博 士 論 文

# ハードウェア解析システムによる
# 実行コードの動的最適化に関する研究

指導教官　田中 清史 准教授

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

請園 智玲

2010 年 3 月

## 要 旨

本論文では、ソフトウェアで動的にトラップ発生条件を設定できるハードウェアを提案し、そのハードウェアを用いたオペレーティングシステムによるバイナリコードの動的最適化手法を提案している。提案するトラップハードウェア機構はUDTと呼ばれ、指定の命令が指定の動作をする場合に、オペレーティングシステムが予め用意したトラップハンドラに例外処理として実行を遷移させる機能を持つ。オペレーティングシステムのトラップハンドラとして提供される最適化ルーチンは、UDTから呼び出され、呼び出す原因となったプログラムバイナリを修正する。UDTと最適化ルーチンをあわせたシステムをHDOSと呼ぶ。本論文では、最適化ルーチンとして、プリフェッチ最適化を提案し、その評価を行う。本論文の評価で、UDTを利用した履歴ベースのプリフェッチ最適化は同様の履歴ベースのハードウェア実装のプリフェッチと比べ、 極めて少ないハードウェア量で同程度の性能が実現されたことを確認した。更に、本論文ではUDTの汎用性を示すため、UDTの最適化以外への2次利用としてハードウェアデバッグアシストを検討し、プリフェッチ最適化以外の最適化への適用を議論する。

# 目 次

# 第 1 章

# 序論

本章では，はじめに，コンパイル時の最適化で対応できないソフトウェア開発及びソフトウェア利用形態を研究背景として紹介し，次節の実行バイナリ最適化の必要性へと結ぶ．次に，実存するコンピューティングシステム上での実行バイナリ最適化の例を挙げ，その利点を述べると共に，問題点を示す．その次節では，示した問題点を解決するための本研究の目的と本研究で提案する動的最適化システムの概要を示す．本章の最後の節で本論文の構成を示す．

## 1.1 研究背景

本節では，近年のソフトウェア開発／利用形態で使用されるようになったDLL／DCL 技術，JIT 技術の背景と特長を説明し，その共通の問題点と解決手法を示す．

### 1.1.1 DLL/DCL 技術

近年，ソフトウェア開発手法は多様化し，Dynamic Link Library(DLL) や Dynamic Class Loading(DCL) 技術がソフトウェアベンダによって一般的に使われるようになってきた．DLL はライブラリ内で必要な関数又は変数を最初に使うタイミングで主記憶上にロードする技術である．結果的にプログラムの実行から終了までの間に実際に使われた関数のみが主記憶にロードされることになり，主記憶の効率利用が可能になると共に，ソフトウェア配布形態として，実行ファイルとは

別に関数コレクションとしてファイル化できる特徴がある．現在では，供給される大規模アプリケーションの殆どがこの技術を採用している．DCL は Java ／ C++ 等のオブジェクト指向型言語で用いられる技術である [1][2]．DLL の関数ロードと同様に，クラスのローディングを動的に行うことができる．ロードのタイミングは実装依存である．DLL と違い，ライブラリをファイルとしてのみ持つのではなく，ネットワーク経由でのクラスのロードを行うことができることから，注目を集めている．DLL ／ DCL は，非常に短いスパンでのソフトウェアの開発が要求される昨今，ステイブルで，かつ，メモリ資源を有効利用できるアプリケーション非依存共有ライブラリの利用がソフトウェア開発効率を大きく左右すること，更に，ソフトウェアアップデート時の即時差分配布を容易にする利点を持ち，機能強化時及び不具合修正時にソフトウェアベンダにとって有用となることから，積極的に利用されている．

### 1.1.2 JIT 技術

また，Microsoft .NET Framework(以降.NET と呼称する) や Java のもつ JIT コンパイラのような動的バイナリコード生成技術も同様に一般化してきている．.NET 言語や JAVA 言語はアプリケーション開発において，現在，大きな普及を見せている．しかしながら，その普及はその言語が言語として特別に優秀なものであることから普及した訳ではなく，フレームワークを提供するベンダが優秀な共有ライブラリを提供することで広まった．これは先に述べた，ソフトウェア開発速度がアプリケーション非依存共有ライブラリに依存することに起因し，その言語を使用するだけで，この問題が解消できると言うことは，ソフトウェアベンダにとって大きな利点を齎すからである．これらの言語の使用は優秀な共有ライブラリが提供される代わりに，.NET CLR(Common Language Runtime) や Java-VM(Virtual Machine) 等，特殊な実行環境上での実行を強制される．以降，本章ではこれら実行環境を仮想プラットフォームと呼称する．仮想プラットフォームは通常，中間言語 (Java ではバイトコード) で実行ファイルが供給される．これは，実行プラットフォーム非依存のアプリケーション開発や言語非依存のアプリケーション開発に役立つ．しかしながら，現在使用されているコンピューティングシステムにお

いて，広く使われる商用アプリケーションを動作させるプラットフォームは仮想プラットフォームを必要とするほど多くはない．これは，団体／個人問わず，コンピューティング環境として，ハードウェア・ソフトウェア共に標準化しているプラットフォームを採用する方が，低コスト性と保守性に優れるからである．また，言語非依存開発の有用性においても，仮想プラットフォームを採用するプログラミング言語を付属するライブラリ抜きで評価したとき，それまでの手続き型言語と比べ，表現力に大差は認められないことから，一定以上のスキルを持つプログラマにとって，一つのアプリケーション開発に多種の言語を用いることは魅力的ではない．結論として，ソフトウェアの開発速度を左右するライブラリ提供が，これら仮想プラットフォーム言語の普及を支えているが，これは仮想プラットフォーム言語を採用する本質とは異なるものである言える．また，仮想プラットフォームは中間言語を解釈して実行する処理を行う特性上，仮想プラットフォーム上でのアプリケーション実行は，ネイティブコードの実行に比べ，著しく遅く，ソフトウェアベンダにとっては，優秀なライブラリを使用できるメリットを大きく損なわせる．このことから，動的バイナリコード生成技術が開発された．これは，中間言語でできたプログラムの最終生成物を仮想プラットフォーム上で，オンタイムにネイティブバイナリに変換する技術，所謂，Just-In-Time Compilation(JIT コンパイル) 技術である．仮想プラットフォームの持つ JIT は中間言語を翻訳した結果のキャッシュを持つことで，再度実行されるコード部の翻訳を省略でき，仮想プラットフォームのオーバヘッドを軽減できる．

### 1.1.3 DLL／DCL／JIT 技術の問題点と解決手法

これら DLL，DCL，JIT 技術は，1.1.1，1.1.2 で示したように，実行時にバイナリコードが確定するため，コンパイル時に行われる静的コード最適化技術では関数のインライン展開やコード配置最適化等，モノリシックプログラムに適用できた広いスコープの最適化を施すことができない．この問題を解決するために実行時に得られる情報をフィードバックしてバイナリコードに最適化を施す技術が研究されてきた．この技術を本研究では実行バイナリ最適化と位置づける．次節では実行バイナリ最適化の実現手法を 2 つ挙げ，それぞれの手法が実際のコンピュー

タシステムに実装された例を述べ，その利点と欠点を述べる．

## 1.2 実行バイナリ最適化

実行バイナリ最適化には大きく分けて 2 つの手法がある．

1. オフライン最適化 (プロファイルベース最適化)
2. オンライン最適化 (動的最適化)

本節ではそれぞれの手法の利点及び欠点を述べる．

### 1.2.1 オフライン最適化

オフライン最適化はプロファイルベース最適化とも呼ばれる．本章では，後述のオンライン最適化とその特性による違いを明確にするため，オフライン最適化と呼称する．オフライン最適化はプロファイルと呼ばれる実行バイナリに得られる情報をシミュレータ等であらかじめ実行して取得し，そのプロファイルを基にバイナリに最適化を施す手法である．本手法が実際に利用されている例として，Hewlett-Packard 社の Itanium ベースシステム用 HP C Compiler[11] が挙げられる．HP C Compiler は “+Oprofile” というコンパイルオプションを持ち，“ +Oprofile=collect” と指定することで，実行時情報計測用実行ファイルを作成する．この計測用実行ファイルを起動すると，自動的にプロファイルデータを収集し，計測結果をファイル “flow.data” として残す．このデータを用い，再度，コンパイルオプション “+Oprofile=use” を指定してコンパイルすることで，flow.data ファイル内プロファイルを用いて実行バイナリ最適化を行い，最終生成物の実行バイナリを作成する．HP C Compiler の持つ上述の実行バイナリ最適化の機能は PBO と呼ばれる．PBO が行う最適化として，変数の優先レジスタ割付変更，コード配置変更，インライン化，投機実行の適用等が挙げられる．また，HP C Compiler と同様の最適化を目的に持つ研究分野では，Software Trace Cache[5] や Hot Path Prediction[4] 等が先行研究として挙げられる．オフライン最適化の利点は実行時でなくオフライン時に最適化を行うため，最適化のためのオーバヘッドを考慮する必要が無い．し

かしながら，予備実行を行い最適化を行うプロセスは入力データが随時変更される場合に実行環境にとって大きな負担となる．また，CPU の動作は複雑であることから，CPU 内全ての振る舞いをプロファイルとして取得すると膨大なデータとなる．本論文の後の章で紹介するデータプリフェッチ最適化のためのプロファイル収集の例では，SPEC CPU 2000 ベンチマークの 20 億命令の実行中に発生するメモリアクセスの履歴だけでも 1 アプリケーションで，10 ギガバイトのオーダに達した．実際のプログラム実行において，20 億命令実行は数秒の出来事であり，それ以上続くプログラム実行において，これ以上の大量のプロファイルデータを収集・格納することは非現実的である．このため，取得するプロファイルは統計的なものとなる．これは最適化の効果に大きく影響すると共に，適用する最適化アルゴリズムを制限することになる．

### 1.2.2 オンライン最適化

オンライン最適化は動的最適化とも呼ばれる．本章では，前述のオフライン最適化と対比をとるためにオンライン最適化と呼称する．オンライン最適化は実行中のバイナリコードを実行中に変更する方法である．この方法を実現するには，ハードウェア又はソフトウェアの実行環境に対するサポートが必要となる．最適化が実行中に行われる利点として，細かな入力データの変化などで生じる非効率に即座に対応できる利点がある．加えて，CPU 内の実行状況を知るための情報は膨大なものとなるが，その情報を状況に応じ取捨できる利点がある．一般的に利用されているオンライン最適化の例では，1.1.2 節で紹介した，仮想プラットフォームの持つ JIT コンパイラが挙げられる．JIT は，中間言語の翻訳結果をキャッシュとして持つが，同時に，主記憶に配置する前にコードに対し最適化を行うことができる．最新の.NET CLR の JIT 最適化は，構造体を入出力とする関数のインライン化を可能としている．また，分岐予測を用いて命令ブロック移動 (コード配置最適化) を行う事でキャッシュメモリの効率利用を可能とし，ASP.NET のいくつかのベンチマークで 10%を超えるスループットの改善があると報告されている．[6] Java-JIT コンパイラは Kaffe OpenVM[10] 等オープンソースの JAVA-VM 環境が提供されているため，研究分野で，多くの最適化アルゴリズムが検証されている

[7][8]．菅沼らの IBM JIT Compiler は，末尾再帰呼出削除，仮想関数インライン展開，ループバージョニング等の最適化アルゴリズムを実装している [9]．しかしながら，これらのオンライン最適化は仮想プラットフォームに備わるもので，JIT で軽減されているとはいえ，それ自身のオーバヘッドは存在し，非効率な実行を強いられる．加えて，2005 年以降，65nm より微細なプロセスで製造される CPU は，発熱と消費電力の問題から，徐々に動作周波数が低下し，32nm プロセスで製造される現在では，3GHz 近辺で頭打ちとなっている．これは，現在の微細加工技術で 1 コア当たりの演算能力が限界に達したことを示している．このことから，将来的に，現在より更に高機能で高いオーバヘッドを要求する JIT の開発は難しくなっていくと考える．

## 1.3 研究の目的

1.2 節で紹介した 2 つの実行バイナリ最適化の手法のどちらも，実行時情報をフィードバックして最適化を行うため，1.1.3 節で述べたの静的なコンパイル時の最適化スコープの問題を解決可能である．しかしながら，それぞれに問題点がある．両手法の利点と欠点は相関しており，まとめれば,「オンライン最適化はオフライン最適化の問題を含まないが，現在利用されているオンライン最適化技術では，仮想プラットフォーム自身が大きなオーバヘッドとなっている」ということがわかる．

このことから，本研究では，オンライン最適化のオーバヘッドの問題を解決するため，CPU がネイティブバイナリを実行する環境で，ハードウェアとオペレーティングシステム (以降 OS と呼称する) が協調し，主記憶上のバイナリコードを修正する動的最適化システムの提案をする．実行するアプリケーションをネイティブバイナリとして，CPU 上で実行することから，1.2.2 節で述べた仮想プラットフォームの持つオーバヘッドを削除できる．提案システムは UDT(User Definable Trap) と呼ばれる特殊なトラップ発生ハードウェアと，トラップハンドラとして実装される最適化ソフトウェアの 2 つのパートに分けられる．最適化ソフトウェアは UDT の発生するトラップからのみ起動される．また，UDT は最適化ソフトウェアの利用する実行時情報を提供する機能を持つ．本研究では提案するハードウェア UDT

と最適化ソフトウェアを合わせた最適化システムを HDOS(Hybrid and Dynamic Optimization System) と呼ぶ.

UDT は最適化ソフトウェアが必要な時にだけ，アプリケーションのネイティブ実行を中断し，最適化ソフトウェアを呼び出す．その後，最適化ソフトウェアは必要な処理を終えると，再びアプリケーションのネイティブの実行を再開する．この繰り返しを行うことで，オンライン最適化のオーバヘッドを最適化のための計算に限定することができる．HDOS の特長及び特性は次章のハードウェア協調最適化の章で他の最適化システムと共に紹介する．

## 1.4 本論文の構成

本章では，本論文の背景と研究目的について述べた．次に，第 2 章では，まず，他のハードウェアと協調して行う最適化システムを紹介し，HDOS の特性を示し，次に本章で紹介した.NET CLR や Java-VM を除くオンライン最適化の関連研究について述べる．第 3 章では，本論文を通し，共通のインフラストラクチャとなるハードウェア，UDT の詳細について述べる．第 4 章では，HDOS の最適化アルゴリズムの例として，プリフェッチ最適化を提案し評価する．第 5 章では，HDOS に関しての追加情報として UDT の初期化及びプログラムアップロード、UDT のデバッカサポート利用、HDOS のソフトウェアトレース最適化の適用例、HDOS の電力最適化の適用例を示す．最後に第 6 章で本論文の研究成果をまとめる．

# 第 2章

# 関連研究

本章では，HDOS の特長を他のハードウェア協調の動的最適化システムと比較し述べる．第 1 節で明らかにするのは，各最適化システムの採用する最適化アルゴリズムの優劣ではなく，最適化システムとしての特性である．第 1 節では，まず，最初に比較対象とする ADORE[29][30]，Ispike[61]，Trident[63][64] について述べ，第 2 節でそれらと HDOS の比較を行う．それに加え，第 3 節では，本研究と関連する研究として，バイナリ変換技術と第 1 節で示したハードウェア協調最適化システム以外の動的最適化を紹介する．

## 2.1 ハードウェア協調最適化システム

本節では，先行研究で発表された比較対象の動的最適化システムを紹介する．

### 2.1.1 ADORE

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(ADaptive Object code REoptimization) を提案し実装した．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 [29] 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 [30] では，alloc 命令を追加することによって，文献 [29] 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．

### 2.1.2 Ispike

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 post-link 最適化システム Ispike を提案した．Ispike はコードとデータのレイアウトを動的に変更することによってキャッシュの局所性を有効に活用する最適化を行う．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り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問題を抱えている．

### 2.1.3 Trident

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 [63]．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 [64] 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．

## 2.2 動的最適化の特性と他の動的最適化システムとの比較

本節では前節で紹介した先行研究の動的最適化システムと HDOS を比べ，システムとしての特性を示す．

各動的最適化システムのハードウェア実装とソフトウェア実装を表 2.1 に示す．

HDOS は 1.3 節で述べたように，ハードウェア部は UDT と呼ばれるトラップ発

表 2.1: 各ハードウェア協調動的最適化の実装

| | HDOS | ADORE | Ispike | Trident |
|---|---|---|---|---|
| H/W | UDT | PMU | PMU | Original H/W |
| S/W | Trap Handler | Thread & Signal Handler | Developer Tools | SMT Helper Thread |

生ハードウェアで実装され，最適化ルーチンは UDT から起動されるトラップハンドラとして実装される．ADORE は Itanium の持つ PMU のイベントを用い，PMU から起動される Signal Handler とそのシグナルハンドラから呼ばれる最適化用のスレッドからなる．Ispike は Itanium の持つ PMU のイベントを用い，PMU から起動される開発者用の最適化ツールである．Trident は独自のハードウェア (Hot Path Profiler，Hot Value Profiler，Cache Access Counters 等) がプログラム実行の解析を行い，適切なタイミングで SMT 上で Helper Thread として実装される最適化ルーチンを起動する．

HDOS，ADORE，Ispike，Trident を以下の 5 つの指標で比較する．

**1** イベント粒度

**2** ハードウェア情報の利用

**3** 最適化結果の再利用性

**4** 安全性

**5** 最適化処理の透過性

イベント粒度とは最適化に適切な粒度でソフトウェアを実行するイベントが発生するかという指標である．ハードウェア情報とは用意されたハードウェアからソフトウェアが最適化に適切な情報を得ることができるかという指標である．最適化結果の再利用性は動的最適化システム無しで一度施した最適化が以降の実行で有効であるかという指標である．安全性とは悪意のあるソフトウェア修正者にとって容易に修正を許さないか否かという指標である．最適化処理の透過性とは最適化処理自体がシステムユーザ及びプログラマにとって透過性のある処理であ

表 2.2: 各ハードウェア協調動的最適化の特性

| | HDOS | ADORE | Ispike | Trident |
|---|---|---|---|---|
| イベント粒度 | ○ | △ | △ | ○ |
| H/W 情報 | ○ | △ | △ | ○ |
| 再利用性 | ○ | △ | ○ | × |
| 安全性 | ○ | × | × | ○ |
| 透過性 | ○ | ○ | × | ○ |

るかという指標である．以上の比較指標毎のそれぞれの動的最適化環境の特性をまとめた表を表 2.2 に示す．

それぞれの項目 (行) に対し，表 2.2 では○，△，×の 3 段階の評価で比較した．○は可能である，もしくは言及は無いが，本質的に可能である場合につけられる．△は論文中のに言及は無いが，十分では無いと判断した場合につけられる．×は本質的に不可能な場合につけられる．

本節の以降は各項目に対してのそれぞれの評価の理由と HDOS に対しての比較の考察を述べる．

### 2.2.1 イベント粒度に関する考察

イベント粒度に関して，HDOS は UDT と呼ばれる独自のトラップハードウェアを用い，命令の動作等のトラップ発生条件を設定することで他の動的最適化システムに比べ細粒度のトラップ発生を実現させている．UDT の機能の詳細は次章で述べる．それに対し，ADORE，Spike は Itanium に備わる PMU を利用して最適化ルーチンを駆動するためのイベントを得ている．Itanium の PMU はハードウェアで規定されたトリガ条件を満たした場合にイベントを発生させる．[39] Itanium の提供するトリガは，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，UDT に比べ粗粒度のイベント発生機能が提供されている．それぞれの論文中で紹介される最適化以外への適用を考えた場合，PMU のイベントは十分であるとはいえないため評価では△とした．Trident は粗粒度のイベント発

生で最適化ルーチンを呼ぶが，解析部の大半をハードウェアプロファイラに委譲するため，適切な粒度でイベントを発生させることができる．しかしながら，UDT等の単なるイベント発生ハードウェアと比べ，プロファイラとして解析ハードウェアを存在させることはハードウェア量の増大を招くため，ソフトウェアオーバヘッドとのトレードオフとなる．

### 2.2.2 ハードウェア情報の利用に関する考察

次章で明らかになるUDTの持つ特性として，詳細なハードウェア情報を提供するインターフェースを持つ点が挙げられる．UDTはリオーダバッファのコミットエントリをソフトウェアから読み込むことを許すことによって，最適化ソフトウェアが詳細な命令の動作の解析を行うことができる環境を提供している．ADORE, Spikeでは，PMUからハードウェアの情報が提供される．しかしながら，その情報はItaniumのパフォーマンスカウンタの情報のみであり，それ以上の詳細な情報を知ることはできない．これは，本質的に2.2.1節のイベント粒度と関係しており，現状示される最適化アルゴリズムに十分であるが，それ以外の最適化の適用を考える場合，不十分と言える．Tridentでは，物理的な結線の問題が生じない限り，独自の解析ハードウェアはCPU上のどの箇所の情報でも得ることができるため，十分なハードウェア情報を得ることができる．

### 2.2.3 最適化結果の再利用性に関する考察

HDOSは主記憶上に展開したバイナリコードを最適化し，後の5.1.2節で示す手法を基に，主記憶上のバイナリを実行ファイルに書き戻すことを明示している．このことから，HDOSは最適化結果の再利用性を持つ．ADORE，SpikeはItaniumのトレース収集能力を基に，トレースを主記憶上に展開し，そのトレースを最適化することで動的最適化を実現する．ADOREに関して，収集したトレースを実行ファイルに書き戻す論述は見受けられないが，その手法が提供された場合，本質的に最適化結果の再利用性を持つ．IspikeはItaniumのみではなく，他のプラットフォームでの実行が想定されているシステムであり[62]，ベースとなるアイデア

が，オブジェクトコードを他の命令セットアーキテクチャに対応するようにバイナリ変換する技術から発展している．このため，本質的に最適化結果の再利用性を持つ．Trident は最適化結果を Helper Thread として動的に生成する．その Helper Thread はプログラム解析ハードウェアの提供するタイミングで実行される必要がある．このため，Helper Thread の実行は最適化システムが存在しない場合は実行できず，本比較基準に基づくところの最適化結果の再利用性を持っていない．

### 2.2.4 安全性に関する考察

HDOS はバイナリの修正をトラップハンドラが実行する．このことから，最適化ルーチンはオペレーティングシステムの機能として提供される．通常，主記憶上に展開されるプログラムは安全のため，メモリ保護上書込み禁止に設定してあり，ユーザレベルでの修正を許していないが，HDOS の場合，特権モードによる修正が可能であるため，この安全性は守られる．一方，ADORE，Spike はオペレーティングシステムのシグナルイベントから起動されるハンドラをユーザレベルで定義することから，容易に最適化ルーチンへの悪意ある修正を許す．両者の安全性に対する本質的な違いは，最適化ルーチンがトラップハンドで実装されるかシグナルハンドラで実装されるかの違いである．ユーザレベルでプログラムコード修正を許すことは安全性を損なうことは明らかであり，本比較では，ADORE，Spike の安全性を×とした．Trident では，Helper Thread は解析ハードウェアと密接に関係し，Helper Thread の実行は Main Thread の実行の結果に影響を与えない制限があることから本比較では○とした．

### 2.2.5 最適化処理の透過性に関する考察

HDOS はハードウェアとオペレーティングシステムによる動的最適化手法であり，最適化のプロセスにシステムユーザ及びソフトウェア開発者の操作は一切存在しない．このため，システムユーザ及びソフトウェア開発者からの視点では，プログラム実行に際して，HDOS の存在を認知することはできない．Adore では，動的最適化のための Thread Libray を実行時のスタートアップでリンクする手法を

採用しており，透過性を重要視したシステムを提唱している．Ispike は開発ツールの一部として提供される．このことから，システムとしての透過性は低く，また，透過性の重要視もされていない．Trident では，Helper Thread の動的生成は完全にシステムユーザ及びプログラマから隠蔽されている．

## 2.3 バイナリ変換技術とその他の動的最適化の関連研究

1.1 節では，現時点で実用に至った実行バイナリ最適化を示した．本章では 1.1 節で示していない研究分野における実行バイナリ最適化を関連研究として紹介する．本章最初の節では，実行バイナリ最適化の動機付けとなったバイナリ変換技術の変遷と関連研究を紹介し，動的最適化研究の紹介へと結ぶ．本章 2 節では本論文で示した HDOS と関連する動的最適化を中心にソフトウェアベースとハードウェア協調 2 種類に分けて関連研究を紹介する．特にハードウェア協調の動的最適化研究に関して，HDOS との違いを明確化しながら紹介する．

### 2.3.1 バイナリ変換

西暦 2000 年当時，Intel の次世代 64 ビットプロセッサと目された Itanium プロセッサ [39][1]が発表され，ISA(Instruction Set Architecure) がそれまでの IA-32 と異なることから，既存のソフトウェア資源が使えなくなる懸念があった．また，Itanium に限らず，x86-64 や PowerPC 等，他のパーソナルコンピュータ向け CPU も同様に 64 ビット化が進められる過渡期であった．バイナリ変換技術[2]はこの頃から注目を集めることとなる [40][41][42][43][44]．例えば，Itanium は VLIW[45] アーキテクチャを採用し，命令の依存に関するスケジューリングをソフトウェア (コンパイラ) が行う．その他にも，プレディケーション，投機的命令実行，ソフトウェアパイプライニングなどコンパイラが高速化のために行わなければならない独自

[1]Intel 社と Hewlett-Packard 社との共同で開発されたマイクロプロセッサ．2002 年には，それまで現行であった Itanium に変わり，マイクロアーキテクチャを変更した Itanium2 を発表．2008 年 2 月 25 日にパッケージ表記を Itanium2 から Itanium に変更したことにより，現在は Itanium2 を Itanium と呼称するようになる．命令セット名は IA-64(EPIC アーキテクチャ)．

[2]バイナリ変換 (X to X binary translators) はバイナリリライティングシステム (Binary Rewriting Systems) 又は ISA マイグレーション (ISA-Migration) とも呼ばれる．

の最適化が多数有る．バイナリ変換においても，この最適化は必須であり，ソースコードの提供されない状態でどのようにしてこの最適化を行うかが，研究のメインターゲットとなる．バイナリ変換における最適化に活動的であったグループの 1 つがアリゾナ大学の SOLAR(Software Optimization at Link-time And Runtime)[46] である．彼らは Itanium の静的バイナリ最適化に関していくつかの論文のを残している [47][48].

