それぞれの命令に対応した 3 種類の最適化を行う．すべての最適化には 3 段階のステップがある．ステップ 1 とステップ 2 はそれぞれループ解析とアドレス解析を行う．ステップ 3 はそれ以前のステップで得た情報を使い，プログラムコードを修正する．プログラム実行の大半はループ実行で占められる傾向があることから，本システムはすべての処理の最初にループ検出を行う．ステップ 2 はステップ 1 で検出したループを何度か実行し，発生するすべてのメモリアクセスを集計して履歴テーブルを作成する．ステップ 3 は履歴テーブルを解析して，追加命令（SLSI，ILI，SILI）に置換する命令の候補を選別し，バイナリコードに修正を加える．3 つの追加命令の置換作業は，ステップ 1 とステップ 2 は共通であり，それぞれの最適化によってステップ 3 の処理が異なる．

3.1　ステップ 1：ループ検出

ステップ 1 では，まず，HDOS は TDT エントリを後方分岐が実行されたときにトラップを発生させるように設定する．この TDT 設定では，UDT は後方分岐が実行されるごとに最適化ルーチンを呼び出すことになる．このステップの主な処理は後方分岐リストの作成である．後方（仮想アドレス上若いアドレス）に分岐する命令はループバック分岐命令になる可能性があることから，後方分岐リストを生成することによってループバック分岐命令を特定することができる．最適化ルーチンは呼び出された後，専用命令を用いて RRBE を読み出し，そこから得られた PC 値を用い，既存の後方分岐リストを検索する．リストが存在しない場合や既存リストになければ，新しくリストに加える．このとき，関数呼び出し，関数戻りに関係する分岐はリストに加えない．同時に，検索で得られたリスト要素の実行回数項目をインクリメントする．実行回数が与えられた閾値を超えた後方分岐を見つけた場合，その後方分岐はループバック分岐命令であると断定され，ステップ 2 に処理が遷移する．

3.2　ステップ 2：メモリアクセス履歴テーブルの作成と検査

ループバック分岐が検出された後，TDT 設定は以前の設定に加えて，ロード/ストア命令が実行された後にトラップを発生する設定が加えられる．最適化ルーチンは該当ループバック分岐命令が指定された回数分 taken 実行されるまで，このループで発生するメモリアクセスの解析を行う．本稿では，この作業を観測フェーズと呼ぶ．**図 4** は観測フェーズで作成されるメモリアクセス履歴テーブルの例を示す．

図の履歴テーブルは 6 つの項目を持つ．図中の項目を左から順に示す．*PC* はロード/ストア命令の PC 値，*Stride* は等間隔アクセスを示すフラグ，*Variable* は等間隔でないアクセスパターンを示すフラグ，*Stride Length* は等間隔アクセス時のアクセス間隔，*Last Address* は最後にアクセスされたアドレスを示す．最後に *Execute* はそのロード/ストア命令が実行された回数を示す．履歴テーブル内の行は観測フェーズの間，最初にロード/ストアが観測されたときに作成される．そのため，各エントリ（行）は異なる PC 値を持つ．最適化ルー

© 2009 Information Processing Society of Japan

PC	Stride	Variable	Stride Length	Last Address	Execute
0x12000	0	0	0	0x200000	1
0x12100	1	0	8	0x210000	30
0x12200	1	0	4	0x220004	100
0x12300	0	1	0	0x230018	50
⋮	⋮	⋮	⋮	⋮	⋮

図 4　メモリ参照履歴テーブルの例
Fig. 4　Example of memory reference history table.

チンは履歴テーブルを RRBE から取得したトラップの原因となった命令の PC 値を用いて検索し，一致を調べる．一致エントリを見つけた場合，以下の 4 ステップを実行し，履歴テーブル内の該当エントリを更新する．

(1)　今回のトラップの原因となった命令の参照アドレスと *Last Address* を減算し，*Stride Length*（*SL*）を得る．

(2)　得られた *SL* はエントリ内にある古い *SL* と比較される．もし，フラグ *Variable* が 0 でかつ，古い *SL* が 0 であるか（2 回目のアクセス：暫定ストライド），またはフラグ *Variable* が 0 でかつ，得られた *SL* と古い *SL* が等しいとき（3 回目以降で同ストライドが観測された：暫定ストライド継続），フラグ *Stride* は 1 にセットされ，フラグ *Variable* は 0 にセットされ，古い *SL* は得られた *SL* で更新される（暫定ストライド継続時は *Stride*，*Variable*，*SL* の値変化なし）．それ以外の場合は，フラグ *Stride* は 0 にセットされ，フラグ *Variable* は 1 にセットされる．

(3)　*Last Address* を新しく観測された参照アドレスで更新する．

(4)　*Execute* をインクリメントする．

最適化ルーチンはエントリの追加と上記の (1)〜(4) ステップの更新を観測フェーズが終了するまで繰り返す．観測フェーズが終了した後，最適化ルーチンは履歴テーブルを順に検査し観測フェーズ中に逐次アクセスを発生させたロード/ストア命令を特定する．

3.3　ステップ 3：バイナリコードへの修正

SLSI を使用する最適化は，履歴テーブルの *Stride* が 1 で，かつ *Stride Length* が 0 でないロード/ストア命令に対して行われる（図 4 の 3 番目のエントリ）．ILI を使用する最適化は *Variable* が 1 である 8 バイトロード命令に対して行われる（図 4 の 4 番目のエントリ）．SILI は *Stride* が 1 で，かつ *Stride Length* が 8 である 8 バイトロード命令に対して行われる（図 4 の 2 番目のエントリ）．本節の以降の段落は，ステップ 3 の処理をそれぞれの最適化に分けて説明する．また，3 つの最適化は履歴テーブルを調べながら，ステップ 3 で同時に行うことが可能である．

