2パス限定投機方式の提案

横田隆史[†] 斎藤盛幸[†,☆] 大津金光[†]
古川文人[†,☆☆] 馬場敬信[†]

LSIの高集積化にともない，計算機システムで利用可能なハードウェア資源の量は拡大の一途をたどっているが，一方でクロック速度の向上が飽和する状況になっており，命令レベル・スレッドレベルの並列性を活かした効果的な実行方式が求められている．本論文は，実行頻度の高いホットループに対して，次のイテレーションで行われる実行経路（パス）を予測して投機実行するパスベースの投機的マルチスレッド処理に関して，スレッドレベル並列性を得るための現実的かつ効果的な方法を検討する．パスを投機の対象とすることで，スレッド間依存の問題の緩和や，スレッドコードの最適化が図れるメリットを享受できるが，その一方で，効果的なパスの予測方法・投機方法が課題となる．本論文では，一般的なプログラムでは多くの場合，予測・投機の対象を実行頻度の高い2つのパスに絞っても実質上問題にならないことを示し，2つのパスに限定して投機実行する2パス限定投機実行方式を提案する．実行頻度の上位2つのパスが支配的である場合は，最初のパスの投機に失敗しても次点のパスが高確率で成功するために実行効率を上げられる．本提案方式をモデル化し解析的に性能見積りを行うとともに，2レベル分岐予測器をもとにしたパス予測器を用い，トレースベースのシミュレータにより評価を行い有効性を示す．

Two-Path Limited Speculation Method

Takashi Yokota,[†] Moriyuki Saito,[†,☆] Kanemitsu Ootsu,[†]
Fumihito Furukawa[†,☆☆] and Takanobu Baba[†]

Modern microprocessor systems take their advantages by exploiting large hardware resources in a single chip and by accelerating clock speed. However, in near future, LSI integration will be continued while clock speed be saturated. Thus efficient instruction- and thread-level parallelism is required to achieve higher performance. This paper addresses a path-based speculative multithreading, where frequently executed path is predicted and executed speculatively. We propose a practical speculation method for path-based speculative multithreading. Most practical programs execute only one or two paths in hot-loops, while there are many possible paths according to many branches. We show most frequent two paths are practical candidates to predict and speculate, and thus we propose the two-path limited speculation method. Analytical performance estimation and trace-based simulation results show effectiveness of the proposed method.

1. はじめに

半導体プロセスの微細化は現在なお着実に進行しており，これにともないLSIチップの高集積化も着実に進んでいる．しかしその一方で，微細化に見合った回路性能の向上を得にくくなっているため，半導体素子の改良やクロック速度の向上に頼ることなく，アプリケーションプログラムの実行を高速化する手法が求められている．

この課題に対して，命令レベル並列性（ILP）およびスレッドレベル並列性（TLP）を抽出し，豊富なハードウェア資源を効果的に使用することで解決を図るアプローチがなされている．マルチスレッド処理はその1つである．しかし，広く一般のプログラムに対してマルチスレッド化を適用しても，十分な高速化を果たせない場合がある．このため十分な効果を得るには，プロファイリング等の手段によってプログラムの実行挙動を把握し，適切な箇所に，適切な方法でマルチスレッド化を適用することが肝要となっている[1)]．本論文では，このようなプロファイリングの適用を前提と

† 宇都宮大学
Utsunomiya University
☆ 現在，富士通SCMシステムズ株式会社
Presently with Fujitsu SCM Systems Limited
☆☆ 現在，帝京大学
Presently with Teikyo University

し，プログラムの実行パス[2)~4)] に着目したマルチスレッド手法について議論する．そして，特に，スレッドが実行されるプログラム上の経路（パス）を投機の対象とするパスベース投機的マルチスレッド処理に対象を絞り，現実のアプリケーションの実行挙動に即した現実的な投機方式を検討する．

マルチスレッド化を行う場合，実行頻度が高く繰返し回数の多いホットループを対象とし，ループ・イテレーションの単位でスレッドを生成することが効果的である．さらに，スレッドの実行パスが特定できれば，それに特化したコードを用意することで，より効果的な実行が可能になるものと期待される．しかし，多くのプログラムでは，ループのイテレーション記述中に複数の条件分岐を含むために，パスを特定できないことが多い．これは実行頻度の高いホットループであっても例外ではない．可能性のあるパスの数は，条件分岐の数に応じて指数関数的に増えていく．

しかし現実のプログラムでは，可能性のあるすべてのパスが満遍なく実行されるケースは稀であり，多くの場合 1 つないしごく少数のパスが実行される．つまり，頻繁に実行されるパス（ホットパス）を抽出し，投機の対象とするのが現実的かつ効果的なアプローチである．本論文では，こうした考えに基づき，投機対象のパスを 2 つに限定することで，実用上必要十分な投機を行えることを示す．

以下本論文では，2 章においてパスベースの投機的マルチスレッド実行モデルについて述べたのち，3 章で投機実行の対象を上位 2 つの頻出パスに限定できることを示し，このことを用いた投機実行方式を提案する．さらに 4 章において，提案手法を現実に応用する際に必要になる予測の手法について，2 レベル分岐予測器から発展させたパス予測器を示し，5 章で，その予測器を用い提案投機方式を適用した場合の評価を行う．その後 6 章で関連研究との比較を行い，最後に 7 章でまとめる．

2. 投機的マルチスレッド処理モデル

まず，本論文において我々が前提にしているマルチスレッド処理，パスベース投機的マルチスレッド処理のモデルを明確にしておく．

2.1 マルチスレッド処理モデルの要件

いわゆる fork 型のマルチスレッド実行を行うものとする．すなわち，先行のスレッドが後続のスレッドを順次 fork することにより複数のスレッドを起動し，先行・後続のスレッドの重畳によってスレッドレベルの並列性を得る．このマルチスレッド実行モデルにおいて高い処理効率を引き出すためには，以下の 2 つの条件を検討しなければならない．

(1) **スレッド間並列性の抽出**　スレッドレベル並列性（TLP）を引き出すために，できるだけ多くのスレッドを同時に実行させるようにしなければならない．そのためには，先行・後続のスレッドの重畳をできるだけ大きくすることが求められる．

(2) **スレッド自身の最適化**　スレッドの実行のために使用できるハードウェア資源は現実的に限られることをふまえた議論を行う必要がある．すなわち，各スレッドに対して最大限の最適化を施し，スレッドが自身の実行のためにハードウェアを占有する時間をできるだけ短くする．

さらに本論文では，議論を簡潔にするため，ループイテレーションを単位にスレッドを生成することとする．

前者の条件に対して障害となるのは，先行のスレッドが生成した値を後続のスレッドが使用するスレッド間依存である．このスレッド間依存があると，後続のスレッドは先行のスレッドが目的のデータを生成するまで待たなければならず，このためにスレッドレベル並列性が低下することになる．

2.2 パスベース投機的マルチスレッド処理

前節に示した 2 つの要件を最もよく満たすのが，スレッドで実行されるパス[3)] を予測し，予測パスに特化したコードを実行するパスベース投機によるマルチスレッド処理である．

