(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2005-504376
(P2005-504376A)
(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	5B081

審査請求 未請求　予備審査請求 未請求　（全 39 頁）

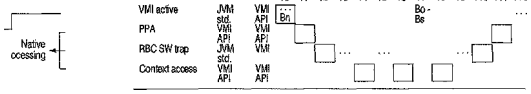
(21)出願番号	特願2003-531316(P2003-531316)
(86)(22)出願日	平成14年9月6日(2002.9.6)
(85)翻訳文提出日	平成15年12月25日(2003.12.25)
(86)国際出願番号	PCT/IB2002/003695
(87)国際公開番号	WO2003/027842
(87)国際公開日	平成15年4月3日(2003.4.3)
(31)優先権主張番号	01402455.8
(32)優先日	平成13年9月25日(2001.9.25)
(33)優先権主張国	欧州特許庁(EP)

(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】仮想マシンインタープリタ（ＶＭＩ）加速ハードウェアのためのソフトウェアサポート

(57)【要約】
ソフトウェアトラップ方法をサポートする仮想マシン命令を処理するシステム及び方法である。アプリケーションプログラミングインターフェース（ＡＰＩ）は、再帰的仮想マシン命令の処理を仮想マシンハードウェアから削除し、代わりに再帰的仮想マシン命令を、ソフトウェアを使用して処理するようなソフトウェアトラップのための追加の機能を規定する。該追加の機能は、再帰的仮想マシン命令を処理するためのコンテキストのコンフィギュレーション（該コンフィギュレーションは上記仮想マシン命令がＣＰＵレジスタにアクセスして、必要に応じて変数の値を取り込み及び変更するのを可能にする）、再帰的仮想マシン命令の処理が完了した場合の上記コンフィギュレーションされたコンテキストの解放、及び非再帰的仮想マシン命令の処理のために仮想マシンへの制御の返還を含む。



【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、
パラメータを初期化して前記仮想マシン命令のうちの特定の特徴の組を持つ部分集合を識別するステップと、
前記仮想マシン命令のうちの前記識別された部分集合のメンバに出会うまで、一連の前記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に翻訳するステップと、
前記仮想マシン命令のうちの前記識別された部分集合のメンバに出会った場合に、前記一連の仮想マシン命令の翻訳を中断するステップと、
インターフェースを実施化するステップであって、該インターフェースが、
前記仮想マシン命令のうちの前記識別された部分集合の前記メンバにより表されるネイティブプロセッサ命令を実行する前にプリアンブルを実行し、
前記プロセッサから、前記仮想マシン命令のうちの前記識別された部分集合の前記メンバにより表される前記命令に対応するネイティブ機能を取り込み、
前記取り込まれたネイティブ命令を実行し、
前記仮想マシン命令のうちの前記識別された部分集合の前記メンバにより表される前記ネイティブ命令を実行した後にポスタンブルを実行し、
仮想マシンにおいて前記一連の仮想マシン命令の翻訳を再開し、ここで、前記仮想マシン命令における前記識別された部分集合の他のメンバに出会うまで実行を継続する、
ような一群のネイティブプロセッサ命令を規定するようなステップと、
前記一群のネイティブプロセッサ命令を実行するステップと、
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、
メソッドを構成する取り込まれたネイティブ命令を実行する前にポスタンブルを実行するステップであって、該ポスタンブルが前記プロセッサにおけるレジスタ内の変数の値を変更して、前記メソッドを実行するために要するコンテキスト変更を実施するようなネイティブ命令を有するステップと、
メソッドを構成する取り込まれたネイティブ命令を実行した後にプリアンブルを実行するステップであって、該プリアンブルが、前記プロセッサにおけるレジスタからの値を前記メソッドが実行された後に実行されるネイティブ命令に対しアクセス可能な変数に書き込むネイティブ命令を有するようなステップと、
を更に有してい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、前記プリアンブル及び前記ポスタンブルが前記仮想マシンにより発生されることを特徴とする方法。
【請求項４】
請求項１に記載の方法において、前記プリアンブル及び前記ポスタンブルがプロセッサメモリから取り込まれることを特徴とする方法。
【請求項５】
請求項１に記載の方法において、前記プリアンブルは前記プロセッサにおけるレジスタからの値を、前記仮想マシン命令のうちの前記識別された部分集合により表される命令に対してアクセス可能な変数に書き込むようなネイティブ命令を有していることを特徴とする方法。
【請求項６】
請求項１に記載の方法において、前記ポスタンブルは前記プロセッサにおけるレジスタ内の変数の値を、前記仮想マシン命令のうちの前記識別された部分集合により表される命令に従って変更するようなネイティブ命令を有していることを特徴とする方法。
【請求項７】
請求項１に記載の方法において、前記プリアンブル及び前記ポスタンブルが、前記仮想マ

