データプリフェッチ最適化のためのバイナリ変換手法

請園智玲† 田中清史†

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有り，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．本稿は，その速度差を隠蔽可能なデータプリフェッチ手法を用いた最適化済みバイナリコードを生成することを目的とする．本稿ではプリフェッチ最適化を適用するためにバイナリ変換システム HDOS を提案する．本稿における性能評価では，最適化後のバイナリは SPEC2000 の 21 のアプリケーションで IPC 性能が最大で 28%改善された．

A Binary Code Translation Technique for Data Prefetch Optimization

TOMOAKI UKEZONO† and KIYOFUMI TANAKA†

There is a significant gap between processor's operational speed and memory access speed, and currently the gap is increasing. The primary goal of this paper is to create an optimized binary code using data prefetch technique, which can significantly hide the speed gap. The prefetch tequnique is effective in hiding this speed gap. We propose a new binary translation system for prefetching which is called HDOS. In this paper, the performance results on 21 applications of the SPEC2000 benchmark suite show that the L2 cache miss rate can be improved by up to 5 percent and the IPC performance can be improved up to 28 percent by the proposed optimization.

1. はじめに

近年，ソフトウェア開発手法は多様化し，ダイナミックリンクライブラリやダイナミッククラスローディング技術，JAVA JIT コンパイラのような動的コード生成技術が一般的に使われるようになってきた．また，それらの技術を応用し，コンピュータネットワークを通して，自動的にソフトウェアの差分アップデートや差分配布を行うようになってきている．このようなソフトウェア配布形態において，クライアントにソースプログラムが受給されることは殆ど無く，大抵の場合，ソフトウェアベンダーによるコンパイル済みのバイナリが配布される．一方，近年の CPU の進化に代表されるように，ハードウェアの進歩は著しく，同一の ISA を用いた CPU であっても，計算資源やアーキテクチャ，命令種等が個体により異なる．この機種依存問題に対してコンパイラには機種依存最適化が備わっているが，今日のソフトウェア配布形態において，ソフトウェアベンダーはその適用を行うことが困難である．

本稿では，このようなプログラム配布形態に対応するために，配布済みバイナリを実行環境で最適化し，変換する手法を提案する．この手法で変換したバイナリは共通 ISA に対するバイナリと比べ，最適化を施行した実行環境でのみ動く制約と引き換えに，ハードウェア資源量などの情報や，特殊命令等を用いて高速に実行することができる．また，このような最適化形態が確立すれば，ハードウェアのリリース毎に特殊命令を追加／多用して高速に動作するハードウェアを設計することができ，ハードウェア設計者の制約緩和にも繋がる．

本稿は適用する最適化としてデータプリフェッチ最適化に着目した．データプリフェッチ最適化はプログラム実行中に未来に要求されるメモリアクセスを予測し，事前に主記憶にノンブロッキングロードを発行することで，メモリアクセスレイテンシを隠蔽することができる最適化である．しかしながら，分類的には投機的な最適化に属しており，最適化がうまく適用できないときは，キャッシュメモリの非効率利用やメモリバス占有など重大なパフォーマンス低下を招く恐れがあるため，コンパイラによる機種依存最適化としては適用が困難な部類に入る．本稿ではこの最適化をハードウェア資源情報と，特殊命令を用いてバイナリ変換中に行う．

† 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

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最適化手法はコンパイラによりアセンブリ言語の段階でプリフェッチ命令をコード中に挿入することによって行われる．[2)~4)]．この手法はプログラム毎に変化するメモリアクセス傾向に対し柔軟で，ハードウェアによる特殊なサポートを必要としない．コンパイラによるプリフェッチ最適化としてループ中の大規模配列へのアクセスへのプリフェッチをターゲットとする提案がなされた[2),5)]．これら最適化はメモリアクセスレイテンシを隠蔽する上で大きな貢献がある[3)]．一方で，命令数増加による命令フェッチオーバヘッド，プログラムコード解析の困難さに起因する不要なロードや，効果的でないタイミングでのプリフェッチ発行など，多くの問題を抱えている．