逐次プリフェッチ最適化を行う場合，ステップ 2 の解析が終了した後，最適化ルーチンは逐次アクセスを発生させた命令を SLSI で置換する．

間接参照プリフェッチ最適化を行う場合，ステップ 2 で置換候補が決まった後に，バイナリコードの解析を行う．該当命令の後に実行される命令列をたどり，該当命令でロードされた値の入ったレジスタが，他のロードストアのアドレス計算のベースアドレスレジスタとして使われる場合に，候補のロード命令はポインタをロードしていることが確定する．間接参照プリフェッチ最適化処理は，この条件を満たす候補にのみ，ILI との置換を行う．分岐で追跡ができない場合や，該当レジスタが上書される場合など，ポインタ値としてのレジスタ使用が確認できない場合は，置換を諦める．

逐次間接参照プリフェッチ最適化処理は，逐次プリフェッチ最適化の解析処理中に 8 バイト逐次参照を確認した場合に特別に行う．その該当ロード命令に対して間接参照プリフェッチ最適化のポインタ値使用チェックを行い，ポインタとして使用されていた場合に SILI との置換を行う．

4.　ソフトウェアオーバヘッド

HDOS では，UDT によりトラップハンドラが呼び出され，ソフトウェアで最適化が実現するシステムであることから，最適化対象コードの実行時間のほかに，最適化ルーチンを実行するソフトウェアオーバヘッドが存在する．本章では，HDOS のソフトウェアオーバヘッドについて議論する．HDOS は実行中のバイナリコードに対し，トラップハンドラが最適化を行う．その際に，最適化を行うトラップハンドラ自身の実行がソフトウェアオーバヘッドになる．本提案はソフトウェアプリフェッチの利点が生かされているため，バイナリ生成後の 2 回目以降の実行は，このオーバヘッドなしで実行が可能になる．しかしながら，1 回目の実行の場合でもこのオーバヘッドは許容できるものでなくてはならない．本章では，ハンドラ実行による実行時間の増加をシミュレーション上で得られるトラップ回数と Simple Scalar 3.0 [19] 上で実行したトラップハンドラの実行クロック数から見積もる．

4.1　ソフトウェアオーバヘッド見積り環境

ソフトウェアオーバヘッド見積りのためのシミュレーションは PISA 命令セット [17] 用の Simple Scalar 3.0 を使い行った．シミュレーションの方式はスーパスカラ，アウトオブオーダ実行シミュレーションである sim-outorder で行った．評価は SPEC2000 [20] ベンチマー

© 2009 Information Processing Society of Japan

表 2　SimpleScalar シミュレーションパラメータ
Table 2　SimpleScalar simulation parameters.

issue width	4
decode width	4
ruu size	16
lsq size	8
dl1 size	64 KB
dl1 way	4
dl1 block size	32 B
dl1 access latency	1 cycle
il1 size	64 KB
il1 way	4
il1 block size	32 B
il1 access latency	1 cycle
ul2 size	2 MB
ul2 way	8
ul2 block size	64 B
ul2 access latency	6 cycles
memory access latency	[first]:120 [inter]:12 cycles
memory access bus width	8 B

クの中から 10 のアプリケーションを選択し，行った．バイナリは SimpleScalar のサイト[21]よりプリコンパイルバイナリを取得し，使用した．

Simple Scalar のシミュレーションパラメータを**表 2** に示す．すべての SPEC2000 アプリケーションは 20 億命令完了まで実行した．Simple Scalar 3.0 のシミュレーションモデルに対し，プリフェッチ機能を追加するため，メモリシステムモデルに修正を加え，PISA 命令セット定義に SLSI，ILI，SILI を加えた．また，UDT ハードウェアとトラップハンドラ（最適化ルーチン）はシミュレータコード内に直接記述した．HDOS の最適化時に使用するパラメータは，後方分岐ループバック回数閾値を 100 回，メモリアクセスを観測するためのループバック回数を 100 回とした．使用した UDT エントリは 4 エントリである．本シミュレーションを行うことにより，ソフトウェアオーバヘッドを含まない実行クロックサイクルを得ることができる．

4.2　ソフトウェアオーバヘッド見積り手法

Simple Scalar 3.0 は精密なスーパスカラ・アウトオブオーダ実行プロセッサのパフォーマンスシミュレーションを行うことができるが，トラップ/例外/割込みのハンドリングを含むオペレーティングシステム全体に対してのシミュレーションには対応しておらず，計測対象のアプリケーションのバイナリのコードに対してのみ性能評価を行う．たとえば，アプリケーションが呼び出すシステムコールなどに起因するトラップは Simple Scalar 内でシステムコールエミュレーションが行われ，OS 部のコードのシミュレーションが省略されている．HDOS はトラップを起点に最適化ルーチンを呼ぶシステムであることから，トラップで遷移した OS 部のコードが実行できるシミュレータでなければ，正確にシステム全体のパフォーマンスを計測することができない．

このことから，本節ではソフトウェアオーバヘッドを見積もるために，計測対象アプリケーションの実行クロックサイクル，HDOS に起因するトラップ発生回数，最適化ルーチンの実行クロックサイクルを Simple Scalar で個別に求め，それらの数値を使ってオーバヘッドを見積もる．本稿では以下の式でオーバヘッドを試算する．最適化適用しない実行クロックサイクルを c，最適化適用した場合の実行クロックサイクルを C（オーバヘッドを含まない），HDOS 適用実行時のトラップ数を T，トラップハンドラの実行サイクルを C_h とした場合，HDOS を適用した実行時間の増加比 R_c は以下の式で見積もることができる．