図 1 に基本ブロックを単位としたループ構造の例を示す．このループでは A → B → G（#1），A → C → D → F → G（#2），A → C → E → F → G（#3）の 3 本の実行パスがある．1 回のループイテレーションで実行されるのは，これら 3 本のパスのいずれかである．パスごとに特化した最適化コード（以下**投機スレッドコード**）を用意しておき，スレッドを起動する際に，そのスレッドがどのパスを実行するかを予測し，対応する最適化コードを実行する（**図 2** 参照）．

この投機スレッドコードは，条件分岐をはじめパスの実行に必要ない処理をすべて削除したうえで，パスに沿った基本ブロックをひとまとめにして，最大限の最適化を行ったものとなる．ただし，予測したとおりのパスが実行されたことを保障するために，投機スレッドコードには，条件分岐に相当する `assert` 命令を挿入する．たとえば図 1 のパス#1 に対応するコードでは，基本ブロック A → B が実行されることを確認するための `assert` 命令を挿入する．もしこの `assert`

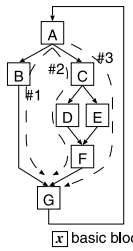


図 1 ループ構造の例
Fig. 1 Loop structure example.

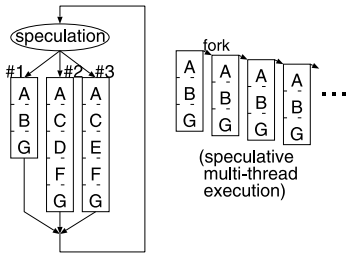


図 2 本論文での投機的マルチスレッド実行モデル
Fig. 2 Multithread model.

命令によって#1 以外のパスの実行が検出された場合，当該スレッドの投機は失敗となり，その実行はアボートされる．

プログラムに記述され実行される可能性のあるパスすべてに対して，スレッド間依存を満足するように，スレッドの実行内容を決めなければならない．このため，一般に後続のスレッドは，依存の可能性のある変数を参照するとき，少なくとも先行のスレッドの実行パスが確定するまで待つ必要がある．先行スレッドの実行パスによっては必要のない同期待ちが行われることになり，実行効率を大きく損なうことになる．

パス単位での予測と投機実行を導入することで上記の問題が大きく改善されるほか，さらに予測対象パスを頻度の高いものに絞り込むことによって，投機スレッド間（パス間）の依存を高精度に把握することが可能になる．すなわち前節の条件 (1) が満たされる．

またさらに，予測パスに特化したコードを投機実行することにより，前節の条件 (2)，すなわち，スレッド自身の最適化が図られる．

3. 2 パス限定投機方式

3.1 対象パスの限定

前章の投機的マルチスレッド実行モデルでは，実行する可能性のあるパスの中から 1 つを予測して投機実行していた．このモデルをそのまま実現する場合，

(1) 可能性のあるすべてのパスについて投機スレッドコードを用意しておかなくてはならない，

さらに，

(2) すべてのパスの組合せについて依存関係の解決をしておく必要がある，

などの問題がある．特に (2) については，重畳実行の可能性のあるパスすべての組合せを考慮する必要があり，基本的にはこの組合せの数だけの投機スレッドコードを用意する必要が生じる．

しかし，実際には可能性のあるパスが満遍なく均等に実行されるケースは稀であり，たいていの場合，頻繁に実行されるパスとそうでないパスに明確に分かれる．SPECint95 ベンチマークのいくつかのプログラムについて，実行時間の比率の高いホットループを求め，そのホットループの中で実行されるパスの実行頻度を調査した．その結果を**図 3** に示す．`compress` はプログラム compress の中の関数 compress()，`forward_DCT` はプログラム ijpeg の中の関数 forward_DCT()，`killtime` はプログラム m88ksim の中の関数 killtime()，`sweep` はプログラム li の中の関数 sweep() を示す．使用したデータセットは `train` である．

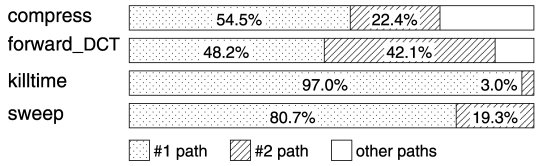


図 3 上位 2 つのパスの実行頻度
Fig. 3 Execution frequncy of top-two paths.

図に示されているように，実行頻度の上位 2 つのパスが全体の約 77%から 100%を占めている．他のベンチマークプログラム（MediaBench）でも，ほとんどのループで実行頻度の高い 2 つのパスが支配的であることが確認されている[5]．このことから，予測の対象を頻度の多い上位 2 つのパスに限定しても実質的に支障がないことが分かる．さらに第 3 位以下のパスは上位の 2 つに比べ実行頻度が低いことから，第 3 位以下のパスを予測して投機実行したとしても，実行頻度の低さゆえに投機が失敗に終わる確率が高くメリットがない．

以上から，現実のプログラムではほとんどの場合，予測・投機の対象を実行頻度の高い上位 2 つのパスに限定することが必要十分であるといえる．

以降，本論文では実行頻度の高い順に#1 パス，#2 パスと呼ぶ．

3.2 対象パスを 2 つに限定した投機実行方式

2.2 節に示したパスベース投機的マルチスレッド処

理モデルをベースとし，さらに前の 3.1 節のように投機対象を実行頻度の高い上位 2 つのパスに絞る場合の実行方式を考える．

2.2 節で述べたように，予測の結果に従ってパスを順次投機実行する．投機が成功している間は図 2 のように次々と投機スレッドを起動していけばよい．しかし，投機が失敗した場合には，当該スレッドの実行を中止し，結果を破棄しなければならない．このとき後続のスレッドがあれば，それらの実行をすべてを中止し結果を破棄する必要がある．そして当該スレッド開始時の状態に戻したのち，適切な方法により実行を再開しなければならない．この投機失敗後の実行再開のときに行う処理内容に選択の余地があり，実行効率が変わる可能性があるため，以下に検討する．

ここで 3.1 節の結果，すなわち，実行頻度の高い上位 2 つのパスが支配的であること，これら上位 2 つのパスを使用すれば十分であること，の 2 点をもとに現実的かつ効果的な実行方式を考える．

図 3 に示されているパスの実行頻度から次のことがいえる．`killtime`，`sweep` の 2 つのパスの実行頻度の合計は 100%である．つまり，`killtime`，`sweep` においてパスの予測が外れて投機に失敗したとしても，実行再開時にもう一方のパスの投機実行を行えばよい（必ず成功する）．`forward_DCT` の場合は，#1，#2 パスの実行頻度の和が 100%に満たないため，上述の `killtime` や `sweep` のようにはいかない．しかし，たとえば，当初#1 を予測して外れたとしても，実行再開時に#2 パスを選べば，$42.1/(100-48.2)=81.3\%$ の高確率で成功することが期待される．`compress` についても同様に，実行再開時に行われるパスの予測成功率は高いことが期待される．

