(19)日本国特許庁(JP)　(12)公表特許公報(A)　(11)特許出願公表番号
特表2005-504390
(P2005-504390A)
(43)公表日　平成17年2月10日(2005.2.10)

(51)Int.Cl.[7]	F I			テーマコード（参考）
G06F 9/45	G06F	9/44	320C	5B013
G06F 9/38	G06F	9/38	330A	5B081
	G06F	9/38	330K	

審査請求 未請求　予備審査請求 未請求　（全 42 頁）

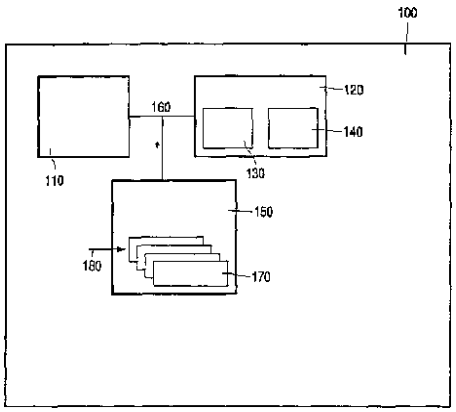
(21)出願番号　特願2003-533105(P2003-533105)
(86)(22)出願日　平成14年9月9日(2002.9.9)
(85)翻訳文提出日　平成16年1月30日(2004.1.30)
(86)国際出願番号　PCT/IB2002/003646
(87)国際公開番号　WO2003/029961
(87)国際公開日　平成15年4月10日(2003.4.10)
(31)優先権主張番号　01402545.6
(32)優先日　平成13年10月2日(2001.10.2)
(33)優先権主張国　欧州特許庁(EP)

(71)出願人　590000248
コーニンクレッカ　フィリップス　エレクトロニクス　エヌ　ヴィ
Ｋｏｎｉｎｋｌｉｊｋｅ　Ｐｈｉｌｉｐｓ　Ｅｌｅｃｔｒｏｎｉｃｓ　Ｎ．Ｖ．
オランダ国　５６２１　ベーアー　アインドーフェン　フルーネヴァウツウェッハ　１
Ｇｒｏｅｎｅｗｏｕｄｓｅｗｅｇ　１，５６２１　ＢＡ　Ｅｉｎｄｈｏｖｅｎ，Ｔｈｅ　Ｎｅｔｈｅｒｌａｎｄｓ
(74)代理人　100087789
弁理士　津軽　進
(74)代理人　100114753
弁理士　宮崎　昭彦

最終頁に続く

(54)【発明の名称】ジャバハードウェアアクセラレータ用の投機的実行

(57)【要約】
条件付き分岐バイトコードは仮想マシンインタープリタ（ＶＭＩ）ハードウェアアクセラレータにより処理され、該アクセラレータは分岐予測方法を使用して、ＣＰＵが条件制御変数を戻すのを待つ間にバイトコードを投機的に処理すべきかを決定する。或る分岐予測方法においては、上記ＶＭＩは、条件付き分岐バイトコードが後方ジャンプを求める場合は分岐条件が満たされると仮定し、条件付き分岐バイトコードが前方ジャンプを求める場合は分岐条件が満たされないと仮定する。他の分岐予測方法においては、上記ＶＭＩは条件付き分岐バイトコードが後方ジャンプを求める場合にのみ仮定を行う。更に他の投機的実行方法において、上記ＶＭＩは、条件付き分岐バイトコードを処理する場合は常に分岐条件が満たされると仮定する。該ＶＭＩは、容易に元に戻すことが可能なバイトコード（例えば、単純なスタック操作を表すバイトコード）のみを投機的に処理し、容易に元に戻すことができないバイトコードに出会うと、バイトコードの投機的処理を保留する。該ＶＭＩによりなされた仮定は、上記条件制御変数が入力された際に確認又は無効



【特許請求の範囲】
【請求項１】
仮想マシン命令を処理する方法において、該方法が、
処理が条件制御変数の値に条件的に依存するような仮想マシン命令の部分集合を識別するステップと、
一連の仮想マシン命令を取り込むと共に、これら仮想マシン命令をプロセッサにより実行可能なネイティブ命令へと処理するステップであって、前記一連の仮想マシン命令における仮想マシン命令を処理し、実行のために前記プロセッサにディスパッチした後に仮想マシン命令カウンタがインクリメントされ、当該ステップを前記仮想マシン命令の部分集合のメンバに出会うまで継続するようなステップと、
前記メンバに出会うと、該メンバにより表されるネイティブ命令を実行のために前記プロセッサにディスパッチすることにより制御値取出処理を開始するステップと、
前記条件制御変数の実際値の仮定値に関して仮定がなされた場合に仮想マシン命令の投機的シーケンスを処理するステップであって、該投機的シーケンスを、
投機モードスイッチをオンに切り換え、
前記条件制御変数の前記実際値が入力されるまで、前記仮想マシン命令の投機的シーケンスにおける各仮想マシン命令をネイティブ命令へと処理するために前記仮定値に従って前記仮想マシン命令カウンタを更新し、前記各仮想マシン命令が処理された後に前記仮想マシン命令カウンタをインクリメントすると共に前記ネイティブ命令をディスパッチし、
前記実際値を前記仮定値と比較し、
前記実際値が前記仮定値と等しくない場合に、前記仮想マシン命令の投機的シーケンスの処理を元に戻し、投機モード履歴を除去し、前記仮想マシン命令カウンタの前記更新を元に戻すか、又は
前記実際値が前記仮定値に等しい場合に、前記投機モードスイッチをオフに切り換えると共に、前記投機モード履歴を除去する、
ことにより処理するようなステップと、
前記条件制御変数の実際値の仮定値に関して仮定がなされない場合に、前記一連の仮想マシン命令の更なる取り込み及び処理を前記条件制御変数の前記実際値が入力されるまで遅らせ、次いで次の仮想マシン命令を該実際値に従って処理するステップと、
を有することを特徴とする方法。
【請求項２】
請求項１に記載の方法において、前記一連の仮想マシン命令を取り込み及びプロセッサにより実行可能なネイティブ命令へと処理するステップが、仮想マシンインタプリタ（ＶＭＩ）の仮想マシンハードウェアアクセラレータによりなされることを特徴とする方法。
【請求項３】
請求項１に記載の方法において、前記仮想マシン命令の投機的シーケンスを処理するステップが、
前記投機的シーケンスの各仮想マシン命令を、該仮想マシン命令が容易に元に戻すことができるものである場合にのみ処理するステップと、
次の仮想マシン命令がスタック操作を表していない場合は、前記仮想マシン命令の投機的シーケンスの処理を、前記条件制御変数の実際値が入力されるまで保留するステップと、
を有していることを特徴とする方法。
【請求項４】
請求項３に記載の方法において、前記投機的シーケンスの処理を保留するステップが、前記条件制御変数の実際値が入力されるまで、前記プロセッサに対して一連の"ノーオペレーション"命令をディスパッチするステップを有していることを特徴とする方法。
【請求項５】
請求項２に記載の方法において、前記条件制御変数の実際値の仮定値に関して仮定を行うべきかを決定するステップが、
識別された前記仮想マシン命令の部分集合の前記メンバにより求められる前記仮想マシン

命令カウンタに対する変位を決定するステップと、
前記識別された仮想マシン命令の部分集合の前記メンバが前記仮想マシン命令カウンタの負の変位（オフセット<０）を求める場合に、前記条件制御変数の前記仮定値を分岐条件が満たされることを示すように設定するステップと、
を有していることを特徴とする方法。
【請求項６】
請求項５に記載の方法において、前記識別された仮想マシン命令の部分集合の前記メンバが前記仮想マシン命令カウンタの正の変位（オフセット>０）を求める場合に、前記条件制御変数の前記仮定値を前記分岐条件が満たされないことを示すように設定するステップを更に有していることを特徴とする方法。
【請求項７】
請求項５に記載の方法において、前記識別された仮想マシン命令の部分集合の前記メンバが前記仮想マシン命令カウンタの正の変位（オフセット>０）を求める場合に、前記条件制御変数の前記実際値に関して仮定を行わないステップを更に有していることを特徴とする方法。
【請求項８】
請求項２に記載の方法において、前記条件制御変数の実際値の仮定値に関して仮定を行うべきかを決定するステップが、前記条件制御変数の前記仮定値を分岐条件が満たされることを示すように設定するステップを更に有していることを特徴とする方法。
【請求項９】
請求項２に記載の方法において、前記条件制御変数の実際値の仮定値に関して仮定を行うべきかを決定するステップが、前記条件制御変数の前記仮定値を分岐予測方法に従って設定するステップを更に有していることを特徴とする方法。
【請求項１０】
請求項２に記載の方法において、前記条件制御変数の実際値の仮定値に関して仮定を行うべきかを決定するステップが、前記条件制御変数の前記仮定値を投機的実行方法に従って設定するステップを更に有していることを特徴とする方法。
【請求項１１】
仮想マシン命令を処理する方法において、該方法が、
一連の仮想マシン命令を取り込むと共に、これら仮想マシン命令をプロセッサにより実行可能なネイティブ命令へと処理するステップであって、当該ステップを条件付き分岐命令に出会うまで継続するようなステップと、
前記条件付き分岐命令に出会うと、該条件付き分岐命令により表されるネイティブ命令を前記プロセッサに実行のためにディスパッチすることにより制御値取出処理を開始するステップと、
条件制御変数の値に関して仮定を行うかを決定するステップと、
前記条件制御変数の値に関して仮定が行われた場合に、仮想マシン命令の投機的シーケンスを処理するステップであって、該投機的シーケンスを、
前記仮想マシン命令の投機的シーケンスにおける仮想マシン命令を、該仮想マシン命令が容易に元に戻すことが可能である場合にのみ、予備的ネイティブ命令へと処理し、
該仮想マシン命令がスタック操作を表さない場合に、前記仮想マシン命令の投機的シーケンスの処理を前記条件制御変数の値が入力されるまで保留する、
ことによって処理するステップと、
前記条件制御変数の値に関して仮定がなされない場合に、前記一連の仮想マシン命令の更なる取り込み及び処理を前記条件制御変数の値が入力されるまで遅らせ、次いで次の仮想マシン命令を該入力された値に従って処理するステップと、
を有することを特徴とする方法。
【請求項１２】
請求項１１に記載の方法において、前記仮想マシン命令の投機的シーケンスを処理するステップが、

前記入力された条件制御変数の値を仮定値と比較するステップと、
前記条件制御変数の値が前記仮定値と等しくない場合に、前記仮想マシン命令の投機的シーケンスの処理を元に戻すと共に、投機モード履歴を除去するステップと、
前記条件制御変数の値が前記仮定値と等しい場合に、投機モードスイッチをオフに切り換えると共に、前記投機モード履歴を除去するステップと、
を更に有することを特徴とする方法。
【請求項１３】
仮想マシン命令を処理する装置において、
ネイティブ命令セットを有すると共にネイティブ命令を実行するように構成されたプロセッサと、
仮想マシン命令を記憶するように構成された命令メモリと、
前記命令メモリから仮想マシン命令を取り込むと共に該取り込まれた仮想マシン命令を前記プロセッサにより実行可能なネイティブ命令へと処理し、条件付き仮想マシン命令を識別し、前記プロセッサにより条件制御変数の値が送られつつある間に仮想マシン命令を投機的に処理すべきかを決定し、これら投機的に処理された仮想マシン命令を前記条件制御変数が入力された際に確認するか又は元に戻すように構成されたプリプロセッサと、
を有し、前記プリプロセッサが、更に、
前記プロセッサから入力された前記条件制御変数の値を記憶するように構成された制御レジスタと、
処理されるべき次の仮想マシン命令を示すように構成された仮想マシン命令カウンタと、
仮定された制御変数の値を記憶するように構成された仮定変数レジスタと、
当該プリプロセッサが前記条件制御変数の値に関して仮定を行ったかを示すように構成された投機モードビットと、
前記条件制御変数の値が送られつつある間に投機モードビットを記憶するように構成された投機モード履歴と、
を有していることを特徴とする装置。
【請求項１４】
請求項１３に記載の装置において、前記プリプロセッサが仮想マシンインタープリタ（ＶＭＩ）の仮想マシンハードウェアアクセラレータであることを特徴とする装置。
【請求項１５】
請求項１３に記載の装置において、前記プリプロセッサは、容易に元に戻すことができる仮想マシン命令のみを投機的に処理し、容易に元に戻すことができない仮想マシン命令に出会った場合は仮想マシン命令の投機的シーケンスの処理を前記条件制御変数の値が入力されるまで保留するように更に構成されていることを特徴とする装置。
【請求項１６】
請求項１５に記載の装置において、前記プリプロセッサは、投機的処理が保留されている間において、且つ、前記条件制御変数の値が入力されるまで、一連の"ノーオペレーション"命令をディスパッチするように更に構成されていることを特徴とする装置。
【請求項１７】
請求項１５に記載の装置において、前記プリプロセッサは、スタック操作を表すような仮想マシン命令のみを投機的に処理し、スタック操作でない仮想マシン命令に出会った場合は仮想マシン命令の投機的シーケンスの処理を前記条件制御変数の値が入力されるまで保留するように更に構成されていることを特徴とする装置。
【請求項１８】
請求項１７に記載の装置において、前記プリプロセッサは、投機的処理が保留されている間において、且つ、前記条件制御変数の値が入力されるまで、一連の"ノーオペレーション"命令をディスパッチするように更に構成されていることを特徴とする装置。
【発明の詳細な説明】
【技術分野】
【０００１】

