社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－ARC－159 (2)
2004／7／30

命令ウインドウ拡張による命令レベル並列性の利用

石井康雄† 平木敬†

大きいスケジューリングウインドウはILPを利用するプロセッサで性能の向上に貢献している。しかし、CAM構造の大きなスケジューリングウインドウは高いクロックを実現する妨げとなる。そこで、本稿では新しい拡張性のある階層化スケジューリングアルゴリズムを提案した。これはList構造を利用したWakeupアルゴリズムを利用している。その評価をシミュレータを用いてSPEC CPU2000の一部に対して行った。その結果、従来の階層化スケジューリングアルゴリズムに対して10.4%の性能の向上が得られた。

Extending Instruction Window for Exploiting ILP

Yasuo Ishii† and Kei Hiraki†

Large scheduling window is an effective mechanism for ILP processor performance. But large scheduling window implements based on CAM structure decrease clock frequency. So, we proposed a scalable scheduling window that is based on hierarchical scheduling algorithm. It uses a sequential wakeup algorithm based on a list structure. We evaluated it by processor simulations. Simulation Results shows that our algorithm achives 10.4 % speedups for a subset of the SPEC CPU2000.

1. はじめに

現代のプロセッサにおいては高いクロックとクロックあたりの命令実行数 (IPC) を実現することにより高いパフォーマンスを達成している。命令レベル並列性 (ILP) を効果的に利用するためにプロセッサではスケジューリングウインドウというバッファを利用している。この中で命令は依存関係を解決し発行できるようになった命令から演算器へ Out-of-Order に発行される。特に、スケジューリングウインドウの大きさは命令のレイテンシを隠蔽する重要な要素となっており、メインメモリへのアクセスレイテンシが増大している現代では大きなスケジューリングウインドウは高いIPCを実現するために重要な要素となっている[1]。しかし、スケジューリングウインドウはCAMを利用したWakeup-Select回路を利用しているためその大きさに拡張性がなく、大きなスケジューリングウインドウはクリティカルパスに入るためクロックを制限することが知られている[2]。そのため、クロックを損なうことなくILPを利用するための新しい命令スケジューリングアルゴリズムが必要となっている。

これを実現するためにCAM構造をRAMの参照を用いることにより回路の複雑さを軽減できるEDF方式[3]が提案されている。HSW方式[4]ではEDF方式を利用した拡張性のあるスケジューリングウインドウとCAM構造のスケジューリングウインドウを階層化して高いIPCを実現している。本稿ではSequential Wakeup Algorithmという新しいスケジューリングアルゴリズムを提案する。この手法は階層化アルゴリズムで、HSW方式と近い方式である。大きいウインドウではList構造で依存関係を管理し、RAM参照による依存関係の解消を実現することにより従来のBroadcastを利用したスケジューリング方式と比較して高い拡張性を実現する。クリティカルパスに入った命令に関しては従来手法のWakeup-Select方式のスケジューリングウインドウで処理することにより効率的にILPを利用する。Wakeup-Select回路は小さいものであれば従来どおりのアルゴリズムでもクロックへの影響を及ぼさずに実装できる[5][6]。従って、高いクロックを維持したままでILPを効率よく利用することができる。

本稿での章別の構成を述べる。2節ではメモリ参照を利用したアルゴリズムであるEDF方式を検討しその欠点を明確化する。3節ではその欠点を解決するための新しいスケジューリングアルゴリズムを提案する。4節でシミュレーションにより提案手法の評価を行い5節で全体をまとめるという構成になっている。