$$R_c = \frac{T * C_h + C}{c} \tag{1}$$

しかしながら，実際の実行では，トラップが発生した状況により，ハンドラの実行サイクル数にばらつきがあり，式 (1) の C_h のように定数値で与えて，精度の高い見積りを行うことが難しい．このことから，3 章で説明した最適化アルゴリズムの特性より，T と C_h のパラメータをどのステップにおいてトラップが発生したか，場合分けを行い，より精度の高い見積りを行う．T と C_h は 3 章のアルゴリズムのステップに対応してそれぞれ T_1，T_2，T_3，C_{h1}，C_{h2}，C_{h3} に分類する．これらパラメータを用いて式 (1) を表現すると

$$R_c = \frac{T_1 * C_{h1} + T_2 * C_{h2} + T_3 * C_{h3} + C}{c} \tag{2}$$

となる．T_1，T_2，T_3 は Simple Scalar で個々のベンチマークごとに計算し，シミュレーション終了時にダンプすることで，容易に得ることができる．本稿では，C_{h1}，C_{h2}，C_{h3} をそれぞれステップごとに独立したプログラムとして Simple Scalar 上で実行し，その実行クロックサイクル数を固定パラメータ値として用いる．ステップ 1 とステップ 2 はそれぞれ，分岐履歴テーブルとメモリアクセス履歴テーブルの更新を行っている．PC 値をキーとしてハッシュ関数でテーブルのインデックスを計算した後，該当エントリに対して指定フィールドの上書きおよびインクリメント処理を行う．また，必要がある場合のみ，エントリの初期化を行う．本稿では，ステップ 1 とステップ 2 に関してワーストケースであるエントリの

© 2009 Information Processing Society of Japan

表 3　見積りに使用する各ステップごとのハンドラ実行クロックサイクルパラメータ
Table 3　Parameters of handler-execution clock cycles for each step in the estimation.

C_{h1}	C_{h2}	C_{h3}
180	790	4928.4

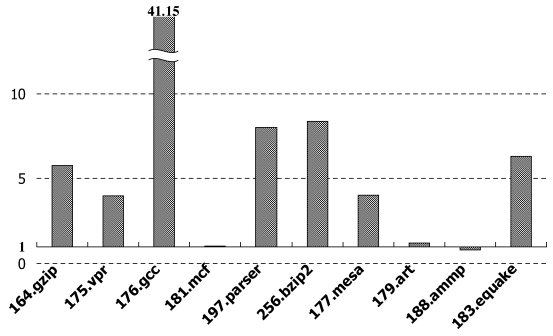


図 5　ソフトウェアオーバヘッドを含む実行時間の増減
Fig. 5　Increase of execution time including software overheads.

初期化を含むパスの実行クロックサイクルをパラメータ値として採用した．ステップ 3 はメモリアクセス履歴テーブルをテーブルウォークしながらチェックし，最適化適用条件に一致したエントリを探す処理である．この処理は与えられるメモリアクセス履歴テーブルの内容により，処理時間が変化する．このことから，本研究ではステップ 3 の実行の入力として与えるメモリアクセス履歴テーブルとして，事前に実行した Simple Scalar 上でのシミュレーションからテーブルスナップショットを 10 個ずつ取り出し，それぞれのスナップショットに対してシミュレーションを実行して実行クロックサイクル数を取得し，平均した．以上の手順で得られた C_{h1}，C_{h2}，C_{h3} のパラメータ値を**表 3** に示す．

4.3　ソフトウェアオーバヘッド見積り評価

表 3 で与えられる固定パラメータ値と Simple Scalar により取得した T_1，T_2，T_3，C，c で計算した最適化適用時の実行時間の変化 Rc を**図 5** に示す．

176.gcc が突出して約 41 倍の実行速度低下となるが，164.gzip，175.vpr，197.parser，256.bzip2，177.mesa，183.equake はおおむね 5 倍〜10 倍までの実行速度低下となっている．また，181.mcf は最適化による速度向上がほぼ完全にオーバヘッドを隠蔽した例であり，179.art は HDOS による割込みが極端に少なく，性能に大きな影響を与えなかった例である．今回の見積りで 188.ammp のみが，若干ではあるが，最適化による速度向上が，オーバヘッドによる速度低下を上回っている．

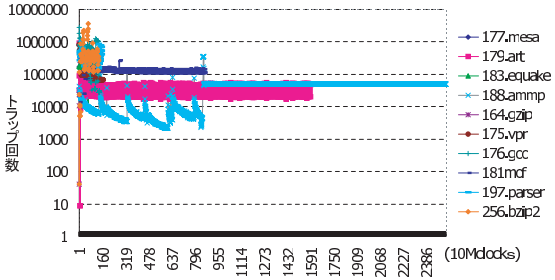


図 6　SPEC2000 における経過クロックサイクル数とトラップ発生数の関係
Fig. 6　Relationship between elapsed clock cycles and trap occurrences in SPEC 2000.

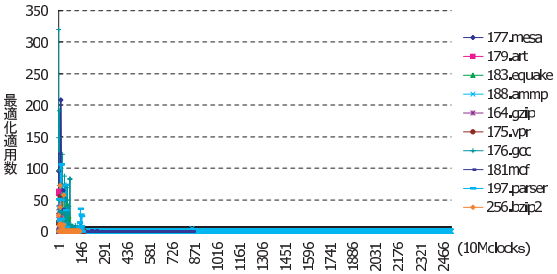


図 7　SPEC2000 における経過クロックサイクル数と最適化適用箇所数の関係
Fig. 7　Relationship between elapsed clock cycles and the number of applications of optimization in SPEC 2000.