本発明は、広くはコンピュータプログラミング言語に係り、更に詳細には仮想マシン言語の翻訳及び実行に関する。

【背景技術】

【０００２】

コンピュータプログラミング言語は、コンピュータが実行する命令を表すような人が読むことが可能なソースコードからなるアプリケーションを作成するために使用される。しかしながら、コンピュータが理解することができる前に、上記ソースコードはコンピュータが読み取り可能な二進マシンコードに翻訳されねばならない。

【０００３】

Ｃ、Ｃ＋＋又はコボルのようなプログラミング言語は、典型的には、上記ソースコードからアセンブリ言語を発生し、次いで該アセンブリ言語をマシンコードに変換されるマシン言語に変換するためにコンパイラを使用している。このように、ソースコードの最終的翻訳は実行時の前に行われる。異なるコンピュータは異なるマシン言語を必要とするので、例えばＣ＋＋で書かれたプログラムは、該プログラムが書かれた特定のハードウェアプラットフォーム上のみで実行することができる。

【０００４】

解釈（interpreted）プログラミング言語は、複数のハードウェアプラットフォーム上で実行するソースコードでアプリケーションを作成するように設計されている。ジャバ（登録商標）は、実行時前に"バイトコード"又は"仮想マシン言語"として知られている中間言語に変換されるようなソースコードを発生することにより、プラットフォームからの独立性を達成する解釈プログラミング言語である。実行時において、上記バイトコードは、米国特許第4,443,865号に開示されているように、インタープリタソフトウェアを介してプラットフォームに適したマシンコードに変換される。各バイトコードを解釈するために、インタープリタソフトウェアは"取り込み、デコード及びディスパッチ"（ＦＤＤ）なる一連の処理を実行する。各バイトコード命令に対して、上記インタープリタソフトウェアは、ネイティブ中央処理装置（ＣＰＵ）命令で表された対応する実行プログラムを含んでいる。該インタープリタソフトウェアはＣＰＵにメモリから仮想マシン命令を取り込み又は読み取らせ、当該バイトコード命令に関して実行プログラムのＣＰＵアドレスをデコードさせ、ＣＰＵの制御を該実行プログラムに移行させることによりディスパッチさせる。斯かる解釈処理は時間の掛かるものであり得る。

【０００５】

国際特許出願公開第WO9918484号に開示されているように、メモリとＣＰＵとの間にプリプロセッサ（仮想マシンインタープリタ（ＶＭＩ））を付加することにより、仮想マシン命令の処理が改善される。本質的に、仮想マシンは物理的構造ではなく、むしろ、ＶＭ内又はＣＰＵ内に記憶された対応するネイティブマシン言語命令を選択することにより当該ハードウェアプラットフォーム用にバイトコードを解釈する自立型動作環境である。この場合、上記ネイティブ命令は当該ハードウェアプラットフォームのＣＰＵに供給され、次いで該ＣＰＵにおいて実行される。典型的な仮想マシンは、ＦＤＤ系列の処理を実行するのに、バイトコード当たり２０ないし６０サイクルの処理時間を必要とする（当該バイトコードの質及び複雑さに依存する）。先ず、ＶＭＩはメモリからバイトコードを読み取る（取り込む）。次に、ＶＭＩは該取り込まれたバイトコードの複数のプロパティをルックアップする（デコードする）。ＶＭＩによりアクセスされたプロパティは、当該ＣＰＵにおいて実行するために、該バイトコードがどの様にネイティブ命令に処理されるかを決定する。ＣＰＵが命令を実行している間に、ＶＭＩは次のバイトコードを取り込み、該次のバイトコードをＣＰＵ命令に処理する。ＶＭＩは単純なバイトコードを１ないし４サイクルで処理することができる。

【０００６】

一連のバイトコードを解釈している間に、仮想マシンは、条件付き分岐命令（以下、ＣＢＩと呼ぶ）を表すバイトコードに出会う可能性がある。ＣＢＩに出会うと、ＶＭＩは、ＣＰＵに当該条件が満たされているかを判定させる一連のネイティブ命令を発生する。従っ

て、当該分岐を実行するとの決定は以前の計算に依存し、斯かる計算はＶＭＩの概念ではＣＰＵレジスタに残存している結果によりＣＰＵで実行されたものである。例えば、ジャバ（登録商標）のバイトコード"ifeq n"はバイトコードカウンタを"ｎ"だけオフセットするが、これは当該スタックの最上部が零である（即ち、前の計算が該スタックに値零を残した）場合のみである。分岐条件（ここでは、スタックの最上部）の値は取り出され、ＶＭＩの制御レジスタ（分岐条件のために特別に確保される）に書き込まれなければならない。条件が満たされている場合は、該ＣＢＩはＶＭＩバイトカウンタに対する更新を生じさせ（ジャンプ）、該カウンタが実行されるべきバイトコードのシーケンスを変更させる。典型的には、１つの命令がＶＭＩで処理されつつある場合、処理されるべき次の命令は既にＶＭＩパイプライン内にあるので、或る命令が分岐となる場合、ＶＭＩパイプライン内に既にあるバイトコードは流されねばならない。更に、プロセッサハードウェアの"パイプライン化"構造は、命令及び／又はデータがプロセッサにディスパッチされる時点と、プロセッサが実効的に斯かる命令を実行し及び／又は斯かるデータを処理する時点との間で斯かる命令及びデータを移送する固有の遅延を生じる。詳細には、典型的なＣＰＵは多段（典型的には、３ないし８段）のパイプラインを有しているので、書込命令は該命令が送出された直後には実行されないであろう。ＣＢＩの場合、ＣＰＵが条件が満たされるかを判定し、この判定の結果をＶＭＩに転送する間に付加的な遅延を生じる。分岐条件の値（制御値）が当該分岐条件が満たされていることを示す場合、幾つかの（ＣＰＵパイプラインの大きさに依存する）命令が既にＣＰＵパイプラインに入っているであろう。ＣＰＵ及び命令キャッシュをビジーに維持するために、当該条件が満たされるかを示す制御値を待つ間に、一連の"ノーオペレーション"（ＮＯＰ）コマンドを発生させることができる。制御値は、ＣＰＵが最後より１つ前のＮＯＰを実行し、ＶＭＩが最後のＮＯＰを発生する間に入力される。上記判定を行った後、ＶＭＩパイプラインは、ＶＭＩが次の命令を表すバイトコードをＶＭＩキャッシュから取り出すのに数サイクルを要する。

【０００７】

他の方法は、命令が結果として他のロケーションへの分岐となるかを予測することにより、可能性のある分岐命令を投機的に実行する。この方法の一例はＲＩＳＣ（縮小命令セット計算）マイクロプロセッサに関するもので、どの条件付き分岐が予測するのが"容易"であるかを判断するために分岐命令ビットを設け、これらの分岐に対してはジャンプを実行すべきかを判断するためにソフトウェア分岐予測を使用する。ソフトウェア分岐予測は、ソフトウェア制御された予測ビットを用いて分岐を予測する。分岐が予測するのが"困難"であると判断されたら、該分岐はハードウェア分岐予測（分岐予測アレイのような）を用いて予測される。この方法は、オフセットが零より小さい（後方分岐）なら分岐がとられ、オフセットが零より大きければ（前方分岐）分岐がとられないと予測するような分岐予測方法を用いることを開示している。この方法の問題点は、とられた分岐の履歴的処理が該分岐がとられるかを判断する場合に重要であるか否かに基づくような、前記予測の容易さの判断を行い且つ更新するためにプロセッサの資源を消費する点にある。

【０００８】

他の分岐予測方法においては、可能性のある分岐命令のアドレスからのビットが、ローカル分岐履歴テーブル及び全体履歴レジスタから連結されたビットと比較される。該比較の結果は分岐予測テーブルを読み取るために使用される。この方法の問題点は、上記連結及び比較処理を実行し、上記分岐予測テーブルを記憶及びアクセスするために要する資源の消費である。更に、該方法は予測誤りを補正する手段を開示していない。同様の方法論が米国特許第5,136,696号に開示されており、該方法において分岐予測は分岐キャッシュにより可能性のある分岐命令のアドレスに基づいて実行される。該開示によれば、予測が誤っている場合、対応する命令は無効化されるが、いずれにせよ実行されるので、分岐キャッシュは同一の命令に再び出会う場合に正しい予測により更新することができる。ＣＰＵパイプラインは、上記分岐キャッシュ更新と同一のサイクルの間において、当該パイプラインの最初の７段の命令の全てを無効化し、プログラムカウンタレジスタの内容をロードすることにより流される。

【発明の開示】
【発明が解決しようとする課題】
【０００９】
条件付き分岐は頻繁に発生し（全仮想マシン命令のうちの約１０％）、高い精度は達成するがプロセッサ資源を消費するような既存の方法により処理される場合に処理が大変であるので、条件付き分岐命令バイトコードにより意図される命令を正確且つ効率的に実行し、それでいて処理速度を向上させるような、プログラミング言語を解釈するシステムへの要求が存在する。
【課題を解決するための手段】
【００１０】
本発明は、仮想マシンハードウェアアクセラレータ（ＶＭＩのような）を、条件付きバイトコード（ＣＢＩ）に出会った場合に該ＶＭＩが分岐予測を実行すると共に投機的実行処理を開始すべきかを決定するように構成することにより、上述した要求を満たす。決定された場合、投機的実行は、投機的に実行されたバイトコードが容易に元に戻し得る（reversible）限り、又は予測が確認されるまで、継続する。殆どの場合予測は正しく、これにより、予測が確認された後ＶＭＩ及びＣＰＵが投機的に選択されたバイトコードのシーケンスに沿って動作し続けるのを可能にする。予測が正しくない場合（従って無効化される）、ＶＭＩ及びＣＰＵで実行された投機的処理は流され、ＶＭＩは当該分岐の直前の状態に戻される。効率性は、仮想マシンハードウェア加速技術の性能利得と分岐予測方法及び可能性のある投機的実行とを組み合わせることにより、及び影響を容易に元に戻すことができる限りにおいてのみ命令を投機的に実行することにより実現される。正確さは、如何なる予測誤りも補正することにより達成される。従って、平均すると、条件付き分岐（ＣＢＩ）は比較的少ない遅延しか生じない。
【００１１】
本発明は、より複雑な分岐予測方法論（オブジェクト指向技術及びシステムに関する第５回USENIX会議（１９９９）の原稿集、第２１７～２２８頁の"ジャバにおける仮想方法呼出をサポートするための分岐予測器のチューニング"で推奨されているハッシング方法及び経路履歴方法論のような）及び出会った各潜在的分岐命令のタイプに基づいて簡単な（しかし確かな）予測を行う方法に匹敵する。
【００１２】
簡単に述べると、本発明は条件付き分岐仮想マシン命令を処理するシステム及び方法を含むもので、斯かる命令は本発明の実施例においてはジャバ（登録商標）プログラミング言語により発生される。プログラミングレベルにおいては、ジャバ（登録商標）ソースコードは、バイトコードと呼ばれる中間言語にコンパイルされる。バイトコードは、プロセッサにより実行することができるように仮想マシンによって解釈することができるような仮想マシン命令からなる。本発明の実施例によれば、ランタイム時において、仮想マシン（実施例では、ＶＭＩ）は初期化される。各バイトコードの特定のプロパティの識別を可能にするパラメータが初期化される。例えば、バイトコードは単純又は複雑なものとして特徴付けることができ、更に条件付き又は再帰的として特徴付けることができる。条件付きバイトコードは、実行されるべきバイトコードのシーケンスを、分岐条件が満たされる場合にのみ、変更する仮想マシン命令である。
【００１３】
ＶＭＩは、一連のバイトコードの各々を１以上のネイティブ命令に処理するよう進行する。ＶＭＩは、各バイトコードが命令メモリから取り出される毎にインクリメントされるバイトコードカウンタを維持する。ＣＢＩに出会うと、ＶＭＩはＣＰＵに分岐条件制御値を取り出すことにより分岐条件が満たされているかを判断させるようなネイティブ命令のシーケンスを発生する。本発明の実施例のシステム及び方法によれば、制御値の取り出しが完了するまで処理を一次中止するというよりは、ＶＭＩは分岐予測を実行し、投機的実行を実行すべきか、即ち分岐条件が満たされるであろうかについて仮定を行うべきか、を決定する。仮定が行われた場合、該仮定が確認若しくは無効化されるまで、又は容易に元に