一方，バイナリ変換を動的に行うシステムも多数提案されている．西暦 2000 年当時では，Transmeta 社[3]の Crousoe プロセッサ [49] が代表的な例と言える．Crousoe の内部の命令実行パイプラインは VLIW で構成されるが，IA-32 命令を実行可能で，IA-32 命令を内部 VLIW 命令に変換する作業をソフトウェアが行う．この変換ソフトウェアはコードモーフィングソフトウェア (CMS) と呼ばれ CPU 内のフラッシュメモリに格納される．CMS は IA-32 を VLIW に変換するときに独自の最適化を施している [50]．また，Itanium に関して，2003 年以降の Itanium プロセッサ上で動作する OS[4]は IA-32 Execution Layer(IA-32 EL) と呼ばれる機能を有し，Itanium プロセッサのプラットフォーム上での IA-32 アプリケーションの実行を可能としている [51]．IA-32 EL は Cold Code Translation と Hot Code Translation の 2 段階に分けてバイナリ変換を行い検出した Hot Code に対して強力な最適化を施す．IA-32 EL は IA-32 命令を Itanium 命令に動的変換するソフトウェアであり，その発想は Crusoe と酷似している．両者の大きな違いは Crusoe が OS を含む全実行を動的変換することに対し，IA-32 EL は動的変換の対象をアプリケーションに限定している点にある．この様な動的バイナリ変換システムは IA-32 EL の他にも，FX!32[52][53]，Solaris-SPARC バイナリを x86 ／ Itanium 上で動作させる QuickTransit[5]，Hewlett-Packard 社の PA-RISC トランスレータ [54] が挙げられる．

### 2.3.2 ソフトウェアベース動的最適化

2.3.1 節で示したバイナリ変換の過程で行う最適化の関連研究とは異なり，純粋に実行時情報を用いてバイナリの最適化を行うことのみが目的の分野がある．動

[3]Novafora 社に買収された後 2009 年 8 月に完全に操業を停止
[4]Linux 及び Windows Server 2003 SP1 以降
[5]製作元である Transitive 社が IBM 社に買収されたため，2009 年 1 月 13 日に販売停止

的最適化の研究で，最も頻繁に参照される研究がDynamo[35]である．DynamoはPA-8000プロセッサ上で動作するソフトウェアインタプリタである．Dynamo上でPA-8000バイナリコードをインタプリット実行することで，最適化のための情報収集と最適化タイミングの取得を実現している．DynamoはPA-8000上でPA-8000バイナリのインタプリット実行をするため非効率な実行を強いられるが，最適化が行われた後は，最適化後のコードをネイティブ実行する．主な最適化アルゴリズムとしてオリジナルコードからのトレース生成が挙げられる．トレースを抽出することによって，トレース実行中の命令キャッシュミスを減らすことが可能となる．しかしながら，オリジナルコードをインタプリタ実行するオーバヘッドは動的最適化による実行速度向上のポテンシャルを下げる結果をもたらしている．また，ハードウェアのサポート無しに，ユーザプロセスにおけるプログラムコード解析を行うため，キャッシュメモリやバッファなどのハードウェア資源の利用状況が解析できない．そのため，Dynamoは実行時最適化を行うが，その最適化はコードスケジューリングとレジスタ割り当てに特化している．

Dynamoの発表後，RIO[6] Project[55]が立ち上げられ，様々な動的最適化フレームワークが提供された．その1つにDynamoRIO[57]がある．DynamoRIOはIA-32のWindows／Linux用に開発されたフレームワークであり，Dynamoと実装方法は異なるものの，Dynamoと同様に実行中バイナリをソフトウェアのみで解析し，最適化を行うことができる．現在，DynamoRIOはオープンソースソフトウェアとして配布[56]され，様々な研究に活用されている[58][59].

[6]Runtime Introspection and Optimization from MIT's Laboratory for Computer Science

# 第 3章

# User Definable Trap

本章では，本論文を通し，共通のインフラストラクチャとなるハードウェア，User Definable Trap(UDT) の詳細について述べる．

本章を始める前に，本論文で用いる言葉の定義を行う．

**1** トラップ

トラップと言う言葉は現存する各アーキテクチャにより扱いが異なる．本論文では，“割り込み”，“例外”，“ソフトウェアトラップ”を含む全てのプログラム実行シーケンスを強制的に遷移させる機能の総称に位置づける．

**2** アーキテクチャ

本論文では単にアーキテクチャと呼称する場合は命令セットアーキテクチャを意味する．

**3** マイクロアーキテクチャ

本論文では，マイクロアーキテクチャと呼称する場合は命令セットアーキテクチャで規定されない部分を含む設計・実装を意味する．

HDOS の最適化は OS のトラップハンドラが行い，トラップを発生させる特殊なハードウェアである UDT がハンドラの呼び出しと最適化のための実行時情報の提供を行うことは 1 章で既に述べた．トラップは通常，プログラマが予期しないイベントが CPU 内で発生した時に OS が対処する場合，又は，プログラマが故意

に OS を呼び出して OS の機能を利用したい場合に使われる．一方，UDT は例外的な処理のためにトラップを発生させる訳ではなく，システムのユーザ (動的最適化での利用の場合は HDOS) が設定した条件を満たした場合に，OS に制御を移させるためにトラップを発生させる．HDOS はこの UDT を利用して，ネイティブバイナリ実行に割り込んで最適化処理を行う．本論文は提案する動的最適化を論じるため，UDT を含んだシステムとして，HDOS を議論するが，UDT は HDOS 専用ハードウェアではなく，独立した多目的の汎用ハードウェアであり，動的最適化以外にも利用できる．UDT の動的最適化以外の利用方法は 5 章で議論する．

本章は，最初の節でハードウェアブロック図を用いて UDT の機能と利用方法を述べる．次の節で UDT が実存マイクロアーキテクチャ上でどのようにトラップハンドラを呼び出すかを例示して議論する．最後の節で UDT の実装のためにはどのようなソフトウェアインターフェースが必要か議論する．

## 3.1 UDT ハードウェア

バイナリを実行中に最適化するためには，プログラムの振る舞いを記録しつつ，最適化処理を開始するタイミングを見つけなければならない．そのための手段として，HDOS は UDT を用いる．本節では，UDT ハードウェアの詳細を述べ，どのようにトラップを発生させるかを述べる．

UDT はシステムユーザが指定した命令が指定の動作をしたときに指定のトラップハンドラを駆動させるシステムである．そのため，UDT はコミットする全ての命令を監視し，その命令がシステムユーザの指定する命令か，指定の動作を行ってコミットしたかを監視する．UDT ハードウェアを図 3.1 に示す．

[12] 図 3.1 は指定命令の特定動作からトラップハンドラを駆動する様子を UDT ハードウェアのブロック図上で示している．図中に示す命令は 4 章の評価で用いる Alpha[12] アーキテクチャの持つ命令である．以降，本論文中の HDOS に関するアーキテクチャ依存部の議論は全て Alpha アーキテクチャに基づく．HDOS はプログラムを実行する以前に Trap Definition Table(TDT) にトラップを起こしたい条件を記述しておく．記述は命令のオペコードとハードウェアにより定められた動作条件を示すビットパターンの組み合わせにより行う．図 3.1 では簡単のため

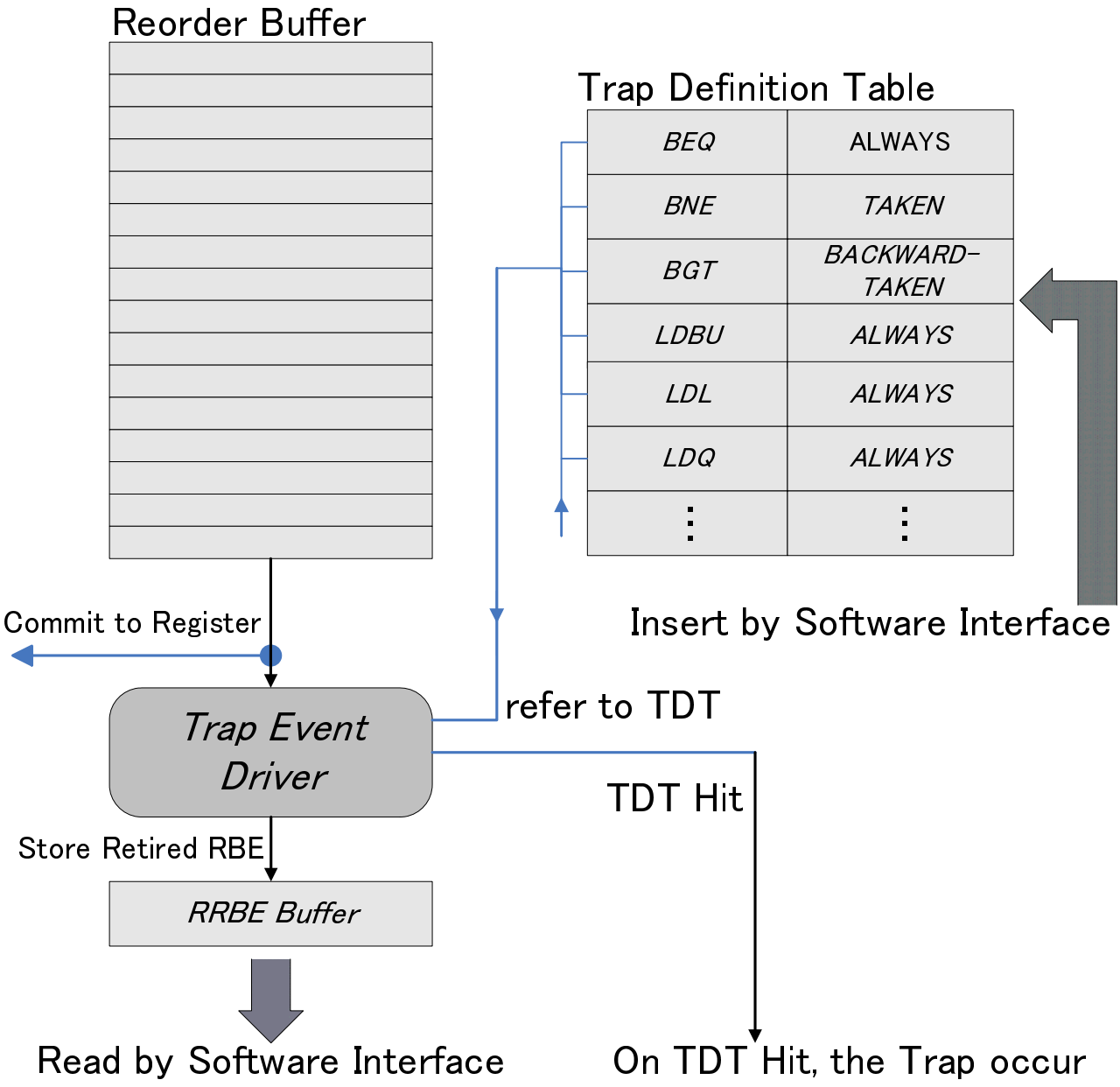


図 3.1: UDT ハードウェア.

ビットパターンによる記述はしていないが，例えば，TDT の最初のエントリには BEQ(Branch on Equal) 命令が実行された場合，無条件でトラップ発生 (ALWAYS) という記述がなされている．つまり，図中のエントリの構造は左が指定命令を示し，右がその動作を示している．2 番目のエントリは BNE(Branch on Not Equal) が taken 実行された場合，3 番目のエントリは BGT(Branch on Greater) が taken でかつ分岐先が仮想アドレス空間上下位のアドレスである場合にトラップを発生させる条件設定がなされていると見ることができる．4 番目以降のエントリでは，LDBU(Load Byte Unalined)，LDL(Load Long)，LDQ(Load Quad) 等のロード系の命令が呼び出されたとき常にトラップが発生する条件設定がなされている．図中の例では分岐命令，ロード命令の条件設定がなされているが，TDT は全ての命令に関しての設定が可能である．

一方，UDT で TDT 以外の主たるハードウェアは Trap Event Driver(TED) である．TED 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．TED は命令コミット時にリオーダバッファからリタイアするエントリを解析し，コミットされる命令がテーブルに記述されていないかを探索し，該当エントリがテーブル中に存在した場合，トラップを発生させる．この例外は通常のプロセッサが持つ例外と異なり，あらかじめシステムユーザにより登録されているハンドラが実行される．UDT トラップ発生以降の UDT トラップのハンドリングの議論は次の節で行う．また，ハードウェアは例外を発生させたと同時に，リタイアしたリオーダバッファエントリをソフトウェアから読み出し可能なレジスタに格納する．このレジスタを Retired Reorder Buffer Entry(RRBE) と呼ぶ．ハンドラはこのレジスタを参照することにより，必要な情報を得ることが可能となる．UDT が HDOS のシステムとして利用される場合，最適化のための情報はこの RRBE から得ることができる．リオーダバッファのエントリはマイクロアーキテクチャ依存で，通常は CPU 設計者のみが知る情報であるが，HDOS の最適化ルーチンの作成者はエントリの構造を知っておかねばならない．少なくとも，コミット時のリオーダバッファのエントリには完了した命令の種類と演算の結果が格納されているため，プログラムの振る舞いを解析する情報としては最適である．例えば，リオーダバッファは正確な例外復帰に対応するため，全ての実行中命令の PC を保持している．また，分岐命令の場合は，分

岐予測の成立／不成立を命令コミット時にチェックするため，必ず飛び先アドレスの計算結果を保持している．

表 3.1 に命令を識別するビットパターンと表 3.2 に分岐命令における動作条件を示すビットパターンを示す．

表 3.1: 命令識別ビットパターン

| ビットパターン [ opcode ] | 命令種 |
|---|---|
| 00 [ 000000 ] | Control |
| 01 [ 000000 ] | Load/Store |
| 10 [ 000000 ] | Integer Arithmetic & Logical |
| 11 [ 000000 ] | Miscellaneous |

表 3.2: 分岐命令における動作条件ビットパターン

| ビットパターン | 動作条件 |
|---|---|
| 0000 0001 | not-taken |
| 0000 0010 | taken |
| 0000 0100 | forward |
| 0000 1000 | backward |
| 0001 0000 | unconditional （Branch Type） |
| 0010 0000 | indirect （Branch Type） |
| 0100 0000 | return （Branch Type） |
| 1000 0000 | PC specified |

命令を識別するビットパターンは 8 ビット分用意されている．上位 2 ビットが表に対応する命令の種類を意味し，下位 6 ビットが命令を識別する命令セットアーキテクチャ上のオペコードを意味する．この表は Alpha アーキテクチャに基づいているため opcode フィールドは 6 ビットである．また，命令個々にエントリを作成することが可能だが，opcode フィールドを 0 にセットすることによって指定カテゴリを代表する集約エントリを作成できる．例えば，“01”＋“000000” のビット

パターンは全てのロード命令を対象にトラップを発生させることができる．これによりエントリ使用数を削減することが可能となる．この他にも，エントリを無効化する無効化ビットフィールドが存在する．

動作条件を示すビット幅は 8 ビット分用意されている．表 3.2 は個々のビット位置が分岐命令の動作条件と分岐命令のタイプを示している．分岐タイプビットフィールドは分岐に関する命令の動作条件を 1 エントリに集約した際に使用される．ソフトウェアはハンドラを起動したい条件に該当するビットフィールドの位置に 1 をセットする．複数ビット立てる場合は，優先度が背反する条件でない限り OR 条件となる．例えば，PC specified は優先条件でそれ以外は非優先条件である．このため，PC specified が指定されたときは PC specified が設定されたときはそれ以外のビットは無視される．

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 エントリを使用して実現される．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，トラップマスク機能とは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．例えば,「全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，関数戻り以外の全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．このため，TDT にはマスクビットフィールドが用意される．トラップマスクとして使うエントリにはマスクビットフィールドにマスクしたい条件のエントリのインデックスをセットする．また，トラップマスクとして使わないエントリはマスクビットを自分のエントリのインデックスに設定することで，ハードウェアにトラップマスクか否かを識別させる．このことから，マスクビットのビット幅は用意する TDT エントリ数によって決まる．

TDT には PC 指定機能が備わる．そのため，TDT には指定 PC を格納するためのフィールドが存在する．例えば，命令識別パターンを “00 000000”，動作パターンを “1000 0000” とした時，PC フィールドで指定された分岐命令が実行されるときのみトラップを発生させることができる．また，2 エントリ使用して，トラップ

マスクと組み合わせ,「全ての分岐命令でトラップを発生させるが，例外的に指定PCではトラップさせない」といった設定が可能になる．

これらハードウェアの導入により，トラップハンドラは必要な時にだけ呼び出されるように，細かくトラップを制御することが可能となる．例外を発生させるとパイプラインを通過中の命令を全てキャンセルしなければならないため，無用な例外発生を避けた方がHDOS導入による実行時性能低下を抑えられる．

## 3.2 UDTトラップハンドリング

UDTは特定のアーキテクチャ及びその実装に依存するハードウェアではない．しかしながら，UDTはソフトウェアからトラップ発生条件を制御するハードウェアであることから，実装対象アーキテクチャのトラップハンドリングに深く関係する．このことから，本節では実存するアーキテクチャのトラップハンドリングの例を挙げ，UDTで発生するトラップをハンドリングするOSの実装を議論する．

HDOSは最適化ルーチンがOSのトラップハンドラとして実装されることから，全ての最適化処理の起点はUDTトラップが発生するタイミングとなる．通常，トラップはアーキテクチャ設計者及びマイクロアーキテクチャ設計者 (これらをまとめ，以降設計者と総称する) がシステムの利用に必要な要素をあらかじめ考慮し，用意する．このため，トラップが発生する要因は設計者が想定したもののみとなる．UDTは通常のアーキテクチャに用意されないトラップ機構なので，実装対象のアーキテクチャに親和する形で実装しなければならない．

Alphaアーキテクチャで実装する場合，例えば，System Control Block(SCB) のオフセット0x700-0x7F0の16のSCBエントリが未使用エントリであることから，この1つを活用できる．この未使用エントリの最初のエントリ (オフセット0x700) をUDT用に使う前提で，exception service routineとして実装したHDOSの最適化ルーチンのエントリポイントと起動時パラメータをこのSCBエントリに記述できる[1]．更にUDTから発生したトラップは当該SCBエントリを参照するようにハードウェアを設計する必要がある．

[1] 1 SCBエントリには8バイトのexception service routineの先頭仮想アドレスとexception service routine起動時に渡すことができるパラメータ値を設定できる

また，既存アーキテクチャにUDTを実装する注意点として，UDTトラップ発生条件を対象アーキテクチャのプロセッサステータスレジスタに関連させる必要がある．例えば，HDOSにUDTが使われる場合，最適化ルーチン内部のコードがUDTトラップを発生させると，無限のトラップネストに陥る可能性がある．この問題を避けるために，UDTハードウェアによるトラップ発生はOS部のコードの実行中に発生しないように制御しなければならない．Alphaアーキテクチャでこの制御を行うためには，UDTトラップを発生させる前にステータスレジスタのPS<mode>フィールドが0になっている(カーネルモードではない)ことを確認する必要が有り，PS<mode>が1で有る場合，UDTトラップは無視されるように設計されるべきである．

## 3.3 UDTソフトウェアインタフェース

図3.1で示すUDTにはTDTエントリやRRBEレジスタ等，ソフトウェアインターフェスから読み書きするハードウェアが存在する．本節ではAlphaアーキテクチャへの実装を前提にUDT制御に必要な命令追加について議論する．

RRBEとTDTエントリへのアクセスのためにAlphaアーキテクチャにMFRRBE(Move From RRBE)とMTTDT(Move To TDT)を追加する．MFRRBE命令はRRBEレジスタから汎用レジスタにデータを転送する．MTTDT命令は汎用レジスタからTDTエントリにデータを転送する．MFRRBEを使うことで，トラップハンドラはトラップの原因となった命令の振る舞いを解析することができ，MTTDTを使うことで，TDTエントリに3.1節で示したビットパターンをトラップ発生条件としてセットすることができる．MFRRBEのフォーマットとMTTDTのフォーマットを図3.2に示す．

Alphaアーキテクチャとして両命令はMiscellaneous Instructions内に定義可能で，特権命令として実装される．Miscellaneous Instructions(オペコード上はMISC)はFunction Codeに十分な余裕があり，両命令を追加しても問題はない．両命令のフォーマット上の違いはFunction Codeの違いと0〜7ビット目までの8ビットの利用方法である．RRBEレジスタは64ビットの汎用レジスタより大きなサイズであるため，RRBE-Indexは8バイトアラインされたRRBEレジスタの何番目のデータを取得するかを指定する．図3.1に示す様にTDTは複数エントリ存在する．

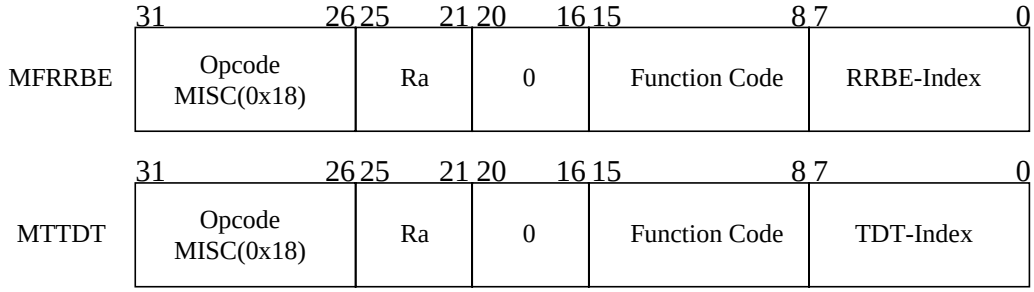


図 3.2: Alpha アーキテクチャにおける MFRRBE と MTTDT の命令フォーマット.