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ロードを発行することによってメモリアクセスレイテンシを隠蔽する方法である．この手法の特徴は，物理アドレスなどの実行時情報を用いてプリフェッチを行うことにある．等間隔アクセスなど，アドレスから判断が容易なプログラム特性を利用するハードウェアプリフェッチ手法が提案された[6)~8)]．これらの提案は参照を解析するためのハードウェアを用い高精度でプリフェッチ発行を行うことを可能とする．しかしながら，遅延設計上，回路の動作周波数を著しく低下させる要因となる大規模の解析ハードウェアを追加することはできない．このため，ハードウェアが行うことが可能な解析は制限される．また，完全に静的なハードウェア実装である特性上，プログラム毎に変化するメモリ参照特性に柔軟に対応することが難しい．

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最適化手法を提案する．本手法はアプローチとしてソフトウェアプリフェッチを採用するが，コンパイラによるプログラムコード解析ではなく，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する仕組みを持つシステムを導入する．本稿ではこのシステムを Hybrid and Dynamic Optimization System (HDOS) と呼ぶ．HDOS において最適化はこのシステムによる解析により得られたプロファイルを用い実行時 (オンライン) に行われる．このシステムにより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され，次回以降の実行でプリフェッチの効果のあるバイナリとなる．本稿ではこのプロセスをバイナリ変換と呼ぶ．HDOS の詳細は次節で述べる．

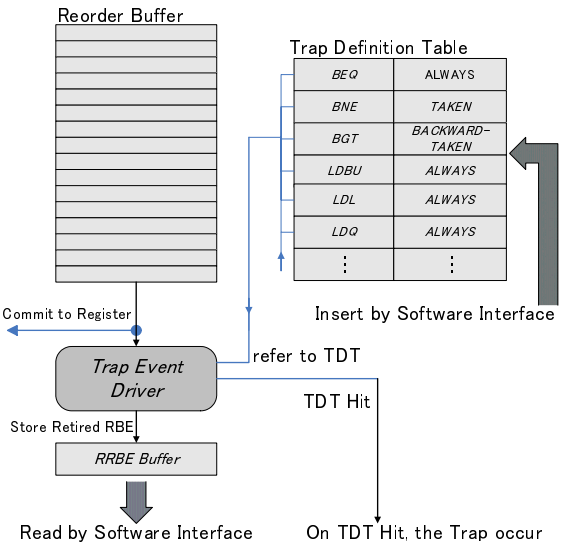

図 1 User Difinable Trap (UDT) ハードウェア．

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．本節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ー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図を図 1 に示す．

図中では，例えば，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．この機能により，解析システムの大部分をトラップハンドラに委譲することが可能となる．このことか

ら，HDOS はプログラム毎に変化する参照傾向に対して柔軟性を持ち，複雑な解析アルゴリズムの適用が可能となる．この利点が同様にハードウェア資源を投入する従来のハードウェアプリフェッチに対しての大きな優位点となる．

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もつ．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 Difinition 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る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 バッファと 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．このインター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．

3. 命令セット変更

HDOS は機種依存最適化を行うシステムであることから，対象 ISA にとって標準でない追加命令セットを自由に扱うことができる．本稿では，データプリフェッチ最適化のために，Alpha 命令セットアーキテクチャ[9] を対象として，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一部説明する．

3.1 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．本稿では，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，Alpha 命令セットに新しいプリフェッチ命令 *Sequential Load or Store Instructions (SLSI)* を加えた．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．そのため，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 は Alpha 命令セットにおいて *miscellaneous instructions* で提供することが可能である．

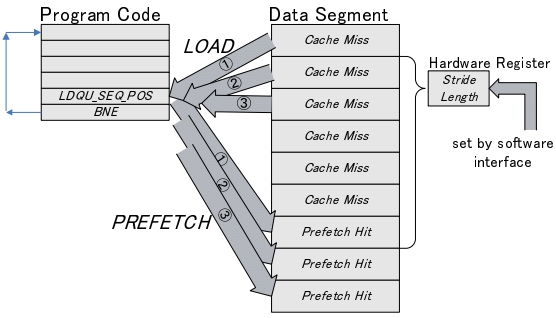

図 2 SLSI の動作例.