シン命令のうちの前記識別された部分集合の前記メンバと関連する競合する命令をオーバーライドすることを特徴とする方法。
【請求項８】
請求項１に記載の方法において、前記プリアンブル及び前記ポスタンブルが、前記仮想マシン命令のうちの前記識別された部分集合の前記メンバと関連する競合する命令を変更することを特徴とする方法。
【請求項９】
請求項１に記載の方法において、更に戻りサブルーチンを開始するステップを有し、該戻りサブルーチンが、
前記プリアンブル及びポスタンブルによりアクセスされた前記プロセッサ内のレジスタを解放するステップと、
前記仮想マシンに、前記一連の仮想マシン命令における或る位置において翻訳を再開するよう指示するステップと、
を有していることを特徴とする方法。
【請求項１０】
仮想マシン命令を処理する方法において、
中央処理ユニット内に環境をコンフィギュレーションするステップと、
一連のネイティブプロセッサ命令を規定するようなインターフェースを実施化するステップであって、該一連のネイティブプロセッサ命令が、
前記仮想マシン命令により表される前記ネイティブプロセッサ命令を実行する前にプリアンブルを実行するステップ、
前記プロセッサから、前記仮想マシン命令により表される命令に対応するネイティブ機能を取り込むステップ、
前記取り込まれたネイティブ命令を実行するステップ、
前記仮想マシン命令により表される前記ネイティブ命令を実行した後にポスタンブルを実行するステップ、
及び前記コンフィギュレーションされた環境を解放するステップ、
を有するようなステップと、
前記一連のネイティブプロセッサ命令を実行するステップと、
を有す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装置において、
ネイティブ命令セットを有し、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ように構成されると共に、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に翻訳するように構成された仮想マシンであるような前置プロセッサと、
前記命令メモリと前記プロセッサとの間に配されると共に、前記仮想マシン命令により表される命令に対してアクセス可能なプロセッサユニット内の変数の値を書くようなサブルーチンを開始し、前記仮想マシン命令により表される前記命令に対応するようなネイティブサブルーチンコールを発し、前記変数の値を前記仮想マシン命令に従って変更し、前記前置プロセッサに制御を返すように構成されたインターフェースと、
を有していることを特徴とする装置。
【請求項１２】
仮想マシン命令により呼び出されたネイティブプロセッササブルーチンを実行する方法において、
前記呼び出されたサブルーチンを実行する前にネイティブ命令のプリアンブルの群を実行するステップであって、前記プリアンブルが前記プロセッサ内のレジスタからの値を前記呼び出されたサブルーチンに対してアクセス可能な変数に書き込むように構成されている

ようなステップと、
前記呼び出されたサブルーチンを実行した後にネイティブ命令のポスタンブルの群を実行するステップであって、前記ポスタンブルが前記プロセッサにおけるレジスタ内の変数の値を前記呼び出されたサブルーチンに従って変更するように構成されているようなステップと、
を有することを特徴とする方法。
【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み取り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、コンピュータが命令を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、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翻訳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な翻訳は、実行時の前に生じる。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でのみ動作することができる。
【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動作するようなソースコードを用いて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した解釈プログラミング言語である。実行時においては、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翻訳される。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、解読及びディスパッチ”（ＦＤＤ）なる一連の動作を実行する。各バイトコード命令に対して、インタープリタソフトウェアは、ネイティブな中央処理ユニット（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。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ませ又は読み込ませ、該バイトコード命令のための実行プログラムのＣＰＵアドレスを解読させ、ＣＰＵの制御を該実行プログラムに移すことによりディスパッチさせる。斯かる解釈処理は時間が掛かるものであり得る。
【０００５】
国際特許出願公開第WO09918484号に開示されているように、メモリとＣＰＵとの間に前置プロセッサ（仮想マシンインタープリタ（ＶＭＩ））を追加することにより、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、当該ハードウェアプラットフォーム用のバイトコードをＶＭ又はＣＰＵ内に記憶された対応するネイティブなマシン言語命令を選択することにより解釈するような自立型（self-contained）動作環境である。この場合、上記ネイティブな命令は当該ハードウェアプラットフォームのＣＰＵに供給されると共に、該ＣＰＵにおいて連続的に実行される。典型的な仮想マシンは、ＦＤＤ系列の動作を実行するためにバイトコード当たり（該バイトコードの品質及び複雑さに依存する）２０ないし６０サイクルの処理時間を必要とする。
【０００６】