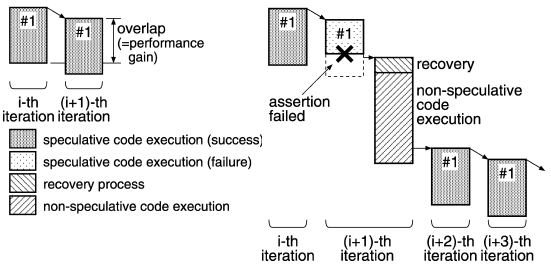
以上から，投機失敗後に実行を再開するときの処理の違いにより，以下の 2 つの投機実行方式を考えた．

Single Speculation（SS）方式

予測に基づいた投機を単に 1 回行う方式である．**図 4** に示すように，投機が成功すればスレッド単位の重畳実行が続けられるが，失敗した場合は，回復処理を行った後に非投機スレッドコードの実行を行う．非投機スレッドコードの実行では，実行パスが特定できないため，後続の投機スレッドの起動を再開するのは，それ以降に依存の可能性のある変数の更新がないことが確定した時点よりあとになる．

Double Speculation（DS）方式

予測に基づいた最初の投機が失敗した場合，回復処理を行った後，もう一方のパスの投機スレッドコードを実行する方式である（**図 5** 参照）．最初の投機失敗時に後続スレッドの実行破棄を行うが，2 回目の投機の際に重畳実行を再開する．2 回目の投機実行も失敗した場合は上述の SS 方式と同様に，回復処理を行った後，非投機スレッドコードの実行を行う．



(a) 投機成功の場合 (b) 投機失敗の場合

図 4 Single Speculation（SS）方式
Fig. 4 Single Speculation (SS) method.

なお，非投機スレッドコードは，条件分岐など元のプログラムの構造を保ったスレッド実行コードを指す．パスベース投機的マルチスレッド処理では，先行スレッドの実行が確定しない状態で次々に後続の投機スレッドを起動する多重投機状態が起こりうる．このため，非投機スレッドコードを実行しているスレッドであっても，先行スレッドの実行が確定していなければ投機状態であり，先行スレッドの投機失敗によりアボートされる．

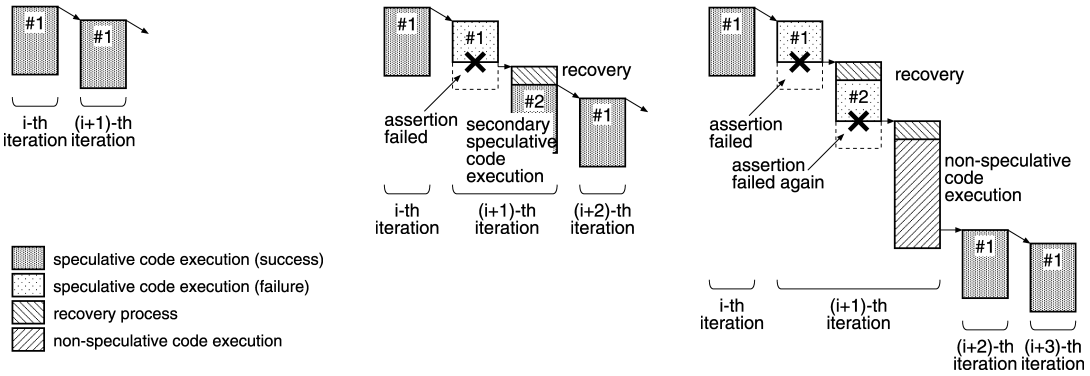
以下，本論文では上の 2 方式をまとめて 2 **パス限定投機方式**と呼ぶ．

3.3 2 パス限定投機方式での性能見積り

ここで，前の 3.2 節に示した 2 パス限定投機方式での性能見積りを，モデルを簡略化することで解析的に行う（**図 6** 参照）．

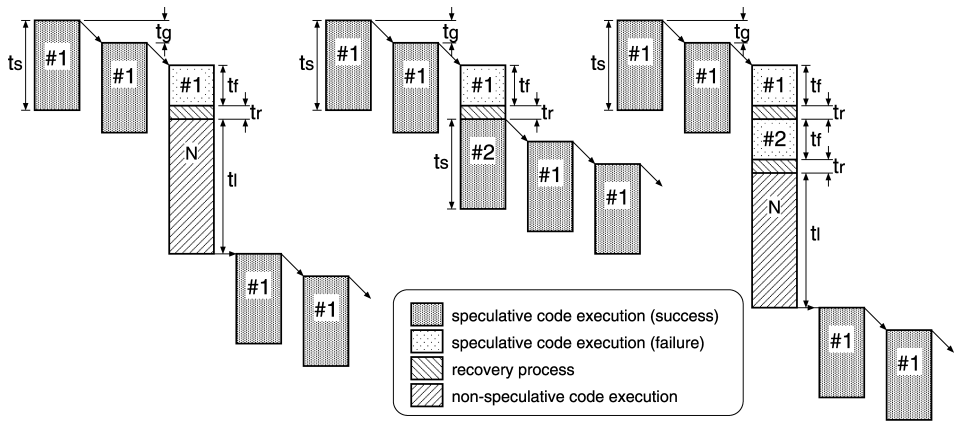
簡略化のため，投機スレッドコードの実行時間（t_s），後続スレッドが実行開始するまでの遅延時間（ギャップ：t_g），投機スレッドコードの実行を開始してから失敗するまでの時間（t_f）は，それぞれパスのいかんにかかわらず一定とし，非投機スレッドコードの実行時間（t_l）も一定とする☆．さらに投機失敗後の回復にかかる時間（t_r）も一定とする．また，#1 パスの出現頻度，#2 パスの出現頻度をそれぞれ p_1，p_2 とし，イテレーション回数を N とする．議論の単純化のため，パスの予測方式については言及せず，最初の投機実行では最大出現頻度の#1 パスを予測し，2 回目の予測（DS 方式の場合）は#2 パスを予測するものと

☆ 正確には，投機スレッドの起動を再開するまでの実行時間を指す．



(a) 投機成功の場合 (b) 最初の投機に失敗した場合 (c)2 番目の投機も失敗した場合

図 5 Double Speculation（DS）方式
Fig. 5 Double Speculation (DS) method.



(a) SPP (b) DPP (1st path fails) (c) DPP (both paths fail)

図 6 2 パス限定投機方式の簡略化モデル
Fig. 6 Simplified model of the two-path limited execution method.