4.4　ソフトウェアオーバヘッド削減手法の提案と評価

最適化ルーチン実行による性能低下が，プログラム実行のどの時点で起きているかを知るために，経過クロックサイクル数と UDT によるトラップ発生数の関係を**図 6** に，経過クロックサイクル数と最適化適用数の関係を**図 7** に示す．

それぞれの図は個々のデータではなく，すべてのプログラムを通しての分布に着目する．2 つの図からプログラム実行が開始されて 1460 M クロックの時点に到達するころにはトラップおよび最適化適用数が収束しているのが分かる．ここで考えるべきところは，図 7 から分かるように最適化適用箇所が見つからなくなった場合でも，ループバック分岐命令に対する解析はつねに行われ，トラップは恒常的に発生している点である．そこで，トラップハ

© 2009 Information Processing Society of Japan

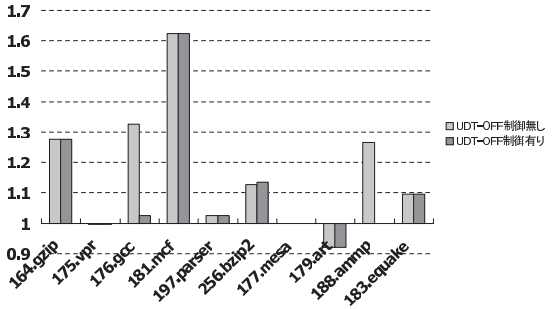


図 8　UDT-OFF 制御を行った場合のソフトウェアオーバヘッドを含まない IPC 性能向上
Fig. 8　IPC improvement without software overheads in case of UDT-OFF.

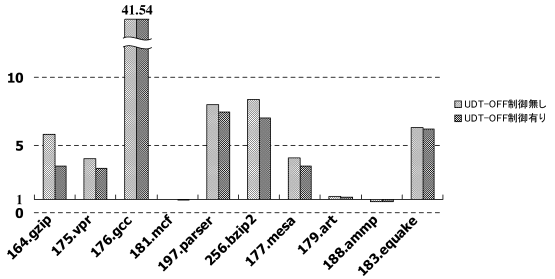


図 9　UDT-OFF 制御を行った場合のソフトウェアオーバヘッドを含む実行時間の増減
Fig. 9　Increase of execution time with software overheads in case of UDT-OFF.

ンドラが最後に行った最適化から，5 億クロック以上の時間が経過して，最適化が行われなかった場合に，UDT のトラップ発生条件をクリア（UDT-OFF 制御）し，これ以上の解析を中断する条件を加えた．この条件を加えた場合のソフトウェアオーバヘッドを含まない IPC 性能向上とソフトウェアオーバヘッドを含めた実行時間の変化をそれぞれ**図 8** と**図 9** に示す．

図 8 より，176.gcc と 188.ammp の性能向上が UDT-OFF 制御により著しく抑えられていることが分かる．これは 5 億クロックの無最適化時間の後に性能向上に大きく影響する最適化候補の命令が検出される予定であったが，それをとりこぼしているためである．それ以外のアプリケーションにおいては，UDT-OFF 制御を加えない場合と比べ，大きな差は見られない．ただし，256.bzip2 のみが UDT-OFF 制御を加えた場合に若干の性能向上が見られる．この性能向上は最適化適用数が減少したことにより，プリフェッチによるキャッシュスラッシングが緩和されたことに起因している．この原因については 5 章において詳しく説明する．

図 9 より，最もソフトウェアオーバヘッドの影響の大きかった 176.gcc に大きな変化が見られなかった．オーバヘッドの変化が少ないうえ，図 8 では，実行速度向上が著しく減少していることから，20 億命令完了時点では，UDT-OFF 制御は性能向上を大きく左右する命令をとりこぼす要因となり，明らかに悪影響を及ぼしている．これは，176.gcc がプログラム実行の全般にわたり最適化を行っており，かつ，オーバヘッドの主たる原因がステップ 3 の実行によることからくるものである．その一方で 176.gcc 以外のアプリケーションではオーバヘッドの減少が確認できる．特に 164.gzip では顕著にオーバヘッドが減少し，約 4 割のオーバヘッド削減に成功している．

UDT-OFF 制御の本質は，大部分の最適化適用箇所候補が見つかり，最適化の必要がなくなったときに，以降の実行のオーバヘッドを削除することにある．本計測では，プログラム実行開始から 20 億命令実行までを観測したが，ほとんどの SPEC 2000 ベンチマークでは計算終了までに，それ以上の計算時間が要求される．つまり，これ以降のプログラム実行を観測していく場合，計算終了までの実行時間が長ければ長いほど最適化による性能向上が得られたアプリケーションはオーバヘッドの割合は相対的に小さくなっていく．逆に最適化により得られる性能向上は恒常的に得られる場合があるので，その場合，最適化による IPC 向上がオーバヘッドを上回る可能性がある．

4.5　本章評価環境と 5 章評価環境との関連性

本章の最後に，次章の性能評価で示す環境と本章で示した環境の違いのを述べ，関連性を示す．

本章の評価では，PISA 命令セットを使用した．これは，ハンドラを生成するためのコンパイラやワークロードとなるプリコンパイル済みの SPEC バイナリを使用する都合によるものである．しかしながら，後の 5 章では，性能評価のために，Alpha[18] 命令セットを使用している．これは，PISA のようなシミュレーション用の架空の ISA でなく，実在する ISA における実装例を示すことに重点を置いたためである．本章と 5 章の評価を関連付けるために，両章で使用した SPEC2000 バイナリのうち，重複する 6 つの各アプリケーションにおける IPC 性能差の平均とデータ参照回数（L1 データキャッシュ参照回数）の差の平均を計算した．IPC 性能で PISA バイナリは Alpha バイナリに比べ，平均で 0.5876%の差，データ参照回数で平均 3.401%の差となった．これは Simple Scalar 上で，両バイナリはほ