処理時間の更なる低減は、国際特許出願公開第WO9918484号及び第WO9918486号に開示されているように、ハードウェアアクセラレータを実施化することにより達成することができる。先ず、ＶＭＩがメモリからバイトコードを読み込む（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特性（プロパティ）をルックアップする（解読する）。ＶＭＩによりアクセスされた上記特性のうちの１つは、当該バイトコードが単純か又は複雑かを示し、これが、当該ＶＭＩが該バイトコードをハードウェアで翻訳することが可能かを決定する。ＶＭＩは単純なジャバ（登録商標）のバイトコードをネイティブなＣＰＵ命令の特化及び最適化されたシーケンスに変換し、次いで、これら命令が該ＣＰＵにより取り込まれ、実行される。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、ＣＰＵ命令に変換する。ＶＭＩは簡単なバイトコードは１ないし４サイクル内で処理することができる。バイトコードが複雑であると該コードの特性が示す場合は、ＶＭＩはネイティブなＣＰＵ命令の汎用的シーケンスを発生し、これにより、“ソフトウェアトラップ”を実施し、該トラップは上記複雑なバイトコードを解釈及び実行のためにソフトウェアに向かわせる。複雑なバイトコードに出くわすと、ＶＭＩはＣＰＵに対して、対応するネイティブな機能、即ち当該ＣＰＵ内に存在するマシンコードサブルーチン、を実行させるようなマシンコード命令を送出する。これに応答して、ＣＰＵは、前のバイトコード翻訳の結果としてＶＭＩにより発生されたネイティブ命令の実行を中断して、上記複雑なバイトコードにより呼び出されたネイティブ機能を実行する。ＶＭＩは自身の出力バッファからの更新された取り込みの検出を待ち、次いで、バイトコードのシーケンスの翻訳に戻る。上記ＶＭＩは複雑なバイトコードを命令毎に翻訳するというよりは、既存のネイティブ機能にアクセスするが、該ＶＭＩはＦＤＤ時間の効果を略零サイクルに低減する。何故なら、各ソフトウェアトラップのＶＭＩ処理（５～２０サイクル）は、他のバイトコードのＣＰＵ実行と同時に生じるからである。

【０００７】

一連のバイトコードを解釈している間に、仮想マシンは“再帰的”であるような複雑なバイトコードに出会う可能性がある。再帰的バイトコード（ＲＢＣ）を実行する場合には困難さが生じる。何故なら、各ＲＢＣは、最終的にメソッド呼び出し（method call）となるようなネイティブ機能を呼び出すからである（即ち、当該ＶＭＩはジャババイトコードの他のシーケンスに関して再起動される）。このように、他のシーケンスからのバイトコードを解読するために、ＲＢＣにより呼び出された上記ネイティブ機能はＣＰＵレジスタに記憶された変数の値にアクセスしなければならない。ネイティブ機能は、通常は（常にとは限らない）、以下で“プリアンブル”及び“ポスタンブル”と呼ぶような標準の命令の組を含む。該標準のプリアンブルは、サブルーチンが実行される前に、幾つかの又は全てのＣＰＵレジスタの内容を待避するように設計されている。待避される上記内容の幾つかは、当該ネイティブ機能の実行の間に変更される必要があり得るような変数（当該仮想マシンを再起動するためのスタックポインタ及び戻りアドレス等の）の値に関するものである。しかしながら、これらの変数は通常は当該ＲＢＣにより意図されるネイティブ機能内ではアクセス可能ではない。何故なら、ＲＢＣメソッド呼び出しは、ＣＰＵレジスタにアクセスする必要があるマシンコードというよりは、プログラミング言語のソースコードで規定されているからである。ＲＢＣメソッド呼び出しはＣＰＵレジスタにアクセスすることができないので、標準サブルーチンポスタンブルは、当該ＲＢＣメソッド呼び出しにより変更された変数の値をＣＰＵレジスタに書き戻すことができない。言い換えると、再帰的バイトコードにより呼び出されたサブルーチンは、適切に実行することができない。何故なら、ＲＢＣサブルーチンはＣＰＵレジスタに記憶された変数に、コンテキスト設定の欠乏（プリアンブル又はポスタンブルが無い）により、又は呼び出されるネイティブ機能との互換性のないコンテキスト設定の関連付けの何れかにより、アクセスすることができないからである。この問題に、ＲＢＣサブルーチンの変数の値を書き込み及び変更するためのアセンブリ言語を発生するような手でコード化されたソフトウェアを実施することにより対処することもできる。しかしながら、この方法は、複雑な開発努力を必要とする。