戻すのができないバイトコードに出会うまで、バイトコード系列は投機的に実行される。

【００１４】

本発明の最初の３つの実施例においては、上記判断はＣＢＩにより指定された分岐のタイプに基づくものである。第１の実施例によれば、ＶＭＩはＣＢＩが後方分岐を指定しているなら分岐条件は真であると仮定し、該分岐により目標とされる次のバイトコードにジャンプするようバイトカウンタを更新する。当該ＣＢＩが前方分岐を指定する場合、ＶＭＩは分岐条件が満たされないであろうと仮定し、該分岐を実行しない。代替実施例は、ＣＢＩが後方分岐を求める場合、当該分岐条件は満たされると仮定するが、該ＣＢＩが前方分岐を求める場合は仮定を行わない。他の代替実施例においては、ＶＭＩは分岐条件が常に満たされると仮定し、投機的分岐に沿うバイトコードが容易に元に戻せるものである限り、及び当該仮定が確認又は無効化されるまで、該投機的分岐に沿ってバイトコードを処理する。

【００１５】

更に他の実施例においては、分岐を投機的に実行すべきかを判断するのに既知の分岐予測方法を使用するが、ＶＭＩは容易に元に戻すことが可能なバイトコードを、当該仮定が確認又は無効化されるまでしか、投機的に処理しない。この方法論は、ＶＭＩを用いた実施化に関して後に詳細に説明されるが、より多くの分岐予測又は投機的実行方法を用いて実施化することができる。

【００１６】

本発明の実施例によれば、仮定がなされた場合、ＶＭＩは投機的に実行される分岐に沿うか又はバイトコードの元のシーケンスに沿うかの何れかの順次のバイトコードの翻訳であるネイティブ命令を、これらのネイティブ命令が、例えばスタック操作に関わる命令（スタックプッシュのような）のような、容易に元に戻すことができるものである限り、ＣＰＵに対してディスパッチし続ける。容易に元に戻すことができないような命令に出会うと、制御値が入力されるまでＣＰＵをビジーに維持するために、ＶＭＩにより一連の"ノーオペレーション"（ＮＯＰ）コマンドが発生され、ディスパッチされる。続いて入力された制御値が仮定は正しいことを示すなら、ＶＭＩはＶＭＩパイプラインに存在するバイトコードを処理し続け、ＣＰＵはＣＰＵパイプライン内のネイティブ命令を処理し続ける。続いて入力された制御値が当該仮定は正しくない（即ち、予測誤りがあった）ことを示している場合は、ＣＰＵは当該分岐の直前の状態に戻され、これにより、当該投機的に実行された容易に元に戻すことが可能な（逆転可能な）バイトコードを元に戻す。この様に、正しくない仮定による予測誤りの逆転は、ＶＭＩ及びＣＰＵの両者を当該分岐の直前の各状態に戻すことを必要とし、投機的に実行されたバイトコードの特徴により容易に達成される。例えば、本発明の一実施例においては、"容易に元に戻すことが可能なバイトコード"とは、スタック操作のみを実行するか又はＶＭＩ外の如何なる状態も変更しないようなバイトコードと定義され、従って斯様なバイトコードのシーケンスの実行は、ＶＭＩにおけるバイトコードカウンタ及びレジスタスタックポインタをリセットすることにより逆転することができる。本発明によれば"容易に元に戻すことが可能な"の他の定義を実施することもでき、これらの定義の幾つかは、投機的に実行されたバイトコードを元に戻すためにＶＭＩ及びＣＰＵのパイプラインが流されることを必要とし得る。

【００１７】

本発明の他の態様は、ジャバ（登録商標）のような解釈言語からの仮想マシン命令を実行するシステムである。該システムは、プロセッサ（ＣＰＵ）及びプリプロセッサ（ＶＭＩ）、命令メモリ、並びにトランスレータ（ＪＶＭ）を含む。上記プロセッサは、ハードウェアに固有の命令（以下、ネイティブ命令と呼ぶ）を含み且つ斯かる命令を実行するように構成される。上記プリプロセッサは、上記命令メモリからバイトコードを取り込むと共に該バイトコードをネイティブＣＰＵ命令に翻訳するように構成された仮想マシン（例えば、ＶＭＩ）である。該ＶＭＩは制御レジスタ、バイトコードカウンタＢＣＣ、仮定変数レジスタ及び投機的モードスイッチ履歴を含む。本発明の実施例においては、上記ＶＭＩは、各バイトコードにより呼ばれるネイティブ命令を処理のためにＣＰＵにディスパッチ

するように構成されている。処理されるバイトコードがＣＢＩである場合、ＶＭＩによりディスパッチされたネイティブ命令は、ＣＰＵに分岐が実行されるべきかを示す制御値を送出させる。また、ＶＭＩは、分岐条件が満たされると推測又は確認された場合に、ＣＢＩにより呼ばれたジャンプを実行すべくＢＣＣを更新するように構成される。制御値を待つ間、ＶＭＩは、上記において実施例として掲げられた方法による分岐予測方法に基づいて、分岐を投機的に処理し又は処理をやめるように構成される。ＶＭＩは、仮定された制御値ＡＣＶを記憶することにより、仮定を行い又は仮定をするのをやめる際に仮定変数の値を更新するように構成される。また、ＶＭＩは投機的モードスイッチ履歴を維持するように構成され、該履歴はシーケンスが投機的に実行される場合又は投機的実行が終了する場合に更新され、制御値が入力される場合に除去される。分岐が投機的に実行されるべきであるとの仮定をした後、ＶＭＩは逆転可能なバイトコードを識別すると共に処理し、制御値を入力し、該ＶＭＩが前の仮定は誤っていたことを示すような制御値を後に入力した場合に投機的に実行されたバイトコードを逆転させる（元に戻す）ように構成される。

【００１８】

本発明の方法を種々のタイプの条件付き分岐命令を処理するように実施化することは可能であるが、本発明の実施例は、ジャバ（登録商標）の条件付き分岐バイトコードの処理に向けられている。

【００１９】

本発明は、ジャバ（登録商標）バイトコードをサンマイクロシステムズにより作成されたＪＶＭのような仮想マシンを用いて実行するようなシステムにおいて実施化することができる。しかしながら、本発明は、マイクロソフト仮想マシンのような他のジャバ（登録商標）仮想マシンを使用しても実施化することができ、ビジュアルベーシック、ｄＢＡＳＥ、ベーシック及び.ＮＥＴ等の他の解釈言語を実行するシステムにも適用可能である。

【００２０】

本発明の更なる目的、利点及び新規な特徴は、下記の記載に一部述べられ、当業者にとっては下記の説明を調べることにより一部明らかとなり、又は本発明の実施により知ることができるであろう。

【００２１】

明細書に組み込まれると共にその一部を形成する添付図面は、下記の説明を参照して見ることにより本発明を解説するものである。

【発明を実施するための最良の形態】

【００２２】

ここでは、必要に応じて、本発明の実施例が開示されるが、開示された斯かる実施例は種々の且つ代わりの形態で実施化することができるような本発明の単なる例示に過ぎないことに注意すべきである。また、各図は必ずしも寸法通りではなく、特定の構成要素の細部を示すために幾つかの特徴は誇張され又は極小化されている。従って、ここに開示された特定の構造的及び機能的詳細は、制限するものとしてではなく、単なる請求項の基礎として及び当業者に本発明を多様に使用することを教示するための代表的基礎として解釈されるべきものである。

【００２３】

添付図面（これら図面において、同様の符号は同様の構成要素を示す）に示された本発明の実施例を詳細に参照すると、図１は本発明の環境の実施例のブロック図である。該環境の基本的構成要素はハードウェアプラットフォーム１００であり、該プラットフォームはプロセッサ１１０、プリプロセッサ１２０及び命令メモリ１５０を含み、これらは全てシステムバス１６０に接続されている。プリプロセッサ１２０は制御レジスタ１３０とトランスレータ１４０とを含んでいる。ハードウェアプラットフォーム１００は、典型的には、中央処理装置（ＣＰＵ）、基本周辺機器及びオペレーティングシステム（ＯＳ）を含んでいる。本発明のプロセッサ１１０は、ＭＩＰＳ、ＡＲＭ、インテルｘ８６、パワーＰＣ又はSPARC型マイクロプロセッサであり、ハードウェア固有の命令（以下、ネイティブ命令と称す）を含むと共に、斯かる命令を実行するように構成されている。本発明の実施例

において、トランスレータ１４０は例えばサンマイクロシステムズによるＫＶＭ等のジャバ（登録商標）仮想マシン（ＪＶＭ）である。命令メモリ１５０は例えばジャバ（登録商標）バイトコードのような仮想マシン命令を含んでいる。当該実施例におけるプリプロセッサ１２０は国際特許出願公開第WO9918486号に開示された仮想マシンインタープリタ（ＶＭＩ）であり、仮想マシン命令（例えば、バイトコード１７０）を命令メモリ１５０から取り込むと共に、該仮想マシン命令をネイティブＣＰＵ命令のシーケンスに変換するよう構成されている。ＶＭＩ１２０は、バス１６０上の周辺機器であって、メモリマップ型周辺機器として動作することができ、そこでは、所定の範囲のＣＰＵアドレスがＶＭＩ１２０に割り当てられる。ＶＭＩ１２０は、命令メモリ１５０における現在の（又は次の）仮想マシン命令を示す独立の仮想マシン命令ポインタ１８０を管理する。また、該ＶＭＩは投機的モードスイッチ履歴及び仮定変数（図示略）も管理する。

【００２４】

ここで図２を参照すると、本発明の実施例によれば、プロセッサ１１０はマシン命令の実行が連続したクロックサイクルの間において並列且つ密集行進的に行われるように多段パイプライン２００を含んでいる。言い換えると、第１命令Ｉ１が当該パイプラインの段２に入る時、後続の命令Ｉ２は段１に入り、等々となる。理想的には、各命令は連続するサイクルにおいて当該パイプラインに入り続け、これにより、８つの命令の同時的処理を可能にし、ここで、１つの命令はクロックサイクル毎に完全に実行される。しかしながら、典型的には、平均的命令の実行は実際には２クロックサイクル以上を必要とする。

【００２５】

本発明の動作の一例として、ＶＭＩ１２０は第１系列３１０のバイトコードの各々を１以上のネイティブ命令に翻訳して進む。ここで図３を参照すると、バイトコードＢ０ないしＢ２は非条件付きであるので、ＶＭＩ１２０は命令メモリ１５０からＢ０ないしＢ２を単に取り込み、各バイト１７０に対して定義されたネイティブ命令又は複数の命令を選択し、斯かる命令（又は複数の命令）を実行のためにプロセッサ１１０に供給する。Ｂｎは条件付き分岐命令（ＣＢＩ３３０）であるので、分岐条件が満たされるならＢｎは当該ＶＭＩを第１系列３１０から系列３２０におけるバイトコードＢｒへジャンプさせる。分岐条件が評価されている（即ち、ＶＭＩ２０がＣＰＵ１１０から返送されるべき制御値を待つ）間に、分岐予測方法に従い当該分岐条件が満たされるかについて予測がなされ、ＶＭＩ１２０は該予測を検証することができるまで命令を投機的に実行すべきかを決定する。分岐予測方法は、分岐条件が満たされるかについての予測を与えるような多様な複雑さの発見的処理及び、多分、予測の精度の統計的評価である。本発明の一実施例によれば、ＶＭＩ１２０は、ＣＢＩ３３０（Ｂｎ）が後方分岐（即ち、バイトコードカウンタＢＣＣに対して負のオフセット）を求める場合は分岐条件が満たされると仮定するような分岐予測方法を実施し、ＶＭＩ１２０はバイトコードＢｒへのジャンプを実行する。ＶＭＩ１２０は、同時的に、ＣＰＵ１１０に当該分岐条件が満たされているかを示す制御値を取り出させるようなネイティブ命令のシーケンスをディスパッチする。他の実施例の分岐予測方法によれば、ＣＢＩ３３０（Ｂｎ）が前方分岐（即ち、ＢＣＣに対して正のオフセット）を求める場合、ＶＭＩ１２０は当該分岐条件が満たされないと仮定し、ジャンプを実行しない。更に他の実施例の分岐予測方法によれば、ＣＢＩ３３０が分岐を求める場合、ＶＭＩ１２０は当該分岐条件が満たされると仮定し、ジャンプを実行する。