図 2 に SLSI の実行時の振る舞いを示す．

図中の LDQU_SEQ_POS は SLSI 命令の 1 つである．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，LDQU_SEQ_POS は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，ロードを行う時に，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 は常に *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．

適切な *Stride Length* 値は対象システムのメモリアクセスレイテンシと対象ロード命令が繰り返し実行される時間間隔で決定されるため，本来は命令内の即値パラメータとして提供されるべきであるが，本稿の評価では Alpha 命令セットの命令フィールドの制限から実行環境で 1 つの値とした．

3.2 間接参照プリフェッチ命令

C 言語などのポインタをもつ高級言語では，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．ILI は通常のロード機能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，*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ィールドをもつキャッシュのフラグ制御を行う命令である．特殊な制御フィールドとは，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．例えば，64 ビットアドレスを扱うアーキテクチャで，キャッシュブロックサイズが 64Byte の場合，一つのポインタは 8Byte なので，1 ブロック内に 8Byte 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 ブロックに付き 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．このキャッシュはアクセスされる際，ヒットしたキャッシュブロックの P ビットが 1 つ以上有効である場合に，P ビットが示す位置のデータ (ポインタ値) を全て読み出し，そのポインタを用いてプリフェッチを発行する．

ILI は最初に P ビットに対し，フラグセットを行い，次にロードを実行するため，一回の実行で間接参照先へのプリフェッチを行うことができる．さらに通常のロード命令で P ビットが有効なブロックがヒットした場合であっても，同様にプリフェッチが発行される．

3.3 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．SILI は逐次プリフェッチ機能で主記憶からロードして，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全ての 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．

4. メモリ参照解析と最適化適用

本節では，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．3 節で 3 種類の命令を Alpha 命令セットに導入した．最適化ソフトウェアはこの 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．全ての最適化には 3 段階のステップがある．ステップ 1 とステップ 2 は命令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．ステップ 1 はループ検出である．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，本システムは全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，発生する全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 種類の最適化を通して，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．以降の段落はこれらの 3 ステップに沿って説明される．

4.1 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毎に最適化ルーチンを呼び出すことになる．その中から PC 値を取り出す．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無ければ，新しくリストに加える．この時，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．

4.2 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を加えられる．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，この作業を解析フェーズと呼ぶ．図 3 は解析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*

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 3 メモリ参照履歴テーブルの例.

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中に最初にロード/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．最適化ルーチンは履歴テーブルを RRBE レジスタから取得したトラップの原因となった命令の PC 値を用い検索し，一致を調べる．一致エントリが見つからなかった場合，上述のとおり，最適化ルーチンは新しいエントリを作成する．一致エントリを見つけた場合，最適化ルーチンは以下の 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，得られた *SL* が 0 であるか，又はフラグ *Variable* が 0 でかつ，得られた *SL* と古い *SL* が等しいとき，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．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．

4.3 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 3 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 8Byte ロード命令に対して行われる (図 3 の 4 番目のエントリ)．SILI は *Stride* が 1 でかつ *Stride Length* が 8 である 8Byte ロード命令に対して行われる (図 3 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．

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．しかしながら，置換候補が単一ループ内に多量に存在する場合，全ての候補を置換することは，許容できないパフォーマンス低下を招く恐れがある．この問題は，キャッシュWAY 数を超えるプリフェッチを単一ループ内で行った場合，キャッシュ内でプリフェッチしたキャッシュブロック同士が，実際に使用される前に，インデックス競合を起し，追い出される可能性があることに起因する．最悪の場合，多量のプリフェッチはキャッシュにスラッシングの危険をもたらす．この問題を避けるために，置換候補は *Stride Length* と *Execute* の積で降順ソートされ，上位のキャッシュWAY 数分が置換されるアルゴリズムを採用する．