© 2009 Information Processing Society of Japan

ぼ同じワークロードとして機能していることを示している．この理由の 1 つとして，PISA が MIPS 命令セットを基にして考案された ISA であり，MIPS 命令セットと Alpha 命令セットが同様の機能を備えた命令を多く持つことがあげられる．特に，今回対象となるロード/ストア命令に関する両 ISA 間の相違点は PISA の持つ拡張アドレッシングモード以外に見つけることができない．これは，RISC の持つ最小の機能単位で命令を構成するポリシによるものである．この特性から，Simple Scalar 3.0 の実装では，両 ISA のオペコードを内部的なオペコードにそれぞれバインドするだけで，命令実行パイプラインのシミュレーション部を両 ISA に対し共通でシミュレーションしている．このことから，本章評価環境と 5 章評価環境はまったく同じではないが，酷似した環境での評価であることが分かる．

5. 性能評価

本章では，本研究と同様にアドレス履歴を用いるプリフェッチ手法に関する関連研究を示し，その後にそれら研究と HDOS の生成したバイナリを比較し評価を行う．

5.1 比較手法

本研究の生成バイナリの性能を他の履歴アドレスを用いたプリフェッチ手法と比較するために，ここでは，いくつかのよく知られてる関連研究を紹介する．本稿では，3 つのアドレス解析手法を関連研究として選択した．Stride プリフェッチと Correlation プリフェッチと Stream プリフェッチである．

Nesbit らは，グローバルヒストリバッファ（GHB）と呼ばれるアドレス記憶機構を使ったデータプリフェッチ手法を提案している[9]．GHB は FIFO 構造のアドレス記録用テーブルを用意し，その FIFO 内の各エントリを指すインデックステーブルを使うことで，FIFO 内のエントリの関連性を記憶しアドレス系列を解析できるようにしている．比較対象として，この GHB を利用したデータプリフェッチの中からストライドアクセスに対応するプリフェッチ手法とアクセス間隔変化の相関性（デルタ・コリレーション）に基づくプリフェッチ手法を選択した．GHB のインデックステーブルの使用法には 2 種類ある．グローバルインデックスとプログラムカウンタインデックスである．グローバルインデックスは対象とする FIFO 内のアドレス系列が，どの命令が発行したアドレスかを関係なく解析する手法で，プログラムカウンタインデックスは PC により FIFO 内をローカライズすることによって，命令ごとのアドレス系列に区分けする手法である．Nesbit らはプログラムカウンタインデックスが良い傾向を示すと主張することから，本稿では比較対象にプログラムカウンタインデックスを選択した．Nesbit らはこれらのプリフェッチ手法を独自の命名規則に従い，それぞれを PC/CS（Program Counter/Constant Stride），PC/DC（Program Counter/Delta Correlation）と命名しているため，本稿ではそれを踏襲する．

Somogyi らはキャッシュメモリの持つ空間的局所性を拡張した Stream プリフェッチ SMS を提案した[10]．SMS は AGT（Active Generation Table）と PHT（Pattern History Table）と呼ばれる 2 つのテーブルを持ち，ストリームプリフェッチのアドレス列を生成する．PHT は主にストリームアドレス列を生成するための起点アドレスとパターンの情報を持ち，AGT は主に PHT に挿入されるストリームプリフェッチのための情報を生成し，PHT に挿入されるエントリをフィルタリングする機能を持つ．1 つのストリームプリフェッチが適用されるメモリ領域の範囲は Spatial Region と呼ばれ，その中で起きる 1 つのストリームアクセスを Spatial Region Generation と呼ぶ．Spatial Region Generation は対象となる Spatial Region 内で過去に起きた時系列のミスアドレス列であり，これをストリームプリフェッチの対象としている．SMS はこの Spatial Region Generation を使い，トリガとなるあるキャッシュミスに関連した Spatial Region 内の複数のキャッシュブロックを 1 回のキャッシュミスでロードすることで，キャッシュの空間的局所性を拡張することを可能にしている．PHT の 1 エントリは 1 Spatial Region Generation に対応し，キャッシュミスが起きたときに，PHT の持つトリガとなるキャッシュミスアドレスが検索され，ヒットした場合に，ストリームプリフェッチが開始される．

本章の以降の性能評価は PC/CS，PC/DC，SMS を比較対象に行う．

5.2 パフォーマンスシミュレーション

性能評価のシミュレーションは Alpha 命令セット[18] 用の Simple Scalar 3.0[19] を使い行った．シミュレーションの方式はスーパスカラ，アウトオブオーダ実行シミュレーションである sim-outorder で行った．シミュレーションパラメータは 4.1 節で示した表 2 と同様である．

評価は SPEC2000[20] ベンチマークの中から 21 のアプリケーションを選択し，行った．バイナリは SimpleScalar のサイト[21] よりプリコンパイルバイナリを取得し，評価に使用した．このバイナリはプリフェッチ最適化が施されていない．すべてのアプリケーションは 20 億命令完了まで実行した．

Simple Scalar 3.0 のシミュレーションモデルに対する修正は 4.1 節で示した修正と同様である．なお，4.1 節におけるシミュレーションは，オンタイムで最適化を行い，最適化のソフトウェアオーバヘッドを除く性能の変化を得ることができるが，後方分岐ループバックの 100 回分の計測とメモリアクセスを観測するためのループバック 100 回分の計 200 回ループ

© 2009 Information Processing Society of Japan

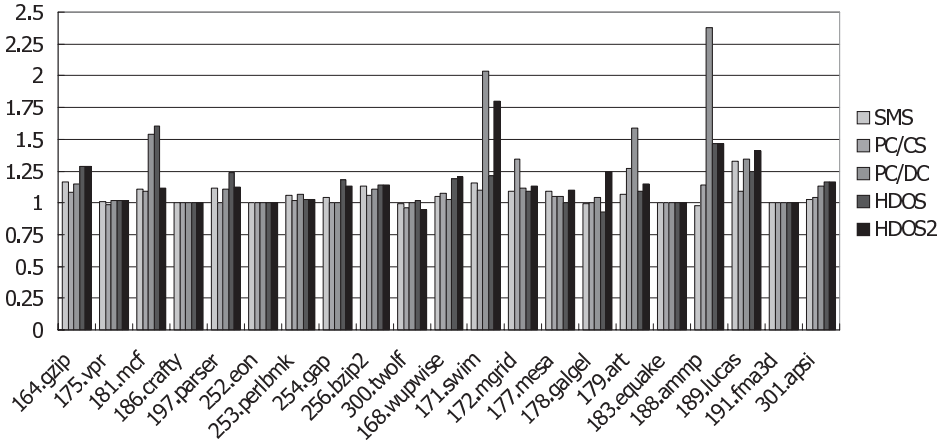


図 10　SPEC2000 における IPC 性能の向上比
Fig. 10　Improvement ratio of IPC in SPEC 2000.