【００２６】

本発明の一実施例によれば、当該分岐予測方法が結果としてバイトコードのシーケンス３２０を投機的に実行することになる場合、不正確な予測の影響は、容易に元に戻す（逆転する）ことが可能なバイトコード１７０のみを投機的に実行することにより最小化される。逆転の容易さは、バイトコード１７０がデコードされる（即ち、該コードのプロパティがアクセスされる）際にＶＭＩ１２０により判断される。ＣＰＵ１１０は、ディスパッチされる各命令を投機的に実行する。例えば、多くの場合、分岐にはスタックプッシュ処理（即ち、定数又は変数がスタックにプッシュされる）が後続し、該処理はスタックポインタ値（ＶＭＩ１２０内に保持される）を当該分岐前の状態に再設定することにより容易に

逆転させることができる。斯様なネイティブ命令（通常、"スタックプッシュ"と呼ばれる）は、制御値が入力されるまで、投機的に実行することができる。何故なら、該投機的実行の逆転は単にレジスタを再設定することにより達成され、従って、予測誤りの悪い影響は無視することができるからである。

【００２７】

このように、ＶＭＩ１２０が分岐条件は満足されると仮定する場合、バイトコードＢｒないしＢｚの処理（投機的分岐）が、ＶＭＩ１２０が該投機的分岐に沿って容易に逆転することができないネイティブ命令のシーケンスを表すようなバイトコードに出会うまで、引き続いて第２のシーケンス３２０において継続する。該投機的分岐に沿うバイトコード１７０が容易に逆転することができないものである場合、投機的実行は一時中止される。次いで、ＶＭＩ１２０が当該分岐条件は満たされなかったことを示すような制御値を入力すると、ＶＭＩ１２０は当該ジャンプを逆転するためにＢＣＣ及びレジスタスタックポインタを再設定する（もし必要なら、ＣＰＵ１１０に対してＣＰＵパイプライン２００を浄化するネイティブ命令を送出する）。ＶＭＩ１２０が分岐条件は満たされないと仮定する場合は、ＢＣＣはインクリメントされ、分岐が投機的に処理されることはない。非分岐バイトコードを投機的に処理している間に、ＶＭＩ１２０は入力された制御値が制御レジスタ１３０に書き込まれたかを継続的にチェックする。

【００２８】

図４のブロック４１０に示すように、ＶＭＩ１２０は、ブロック４２０において命令メモリ１５０から各バイトコード１７０を取り込む前に、仮想マシンカウンタＢＣＣをインクリメントする。ブロック４３０において、ＶＭＩ１２０は各バイトコード１７０を当該バイトコード１７０に関するプロパティにアクセスすることによりデコードする。シーケンスの最初のバイトコード１７０がデコードされた時に、投機モードスイッチＳＭＳはオフであるから、ＶＭＩ１２０は（ブロック４３５において）当該バイトコード１７０に対応するネイティブ命令のシーケンスをディスパッチする。ブロック４４５において、当該バイトコード１７０がＣＢＩ３３０でないと判断された場合は、当該方法はブロック４１０に戻り、ここで仮想マシンカウンタ（ＢＣＣ）１８０がインクリメントされ、取り込み処理に戻る（ブロック４２０）。しかしながら、ブロック４４５において当該バイトコード１７０が条件付きであると判断されたなら、ブロック４３５でディスパッチされたネイティブ命令はブロック４５０内で制御値取込処理を表すようになる。該制御値取込処理を構成するネイティブ命令は、ＣＰＵ１１０に、ＣＢＩ３３０により求められた分岐が実行されるべきであるか（即ち、現在実行されているバイトコードのシーケンス外にある目標バイトコード１７０にジャンプすべきか）を示すような制御値を送出させ、且つ、該制御値を制御レジスタ１３０に書き込ませる。上記制御値取込処理と同時的に、ＶＭＩ１２０は当該分岐条件が満たされるか又は満たされないかの仮定をなすべきかを決定する（ブロック４７０）。該決定は、使用されている特定の分岐予測又は投機実行方法に基づくものである。

【００２９】

ＶＭＩ１２０が当該分岐条件の満足に関する仮定を行うと決定すると、投機モードスイッチ（ＳＭＳ）がオンされ、仮定された制御値（ＡＶＣ）がＶＭＩレジスタに記憶される（仮定変数１９５）。ブロック４７５において、当該分岐が投機的にとられた場合、ＢＣＣが更新される（ＣＢＩ３３０により求められたジャンプを反映するために）。分岐が投機的になされるかに関係なく、当該方法はブロック４１０に進み、ここでＢＣＣはインクリメントされる。投機的に決定されたバイトコードのシーケンスに沿う次のバイトコード１７０は、該バイトコードが逆転可能（例えば、該次のバイトコード１７０に対応するネイティブ命令が、単純なスタック操作を構成する）ならば、処理される。逆転の容易性の判断は、当該バイトコード１７０のプロパティに基づくものである。容易に逆転可能なバイトコードは、典型的には、ＣＢＩ３３０の翻訳の直後のスタック最上部より上のスタック位置上で生じる処理であって、当該システム（特に、ＶＭＩ１２０の外側）の状態を変更するものではないような処理を表す。バイトコード１７０はブロック４２０において取り

込まれ、ブロック４３０においてデコードされる。ＳＭＳはオンである。何故なら、ＶＭＩ１２０は仮定を行うように決定したからである（当該シーケンスは投機的に実行されている）。従って、制御値が未だ返送されておらず（ブロック４８５）、且つ、デコードされたプロパティが取り込まれたバイトコード１７０は容易に逆転可能であることを示している場合は、該取り込まれたバイトコードに対応するネイティブ命令がブロック４３５においてディスパッチされる。次いで、取り込まれるバイトコード１７０が他のＣＢＩ３３０でない限り（ブロック４４５）、ＢＣＣがインクリメントされ（ブロック４１０）、投機的シーケンスに沿う次のバイトコードが取り込まれ、デコードされる（ブロック４２０及び４３０）。このように、投機的シーケンスの処理は、次の各バイトコード１７０が容易に逆転可能なものである限り、且つ、分岐条件が入力されていない限り、ブロック４１０、４２０、４３０、４３２、４８５、４８６、４６０及び４３５を経てループする。

【００３０】

ブロック４６０において、投機的シーケンスの処理の間において容易に逆転可能でないバイトコード１７０に出会うと、ブロック４６１においてＳＭＳをオフに切り換えることにより投機が中止され、ＶＭＩ１２０はブロック４８５において制御値の入力を待たされる。入力された制御値ＲＣＶはＶＭＩ１２０の制御レジスタ１３０に記憶される。ブロック４８７において、当該分岐条件が評価される。ブロック４９４においてＲＣＶが当該分岐条件は満たされたことを示し、且つ、ＳＭＳ履歴が現シーケンスは投機的に処理されていたことを示す（ＳＭＳ履歴１９０が、当該ＳＭＳが現シーケンスの間における何処かでオンであったことを示す）場合、仮定変数１９５として記憶された仮定制御値ＡＣＶが、制御レジスタ１３０に記憶された入力制御値ＲＣＶと比較され、ＶＭＩ１２０によりなされた仮定が正しかったか（即ち、上記投機的シーケンスが処理されるべきであったか）を判断する。仮定が正しかった場合（即ち、ＡＣＶ＝ＲＣＶ）、現分岐シーケンスの投機的実行は確認されたことになる。言い換えると、ＶＭＩ１２０は、当該分岐がなされるべきと正しく仮定した。そして、ＢＣＣがブロック４１０においてインクリメントされ、取り込みに戻る。仮定が正しくなかった場合は、ＶＭＩ１２０は当該分岐を処理し損ない、ＶＭＩ１２０及びＣＰＵ１１０は、投機的に処理された非分岐シーケンスのバイトコードを逆転するために浄化される。そして、当該方法はブロック４９５においてジャンプを反映するためにＢＣＣを更新し、次いでブロック４１０に戻ってＢＣＣをインクリメントし、当該分岐シーケンスに沿う次のバイトコードを処理する。

【００３１】

ブロック４９４において、上記ＲＣＶが当該分岐条件は満足されなかったことを示し、且つ、上記ＳＭＳ履歴１９０が現シーケンスは投機的に処理されていたことを示す場合、仮定制御値ＡＣＶは入力制御値ＲＣＶと比較され、ＶＭＩ１２０によりなされた仮定が正しかったかを判断する。仮定が正しかった場合は、現非分岐シーケンスの投機的実行は確認されたことになる。言い換えると、ＶＭＩ１２０は分岐がされるべきでないことを正しく仮定した。そして、ＢＣＣがブロック４１０においてインクリメントされ、取り込みに戻る。上記仮定が正しくなかった場合は、当該分岐はなされるべきでなかったことになり、投機的に処理された分岐に沿う全てのバイトコードは、ブロック４９５においてＢＣＣ及びスタックレジスタポインタを再設定することにより逆転される。もし必要なら、ＶＭＩ１２０及びＣＰＵ１１０のパイプラインがブロック４９５において浄化される。

【００３２】

ＳＭＳ履歴１９０は制御値が入力される際に常にリセットされる。何故なら、分岐条件が解明され、かくして如何なる投機的処理も確認又は除去するからである。

【００３３】

ブロック４７０において、ＶＭＩ１２０が分岐条件の満足に関して仮定を行わないと決定する場合、ＶＭＩ１２０は、ブロック４８５において制御値の入力をチェックすることにより、実際の制御値が入力されるまで待つ。ＳＭＳはオフのままなので、ブロック４８６はブロック４８５に戻り、制御値が入力されるまでチェックを継続する。制御値が入力されると、当該方法はブロック４８７に進み、そこにおいて、当該分岐条件が評価される。

該制御値が分岐条件は満たされたことを示す場合、当該バイトコードシーケンスの実行の間において何の投機も生じていないので、当該方法はブロック４９５に進み、そこでは、ＣＢＩ３３０により求められたジャンプを反映するためにＢＣＣが更新され、次いでブロック４１０においてバイトコードの取り込みに戻る。上記制御値が分岐条件は満たされなかったことを示す場合、当該方法はブロック４１０に戻り、バイトコードの取り込みに復帰する。このようにして、ＶＭＩ１２０が仮定を行わないと決定すると、ＶＭＩ１２０はＣＢＩ３３０を通常のように処理する（分岐条件が満たされる場合にのみ、分岐を行う）。

【００３４】

上述したように、分岐条件の満足に関して仮定を行うべきかの判断は、下記に示すように、実施される分岐予測方法に基づくものとなる。

【表１】

分岐予測方法		
	後方分岐	前方分岐
第１実施例	仮定する	仮定しない
第２実施例	仮定する	仮定無し
第３実施例	仮定する	仮定する
第４実施例	予測方法に依存する	

【００３５】

本発明の第１実施例において、ＶＭＩ１２０は、後方分岐はなされ、前方分岐はなされないと仮定するような投機的実行方法を使用する。第２の代替実施例において、ＶＭＩ１２０は、後方分岐のみを投機的に実行し、前方分岐に関しては仮定を行わない。第３の代替実施例において、ＶＭＩ１２０は、全ての分岐がなされると仮定するような一層単純な投機的実行方法を使用する。更に、第４の代替実施例によれば、ＶＭＩ１２０は、分岐予測方法を使用して、分岐がなされるべきかを決定する。上述した実施例の何れによっても、投機的実行は容易に逆転が可能なバイトコードに対してのみ進行する。