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．該当命令がその後に実行される命令列を辿り，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．分岐で追跡が出来ない場合や，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プリフェッチ数制限チェックを行った後に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う．2 つのチェックを通った候補にのみ SILI との置換を行う．

5. 性能評価

本節では HDOS の生成したバイナリの性能評価を行う．

5.1 パフォーマンスシミュレーション

シミュレーションは Simple Scalar 3.0[10] と Alpha バイナリを使い行った．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，Alpha 命令セット定義に SLSI，ILI，SILI を加えた．UDT ハード

表 1 SimpleScalar シミュレーションパラメータ

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

ウェアとトラップハンドラ (最適化ルーチン) はシミュレーションモデルとして実装せず，シミュレータコード内に直接記述した．シミュレータコード内に直接記述してオンライン最適化を行うことで，シミュレータによる性能評価は変換オーバヘッドを含まない評価結果を得ることができる．本稿で追加した命令は置換前の命令と同じ機能をもち，かつ，追加機能を有することから，バイナリ変換前の命令数とバイナリ変換後の命令数は変わらない．よって，性能評価はキャッシュミス率低減による CPU ストール時間の減少がもたらす IPC 性能の変化のみに着目することができる．評価結果は，正確には各ループにつき開始後 200 ループ以内の追加命令の効果が含まれないものであるが，これに起因する性能の誤差は極めて小さいものであり，無視できる．このことから，本稿では本性能評価をバイナリ変換後に生成された最適化済みバイナリの性能の指標とする．本稿における性能の比較対象は最適化を行わないバイナリの性能である．シミュレーションの方式はスーパースカラ，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を表 1 に示す．HDOS の最適化時に使用するパラメータを表 2 に示す．評価は SPEC2000[11] 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[12] 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．全てのアプリケーションは 20 億命令完了まで実行した．

5.2 逐次プリフェッチ最適化の性能評価

図 4 に SPEC2000 ベンチマークにおける SLSI 適用済みバイナリのキャッシュミス率の変化を示す．全

表 2 HDOS パラメータ

threshold of backward loop count	100
loop-back count for mem-access observation	100
stride length(for SLSI,SILI)	20

てのアプリケーションで L2 キャッシュミス率は低下した．特に gzip と parser は大きく低下した．gzip が全ての INT，FP アプリケーション中最大の低下を示し，46%改善している．図 5 に SPEC2000 ベンチマークにおける SLSI 適用後のバイナリの IPC 性能の変化を示す．図中，差が見えにくいアプリケーションがあるが，ほぼ全ての INT アプリケーションについて，IPC パフォーマンスは向上した．gzip が全てのアプリケーション中最大の改善であり，28%改善された．これは gzip の性能が配列参照に大きく依存していることを示している．この結果は配列は多種のプログラムで頻繁に使われており，SLSI の効果が高く，既に配布済みバイナリに適用した場合，高確率で性能向上が見込めることを示している．また，対象ベンチマークプログラム全てにおいて SLSI 適用による性能低下がなかったことから，この結果は投機的な最適化でありながら，実行時情報のフィードバックにより，プリフェッチ精度が高く保たれたことを示している．

5.3 間接参照プリフェッチ最適化の性能評価

図 6 に SPEC2000 INT ベンチマークにおける ISI 適用済みバイナリのキャッシュミス率の変化を示す．FP ベンチマークは，最適化適用箇所がないアプリケーションが多く，適用できたアプリケーションであっても，性能値変化が観測できなかったため，示していない．これは FP ベンチマークではポインタを多用する複雑なデータ構造を使っていないことに起因していると考えられる．ILI の適用により，全ての INT アプリケーションで L2 キャッシュミス率は低下した．特に 181.mcf では大幅にキャッシュミス率が低下している．図 7 に SPEC2000 ベンチマークにおける ILI 適用後のバイナリの IPC 性能の変化を示す．181.mcf, 153.perlbmk, 254gap では性能の向上が見られる．これらアプリケーションは間接参照よるデータアクセスが多かったことがわかる．ILI は SLSI と同様に適用による性能低下がなかった．