する．

3.3.1 Single Speculation の場合

N 回のイテレーションのうち，投機が成功するのは $N \cdot p_1$ 回であり，失敗するのは $N \cdot (1-p_1)$ 回である．投機失敗のたびにペナルティと実行時間 $t_f + t_r + t_l$ が課される．逆に，投機が成功すれば，見掛け上 1 イテレーションあたり t_g の時間消費で済む．よって実行時間は

$$T_{SS} \approx N \cdot (t_g p_1 + (t_f + t_r + t_l)(1-p_1)) \quad (1)$$

となる．

3.3.2 Double Speculation の場合

N 回のイテレーションのうち，最初の投機実行で #1 が成功するのは $N \cdot p_1$ 回である．最初の投機が失敗して 2 回目の投機が行われる回数は $N(1-p_1)$ 回である．このうち，#2 の投機が成功する回数は，パスの出現頻度から単純に求められ $N \cdot p_2$ である．また，2 回目の投機も失敗するのは $N(1-p_1-p_2)$ 回となる．

#1 または#2 の投機スレッドが成功した場合，1 イテレーションあたりの見掛けの時間消費は，投機スレッド間ギャップの t_g となる．#1 が失敗し#2 が成功した場合のペナルティは $t_f + t_r$ である．さらに両投機スレッドが失敗した場合のペナルティと非投機実行時間は $2t_f + 2t_r + t_l$ である．上記をまとめると，Double Speculation での実行時間は

$$T_{DS} \approx N(t_g p_1 + (t_g + t_f + t_r)p_2 + (2t_f + 2t_r + t_l)(1-p_1-p_2)) \quad (2)$$

となる．

3.3.3 SS，DS 両方式の比較

式 (1)，(2) を比較することにより，それぞれの方法

が有利になる条件が求められる．DS のほうが実行時間が短くなる条件を求めると，$T_{DS} < T_{SS}$ より以下が求められる．

$$p_2 > \frac{t_f + t_r}{t_f + t_r + t_l - t_g}(1 - p_1). \qquad (3)$$

この式を用いることで，実際に実行して確認しなくても SS，DS のどちらが適しているのかをアプリケーションごとに見積もることができる．たとえば，$t_s = 8$，$t_g = 2$，$t_f = t_s/2$，$t_r = 1$，$t_l = t_s * 1.5$ のとき，式 (3) にあてはめると $p_2 > (1-p_1)/3$ である．これを用いて，$p_1 = 0.7$ ならば $p_2 > 0.1$ で DS のほうが速く実行できることが分かる．

4. 2 レベルパス予測手法

3.1 節の結果から，予測対象とすべきパスは実行頻度の高い上位 2 つで十分であることが分かった．予測対象パスを 2 つに限定することで，現実的なハードウェア量で必要十分な精度を得られるパス予測機構を検討することが可能になる．パス予測機構は，次に投機実行すべきものが#1 パスなのか，#2 パスなのかの二者択一をすればよいのである．

図 3 によれば，#1，#2 の上位 2 つのパスが支配的であるが，両者の比率はプログラムによって違っている．すなわち，`killtime` や `sweep` では#1 単独で実行頻度のほとんどを占めているが，`forward_DCT` では#1，#2 が近い値になっており，2 つのパスの頻度の和が 90%になっている．`compress` は上記 2 つのグループの中間的な割合である．#1 が支配的なプログラムでは#1 パスのみを予測しても十分な効果が得られるものと予想されるが，一方で，図 3 での `compress` や `forward_DCT` のように#1 パスの実行頻度がたかだか 50%程度である場合には，十分な投機成功率を望むことはできない．

また，図 3 に示したパスの頻度は，プログラムの実行全体を通しての値であり，プログラムの実行挙動として現れるパターン（ないし規則性）や “phased behavior” を反映したものではない．分岐予測で行われているのと同様に，プログラムの挙動にともなう履歴情報を有効に使用することで，効果的な予測機構を実現できる可能性がある．たとえば，プログラム全体での#1，#2 パスの実行頻度がともに 50%であった場合でも，履歴情報をもとにその時点で頻度の高いほうのパスを選択することにより，50%より高い予測成功率を得られるものと期待できる．

そこで本論文において我々が注目したのは，履歴とカウンタテーブルを用いる 2 レベル分岐予測の手法[6] である．条件分岐の taken，not taken による履歴を，注目しているパスが実行された/されなかったの履歴に置き換える．履歴として蓄積された情報を次に実行されるパスの予測に使用することで，予測の成功率を向上させられることが期待できる．

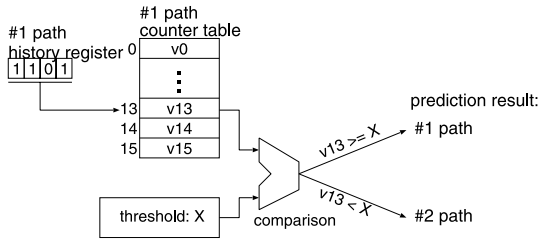


図 7 2 レベルパス予測器
Fig. 7 2-level path predictor.

たとえば，#1 パス用の履歴レジスタを用意しておく．1 イテレーション実行されるごとに 1 ビット左にシフトしたのち，最下位ビットには#1 パスが実行されたら ‘1’ を，実行されなかったら ‘0’ を記録する．こうすることで，履歴レジスタには#1 パスの実行履歴が記録される．

履歴レジスタ値はカウンタテーブルのインデックスとなる．カウンタテーブルの各エントリは，履歴レジスタの内容が当該エントリのインデックス値に対応するパターンになったとき，#1 パスが実行された回数を表している．

イテレーションの開始前に，履歴レジスタの値をもとにカウンタテーブルのエントリを読み出す．そこには，過去に同じ履歴レジスタ値だったときの#1 パスの実行回数が記録されている．この値をもとにして#1 パスを予測するか否かを決めればよい．この予測結果に従って 1 イテレーション分を投機実行する．その結果，予測どおりに正しく実行された場合は，当該カウンタテーブルエントリを 1 インクリメントする．逆に予測が外れた場合には 1 だけデクリメントする．カウンタのビット数は有限であるため，飽和カウンタを用いる．

このように 2 レベル分岐予測の手法をパス予測に適用した方法を 2 **レベルパス予測手法**と呼び，予測器を 2 レベルパス予測器と呼ぶことにする．

図 7 に具体的な 2 レベルパス予測器の構成を示す．この予測器は，#1 パス用の履歴レジスタとカウンタテーブルを用い履歴レジスタの値をインデックスとして参照されるカウンタの値と閾値との比較により，予測を行う方式である（図 7 参照）．閾値 X は，カウンタのビット数を n としたとき $2^{(n-1)}$ とする．この方

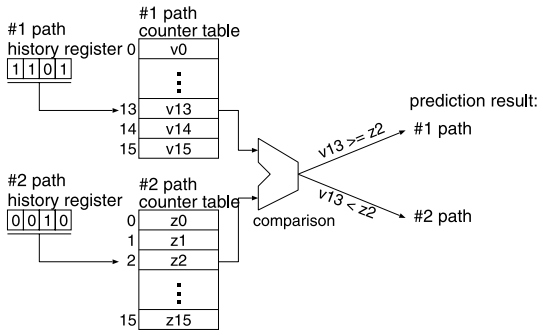


図 8 2 レベルパス予測器の発展型
Fig. 8 Modified 2-level path predictor.