【００３６】

本発明の第１実施例によれば、ＶＭＩ１２０はＣＢＩ３３０により示される分岐条件の結果に関する仮定を常に行うが、投機はＣＢＩ３３０により求められる分岐のタイプに基づかせる。ＣＢＩ３３０が前方分岐を求める（ＢＣＣオフセット>０）場合、ＶＭＩ１２０は分岐条件が満足されないと仮定し、容易に反転可能である限り、非分岐バイトコードシーケンスの処理を投機的に継続する。ＣＢＩ３３０が後方分岐を求める（ＢＣＣオフセット<０）場合、ＶＭＩ１２０は分岐条件が満足されると仮定し、容易に逆転可能である限りにおいて、分岐バイトコードシーケンスを投機的に処理する。

【００３７】

本発明の第２実施例によれば、ＶＭＩ１２０は、ＣＢＩ３３０が後方分岐を求める場合にのみ、ＣＢＩ３３０により示される分岐条件の結果に関する仮定を行う。ＣＢＩ３３０が前方分岐を求める場合、ＶＭＩ１２０は分岐条件を通常のように処理する。即ち、ＶＭＩ１２０は制御値の入力を待ち、それに従ってバイトコードの適切なシーケンスを処理する。

【００３８】

本発明の第３実施例においては、ＣＢＩに出会った場合に常に分岐がなされるように投機

する。ＲＩＳＣコードを処理する場合、この投機的実行方法は時間の約７０％となる。何故なら、分岐は通常はループの最下部で生じ、ループは典型的には繰り返して実行されるからである。該分岐予測は、ＶＭＩの全体としてのＲＩＳＣコード処理性能を約３５％改善する。アムダールの法則によれば、ジャバコードを処理する場合、この簡単な予測はバイトコード処理速度を７～２０％上昇させるであろう。分岐予測及び投機的処理を使用して、分岐のオーバーヘッドは、ネイティブに実行するＣＰＵと比較して１０～２０サイクルから約２～４サイクルに減少させることができる。第３実施例の態様によれば、ＶＭＩ１２０が分岐条件は満足されたと仮定する場合、ＶＭＩ１２０は分岐オフセットを反映するためにバイトコードカウンタＢＣＣを更新し、該オフセットアドレスからの取り込みを開始し、該ＶＭＩのパイプラインを回復する。仮定が正しい場合、当該分岐を処理するために要する時間は約５サイクルに低減される。しかしながら、本発明のこの実施例による分岐予測が処理時間を全ての分岐に関して約３なるファクタにより低減する（１４又は１５サイクルから）と結論することはできない。何故なら、全ての分岐のうちの約１０％はキャッシュミスに終わるからである。

【００３９】

第４実施例においては、各ＣＢＩ３３０を処理するために、何れかの既知の分岐予測方法がＶＭＩ１２０との組み合わせで使用される。

【００４０】

本発明のシステム及び方法の利点の多くを述べたが、当業者であれば他の利点もあることを認識するであろう。例えば、最適なバス利用のために、ＣＰＵ１１０はネイティブ命令をバースト（典型的には、少なくとも４サイクルからなる）で読み取る。従って、ＣＰＵ１１０は、制御値をＶＭＩ１２０に戻す前に、全バーストを読み取らねばならない。ＶＭＩ１２０は、制御値取出コマンドを含むバーストを充填するために幾つかのＮＯＰを発生して、該ＶＭＩ１２０が投機的分岐を処理する間にＣＰＵ１１０が該取出コマンドを処理することを保証しなければならない。本発明の利点は、ＶＭＩ１２０が、ＣＰＵバーストを満たすためにプロセッサ時間をＮＯＰにより占有しなければならないというよりは、各命令の効果を逆転することができる限りにおいて投機的シーケンス３２０から次の命令を投機的にディスパッチすることによりＣＰＵ１１０及びＶＭＩ１２０のパイプラインを意味のある命令で満たし続ける点にある。

【００４１】

上記の点から、本発明が条件付き分岐仮想マシン命令を正確且つ効率的に処理するシステム及び方法を提供することが理解されるだろう。しかしながら、上記は本発明の実施例にのみ関するものであって、添付請求項に記載された本発明の趣旨及び範囲から逸脱すること無しに種々の変更をなすことができると理解されるべきである。

【図面の簡単な説明】

【００４２】

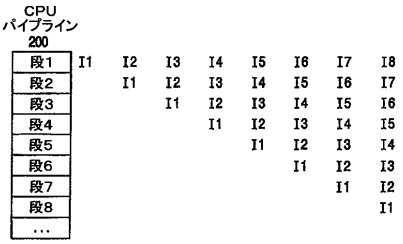
【図１】図１は、本発明の環境の実施例の機能的エレメントを示すブロック図である。

【図２】図２は、典型的なＣＰＵパイプラインにおける命令の処理を示す概念図である。

【図３】図３は、一例としてのバイトコード処理シーケンスを示す。

【図４】図４は、本発明の一実施例による方法のフローチャートである。

【図２】


CPU
パイプライン
200
段1
段2
段3
段4
段5
段6
段7
段8
...
I1 I2 I3 I4 I5 I6 I7 I8
I1 I2 I3 I4 I5 I6 I7
I1 I2 I3 I4 I5 I6
I1 I2 I3 I4 I5
I1 I2 I3 I4
I1 I2 I3
I1 I2
I1

【国際公開パンフレット】

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
10 April 2003 (10.04.2003)

PCT

(10) International Publication Number
WO 03/029961 A1

(51) International Patent Classification⁷: G06F 9/38, 9/318

(21) International Application Number: PCT/IB02/03646

(22) International Filing Date:
9 September 2002 (09.09.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
01402545.6 2 October 2001 (02.10.2001) EP

(71) Applicant: KONINKLIJKE PHILIPS ELECTRONICS N.V. [NL/NL]; Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(72) Inventor: LINDWER, Menno, M.; Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL).

(74) Agent: GROENENDAAL, Antonius, W., M.; Internationaal Octrooibureau B.V., Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL).

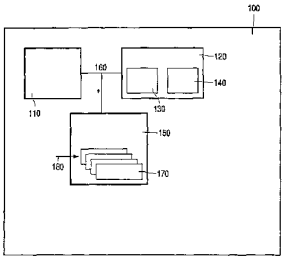
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
with international search report

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: SPECULATIVE EXECUTION FOR JAVA HARDWARE ACCELERATOR



(57) Abstract: Conditional branch bytecodes are processed by a Virtual Machine Interpreter (VMI) hardware accelerator that utilizes a branch prediction scheme to determine whether to speculatively process bytecodes while waiting for the CPU to return a condition control variable. In one branch prediction scheme, the VMI assumes the branch condition will be fulfilled if a conditional branch bytecode calls for a backward jump and that the branch condition will not be fulfilled if a conditional branch bytecode calls for a forward jump. In another branch prediction scheme the VMI makes an assumption only if a conditional branch bytecode calls for a backward jump. In yet another speculative execution scheme, the VMI assumes that the branch condition will be fulfilled whenever it processes a conditional branch bytecode. The VMI only speculatively processes bytecodes that are easily reversible, (for example, bytecodes representing simple stack manipulations), and suspends speculative processing of bytecodes upon encountering a bytecode that is not easily reversible. The assumptions made by the VMI are confirmed or invalidated upon receipt of the condition control variable. If an assumption is invalidated, any speculatively processed bytecodes are reversed.

WO 03/029961 A1

Speculative execution for java hardware accelerator

FIELD OF THE INVENTION

The present invention relates generally to computer programming languages, and more particularly to the translation and execution of a virtual machine language.

BACKGROUND OF THE INVENTION

Computer programming languages are used to create applications consisting of human-readable source code that represents instructions for a computer to perform. Before a computer can follow the instructions however, the source code must be translated into computer-readable binary machine code.

A programming language such as C, C++, or COBOL typically uses a compiler to generate assembly language from the source code, and then to translate the assembly language into machine language which is converted to machine code. Thus, the final translation of the source code occurs before runtime. Different computers require different machine languages, so a program written in C++ for example, can only run on the specific hardware platform for which the program was written.

Interpreted programming languages are designed to create applications with source code that will run on multiple hardware platforms. Java™ is an interpreted programming language that accomplishes platform independence by generating source code that is converted before runtime to an intermediate language known as "bytecode" or "virtual machine language." At runtime, the bytecode is translated into platform-appropriate machine code via interpreter software, as disclosed in U.S. Patent No. 4,443,865. To interpret each bytecode, interpreter software performs a "fetch, decode, and dispatch" (FDD) series of operations. For each bytecode instruction the interpreter software contains a corresponding execution program expressed in native central processing unit (CPU) instructions. The interpreter software causes the CPU to fetch or read a virtual machine instruction from memory, to decode the CPU address of the execution program for the bytecode instruction, and to dispatch by transferring control of the CPU to that execution program. The interpretation process can be time-consuming.

As disclosed in PCT Patent Application No. WO9918484 adding a preprocessor (a virtual machine interpreter (VMI)) between a memory and a CPU improves the processing of virtual machine instructions. In essence, the virtual machine is not a physical structure, but rather is a self-contained operating environment that interprets bytecode for the hardware platform by selecting the corresponding native machine language instructions that are stored within the VM or in the CPU. The native instructions are then supplied to and consecutively executed in the CPU of the hardware platform. A typical virtual machine requires 20-60 cycles of processing time per bytecode (depending on the quality and complexity of the bytecode) to perform an FDD series of operations. First, a VMI reads (fetches) a bytecode from memory. Next, the VMI looks up a number of properties of (decodes) the fetched bytecode. The properties accessed by the VMI determine how the bytecode will be processed into native instructions for execution in the CPU. While the CPU is executing an instruction, the VMI fetches and processes the next bytecode into CPU instructions. The VMI can process simple bytecodes in 1-4 cycles.

While interpreting a sequence of bytecodes, a virtual machine may encounter a bytecode that represents a conditional branch instruction, hereinafter referred to as a CBI. When a CBI is encountered, the VMI generates a sequence of native instructions that causes the CPU to determine whether the condition is fulfilled. The decision to execute the branch therefore depends on earlier computations, which in the VMI concept were executed in the CPU with the results remaining in CPU registers. For example, the Java™ bytecode "ifeq n" offsets the bytecode counter by "n", but only if the top of the stack is zero (i.e., the previous computation left the value 0 on the stack). The value of the branch condition (here, the top of the stack) must be retrieved and written to the control register of the VMI (which is reserved specifically for branch conditions). If the condition has been fulfilled, the CBI causes an update to the VMI bytecode counter (a jump) which alters the sequence of bytecodes to be executed. Typically, when one instruction is being processed in the VMI the next instructions to be processed are already in the VMI pipeline, so if an instruction results in a branch the bytecodes already in the VMI pipeline must be flushed. Additionally, the "pipelined" structure of processor hardware creates an inherent delay for transporting instructions and data between the instant that the instructions and/or data are dispatched to the processor and the instant when the processor effectively executes the instruction and/or processes the data. Specifically, because the typical CPU has a multistage (typically, 3 to 8 stages) pipeline the write operation will not be executed immediately after the instruction is issued. In the case of a CBI, additional delay occurs while the CPU determines whether the

condition is fulfilled and transfers the result of this determination to the VMI. If the value of the branch condition (the control value) indicates that the branch condition is fulfilled, several (depending on the size of the CPU pipeline) instructions will already have entered the CPU pipeline. To keep the CPU and instruction cache busy, a series of "no operation" (NOP) commands can be generated while waiting for the control value that indicates whether the condition is fulfilled. The control value is received while the CPU executes the next to the last NOP and the VMI generates the last NOP. After making the determination, the VMI's pipeline requires several cycles for the VMI to retrieve the bytecode representing the next instruction from the VMI's cache.

Other approaches speculatively execute potential branch instructions by predicting whether an instruction will result in a branch to another location. An example of this approach is directed to RISC (Reduced Instruction Set Computing) microprocessors, and provides a branch instruction bit to determine which conditional branches are "easy" to predict, and for those branches, uses software branch prediction to determine whether to execute the jump. Software branch prediction predicts branches using a software-controlled prediction bit. If the branch is determined to be "hard" to predict, the branch is predicted using hardware branch prediction (such as a branch prediction array). This approach discloses using a branch prediction scheme which predicts that a branch will be taken if the offset is less than zero (a backward branch) and that a branch will not be taken if the offset is greater than zero (a forward branch). A disadvantage of this approach is the consumption of processor resources for the making and updating the ease-of-prediction determination, which is based upon whether historical operation of the branch taken is important in determining whether the branch will be taken.