【発明の開示】
【発明が解決しようとする課題】
【０００８】
このように、再帰的バイトコードにより意図される命令を正確に且つ効率的に実行し、それでいて実施化が極めて容易であるような解釈プログラミング言語のシステムに対する要求が存在する。
【課題を解決するための手段】
【０００９】
本発明は、再帰的バイトコードにより呼び出されるネイティブ機能に対してプリアンブル及びポスタンブルを供給すると共に斯かるネイティブ機能を実行するような特別な機能を規定することにより上記要求を満たすものである。上記プリアンブル及びポスタンブルは、ネイティブ機能に関連する標準の如何なるプリアンブル及びポスタンブルも置き換え（オーバーライドし）又は変更すると共に、上記ネイティブ機能が所要のＣＰＵレジスタの内容にアクセスし及び斯かる内容を変更するのを可能にする。他の実施例においては、如何なる複雑なバイトコードに対しても、特別な機能をプロセッサメモリから取り込むか又は発生することができる。これにより、本発明は正確さ及び速度を維持する一方、ソフトウェアトラップ又は“ＶＭＩ加速ハードウェア”方法の実施化を簡素化する。
【００１０】
更に詳細には、本発明はＶＭＩサポートをアプリケーションプログラミングインターフェース（ＡＰＩ）の形で提供する。ＶＭＩが一連のバイトコードを解釈及び実行している間に再帰的バイトコードに出くわすと、本発明のＶＭＩサポートソフトウェアは、該再帰的バイトコードにより呼び出されたトップレベルサブルーチンを追加のプリアンブル及び追加のポスタンブルでカプセル封止する。本発明の上記追加のプリアンブルは、当該サブルーチンが呼び出された場合にＣＰＵレジスタ内に含まれる変数の値を操作し、これにより、ＣＰＵが、これらの値を当該呼び出されたサブルーチン内でアクセス可能な変数に書き込むようにさせる。このようにして、呼び出されたサブルーチンは上記変数の値を変更することができる。標準のポスタンブルの後に実行されて、本発明の上記追加のポスタンブルは、ＣＰＵに、変更された変数の値を取り込ませ、斯かる値をＣＰＵレジスタに書き込ませる。従って、上記の変更された変数は、ＣＰＵが後続のバイトコード翻訳から生じるネイティブ命令の実行に戻る場合に利用可能となる。
【００１１】
簡単に述べると、本発明は仮想マシン命令を処理する方法及びシステムを含み、これら命令は本発明の実施例ではジャバ（登録商標）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ために仮想マシンにより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、実行時に仮想マシン（実施例では、ＶＭＩ）は初期化される。全てのソフトウェアトラップに対するエントリアドレスを含むようなテーブルがコンパイルされ、その場合に、斯かるソフトウェアトラップは特定の複雑なバイトコードの処理をＶＭハードウェアから除去し、代わりに複雑なバイトコードを、ソフトウェアを用いて処理する。パラメータが初期化され、これらパラメータはバイトコードを各バイトコードの特徴に適した処理方法に従って分類する。例えば、バイトコードは簡単、複雑又は再帰的（ＲＢＣ）として特徴付けることができる。再帰的バイトコードは、第２シーケンスのバイトコード内の１以上のバイトコードを実行し得るサブルーチンを呼び出すと共に第１シーケンスのバイトコードに戻るような、第１シーケンスのバイトコード内の仮想マシン命令である。再帰的ジャババイトコードの例は、INVOKESTATIC及びNEWを含む。ＲＢＣを処理するようにコンフィギュレーションされたソフトウェアトラップには、適切なプリアンブル及びポスタンブルが設けられる。
【００１２】
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翻訳する。ＲＢＣに出くわすと、ＶＭＩは当該シーケンスのバイトコードの翻訳を中断し、該ＲＢＣを適切なソ

フトウェアトラップを介して処理する。通常は、当該ソフトウェアトラップは、上記ＲＢＣの仮想マシン命令に対応するネイティブ機能（又は複数の機能）を取り出す。本発明のアプリケーションプログラミングインターフェース（ＡＰＩ）は、プリアンブルを実行し、上記ＲＢＣに対応するネイティブ機能を呼び出し、ポスタンブルを実行するような適切なサブルーチンを規定する。上記プリアンブルは、上記ＲＢＣにより呼び出されるネイティブ機能（又は複数の機能）の実行の前に実行される。上記ポスタンブルは、上記ＲＢＣにより呼び出されるネイティブ機能（又は複数の機能）の実行の後で、且つ、当該ＲＢＣネイティブ機能（又は複数の機能）における全てのRETURNステートメントの後で実行される。当該ＲＢＣにより呼び出されたサブルーチンが、標準のプリアンブル及び／又は標準のポスタンブルを含んでいる場合は、本発明のプリアンブルが優先し、典型的には上記標準のプリアンブルを実行する前に実行され、また、本発明のポスタンブルが優先し、典型的には上記標準のポスタンブルの実行の後に実行される。他の例として、本発明のプリアンブル及びポスタンブルは、標準のプリアンブル又は標準のポスタンブルに各々含まれる１以上の命令を、物理的に変更し又は書き換えることもできる。本発明の実施例の或る見地によれば、上記プリアンブルは、ＲＢＣにより呼び出されるサブルーチンにアクセス可能な変数に値を書き込む。本発明のポスタンブルは、呼び出されたサブルーチンにより変更された変数の値を、これら変数を含むＣＰＵレジスタに書き戻す。次いで、上記ＡＰＩにより規定されたサブルーチンは、前記シーケンスのバイトコードの翻訳処理を再開し、該翻訳処理は他のＲＢＣに出会うまで継続するか、又は、さもなければ該翻訳処理は終了する。他の例として、当該ＡＰＩにより指定されたサブルーチンは、ＶＭＩのコンテキストに対するネイティブ演算のアクセスを可能にするように必要に応じてプリアンブル及びポスタンブルを実行することができる。例えば、ネイティブ処理が開始する前に第１プリアンブルを、当該メソッドが処理される前に第１ポスタンブルを、当該メソッドの実行の後に他のプリアンブルを、ネイティブ処理が完了した後に最後のポスタンブルを実行することができる。