式は，#1 パスの情報とそれ以外とを区別するのみの簡単な構造となっている．

これに対して，2 つのパスの各々に対して 2 レベルパス予測手法を適用し，各々のカウンタ値の比較をすることでパス予測を行う方法も考えられるが（**図 8** 参照），上記の予測器に比べてハードウェアコストが約 2 倍になるうえに，予測精度は大差ないことが明らかになっているため[7),8)]，本論文での議論の対象とはしない．

5. 2 レベルパス予測器を用いた性能評価

前節で述べた 2 レベルパス予測器を用いて実行パスを予測し，その結果に基づいて 3.2 節で提案した SS（Single Speculation）/DS（Dual Speculation）方式を用いて投機実行する場合の性能をシミュレータにより評価した．

5.1 評 価 方 法

評価は以下の方法により行った．

(1) 対象となるループの選定

2 章で述べたように，本論文の提案手法は基本的にループイテレーションを対象とする．このため，対象プログラムの中で実行時間の多くを占め高速化による効果の大きいループをマルチスレッド実行の対象として選定する．本論文で用いるのは，3.1 節および図 3 に示したプログラム，関数の中のループである．

(2) シミュレータでの実行およびトレースログの取得

対象プログラムを SimpleScalar ベースのシミュレータ SIMCA[9)] 上で実行し，(1) で求めたループ部分の実行トレースを取得する．この実行トレースから実行頻度の高い上位 2 つのパスを特定する．上位 2 つのパスを ‘P1’，‘P2’ とし，実行トレースをパスの出現シーケンス（パストレース）に変換する．ただし上位 2 つのパス以外は，基本ブロックで表現する．

```
(1)          lbu     $3, 0($17)
(2)          andi    $2, $3, 1
(3)          beq     $2, $0, L1
(4)          andi    $2, $3, 252
(5)          sb      $2, 0($17)
(6)    L1:   addiu   $17, $17, 12
(7)          addiu   $18, $18, 12
(8)          addiu   $19, $19, -1
(9)          bne     $19, $21, L2
             ...
       L2:   ...
```

図 9 PISA によるコードの例
Fig. 9 PISA code example.

(3) 非投機/投機スレッドコードの生成

シミュレーションで用いた対象プログラムの実行コードから，評価対象ループ部分のアセンブリコードを得る．このコードをもとにして，非投機スレッドコードおよび#1，#2 パスの投機スレッドコードを生成する．生成コードは，SimpleScalar の命令セット（PISA）をもとに，VLIW ライクに命令レベル並列性（ILP）を陽に指定できるようにした仮想的なものである[10)]．本評価では，同時に 4 つまでの演算を行えるものとし，最大限の ILP を引き出したコードを使用した．

非投機スレッドコードは，元のバイナリコードの構造を保っている．#1，#2 パスの投機スレッドコードは，2 章で述べたように，予測パス以外のコードをすべて削除し，条件分岐命令を対応する `assert` 命令に置き換え，さらに不要な命令を削除したうえで，VLIW 仮想コードに変換したものである．この非投機スレッドコードから基本ブロックごとの実行ステップ数を求めておき，投機スレッドコード実行の失敗後，非投機スレッドコードを実行する場合の実行ステップ数の算出に用いる．

さらに，#1，#2 パスの投機スレッドコードから，{#1，#2}×{#1，#2} の 4 通りのパスの組合せについての依存関係を検出する．また，投機スレッドコードに対しては，`assert` 命令の平均位置を求めておく．

たとえば，PISA 命令セットで記述された**図 9** のコードを考える．VLIW アーキテクチャを仮定し，横一列に並べた命令が 1 つの VLIW 命令としていっせいに実行されるものとし，元のコードを静的単一代入（SSA）形式に変換し命令レベルの並列度（ILP）を抽出してコードスケジューリングをする．条件分岐により実行パスが変化しても正しく動作するには，**図 10**

(1) lbu $3, 0($17)	(7) addiu $18, $18, 12		
(2) andi $2, $3, 1	(4) andi v1, $3, 252	(8) addiu $19, $19, -1	
(3) assert $2, $0, neq	(5) sb v1, 0($17)	(6) addiu $17, $17, 12	(9) assert $19, $21, neq

図 11 投機実行コードの例
Fig. 11 Speculation code example.

	(1) lbu $3, 0($17)	(7) addiu $18, $18, 12	
	(2) andi $2, $3, 1	(8) addiu $19, $19, -1	
	(3) beq $2, $0, L1	(4) andi v1, $3, 252	
	(5) sb v1, 0($17)		
L1:	(9) bne $19, $21, L2	(6) addiu $17, $17, 12	
	...		
L2:	...		

図 10 VLIW コード化の例
Fig. 10 VLIW code example.

のように 5 ステップが必要になる☆.

しかし，特定のパスのみを正しく実行できればよく，それ以外の場合には，条件分岐命令を改変した assert 命令によって投機実行を「失敗」させればよいものとすると，スケジューリングの自由度が増す．図 9 の例で (1)〜(9) の全命令が実行されるパスを仮定し，コードスケジューリングを行うと，**図 11** のように 3 ステップに圧縮される．なお，図 10 および図 11 中の v1 は，SSA 形式への変換の過程で変更されたレジスタ（仮想レジスタ）を表している．

(4) 提案方式による予測成功率・性能向上比の評価
(2) で求めたパストレースを入力とし，上で述べたパス予測方式および投機実行方式（SS/DS 方式）をシミュレートすることで，予測成功率と実行サイクル数を求める．

ここで，投機スレッドを実行するスレッドユニットの台数は 4 とし，投機失敗時にかかる回復処理の時間（3.3 節での t_r に相当）を 1 サイクルとした．また，投機成功時のスレッドの実行時間（t_s に相当）は (3) で求めた投機スレッドコードから，VLIW 実行時にかかるサイクル数を求めている．投機開始から失敗判明までの時間（t_f に相当）は，(3) で求めた assert 命令の平均位置を用いた．投機スレッドの開始遅延時間（t_g）は，(3) の過程で求めたスレッド間依存を解消できるだけのサイクル数とした．

3.3 節の評価とは異なり，t_s，t_g，t_f を一律とはせず，予測パス（t_s，t_f）およびその組合せ（t_g）により，VLIW 実行を仮定した場合の現実の値を用いている．評価に使用した各パラメータの値を**表 1**，**表 2** に示す．なお，#1，#2 以外のパスの実行時間については，複数のパスが存在する（compress, forward_DCT），出現しない（killtime），あるいは非常に頻度が低い（sweep），との理由により表示を省略する．

☆ 図 10 では ILP=2 として表記しているが，依存により (1) (2) (3) (5) (9) がクリティカルパスになっているため，ILP を増しても所要ステップ数に変化はない．

表 1 評価パラメータ（t_s, t_f, t_l）
Table 1 Evaluation parameters (t_s, t_f, t_l).

	#1 パス			#2 パス		
(function)	t_s	t_f	t_l	t_s	t_f	t_l
compress	8	5	17	13	7	24
forward_DCT	7	5	12	7	5	14
killtime	3	3	6	3	3	7
sweep	3	2	7	8	3	17

表 2 評価パラメータ（t_g）
Table 2 Evaluation parameters (t_g).