In another branch prediction approach, bits from the address of the potential branch instruction are compared to bits concatenated from a local branch history table and a global history register. The result of the comparison is used to read a branch prediction table. A disadvantage of this approach is the consumption of resources required to perform the concatenation and comparison operations and to store and access the branch prediction table. Furthermore, the approach does not disclose a means of correcting mispredictions. A similar methodology is disclosed in U.S. Patent No. 5,136,696, wherein a branch prediction is made by the branch cache based on the address of a potential branch instruction. According to that disclosure, where the prediction is wrong the corresponding instruction is invalidated but is executed anyway, so that the branch cache can be updated with the correct prediction in case the same instruction is encountered again. The CPU pipeline is flushed during the same

cycle as the branch cache update by invalidating all of the instructions in the first seven stages of the pipeline and loading the contents of a program counter register.

Because conditional branches occur frequently (approximately 10% of all virtual machine instructions) and are process-intensive when processed according to existing approaches which achieve high accuracies but consume processor resources, there is a need for a system of interpreting programming languages that accurately and efficiently executes instructions intended by conditional branch instruction bytecodes while increasing the processing speed.

SUMMARY OF THE INVENTION

The present invention fulfills the needs described above by configuring a virtual machine hardware accelerator (such as the VMI) such that when a conditional bytecode (CBI) is encountered the VMI performs branch prediction and elects whether to commence a speculative execution process. If elected, speculative execution continues as long as the speculatively executed bytecodes are easily reversible or until the prediction is confirmed. In most cases the prediction is correct, thereby enabling the VMI and CPU to continue operating along the sequence of bytecodes that was speculatively chosen after the prediction is confirmed. If the prediction is incorrect (and thus invalidated) the speculative processes executed in the VMI and the CPU are flushed and the VMI is returned to the state just before the branch. Efficiency is realized by combining the performance gains of virtual machine hardware acceleration technology with branch prediction schemes and potential speculative execution, and by speculatively executing instructions only as long as the effects can easily be reversed. Accuracy is achieved by correcting any misprediction. On average therefore, a conditional branch instruction (CBI) will introduce only a relatively small delay.

The present invention comports with more complex branch prediction methodologies (such as the hashing scheme and path history methodologies suggested in *Tuning Branch Predictors to Support Virtual Method Invocation in Java*, Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems (1999), pp. 217-228), as well as with schemes that make simple (but probable) predictions based upon the type of each potential branch instruction encountered.

Briefly, the present invention includes systems and methods for processing conditional branch virtual machine instructions, which in the exemplary embodiment of the present invention are generated by the Java™ programming language. At the programming level, Java™ source code is compiled into an intermediate language called bytecode.

Bytecode consists of virtual machine instructions that can be interpreted by a virtual machine for execution by a processor. According to the exemplary embodiment of the present invention, at runtime a virtual machine (in the exemplary embodiments, a VMI) is initialized. Parameters are initialized which enable the identification of certain properties of each bytecode. For example, bytecodes may be characterized as simple or complex, and can further characterized as conditional or recursive. A conditional bytecode is a virtual machine instruction that alters the sequence of bytecodes to be executed, but only if a branch condition is fulfilled.

The VMI proceeds to process each of a series of bytecodes into one or more native instructions. The VMI maintains a bytecode counter which is incremented after each bytecode is retrieved from an instruction memory. When a CBI is encountered, the VMI generates a sequence of native instructions that causes the CPU to determine whether the branch condition has been fulfilled, by retrieving the branch condition control value. According to the systems and methods of the exemplary embodiments of the present invention, rather than suspending processing until the control value retrieval is complete the VMI performs branch prediction and elects whether to perform speculative execution, i.e., whether to make an assumption as to whether the branch condition will be fulfilled. When an assumption is made, a bytecode sequence is speculatively executed until the assumption is confirmed or invalidated or until a bytecode that is not easily reversible is encountered.

In the first three embodiments of the present invention, the determination is based upon the type of branch that is designated by the CBI. According to the first embodiment, the VMI assumes that the branch condition is true if the CBI designates a backward branch, and updates the bytecode counter to jump to the next bytecode targeted by the branch. If the CBI designates a forward branch, the VMI assumes that the branch condition will not be fulfilled, and does not execute the branch. An alternative embodiment assumes that the branch condition is fulfilled if the CBI calls for a backward branch, but makes no assumption if the CBI calls for a forward branch. In another alternative embodiment, the VMI assumes that the branch condition is always fulfilled, and thus processes bytecodes along the speculative branch as long as the bytecodes along the speculative branch are easily reversible and until the assumption has been confirmed or invalidated.

In yet another embodiment a known branch prediction scheme is used to determine whether to speculatively execute a branch, but the VMI only speculatively processes bytecodes that are easily reversible, and then only until the assumption has been

confirmed or invalidated. This methodology is described in detail below with respect to implementation with a VMI, but can be implemented with more branch prediction or speculative execution schemes.

According to the embodiments of the present invention, when an assumption has been made the VMI continues to dispatch native instructions to the CPU which are translations of successive bytecodes either along the speculatively executed branch or along the original sequence of bytecodes as long as these native instructions are easily reversible, such as instructions that involve stack manipulations (such as stack pushes). When an instruction that is not easily reversed is encountered, a series of "no operation" (NOP) commands are generated and dispatched by the VMI to keep the CPU busy until the control value is received. If the subsequently received control value indicates that the assumption is correct, the VMI thus continues to operate on the bytecodes present in the VMI pipeline and the CPU continues to operate on the native instructions in the CPU pipeline. If the subsequently received control value indicates that the assumption is incorrect (i.e., there has been a misprediction) the CPU is caused to return to the state just before the branch, thereby reversing the speculatively executed easily reversible bytecodes. Thus, reversal of a misprediction due to an incorrect assumption requires returning both the VMI and the CPU to their respective states just prior to the branch and is easily accomplished according to the characteristics of the speculatively executed bytecodes. For instance, in an embodiment of the present invention "easily reversible bytecode" is defined as a bytecode that performs only stack manipulations or that does not modify any state outside the VMI, so execution of a sequence of such bytecodes can be reversed by resetting the bytecode counter and the register stack pointer in the VMI. Other definitions of "easily reversible" can be implemented according to the present invention and some of those definitions may require that the pipelines of the VMI and the CPU be flushed to reverse speculatively executed bytecodes.

Another aspect of the present invention is the system for executing virtual machine instructions from an interpreted language such as Java™. The system includes a processor (the CPU) and a preprocessor (the VMI), an instruction memory, and a translator (a JVM). The processor contains and is configured to execute hardware-specific instructions, hereinafter referred to as native instructions. The preprocessor is a virtual machine, for example a VMI, configured to fetch bytecode from the instruction memory and to translate the bytecode into native CPU instructions. The VMI includes a control register, a bytecode counter BCC, an assumption variable register, and a speculation mode switch history. In the exemplary embodiment of the present invention, the VMI is configured to dispatch native

instructions called for by each bytecode to the CPU for processing. Where the processed bytecode is a CBI, the native instructions dispatched by the VMI cause the CPU to send a control value that indicates whether a branch is to be executed. The VMI is also configured to update the BCC to execute the jump called for by a CBI when it has been speculated or confirmed that the branch condition is fulfilled. While waiting for a control value, the VMI is configured to speculatively process or decline to process a branch based upon a branch prediction scheme according to the methods listed as the embodiments above. The VMI is configured to update the value of an assumption variable upon making or declining to make an assumption, by storing an assumed control value ACV. The VMI is configured to maintain a speculative mode switch history, which is updated when a sequence is speculatively executed or when speculative execution ceases, and is purged when a control value is received. After making the assumption that a branch should be speculatively executed, the VMI is configured to identify and process bytecodes that are reversible, to receive the control value, and to reverse speculatively executed bytecodes if the VMI subsequently receives a control value that indicates that the earlier assumption was incorrect.

Although it is possible to implement the methods of the present invention to process various types of conditional branch instructions, the exemplary embodiment of the present invention is directed to the processing of Java™ conditional branch bytecodes.

The present invention can be implemented in systems that execute Java™ bytecode using virtual machines, such as JVMs made by Sun Microsystems. However, the invention can also be implemented using other Java™ virtual machines such as the Microsoft Virtual Machine, and is also applicable to systems that execute other interpreted languages such as Visual Basic, dBASE, BASIC, and .NET.

Additional objects, advantages and novel features of the invention will be set forth in part in the description which follows, and in part will become more apparent to those skilled in the art upon examination of the following, or may be learned by practice of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and form part of the specification, illustrate the present invention when viewed with reference to the description, wherein:

FIG. 1 is a block diagram that shows the functional elements of an exemplary embodiment of the environment of the present invention.

FIG. 2 is a chart that shows processing of instructions in the typical CPU pipeline.

FIG. 3 illustrates an exemplary bytecode processing sequence.

FIG. 4 charts the flow of a method according to an exemplary embodiment of the present invention.

DESCRIPTION OF PREFERRED EMBODIMENTS

As required, detailed embodiments of the present invention are disclosed herein; however, it is to be understood that the disclosed embodiments are merely exemplary of the invention that may be embodied in various and alternative forms. The figures are not necessarily to scale; some features may be exaggerated or minimized to show details of particular components. Therefore, specific structural and functional details disclosed herein are not to be interpreted as limiting, but merely as a basis for the claims and as a representative basis for teaching one skilled in the art to variously employ the present invention.

Referring now in detail to an exemplary embodiment of the present invention, which is illustrated in the accompanying drawings in which like numerals designate like components, FIG. 1 is a block diagram of the exemplary embodiment of the environment of the present invention. The basic components of the environment are a hardware platform 100 which includes a processor 110, a preprocessor 120, and an instruction memory 150 which are all connected by a system bus 160. The preprocessor 120 includes control register 130 and a translator 140. A hardware platform 100 typically includes a central processing unit (CPU), basic peripherals, an operating system (OS). The processor 110 of the present invention is a CPU such as MIPS, ARM, Intel x86, PowerPC, or SPARC type microprocessors, and contains and is configured to execute hardware-specific instructions, hereinafter referred to as native instructions. In the exemplary embodiment of the present invention, the translator 140 is a Java™ virtual machine (JVM), such as the KVM by Sun Microsystems. The instruction memory 150 contains virtual machine instructions, for example, Java™ bytecode 170. The preprocessor 120 in the exemplary embodiment is the Virtual Machine Interpreter (VMI) disclosed in WO9918486, and is configured to fetch a virtual machine instruction (for example, a bytecode 170) from the instruction memory 150 and to translate the virtual machine instruction into a sequence of native CPU instructions. The VMI 120 is a peripheral on the bus 160 and may act as a memory-mapped peripheral, where a predetermined range of CPU addresses is allocated to the VMI 120. The VMI 120

manages an independent virtual machine instruction pointer 180 indicating the current (or next) virtual machine instruction in the instruction memory 150. The VMI also manages a speculation mode switch history and an assumption variable (not shown).

Referring now to FIG. 2 and according the exemplary embodiment of the present invention, the processor 110 includes an multi-stage pipeline 200 such that execution of machine instructions occurs in parallel and lockstep fashion during consecutive clock cycles. In other words, when a first instruction I1 enters stage two of the pipeline, the following instruction I2 enters stage one, and so on. Ideally, the instructions continue to enter the pipeline in consecutive cycles thereby enabling the concurrent processing of eight instructions where one instruction is completely executed every clock cycle. Typically however, execution of the average instruction actually requires more than one clock cycle.

As an example of the operation of the present invention, the VMI 120 proceeds to translate each of a first series 310 of bytecodes 170 into one or more native instructions. Referring now to FIG. 3, bytecodes B0 through B2 are non-conditional, so the VMI 120 simply fetches B0 through B2 from the instruction memory 150, selects the native instruction or instructions defined for each bytecode 170, and supplies the instruction(s) to the processor 110 for execution. Bn is a conditional branch instruction (CBI 330), therefore if the branch condition is fulfilled Bn causes the VMI to jump from the first sequence 310 to a bytecode Br in a sequence 320. While the branch condition is being evaluated (i.e., the VMI 120 waits for a control value to be returned from the CPU 110), a prediction is made according branch prediction scheme as to whether the branch condition will be fulfilled and the VMI 120 elects whether to speculatively execute instructions until the prediction can be verified. Branch prediction schemes are heuristic processes of varying complexities that provide predictions as to whether a branch condition will be fulfilled and possibly a statistical assessment of accuracy of the prediction. According to one embodiment of the present invention, the VMI 120 implements a branch prediction scheme that assumes that if the CBI 330 (Bn) calls for a backward branch (i.e., a negative offset to the bytecode counter BCC), the branch condition is fulfilled and the VMI 120 executes the jump to bytecode Br. The VMI 120 concurrently dispatches the sequence of native instructions that causes the CPU 110 retrieve the control value that indicates whether the branch condition has been fulfilled. According to the branch prediction scheme of another embodiment, if the CBI 330 (Bn) calls for a forward branch (i.e., a positive offset to the BCC), the VMI 120 assumes that the branch condition is not fulfilled and does not execute the jump. According to the branch prediction

scheme of yet another embodiment, if the CBI 330 calls for a branch at all, the VMI 120 assumes that the branch condition is fulfilled and executes the jump.