TDT-Index は何番目のエントリが書き込み対象エントリかを指定する．`MFRRBE` 命令の Ra フィールドはデータ格納先の汎用レジスタ番号，`MTTDT` 命令の Ra フィールドは TDT エントリにセットするデータのある汎用レジスタ番号が指定される．`MTTDT` は TDT の 1 エントリの情報量が汎用レジスタのビット数を超える場合複数命令用意される．例えば，PC 指定のトラップマスク機能を利用する場合，明らかに汎用レジスタのビット数を超えることから，この場合は `MTTDT1` と `MTTDT2` の 2 命令が用意される．

# 第 4章

# データプリフェッチ最適化

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．この問題はメモリウォール問題 [14] 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一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出来る．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．本章では，ネイティブバイナリに対する動的最適化手法の HDOS 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．

データプリフェッチはハードウェアによる実装，ソフトウェアによる実装を問わず，多岐にわたる手段で実現されている．本章では最初の節で，データプリフェッチの背景と HDOS により実現するデータプリフェッチの利点を述べる．次節では，HDOS によるデータプリフェッチ最適化の実現方法を述べる．3 節で，HDOS の生成したバイナリの性能評価を他のハードウェアで実現した手法と比べ，評価する．4 節で HDOS が最適化対象バイナリの最初の実行でデータプリフェッチ最適化のために支払うソフトウェアオーバヘッドについて言及し，評価する．5 節では，3 節で性能比較対象としたハードウェア実装のデータプリフェッチ手法のハードウェア量と HDOS のハードウェア量，即ち UDT のハードウェア量を示し，提案手法の有効性を議論する．最後に，6 節では，UDT 実装時のハードウェア規模の推定

と，CPU 実装時の設計への影響を考察する．

## 4.1 データプリフェッチ

メモリウォール問題 [14] はコンピュータシステム設計者が克服すべき課題であると広く認識されてから，今日まで，30 年以上の間，解決しない問題である [22]．データプリフェッチは将来使われるアドレスを予測し，あらかじめロードすることで，この問題に対処する 1 つのアプローチである．キャッシュメモリの持つ空間的局所性はデータプリフェッチの 1 つであると言える．本節では，最初に，今まで研究されてきた代表的なデータプリフェッチ手法を紹介し，その利点及び欠点を述べる．次に，今までのデータプリフェッチ手法と比べて，本論文で提案する HDOS をデータプリフェッチ最適化に適用する利点を述べる．

### 4.1.1 代表的なデータプリフェッチの背景と問題点

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等に格納される．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一つとして実現される．[18, 15, 16, 17]．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等から解析される．コンパイラがプリフェッチアドレスが予測できた場合，コンパイル対象のアーキテクチャに備わるプリフェッチ命令を使用して最適化を行う．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もつプログラムコー 23 ドに対し，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．また，1.1.3 節で示した通り，最近のソフトウェア開発・利用形態では DLL ／ DCL ／ JIT が利用される場合が多く，これら技術を利用する場合には実行時に実行コードが確定する特性から，制御構造及びデータ構造等が確定するため，ソフトウェアアプローチによる解析スコープは限定される．

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レスを予測する点にある [19, 20, 21]．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．

### 4.1.2 HDOS によるデータプリフェッチ最適化の利点

本章では，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．動的最適化システムを用いることで，4.1.1 節で示したコンパイラ最適化の持つ問題を回避できる．また，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．このことから，本章4.3節で示す性能評価の比較対象はハードウェア実装のデータプリフェッチとした．一般的にソフトウェアプリフェッチはハードウェアプリフェッチよりプリフェッチの精度で劣ることから，性能評価の比較対象としてハードウェアプリフェッチのみで十分である．

本提案の目的はハードウェアプリフェッチと同程度の性能をHDOSで実現し，その性能を実現するためのハードウェア量を低く抑えることにある．これはUDT実現のためのハードウェア量が極めて小さいことから実現できるものである．但し，本提案は履歴ベースのデータプリフェッチを小規模ハードウェアで実現することを可能にする反面，トレードオフが存在する．ハードウェアプリフェッチはプログラム実行のバックグラウンドで専用ハードウェアが常にアドレス履歴解析を行うのに対し，HDOSはソフトウェアでこの解析を行うことから，最適化のオーバヘッドを含むプログラム実行の全体のスループットはハードウェアプリフェッチに比べ劣る．しかしながら，1.3節で述べた，HDOSの最初の利点の“最適化結果の再利用性”から，この最適化オーバヘッドは2回目以降の実行からは無視できる．つまり，本提案の本質は,「同じプログラムを複数回実行する場合において，ハードウェアプリフェッチの専用ハードウェアは同じ計算を行うであろうことから，この計算をソフトウェアに委譲することにより，1度目の最適化オーバヘッドの付与をトレードオフとしてハードウェア量を節約できる」点にある．これは，贅沢にハードウェア資源を投入できるハイパフォーマンスコンピューティング向けのシステムには利便性を損なう提案となるが，例えば，組み込みシステムなどの小規模かつ，コスト性を重視するシステムには受け入れられるトレードオフとなる．

加えて，本章では，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解決するために，既存のAlphaアーキテクチャのプリフェッチ命令を使わず，新しいプリフェッチ命令を導入する．この命令はAlphaアーキテクチャの持つ通常のロード／ストア機能と，プリフェッチ機能の2つの機能を1命令で実現するもので，プリフェッチアドレスをロード／ストア機能で使うアドレスから計算できる．この命令の導入で，プリフェッチアドレス計算のための命令数増加を回避できる．さらに，HDOSの実行中バイナリに対する

修正を最小限に抑えることが可能となる．この命令の詳細は次節で詳しく述べる．

## 4.2 HDOSによるデータプリフェッチ最適化の実現手法

HDOS をデータプリフェッチに適用するために，本論文は命令セットの変更とトラップハンドラとして実装する最適化のアルゴリズムを提案する．本節は最初に，例として Alpha アーキテクチャ上でのデータプリフェッチ最適化のための命令セット変更を述べ，次に最適化アルゴリズムを述べる．

### 4.2.1 命令セット変更

本論文では，HDOS のデータプリフェッチ最適化のために，アーキテクチャに新しいプリフェッチ命令を導入する．導入する新しいプリフェッチ命令はロード／ストア機能と，プリフェッチ機能の 2 つの機能を 1 命令で実現する．これら命令はストライドロード/ストア命令 (SLSI)，間接参照プリフェッチ命令 (ILI)，ストライド間接参照プリフェッチ命令 (SILI) に分類される．プリフェッチアドレス参照をストライド参照，間接参照，ストライド間接参照の 3 分類に分ける手法は本手法においての提案ではなく，プリフェッチを行うシステムでは一般的な手法でり，この分類でアドレス予測をすることで高い効果が得られることは既に報告されている [29][30]．本手法による提案はこれら 3 分類のプリフェッチアドレス計算をロード命令機能と合わせて 1 命令にパッキングすることのみに有る．これは 4.1.2 節で述べたプリフェッチアドレス計算のためのソフトウェアオーバヘッド削除と HDOS によるバイナリ修正の簡単化に貢献する．

しかしながら，特定のプリフェッチアドレス計算手法をハードウェア実装し，既存命令にバインドする行為は，RISC 設計ポリシーに反して命令単体の高機能化を促すと共に，命令セットのプリフェッチアドレス計算の自由度を奪うという考え方がある．これに対し，本手法の命令設計ポリシーを肯定する要因の一つとして，プログラムの持つアドレス参照傾向の PC 局所性が挙げられる．K.Nesbit らは SPEC CPU 2000 ベンチマークにおいて，グローバルに供給されるアドレス列でプリフェッチアドレスを予測するより，そのアドレス列を PC で系統立てて整理

し，PC 毎にプリフェッチアドレスを予測する方が性能の高いプリフェッチを行えると報告している [23]．これは，これらアプリケーションにおいて，1 つのデータのアドレス計算するための命令群が 1 つのロード／ストア命令に対になって存在していることを示しており，多くのロード命令は制御構造により参照傾向を変えないことを示している．

また，新しい命令を追加したことにより，プリフェッチアドレス計算用の加算器など，従来のプリフェッチ命令を使う最適化では不必要であったハードウェアが追加される．しかしながら，これらハードウェアは近年のスーパスカラ・アウトオブオーダプロセッサの要するハードウェア規模から比較すると，極めて小規模であり，データパスに関しても，クリティカルパスに影響しない部分で追加されるため，遅延及びダイ面積に対する影響は小さいと考える．

新しい命令を定義するために，その命令のハードウェア上での振る舞いを定義しなければならない．以降は，SLSI，ILI，SILI，のハードウェア上の機能の詳細を述べる．

### ストライドロード/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 *Stride Load or Store Instructions (SLSI)* を加えた．本章で性能評価を行う際に，Alpha アーキテクチャ[12] 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 4.1 に示す．

SLSI は既存のロード／ストア機能とストライド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 [13] に対し，末尾にストライドアクセスを示す_SEQ を付与したニモニックで表現される．説明の符有／無，整有／無の表記は既存命令の持つ符号拡張機能，アドレスアラインの必要性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 4.1 で示される命令数だけでなく，9 命令存在する．また，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 (7 命令)，さらに，浮動小数ロード／

表 4.1: ALPHA 命令セットに追加した整数ロードに関する SLSI の命令

| 追加命令 | 拡張元命令 | 説明 |
|---|---|---|
| LDBU_SEQ | LDBU | 1B ストライドロード/符無 |
| LDWU_SEQ | LDWU | 2B ストライドロード/符無 |
| LDL_SEQ | LDL | 4B ストライドロード/符有 |
| LDQ_SEQ | LDQ | 8B ストライドロード/符有 |
| LDQU_SEQ | LDQU | 8B ストライドロード/整無 |

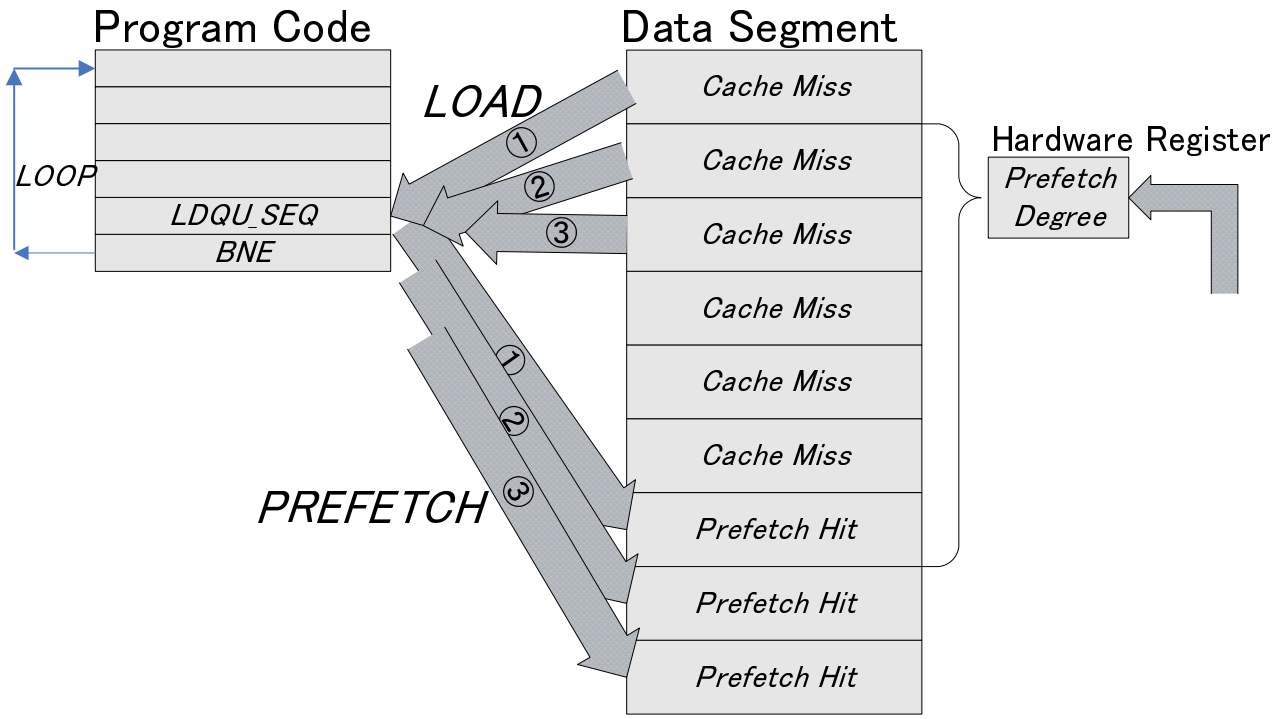


図 4.1: SLSI の動作例.

ストア命令に対してもストライドプリフェッチ命令 (8 命令) が追加される．この様に，Alpha で提供される全てのデータ転送命令に対応するストライドプリフェッチ機能を備えた命令 (計 24 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．全ての 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 4.1 に SLSI の実行時の振る舞いを示す．図中の LDQU_SEQ は SLSI 命令の 1 つである (アライン無し 8B ストライドロード)．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 (図中の *Prefetch Degree* 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 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 *Prefetch Degree* はキャッシュブロック数で表現される．図の例では，LDQU_SEQ 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 *Prefetch Degree* 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 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 は常に *Prefetch Degree* 分先のブロックを先見してプリフェッチする．この *Prefetch Degree* の例では， LDQU_SEQ 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．

図 4.2 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 4.2 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及びストア値のフォワードのため，Load/Store Queue を備えている．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全く同じ動作をする．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 4.1 及び図 4.2，に示す stride length 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．これは当該命令を実装する対象のアーキテクチャの空き命令フィールドの都合で決まる．図 4.2 では，便宜的に固定レジスタから取得する場合の図を示している．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

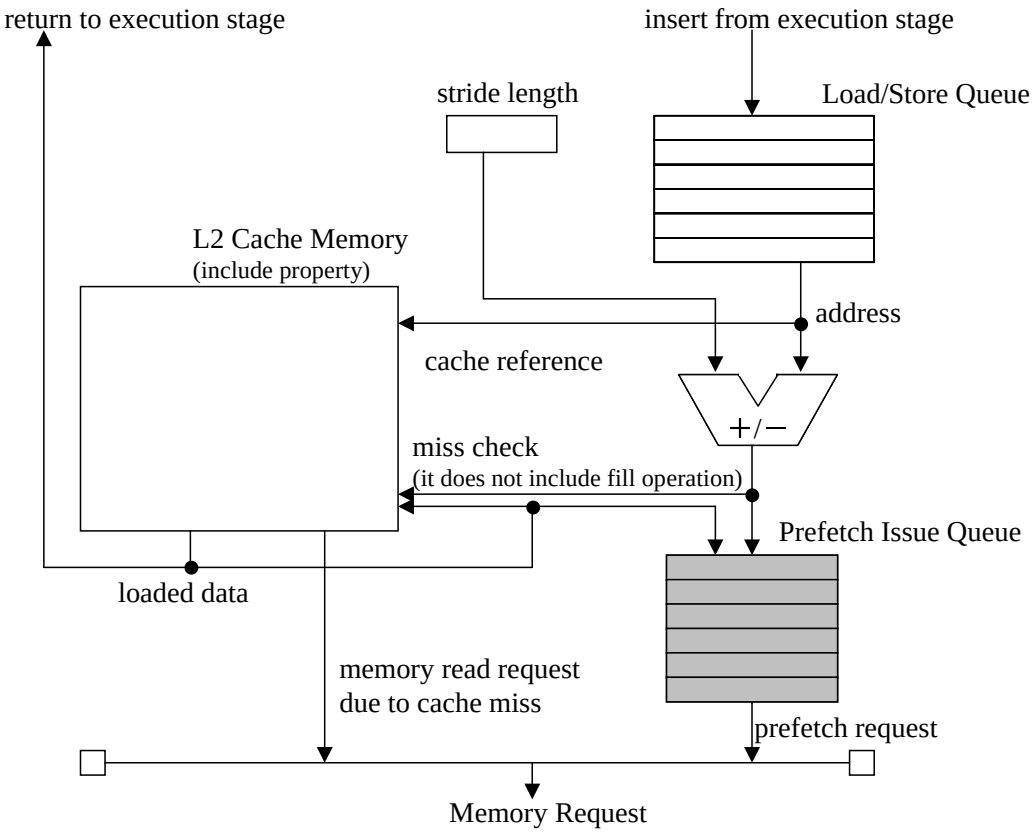


図 4.2: SLSI/ILI/SILI を実現するメモリシステムモデル.

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 がオーバー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既に 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．また，Prefetch Issue Queue から主記憶にプリフェッチリクエストが発行される際，TLB 参照を行うが，SLSI によるプリフェッチ発行では，TLB ミス時に TLB ミスハンドリングを行わず，例えアクセス保護違反を起こした場合も同様に無視され，プリフェッチリクエストはキャンセルされる．

プリフェッチが発行され，主記憶からデータが到着した場合，キャッシュに格納，

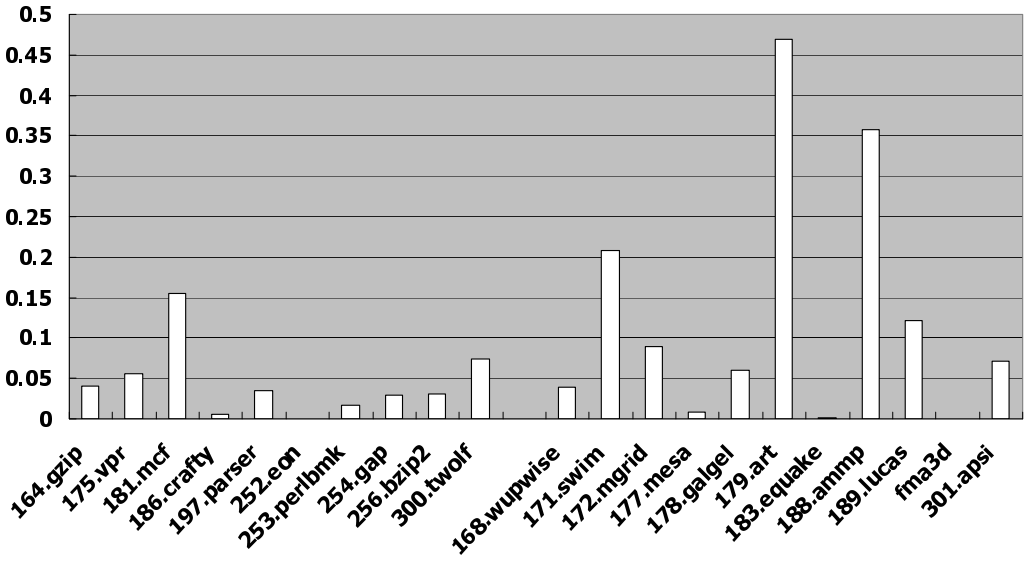


図 4.3: L2 キャッシュと L1 データキャッシュの参照比.

プリフェッチ専用バッファに格納等，様々な取り扱いが可能である．以降，本章ではプリフェッチデータは図中の L2 キャッシュ(記憶階層における最下層のキャッシュ) へのみ格納することを前提にして，議論する．

本章のメモリシステムモデルでは，通常のロード／ストアに関するキャッシュ参照に加えて，プリフェッチ発行と到着データ格納のためのキャッシュ参照が存在する．L1 キャッシュへのプリフェッチブロックロードを前提とした場合，L1 キャッシュへの参照が頻発し，通常のロード／ストアの L1 キャッシュ参照を阻害する恐れがある．図 4.3 に本章評価で使用する SEPC CPU 2000 ベンチマークのアプリケーションにおける L1 キャッシュ参照回数に対する L2 データキャッシュ参照回数の比を示す．L1 データキャッシュの構成はキャッシュ容量 32KB，ブロックサイズ 32 バイト，4 ウェイアソシアティブである．L2 キャッシュは L1 データキャッシュと同様の構成の L1 命令キャッシュのミス時にもアクセスされる．

全てのアプリケーション参照比は 50%未満であり，多数のアプリケーションで 10%未満の参照比を示している．全 21 アプリケーションの平均で参照比は 8.89%であった．このデータは，L1 データキャッシュのアクセスポートに比べ，L2 キャッシュのアクセスポートの混雑具合は平均で 1／10 以下であることを示している．本

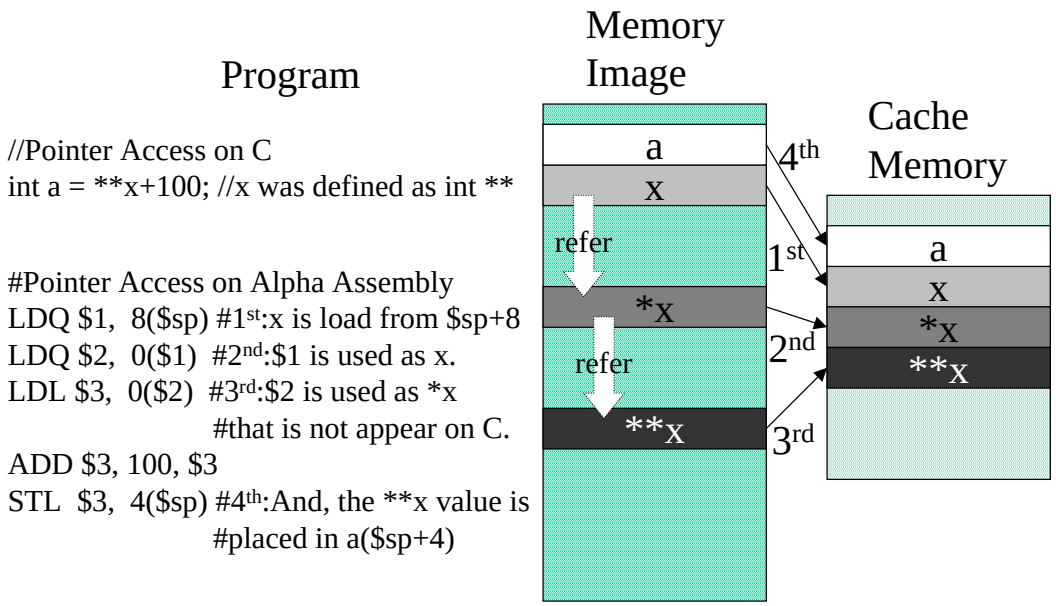


図 4.4: 間接参照時のキャッシュ状態.

章のメモリシステムモデルはプリフェッチ機能を加えることで発生したキャッシュ参照を L2 キャッシュに限定することによって，通常のロード／ストアのキャッシュ参照の性能低下を防いでいる．加えて，L1 キャッシュへのプリフェッチブロックのロードは性能向上に大きな貢献を齎す一方で，プリフェッチブロックによるキャッシュスラッシングの原因となる．本章では，L1 キャッシュへのプリフェッチブロックのロードを行わないことで，L1 キャッシュスラッシングの危険を排除し，プリフェッチによるオフチップアクセスレイテンシの軽減に着目した．

尚，本章，4 節と 5 節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．

### 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．

図 4.4 に間接参照時のキャッシュ状態の例を示す．図は 2 段階の間接参照時の C

言語と Alpha アセンブリ言語の例である．C 言語上での変数 x と変数 a はローカル変数である．図の Alpha アセンブリ言語は C 言語をコンパイルしたものである．最上部の命令から順に，スタックからポインタを取得する LDQ, そのポインタを使ってさらにポインタを取得する LDQ，そのポインタを使って 4 バイトロードを行う LDL，LDL でロードした値に 100 を加算する ADD，最後の命令 STL は加算した値をストアしている．このプログラムが実行されると x，*x，**x，a の順でメモリアクセスし，同時にキャッシュメモリに配置される．

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，例えば，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 と併せて *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も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．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 4.2 の Prefetch Issue Queue に渡す．

図 4.4 で示したプログラムを HDOS で最適化した例を図 4.5 に示す．図左上プログラムは HDOS により最適化され LDQ が IND_LDQ に変更されている．このプログラムを実行し，最初に x，*x，**x，a をキャッシュメモリにロードした状態が左

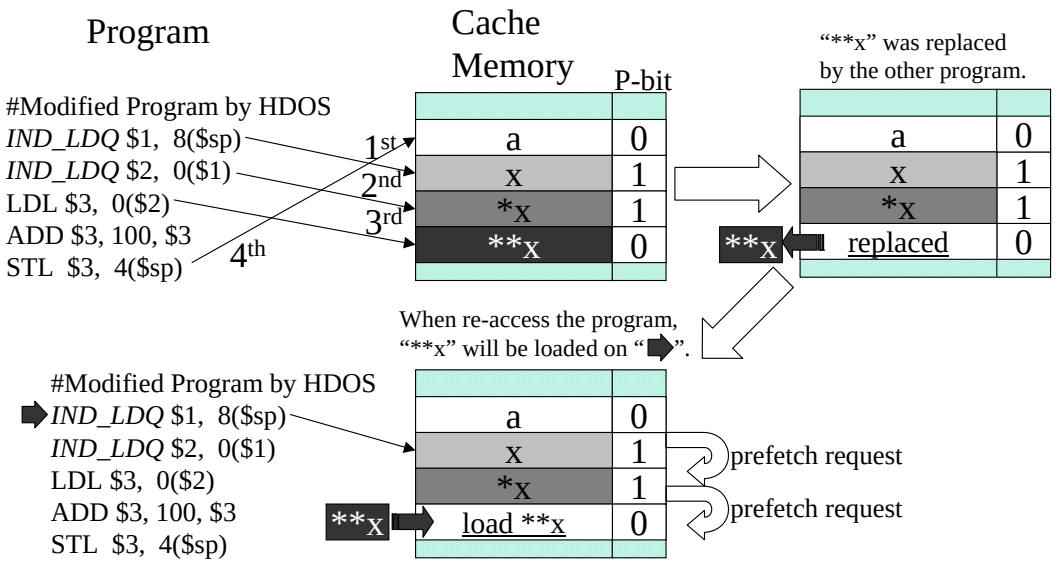


図 4.5: ILI 使用時ののキャッシュ状態の遷移.