	投機スレッド開始遅延時間 (t_g)			
	from #1		from #2	
(function)	→#1	→#2	→#1	→#2
compress	8	8	13	11
forward_DCT	2	2	2	2
killtime	2	2	2	3
sweep	2	2	7	5

シミュレータは図 4，図 5 に示された動作を正確に模擬する．本提案の投機実行方式では，投機が成功している場合に複数のスレッドが重畳して実行される．起動したスレッドは，その実行パスがまだ確定していない状態（投機状態）にある場合でも，パスを予測して後続のスレッドを起動しなければならない．そのため，各スレッドユニットに履歴レジスタを持たせた．後続スレッドの起動の際，自スレッドの履歴レジスタの値を継承させる．後続スレッドは，前スレッドから継承された履歴レジスタをもとに予測器のカウンタテーブルをアクセスし，次のパスを予測する．カウンタテーブルの値は，スレッドの実行が確定したときに更新する．投機が成功した場合は，自スレッドの実行を終えるときに更新（カウンタをインクリメント）する．投機が失敗した場合は，非投機スレッドコードの実行を開始するときに更新（カウンタのデクリメント）を行う．先行スレッドの投機失敗によりスレッド実行がアボートされる場合には，カウンタテーブルの更新は行わない．

後続のスレッドを fork する際，fork 先のスレッドユニットが使用中だった場合は，当該ユニットが空くまで待つ．

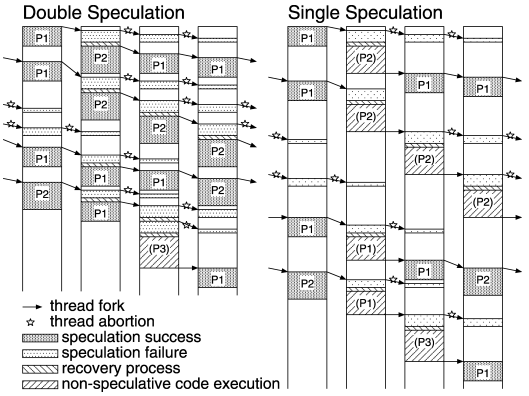


図 12 DS, SS 投機方式での実行の様子
Fig. 12 Execution examples of DS and SS methods.

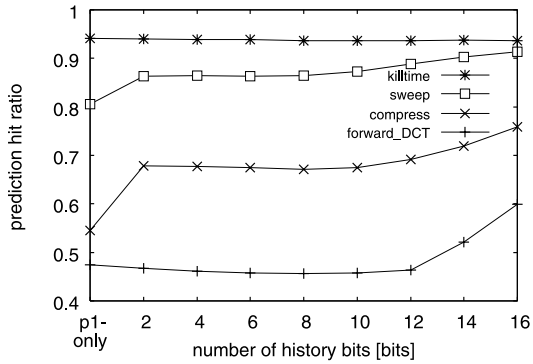


図 13 履歴長と予測成功率の関係
Fig. 13 History length and prediction hit ratio.

5.2 評価結果

5.2.1 実行の様子

SS，DS 投機方式を用い 4 台のスレッドユニット上で投機/非投機スレッドコードが実行されている様子を**図 12** に例示する．ここでスレッド実行時間等のパラメータやパスの出現パターンは，`forward_DCT` をもとに現実性を損なわない範囲で実行の様子が明確になるように改変したものを用いている☆．図中 `P1`，`P2`，`P3` と付された部分は，有効な処理を行っている箇所であり，それぞれ#1 パス，#2 パス，それら以外のパスを実行していることを表している．括弧内に示されているものは，非投機スレッドコードで実行されたパスを示している．空白部分はスレッドが割り当てられずにアイドル状態になっていることを示し，それ以外の部分は，実行中のスレッド，投機失敗スレッドとそれによりアボートされたスレッドを表している．スレッドユニットは左から右方向に環状に割り当てられる．

DS 方式によれば，最初の投機が失敗しても 2 回目の投機がヒットする確率が高いために，スレッドユニットが割り当てられずに効率が低下する事態を防げることが分かる．

5.2.2 予測成功率

図 13 は，2 レベルパス予測器において履歴レジスタのビット長を変化させたときの予測成功率を示している．比較のため，#1 パスの出現頻度を `p1only` としてプロットするとともに，比較を容易にするため他のプロットと結んでいる．

図 13 によると，`compress`, `forward_DCT`, `sweep` においては，履歴長 10 以下では，予測成功率に有意な差は見られない．しかし履歴長 10 以上では，履歴が長いほど，予測成功率が高くなる傾向にあることが分かる．これはループ中で実行されるパスに出現パターンがあり，履歴長が 10 以上になるとそのパターンが履歴として顕在化したものと考えられる．一方で `killtime` は，#1 パスのループ中の実行頻度が 97%となっているため，履歴の長さによらず予測成功率はあまり変わらない．

また，図 13 には陽に現れていないが，予測成功率の値に注目したい．我々が提案する投機方式では，自スレッドが投機状態にあるときでも後続のパスを予測・投機する連鎖が生じる．つまり，前段での予測結果の上に，さらに予測を重ねることになるため，予測器の性能が低下することが考えられる．しかし実際には，#1 パスの出現頻度と同等（`killtime`, `forward_DCT`），もしくはそれを上回る成功率（`compress`, `sweep`）が得られている．本提案方式によりパスを限定したことで，多重投機による予測器の性能低下が抑えられたものと推測されるが，この点については今後検証を考えたい．

本論文で評価に用いた 2 パス予測器のヒット率の時間変化を調べた．ヒット率は一定時間（10,000 クロック）のウインドウを単位として計測した．ウインドウ期間内のヒット数，ミス数をもとにウインドウでのヒット率を求めている．シミュレーション開始時から 1,000,000 クロックまでの結果を**図 14** に示す．`compress`，`sweep` では，シミュレーション開始後，時間とともにヒット率が向上していることが分かる．

5.2.3 性能向上比

4 章に示した 2 レベルパス予測器と投機実行方式（SS，DS）とを組み合わせて，各ホットループでの非投機実行時に対する速度向上率を評価した．この評価により，前項で示した 2 レベルパス予測器の性能がプ

☆ 改変の内容は，t_s, t_g, t_f の値に差を持たせたこと，非投機スレッドの実行時間を短縮したこと，実行開始冒頭で出現していた#1，#2 以外のパスを削除したこと，である．

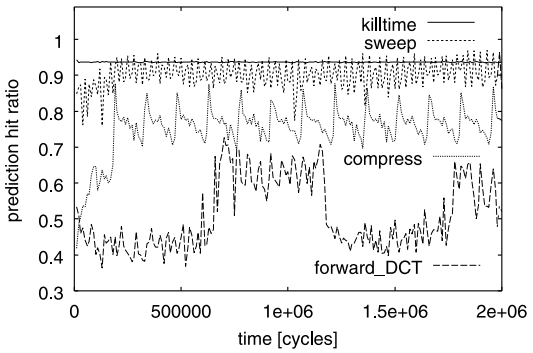


図 14 2 レベルパス予測器のヒット率の時間変化 (SS 方式)
Fig. 14 Changes of hit ratio of 2-level path predictor.