【００１３】

ＶＭＩ翻訳には、C RETURNステートメント等のコードを使用して戻る。しかしながら、本発明による実施例においては、ＲＢＣにより発せられる上記C RETURNステートメントは、該ＲＢＣを処理するための環境としてコンフィギュレーションされたＣＰＵの領域を解放すると共にＶＭＩ実行を当該バイトコードシーケンス内の位置、例えば最も最近に翻訳されたＲＢＣの直後の位置に戻すような復帰サブルーチンにより拡張される。

【００１４】

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）及び前置プロセッサ（ＶＭＩ）、命令メモリ、トランスレータ（ＪＶＭ）並びにアプリケーションプログラミングインターフェース（ＡＰＩ）を含む。上記プロセッサは、以下ではネイティブ命令と呼ぶハードウェア固有の命令を含み、且つ、斯かる命令を実行するように構成される。上記前置プロセッサは、上記命令メモリからバイトコードを取り込むと共に斯かる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、例えばＶＭＩ等の仮想マシンである。特定のタイプのバイトコードに対して、上記ＡＰＩは、ＣＰＵレジスタから変数に値を、これら変数が当該バイトコードにより表されるネイティブＣＰＵ命令にたいしてアクセス可能となるように書き込むと共に、斯かる変数の値を当該バイトコードにより表されるネイティブＣＰＵ命令に従って変更するように構成される。本発明の実施例においては、該ＡＰＩは、プリアンブルを当該バイトコードにより呼び出されるサブルーチンの実行よりも前に、ポスタンブルを斯かる実行の後に実施するようなサブルーチンを指定する。上記プリアンブルはＣＰＵレジスタからの値を、呼び出されるサブルーチンに対してアクセス可能な変数に書き込む。当該バイトコードにより表される命令を実行した後、当該ＡＰＩは上記変数の値を、それに応じて、ポスタンブルサブルーチンを実行することにより変更する。

【００１５】

種々のタイプのバイトコードを処理するために本発明のＡＰＩと組み合わされたソフトウェアトラップ方法を実施することは可能であるが、本発明の実施例は再帰的バイトコードの処理を目指すものである。

【００１６】

本発明は、例えばサンマイクロシステムズ（Sun Microsystems）により作成されたＪＶＭのような仮想マシンを使用してジャバ（登録商標）バイトコードを実行する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・バーチャル・マシン（Microsoft Virtual Machine）のような他のジャバ（登録商標）仮想マシンを用いて実施化することもできると共に、ビジュアルベーシック、ｄＢＡＳＥ、ベーシック及びＭＳＩＬ（マイクロソフト中間言語）のような他の解釈言語を実行するシステムにも適用可能である。

【００１７】

本発明の更なる目的、利点及び新規なフィーチャは、一部は以下の説明に記載されると共に、一部は下記の精査により当業者にとり一層明らかとなるか、又は本発明の実施により分かるであろう。

【発明を実施するための最良の形態】

【００１８】

添付図面（同様の符号は同様の構成要素を示している）に示された本発明の一実施例を詳細に参照するに、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、該ハードウェアプラットフォームはプロセッサ１１０、前置プロセッサ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。前置プロセッサ１２０は、少なくとも１つのテーブル１４０と、トランスレータ１３０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、中央処理ユニット（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（登録商標）ｘ８６、パワーＰＣ（登録商標）又はＳＰＡＲＣタイプのマイクロプロセッサ等のＣＰＵであり、以下ではネイティブ命令と呼ぶハードウェア固有の命令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。本発明の該実施例においては、トランスレータ１３０は、サンマイクロシステムズによるＫＶＭのような、ジャバ（登録商標）仮想マシン（ＪＶＭ）である。該実施例における前置プロセッサ１２０は、好ましくは、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、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る。ＶＭＩ１２０はバス１６０上の周辺機器であり、所定の範囲のＣＰＵアドレスが該ＶＭＩ１２０に割り当てられるようなメモリマップ型周辺機器として動作する。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仮想マシン命令ポインタを管理する。命令メモリ１５０は、例えばジャバ（登録商標）バイトコード１７０のような仮想マシン命令を含む。

【００１９】

ＶＭＩ１２０は、ジャバ（登録商標）バイトコードの解釈を、殆どのバイトコード１７０をネイティブＣＰＵ命令の最適なシーケンスに変換することにより加速する。しかしながら、ＶＭＩ１２０は、複雑なバイトコード１７０を処理するソフトウェア解決策を、ソフトウェアトラップを実行することにより実施する。本発明は、概略的には、再帰型バイトコード（ＲＢＣ）２３０として知られている特定のタイプの複雑なバイトコードを目標とするソフトウェアトラップの実行を拡張するシステム及び方法を目指すものである。図２は、バイトコード１７０の第１系列２１０におけるＲＢＣ２３０の処理が、該バイトコード１７０の第２系列２２０におけるバイトコード１７０の処理、及び続いての系列２１０におけるバイトコード１７０の処理への戻りを生じさせるような、ＲＢＣ２３０の限定的な特徴を示している。このように、ＲＢＣ２３０の実行の結果、“メソッド”として知られているバイトコードサブルーチンとなる。系列２１０が系列２２０に等しい場合は、系列２１０は、“再帰的バイトコード”とは等価ではない用語の“再帰的メソッド”を構成