上のキャッシュ状態図となる．それぞれのＰビットはIND_LDQでロードされた場合のみ1が，それ以外の場合で0 がセットされる．初期状態になった後，図で示されるプログラム以外が実行され，右上のキャッシュ状態になったとする．`**x`は不運にも他のデータによってリプレースされ，再度図中例示のプログラムが実行された時に，`LDL $3, 0($2)`の行の実行でキャッシュミスが発生する筈である．しかしながら，再度図中例示のプログラムが実行された場合，最初の`IND_LDQ $1, 8($sp)`の行が実行され，キャッシュヒット時にＰビットの確認がなされる．Ｐビットは1でポインタと確認されるので，再度`x`をもってプリフェッチリクエストが発行される．発行されたプリフェッチリクエストは再度キャッシュヒットするので，再度`*x`のＰビット確認後，プリフェッチリクエストが発行される．`*x`の指す先(`**x`)は既にキャッシュに存在しないので，キャッシュミスとなり，この場合のキャッシュミスはノンブロッキングロードとして主記憶に発行され，これがプリフェッチとなる．以上の説明はILIの概念的な説明である．

実際に使用されるキャッシュブロックは複数ポインタを含むことができるのでＰビットは1キャッシュブロックに複数存在できる．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．また，複数Pビットを持つ場合の特長として，ヒットしたキャッシュブロック内全てのPビットを持つデータをプリフェッチリクエストとして，発行することができる．これは，キャッシュの空間的局所性を利用した先見の間接参照プリフェッチである．

また，図4.5では，簡単のため，2度目のプログラム実行からプリフェッチリクエスト発行をしているが，実際には，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．このため，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，一度でも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無いことから，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．

もし，最適化が間違って実際のポインタでないデータにPビットが立った場合，プリフェッチリクエストがセグメンテーションフォルトの様な例外を招く恐れがあるが，実際の実装では，SLSIと同じくPビットによるプリフェッチリクエストはTLB参照でミスになった場合，このTLBミスハンドリングは行わず，アクセス保護違反に関しても同様に無視され，リクエストはキャンセルされる．

本章では，64ビットアドレスを扱う処理系における実装としてILIを8Bロードに適用する例を示したが，例えば，32ビットアドレス扱う処理系では4BロードにILIを適用する必要がある．命令セットにILIを追加する際に，実際にポインタを扱うロードのデータサイズは事前の調査が必要である．

SLSIと同様に，本章，4節と5節で示される評価のためのSimple Scalarへの修正は，このPビットを持つキャッシュを含めた図4.2で示されるメモリシステムモデルに基づいている．

### ストライド間接参照プリフェッチ命令

本節ではSLSIとILIを組み合わせた*Stride 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はストライ

ド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．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 と併せて *miscellaneous instructions* で提供することが可能である．

### 4.2.2 最適化アルゴリズム

本節の以降から，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本節4.2.1 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．全ての最適化には3 段階のステップがある．

**1** ループ検出

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，本最適化は全ての処理の最初にループ検出を行う．

**2** メモリアクセス履歴テーブルの作成

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．

**3** メモリアクセス履歴テーブルの検査とプログラムコード修正

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，SLSI，ILI，SILI のそれぞれの最適化によってステップ3 の処理が異なる．以降はそれぞれのステップの詳細な説明を述べる．

#### 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．また，トラップマスク機能を用いて関数

呼び出し，関数戻りに関係する分岐ではトラップが発生しないようにする．このTDT 設定で，UDT は関数呼び出し，関数戻り以外の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．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．

ステップ 1 における後方分岐リスト作成アルゴリズムを示す．

**アルゴリズム 1** 後方分岐リスト作成アルゴリズム

**1** 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．

**2** この時，既存リストに無ければ，新しくリストに加える．

**3** 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．

**4** 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．見つけられなかった場合はトラップハンドラを終了し，次の起動を待つ．

### ステップ 2: メモリアクセス履歴テーブルの作成

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．本章では，この作業を観測フェーズと呼ぶ．図 4.6 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

| 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 4.6: メモリ参照履歴テーブルの例.

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 値を用い検索し，一致を調べる．一致エントリが見つけられなかった場合，初期値 (*PC*=命令 PC，*Stride*=0，*Stride Length*=0，*Last Address*=0，*Execute*=1) をセットしたエントリを作成する．一致エントリを見つけた場合，以下のアルゴリズムを実行し，履歴テーブル内の該当エントリを更新する．

**アルゴリズム 2** メモリアクセス履歴テーブル更新アルゴリズム

**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．得られた *SL* はエントリ内にある古い *SL* と比較される．

**2** 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)，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 (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

化無し) 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アルゴリズムの 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，つまり，観測フェーズが終了するきっかけとなるトラップ処理で，ステップ 3 の処理に遷移して履歴テーブルを順に検査し，最適化候補となるロード/ストア命令を特定する．

#### ステップ 3: メモリアクセス履歴テーブルの検査とプログラムコード修正

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 4.6 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 4.6 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 4.6 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．

**1** SLSI

ストライド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，最適化ルーチンはストライドアクセスを発生させた命令を SLSI で置換する．

**2** ILI

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辿り，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．

**3** SILI

ストライド間接参照プリフェッチ最適化処理は，ストライドプリフェッチ最適化の解析処理中に 8 バイトストライド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．

## 4.3 性能評価

本節では，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．

### 4.3.1 比較対象手法

HDOS 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．本節では，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．[23] 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，命令毎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，本節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，本節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．[24] 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時に，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．

本節の以降の性能評価は PC/CS，PC/DC，SMS を比較対象に行う．

表 4.2: 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 |
| memory access latency | [first]:120 [inter]:12 cycles |
| memory access bus width | 8B |

### 4.3.2 シミュレーション環境

性能評価のシミュレーションは Alpha アーキテクチャ[12]用の Simple Scalar3.0[25]を使い行った．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．

Simple Scalar のシミュレーションパラメータを表 4.2 に示す．

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．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．本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，極めて少ないため，この差分は極めて小さい．本節では，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．

評価は SPEC2000[26] 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 [27] 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．

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及び 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 8KB，PHT は 16K 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．

### 4.3.3 バイナリ性能評価

図 4.7 はプリフェッチ無し実行の IPC 性能を 1 とした時の IPC 性能の比を示している．図 4.7 の濃さの違う棒グラフは左から SMS，PC/CS，PC/DC，HDOS，

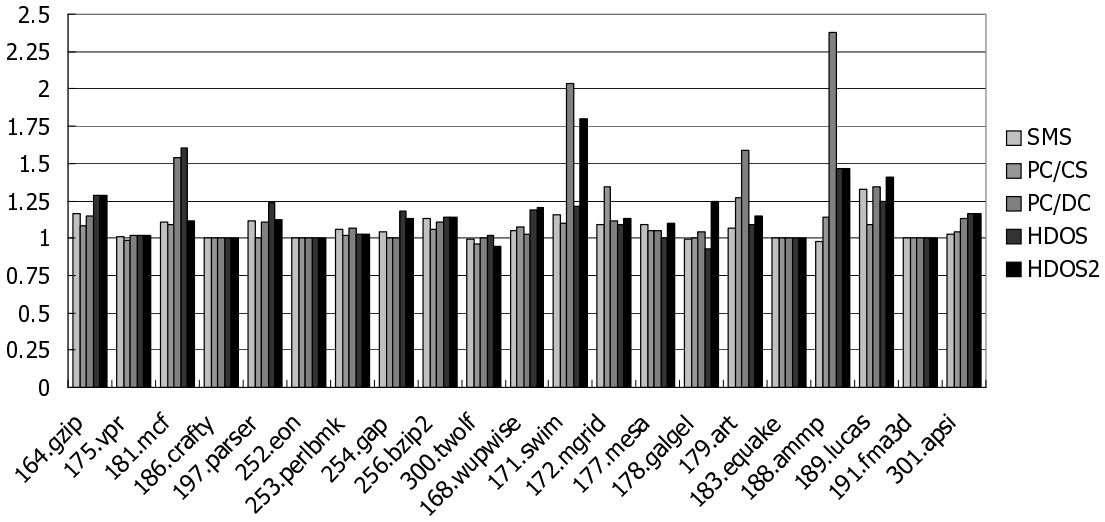


図 4.7: SPEC2000 における IPC 性能の向上比.

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 命令セットの制限等が無く，検出した全ての置換候補命令が置換できた場合の結果である．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．(Alpha 実装で符号付き 16 ビット即値を符号付 8 ビット即値で表現できない命令) 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無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言う形では 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．

図 4.7 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．PC/DC は提案したプリフェッチ命令では実現されていないプリフェッチアドレス算出アルゴリズムを持ち，171.swim，179.art，188.ammp ではその参照傾向が顕著に現われたと考えられる．HDOS，HDOS2 は PC/DC 程の大きな改善が認めら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．

HDOS／HDOS2が既存ハードウェアプリフェッチより性能が向上する理由は，扱うアドレス履歴の容量の差により生じていると考えられる．ハードウェアプリフェッチはアドレス履歴をSRAMのテーブル内に保持する．SRAMは高価なハードウェアであり，実装に際しては，履歴保持のために提供できるハードウェア領域は限定される．比較対象のハードウェアプリフェッチの履歴テーブルは現実的な容量であることから，プリフェッチアドレス予測のためのアドレス履歴はこれに制限されている．これに対し，HDOSはソフトウェアがアドレス履歴の収集を行う．安価なDRAMにアドレス履歴は蓄積されるため，ハードウェアプリフェッチの扱うことのできるアドレス履歴の容量の限界を大きく超えて扱うことができる．この予測のための履歴テーブルの大きさの違いが，ハードウェアプリフェッチに性能が勝る理由であると考察する．

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言える．

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．とくに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．これは本研究固有の問題ではなく，全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．これに関しては更に研究が必要となる．

## 4.4 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，ソフトウェアが最適化を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他に，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．本節では，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 4.1.2 節で述べたバイナリへの直接修正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，本節で示すオーバヘッド無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．本節では，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[25] 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り評価する．また，UDT によるトラップ発生傾向を解析し，実行途中で UDT 機能を停止させオーバヘッドを減少させる手法を議論する．

### 4.4.1 ソフトウェアオーバヘッド見積もり環境

ソフトウェアオーバヘッド見積もりのためのシミュレーションは PISA 命令セット [28] 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．Simple Scalar のシミュレーションパラメータ，HDOS 最適化パラメータ及び計測条件は 4.3.2 節と同じである．評価は SPEC2000[26] ベンチマークの中から 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 [27] よりプリコンパイルバイナリを取得し，使用した．

本節の評価では，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，4.3.2 節では，性能評価のために，Alpha[12] 命令セットを使用した．これは，PISA の様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事に重点を置いたためである．

本節と 4.3.3 節の評価を関連付けるために，両節で使用した 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．この理由の一つとして，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挙げられる．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 のもつ最小の機能単位で命令を構成するポリシーによるものである．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，本節評価環境と 4.3.2 節評価環境は全く同じではないが，酷似した環境での評価であることがわかる．

### 4.4.2 ソフトウェアオーバヘッド見積もり手法

Simple Scalar 3.0 は精密なスーパースカラ・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／割り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．例えば，アプリケーションが呼び出すシステムコール等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．本節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．

$$R_c = \frac{T * C_h + C}{c} \tag{4.1}$$

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 4.1 の $C_h$ のように定数値で与えて，精度の高い見積もりを行うことが難しい．このことから，4.2.2 節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もりを行う．$T$ と $C_h$ は 4.2.2 節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 4.1 を表現すると

$$R_c = \frac{T_1 * C_{h1} + T_2 * C_{h2} + T_3 * C_{h3} + C}{c} \tag{4.2}$$

となる．$T_1$，$T_2$，$T_3$ は Simple Scalar で個々のベンチマーク毎に計算し，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．本節では，$C_{h1}$，$C_{h2}$，$C_{h3}$ をそれぞれステップ毎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．各ステップの処理の詳細は 4.2.2 節で述べた．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及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．本節では，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 4.3 に示す．

表 4.3: 見積もりに使用する各ステップ毎のハンドラ実行クロックサイクルパラメータ

| $C_{h1}$ | $C_{h2}$ | $C_{h3}$ |
|---|---|---|
| 180 | 790 | 4928.4 |

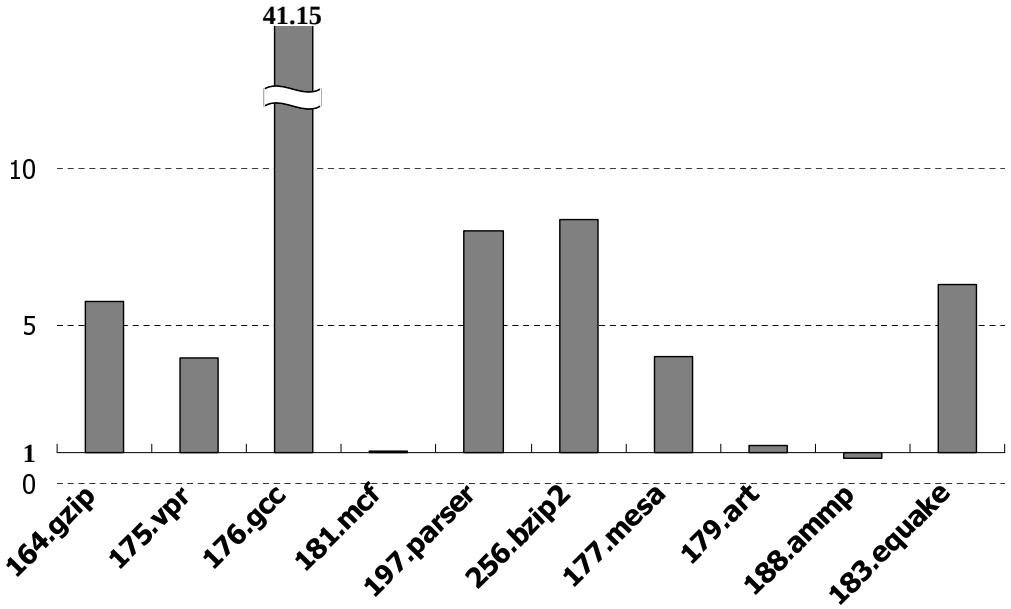


図 4.8: ソフトウェアオーバヘッドを含む実行時間の増減.

### 4.4.3 ソフトウェアオーバヘッド見積もり評価

表 4.3 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 4.8 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 は概ね 4 倍〜10 倍迄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 による割り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．今回の見積もりで 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.

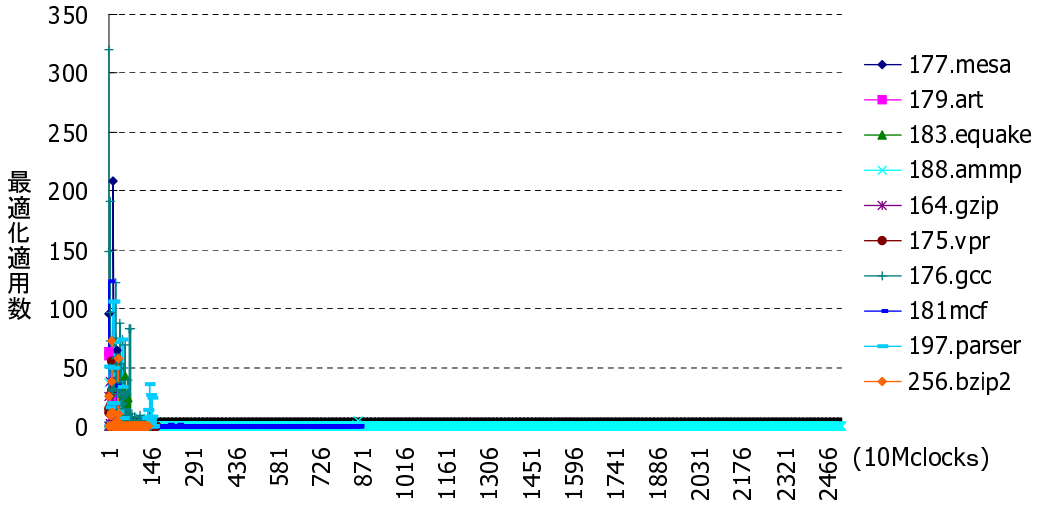


図 4.9: SPEC2000 における経過クロックサイクル数と最適化適用箇所数の関係.

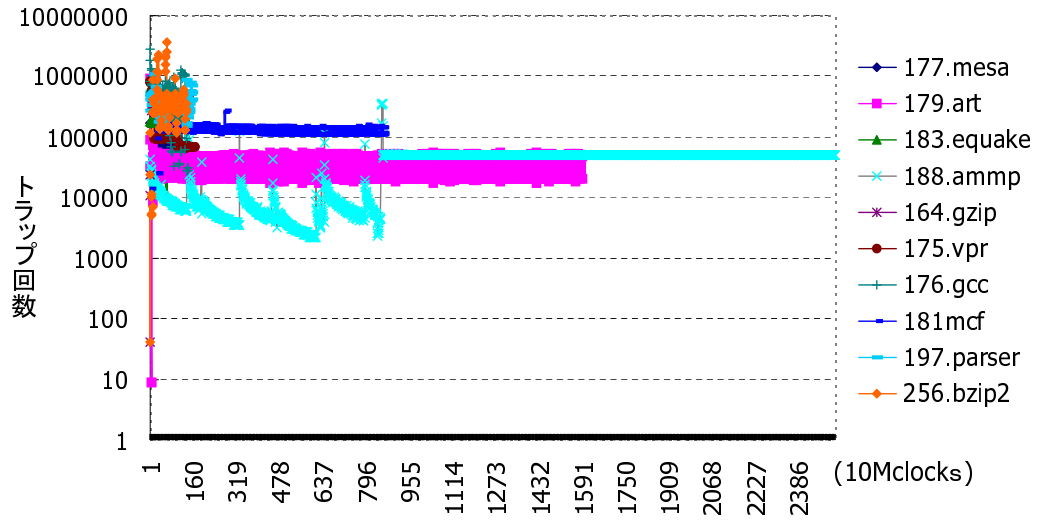


図 4.10: SPEC2000 における経過クロックサイクル数とトラップ発生数の関係.

### 4.4.4 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最適化適用数の関係を図 4.9 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 4.10 に示す.

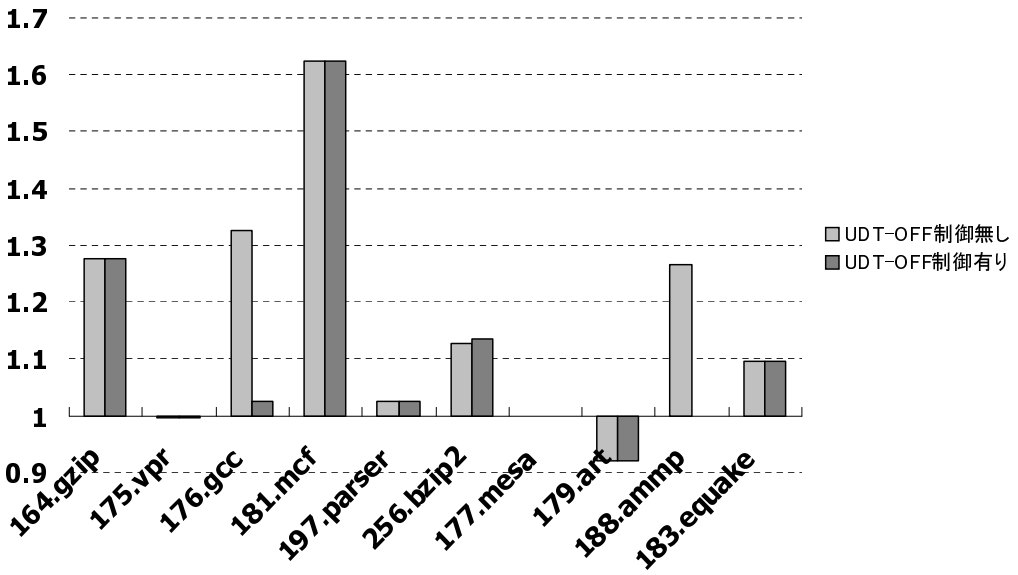


図 4.11: UDT-OFF 制御を行った場合のソフトウェアオーバヘッドを含まない IPC 性能向上.

それぞれの図は個々のデータではなく，全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 1460M クロックの時点に到達する頃にはトラップ及び最適化適用数が収束しているのがわかる．ここで考えるべきところは，図 4.10 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常に行われ，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 4.11 と図 4.12 に示す.

図 4.11 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わかる．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関係する最適化候補の命令が検出される予定であったが，それを取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

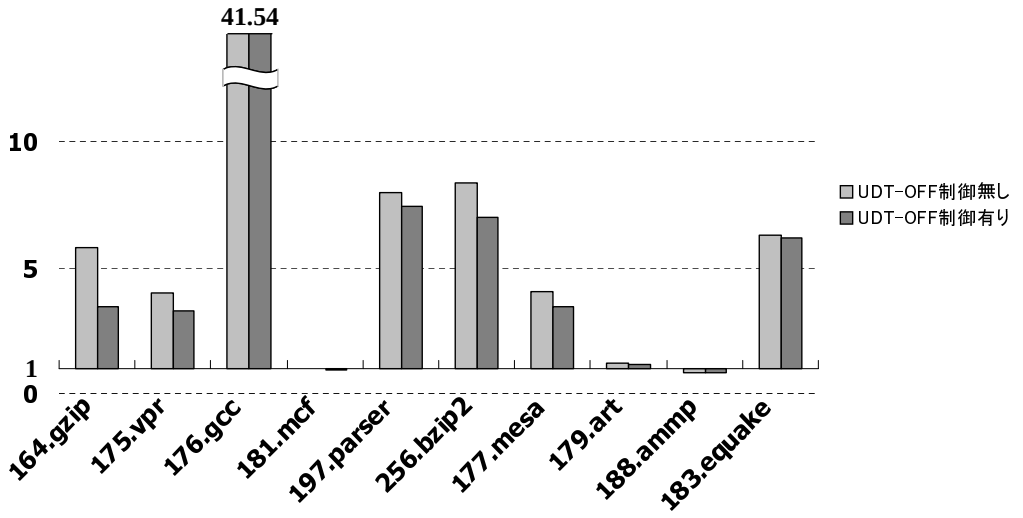


図 4.12: UDT-OFF 制御を行った場合のソフトウェアオーバヘッドを含む実行時間の増減.

御を加えない場合と比べ，大きな差は見られない．これは，176.gcc と 188.ammp とは違い，性能向上に関係する最適化候補を取りこぼしていないことを意味している．但し，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 4.3.3 節で述べた．

図 4.12 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上，図 4.11 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取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，プログラム実行の後半で，最適化適用数は少ないが，性能に大きく影響する最適化候補が存在したことを意味している．解析の結果，176.gcc のオーバヘッドが他のアプリケーションに比べ著しく大きいのはオーバヘッドの主たる原因がステップ 3 の実行によるものであるためである．このステップ 3 のテーブルウォークが処理の前半で多発するも，その成果の最適化は性能を向上させることができず，

表 4.4: 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もり

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

UDT-OFF 制御を行った後半の処理で，本来性能を上げるはずだった最適化候補を取りこぼす例であったといえる．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．

図 4.12 で得られた，UDT-OFF 制御は最大で 4 割のオーバヘッド削減の結果となったが，これは全体実行を 20 億命令とした時の割合でしかない．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全て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，これは精細なアウトオブオーダシミュレーションに計算時間を要するための制限であり，殆どの SPEC2000 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，1 回目の実行であっても最適化による IPC 向上がオーバヘッドを上回る可能性がある．

## 4.5 ハードウェア量の見積もり

本提案の第一の目的は，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 4.4 に示す．

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 エントリ当たり 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 エントリ当たり 173 ビットとなる．以上より，評価した 16K エントリ PHT の SMS の実装のための必要メモリ量は 347.74K 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 共に提案者が論文中にて 32bit-Tag でそれぞれ 4K バイトと見積もりをしている．

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もりを行う．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出来ないサイズである．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 590bit 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．TDT の 1 エントリに必要な情報は，3.1 節で示した TDT の詳細設計から，命令識別ビットパターンが 8bit，動作条件ビットパターンが 8bit，エントリの valid ビットが 1bit，トラップマスク指定ビットが 2bit，プログラムカウンタ指定のトラップマスクのために 32bit 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

4.4 から明らかな様に，HDOS に必要なメモリ量が他の手法に比べ，極めて小さいことが分かる．この差は，ハードウェアプリフェッチ手法の持つ，履歴蓄積用の SRAM テーブルが HDOS には存在しないため生じている．HDOS はソフトウェアで解析を行うことから，メモリアクセス履歴テーブルは主記憶 (DRAM) 内に存在する．HDOS がメモリアクセス解析のために使用した主記憶領域は 128KB であり，仮にこれがオンチップの SRAM で実装されていたならば，L1 キャッシュの総容量と等しくなり，CPU のダイエリアを圧迫する結果となる．このことから，本章の履歴ベースプリフェッチのためのハードウェア削減手法はメモリアクセス履歴テーブルの所在を変えたことが，最大の特長であると言える．

## 4.6 UDT 実装のハードウェア規模の推定と CPU 設計への影響の考察

本節では，UDT 実装の際の実現可能性を議論するため，ハードウェア規模の推定と CPU 設計への影響を考察する．本論文で紹介される全ての最適化は，最低限，4 エントリの TDT をもつ UDT で実現可能である．しかしながら，デバッカサポートやソフトウェアオーバヘッド削減のための高度なトラップマスクを実現する場合，より多くの TDT エントリを必要とする．現代のマイクロアーキテクチャにおいて，どの程度の TDT エントリのを持つ UDT の実装を許すかを考察する．

### 4.6.1 TLB を指標としたハードウェア規模の推定

UDT の TDT は各エントリの値比較を行うハードウェアであることから CAM(Content Addressable Memory) で構成される．UDT と同様に，CPU 上で CAM を必要とするハードウェアに TLB が挙げられる．本節では，UDT の実装上の面積と遅延の指標を TLB の実装を例に上げ比較する．CAM の場合，全てのエントリ比較を行うことから，メモリ自身に必要な面積と遅延に加え，比較器に要する面積と遅延が加わる．TLB と TDT，それぞれの比較器ビット数を考えると，32 ビットアドレス空間の場合，TLB の 1 エントリに 4KB ページで 20 ビットの比較器，TDT の 1 エントリに 50 ビットの比較器を要する．TLB には，タグと同サイズ以上のデー

タメモリがあり，かつ，そこからデータを取り出すロジックが存在するのに対し，TDT はヒット／ミス判定を返すロジックのみを持つ．まず，面積で両者を比較する場合，32 エントリのフルアソシアティブ TLB と 16 エントリの TDT の比較で，TDT のサイズが TLB を上回ることは無い．現代の高集積化の進んだ CPU では，このような TLB はダイ写真で判別が難しいほど，小規模であることは近代のマイクロアーキテクチャ設計で明らかになっている．このことから，16 エントリ規模の TDT を実装した場合，面積的な影響はほとんど見られないと考察できる．