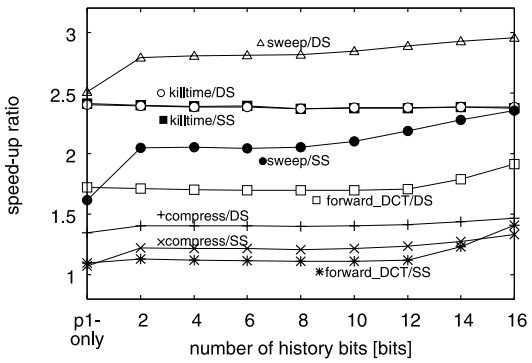


図 15 履歴長と速度向上率の関係
Fig. 15 History length and speed-up ratio.

表 3 DS 方式での性能向上比の比較
Table 3 Performance improvement ratios in DS method.

(function)	2 パス予測器 (履歴長 12)	100% 予測器
`compress`	1.416	1.739
`forward_DCT`	1.707	3.419
`killtime`	2.378	2.836
`sweep`	2.886	3.284

ログラムの実行速度に及ぼす影響を確認するとともに，DS 投機方式が SS よりも優れていることを検証することを目的としている．結果を**図 15** に示す．ここでは，5.1 節 (3) で作成した非投機 VLIW コードにより逐次的に（マルチスレッドでなく）実行した場合のサイクル数を基準としている．横軸が履歴レジスタのビット長を示し，縦軸が速度向上比を示している．図 13 と同様につねに#1 パスのみを予測する場合（`p1only`）もあわせて表示している．

プログラム全体に対する実行時間の比は，`compress` が 47.6%，`forward_DCT` が 15.6%，`killtime` が 7.1%，`sweep` が 6.7%である．図 15 の結果から，たとえば，DS 方式で履歴長 12 ビットの場合，プログラム全体の速度向上比は，`compress` で 1.163 倍，`forward_DCT` で 1.069 倍，`killtime` で 1.043 倍，`sweep` で 1.046 倍となる．

また，プログラムが#1，#2 パスを実行した場合に全投機が成功する（#1，#2 パス以外の場合は投機失敗する）ように改変したシミュレータにより，本提案が最もよく働く場合の性能向上比を求めた．比較のため履歴長 12 の 2 パス予測器を用いた場合とともに，結果を**表 3** に示す．

図 15 から，`killtime` を除いて，DS 投機実行方式を適用することで SS よりも高い性能が得られていることが分かる．これは 3.2，3.3 節で検討した結果と符合する．すなわち，上位 2 つのパスが実行頻度の大半を占めるのであるから，DS 方式により最初の投機に失敗しても 2 番目の投機が成功する確率は高い．このために，DS 方式は SS に比べ高い実行性能を得ている．これは DS 投機実行方式が実際のプログラムにおいて有効であることを示している．

このことは，3.3 節の議論により導出された式 (3) から，ある程度見積もることができる．粗い近似として，表 1，表 2 に示した#1 パスの評価パラメータと図 3 に示している各パスの出現頻度を式 (3) に適用すると，`compress` では $p_2 > 0.182$，`forward_DCT` では $p_2 > 0.19425$，`killtime` では $p_2 > 0.015$，`sweep` では $p_2 > 0.072375$ であり，いずれも式 (3) の条件を満たしている．

また，`compress` や `sweep` では，`p1only` に比べ明らかに高い速度向上比を達成している．`forward_DCT` および `killtime` については，際立った違いは見られないが，`forward_DCT` については履歴長 10 以上で `p1only` をやや上回っている．このことから本論文で用いた 2 レベル予測器が有効に働いていることが分かる．

6. 関 連 研 究

パスをベースとした制御投機の考えは，トレースキャッシュ[11] から発展させたトレースプロセッサ[12] に見ることができる．ここでパスとトレースは同義のものとして扱われている．トレースプロセッサでは，過去の履歴をもとに適切なトレースを選び投機実行する方式をとっており，パスベースの予測器[13] を提案しているほか，この考えを発展させた trace preconstruction[14] も提案している．この予測器では，投機対象として選択の対象とするトレースの数がハードウェアコストの要因となると考えられるが，最適な対象個数を求める議論はなされていない．

また PARROT[15] では，電力消費の制約下で最適性能を得るためのトレースを選択する手法を提案している．これも前者と同様，予測および投機の適切な対象個数を議論したものではない．

本論文で扱ったパスベースの投機的マルチスレッド処理方式に類似した手法として，複数パス投機実行がある[16)~20)]．これらは，分岐予測ミスの結果がパイプライン実行のストールなど性能に悪影響を及ぼすことを避ける目的で，実行の可能性のある複数のパスを同時に投機的に実行するものである．投機実行されたパスのうち成功したものを残すことで分岐予測ミスの影響が隠蔽されることが期待される．こうした複数パス投機実行は，本論文で前提としているマルチスレッドモデル，すなわち，スレッド単位での重畳実行により高速化を図る方法とは根本的に異なるものである．

ただし，パスを投機実行の対象にしている点で，本論文で扱っているパスベース投機的マルチスレッド処理と類似している．さらに文献 16)，17)，20) では 2 つのパスに限って投機実行しており，本論文に近い考えといえよう．さらに Tyson らは，分岐の taken/not-taken の出現シーケンス（履歴）から，実行中に出現するパスを，taken が多く続く場合，not-taken が続く場合，連続する taken の中に少数の not-taken が混じる場合，その他の場合，の 4 パターンに類型化できることを示し，パス予測に役立てている[17)]．条件分岐の taken/not-taken の出現パターンは，パスに相当するから，上記の手法はすなわち 4 種類のパスを対象としていることと等価である．また上記手法では taken が多く続く場合，not-taken が続く場合を頻出パスとして示しているが，一般のプログラムで実際の最頻出パスがそうなっている（taken/not-taken が連続する）保証はない．上位 2 個の頻出パスだけに限定する本論文の提案手法が合理的である．

7. おわりに

本論文では，頻繁に繰り返し実行されるホットループに対して，各イテレーションで実行される処理の内容，すなわちプログラムの実行経路（パス）を予測して投機実行を行う，パスベースの投機的マルチスレッド処理モデルを対象として，現実的な投機実行手法を論じた．

ループ中の処理に複数の条件分岐を含み，実行可能性のあるパスが複数あるケースは多いが，実際のアプリケーションでは，多くの場合，たかだか 1 ないし 2 個のパスで実行時間の多くを占める．本論文はこれに着目し，投機対象を実行頻度の上位 2 つのパスに限定することが現実的に必要十分であることを示し，このことに基づいた 2 パス限定投機方式を提案した．出現頻度の高い順に選んだ 2 つのパスのいずれかを予測し投機実行する．予測が外れた場合に非投機コードの実行を行う Single Speculation（SS）方式と，他方のパスを投機実行する Double Speculation（DS）方式を示し，簡略化モデルによる解析とシミュレーションの評価により，特に後者が有効であることを示した．

本論文では，提案投機実行方式の有効性を確認するにあたり，2 レベル分岐予測器をパス予測に適用した簡単な予測器（2 レベルパス予測器）を仮定した．投機的マルチスレッド処理では，多重投機状態が発生し予測成功率が低下することが予想されるが，本提案方式では評価の結果，最頻出パスの出現頻度と同等ないしそれを上回る予測成功率が確認できた．これが本提案方式でパスを 2 つに限定したことに起因するか否かについては，今後検証を行いたい．