According an exemplary embodiment of the present invention, if the branch prediction scheme results in speculative execution of a sequence 320 of bytecodes, the impact of an inaccurate prediction is minimized by speculatively executing only easily reversible bytecodes 170. Ease of reversal is determined when the bytecode 170 is decoded (i.e., its properties are accessed) by the VMI 120. The CPU 110 speculatively executes each dispatched instruction. For example, in many cases a branch is followed by a stack push operation (i.e. a constant or variable gets pushed on the stack), which can easily be reversed by resetting the stack pointer value (which is maintained in the VMI 120) to its state prior to the branch. Such native instructions (commonly referred to as "stack pushes") can be speculatively executed until the control value has been received, because reversing the speculative execution is accomplished by merely resetting a register, and thus the negative effects of a misprediction are negligible.

Thus, when the VMI 120 assumes that a branch condition will be fulfilled, processing of bytecodes Br through Bz (the speculative branch) subsequently continues in the second sequence 320 unless the VMI 120 encounters a bytecode along the speculative branch that represents a sequence of native instructions that cannot be easily reversed. When a bytecode 170 along the speculative branch cannot easily be reversed, speculative execution is suspended. If the VMI 120 subsequently receives a control value that indicates that the branch condition has not been fulfilled, the VMI 120 resets the BCC and the register stack pointer to reverse the jump (and if necessary, issues a native instruction to the CPU 110 to purge the CPU pipeline 200). If the VMI 120 assumes that a branch condition will not be fulfilled, the BCC is incremented and the branch is speculatively not processed. While speculatively processing non-branch bytecodes, the VMI 120 continuously checks whether the received control value has been written to its control register 130.

As shown in block 410 of FIG. 4, the VMI 120 increments a virtual machine counter BCC before proceeding in block 420 to fetch each bytecode 170 from the instruction memory 150. In block 430, the VMI 120 decodes each bytecode 170 by accessing the properties for the bytecode 170. Because the speculation mode switch SMS is off when the first bytecode 170 of a sequence has been decoded, the VMI 120 (in block 435) dispatches the sequence of native instructions that corresponds to the bytecode 170. If in block 445 it is determined that the bytecode 170 is not a CBI 330, the method returns to block 410 where the virtual machine counter (BCC) 180 is incremented, and the fetching process resumes (block

420). However, if it is determined in block 445 that the bytecode 170 is conditional, the native instructions dispatched in block 435 will represent a control value retrieval process within block 450. The native instructions that constitute the control value retrieval process will cause the CPU 110 to send a control value that indicates whether the branch called for by the CBI 330 is to be executed (i.e., whether to jump to a target bytecode 170 that is outside of the current sequence of bytecodes being executed), and to write that control value to the control register 130. Concurrently with the control value retrieval process, the VMI 120 elects (in block 470) whether to make an assumption that the branch condition will or will not be fulfilled. The election is based upon the particular branch prediction or speculative execution scheme in use.

If the VMI 120 elects to make an assumption as to the fulfillment of the branch condition, the speculation mode switch (SMS) is turned on and an assumed control value (ACV) is stored in a VMI register (the assumption variable 195). In block 475, the BCC is updated (to reflect the jump called for by the CBI 330) if the branch is speculatively taken. Regardless of whether the branch is speculatively taken, the method proceeds to block 410 where the BCC is incremented. The next bytecode 170 along the speculatively elected sequence of bytecodes is processed if it is reversible (e.g., the native instructions corresponding to the next bytecode 170 constitute a simple stack manipulation). The determination of ease of reversibility is based upon the properties of the bytecode 170. Easily reversible bytecodes typically represent operations that take place on stack positions above the top-of-stack just after the CBI 330 translation and that do not modify the state of the system (in particular, outside the VMI 120). The bytecode 170 is fetched in block 420 and decoded in block 430. The SMS is on because the VMI 120 elected to make an assumption (the sequence is being speculatively executed), so if the control value has not yet been returned (block 485) and the decoded properties indicate the fetched bytecode 170 is easily reversible, the native instructions corresponding to the fetched bytecode are dispatched in block 435. Then as long as the fetched bytecode 170 is not another CBI 330 (block 445), the BCC is incremented (block 410) and the next bytecode along the speculative sequence is fetched and decoded (blocks 420 and 430). Thus, the processing of a speculative sequence loops through blocks 410, 420, 430, 432, 485, 486, 460, and 435 as long as each next bytecode 170 is easily reversible, and as long as the branch condition has not been received.

If in block 460 a bytecode 170 that is not easily reversible is encountered during the processing of a speculative sequence, speculation is discontinued in block 461 by switching the SMS off, which causes the VMI 120 to wait for receipt of the control value in

block 485. The received control value RCV is stored in the control register 130 of the VMI 120. In block 487, the branch condition is evaluated. If the RCV indicates that the branch condition was fulfilled and the SMS history indicates that current sequence was being speculatively processed (the SMS history 190 shows that the SMS has been on at some point during the current sequence) in block 494 the assumed control value ACV stored as the assumption variable 195 is compared to the received control value RCV stored in the control register 130 to determine whether the assumption made by the VMI 120 was correct (i.e., whether the speculative sequence should have been processed). If the assumption was correct (i.e., ACV = RCV) then the speculative execution of the current branch sequence has been confirmed. In other words, the VMI 120 correctly assumed that the branch should be taken. The BCC is incremented in block 410 and fetching resumes. If the assumption was incorrect then the VMI 120 failed to process the branch, and the VMI 120 and the CPU 110 are flushed to reverse the speculatively processed non-branch sequence of bytecodes, the method updates the BCC to reflect the jump in block 498 and then returns to block 410 to increment the BCC and process the next bytecode along the branch sequence.

If the RCV indicates that the branch condition was not fulfilled and the SMS history 190 indicates that current sequence was being speculatively processed, in block 494 the assumed control value ACV is compared to the received control value RCV to determine whether the assumption made by the VMI 120 was correct. If the assumption was correct then the speculative execution of the current non-branch sequence has been confirmed. In other words, the VMI 120 correctly assumed that the branch should not be taken. The BCC is incremented in block 410 and fetching resumes. If the assumption was incorrect then the branch should not have been taken, and all bytecodes along the speculatively processed branch are reversed by resetting the BCC and the stack register pointer in block 495. If necessary, the pipelines of the VMI 120 and the CPU 110 are flushed in block 495.

The SMS history 190 is reset whenever a control value is received because the branch condition has been resolved thus confirming or purging any speculative processes.

If the VMI 120 elects in block 470 not to make an assumption as to the fulfillment of the branch condition, the VMI 120 waits until the actual control value is received by checking in block 485 for receipt of the control value. Because the SMS remains off, block 486 returns to block 485 to continue checking until the control value is received. When the control value is received, the method proceeds to block 487, where the branch condition is evaluated. If the control value indicates that the branch condition was fulfilled, because no speculation has occurred during execution of the bytecode sequence, the method

proceeds to block 498 where the BCC is updated to reflect the jump called for by the CBI 330 and then bytecode fetching is resumed in block 410. If the control value indicates that the branch condition was not fulfilled, the method returns to block 410 to resume bytecode fetching. In this manner, if the VMI 120 elects not to make an assumption the VMI 120 processes the CBI 330 normally (taking the branch only if the branch condition is fulfilled).

As discussed above, the decision whether to make an assumption regarding the fulfillment of the branch condition is based upon the branch prediction scheme implemented, as illustrated below.

Branch Prediction Schemes

	Backward Branch	**Forward Branch**
First embodiment	Assume taken	Assume not taken
Second embodiment	Assume taken	No assumption
Third embodiment	Assume taken	Assume taken
Fourth embodiment	Depends on prediction scheme	

In a first embodiment of the present invention, the VMI 120 utilizes a speculative execution scheme that assumes that backward branches will be taken, and that forward branches will not be taken. In a second and alternative embodiment, the VMI 120 speculatively executes only backward branches, and makes no assumption regarding forward branches. In a third and alternative embodiment, the VMI 120 utilizes a simpler speculative execution scheme that assumes that all branches will be taken. Furthermore, according to a fourth and alternative embodiment the VMI 120 utilizes a branch prediction scheme to determine whether the branch is to be taken. According to any of the forgoing embodiments, speculative execution only proceeds for bytecodes that are easily reversible.

According to the first embodiment of the present invention, the VMI 120 always makes an assumption regarding the outcome of the branch condition represented by a CBI 330, but bases the speculation upon the type branch called for by the CBI 330. If the CBI 330 calls for a forward branch (BCC offset > 0) the VMI 120 assumes that the branch condition will not be fulfilled and speculatively continues to process a non-branch bytecode sequence as long as it is easily reversible. If the CBI 330 calls for a backward branch (BCC offset < 0) the VMI 120 assumes that the branch condition will be fulfilled and speculatively processes the branch bytecode sequence as long as it is easily reversible.

According to the second embodiment of the present invention, the VMI 120 makes an assumption regarding the outcome of the branch condition represented by a CBI

330 only if the CBI 330 calls for a backward branch. Where the CBI 330 calls for a forward branch the VMI 120 processes the branch condition normally, i.e., the VMI 120 waits for receipt of the control value and processes the appropriate sequence of bytecodes accordingly.

In a third embodiment of the present invention, a scheme speculates that the branch is always taken when a CBI is encountered. When processing RISC code this speculative execution scheme is accurate approximately 70% of the time because branches usually occur at the bottom of a loop, and loops are typically executed repeatedly. The branch prediction improves the overall RISC code processing performance of VMI by approximately 35%. According to Amdahl's law, when processing Java code this simple prediction will increase bytecode processing speed by 7-20%. Using branch prediction and speculative operation, the overhead of branches as compared to a natively executing CPU can be reduced from 10 to 20 cycles to approximately 2 to 4 cycles. According to an aspect of the third embodiment, when the VMI 120 assumes that a branch condition has been fulfilled the VMI 120 updates the bytecode counter BCC to reflect the branch offset, starts fetching from the offset address and recovers its pipeline. If the assumption is correct, the time required to process that branch is reduced to about 5 cycles. However, it cannot be concluded that branch prediction according to this embodiment of the present invention would reduce the processing time for every branch by an approximate factor of three (from 14 to 5 cycles) because approximately 10% of all branches results in cache misses.

In a fourth embodiment, any known branch prediction scheme is utilized in combination with the VMI 120 to process each CBI 330.

Many of the advantages of the systems and methods of the present invention are described herein, although those skilled in the art will recognize other advantages exist. For example, for optimal bus usage the CPU 110 reads native instructions in bursts (typically consisting of at least 4 cycles). Therefore the CPU 110 must read an entire burst before it will send the control value back to the VMI 120. The VMI 120 must generate some NOPs to fill the burst containing the control value retrieval command so as to ensure that the CPU 110 processes the retrieval command while the VMI 120 processes a speculative branch. An advantage of the present invention is that when a the VMI 120 keeps the CPU 110 and VMI 120 pipelines filled with meaningful instructions by speculatively dispatching the next instructions from the speculative sequence 320 as long as the effects of each instruction can be reversed, rather than having to occupy processor time with NOPs to fill the CPU burst.

In view of the foregoing, it will be appreciated that the present invention provides a system and a method for accurate and efficient processing of conditional branch

virtual machine instructions. Still, it should be understood that the foregoing relates only to the exemplary embodiments of the present invention, and that numerous changes may be made thereto without departing from the spirit and scope of the invention as defined by the following claims.