### 4.6.2 TLB を指標とした回路遅延の推定

一方，遅延に関して，TLB は命令のフェッチ毎に参照されるメモリであり，TDT は命令のコミット毎に参照されるメモリである．参照の頻度に関して，TLB と TDT は大差が無いことが考察できる．このことから，同規模の CAM で遅延が大きく違わなければ，遅延設計に関する難しさも同等であることが考察できる．厳密には，RRBE から TDT を検索するためのビットエンコーディングやトラップ発生シグナルを伝達するためのランダムロジックが存在するため，遅延が同規模であるとは言い難いが，メモリに比べランダムロジックの遅延は小さいことが一般的であることから，決定的に遅延設計の難しさを増大させる要因にはならないと考える．このことから，本節では，TLB と同等の規模の TDT を想定した場合，実装に対する影響は少なくなると結論づける．

### 4.6.3 UDT ソフトウェアインターフェス実装のための面積増加に対する考察

UDT のソフトウェアインターフェス実装に関して，データパスの増加が，回路面積の増大を増やす懸念がある．これに対する考察として以下の事項があげられる．現代のプロセッサアーキテクチャでは，ステータスレジスタ等，多数の特殊レジスタとそれに対するアクセス命令が存在している．これら特殊レジスタへのアクセスは 1 対 1 で接続されるわけではく，各種レジスタのレイアウトを工夫し，データパスを共有することで，データパスの面積に対する影響を最小にしている．

UDT の RRBE や TDT へのアクセスも同様に，レイアウトの工夫によりデータパスの共有が可能である．つまり，UDT のソフトウェアインターフェス実装による面積増加は他の特殊レジスタを含めた UDT のレイアウトの工夫により軽減できる．

# 第 5章

# HDOS実装に要するオペレーティングシステム機能及びHDOSの発展的な利用方法

本論文 1 章，3 章及び 4 章において提案した HDOS の背景，目的，ハードウェアにおける提案，最適化適用例としてのデータプリフェッチ最適化の実現方法と評価を述べた．これまでの章で一連の HDOS の利用方法及びその特性について十分に議論した．本章では，HDOS の発展的な利用方法に言及するため，HDOS に関する追加情報を示す．本章最初の節で，HDOS を実装する際に要求されるオペレーティングシステムの機能であるプログラム起動時の HDOS の初期化と，??節で予告した最適化済みバイナリコードを実行ファイルに書き戻す処理の詳細を述べる．次節では，ハードウェア UDT の HDOS 以外への適用例としてハードウェアデバッガサポートへの適用を示す．その次節では，4 章で述べた HDOS のデータプリフェッチ最適化以外の適用を議論するため，最適化適用例の 1 つ目として，ソフトウェアトレース最適化を提案する．本章最終節では，最適化適用例の 2 つ目である，HDOS のキャッシュメモリの電力最適化への適用を提案する．

## 5.1 HDOSの初期化の実装と最適化結果の再利用手法

本節では，プログラム起動時のHDOSの初期化の方法と??節で予告した最適化済みバイナリコードを実行ファイルに書き戻す処理を議論する．

### 5.1.1 HDOSの初期化

まず，現行UNIXシステム上でのHDOSの初期化の実装例を示す．??節で述べたように，HDOSはシステムユーザ及びアプリケーション開発者にとって透過性の高いシステムであることから，UNIXシステムにおいても操作レベルでその存在が完全に隠蔽されていることが望ましい．

HDOSはUDTとトラップハンドラの組み合わせで実装されることから，UDTの初期化などの実行前準備は全てOSが行う．[1]例えば，UNIXシステムコールを拡張してHDOSを実装する場合，exec()システムコールを拡張する必要がある．プログラム起動前のUDT初期化とトラップハンドラのエントリポイント設定を行うための拡張はexec()システムコールのパラメータを増やすことで可能である．UNIXシステムにおけるプログラムの起動は以下のようなコードで実行される．

```
int status;
pid_t pid = fork();
if ( pid < 0 )
{
    exit(1);
}
if ( pid == 0 )
{
    execl("/bin/hoge.out","hoge.out",NULL);
    exit(-1);
}else
{
```

[1]UDTのTDTは特権命令で書き込む必要があり，トラップハンドラのエントリポイントは特権レベルで読み書き可能なメモリ領域にあるため，この修正はOSにのみ許される

```
    waitpid( pid, &status, 0 );
}
```

上記例ではプログラムのロード及び起動は execl() システムコールが行う．全てのプログラム実行で HDOS が必要なわけではない．例えば，2 回目以降のプログラム実行の場合は HDOS を伴って起動する必要が無い．また，後の 5 章で示すデバッガとしての UDT の使用や，異なる HDOS の最適化適用の場合はそれを指定する必要がある．HDOS の初期化指定を含む execlhdos() システムコールを OS が持つと仮定する場合，以下の実装が考えられる．

```
int status;
pid_t pid = fork();
if ( pid < 0 )
{
    exit(1);
}
if ( pid == 0 )
{
    if ( optimize == PREFETCH )
         execlhdos("/bin/hoge.out", HDOS_PREF "hoge.out", NULL);
    else if ( optimize == SOFT_TRACE )
         execlhdos("/bin/hoge.out", HDOS_SOFT_TRACE "hoge.out", NULL);
    else if ( optimize == UDT_DEBUG )
         execlhdos("/bin/hoge.out", UDT_DEBUG "hoge.out", NULL);
    else
         execl("/bin/hoge.out", "hoge.out", NULL);
    exit(-1);
}else
{
    waitpid( pid, &status, 0 );
}
```

execlhdos() の第 2 パラメータが HDOS の種類の指定とした場合，起動時の状況に合わせて，そのスイッチを操作することで，execlhdos() 内でそれに合わせて UDT の初期化とトラップハンドラのエントリポイントの設定を行うことができる．通常，これらのプログラム起動処理はシェル内で行われ，システムユーザは関知しない．また，1 度目の起動か 2 度目の起動かをシェル内で把握し，例示プログラム内の optimize 変数を制御することで完全にシステムユーザから HDOS を透過することができる．

### 5.1.2 最適化結果の再利用手法

以降，本節では 4.1.2 節で述べたデータプリフェッチ最適化における HDOS の最大の利点である「同じプログラムを複数回実行する場合において，ハードウェアプリフェッチの専用ハードウェアは同じ計算を行うであろうことから，この計算をソフトウェアに委譲することにより，1 度目の最適化オーバヘッドの付与をトレードオフとしてハードウェア量を節約できる」特性を実現するため，最適化を行った 1 回目の実行終了後に行う HDOS の最適化結果の保存方法を議論する．

5.1.1 節では HDOS の初期化処理がシステムコール拡張で可能であることを示した．これと同様に，最適化結果の再利用もシステムコールの拡張で実装可能である．

最適化結果の再利用処理を行うコードは，UNIX システムにおいては exit() システムコールの内部コードが妥当である．通常，プロセスの終了はシグナルによる強制終了を除き，exit() システムコールを経て終了することになる．従って，この実装では exit() システムコールで通常終了したときのみ実行される．最適化結果の再利用処理は 5.1.1 節における exec() システムコールの拡張実装がなされる場合，対象プロセスが HDOS を伴う実行であるか否かを OS は知ることができるため，exit() システムコールのインターフェースの修正は必要なく，内部の実装の修正が必要である．通常の exit() システムコールはプロセス終了の後処理として以下の処理を行う．

**1** プロセスタイマの停止

**2** IPC セマフォの解放

**3** 仮想空間の解放

**4** ファイルのクローズと管理領域の解放

**5** カレントディレクトリ，umask 情報の解放

**6** シグナル破棄と管理領域の解放

**7** プロセス状態を TASK_ZOMBIE に変更

**8** 親プロセスへ通知

**9** CPU の放棄

`exit()` システムコールに対する修正とは，この 9 つの処理に加えて，HDOS が最適化のために修正したプログラムテクストの書き戻し処理を記述することである．本節では以降，この書き戻し処理をプログラムアップロードと呼称する．少なくとも，プログラムアップロードは，対象プロセスの親プロセスに上記 8 番目の処理を経てプロセス終了を通知される前に行われなければならない．[2]また，OS 内の仮想空間提供の実装により異なるが，3 番目の処理以降，対象プロセスの仮想空間へのアクセスが不可能になる可能性があるため，3 番目より前の段階でプログラムアップロードを行うべきである．

図 5.1 は UNIX システムにおける ELF32 実行バイナリフォーマットのプログラムアップロードの例を示す．

通常，OS のプログラムローダ (`exec()` システムコール) は実行バイナリファイルの中にあるプログラムヘッダテーブル情報を利用してバイナリコードを主記憶に配置する．プログラムヘッダテーブルの各エントリは以下の情報を持つ．

```
typedef struct {
  uint32_t  p_type;
  Elf32_Off p_offset;
  Elf32_Addr p_vaddr;
  Elf32_Addr p_paddr;
  uint32_t  p_filesz;
```

[2]親プロセスは SIGCHLD シグナルを受け取ると，ZOMBE 状態になった子プロセスを探し出しタスク構造体を解放する．

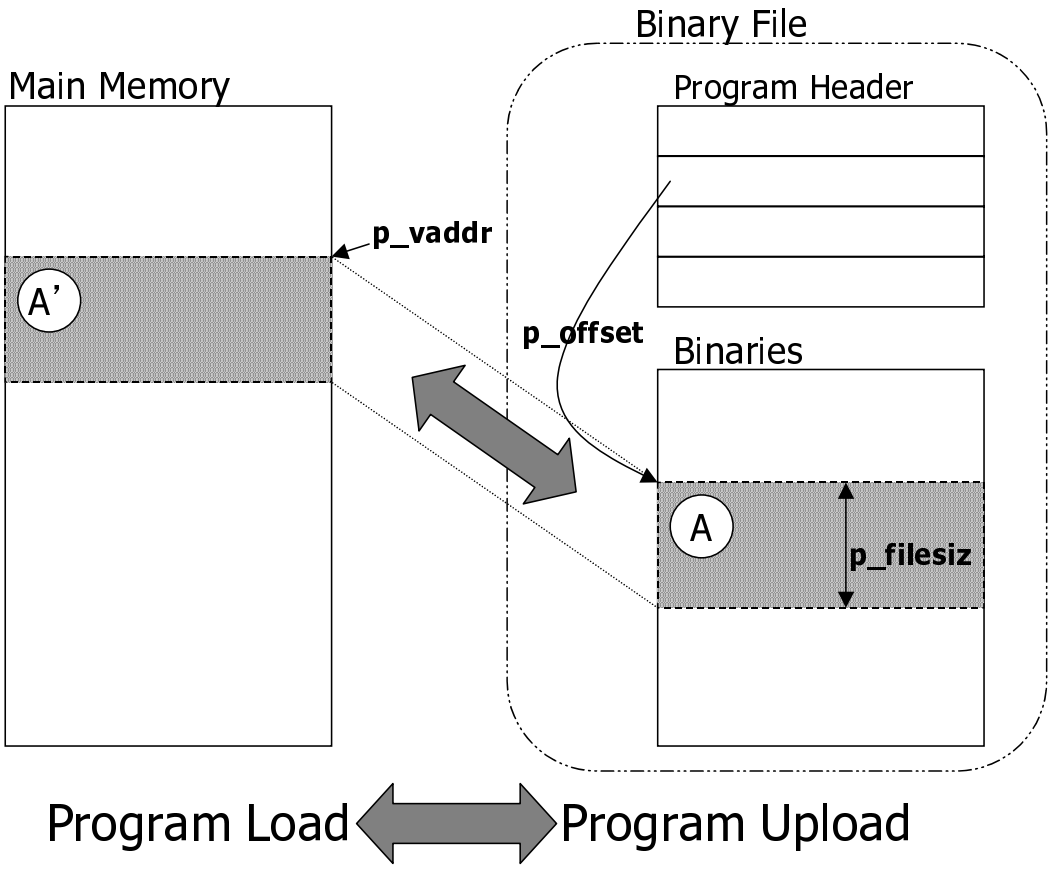


図 5.1: ELF32 実行バイナリフォーマット上でのプログラムアップローダの例.

```
  uint32_t  p_memsz;
  uint32_t  p_flags;
  uint32_t  p_alin;
} Elf32_Phdr;
```

配置の際, ELF32 ヘッダエントリの重要な情報は p_offset, p_vaddr, p_filesz である. p_offset は実行バイナリの先頭からのオフセットを示す. p_vaddr はプログラムバイナリがロードされるべき仮想アドレスを示す. p_filesz は主記憶上にコピーされるプログラムバイナリのサイズを示す. つまり, プログラムローダはバイナリファイルの先頭から p_offset 先の位置のプログラムバイナリを p_filesz 分 p_vaddr からはじまる主記憶にコピーする. 図 5.1 では, プログラム実行前に OS がファイル領域 A を主記憶領域 A' にこのプログラムヘッダ情報を用いてセッ

トしている．

プログラム終了時 (exit() システムコール実行時) プログラムアップローダはプログラムヘッダ情報を参照し，実行バイナリ内の全ての読み込み／実行許可領域のバイナリコードを主記憶内に存在する最適化済みのバイナリコードで置き換える．プログラムローダは p_type を参照することで，読み込み／実行可能セグメントか否かを判別できる．[3]図 5.1 では，プログラムアップローダは主記憶領域 A' をファイル領域 A に上書きしている．

## 5.2 UDT によるプログラムデバッガサポート

HDOS は UDT と UDT から呼び出されるトラップハンドラが協調して最適化を行うシステムである．4 章まで，UDT は HDOS の 1 つのコンポーネントとして使われてきたが，3 章の冒頭の導入部で UDT は HDOS 専用ハードウェアではなく，独立した多目的の汎用ハードウェアであり，動的最適化以外にも利用できることを述べた．また，4.5 節では，UDT のハードウェア量の見積もりに言及しており，比較的小さいハードウェアであることを示している．

この UDT の HDOS 以外への適用として，本論文では UDT によるプログラムデバッガへのサポートを提案する．本節ではこの UDT によるプログラムデバッガサポートの実現方法について議論する．

まず，本節の最初にプログラムデバッガの必要性について現在主流となったソフトウェア開発工程の検知から議論する．次に，ソフトウェア開発現場で利用されるソフトウェアデバッガの欠点を述べ，最後にハードウェアによるデバッガサポートの利点と UDT の適用可能性を議論する．

### 5.2.1 ソフトウェアテストにおけるデバッガの必要性

ソフトウェア開発は概要設計，詳細設計，実装，単体テスト，結合テスト，機能テスト，運用テスト (システムテスト) を経てプログラムを生産する．一般的には

[3]通常，プログラムテキストは読み取り／実行可能にセットされている．プログラムローダはこの情報を用いて対象プロセスのページテーブルの保護ビットを操作する．

概要設計と詳細設計が “設計” 呼ばれる作業で，実装，単体テスト，結合テスト，機能テスト，運用テストが “製造” と呼ばれる作業である．設計作業，特に概要設計はフロー設計，データ構造設計，テスト設計，通信プロトコル設計などソフトウェア開発プロジェクトを通してポリシーが統一が必要とされる作業が多種含まれるため，一般的には少人数で行う．一方，製造作業は設計されたものをプログラムコードとして具現化し，設計どおりの動作をするかの確認作業になるため，多人数で行う．プロジェクトを通して，設計作業と製造作業は同等か，若しくは設計作業の方が長く時間を掛けることが多いが，人的コストは製造作業に圧倒的に多く投入される．このことから，安価でかつ，より短いスパンでのソフトウェア開発がベンダに求められる昨今，設計の効率化手法より製造の効率化手法がソフトウェア開発には強く求められる．

不具合密度という言葉がある [31]．ソフトウェア開発現場により，バグ密度，バグ率など色々と呼称と計算方法は変わるが，同じ指標を指す．これは，コードレビュー又は単体テスト (ホワイトボックステスト) を終える工程まで完了したプログラムがソースコード規模に対してどれだけ不具合を含んだかという指標である．単体テストまで完了している場合，テストの密度によって品質の指標が変わることから

$$\text{試験密度} = \frac{\text{試験件数}}{\text{ソースコード規模 (ステップ数)}} \tag{5.1}$$

$$\text{不具合検出密度} = \frac{\text{検出不具合箇所数}}{\text{ソースコード規模 (ステップ数)}} \tag{5.2}$$

5.1，5.2 の 2 つを定義して品質の指標とすることが多い．例えば，試験密度 0.01，不具合検出密度を 0.0002 とし，その 1 割を品質許容範囲としたとき，10000 ステップ規模のプログラムの試験項目数は 90〜110 で，不具合数は 2 が上限となる．実際には，試験項目数をクリアしているのにも関わらず，不具合数が上回っているときに，若しくは不具合発生数が 0 であったときに，再製造又は試験項目の再検討を含む再テストを行う目的で使用される数値である．

ここで問題となるのが，ソフトウェア制作期間及びコスト面でのホワイトボックステストの効率である．C0，C1，C2[4] を行う際にテスト作業には一般的にデバッ

[4] プログラムの構造に着目したソフトウェアテスト．C0:命令網羅，C1:分岐網羅，C2:条件網羅

ガが用いられる．これはホワイトボックステストの特性上，デバッガによる確認が最も効率的であることに起因している．カバレッジは試験密度として品質基準を与えられてるため，そのカバレッジに沿うように作業する．基本的にテスト作業量はカバレッジで決まるが，全てのテスト項目が一定時間で完了するわけではない．例えば，C2 基準のテストの場合，テストデータによっては長時間の演算の上，初めて成立する複数分岐条件が有りえる．当然ながら，該当テスト項目を確認するための待ち時間が増大するほど，単体テストの効率は低下する．つまり，作業時間は試験密度とステップ規模でのみでまらず，テスト対象の制御構造に起因する実行時間も含めて決まる．また，不具合が検出され，修正を行う際にも再現の確認が必要であることから同様に待ち時間が要求される．ソフトウェアの製造時間の殆どはテスト／修正作業で消費されることから，テスト／修正作業の効率化を図ることは，リリース周期の早い低コストソフトウェア開発の要となる．

### 5.2.2 ソフトウェアデバッガの問題

一般的に，ソフトウェアデバッガを伴う実行はデバッガ無しの実行に比べ，ターゲットプログラムの実行が遅くなる傾向にある．これは，デバッガが実行を止めたり，バックトレースや値履歴の記憶処理を行うためにスタブ機能を利用することに起因する．スタブとはプログラムバイナリ中に故意に未定義命令等を埋め込み，OS にシグナルを発生させ，プログラム実行を別プロセスに強制遷移させる手法である．デバッガの他にダイナミックリンクなどで一般的に使われる手法である．ソフトウェアデバッガがターゲットプログラムの処理に割り込んで，モニタリングを行うためには，ターゲットプログラムのスタブ例外からシグナルを発生させ，OS のプロセススケジューリングを経てソフトウェアデバッガプロセスに制御を移す必要がある．このプロセススイッチのオーバヘッドがデバッガ上実行での低速化の主な原因となる．デバック実行で最も速度低下させる要因はウォッチポイント設定である．ウォッチポイントとは，変数／レジスタ値が変更された又は参照された場合などに，実行を一時中断させる機能である．単純にブレイクポイントを設定して実行を止めたい場合は，止めたい箇所に相当する PC の命令をスタブで上書きすれば，そこで必ず中断するため，問題は無い．ウォッチポイントの様

にデバッガが監視して中断する／中断しないを判断する場合，結果的に中断しない場合であってもデバッガに制御を遷移する必要がある．このウォッチポイントを設定した場合の “結果的に中断しないデバッガへの制御遷移” は頻繁なプロセススイッチを誘発し決定的な速度低下要因となる．

先で述べた単体テストの例では，このオーバヘッドがソフトウェア生産効率を下げる一因となる．例えば，C2 基準のテストの場合，“指定された変数 (条件網羅に関わる) が変更されるまで実行を続ける” というウォッチポイントの使い方する．しかしながら，デバッガは値変更が有る無しに関わらず，全てのメモリアクセスで呼び出されなければならないため[5]，デバッガ上の実行速度が決定的に低下する．GDB[32] における実装の例では，ウォッチポイント機能は全てのメモリ／レジスタアクセスを監視するためにシングルステップ実行を強いる．また，不具合修正時の再現に関しても，同様のウォッチポイントの利用がなされるため，プログラマはオーバヘッド含んだ実行を待たなければならない．

最近の組み込みシステムでは，クロスデバッガを使い組み込みソフトウェアをテスト／デバックすることが一般的に行われるようになってきた．高機能な表示器を持たないシステムではターゲットデバイス上でのデバックは難しい．このため，クロスデバッガはデバッガの制御及び表示機能をパソコン等の外部の機器 (ホスト) に委譲し，その他のデバッガ機能をターゲットデバイス上で処理させ，ネットワーク通信プロトコルを使い，ホスト－ターゲットデバイス間でのデバックを行う[6]．クロスデバッガによるテスト／デバックの場合，ソフトウェアデバッガと比べ，このウォッチポイントの仕組みは頻繁な外部通信の原因となり，許容できない処理オーバヘッドとなる．

### 5.2.3 ハードウェアデバッガサポートの利点と UDT 適用の可能性

本節は UDT をデバックアシストハードウェアとして使用した場合の可能性を議論する．現在のアーキテクチャでは，ハードウェアによるデバックアシストは珍しくない．例えば，Intel 64 及び IA-32 にはデバックサポート機能が備わる [33]．当該アーキテクチャには 8 つのデバックレジスタが備わり，4 つの停止位置 (PC)

[5] 変数とアドレスの関係はソース上からは完全に解析できないため
[6] 一般的にはリモートデバックと呼ばれる

を登録できる．デバックレジスタは常にPCを監視して条件が一致した場合，INT 1 のデバック例外を発生させる機能を有する．この機能を応用することでスタブ命令[7]をブレイク予定箇所に書き込むことなくデバッガを動かすことができる．また，ウォッチポイントも同様にサポートしており，上記の4つのブレークポイントの変わりにウォッチポイント変数を登録することが可能である．ハードウェアブレイクポイント機能はスタブの代替機能のみである一方[8]，ハードウェアウォッチポイントは強力なデバックサポート機能として利用されている．デバックレジスタにウォッチポイントとなる変数のアドレスを登録することで，飛躍的にデバッガ呼び出し回数を削減させることを可能とする．

現存アーキテクチャのデバックサポート機能は強力であるが，更にソフトウェアの生産性を高めるためには十分とは言えない．本節では，ホワイトボックステストを例にとってデバッガの必要性を述べたが，実際には，実装段階でのデバック実行による確認を望むプログラマが多いことや，近年では，Subversion[34] の様なマージ型ソース共有ツールを用いて多人数開発を行う機会が多くなったため，テストのためにソースコードを汚すことを良しとしない風潮がある[9]．GDB の stepping inside range[10]や配列などのアドレス範囲へのウォッチポイント，一定プログラム区間へのウォッチポイントのマスクなど，プログラマがデバッカへ要求する機能は多いが，全てを効率実行するためのサポートハードウェアを現存の CPU は持っていない．

本節では，現存アーキテクチャのデバッガサポート強化のためにUDTを使用することを提案する．UDTは命令実行条件により，トラップを発生させるハードウェアであることから，本質的にスタブや上記のデバッガサポートハードウェアと変わらない．UDTトラップを発生させてそれを扱うトラップハンドラを実装する特性上，トラップハンドラの実装次第で，最適化ではなく，単なるシグナル発生を実現することは可能である．また，UDTには命令の動作をトラップ条件に設

[7]IA-32 では 0xCC(INT3)

[8]但し，BIOS 等の ROM 媒体に書かれるプログラムのデバックにはスタブ命令を利用できないため必要不可欠な機能

[9]マージ型では同時にチェックアウトできる代わりに，自動マージされてしまうので，個人が仕様外で書いたデバック用記述がコミット時に同じ意味でありながら複数存在する可能性があり，ソースコード整理を複雑にし，生産性を下げる可能性がある

[10]2 つの PC で示される範囲で実行が行われるとブレイクする機能

定できる大きな特徴がある上，トラップマスク機能を利用した細かい条件設定を行うことができる利点がる．UDT を現行のデバックサポートへの機能強化として利用した場合，UDT に投入するハードウェア量は，単に HDOS 実現のためのコストではなくなり，さらに UDT 導入のための妥当性が増す．

UDT のデバッガサポート適用のために必要な実装を，メモリアクセスのウォッチポイントを例にとって説明する．3.1 節における TDT の命令動作を示すビットフィールドの説明は簡単のため，HDOS の実装のために必要なサブセットの説明に留まった．4.2.2 節の説明では，ステップ 1 では分岐命令のみを観測し，ステップ 2 ではその条件に加えて，ロード／ストア命令の条件を付け加えることを述べた．この追加条件はロード／ストア命令が実行されたとき常にトラップハンドラを起動する TDT 設定である．この時の TDT 設定は動作を示すビットパターンに関係なく，図 3.1 の ALWAYS 条件のみの設定[11]で十分あることから，ロード／ストア命令に対する動作条件設定は示していない．UDT をデバッガのウォッチポイントサポートに適用するためには ALWAYS 条件のみでは不十分であるため，ロード／ストア命令の動作を定義しなければならない．

デバッガのハードウェアサポートの大きな利点はデバッガの呼び出し回数制限機能にある．これにより，デバック実行のオーバヘッドが削減できる．これを高い水準で実現するため，ロード／ストア命令の動作の定義を細かく指定できるように設計する．3.1 節の表 4.1 と表 4.2 で，命令識別ビットパターンと動作条件ビットパターンはそれぞれ 8 ビット，つまり TDT の 1 エントリに必要なビット数は 16 ビットであることを示した．ロード／ストア命令に関する動作のみの定義はこのビット数で十分である．また PC 指定機能の PC フィールドをロードストアのアドレス条件格納フィールドとして使用する場合，TDT エントリのビット数を変化させること無しにロードストアの対応ができる．ロード／ストア命令における TDT 動作条件ビットパターンを表 5.1 に示す．

表 5.1 に示す動作は，上から，ロードが実行された，実装依存予約，ストアによってメモリ値が変化した，アクセスするアドレスがアドレスフィールド値と一致する，アクセスするアドレスがアドレスフィールド値より上位アドレスである，アクセスするアドレスがアドレスフィールド値より下位アドレスである，当該ロー

[11] 表 4.1 の命令識別ビットパターンの OP コード部を 0 セット

表 5.1: ロード／ストア命令における TDT 動作条件ビットパターン

| ビットパターン | 動作条件 |
|---|---|
| 0000 0001 | load |
| 0000 0010 | store |
| 0000 0100 | RESERVED |
| 0000 1000 | greater than AddrReg (Memory Address Check) |
| 0001 0000 | equal to AddrReg (Memory Address Check) |
| 0010 0000 | less than AddrReg (Memory Address Check) |
| 0100 0000 | cache miss (memory system event) |
| 1000 0000 | PC specified |

ド／ストアでキャッシュミスが発生した，PC 値フィールド (＝アドレスフィールド) 値と当該命令の PC が一致したという意味を持つ．このビットのセットの方法は表 4.2 と同様である．

この TDT のロード／ストア動作定義で，デバッガに現存アーキテクチャのハードウェアサポートよりさらに強力なサポートを提供することができる．例えば，新しくサポートできる機能として配列ウォッチが挙げられる．“0x0100 - 0x1000” 間の配列にウォッチポイントを置く場合，2 つの TDT エントリを使用する．エントリ 1 に動作記述 “0000 1000” とアドレス “0x0100” を設定，エントリ 2 にエントリ 1 のマスクで動作記述 “0000 1000” とアドレス “0x1000” を設定することでアドレス範囲 (配列) のウォッチポイントを設定することができる．このエントリ設定の概念図を図 5.2 に示す．

本節では，一例を示したが，UDT はその他にも動作パターンとマスクを組み合わせて多様なトラップ条件設定が可能であり，これは高機能化を求められるデバッカ機能をサポートするために十分な機能を提供することができると考える．

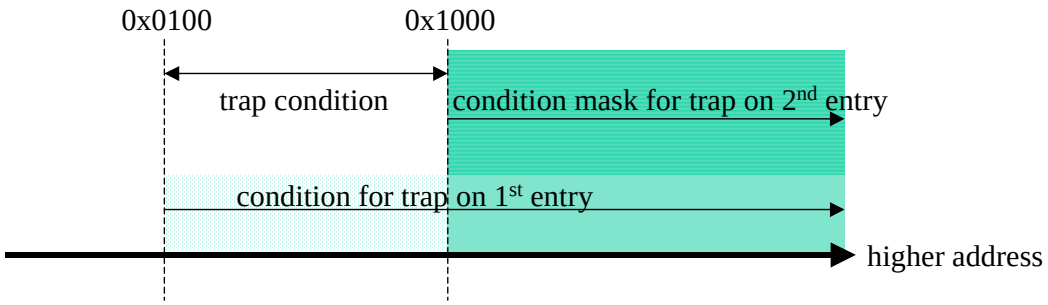


図 5.2: トラップマスクを使用したアドレス範囲設定.