することに注意されたい。本発明は、ＲＢＣ２３０をソフトウェアトラップの一般構造の指定を含むようなＡＰＩを規定することにより処理するようなソフトウェアトラップの一般的動作をサポートする。

【００２０】

本発明によれば、図３を参照すると、実行時にＶＭＩ１２０は初期化される、即ち当該ＶＭＩ１２０の翻訳制御レジスタの設定を含む処理がなされる。全てのソフトウェアトラップに対するエントリアドレスを含むテーブル１４０がコンパイルされる。当該ＶＭＩが各バイトコードを当該バイトコードの特性に従って処理するのを可能にするパラメータが初期化される。当該バイトコードが複雑である場合、ＶＭＩは該バイトコードを適切なソフトウェアトラップを介して処理する。本発明によれば、ＲＢＣ２３０を処理するようにコンフィギュレーションされたソフトウェアトラップは、一緒に該ソフトウェアトラップを形成するネイティブ命令の汎用シーケンスの一部としての適切なプリアンブル及びポスタンブルに向けられる。斯かるプリアンブル及びポスタンブルは、ＶＭＩ１２０により発生することもできるが、本実施例では、プロセッサメモリに記憶され、該メモリにおいてインスタンス生成され、該メモリから実行される。他の例として、プリアンブル及びポスタンブルは当該ＶＭＩ内のテーブル１４０に記憶することもできる。このように、当該ＡＰＩ構成においては、システム初期化に際して、各ＲＢＣソフトウェアトラップエントリポイントはネイティブサブルーチン（“ＰＰＡサブルーチン”）を指すようにプログラムされる。該ＰＰＡサブルーチンは、プリアンブル及びポスタンブルの実行を含み、当該ＲＢＣにより呼び出されるネイティブ機能（又は複数の機能）へのジャンプを含むようにインスタンス生成されている。

【００２１】

本発明の動作の一例として、ＶＭＩ１２０は系列２１０のバイトコード１７０の各々を１以上のネイティブ命令に翻訳する。ここで図２を参照すると、バイトコードＢ０ないしＢ２は非再帰的であるので、ＶＭＩ１２０は単に命令メモリ１５０からＢ０ないしＢ２を取り込み、各バイトコード１７０に対して定義されたネイティブ命令（又は複数のネイティブ命令）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ＲＢＣ２３０であり、該Ｂｎの実行は、結果として、第２シーケンス２２０の１以上のバイトコード（ここでは、ＢｏないしＢｓ）の実行となり得る。ジャバにおいては、メソッド呼び出しは常に第１バイトコードでシーケンスに入るが、他の言語が命令シーケンスに沿う他の位置で入るのを許容しないという技術的理由は存在しない。第２シーケンス２２０におけるバイトコードの実行の後、Ｂｓは、バイトコードの実行を第１シーケンス２１０における何処かで継続させるようなRETURNバイトコードである。ジャバメソッドは、Ｂｓがシーケンス２２０の最後のバイトコードではないものとして示されているように、幾つかの出口点（RETURNバイトコード）を有することができる。Ｂｓの後、実行は典型的には第１シーケンスにおけるＢn+1において継続するが、これが必ずしも当てはまる必要はない。

【００２２】

図３のブロック３１０に示されるように、ＶＭＩ１２０はブロック３２０に進んで各バイトコード１７０を取り込む前に、仮想マシンカウンタをインクリメントする。ブロック３３０において、ＶＭＩ１２０はバイトコード１７０を、当該バイトコード１７０の特性（プロパティ）から該バイトコード１７０が“単純”であるか、即ち該バイトコード１７０に対する少なくとも１つのネイティブ命令からなるハードウェア翻訳が存在するかを判定することによりデコードする。当該バイトコード１７０に対して既存のハードウェア翻訳が存在する場合は、当該方法はブロック３７０に進み、該単純なバイトコードをそれに応じて処理する。当該バイトコード１７０に対するハードウェア翻訳が存在しない場合は、該バイトコードは複雑である。ブロック３４０において、当該ＶＭＩは該バイトコード１７０を、複雑なバイトコードに対するネイティブ命令の適切な汎用シーケンスを識別するテーブル１４０内のパラメータに対して判定する。ＶＭＩ１２０は第１シーケンス２１０のバイトコード１７０の翻訳を中断し、当該複雑なバイトコードを、適切なソフトウェア