CLAIMS:

1. A method of processing virtual machine instructions, the method comprising the steps of:

identifying a subset of virtual machine instructions the processing of which is conditionally dependent upon the value of a condition control variable;

fetching and processing a range of the virtual machine instructions into native instructions executable by a processor, wherein a virtual machine instruction counter is incremented after processing and dispatching to the processor for execution a virtual machine instruction of said range, and continuing until a member of said subset of the virtual machine instructions is encountered;

initiating a control value retrieval process upon encountering said member, by dispatching native instructions represented by said member to the processor for execution; and

processing a speculative sequence of virtual machine instructions when an assumption has been made for an assumed value of a real value of the condition control variable, by:

switching a speculative mode switch on;

updating the virtual machine instruction counter according to the assumed value, so as to process into native instruction each virtual machine instruction of said speculative sequence of virtual machine instructions, incrementing the virtual machine instruction counter and dispatching the native instructions after each virtual machine instruction is processed, until the real value of the condition control variable has been received;

comparing the real value to the assumed value; and

reversing the processing of the speculative sequence of virtual machine instructions, purging a speculation mode history, and reversing the update to the virtual machine counter, if the real value is not equal to the assumed value; or

switching said speculation mode switch off and purging said speculation mode history, if the real value is equal to the assumed value;

delaying further fetching and processing of the range of the virtual machine instructions until the real value of the condition control variable is received and then processing the next virtual machine instruction according to the real control value, if no assumption is made for the assumed value of the real value of the condition control variable.

2. The method of Claim 1, wherein fetching and processing of said range of the virtual machine instructions into native instructions executable by a processor is accomplished by a Virtual Machine Interpreter (VMI) virtual machine hardware accelerator.

3. The method of Claim 1, wherein processing said speculative sequence of virtual machine instructions further comprises:

processing each virtual machine instruction of said speculative sequence only if the virtual machine instruction is easily reversible; and

suspending processing of said speculative sequence of virtual machine instructions until the real value of the condition control variable is received, if the next virtual machine instruction does not represent a stack manipulation.

4. The method of Claim 3, wherein suspending processing of said speculative sequence further comprises dispatching to the processor a series of "no operation" instructions until the real value of the condition control variable is received.

5. The method of Claim 2, wherein determining whether to make an assumption for the assumed value of the real value of the condition control variable further comprises:

determining a displacement to the virtual machine counter that is called for by said member of said identified subset of virtual machine instructions; and

setting the assumed value of the condition control variable to indicate that the branch condition is fulfilled, if the member of said identified subset of virtual machine instructions calls for a negative displacement (offset < 0) of the virtual machine counter.

6. The method of Claim 5 further comprising setting the assumed value of the condition control variable to indicate that the branch condition is not fulfilled, if the member of said identified subset of virtual machine instructions calls for a positive displacement (offset > 0) of the virtual machine counter.

7. The method of Claim 5 further comprising making no assumption for the real value of the condition control variable, if the member of said identified subset of virtual machine instructions calls for a positive displacement (offset > 0) of the virtual machine counter.

8. The method of Claim 2, wherein determining whether to make an assumption for the assumed value of the real value of the condition control variable further comprises setting the assumed value of the condition control variable to indicate that the branch condition is fulfilled.

9. The method of Claim 2, wherein determining whether to make an assumption for the assumed value of the real value of the condition control variable further comprises setting the assumed value of the condition control variable according to a branch prediction scheme.

10. The method of Claim 2, wherein determining whether to make an assumption for the assumed value of the real value of the condition control variable further comprises setting the assumed value of the condition control variable according to a speculative execution scheme.

11. A method of processing virtual machine instructions, the method comprising:

fetching and processing a range of the virtual machine instructions into native instructions executable by a processor, and continuing until a conditional branch instruction is encountered;

initiating a control value retrieval process by dispatching a native instruction represented by the conditional branch instruction to the processor for execution, upon encountering the conditional branch instruction;

determining whether to make an assumption as to a value of a condition control variable; and

processing a speculative sequence of virtual machine instructions when an assumption is made as to the value of the condition control variable, by:

processing a virtual machine instruction of said speculative sequence of virtual machine instructions into a preliminary native instruction, only if the virtual machine instruction is easily reversible; and

suspending processing of said speculative sequence of virtual machine instructions until the value of the condition control variable is received, if the virtual machine instruction does not represent a stack manipulation,

delaying further fetching and processing of the range of the virtual machine instructions until the value of the condition control variable is received and then processing the next virtual machine instruction according to the received value, if no assumption is made as to the value of the condition control variable.

12. The method of Claim 11, wherein processing the speculative sequence of virtual machine instructions further comprises:

comparing the value of the received condition control variable to an assumed value;

reversing the processing of the speculative sequence of virtual machine instructions, purging a speculation mode history, if the value of the condition control variable is not equal to the assumed value; and

switching said speculation mode switch off, and purging said speculation mode history, if the value of the condition control variable is equal to the assumed value.

13. An apparatus for processing virtual machine instructions, comprising:

a processor (110) having a native instruction set and configured to execute native instructions;

an instruction memory (150), configured to store virtual machine instructions;

a preprocessor (120), configured to fetch virtual machine instructions from the instruction memory and to process the fetched virtual machine instructions into native instructions executable by the processor (110), to identify conditional virtual machine instructions, to determine whether to speculatively process virtual machine instructions while the value of a condition control variable is being sent by the processor, and to confirm or reverse said speculatively processed virtual machine instructions upon receipt of the condition control variable, and further comprising:

a control register (180), configured to store the value of the condition control variable received from the processor;

a virtual machine instruction counter, configured to indicate the next virtual machine instruction to be processed;

an assumption variable register, configured to store the value of an assumed control variable;

a speculation mode bit, configured to indicate whether the preprocessor has made an assumption regarding the value of the condition control variable; and

a speculation mode history, configured to store speculation mode bits while the value of the control variable is being sent.

14. The apparatus of Claim 13, wherein the preprocessor (120) is a Virtual Machine Interpreter (VMI) virtual machine hardware accelerator.

15. The apparatus of Claim 13, wherein the preprocessor (120) is further configured to speculatively process only virtual machine instructions that are easily reversible, and to suspend speculative processing of virtual machine instructions until the value of the condition control variable is received when a virtual machine instruction that is not easily reversible is encountered.

16. The apparatus of Claim 15, wherein the preprocessor (120) is further configured to dispatch a series of "no operation" instructions while speculative processing is suspended and until the value of the condition control variable is received.

17. The apparatus of Claim 15, wherein the preprocessor (120) is further configured to speculatively process only virtual machine instructions that represent stack manipulations, and to suspend speculative processing of virtual machine instructions until the value of the condition control variable is received when a virtual machine instruction that is not a stack manipulation is encountered.

18. The apparatus of Claim 17, wherein the preprocessor (120) is further configured to dispatch a series of "no operation" instructions while speculative processing is suspended and until the value of the condition control variable is received.

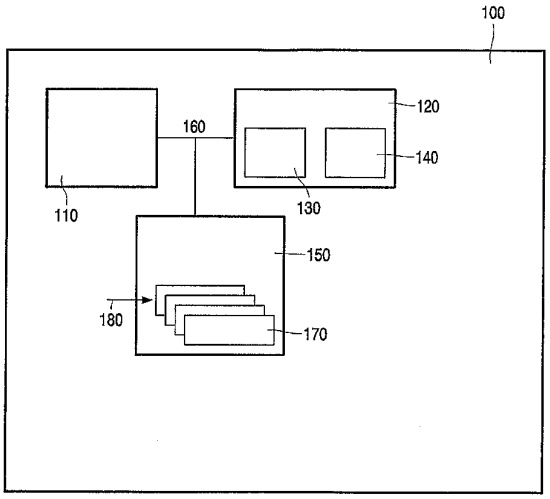


FIG. 1

CPU pipeline 200

Stage 1	I1	I2	I3	I4	I5	I6	I7	I8
Stage 2		I1	I2	I3	I4	I5	I6	I7
Stage 3			I1	I2	I3	I4	I5	I6
Stage 4				I1	I2	I3	I4	I5
Stage 5					I1	I2	I3	I4
Stage 6						I1	I2	I3
Stage 7							I1	I2
Stage 8								I1
...								

FIG. 2

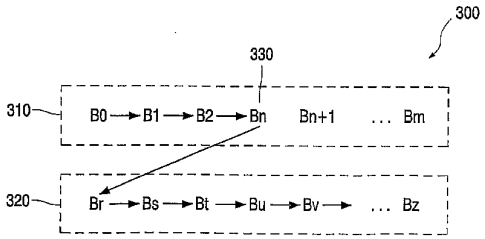


FIG. 3

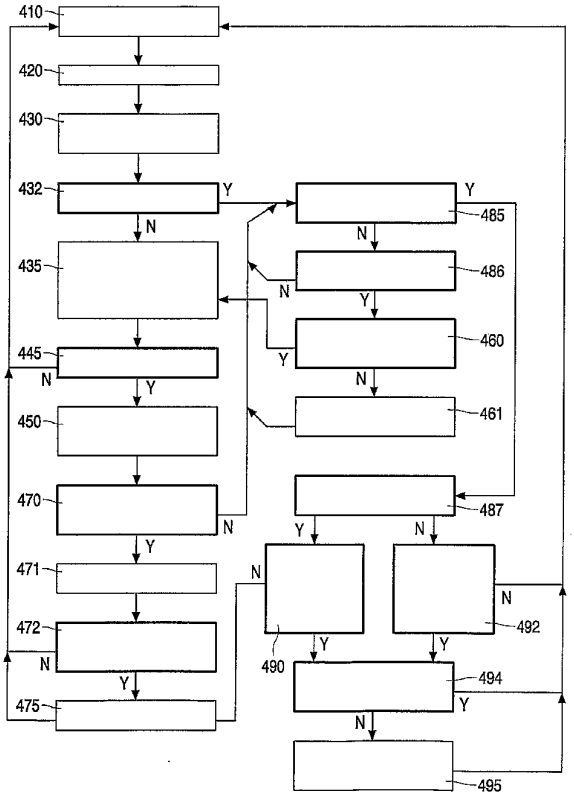

410
420
430
432
435
445
450
470
471
472
475
485
486
460
461
487
490
492
494
495
Y
N

FIG. 4

【国際調査報告】

INTERNATIONAL SEARCH REPORT

International Application No
PCT/IB 02/03646

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/38 G06F9/318

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 01 27752 A (TRANSMETA CORP) 19 April 2001 (2001-04-19) the whole document	1,11,13
A	US 6 167 509 A (SITES RICHARD LEE ET AL) 26 December 2000 (2000-12-26) column 16, line 59 - line 65	5-7
A	US 5 450 560 A (BRIDGES JEFFREY T ET AL) 12 September 1995 (1995-09-12) column 23, line 58 -column 24, line 9	3,11,15

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
17 December 2002	27/12/2002

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Moraiti, M

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/IB 02/03646

Patent document cited in search report		Publication date	Patent family member(s)		Publication date
WO 0127752	A	19-04-2001	CN	1379875 T	13-11-2002
			EP	1230594 A1	14-08-2002
			WO	0127752 A1	19-04-2001
US 6167509	A	26-12-2000	CA	2045791 A1	30-12-1991
			DE	69129881 D1	03-09-1998
			DE	69129881 T2	01-04-1999
			EP	0463977 A2	02-01-1992
			JP	2951064 B2	20-09-1999
			JP	6103067 A	15-04-1994
			KR	190252 B1	01-06-1999
US 5450560	A	12-09-1995	NONE		

Form PCT/ISA/210 (patent family annex) (July 1992)

フロントページの続き

(81)指定国 AP(GH,GM,KE,LS,MW,MZ,SD,SL,SZ,TZ,UG,ZM,ZW),EA(AM,AZ,BY,KG,KZ,MD,RU,TJ,TM),EP(AT,BE,BG,CH,CY,CZ,DE,DK,EE,ES,FI,FR,GB,GR,IE,IT,LU,MC,NL,PT,SE,SK,TR),OA(BF,BJ,CF,CG,CI,CM,GA,GN,GQ,GW,ML,MR,NE,SN,TD,TG),AE,AG,AL,AM,AT,AU,AZ,BA,BB,BG,BR,BY,BZ,CA,CH,CN,CO,CR,CU,CZ,DE,DK,DM,DZ,EC,EE,ES,FI,GB,GD,GE,GH,GM,HR,HU,ID,IL,IN,IS,JP,KE,KG,KP,KR,KZ,LC,LK,LR,LS,LT,LU,LV,MA,MD,MG,MK,MN,MW,MX,MZ,NO,NZ,OM,PH,PL,PT,RO,RU,SD,SE,SG,SI,SK,SL,TJ,TM,TN,TR,TT,TZ,UA,UG,UZ,VC,VN,YU,ZA,ZM,ZW

(74)代理人 100122769
弁理士 笛田 秀仙
(72)発明者 リンドワー メンノ エム
オランダ国 ５６５６ アーアー アインドーフェン プロフ ホルストラーン ６

Ｆターム(参考) 5B013 BB01 BB18
5B081 AA09 DD02

【要約の続き】
化される。仮定が無効化された場合は、如何なる投機的に処理されたバイトコードも元に戻される。