† 東京大学
Tokyo University

2. Explicit Data Forwarding(EDF)

従来の Wakeup-Select を基にしたスケジューリング方式は依存命令が被依存命令を見張ることにより被依存命令の終了を検知して自分の入力オペランドが利用可能であることを知るというアルゴリズムを用いる (図 1(a))。このアルゴリズムでは被依存命令の Broadcast が必要不可欠になり、そのためにスケジューリングウインドウは CAM 構造をとる必要がある。それに対して佐藤らが提案する EDF 方式[3] は被依存命令から依存命令へのポインタを張ることにより明示的に Wakeup を行うようにしたアルゴリズムである。このときには Wakeup が CAM ではなく RAM の参照で実現することができる。従って、従来と比較して高い拡張性が得られる。この EDF 方式では被依存命令が依存命令の Limited Vector で管理して依存情報を持つことによりその Vector 内の命令を明示的に Wakeup させることができる。しかし、この有限な Vector に登録できなかった命令に関しては命令ウインドウの先頭に来るまで実行できないという欠点がある。

この Wakeup が伝わらなかった命令はスケジューリングウインドウ中で先頭に来るまで依存関係の解決ができない。1 つの命令が Wakeup されないとその後続命令も全て実行不可能となる。このペナルティはスケジューリングウインドウの大きさに比例する。このような場合にはスケジューリングウインドウを大きくしてもウインドウサイズを大きくした効果が得られない。EDF 方式を利用した階層化アルゴリズムである HSW 方式[4] でも同様のことがいえる。つまり Limited Vector に挿入できなかった命令による性能低下が避けられない。

例えば図 1(a) での点線は Limited Vector の大きさが 2 であったときに 3 つ目の依存命令に Wakeup が伝わらないことを示している。このときには被依存命令である add が終了すると mul と and に関しては即座に Wakeup をする。しかし、ld に関しては本来ならば Wakeup できるのだが命令ウインドウの先頭に来るまで実行を行うことが不可能になる。

次章以降ではこの欠点を解決するために List 構造で依存関係を管理する逐次型の Wakeup アルゴリズムとその実装方式を提案する。

3. Sequential Wakeup Algorithm

この章では提案手法である Sequential Wakeup Algorithm(SWA) に関して説明する。従来のスケジューリングアルゴリズムは図 1(a) に示すように依存命令を同時に Wakeup させるため拡張性のない CAM や Limited Vector の利用を余儀なくさせている。

従来方式の問題点は図 1(a) のように Broadcast を

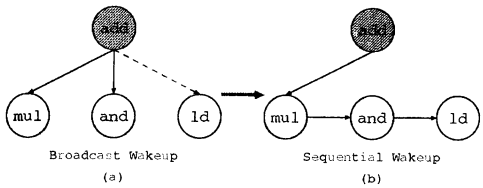

図 1 従来の Wakeup と List 型 Wakeup

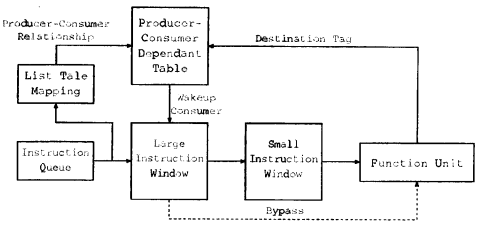

図 2 Sequential Wakeup Algorithm の概要

利用して依存命令を同時に Wakeup させる点である。提案手法ではこの問題を解決するため図 1(b) に示すように依存関係を List 構造で管理する。この List 構造を追跡する順番で逐次に Wakeup を行うようにすれば Limited Vector 型の欠点を解決することができる。

3.1 概　　要

この手法の構造を図 2 に示す。SWA は 4 つの主要な部品から構成される。その 4 つの部品は小さく速い CAM 構造のスケジューリングウインドウ、RAM の参照を用いた大きいスケジューリングウインドウ、依存関係を管理する依存関係表、そして、管理している List の最後尾へのポインタを保持する List Tale Table である。