トラップを形成し且つアドレスがテーブル１４０内に配置されているような該汎用シーケンスに従って処理する。ソフトウェアトラップは、典型的には、複雑なバイトコードを、当該バイトコード１７０により表されるサブルーチンを命令毎に解釈することによるよりは、当該バイトコード１７０に対応するネイティブ機能を取り出し、該ネイティブ機能をＣＰＵ１１０にディスパッチすることにより処理する。本発明のアプリケーションプログラミングインターフェース（ＡＰＩ）は、実施されるソフトウェアトラップ方法をＲＢＣを処理するよう拡張するのに要する機能を規定すると共に、該機能がどの様にアクセスされるかを規定する仕様である。ブロック３８０において仮定されているように当該バイトコード１７０が再帰的でない場合、ブロック３４０において識別される適切なソフトウェアトラップによれば、ブロック３６８におけるように制御がＶＭＩに戻されるまで、ブロック３８４においてネイティブ演算が処理される。当該バイトコード１７０が、ブロック３５０において仮定されているように、再帰的である場合、本発明のＡＰＩは適切なＰＰＡサブルーチン３６０を指定し、該サブルーチンはＲＢＣ２３０のための環境を、ＲＢＣ２３０により表されるネイティブ機能が当該サブルーチンの実行に必要な変数の値を含むＣＰＵ１１０内のレジスタにアクセスすることができるようにコンフィギュレーションする。必要な変数は、一定のプール、スタック、ローカル及びプログラムコードに対するポインタを含み得る。上記ＡＰＩにより指定されたＰＰＡサブルーチン３６０は、ＣＰＵがブロック３６４においてＲＢＣ３２０に対応するネイティブ演算を処理し始める前に、ブロック３６２においてプリアンブルを実行する。ネイティブ処理は、ブロック３６４においてネイティブコンテンツアクセス機能に出会い、ＶＭＩを再起動するまで継続する。ＶＭＩのパラメータは変更されて、RETURNバイトコードが何れかの残存ネイティブ処理を再開させる（ブロック３６６）まで、ブロック３６５においてメソッド２２０を処理する。ネイティブ処理の間に他のRETURNバイトコードに出会うと、制御はＰＰＡサブルーチンに戻され、該サブルーチンはブロック３６７においてポスタンブルを実行し、ブロック３６８においては、該ＰＰＡサブルーチン３６０の実行が完了した後に新たなパラメータに従って処理を行うために、制御がＶＭＩに戻される。適切なＰＰＡサブルーチン３６０のアドレスは、テーブル１４０に記憶され、再帰的であるバイトコード１７０を処理するように構成されたソフトウェアトラップに対応する。プリアンブルは、ＲＢＣ２３０により呼び出されるネイティブ演算（ネイティブ処理）の実行の前に実行される。標準のプリアンブルが、ＲＢＣ２３０により呼び出される上記ネイティブ機能に関連する場合、本発明のプリアンブルは該ネイティブ機能に関連する標準のプリアンブルに優先するか又は斯かる標準のプリアンブルを変更するかの何れかのように実行される。ポスタンブルは、ＲＢＣ２３０により呼び出されたネイティブ機能の実行の後で実行される。同様に、本発明のポスタンブルは、ＲＢＣにより呼び出されたネイティブ機能に関連する標準のポスタンブルに優先するか又は斯かる標準のポスタンブルを変更するかの何れかのように（標準のポスタンブルが存在する場合）、且つ、ＲＢＣネイティブ機能におけるC RETURNステートメントの後に実行される。

【００２３】

本発明の実施例の一態様によれば、プリアンブルは、スタックポインタ等の特定の重要なポインタを表す変数に値を書き込み、これにより斯かるポインタをＲＢＣ２３０により呼び出されるネイティブ機能に対してアクセス可能にする。例えば、ＲＢＣ２３０により呼び出されるネイティブ機能のコンテンツにアクセスするために、幾つかの機能は：

【数１】

```
void* vmi_bcc(void* jfp);

void* vmi_object(void* jfp);

void* vmi_jsp(void* jfp);

void* vmi_cpd(void* jfp);

void* vmi_cpt(void* jfp);

void* vmi_bcc(void* jfp);
```

と定義される。同様に、下記の機能は一定のプールへのアクセスを可能にする：

【数２】

```
<t> vmi_cpdEntry(void* jfp, unsigned n, <t>)

unsigned char vmi_cptEntry(void* jfp, unsigned n)
```

下記の機能は現在の及び次のバイトコード１７０に対すルウアクセスを可能にする：

【数３】

```
unsigned char vmi_bc(void* jfp, n)

unsigned char vmi_shortPar(void* jfp, n)

unsigned char vmi_3bytePar(void* jfp, n)

unsigned char vmi_wordPar(void* jfp, n)
```