5.4 逐次間接参照プリフェッチ最適化の性能評価

図 6 に SPEC2000 INT ベンチマークにおける SILI 適用済みバイナリのキャッシュミス率の変化を示す．FP ベンチマークは，ILI の評価結果と同様の理由で示していない．SILI はポインタ配列へのアクセスに特化しているため，ILI の性能向上に比べて低いが，181.mcf のように大幅な性能向上を持つものがあった．この性

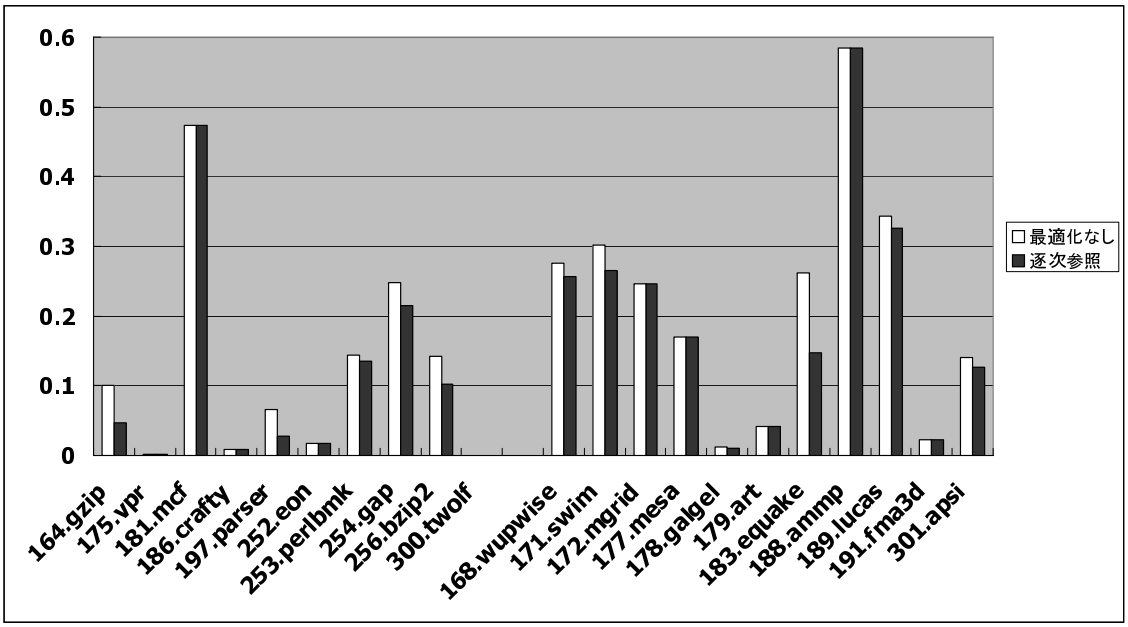

図 4 SPEC2000 における L2 キャッシュミス率 (SLSI 適用).