命令スケジューリングの手順の説明をする。発行される命令はスケジューリングウインドウのエントリを確保する。このとき確保した大きいスケジューリングウインドウのエントリの Index が以降での命令の依存関係 List の確保解放の単位となり、これを命令の ID として扱う。依存関係をうけとった後に取得した依存関係を元に List Tale Table を引いて依存関係表に対して登録して大きいウインドウに挿入される。大きいウインドウ内で依存関係が解決したものは即座に実行ユニットへ発行される。依存関係が未解決な命令に関してはその命令がウインドウ内でもっとも古い命令となったときに小さいスケジューリングウインドウに移動させる。小さいウインドウ内では従来型の CAM 構造方式で依存関係を解決して実行する。実行が終わった命令は依存命令を Wakeup するためその終了を依存関係表に知らせる。依存関係表内でその命令の依存関係を調べる。依存命令を Wakeup させるために大きいウインドウに対して更新情報を知らせる。

以下で各部品の詳細に関して説明する。

3.2 Small Scheduling Window

この小さいウインドウは従来どおりの CAM 方式のアルゴリズムを利用している。各命令が実行可能であるかどうかを動的にチェックして Wakeup した命令から順番に発行していく。実行が終わった命令はこのウインドウ内に自分の実行終了を伝える。この際に CAM 構造を利用しているためこのウインドウ内の全ての命令に対して即座に命令の実行終了が伝えられる。この命令の依存命令はこのときに自分の入力オペランドが準備できたことを確認するのでそれに従い必要に応じて Wakeup する。さらに、小さいウインドウでの発行結果は同時に大きいウインドウにも通知されてそちらの依存命令の発行も続行させる。

3.3 Large Scheduling Window

大きいウインドウは RAM の参照を利用して実現され、EDF 方式のスケジューリングウインドウと似た構成をしている。すなわち、左右に関してそのオペランドが準備できたかのビットがある。これに対してオペランドが準備できた場合 RAM の参照形式でアクセスすることによりそのビットをセットすることができる。全てのオペランドが準備でき次第発行待ちのキューに命令を移動させる。古い命令に関しては小さいウインドウに移動させる。しかし、命令が小さいウインドウに移動される、あるいは、実行ユニットに発行されてもその命令が Commit するまでは各命令はこのウインドウのエントリを保持し続ける。つまり、このウインドウはリオーダバッファのエントリを確保するときに同時に確保されリオーダバッファのエントリ解放時まで保持し続けるということになる。

命令のウインドウ間の移動の説明をする。大きいウインドウには「古い実行不可能な命令」と「新しい実行可能な命令」が存在する。前者に関してはクリティカルパスに入っている命令なので最も古いものから小さいウインドウに命令を移動して依存関係が解消され次第実行するようにする。このウインドウのエントリはサイクリックに割り当てられる。従って、どの命令が一番古いかをポインタの 1 次元の探索で調べることができる。この構造により最も古い命令を大きいウインドウ内で容易に特定することができる。このウインドウ内の実行可能な命令に関してはクリティカルパスに入っていない命令なので演算器に空きがあった場合に順次発行していく。

3.4 依存関係表

依存関係表は List 構造の依存関係を保持する。この提案手法において Wakeup を Sequential に行うためのもっとも重要な部品である。その構成は 2 つの表からなる。その 1 つが実行を受け取って List 構造の先頭の命令を返すもの (Producer Table)、もうひとつが List の依存関係を保持するもの (Consumer Table) である。**図 3** にその構造を示す。