謝辞 本研究は，一部日本学術振興会科学研究費補助金（基盤研究（B）14380135，同（C）16500023，若手研究（B）17700047）の援助による．

参考文献

1) 大津金光，野中雄一，横田隆史，馬場敬信：実行時最適化に向けたソフトウェア・パスプロファイリング手法の検討，電子情報通信学会論文誌（D-I），Vol.J88-D-I, No.5, pp.985–990 (2005).
2) Ball, T. and Larus, J.R.: Efficient Path Profiling, *Proc. 29th Ann. IEEE/ACM Int. Symp. Microarchitecture* (*MICRO-29*), pp.46–57 (1996).
3) Ball, T. and Larus, J.R.: Programs Follow Paths, Technical Report MSR-TR-99-01, Microsoft (1999).
4) Ball, T. and Larus, J.R.: Using Paths to Measure, Explain and Enhance Program Behavior, *Computer*, Vol.33, No.7, pp.57–65 (2000).
5) 増保智久，道口貴史，斎藤盛幸，大津金光，横田隆史，馬場敬信：MediaBench ホットループ並列化のためのパスプロファイリング，情報処理学会第 67 回全国大会講演論文集，pp.1-203–1-204 (2005).
6) Yeh, T.-Y. and Patt, Y.N.: Two-Level Adaptive Branch Prediction, *Proc. 24th ACM/IEEE Int. Symp. on Microarchitecture* (*MICRO24*), pp. 51–61 (1991).
7) 斎藤盛幸，古川文人，大津金光，横田隆史，馬場敬信：投機的マルチスレッド実行のための限定的 2 パス予測方式の検討，情報処理学会研究報告，Vol.004, No.48, pp.7–12 (2004).
8) Yokota, T., Saito, M., Furukawa, F., Ootsu, K. and Baba, T.: Prediction and Execu-

tion Methods of Frequently Executed Two Paths for Speculative Multithreading, *Proc. 16th IASTED Int. Conf. on Parallel and Distributed Computing and Systems* (*PDCS 2004*), pp.796–801 (2004).
9) Huang, J.: The SImulator for Multi-Thread Computer Architecture. http://www-mount.ee.umm.edu/~lilja/SIMCA/index.html
10) 月川 淳，古川文人，大津金光，横田隆史，馬場敬信：メタレベル最適化計算機システムYAWARA のシミュレーション環境—PISA をベースとしたVLIW アセンブラの開発，情報処理学会第 67 回全国大会講演論文集，pp.1-69–1-70 (2005).
11) Rotenberg, E., Bennett, S. and Smith, J.E.: Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching, *Proc. 29th Ann. IEEE/ACM Int. Symp. on Microarchitecture* (*MICRO-29)*, pp. 24–34 (1996).
12) Rotenberg, E., Jacobson, Q., Sazeides, Y. and Smith, J.: Trace Processors, *Proc. 30th Ann. Int. Symp. on Microarchitecture* (*Micro '97*), pp.138–148 (1997).
13) Jacobson, Q., Rotenberg, E. and Smith, J.E.: Path-Based Next Trace Prediction, *Proc. 30th Ann. Int. Symp. on Microarchitecture* (*Micro '97*), pp.14–22 (1997).
14) Jacobson, Q. and Smith, J.E.: Trace Preconstruction, *Proc. 27th Ann. Int. Symp. on Computer Architecture* (*ISCA '00*), pp.37–46 (2000).
15) Rosner, R., Almog, Y., Moffie, M., Schwartz, N. and Mendelson, A.: Power Awareness through Selective Dynamically Optimized Traces, *Proc. 31st Ann. Int. Symposium on Computer Architecture* (*ISCA'04*), pp.162–173 (2004).
16) Heil, T.H. and Smith, J.E.: Selective Dual Path Execution, Technical report, Dept. Electrical and Computer Engineering, University of Wisconsin-Madison (1996).
17) Tyson, G., Lick, K. and Farrens, M.: Limited Dual Path Execution, Technical report, CSE-TR-346-97, University of Michigan (1997).
18) Wallace, S., Calder, B. and Tullsen, D.M.: Threaded Multiple Path Execution, *Proc. 25th Ann. Int. Symp. on Computer Architecture* (*ISCA'98*), pp.238–249 (1998).
19) 高木将通，平木 敬：高度パイプライン化 SMT プロセッサ上のトレースに基づく複数パス実行方式，信学技報，Vol.101, No.2, pp.73–80 (2001).
20) 片山清和，安藤秀樹，島田俊夫：両パス実行の性能評価と実行判定精度の改善，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.42, No.SIG 9 (HPS 3), pp.106–118 (2001).

(平成 17 年 4 月 28 日受付)
(平成 17 年 8 月 5 日採録)

横田 隆史（正会員）

1983 年慶應義塾大学工学部電気工学科卒業．1985 年同大学院電気工学専攻修士課程修了．同年三菱電機（株）に入社，中央研究所，先端技術総合研究所，産業システム研究所に所属．主席研究員．1993 年 12 月から 1997 年 3 月まで新情報処理開発機構（RWCP）に出向．2001 年 4 月より宇都宮大学工学部助教授．計算機アーキテクチャ，設計方法論等の研究に従事．工学博士．ICCD Outstanding Paper Award（1995），FPGA/PLD Design Conference 審査委員特別賞（2002）各受賞．電子情報通信学会，IEEE 各会員．

斎藤 盛幸

2003 年宇都宮大学工学部情報工学科卒業．2005 年宇都宮大学大学院工学研究科情報工学専攻修了．同年富士通 SCM システムズ株式会社に入社．

大津 金光（正会員）

1993 年東京大学理学部情報科学科卒業．1995 年東京大学大学院修士課程修了．1997 年東京大学大学院博士課程退学，同年より宇都宮大学工学部助手となり現在に至る．計算機システムの高性能化に関すること，特にマルチスレッドアーキテクチャ，バイナリ変換処理，実行時最適化等に興味を持つ．

古川 文人（正会員）

1998年宇都宮大学工学部情報工学科卒業．2000年宇都宮大学大学院博士前期課程修了．2003年同大学院博士後期課程修了．同年4月より宇都宮大学ベンチャー・ビジネス・ラボラトリー非常勤研究員．2005年4月より帝京大学ラーニングテクノロジー開発室助手．博士（工学）．高性能計算機システム，授業改善のためのラーニングテクノロジに関する研究に従事．

馬場 敬信（正会員）

1970年京都大学工学部数理工学科卒業．1975年京都大学大学院博士課程単位取得退学．同年より電気通信大学助手，講師を経て，現在宇都宮大学工学部教授．工学博士．1982年より1年間メリーランド大学客員教授．計算機アーキテクチャ，並列処理等の研究に従事．1992年情報処理学会Best Author賞，2002年FPGA/PLD Design Conference審査委員特別賞，PDCS2002国際会議Best Paper Award各受賞．著書“Microprogrammable Parallel Computer”（MIT Press），『コンピュータアーキテクチャ（改定2版）』（オーム社），『コンピュータのしくみを理解するための10章』（技術評論社）等．電子情報通信学会，IEEE各会員．