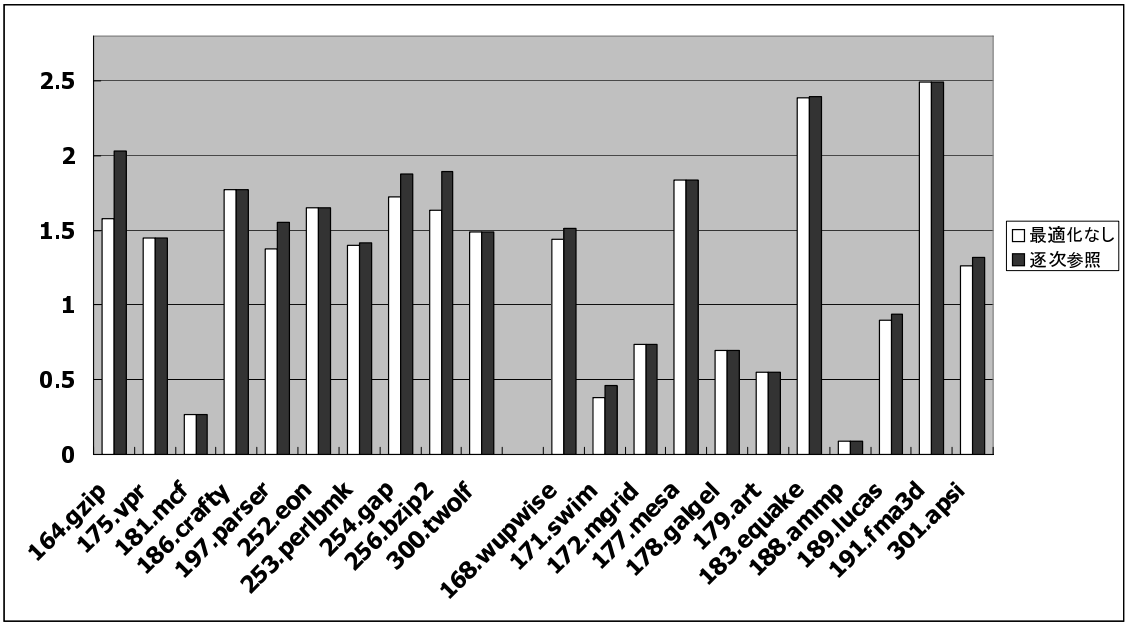

図 5 SPEC2000 における IPC(SLSI 適用).

能差は 181.mcf がポインタ配列参照に性能が大きく依存しており，SILI が ILI と比べ時間的に余裕をもって間接参照先をプリフェッチできることに起因していると考えられる．SILI は ILI，SLSI と同様に適用による性能低下がなかった．

6. おわりに

本稿は近年のソフトウェア配布形態に対応するため，機種依存最適化を行うバイナリ変換手法を提案した．また，その適用例として，データプリフェッチ最適化に対しての評価を行った．ハードウェアとシステムソフトウェアで構成される実行時最適化環境 HDOS を提案した．HDOS はソフトウェア最適化とハードウェアプリフェッチの両方の利点を持ち高精度でかつ柔軟な最適化を行うことが出来る．機種依存命令として各データアクセス傾向に特化したプリフェッチ命令を定

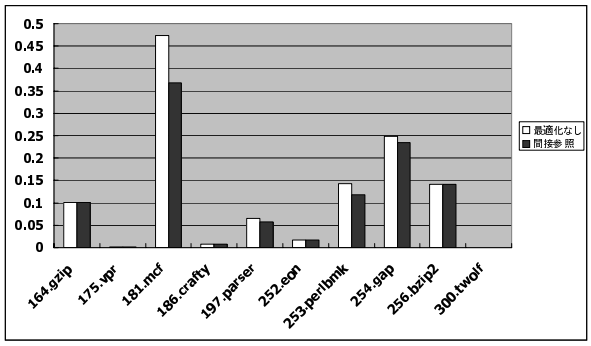

図 6 SPEC2000 INT における L2 キャッシュミス率 (ILI 適用).

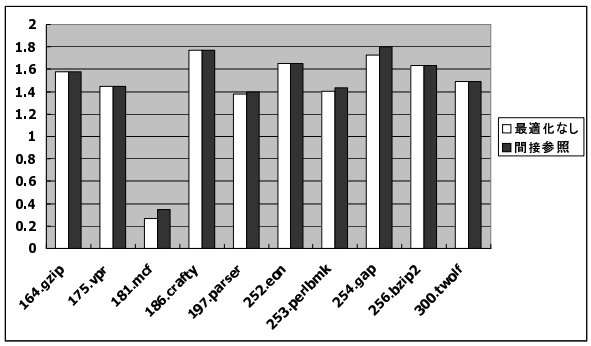

図 7 SPEC2000 INT における IPC(ILI 適用).