バックしている間，2 回目以降の実行で得られる性能に比べ，最適化される予定のロード/ストア命令のプリフェッチの効果が得られず，その分の性能が異なる．しかしながら，経験的に最適化前に該当命令が実行される回数は最適化後に実行される回数と比べ，きわめて少ないため，この差分はきわめて小さい．本稿では，この差分を誤差として，上記シミュレーション実行で得られる性能向上を 2 回目以降の実行の性能とほぼ同じであると位置づける．

比較対象である他のプリフェッチ手法 PC/CS，PC/DC，SMS の評価に関しても同様にメモリシステムモデルに修正を加え，それぞれの提案ハードウェアをシミュレータコード内に記述して評価を行った．PC/CS，PC/DC それぞれのインデックステーブルおよび FIFO は 256 エントリ，Prefetch Degree は 8 で計測を行った．SMS の spatial region サイズは 8 KB，PHT は 16 K エントリでダイレクトマップによるリプレースメント方式で計測を行った．また，Somogyi らが提案した SMS は L1 キャッシュミスを計測し，ロード先も L1 に行うプリフェッチ方式であったが，オフチップアクセスに効果を限定するため，本計測では SMS を L2 に適用している．

図 10 はプリフェッチなし実行の IPC 性能を 1 としたときの IPC 性能の比を示している．図 10 の濃さの違う棒グラフは左から SMS，PC/CS，PC/DC，HDOS，HDOS2 の計測結果である．HDOS と HDOS2 の違いは，HDOS が Alpha 命令セットで実装した結果で HDOS2 は Alpha 命令セットの制限などがなく，検出したすべての置換候補命令が置換できた場合の結果である．Alpha 命令セットエンコーディングのオペコードの未使用部分が追加命令をすべて実装できるほど空いていなかったため，本計測では追加命令は即値加算をアドレス演算に用いるデータ転送命令の即値の一部分をオペコードとして使用した．このため，狭められた即値フィールドで置換前のロード/ ストアを表現できない場合，最適化の適用を諦めている．このことから，シミュレータ上の環境で，既存の命令セットへの適用における制限がなく理想化した実行の結果が HDOS2 である．つまり，既存の命令セットに追加という形では HDOS2 の実現は難しいが，命令セットエンコーディングを考え直せば実現可能な HDOS のポテンシャルを示している．

表 4　各プリフェッチ手法のハードウェア実装に要するメモリ量の見積り
Table 4　Estimation of the amount of memory required to implement each prefetching hardware.

SMS	347.74 K バイト
PC/CS，PC/DC	4 K バイト
HDOS	99.25 バイト

図 10 に関して，171.swim，179.art，188.ammp において，最も大きい性能の改善幅を示したのは PC/DC である．HDOS，HDOS2 は PC/DC ほどの大きな改善が認められるアプリケーションは少ないが，多くのアプリケーションで他のプリフェッチ手法より性能が向上していることが確認できる．また，この結果から，HDOS で生成したバイナリは他のアドレス履歴を用いるハードウェアプリフェッチと多くのアプリケーションで同等，もしくはそれ以上の性能向上が達成されているといえる．HDOS と HDOS2 について，HDOS2 は HDOS よりも最適化を行える箇所が増えてプリフェッチ数を増やすことができる．特に 171.swim では，その差が大きく出ている．しかしながら，増えすぎたプリフェッチによるロードはキャッシュ内の時間的局所性の高いデータを追い出す可能性が高いことから，181.mcf などに見られるように HDOS より HDOS2 の性能が劣ってしまうケースがある．しかしながら，これは本研究固有の問題ではなく，すべてのプリフェッチ方式に共通した問題で，将来使うデータが高精度で大量に予測できた場合に，それをどうバッファリングするかという問題となる．これに関してはさらに研究が必要となる．

5.3　ハードウェア量の見積り

本稿の第 1 の目的は，ハードウェアプリフェッチ方式のプリフェッチの精度を少ないハードウェアで実現することである．ここでは，SMS，PC/CS，PC/DC，HDOS に用いるメモリ量を 32 ビットアドレス空間に適用する実装を前提として見積もる．見積もったメモリ量を**表 4** に示す．

© 2009 Information Processing Society of Japan

SMS は AGT と PHT の 2 種類のテーブルを必要とする．さらに AGT は構成上，役割に応じて Accumulation Table（AT）と Filter Table（FT）の 2 種類のテーブルで構成される．性能計測では，それぞれ 64 と 32 エントリで評価した．AT の 1 エントリは Tag，PC/offset，Pattern の情報を必要とし，SimpleScalar の設定パラメータと Spatial Region サイズより，それぞれ 19 ビット，45 ビット，128 ビットである．また，同様に FT は Tag と PC/offset が必要とされる．このことから，AT は 1 エントリあたり 192 ビット，FT は 64 ビット必要である．また，PHT は PC/offset，Pattern が必要となり，1 エントリあたり 173 ビットとなる．以上より，評価した 16 K エントリ PHT の SMS の実装のための必要メモリ量は 347.74 K バイト必要となる．