## 5.3 ソフトウェアトレース最適化

4 章では HDOS の最適化の例としてのデータプリフェッチ最適化への適用を示した．本節では，HDOS の他の最適化の適用例として，ソフトウェアトレース最適化を示す．本節は最初にパスプロファイリングによる最適化の有用性について述べ，次の HDOS によるパスプロファイル手法の提案に結ぶ．本節の最後には提案したパスプロファイルを用いたソフトウェアトレース最適化を予備評価する．

### 5.3.1 パスプロファイリングによる最適化

パスプロファイリングによるコード配置最適化はソフトウェアアプローチ，ハードウェアアプローチを問わず研究されてきた．[5][4][36][37][38]．これはプログラム実行時に実行パスの偏りをフィードバックしてプログラムコードの配置を変更することで，命令 cache ミス数などを削減し実行効率向上をめざすアルゴリズムである．

本節では，HDOS 適用による動的最適化の提案例としてソフトウェアトレース生成アルゴリズムを示す．ソフトウェアトレースとはパス上の基本ブロックを主記憶上に連続に置き，スーパースカラマイクロプロセッサの命令フェッチ効率向上を狙う最適化である．本質的にはトレースキャッシュと同じ効果を狙うが，ソフトウェアの最適化の貢献のため，トレースキャッシュのハードウェアを必要としない．トレースキャッシュは複雑で高コストであるため，DRAM の安価化が進む昨今，本提案は組み込みシステムなどに適用できると考える．

トレース生成アルゴリズムは実行時情報を基にバイナリコード中の頻繁に実行される基本ブロックとその実行順序を特定する．この処理は一般的にパスプロファイリングと呼ばれる．

### 5.3.2 HDOSによるパスプロファイリング

ソフトウェアトレースを生成するためには，頻出パスを見つけなければならない．これにはいくつかのアルゴリズムが考えられる．本節ではHDOSのトラップを使いハンドラで実装されるパスプロファイルアルゴリズムを示す．

### 5.3.3 アルゴリズム

提案するアルゴリズムはパスプロファイリングとトレースの生成をループに関してのみ行う．基本的なアルゴリズムのコンセプトは比較的コストの小さいMRET（Most Recently Executed Tail）アルゴリズム[4]を採用し，ループ解析を行う．これは，ループ回数の多いループボディ上の基本ブロックは比較的実行回数が多いという予測に基づいた方針である．

最適化トラップハンドラは大きく分類して4つの処理を行う．

**1** ループ検出

実行中プログラムからループボディを見つけ出す．

**2** パスプロファイリング

ループ内の実行パスのプロファイリングを行い，最も実行される可能性の高いパスを見つけ出し，ソフトウェアトレース生成に必要な分岐命令リストを作成する．

**3** トレース生成

生成された分岐命令リストからそれぞれの分岐命令を含む基本ブロックを抽出しソフトウェアトレースとして実行可能なように加工する．

**4** トレース配置／トレースリンク

生成されたトレースを主記憶上に配置し，オリジナルコード上のトレースの先頭命令をトレースへのジャンプ命令に変換し，作成したソフトウェアトレースが実行されるようにオリジナルコードを修正する．

### 5.3.4 ループ検出

最適化処理はソフトウェアトレースを生成するが，プログラム中に存在する全ての実行トレースを生成することは非現実的である．そのため，最適化処理はまず，プログラム中の頻繁に実行される基本ブロックの特定を行う．頻出基本ブロックを特定する足がかりとして，ループ検出を行う．ユーザ定義トラップを用い，ループバックする分岐命令でトラップを発生させその履歴を取る．最適化処理は一定回数以上のループバック (後方分岐) を行った分岐命令を見つけ，その分岐命令の taken 方向の分岐先から辿って，再度，該当分岐命令に到達するまでのパスをループボディと認識する．UDT を使用したループ検出の具体的方法は 4.2.2 節のループ検出方法と同様である．ループボディ内には複数のパスが存在することができることから，ループ内部で頻出するパスを見つけなければならない．頻出パス (HOT トレース) 検出は次に行うパスプロファイリングで行う．

### 5.3.5 パスプロファイリング

ループ検出でループボディを特定した後，一定回数のループバックを実行する間，パスプロファイリングモードとなり，UDT を全ての分岐を実行した際にトラップを発生させるように設定する．パスプロファイリングモード中はループ区間内に存在する分岐命令の解析を行う．

本節では，分岐命令グラフを利用したパスプロファイリング手法を提案する．本手法で解析時に作成される分岐命令グラフの例を図 5.3 に示す．

図 5.3 はパスプロファイリングモードをループバック 10 回として，その間に作成した分岐命令解析状態である．図中 A〜F に対応する節はループ内に存在する分岐命令を表し，H がループバック分岐命令を表す．2 方向に伸びる枝は，分岐命

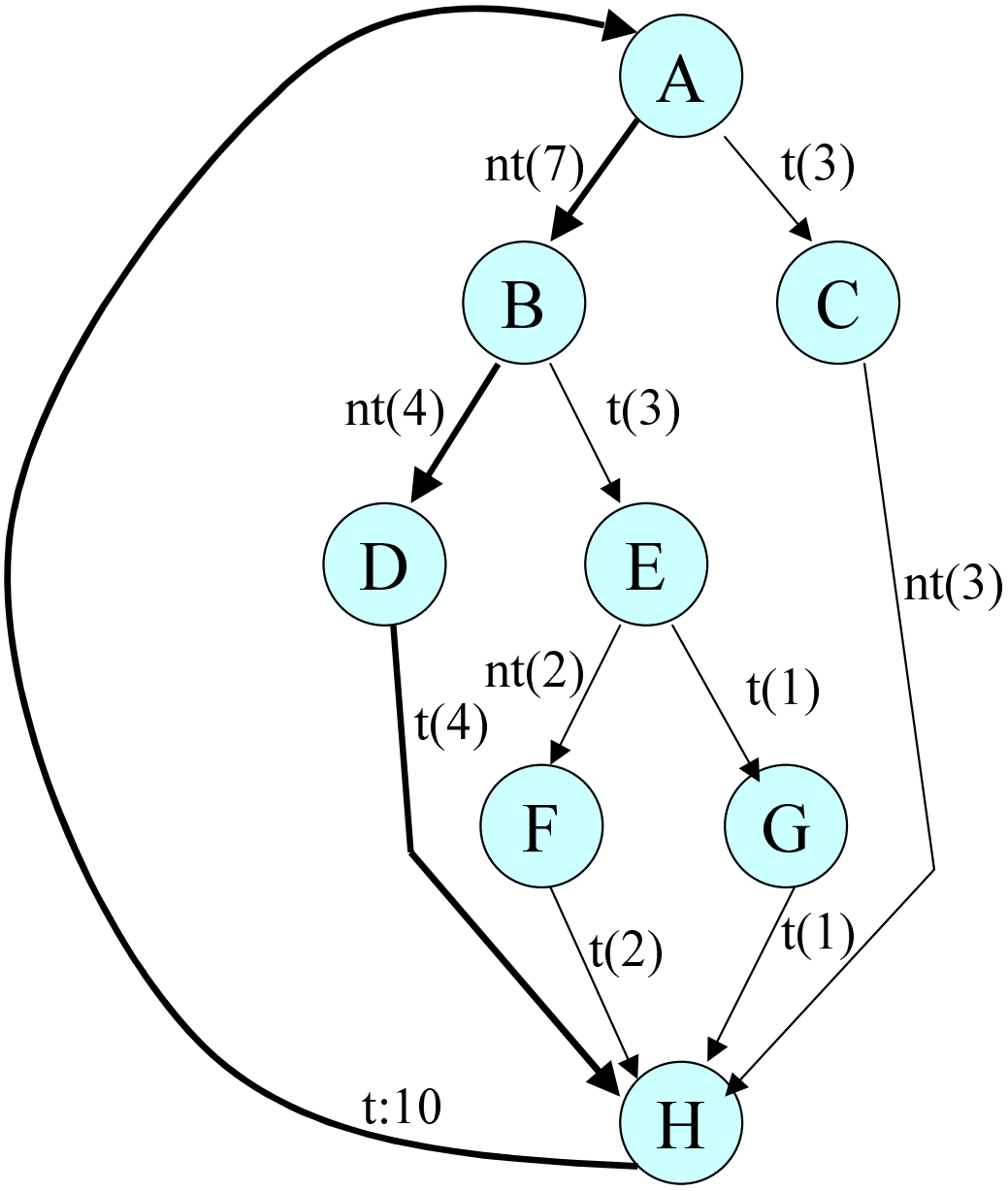


図 5.3: パスプロファイリング中に作成される分岐解析グラフ

令の飛び先方向を表す．それぞれの枝に t と nt と示されるのは taken 方向に分岐したか，not-taken 方向に分岐したかを表している．またこの枝に () 付けで示される値は実際に計測中にこの方向へ分岐した回数を表している．まずプロファイリングを開始したときには H の節のみがあり，分岐を実行する度に発生するトラップで節の追加，枝の重み付けを行っていく．トラップが発生した分岐命令の PC を元にグラフに検索を掛けてグラフ中に同一分岐命令があるかを調べる．無い場合

は新たに節を作成し前回実行した節につなぐ．存在した場合はそのパスは一度実行したパスなので，前回実行節と今回実行節 (ヒットした節) の間の枝の実行回数をインクリメントする．この処理を H から伸びる t の枝の重みが 10 になるまで続ける．解析が終わると，A から H までで実行回数が多い枝を選びながらグラフを辿り，分岐を拾い出していく．図では A，B，D，H が選ばれ，これらの分岐がトレース生成用の分岐リストとして採用される．

### トレース生成

パスプロファイリングで得られた分岐命令リストからソフトウェアトレースを生成するためにはいくつかの手順がある．

**アルゴリズム 3** トレース生成アルゴリズム

**1** 各基本ブロックの抽出

trace tail のループバック先から再度 trace tail に辿り着くまで，収集した分岐履歴を使い，前回ループ実行の軌跡を追いかけながら命令を拾い出していく．

**2** taken 分岐の taken 条件の反転

トレース中に taken で実行した条件分岐が存在した場合，条件反転を行う．(例：bne を beq に変換)

**3** 既存トレースへのリンク

命令拾い出しを行う過程で，インナーループを見つけた場合，インナーループ部は既に作成したトレース中に存在するので，該当するトレースの主記憶上の位置をトレース管理テーブルから検索しその場所へ飛ぶ無条件分岐命令を挿入する．

**4** オリジナルコードへの復帰命令追加

作成したトレースの末尾 (ループバック分岐命令の次) にオリジナルコードに復帰する無条件分岐命令を追加する．

**5** トレース配置位置決定

拾い出しが終了したら，生成したトレースの主記憶上の配置位置を決定する．オリジナルコードに干渉せず，命令キャッシュのブロックサイズにアラインされたアドレスであることが望ましい．

**6** トレース内のリロケーション解決

配置位置が決定した後，トレース中の全てのPC相対分岐命令の飛び先へのオフセットを再計算して，命令の即値を変更する．

以上の処理を行うと主記憶上に配置することが可能なトレースが生成される．

#### トレース配置／トレースリンク

トレースを生成した段階では生成したトレースはトラップハンドラのバッファ内にあり，それを最適化対象バイナリがアクセス可能な範囲に移動し配置しなければならない．また，ただ配置しただけでは生成したトレースは実行されないので，オリジナルコード上のトレースの先頭に相当する命令をトレースへジャンプする命令で上書きしなければならない．これがトレースリンクである．

### 5.3.6 予備評価

予備評価として簡易的なCPUシミュレータを用いてソフトウェアトレースによる命令フェッチ効率の変化を調べた．

#### シミュレーション環境

シミュレーション環境として，MIPS64 ISA のCPUシミュレータを作成した．シミュレータは命令フェッチを計測するためのもので，複数命令フェッチ・実行をするスーパースカラプロセッサシミュレーションを行う．UDTの動作を含む最適化ルーチンはシミュレータコードに記述した．このため，実際のハンドラ部の命令実行は結果に含まれていない．シミュレーションパラメータを表5.2に示す．

ワークロードとして，SPEC 95 INT[26]ベンチマークから8つのアプリケーションを選択した．全てのベンチマークアプリケーションは20億命令完了まで実行した．

表 5.2: MIPS64 シミュレータのパラメータ

| | |
|---|---|
| issue width | 4 |
| branch prediction | ideal |
| ruu size | infinite |
| ruu commit width | infinite |
| L1 cache size (inst. and data) | 64KB |
| L1 block size (inst. and data) | 32B |
| L2 cache size (unified) | 1MB |
| L2 block size (unified) | 64B |

### シミュレーション結果

図 5.4 に計測した命令フェッチスループットを示す．図中の白いバーが非最適化時の命令フェッチスループット，色の濃いバーが最適化後の命令フェッチスループットを示している．命令フェッチスループットは命令実行パイプラインから命令フェッチが要求されたときに，1 アクセスで命令キャッシュから取得できた命令数の平均である．分岐が not-taken 実行される場合，分岐命令の直後のアドレスからのフェッチとなるため，1 ブロックに必要な命令が存在する可能性が高い．逆に多数の分岐命令が taken 実行される場合，命令キャッシュ内に散在するブロックから命令を取得しなければならないため，1 アクセスで命令キャッシュからフェッチできない可能性がある．

`go`，`m88ksim`，`gcc`，`compress`，`ijpeg` ，`perl` で命令フェッチスループットに改善が認められた．しかしながら，`li` と `vortex` は非最適化バイナリコードより命令フェッチスループットが減少した．

この減少はプログラム内のレジスタジャンプ等の可変の飛び先アドレスを持つ命令 (絶対ジャンプ) や条件分岐の分岐方向に偏りが無い命令が多数実行されるアプリケーションでおこる．例えば，`li` はリスプインタリタのアプリケーションであることから，構文解釈後に解釈結果に合わせたルーチンに分岐する処理が存在し，これに起因する絶対ジャンプや分岐方向が偏らない分岐命令が多数実行されていると考えられる．

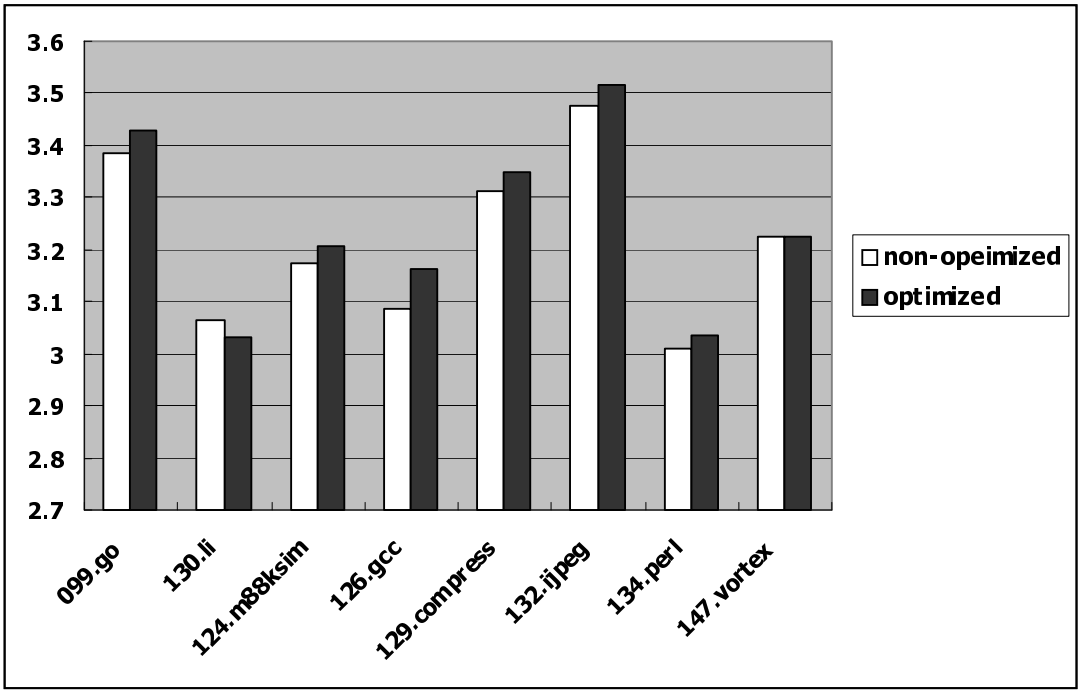


図 5.4: SPEC95 INT における命令フェッチスループットの変化.

本シミュレーションにおいて，8 アプリケーション中 6 アプリケーション，つまり 75%のアプリケーションで提案最適化が有効であることを示した.

## 5.4 キャッシュメモリの電力最適化

本節では、前節で述べたソフトウェアトレース最適化に加え、キャッシュメモリの電力最適化に HDOS を適用する例を示す。

### 5.4.1 研究背景

微細加工技術の進歩により，トランジスタ高集積化が達成されている．近年のマイクロプロセッサはメモリウォール問題がパフォーマンス向上の妨げとなっている．これを緩和するために，高集積化の恩恵をキャッシュメモリの増加に費やす設計傾向がある．このため，現在のマイクロプロセッサは，キャッシュメモリがダイエリアの大部分を占めている．一方で，高集積化によるリーク電力消費の増大が

無視できなくなってきている．リーク電力は面積に比例することから，キャッシュメモリで消費されるリーク電力の割合が大きい．

キャッシュメモリで消費されるリーク電力を削減するために，キャッシュの電源制御を行う手法が提案されてきた．そのような手法として gated-Vdd[65] と cache decay[66] が挙げられる．gated-Vdd はキャッシュメモリを構成する SRAM セルと GND の間に高い閾値を持つトランジスタを儲けることで電力供給を断つ手段を提供している．電力供給が断たれた SRAM セルはリークが発生しない代わりに，それまで保持していた値が破棄される．論文中では，命令キャッシュに着目しており，動的にキャッシュを 2 分割し，それぞれを gated-Vdd により電力供給状態と非供給状態にすることによって，リーク電流削減制御を行う適用例を示している．この方式では 2 分割という粗粒度の制御を行ったが，cache decay は gated-Vdd の制御を細粒度のキャッシュブロック単位で行う．cache decay は当該キャッシュブロックが dead time であるか否かを判断し dead time である場合に電力供給を断つ．キャッシュブロックが deadtime 中であるかの判断を行うためには各キャッシュブロックにカウンタ回路を持たなければならない．

cache decay のように，ハードウェアによる細粒度の電源制御には制御のためのハードウェア資源の追加が必要となる．このことから，追加ハードウェアを必要としない電源制御手法 Software Self-Invalidation[67] が提案された．Software Self-Invalidation は従来研究の Self-Invalidation[68][69] をキャッシュの低消費電力化のために応用した提案である．元来，Self-Invalidation はマルチプロセッサ環境でキャッシュコヒーレント維持のための手法である．Self-Invalidation は他のプロセッサからコヒーレント維持のための無効化要求が来る前に，予測によってあらかじめキャッシュブロックを無効化する．キャッシュブロックの無効化は当該ブロックの保持するデータを破棄する行為であり，無効化された時点で当該ブロックの電力供給を遮断しても構わないことから，電源制御手法に応用することができる．Software Self-Invalidation では，プログラム上，最後のデータアクセスを行う命令をプログラマが認識し，最後のデータアクセスを行う命令を last touch 命令と呼ばれる特殊なロード／ストア命令に置き換えることによって invalidation かつ電源 OFF が実現される．これにより，cache decay の問題であったカウンタ等のハードウェアの追加の必要がなくなる．

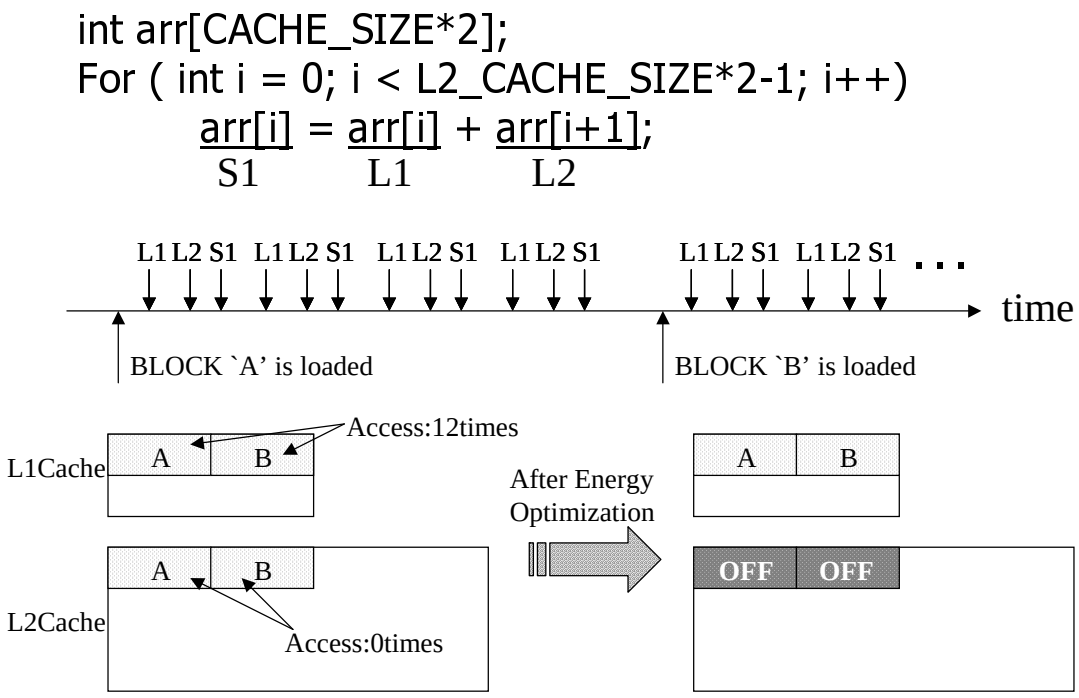


図 5.5: 本研究が対象とする配列アクセス時の電力最適化の適用.