この部品は命令の実行が終了した後に明示的に依存

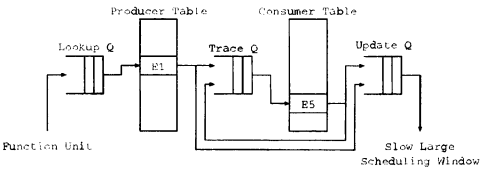

図 3 依存関係表の内部構造

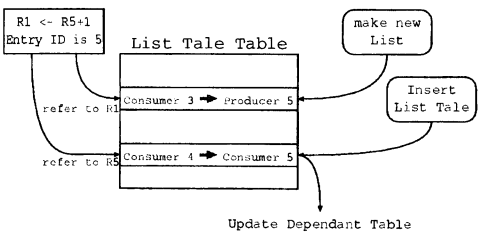

図 4 List Tale Table

命令を Wakeup させる目的で使用される。実行が終了した全ての命令は大きいウインドウでのエントリ ID をこの依存関係表につたえてくる。依存関係表の内部で依存関係 List を探索するためこのエントリ ID で Producer Table を引く。そのときに Producer Table の該当エントリに書いてあった ID は実行終了命令の依存命令である。そこで、この ID を大きいウインドウに渡して依存命令の Wakeup を行う。同時に同じデータを Consumer Table にも引渡しその Consumer 命令間での Wakeup を行う。Consumer Table 内での List の追跡は引いた Consumer Table のエントリが有効である間行われ続ける。Consumer Table 内の要素が List の次の要素のポインタであると同時に Wakeup すべきエントリ ID となっているのでそれを逐次大きいウインドウの更新に利用しつつ依存 List を追跡することができる。

今回はこの Consumer Table を負荷分散するため左右の入力オペランドごとに管理するものとした。従って、Producer Table は左右入力オペランドの List に対する先頭となるから大きいウインドウの各エントリに対して 2 つずつのエントリを持つことになる。この List を左右のオペランドで分割することによりこの依存関係 List を用意に分割することができるため各依存関係表が大きくなることによるアクセス時間の増大を抑えることができる。

この依存関係 List の作成に関しては次の List Tale Table で詳しく説明する。

3.5 List Tale Table

List Tale Table は各論理レジスタが依存関係表のどのエントリに List 構造の末尾があるかを示していて、依存関係を List に登録する役割をもっている。このウインドウのエントリは**図 4** のようになっている。この List Tale Table は依存関係 List の最後尾の命令

のIDを記憶している。つまり依存関係表のどのエントリに新しい命令のIDを登録すればよいかを保持している。

大きいウインドウでは命令は自分のProducerまたは先に登録されたConsumer命令からその入力オペランドが使用可能であるかの情報を受け取る構造になっている。従って各依存命令は被依存命令に対して直接のProducer-Consumer関係を築くことは必ずしもできない。もちろん依存関係の管理が左右のオペランドで別に行われるのでこのList Tale Tableの割り当ても左右別に行われる。

Consumer TableにIDを割り当てることになったときにはListの最後尾は自分となる。その場合はListの最後尾であるからList Tale Tableを引き、そのエントリにその命令のIDを登録するという2つの動作を同時に実行する。

つまり図4を例にとると、ある論理レジスタR5がList Tale Tableを調べに行ったとする。そのときに今までのListの最後尾であったConsumer Tableエントリ4を受け取ると同時にその命令に割り当てられていたエントリIDであるConsumer Tableエントリ5をそのマッピングに登録する。これにより論理レジスタR5の依存命令Listの最後尾にConsumer Tableエントリ5が付け加えられるのである。また、命令の書き込み先(図4ではR1)に関しては後続の命令がそれ以降では自分をListのTaleとする必要がある。そのため自分の書き込み先のProducer Listのエントリである Producer Tableエントリ5をこの表に書き込む。

4. 評　　価

提案手法をSimpleScalarシミュレータを用いて既存の階層化スケジューリングウインドウのアルゴリズムと比較をした。評価に用いたベンチマークはSPEC CPU2000の中からgccなど12個を選択した。

4.1 評価環境

シミュレータのパラメータは**表1**のとおり。SWAの構成としては各Tableに関してRead/Writeポートを8つずつ、大きいウインドウに関してはWriteポート8つReadポートを4つとした。各キューに関してはエントリ数は1024とした。各種テーブルに関してはアクセスレイテンシ4でアクセスができるものとした。キューのエントリ数がスケジューリングウインドウサイズと同一のエントリ数を持つ場合オーバーフローすることはありえない。しかし、もっとキューのエントリが少ないときにはオーバーフローが発生することもありうる。オーバーフローした情報に関しては捨ててしまっても命令は最終的に小さいウインドウに挿入されて実行されるので実行の正しさとしては問題がない。