GHB は本稿の性能計測で行ったインデックステーブル 256 エントリ，GHB（FIFO）256 エントリ構成で，PC/CS，PC/DC ともに提案者が論文中で 32 bit-Tag でそれぞれ 4 K バイトと見積もっている．

HDOS のハードウェア UDT は汎用ハードウェアであることから，トラップ発生条件が細かく設定できる構成が様々なアプリケーションでの活用を可能にするが，本稿では提案するプリフェッチ手法の実現のみを対象に UDT を構成した場合の見積りを行う．UDT で必要となる主だったメモリ領域は TDT と RRBE である．RRBE のサイズはリオーダバッファの 1 エントリのサイズで決まる．これは CPU の実装方法により異なり，通常，CPU 設計者以外は知ることができないサイズである．本稿ではこのサイズを SimpleScalar のリオーダバッファの定義から算出した．SimpleScalar のリオーダバッファ定義によれば，RRBE は 1 エントリでおよそ 590 bit 必要である．RRBE は UDT 内に 1 つのみ存在することから，まず，このメモリ容量が必要となる．UDT 内の TDT は複数エントリを持つ構成である．TDT1 エントリに必要な情報は，以前提案した TDT の詳細設計[1] とその後仕様追加されたトラップマスク機能 2.1 から，命令識別ビットパターンが 8 bit，動作条件ビットパターンが 8 bit，エントリの valid ビットが 1 bit，トラップマスク指定ビットが 2 bit，プログラムカウンタ指定のトラップマスクのために 32 bit である．使用したエントリ数はループバック分岐命令検出のために 1 エントリ，トラップマスク処理のために 2 エントリ，ロードストア命令検出のために 1 エントリの計 4 エントリである．以上より，UDT の実装に必要なメモリ量は 99.25 バイトとなる．

上記 3 つのプリフェッチ手法を実装するために必要なメモリ量を比較して，表 4 から明らかなように，HDOS に必要なメモリ量が他の手法に比べ，きわめて小さいことが分かる．実際のハードウェア量は，これらメモリ資源に対する読み書きを制御するためのランダムロジックの増加分も含まれる．UDT の実装の場合，トラップ発生条件のエンコーディングと命令実行パイプラインへの干渉で，ランダムロジックが比較的に複雑で大規模になるが，他の手法とのメモリ量のスケール比から考察して，これはハードウェア量的には誤差程度となる．

この比較から HDOS によるデータプリフェッチは他の手法に比べ，必要とするハードウェア量をきわめて小さくすることを可能にしていることが分かる．これは CPU の遅延設計を容易にし，ダイサイズを小さくすることに貢献する．

6. 関 連 研 究

本章では，ハードウェアとソフトウェアが協調し最適化を行う近年の関連研究を紹介し，本稿との相違点を明確にする．

Lu らは Itanium のパフォーマンスカウンタを用いて，命令ストリーム内にプリフェッチ命令を挿入する動的最適化システム Adore を提案し実装した[11),12)]．Adore はループの持つホットトレース内のデリンクエントロードに着目し，ロードを 3 つのカテゴリ（direct array/indirect array/pointer chasing）に分類し最適化対象とする．文献 11) では，命令挿入のために使用するレジスタを予約しておくために，システムはコンパイラオプションを利用する．文献 12) では，alloc 命令を追加することによって，文献 11) の問題を解決している．また，Adore は pfmon とよばれるプロファイリングツールを用いて，プリフェッチディスタンスをミスレイテンシの平均値と 1 イテレーションを計算する時間で決定することができる．

Luk らは Adore と同じアプローチで，Itanium のパフォーマンスカウンタと pfmon を使用した動的最適化システム Ispike を提案した[13)]．Ispike はプリコンパイルコードを修正するシステムであるが，Adore と同様にレジスタ割当て問題やプログラムセマンティクスの保全を行わなければならいという問題をかかえている．

これらの最適化手法と HDOS の相違点は多数ある．まず，大きな違いとして Adore と Ispike は Itanium に実装することから，最適化ルーチンの実行は Itanium が定義したトリガ条件を満たした場合に発生するイベント発生時に限定される[16)]．Itanium の提供するトリガは，HDOS の提案する命令のコミット単位ではなく，TLB やキャッシュのミス時や分岐予測ミス時などに発生するものに限定され，HDOS より粗粒度のモニタ機能が提供されている．このことから，Adore と Ispike は既存のハードウェアプリフェッチ機構や HDOS が利用することのできる全発行アドレス情報をプリフェッチアドレス計算に用いることができ

© 2009 Information Processing Society of Japan

ない．また，HDOS は最適化ルーチンを明確に OS 側のコードに位置づけることで，Adore と Ispike の持つレジスタ割当て問題がなく，システムユーザに対してトランスペアレントな最適化環境を提供している．また，SLSI/ILI/SILI という命令を置換することによる最適化は Ispike の持つプログラムセマンティクスの保全の問題を持たず，加えてプリフェッチ適用のための実行命令増加がない．

HDOS に最も近いシステムは Trident [14),15)] である．Zhang らはイベントドリブンマルチスレッド動的最適化フレームワーク Trident を提案した．Trident はホットトレースと最適化を行うヘルパースレッドを識別するためにハードウェアサポートを必要とする．ハードウェアイベントで呼び出されるフレームワークのソフトウェアサイドはベースプログラムの最適化を行うか否かを決定する．文献 15) では，デリンクエントロードテーブルを用いてプリフェッチディスタンスを動的に決定する機能が拡張されている．Trident は適切な粒度でイベントを発生させる追加ハードウェアを想定しているが，Trident の実装は SMT プロセッサに限定され，その最適化対象はホットトレースに限定される．HDOS の Trident に対する明確な違いは，最適化結果の再利用性である．Trident は SMT を利用しヘルパースレッドとして毎実行最適化を実行するが，HDOS は直接オリジナルのバイナリを修正することで，2 回目以降の実行は HDOS なしで実行ができる．