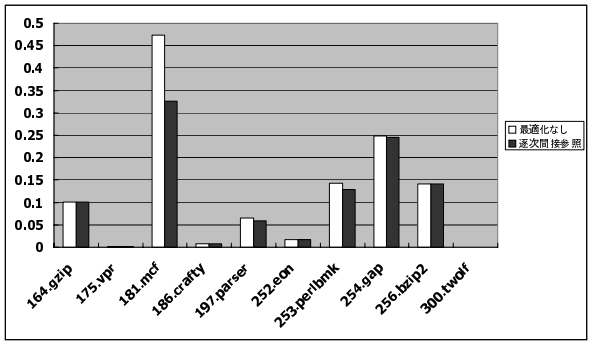

図 8 SPEC2000 INT における L2 キャッシュミス率 (SILI 適用).

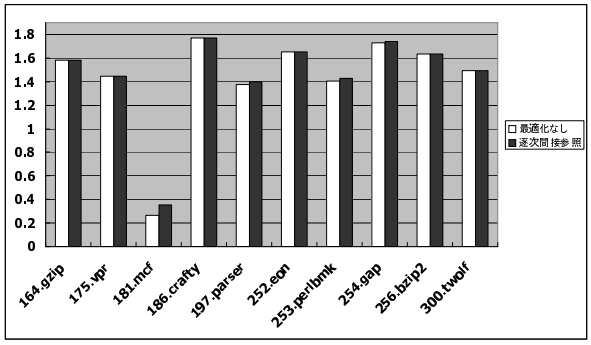

図 9 SPEC2000 INT における IPC(SILI 適用).

義し，それを適用することで最適化を行うデータプリフェッチ最適化の手法を提案した．これら提案を用いることで，既に配布済みのバイナリに対し，最適化が可能となる．本稿で評価したプリフェッチ最適化ではいくつか SPEC2000 ベンチマークプログラムで性能向上が認められた．本稿で提案した HDOS は動的最適化環境としても利用可能であり，トラップハンドラを呼び出すオーバヘッドが十分に小さい最適化を行う場合に利用できる．今後の課題として，HDOS の動的最適化への適用を目標として，最適化に要するオーバヘッドの低減とオーバヘッド時間の計測を行いたい．

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., C.Doron and A.Freund. Compiler Techniques for Data Prefetching on the PowerPC. Proccedings of international conference on parallel architectures and Compilation Techniques, pp.16–19, 1995.
3) V.Santhanam, E.H.Gornish and W.C.Hsu, Data Prefetching on the HP PA-8000, Proc. of 24th annual International Symposium on Computer Architecture, pp.264–273, 1997.
4) K.C.Yeager, 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, San-Francisco, CA, 1997.
6) J-L.Baer and T-F.Chen, Effective Hardware-Based Data Prefetching for High Performance Processors, IEEE Transactions on Comuters, Vol.44, No.5, pp.609–623, 1995.
7) F.Dahlqren, M.Dubois, and P.Stenstrom, Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors, Proc. of International Conference on Parallel Processing, pp.I-56–63, 1993.
8) J.W.C.Fu, J.H.Patel and B.L.Janssens, Stride Directed Prefetching in Scalar Processors, Proc. of 25th International Symposium on Microarchitecture, pp.102–110, 1992.
9) Alpha Architecture Reference Manual, THIRD EDITION. ALPHA ARCHTECTURE COMMITTEE, Digital Press, ISBN 1-55558-202-8.
10) D.Burger, T.Austin, and S.Bennett. Evaluating future microprocessors: The simplescalar toolset. Tech Report CSTR-96-1308, Univ. of Wisconsin, CS Dept., July 1996.
11) http://www.spec.org
12) http://www.simplescalar.com