表1　シミュレータ構成
Table 1　Simulator Configuration

ISA	Alpha 21264[7]
Instruction Window Size	1024
ROB size	1024
LSQ size	1024
Register File Size	1024
Issue Width	8 Way
Execution Core	8 ALU, 4 FPU
L1 Data Cache	32KB : 8Way
L1 Inst Cache	32KB : 4Way
L1 Access Latency	3 Cycle
L2 Cache	1MB : 4Way
L2 Access Latency	20 Cycle
Memory Latency	800 Cycle
Branch Prediction	64K entry Gshare
Mispredict Penalty	20 Cycle

比較対象として従来のLimited Vectorを利用した階層型スケジューリングウインドウの評価を行った。SWAとほぼ同程度のハードウェア量を使用する4エントリのLimited Vector型階層型スケジューリングウインドウとその倍の資源が必要となる8エントリのものに関して評価した。また、Limited Vectorがいっぱいになるまでは従来どおりのLimited Vectorを利用した明示的なWakeupを行い、Limited Vectorがあふれた際にList構造のWakeupを利用するというハイブリッド型も評価の対象として加えた。ハイブリッド型の構成はLimited Vectorが4エントリの構成にList型の構造を加えた形とする。これはList Tale Tableに2ビットのカウンタを併設して参照数をカウントすることにより実現が可能となる。なお、この構成では8エントリのLimited Vector型スケジューリングウインドウとほぼ同程度のハードウェア資源を必要である。

4.2 評価結果

シミュレーションから得られたIPCを従来どおりのスケジューリング方式で実行を行ったときの性能で正規化した結果を**図5**に示す。グラフ中のそれぞれのバーは左から小さいウインドウのみで実装された場合、従来どおりのLimited Vector型でエントリ数が4の場合、提案手法であるSWA、エントリ数が8の場合のLimited Vector方式、ハイブリッド型、Wakeup-Select方式となっている。Wakeup-Select方式は比較のための結果であり、実現することは困難である。

結果としてはSPEC CPU2000全体で10.4%IPCが向上することがわかった。SPEC CPU2000INTではcraftyで1.9%で平均で0.9%と小さい性能変化にとどまったがSPEC CPU2000FPではmgridで最大68.2%の性能向上を示して平均でも18.5%の性能の上昇が見られた。個々のベンチマークに関してみてみるとlucasでは大きさ4のLimited Vectorがそのほかの方式と比較して大変低い性能を示している。しかし、

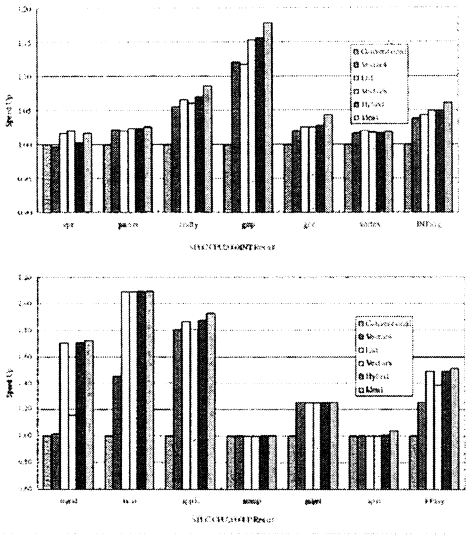

図 5 SPEC CPU2000 シミュレーション結果

8 エントリではそのほかの方式とほぼ同等の性能を示しているのでクリティカルパス上の依存関係の List の大きさが 5 から 8 程度であったであると考えられる。そのため、クリティカルパスに入っている命令が大きい命令ウインドウの先頭に来るまで実行が不可能となってしまうため性能が低下していると考えられる。つまり、このときの 8 エントリと 4 エントリの差は依存関係 List に加えられないために受けたペナルティであると考えることができる。mgrid に関しては 8 エントリのときも 4 エントリのとほぼ同等のところまで性能が低下している。従って、依存関係に加えられないペナルティをこちらでも受けていると考えることができる。つまり、プログラム依存ではあるが従来の階層型スケジューリングウインドウ方式はウインドウサイズが広がったときに大きいペナルティを受ける可能性がある。しかし、List 型で管理している場合には Wakeup-Select 方式から大きな差を受けることはなくなる。これは Limited Vector 型で受けたペナルティを回避できているためと考えられる。