7. おわりに

本稿では，軽い汎用のハードウェアを用いる履歴アドレス解析ベースのプリフェッチ手法を提案した．これにより，専用のハードウェアを用いずに最適化オーバヘッドを含まない 2 回目以降の実行で，既存のハードウェアベースのプリフェッチ手法と同様の効果を得られることがシミュレーション結果より得られた．また，初回実行時の最適化を含む実行のソフトウェアオーバヘッドの見積りを行い，このオーバヘッドを削減する手法である UDT-OFF 制御を示した．UDT-OFF 制御を適用することにより，長時間にわたり実行するアプリケーションにおいて最適化のためのソフトウェアオーバヘッドは相対的に小さくなり，プリフェッチによる恒常的な性能向上が大きく得られることを述べた．さらに，性能評価に用いた既存のハードウェアベースのプリフェッチ手法と必要メモリ量の指標でハードウェア規模を見積もり，HDOS に必要なハードウェアが他のハードウェア手法に比べてきわめて小さいことを示した．

謝辞　本研究の一部は科研費若手研究（B）（20700045）「低消費電力高機能リコンフィギュラブルメモリシステムの研究」の一環として行われた．

参考文献

1) 請園智玲，田中清史：ハードウェア解析システムによるバイナリコードの動的最適化，情報処理学会研究報告，ARC, Vol.2005, No.120, pp.7–12 (2005).
2) Bernstein, D., Doron, C. and Freund, A.: Compiler Techniques for Data Prefetching on the PowerPC, *Proc. International Conference on Parallel Architectures and Compilation Techniques*, pp.16–19 (1995).
3) Santhanam, V., Gornish, E.H. and Hsu, W.C.: Data Prefetching on the HP PA-8000, *Proc. 24th annual International Symposium on Computer Architecture*, pp.264–273 (1997).
4) Yeager, K.C.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol.16, No.2, pp.28–41 (1996).
5) Muchnick, S.S.: *Advanced Compiler Design and Implementation*, Morgan Kaufmann, SanFrancisco, CA (1997).
6) Baer, J-L. and Chen, T-F.: Effective Hardware-Based Data Prefetching for High Performance Processors, *IEEE Trans. Comuters*, Vol.44, No.5, pp.609–623 (1995).
7) Dahlqren, F., Dubois, M. and Stenstrom, P.: Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors, *Proc. International Conference on Parallel Processing*, pp.I-56–63 (1993).
8) Fu, J.W.C., Patel, J.H. and Janssens, B.L.: Stride Directed Prefetching in Scalar Processors, *Proc. 25th International Symposium on Microarchitecture*, pp.102–110 (1992).
9) Nesbit, K. and Smith, J.: Data Cache Prefetching Using a Global History Buffer, *Proc. 10th International Syumposium on High-Performance Computer Architecture*, pp.96–105 (2004).
10) Somogyi, S., Wenisch, T., Ailamaki, A., Falsafi, B. and Moshovos, A.: Spatial Memory Streaming, *Proc. 31st Annual International Symposium on Computer Architecture*, pp.252–263 (2006).
11) Lu, J., Chen, H., Fu, R., Hsu, W., Othmer, B., Yew, P. and Chen, D.: The performance of runtime data cache prefetching in a dynamic optimization system, *Proc. 36th International Symposium on Microarchitecture*, pp.180–190 (2003).
12) Lu, J., Chen, H., Yew, P.-C. and Hsu, W.-C.: Design and implementation of a lightweight dynamic optimization system, *J. Instruction-Level Parallelism*, 6 (2004).
13) Luk, C.-K., Muth, R., Patil, H., Cohn, R. and Lowney, G.: Ispike: A post-link optimizer for the Intel Itanium Architecture, *Proc. '04 the International Symposium on Code Generation and Optimization*, pp.15–26 (2004).
14) Zhang, W., Calder, B. and Tullsen, D.M.: An event-driven multithreaded dynamic

© 2009 Information Processing Society of Japan

optimization framework, *Proc. 14th International Conference on Parallel Architectures and Compilation Techniques*, pp.87–98 (2005).
15) Zhang, W., Calder, B. and Tullsen, D.M.: A Self-repairing Prefetcher in an Event-driven Dynamic Optimization Framework, *Proc. International Symposium on Code Generation and Optimization*, pp.50–64 (2006).
16) Intel Itanium 2 Processor Reference Manual For Softwaree Development and Optimization. http://download.intel.com/design/Itanium2/manuals/25111003.pdf
17) http://www.simplescalar.com/docs/ANNOUNCE-3v0c.txt
18) Alpha Architecture Reference Manual, 3rd edition, ALPHA ARCHTECTURE COMMITTEE, Digital Press, ISBN 1-55558-202-8.
19) Burger, D., Austin, T. and Bennett, S.: Evaluating future microprocessors: The simplescalar toolset, Technical Report CSTR-96-1308, Univ. of Wisconsin, CS Dept. (July 1996).
20) http://www.spec.org
21) http://www.simplescalar.com

(平成 21 年 5 月 11 日受付)
(平成 21 年 9 月 7 日採録)



請園　智玲（正会員）
昭和 53 年生．平成 15 年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修了．修士（情報科学）．平成 17 年より北陸先端科学技術大学院大学情報科学研究科博士後期課程に在学．



田中　清史（正会員）
昭和 46 年生．平成 12 年東京大学大学院理学系研究科情報科学専攻修了．博士（理学）．平成 13 年より北陸先端科学技術大学院大学准教授．並列計算機アーキテクチャ，プロセッサアーキテクチャ，メモリシステム，組み込みシステムの研究に従事．IEEE-CS，ACM，電子情報通信学会各会員．

© 2009 Information Processing Society of Japan