Software Self-Invalidation は last touch 命令の置換候補を見つけるためにメモリアクセストレースを使用する．事前に対象プログラムを実行して発生した全てのメモリアクセスのトレースを収集し，それを解析することによって該当メモリブロックを最後に参照する命令を知ることができる．しかしながら，対象プログラムの実行時間が長い場合，収集するトレースは膨大なものとなり，収集行為が非現実的となる．また，解析とそれに基づくプログラムの修正を人手で行うことはソフトウェア開発効率上，極めて難しい.

この先行研究に対して，本節は，動的最適化システム HDOS を電力最適化のプロセス (トレース収集，解析，バイナリ修正) に適用することによって，Software Self-Invalidation の欠点を克服する.

### 5.4.2 キャッシュ電力制御方式

本研究の目的は L2 キャッシュのリーク電力削減である．図??に本研究が対象とする配列アクセス時の電力最適化の適用事例を示す.

図中で示すプログラムでは，キャッシュブロック A はキャッシュブロック B がロードされるまでアクセスが頻発するが，その後はアクセスされることは無い．この場合，L1 キャッシュでは，ブロック A，ブロック B 共に 12 回アクセスされる (図の例では L1，L2 ともにブロックサイズが 16 バイト，一回のアクセスが 4 バイトであるとする)．この時，L2 キャッシュに着目すると，キャッシュミス時に L2 キャッシュにロードしてから一度もアクセスされておらず，この先アクセスされる予定も無い．このようなメモリ参照では，逐次アクセスや近傍要素同士の演算が原因となり，短期的な空間的局所性のある L1 キャッシュブロックのヒットが頻発する一方で，長期的な時間的局所性を見込んだ L2 キャッシュのヒットが見込めない．本研究ではこのようなアクセスを対象に電力最適化を行う．図 5.5 で示す L2 ブロック A と B はロードされた時点から時間的局所性を見込めないため，L2 キャッシュに存在する意義がないことから，電力供給を断ちリーク電力の低減を行っても性能に影響を与えない．本節では，キャッシュロード時に L1 キャッシュにデータを読み込み，L2 キャッシュにデータを読み込まずに対象ブロックの電力供給を断つ (ただしタグ情報は生成，保持される) ロード／ストア命令を提案する．

提案するキャッシュメモリの電力最適化には，HDOS の他に以下 3 つの要素技術が必要となる．

1. 電源制御が可能なキャッシュメモリ
2. ソフトウェアインターフェース
3. 最適化アルゴリズム

以降はこの 3 つの要素技術の詳細を個々に述べる．

### 電源制御可能キャッシュメモリ

本研究では短期的に時間的局所性の高いキャッシュブロックの参照を L1 キャッシュに任せ，長期的見た場合に，時間的局所性の低い L2 キャッシュブロックの電力供給を止める操作を行う．本研究で対象となる電源制御機能を備えたキャッシュ機構を図 5.6 に示す．

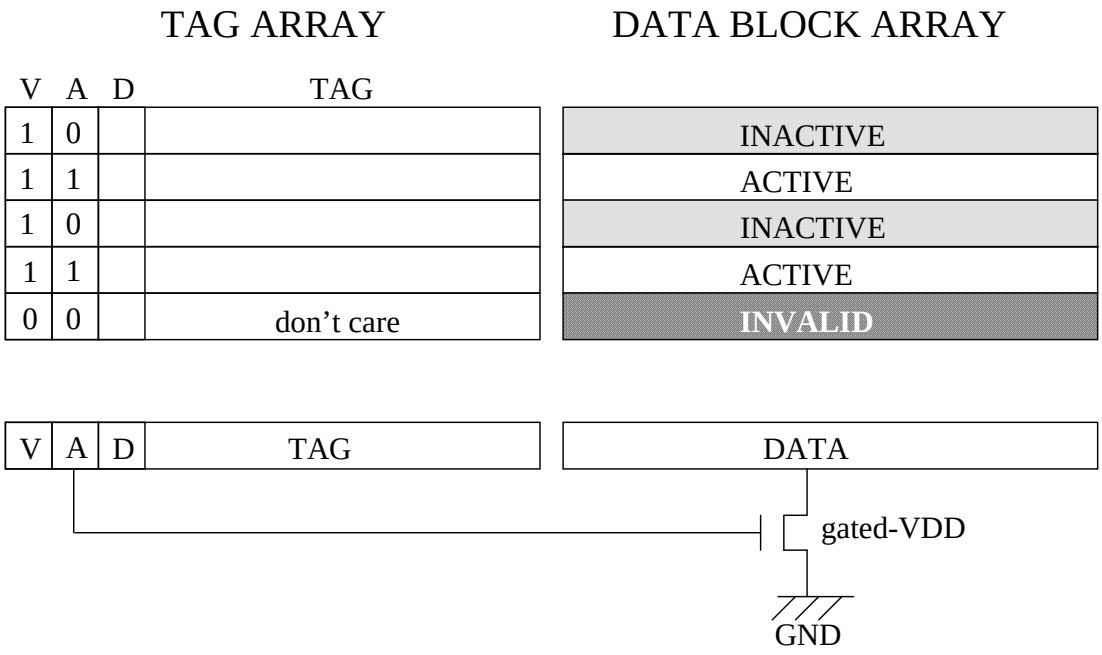


図 5.6: 電源制御機構をもつキャッシュブロック図.

L2 キャッシュの電源制御を行うために，ブロックの電源 ON ／ OFF を示す 1 ビット (図 5.6 の A:Active ビット) を制御情報に追加する．A ビットは gated-Vdd の ON/OFF の制御に直結され (図 5.6 の下部)，A ビットが 1 ならばキャッシュブロックに電力供給がなされている状態で，逆に 0 なら電源供給が止められている状態となる．

対象とする L2 キャッシュは L1 キャッシュのデータの保持状態に関係なく，L2 ブロックの電力供給を断つ．各ブロックの電力供給制御は gated-Vdd を想定するため，ブロックの保持する内容は破棄される．しかしながら，インクルージョンプロパティを守らない場合，マルチプロセッサ環境で，キャッシュコヒーレント問題の解決が難しくなる．そこで，対象キャッシュはブロックのデータ領域の電力供給が断たれた場合でも，タグは放棄せずにブロックの有効ビットを立て続け，インクルージョンプロパティを保全する．(図 5.6 の 1 番目と 3 番目のブロック)

対象とする L2 キャッシュは，データを保持せず，タグを保持する特性から，タグヒットをしたが，電力供給は断たれてデータはすでに破棄されている状態 (ミスシャットダウン) が存在する．ミスシャットダウンは有効ビット (V ビット) と A ビッ

トの参照により検出可能である．ミスシャットダウンの検出時，電力供給は再開 (A ビットをアサート) され，再度主記憶からメモリブロックは当該ブロックエントリに格納される．

また，図 5.6 の 5 番目のブロックのように，invalid であるブロックはタグヒットせず，値を保持する必要が無いため，A ビットは 0 にすべきである．本節における評価では，V ビットを 0 にする場合は同時に A ビットを 0 にするキャッシュ制御を想定して行った．

マルチプロセッサ環境[12]を想定しない場合は，A ビットは削除可能である．その場合，V ビットを gated-Vdd の制御に用いることにより，前述のキャッシュブロック制御と同様の効果が得られる．

### ソフトウェアインタフェース

A ビットを持つ L2 キャッシュを想定した場合に，A ビットをソフトウェアから操作する手段が必要となる．本研究では，A ビットの操作を行うロード／ストア命令を導入した．提案するロード／ストア命令は命令セットに備わるロード／ストア命令とは別に用意される．提案するロード／ストア命令を Inactive-Block(IB) ロード／ストア命令と呼ぶ．IB 命令は命令実行パイプライン上で元来命令セットに備わるメモリアクセス命令と同じ動作をする．唯一違う点は，IB 命令が原因となって L2 キャッシュミスが発生した場合に，置換対象のブロックの A ビットを 0 にする機能を有していることである．冒頭で示した 2 レベルキャッシュの例では，IB 命令が原因で L2 キャッシュミスが起きた場合，まず，L2 キャッシュのタグが更新される．この時，A ビットは 0 にセットされるので，置換対象ブロックの電力供給は断たれる．その後，主記憶から読み出されたメモリブロックが到着すると，対象ブロックは L2 キャッシュには格納されずに (電力供給が断たれているため)L1 キャッシュにのみ格納される．ソフトウェアはこの IB 命令と通常のメモリアクセス命令を使い分けることで，L2 キャッシュにロードするかしないかを制御することができる．

[12]厳密には，コヒーレンスキャッシュの機能を利用した周辺 DMA 転送を含む．

### 最適化アルゴリズム

ここでは，IB 命令を扱う最適化ルーチン上のアルゴリズムを述べる．IB 命令はロード／ストア機能を備えるので，最適化ルーチンの行う操作は最適化対象バイナリ上のロード／ストア命令を IB 命令に置換することのみである．置換対象メモリアクセス命令を見つけるために，最適化ルーチンは最適化対象バイナリ上のどのメモリアクセス命令が長期的に見た場合に低い時間的局所性を示すかを解析しなければならない．

最適化アルゴリズムはまずプログラムのループ構造に着目する．これは，ループ実行中は規則性のあるメモリアクセスを発行することが多く，ループ実行のごく一部のメモリアクセストレースを収集することのみで，対象メモリアクセス命令のアクセスパターンを高確率で予測できるからである．低い時間的局所性を示すアクセスパターンの 1 つとして，冒頭で述べた配列アクセスが挙げられる．このアクセスパターンを見つけるためのアルゴリズムを述べる．

アルゴリズムには以下の 3 段階のステップがある．

1. ループ検出
2. メモリアクセス履歴テーブルの作成
3. 履歴テーブルの検査とバイナリコードへの修正

UDT によって駆動される最適化ルーチンは現在どのステップの処理中であるかで振る舞いを変える．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，HDOS は全ての処理の最初にループ検出を行う (ステップ 1)．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，IB 命令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．ステップ 3 が終了するとステップ 1 に遷移し，プログラム実行で検出できる全てのループに対して同様の最適化を施す．

以降は各ステップの詳細を述べる．

ステップ 1 では，まず，HDOS は UDT のトラップを発生条件設定を後方分岐の Taken 実行にセットする．この UDT 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．

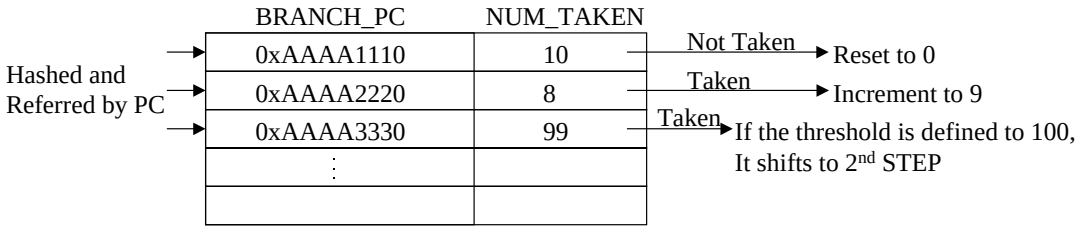


図 5.7: 後方分岐リストの例.

ステップ 1 の主な処理は後方分岐リストの作成である．後方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ため，後方分岐リストを生成することでループバック分岐命令を特定することができる．

作成する後方分岐リストを図 5.7 に示す．テーブルは単純な構成で，後方分岐が連続して Taken で実行された回数 (図中の NUM_TAKEN) を記録している．

最適化ルーチンは呼び出された後，専用命令を用いて RRBE Buffer を読み出し，そこから得られた後方分岐命令の PC 値を用い，既存の後方分岐リストを検索する．テーブルは PC 値をキーにしたハッシュ構造となっており，高速に検索できる．図 5.7 の 1 番目のエントリは Not Taken で最適化ルーチンが呼び出された場合の例である．後方分岐リストは連続した後方分岐回数を記録するテーブルであるので，もし Not Taken でリストにヒットした場合は，NUM_TAKEN を 0 にリセットし，トラップから復帰する．図 5.7 の 2 番目のエントリは Taken で最適化ルーチンが呼び出された場合の例である．この場合は該当するエントリの NUM_TAKEN をインクリメントしてトラップから復帰する．図 4.1 の 3 番目のエントリは最適化ルーチンのステップを次に進める原因となるエントリの例である．最適化ルーチンは後方分岐命令が一定閾値以上連続で Taken 実行された場合に，ステップ 2 に遷移する．例えば，閾値を 100 に設定した場合，NUM_TAKEN が 99 から 100 にインクリメントされた場合に，次回以降のトラップ起動はステップ 2 の処理となる．また，一度 NUM_TAKEN が 100 を超えたエントリは 2 度とステップ 2 への遷移の原因とはならない．

ループバック分岐が検出された後，UDT 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


図 5.8: メモリ参照履歴テーブルの例.

のループで発生するメモリアクセスの解析を行う．この作業を観測フェーズと呼ぶ．図 5.8 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 4 つの項目を持つ. 図中の項目を左から順に示す．MEM_INST_PC はメモリアクセス命令の PC，LAST_ADDR は該当メモリアクセス命令が最後に発行したメモリアドレス，NUM_EXE はメモリアクセス命令の実行回数，UNI_DIREC はメモリアクセスパターンを示すフラグである．履歴テーブルは後方分岐リスト同様，PC 値をキーにしたハッシュ構造となっており，高速に検索できる．図 5.8 の 1 番目のエントリは作成された直後の状態である．初めて観測されるメモリアクセス命令は履歴テーブルにエントリが作成され，LAST_ADDR にその時アクセスしたアドレス，NUM_EXE に 1，UNI_DIREC にはまだアクセスパターンが判明していない NONE がセットされる．の 2 番目のエントリは 2 回目のメモリアクセス命令が実行されたときに行われるエントリの更新結果である．エントリの更新は以下の手順で行われる.

1. 今回アクセスしたアドレスから該当エントリの LAST_ADDR を減算し結果が正か負かを判断する.

2. 該当エントリの UNI_DIREC が NONE なら UNI_DIREC に正 (POSITIVE) か負 (NEGATIVE) のフラグをセットする．もし正でも負でも無いゼロの場合 (前回と同じアドレスにアクセスした) は無効 (INVALID) フラグをセットする.

3. 該当エントリの UNI_DIREC に正か負のフラグがセットされているなら，減算結果と比較する．異なるフラグであるなら，無効 (INVALID) フラグをセッ

トする．

4. 今回アクセスしたアドレスでLAST_ADDRを更新する．

5. NUM_EXEをインクリメントする．

3番目と4番目のエントリは3回目以降の更新の結果である．3番目のエントリは2回目の更新で負のフラグを立てて，3回目のアクセスでも負の結果を得たため，UNI_DIRECは変わっていない．4番目のエントリは50回更新される何処かのタイミングで，UNI_DIRECの不一致があり，無効にセットされている．以上の更新は観測フェーズ終了まで続けられる．

観測フェーズはステップ1で観測した後方分岐命令が一定回数実行されるまで続けられる．観測フェーズが終了した後，最適化ルーチンはステップ3に処理を遷移する．

観測フェーズが終了すると，最適化ルーチンは履歴テーブルをテーブルウォークし，検査を行う．テーブルの中から時間的局所性の低い命令を見つける目的で，NUM_EXEが観測フェーズのループバック数以上[13]であり，かつ，UNI_DIRECがPOSITIVEもしくはNEGATIVEである命令を検索する．条件に一致するエントリが見つかった場合は，MEM_INST_PCの指す番地の命令をIB命令に置換する．

HDOSは部分的なメモリアクセストレースのみを扱うため，ステップ1で検出したループのスコープを超えるような，長期的な時間的局所性の有無を解析することは難しい．しかしながら，大規模なデータセットへのアクセスは容量性ミスを誘発し，有効な時間的局所性を持たないアクセスである可能性が高いことは予測できる．この予測のためにNUM_EXEを命令抽出条件に加え，大規模なデータセットへのアクセスを行うメモリアクセス命令に適用箇所を限定している．

### 5.4.3 評価

ここでは，HDOSのL2キャッシュにおける消費電力削減効果を評価をする．

[13]NUM_EXEはステップ1で検出されないほどループバック回数の少ないループを内包する場合に，観測フェーズのループバック回数より上回る

表 5.3: 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 | 32B |
| ul2 access latency | 6 cycles |
| memory access latency | [first]:120 [inter]:12 cycles |
| memory access bus width | 8B |

**評価環境及び評価方法論**

評価用のシミュレーションは Simple Scalar3.0[25] で行った．対象命令セットは Alpha 命令セット [12] である．シミュレーションの方式は sim-outorder で行った．シミュレーションパラメータを 5.3 に示す．

評価は SPEC CPU 2000[26] ベンチマークの中から 19 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 [27] よりプリコンパイルバイナリを取得し，評価に使用した．全てのアプリケーションは 20 億命令完了まで実行した．

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，シミュレータ上でバイナリ修正を行った．このため，シミュレーション上，最適化によるソフトウェアオーバヘッドは無い．本計測では，このシミュレータ上の実行をバイナリ修正後の実行の評価とみなした．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．

対象とする電源制御可能な L2 キャッシュのシミュレーションを行うために，SimpleScalar のキャッシュシミュレーションモデルに修正を加え，IB 命令で A ビット操作が可能な L2 キャッシュ定義でシミュレーションを行った．

SimpleScalar は命令単位のシミュレーションを行い，クロックサイクル単位でのシミュレーションを行わないため，正確な消費電力評価を行うためには広範囲にわたるシミュレータコードの修正を要する．本評価では，正確な消費電力評価を行わず，約 1000 クロックサイクル毎にアクティブな (A ビットが 1 である)L2 キャッシュブロック数を算出し，その平均をとることで，おおよその消費電力削減効果を見積もる．

### シミュレーション結果

図 5.9 に平均アクティブブロック率を示す．252.eon，256.bzip2，189.lucus に関して，平均アクティブ率が 50%以下となっており，L2 キャッシュの半分以上の領域の電源供給が断たれている．特に 252.eon に関して，19%以下のアクティブブロック率が観測された．252.eon は時間的局所性の低いデータを主に取り扱うアプリケーションであり，HDOS がそれを検出できていたため，積極的に電源制御が働いたと考えられる．それ以外のアプリケーションに関しても，90%〜50%の平均アクティブ率が観測された．INT アプリケーションの平均アクティブ率は 68.50%，FP アプリケーションの平均アクティブ率は 81.65%，全 19 アプリケーションの平均アクティブ率は 74.73%である．

本提案の L2 キャッシュ電力削減手法はインクルージョンプロパティを保全するため，キャッシュのブロックエントリの電力供給を断っても，タグはあたかもそのキャッシュデータがあるかのように存在し続ける．LRU 値も電力制御に関係なく，変動しない．これは，電力制御する場合と，しない場合で，L2 キャッシュミス率が同じであることを意味している．このため，本提案が実行性能を向上させるこ

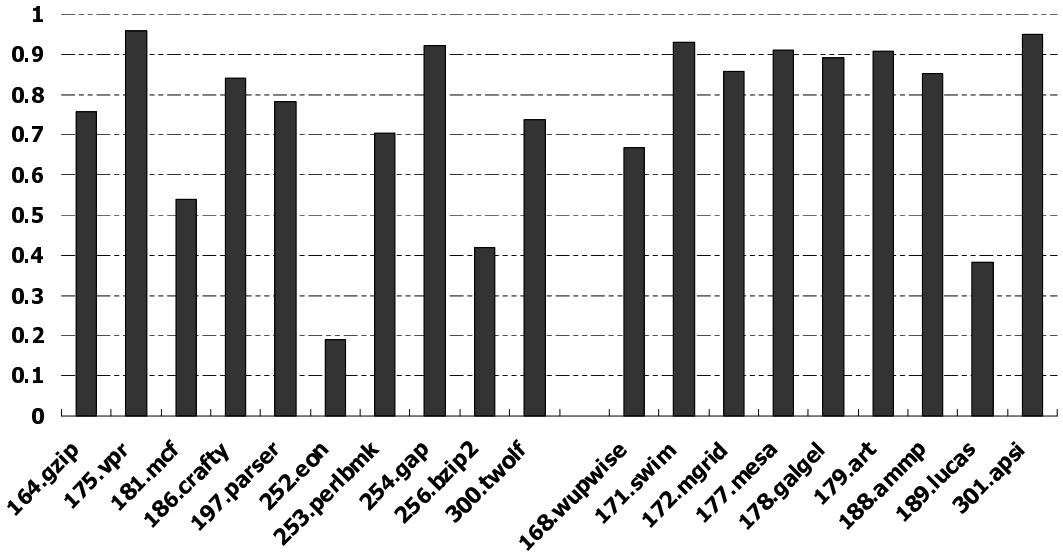


図 5.9: SPEC CPU 2000 における平均アクティブブロック率.

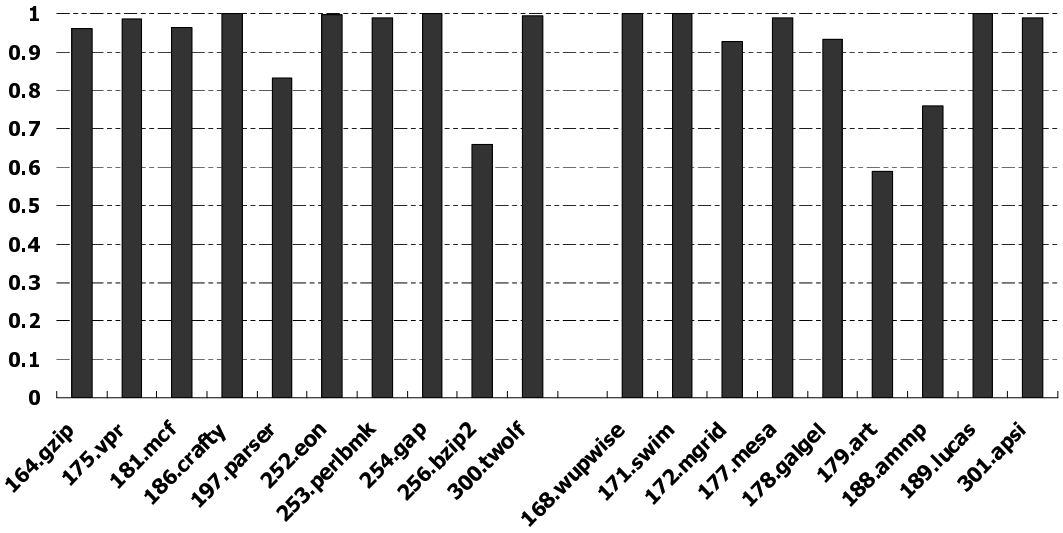


図 5.10: SPEC CPU 2000 における IPC 性能比.

とは無い．逆に，ミスシャットダウンとライトバックによる実行性能の低下は有りえる．これら要因による IPC 性能の低下を図 5.10 にを示す．図は電源制御を行った場合の IPC 性能に対する電源制御を行わない場合の IPC 性能の比である．

197.parser，256.bizp2，179.art，1188.ammp で，性能低下が著しい．それ以外のアプリケーションでは目だった性能比が見られず，性能比はおおむね 90%以内

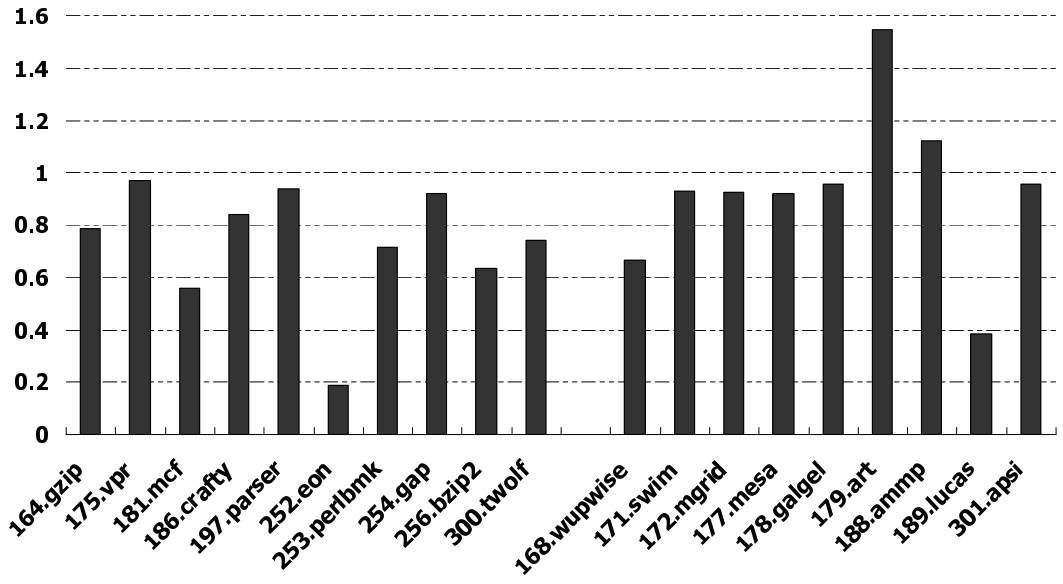


図 5.11: SPEC CPU 2000 における ED 積比.