CPU2000INT に関しては従来の階層型と性能的にほぼ同一の結果が出ている。しかし、全てのベンチマークにおいて理想状態 (グラフ中では Ideal) から平均で 0.9%、最大でも gzip の 4.9%しか性能低下が見られない。よって、もともと十分に性能が出ていたと考えることができる。

gzip などでは SWA が Limited Vector 型の性能を下回っている。これは SWA では Wakeup のレイテンシが List を追跡するために大きくなったためである。これはレイテンシの増大の影響を受けるのがクリティカルパスに入らない命令であるため大きな性能低下にはつながらない。実際、性能低下は SPEC CPU2000INT では高々0.2%とわずかである。

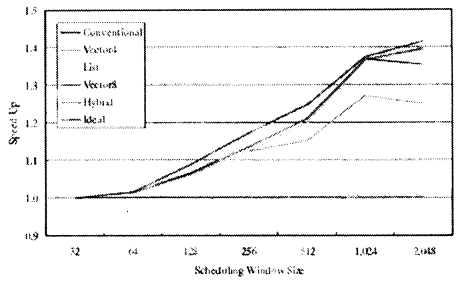

図 6 ウインドウサイズを変化させた際の性能変化

次に提案手法のスケジューリングウインドウがどの程度の大きさを実現したときに有効となるかを検証する。そのためにスケジューリングウインドウサイズを変化させた際の IPC の従来手法との性能比率を図 6 に示した。図 6 での List 方式と Vector 方式における性能の変化に注目してみる。ウインドウの大きさが 256 エントリほどまでは Vector 型の方が 4 命令を同時に発行できるため性能で List 型よりも高い性能を示している。しかし、1024 エントリ以上では Wakeup できない命令のペナルティが大きくなるため List 型のほうが高い性能を示している。一般に Wakeup-Select 回路をパイプライン化するなどしてレイテンシを増大させることは IPC の低下を招く[8]。しかし、大きいスケジューリングウインドウを持つプロセッサにおいてのクリティカルパスに入らない命令の実行に関しては Wakeup-Select のレイテンシが大きくなっても性能に大きな影響を与えないといえる。したがって、大きいスケジューリングウインドウでは Wakeup までのレイテンシが増大したとしても全ての Wakeup 可能な命令をきちんと Wakeup させることが重要であるといえる。そのための実現方法として逐次型の Wakeup アルゴリズムは非常に有力である。

5. 関連研究

大きなスケジューリングウインドウの実現に関しては EDF 方式のほかにも多くの先行研究が存在する。

階層化スケジューリングアルゴリズムとしては Data Flow 予測に基づいて命令の発行タイミングを予測するというスケジューリング方法がある[9]。この方法では FIFO 構造のスケジューリングウインドウに動的に発行タイミングを予測して命令を挿入する。その発行タイミングに CAM 構造のスケジューリングウインドウに命令を挿入する。これにより CAM 構造のスケジューリングウインドウのエントリを有効利用することができる。Data Flow 予測を利用した手法として