に収まっている．INT アプリケーションの平均 IPC 性能比は 93.85%，FP アプリケーションの平均 IPC 性能比は 90.99%，全 19 アプリケーションの IPC 性能比は 92.47%である．全体平均で，性能低下は 8%以内に抑えられている．

本提案は L2 キャッシュのリーク電力の削減を目的とする．リーク電力は時間に比例して増大することから，図 5.9 で示した通り，実行性能の低下，つまり，指定された命令数を実行し終わるまでのクロックサイクル数の増加はリーク電力の増大につながる．そこで，L2 キャッシュのリーク電力の指標として，電源制御を行った場合の ED 積 (エネルギー遅延積) に対する電源制御を行わない場合の ED 積の比を図 5.11 に示す.

ED 積比は INT アプリケーションで平均 72.99%，FP アプリケーションで平均 93.42%，全 19 アプリケーションの平均で 82.67%となった．179.art と 188.ammp を除いたアプリケーションでは図 5.9 で示した平均アクティブ率と概ね同じ傾向であることがわかる．一方，179.art と 188.ammp は比が 1 を超えていることから，明らかにリーク電力は増加している．特に，179.art の ED 積比は 1.5 倍に達している．このリーク電力増大を説明するために，ミスシャットダウン率を図 5.12 を示す．ミスシャットダウン率はミスシャットダウン回数／L2 参照回数で計算した.

ほとんどのアプリケーションで，ミスシャットダウン率が 5%以下であるのに対

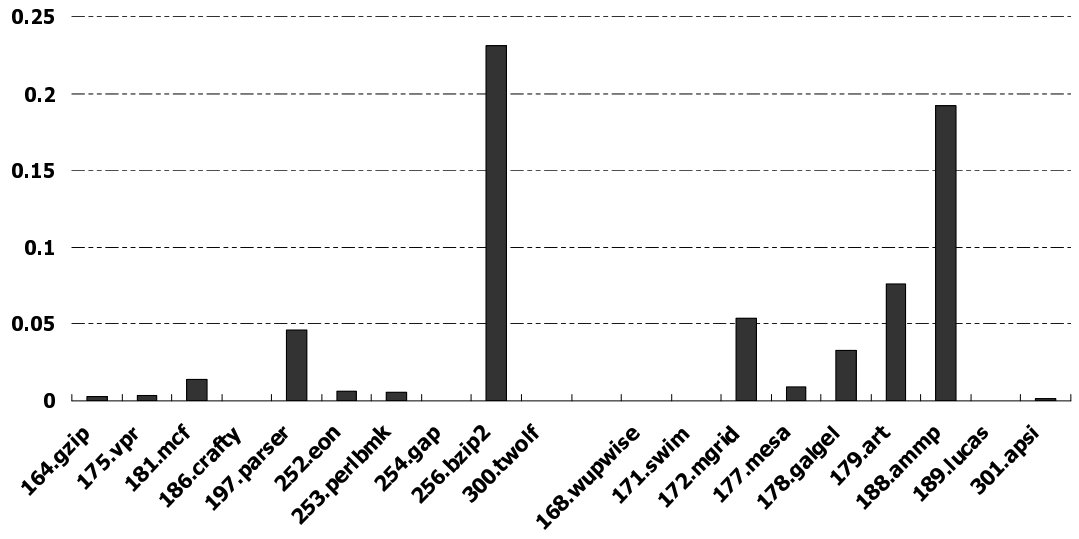


図 5.12: SPEC CPU 2000 におけるミスシャットダウン率.

して，256.bzip2，179.art，188.ammp において，高いミスシャットダウン率が観測されていることがわかる．この 3 アプリケーションに着目して，図 5.9 の平均アクティブブロック率を見ると，256.bzip2 では，ほぼ 60%のブロックの電力供給が遮断されているのに対して，179.art，188.ammp では，電力供給が遮断されているブロックは 15% 未満である．図 5.12 から考察すると，256.bzip2 では，時間的局所性の低いブロックが多数存在し，そうでないブロックも一定数存在した．結果として HDOS はそれらを区別できずに電源供給を断ち，ミスシャットダウンが頻発したと考えられる．これによる性能低下は図 5.10 において顕著に現れているが，それを上回るほど，平均アクティブブロック率が低いため，ED 積比は低い値を示した．一方，179.art，188.ammp では，平均アクティブブロック率が 15%未満であるため，ミスシャットダウンが原因となる性能低下によるリーク電力増大を打ち消すことができず，図 5.11 の結果となったと考えられる．

この結果では，252.eon や 189.lucas のように，極端に良い傾向を示すアプリケーションがある反面，179.art や 188.ammp のように明らかにリーク電力消費が増大する結果が存在した．これは図 5.12 で示したミスシャットダウンが支配的要因となっており，本提案の HDOS の解析では，これらのアプリケーションが要求するアクセスパターンを見極めるためには不十分であるといえる．この主たる要因と

して考えられるのが，ロード／ストア命令間のキャッシュブロックの共有である．最適化アルゴリズムによるメモリアドレストレースの解析は個々のロード／ストア命令のキャッシュミスでロードされるブロックの時間的局所性のみを対象としていたため，ループの中に存在する複数のロード／ストアのアクセスが交差して，長期的な時間的局所性を作り出すケースを解析していない．179.art や 188.ammp ではそのケースが顕著であったのではないかと考察する．今後，HDOS の時間的局所性解析を改善する必要がある．

### 5.4.4 結論

本節では，動的最適化システム HDOS を L2 キャッシュの低消費電力化に適用する手法を示した．提案手法の目的は省ハードウェア資源によるキャッシュメモリの低消費電力化である．HDOS をキャッシュの低消費電力化に適用することによって，事前実行でメモリアクセストレースを収集する必要が無くなり，ソフトウェア生産性を向上させている．

本研究では HDOS によるキャッシュの低消費電力化の手段として，L2 キャッシュにロードせず，L1 キャッシュにロードする IB 命令を提案した．また，HDOS の最適化アルゴリズムとして長期的な時間的局所性の無いアクセスを発生させる命令を特定するアルゴリズムと，IB 命令を使用したネイティブバイナリの修正を提案した．

本節の評価では，SPEC CPU 2000 の中から 19 のアプリケーションを用い，最大で 81%，全アプリケーション平均で 18%の L2 キャッシュのリーク電力削減効果が得られることを示した．個々のアプリケーションの結果から，リーク電力削減に良い傾向を示すアプリケーションが存在する一方で，逆にリーク電力が増大するアプリケーションが存在することが確認された．本節では，ロード／ストア命令間のキャッシュブロック共有に原因があると考察した．

HDOS は部分的なメモリアクセストレースによる予測であるため，ミスシャットダウンが存在する．しかしながら，HDOS の特長である，部分的なメモリアクセストレース収集方法と，最適化の自動化による生産性の向上はソフトウェアによるキャッシュの電源制御を現実的なものにしている．

今後の課題として，ミスシャットダウンを減らすための HDOS の最適化アルゴリズムの洗練が挙げられる．また，今回の評価はアクティブなブロック数のみに着目したが，今後はキャッシュブロックの ON ／ OFF 時に発生する動的電力とミスシャットダウン時のリフィルに伴う動的電力を考慮して評価していく予定である．

# 第 6 章

# 結論

本論文序章では，研究背景として DLL ／ DCL ／ JIT 技術と実行バイナリ最適化の関連研究を紹介し，本論文で提案する HDOS の利点を述べた．

本論文 2 章では，汎用のトラップ発生ハードウェア UDT を提案し，UDT から呼び出されるトラップハンドラで最適化ルーチンを実装した動的最適化システム HDOS を提案した．

本論文 3 章で，HDOS の適用例として履歴アドレス解析ベースのプリフェッチ手法を提案し，評価した．この評価で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もりを行い，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．更に，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極めて小さいことを示した．

本論文の 4 章では，UDT の 2 次利用として，UDT のデバッガサポートを提案し，HDOS のデータプリフェッチ最適化以外の適用の可能性としてソフトウェアトレース最適化を示した．

本論文の 5 章では，バイナリ変換技術と動的最適化研究を紹介し HDOS の違いを述べた．

本研究の成果は動的最適化の細かい実装方法や最適化アルゴリズムの提案ではない．本研究における重要な提案は 2 つある．“UDT” と “コンパイラ最適化の利点を持つオンラインの最適化” である．

UDT の本質は OS 実装者に自らの定義するトラップを提供することにある．UDT は今までのコンピュータシステム設計の常識であった,「ハードウェア設計者の用意したトラップのハンドリングを OS 実装者が実現する」関係に加えて,「OS 実装者がトラップとトラップハンドリングの両方を用意する」概念を加えることになる．両者の住み分けは明確である．例えば外部割込みや TLB ミスのようなプログラムが関知できないトラップ処理は前者の実装で行い，4 章で示したデバッガサポートのような実行するプログラムに関するトラップは後者で実装する．4 章で示した Intel 64 及び IA-32 のデバッガサポート機能の例では専用のレジスタをいくつも用意し，専用機能を実現している．少なくとも，UDT を導入すればこの単機能専用レジスタを廃することができ，かつ高機能なデバッガを構築できる．Intel が発行する System Programming Guide は計 1700 ページを超え，そこで紹介される専用レジスタのいくつかは UDT で代用できる．まとめれば，今日の CPU 設計者は OS 実装者に対して「多種のトラップと言うサービスを提供しなければならない」という既成概念を持ち，自ら CPU 機能の肥大化を招いているとも言える．4 章で示した UDT 適用例の他に以下の使用例が考えられる．

**1** パフォーマンスモニタとしての利用

**2** OS が不正実行を監視するセキュリティシステムへの適用

**3** OS に対して効率的なスケジューリングを行うチャンスを付与

**4** SMT ／マルチコア CPU の投機実行への適用

これら，適用例は UDT でなくとも専用のトラップ発生ハードウェアを持つことで実現可能であるが，汎用トラップ発生ハードウェアである UDT が 1 つ有れば十分である．コンピュータシステムに新しい機能を要求するとき，演算器，メモリ，データパスをアルゴリズムに囚われずに使いまわす今日までの CPU の設計方針か

ら考えると，UDT のようなトラップ発生ハードウェアを使いまわす設計は当然の流れであるかもしれない．

最後に，重要な提案のもう一方の “コンパイラ最適化の利点を持つオンラインの最適化” の詳細を述べる．HDOS の最適化の本質は，コンパイラ／プロファイルベース最適化の実行時に最適化オーバヘッドを支払わなくて良いメリットを動的最適化における 1 度目の実行に関連付けたことにある．つまり，動的最適化であろうと，1 度支払えば良いオーバヘッドは毎回支払う必要はないという提案である．この顕著な例として，インタプリタや VM といったプログラム実行環境が挙げられる．インタプリタや VM は 1 度行えばよい構文解析や命令解釈を毎回行う．これにより，当然ながら，CPU のネイティブバイナリ実行に比べて実行効率は著しく劣る．本論文で示したデータプリフェッチの例では，この非効率要因としてデータプリフェッチのための専用ハードウェアの行う計算を取りあげている．ハードウェアプリフェッチはプログラムのバックグラウンドで専用ハードウェアが履歴解析などの計算を行うことから，この計算がソフトウェアの実行速度低下にはつながらない．しかしながら，そこには同一プログラム実行に際して毎回同じ計算を行うハードウェアが存在している．今回提案したように，履歴ベースのプリフェッチはハードウェアを用意しなくとも実現できる手法であり，ハードウェアプリフェッチはプリフェッチのためのオーバヘッドを隠蔽するためのハードウェアであるといえる．計算を行う手段がソフトウェアかハードウェアかは問題ではなく，1 度行えば良い計算を複数回行うことは計算力を無駄に使用していることに他ならない．そこで，3 章における重要な提案は “コンパイラ最適化の利点を持つオンラインの最適化” の特性を用いて，1 度目の最適化オーバヘッドを被ることを条件にプリフェッチのためのハードウェアを除去できるというものである．これは，システムユーザに選択肢を与える提案であり，1 度目の実行速度の低下が致命的なシステムには採用されないが，例えば組み込みシステムのような，システム上で生涯実行されるプログラムが固定である場合には有効な手段であると考察する．

# 謝辞

本研究を行なうに当たり, 終始御指導を賜わった田中 清史 准教授に深謝致します.

また, 日頃から有益な御助言をいただき, 多面に渡って励ましていただいた日比野 靖 教授に感謝致します.

本論文をまとめるに当たって御協力いただいた日比野・田中研究室の諸兄に厚く御礼申し上げます.

最後に，博士後期課程在学中に，学位取得に向け，絶えぬ応援を下さった母，姉，義兄，姪，多くの友人達に心より感謝します.

# 参考文献


[1] Sheng L., Gilad B., Sun Microsystems Inc., Dynamic Class Loading in the Java Virtual Machine, ACM SIGPLAN Notices, Vol 33, Issue 10, pages 36–44, 1998

[2] James N., Dynamic Class Loading for C++, Linux Journal, Vol 2000, Issue 73es, Article No.38, ISSN:1075-3583, 2000.

[3] Stephen J. Fink and Feng Qian, Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, pages 241–252, 2003.

[4] Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is more. Proceedings International Conference on Architectural Support for Programming Languages and Operating Systems, pages 202–211 2000.

[5] Alex Ramirez, Josep-L. Larriba-Pey, Carlos Navarro, Josep Torrellas, Mateo Valero, Software Trace Cache, IEEE TRANSACTION ON COMPUTERS VOL.54 NO. 1 JUNUARY 2005,

[6] Microsoft Developer Networks, The Microsoft Journal for Developers, msdn magazine aprill 2009, columns, CLR INSIDE OUT:CLR Optimizations In .NET Framework 3.5 SP1. http://msdn.microsoft.com/en-us/magazine/dd569747.aspx

[7] Ishizaki K.,Kawahito M.,Yasue T.,Komatsu H., and Nakatani T., A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, Proc. of ACM

SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pages 294–310, 2000.

[8] Paleczny M.,Vick C. and Click C.,The Java HotSpot Server Compiler, Proc. of Java Virtual Machine Research and Technology Symposium(JVM), 2001.

[9] Suganuma T.,Ogasawara T.,Takeuchi T.,Yasue T.,Kawahito M.,Ishizaki K.,Komatsu H. and Nakatani T., Overview of the IBM Java Just-In-Time Compiler, IBM Systems Journal, Java Performance Issue, Vol.39, No.1, pages 175–193, 2000.

[10] Kaffe Core Team, Kaffe OpenVM, http://www.kaffe.org/

[11] HP-UX Developer Edge, Itanium の「底力」を引き出す HP のコンパイラ技術・後編, http://h50146.www5.hp.com/products/software/oe/hpux/developer/column02/compiler_02/

[12] Alpha Architecture Committee, Alpha Architecture Reference Manual, Third Edition(HP Technologies), ISBN-10:1555582028 ISBN-13:978-1555582029, 1998.

[13] Randal E. Bryant,Alpha Assembly Language Guide, http://www.cs.cmu.edu/afs/cs/academic/class/15213-f98/doc/alpha-guide.pdf, 1998.

[14] Wm.A.Wulf,Sally A. McKee,Hitting the Memory Wall:Implications of the Obvious, ACM SIGARCH Computer Architecture News, Vol.23, Issue 1, 1995, pages 20–24.

[15] Bernstein, D., C.Doron and A.Freund. Compiler Techniques for Data Prefetching on the PowerPC. Proc. of International Conference on Parallel Architectures and Compilation Techniques, pages 16–19, 1995.

[16] 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, pages 264–273, 1997.

[17] K.C.Yeager, The MIPS R10000 Superscalar Microprocessor, IEEE Micro, Vol.16, No.2, pages 28–41, 1996.

[18] Muchnick, S.S. Advanced Compiler Design and Implementation. Morgan Kaufmann, SanFrancisco, CA, 1997.

[19] J-L.Baer and T-F.Chen, Effective Hardware-Based Data Prefetching for High Performance Processors, IEEE Transactions on Comuters, Vol.44, No.5, pages 609–623, 1995.

[20] 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.

[21] 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.

[22] Alan Jay Smith,Cache Memories,ACM Computing Surveys(CSUR),Vol.14,Issue 3,pages 473–530.1982.

[23] K.Nesbit and J.Smith, Data Cache Prefetching Using a Global History Buffer, Proc. of 10th International Syumposium on High-Performance Computer Architecture, pages 96–105 , 2004.

[24] S.Somogyi, T.Wenisch, A.Ailamaki, B.Falsafi and A.Moshovos, Spatial Memory Streaming, Proc. of 31st Annual International Symposium on Computer Architecture, pp.252–263, 2006.

[25] 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.

[26] http://www.spec.org/

[27] http://www.simplescalar.com/

[28] http://www.simplescalar.com/docs/ANNOUNCE-3v0c.txt

[29] J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, P. Yew, and D. Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System, Proc. of 36th International Symposium on Microarchitecture, pages 180–190, 2003.

[30] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system, J. Instruction-Level Parallelism, 6, 2004.

[31] 天嵜聡介, 吉富隆, 水野修, 菊野亨, 高木徳, ソフトウェア開発における不具合発見履歴最終品質の関係に対する統計的分析, 信学技報 TECHNICAL REPORT OF IEICE.,Vol.SS2002,No.6,pages 31–36, 2002.

[32] GDB:The GNU Project Debugger, GDB Internals Manual, http://sourceware.org/gdb/current/onlinedocs/gdbint/

[33] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:System Programming Guide, Part 1, Chapter 16 Debugging, Profiling Branches and Timestamp Counter, http://www.intel.com/Assets/PDF/manual/253668.pdf ,2009.

[34] http://subversion.tigis.org/

[35] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. Dynamo：A Transparent Dynamic Optimization System. SIGPLAN Conference on Programming Language Design and Implementation, pages 1–12, 2000.

[36] Kistler.T.,Franz.M.,Continuous Program Optimization:Design and Evaluation, IEEE Transaction on Conputers, Vol.50, Issue 6, pages 549–566, 2001.

[37] Merten M.C, Trick A.R., Nystrom E.M, Barnes R.D., Hwu W.-M.W,A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots, Proc. of the 27th International Symposium on Computer Architecture, pages 59–70, 2000.

[38] Matt T. Yourst, Kanad Ghose, Incremental Commit Groups for Non-Atomic Trace Processing, Proc. of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pages 67–80. 2005.

[39] http://www.intel.com/products/processor/itanium/index.htm

[40] J. Larus and E. Schnarr, EEL: Machine-Independent Executable Editing, Proc. of PLDI'95, 1995.

[41] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey and Brian Lewis, Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework, SUN Microsystem Technical report, TR-2002-105, http://research.sun.com/techrep/2002/abstract-105.html, 2002.

[42] Rosetta,Apple Inc.,http://www.apple.com/rosetta/, Since 2001.

[43] E. K. Kukureshkov,N Grozdanov,G. N. Gaydadjiev, S. Vassiliadis, SCISM IA-32 Binary Translator, Proc. of ProRISC'03, pages 501–504, 2003.

[44] Jiunn-Yeu Chen,Wuu Yang,Jack Hung, Charilie Su and Wei Chung Hsu, A Static Binary Translator for Efficient Migration of ARM baseed Applications, Proc. of the 6th Workshop on Optimization for DSP and Embedded Systems(ODES-6), pages 55-64, 2008.

[45] Philips Semiconductors, An Introduction To Very-Long Instruction Word(VLIW) Computer Architecture, http://www.nxp.com/acrobat_download/other/vliw-wp.pdf

[46] http://www.cs.arizona.edu/solar/index.html

[47] Noah Snavely,Saumya Debray,Gregory Andrews, Predicate Analysis and If-Conversion in an Itanium Link-Time Optimizer,Proc. of the Workshop on Explicitly Parallel Instruction Set (EPIC) Architectures and Compilation Techniques (EPIC-2), 2002.

[48] Noah Snavely,Saumya Debray and Gregory Andrews, Unspeculation. Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pages 205–214, 2003.

[49] Alexander Klaiber, Transmeta Corporation, The Technology Behind Crusoe Processors, http://perso.citi.insa-lyon.fr/afraboul/master/Crusoe.pdf

[50] Transmeta Corporation, Crusoe Processor Software Optimization Guide, http://www.labri.fr/perso/fleury/hacks/bug_cms/CMS_Reverse/TM5800/Crusoe_SWOptGuide_8-3-01.pdf, 2001.

[51] Baraz L.,Devor T.,Etzion O.,Goldenberg S.,Skaletsky A.,Yun Wang Zemach Y., IA-32 execution layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems, Proc. of 36th Annual IEEE/ACM International Symposium on Microarchitecture, pages 191–201, 2003.

[52] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, John Yates, FX!32: A Profile-Directed Binary Translator, IEEE Micro, Vol.18 Issue 2, pages 56–64, 1998

[53] Paul J. Drongowski, David Hunter, Morteza Fayyazi, David Kaeli, Studying the Performance of the FX!32 Binary Translation System, Proc. of the 1st Workshop on Binary Translation, 1999.

[54] Cindy Zheng,Carol Thompson,PA-RISC to IA-64:Transparent Execution,No Recompilation, IEEE Computer Magazine, Vol.33, Issue 3, Pages 47–52, 2000.

[55] http://groups.csail.mit.edu/cag/rio/

[56] http://dynamorio.org/

[57] D. Bruening. Efficient, Transparent and Comprehensive Runtime Code Manipulation. PhD thesis, M.I.T., 2004.

[58] Q.Zhao, R.Rabbah, S.Amarasinghe, L.Rudolph, and W.-F.Wong. Ubiquitous memory introspection. Proc. of International Symposium on Code Generation and Optimization 2007, Washington, DC, USA, March 2007.

[59] D. Bruening,S. Amarasinghe, Maintaining Consistency and Bounding Capacity of Software Code Caches, Proc. of the International Symposium on Code Generation and Optimization(CGO), pages 20–23, 2005.

[60] Alex Shye. Exploring the Potential of Performance Monitoring Hardware to Support Run-Time Optimization. Master thesis, Univ of Colorado. 2005.

[61] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney, Ispike: A post-link optimizer for the Intel Itanium Architecture, Proc. of the International Symposium on Code Generation and Optimization 2004, pages 15–26, 2004.

[62] R.Cohn, D.Goodwin, and P.G.Lowney, Optimizing Alpha executables on Windows NT with Spike. Digital Technical Journal, 9(4):3–20, 1997.

[63] W. Zhang, B. Calder, and D. M. Tullsen, An Event-driven Multithreaded Dynamic Optimization Framework, Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques, pages 87–98, 2005.

[64] W. Zhang, B. Calder, and D. M. Tullsen,A Self-repairing Prefetcher in an Event-driven Dynamic Optimization Framework, Proc. of the International Symposium on Code Generation and Optimization, pages 50–64, 2006.

[65] Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.N., "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories"",Proc. of ISLPED, pp.90-95, 2000.

[66] Kaxiras, S., Hu, Z., Martonosi, M." Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power", Proc. of 28th ISCA, pp.240-251, 2001.

[67] Kiyofumi Tanaka, Takenori Fujita, " Leakage Energy Reduction in Cache Memory by Software Self-Invalidation ", Proc. of 12th Asia-Pacific Computer Systems Architecture Conference (ACSAC), LNCS 4697, ISBN 3-540-74308-1, Springer, pp.163-174, 2007.

[68] Lebeck, A.R., Wood, D.A.: Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors, Proc. of ISCA, pp.48-59, 1995.

[69] Lai, A.C., Falsafi, B.: Selective, Accurate and Timely Self-Invalidation Using Last-Touch Prediction, Proc. of ISCA, pp.139-148, 2000.

# 本研究に関する発表論文


[1] 請園 智玲，田中　清史: “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” 情報処理学会論文誌 IPSJ Trans. ACS 第 28 号 (2009 年 11 月) 採録通知済.

[2] Tomoaki Ukezono，Kiyofumi Tanaka: “Dynamic Binary Code Translation for Data Prefetch Optimization”，The 2008 International Symposium on Frontiers in Computer Architecture Design. In conjunction with The Thirteenth IEEE Asia-Pacific Computer Systems Architecture Conference (ACSAC ’08), IEEE Digital Publishing Proc. of ACSAC 2008，(2008 年 9 月).

[3] Tomoaki Ukezono，Kiyofumi Tanaka: “HDOS:An Infrastructure for Dynamic Optimization”，Proc. of The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA ’08). pages 33–39.

[4] 請園 智玲，田中　清史: “データプリフェッチ最適化のためのバイナリ変換手法”，The 2008 Symposium on Advanced Computing Systems and Infrastructure (SACSIS ’08). pages 187–194.

[5] 請園 智玲，田中　清史: “ソフトウェアトレースにおけるレジスタ再割り当てアルゴリズムの検討”, 平成 18 年度電気関係学会北陸支部連合大会講演論文集 CD-ROM E-65.

[6] 請園 智玲，田中　清史: “ソフトウェアトレース生成による動的最適化の予備評価”，情報処理学会研究報告 ARC, Vol.2006, No.88, pages 169–174, SwoPP 高知 2006.

[7] 請園 智玲，田中　清史: “バイナリコードの動的最適化を実現する実行時ハードウェア解析システム”，平成 17 年度電気関係学会北陸支部連合大会講演論文集 CD-ROM E-50.

[8] 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, pages7–12, デザインガイア 2005.