Cyclone 方式[10] がある。この手法では命令の発行タイミングに発行可能かをチェックする。もし発行不可能であった場合には発行タイミングの予測からやり直すことにより回路の複雑さを軽減している。しかし、これらの方式ではスケジューリングの精度が低く HSW 方式などと比較して ILP を有効に利用することは不可能である。メインメモリのアクセスのレイテンシの隠蔽に関しては WIB 方式[11]、SLIQ 方式[12] などが存在する。これらの手法では従来の小さいスケジューリングウインドウの他に大きい FIFO を用意しておく。キャッシュミスしたロードなどのように大きなレイテンシを持つ命令が発行されたときにその依存命令をそのバッファに退避させることにより小さいスケジューリングウインドウを有効に利用するというアルゴリズムである。しかし、この手法は大きいレイテンシを持つ命令の実行終了が終わった後に FIFO から従来のウインドウに再挿入されて実行するためクリティカルパスの命令の実行が遅れてしまう。

6. ま と め

本稿の中で SWA という拡張性のある命令スケジューリング方式を提案した。従来の EDF 方式を利用したスケジューリングアルゴリズムでは Limited Vector に依存情報が乗り切らなかったときのペナルティがスケジューリングウインドウの大きさに比例して大きくなるという欠点を持つ。この欠点を SWA というスケジューリングアルゴリズムを利用することにより解決できることを示した。このアルゴリズムは依存関係を List 構造で管理することにより Limited Vector を用いたアルゴリズムと比較して高い拡張性を実現する。結果として従来の単純な Limited Vector を利用したアルゴリズムと比較して 10.4%性能が向上した。また、従来通りの Limited Vector 方式を併用したハイブリッド型では同程度の資源を利用した Limited Vector と比較して 4.2%性能が向上した。EDF 方式のスケジューリングアルゴリズムにおいて List 型のデータの管理を用いた逐次型の Wakeup 方式が有効であることが示されたといえる。

参 考 文 献

1) Austin, T. M. and Sohi, G. S.: Dynamic dependency analysis of ordinary programs, *Proceedings of the 19th annual international symposium on Computer architecture*, ACM Press, pp. 342–351 (1992).
2) Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-effective superscalar processors, *Proceedings of the 24th annual international symposium on Computer architecture*, ACM Press, pp. 206–218 (1997).
3) Sato, T., Nakamura, Y. and Arita, I.: Revisiting Direct Tag Search Algorithm on Superscalar Processors, *Workshop on Complexity-Effective Design held in conjunction with the 28th Annual International Symposium on Computer Architecture* (2001).
4) Brekelbaum, E., Rupley, J., Wilkerson, C. and Black, B.: Hierarchical Scheduling Windows, *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society Press, pp. 27–36 (2002).
5) Goshima, M., Nishino, K., Kitamura, T., Nakashima, Y., Tomita, S. and ichiro Mori, S.: A high-speed dynamic instruction scheduling scheme for superscalar processors, *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 225–236 (2001).
6) Canal, R. and Gonzlez, A.: A low-complexity issue logic, *Proceedings of the 14th international conference on Supercomputing*, ACM Press, pp. 327–335 (2000).
7) Kessler, R. E.: The Alpha 21264 Microprocessor, *IEEE Micro*, Vol. 19, No. 2, pp. 24–36 (1999).
8) Stark, J., Brown, M. D. and Patt, Y. N.: On pipelining dynamic instruction scheduling logic, *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ACM Press, pp. 57–66 (2000).
9) Michaud, P. and Seznec, A.: Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors, *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, IEEE Computer Society, pp. 27–36 (2001).
10) Ernst, D., Hamel, A. and Austin, T.: Cyclone: a broadcast-free dynamic instruction scheduler with selective replay, *Proceedings of the 30th annual international symposium on Computer architecture*, ACM Press, pp. 253–263 (2003).
11) Lebeck, A.R., Koppanalil, J., Li, T., Patwardhan, J. and Rotenberg, E.: A large, fast instruction window for tolerating cache misses, *Proceedings of the 29th annual international symposium on Computer architecture*, IEEE Computer Society, pp. 59–70 (2002).
12) Cristal, A., Ortega, D., Llosa, J. and Valero, M.: Out-of-Order Commit Processors, *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA'04)* (2004).