下記の機能は、ジャバ（登録商標）スタックにアクセスするために使用される：

【数４】

```
<t> vmi_stkEntry(void* jfp, n, <t>)
```

【００２４】

次に図４を参照すると、ＶＭＩの能動的翻訳（ジャババイトコード処理）機能がジャバ仮想マシン（ＪＶＭ）規格により規定される。図４は、仮想的なバイトコード処理タイムラインを表し、該タイムラインにおいて間隔Ｔ０ないしＴ１２は、必ずしも等しくない任意の期間を表している。期間Ｔ０の間においては、ＶＭＩはバイトコード１７０を、再帰的バイトコード（ＲＢＣ）２３０に出会うまでインターフェースとしての本発明のＡＰＩを使用して翻訳する。図２の仮想図を相互参照すると、ＶＭＩはバイトコードＢ０ないしＢ２を処理し、次いでＢｎに出会う。ＲＢＣ２３０（Ｂｎ）は、適切なＲＢＣソフトウェアトラップ（ＡＰＩにより示されるように）が識別されると、期間Ｔ１において制御がネイティブ処理に渡されるようにし、対応するＰＰＡサブルーチンが実行される。期間Ｔ１ないしＴ１０においては、上記ソフトウェアトラップはＲＢＣ２３０により呼び出されたネイティブ機能を実行し、プリアンブルにより初期化されたコンテキストにアクセスする（期間Ｔ４において）。期間Ｔ３及びＴ４の処理サイクル（ＲＢＣ２３０に固有のネイティブ演算及びコンテキストアクセス機能への呼び出し）は、例えば期間Ｔ５及びＴ６において必要なだけ繰り返される。上記ネイティブ機能は最終的に（数千のネイティブに実行されたサイクルの後に可能となる）ネイティブコンテキストアクセス機能が得られ、これが

期間Ｔ７においてＶＭＩを再起動する。ＶＭＩは、期間Ｔ７においてRETURNバイトコード（Ｂｓ）に出会うまで、新たに呼び出されるメソッド（仮想的なバイトコードＢｏないしＢｒ）に関してバイトコードを能動的に処理し、上記RETURNバイトコードは制御をネイティブ機能の処理に戻す。期間Ｔ８において、制御はＶＭＩ１２０のハードウェアによるか又はソフトウェア（ＡＰＩを介して、及びＣプログラミング言語のRETURNステートメントのようなコードを介して達成される）によるかの何れかにより、暗黙的にＶＭＩ１２０からネイティブ処理に戻される。ネイティブ処理は期間Ｔ１０を介して継続し、他のC RETURNステートメントが制御をＰＰＡサブルーチンに移すと、ポスタンブルが期間Ｔ１１において実行される。本発明のポスタンブルは、ＲＢＣサブルーチンにより変更された変数の値を、期間Ｔ１２において制御が再びＶＭＩ１２０に戻される（仮想的バイトコードＢn+1において）前に、斯かる変数を含むＣＰＵレジスタに書き戻す。図５に示す他の実施例によれば、プリアンブル及びポスタンブルは、新たに呼び出されるメソッド（例示としてのバイトコードＢｏないしＢｓ）を処理するために必要とされるコンテキスト変更を実施するように実行することができる。例えば、ネイティブ処理の前の第１プリアンブルは期間Ｔ１で始まり、第１ポスタンブルは上記メソッドがＴ９ないしＴ１１において処理される前であり、他のプリアンブルは該メソッドの実行の後の期間Ｔ１２においてであり、最終のポスタンブルはネイティブ処理が完了した後の期間Ｔ１６においてである。

【００２５】

再び図３を参照すると、ブロック３６８において当該ＡＰＩにより指定されたＰＰＡサブルーチン３６０は、ＶＭＩ１２０におけるシーケンス１２０のバイトコード１７０の翻訳の再開を始めるようにコンフィギュレーションされる。制御は、ＶＭＩ１２０によるか又はＡＰＩの実施を介して、従ってハードウェアかソフトウェアの何れかにより実施されるかの何れかにより、ＶＭＩ１２０からネイティブ処理に暗黙的に戻される。この戻しルーチンのソフトウェア実施は、CプログラミングのRETURNステートメント（通常は、ネイティブ機能に含まれる）のようなコードを介して達成される。ネイティブ処理からの帰還の戻りの後に、上記コンテキストは例えば“vmi_freeFrame”により壊されるので、制御をＶＭＩ１２０に渡すことができる。上記コンテキストの破壊は、ＲＢＣ２３０を処理するための環境としてコンフィギュレーションされたＣＰＵ１１０の領域を解放し、ＶＭＩ１２０の翻訳をバイトコード１７０のシーケンス２１０内の位置に、例えば（必ずしもそうである必要はないが）最も最近に翻訳されたＲＢＣ２３０の直後の位置に戻す。

【００２６】

ブロック３１０において制御がＶＭＩ１２０に戻されると、前記仮想マシンカウンタはインクリメントされ、ＶＭＩ１２０は次のバイトコード１７０を取り込む。このように、ＶＭＩ１２０による翻訳は、他のＲＢＣ２３０に出会うか、又は、さもなければ当該翻訳処理が終了する（例えば、ＶＭＩが他の複雑なバイトコードに出会うか、又は処理すべきバイトコードを処理し尽くす）まで継続する。

【００２７】

上記から、本発明は仮想マシン命令を処理するソフトウェアトラップ方法をサポートするシステム及び方法を提供し、再帰的仮想マシン命令を正確且つ効率的に処理する方法の実施を容易にすることが分かるであろう。更に、上記は本発明の実施例に関係するのみで、添付請求項に記載される本発明の趣旨及び範囲から逸脱すること無しに本発明に対し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】

【００２８】

【図１】図１は、本発明の環境の一実施例の機能的構成要素を示すブロック図である。

【図２】図２は、例示的なバイトコード処理シーケンスを示す。

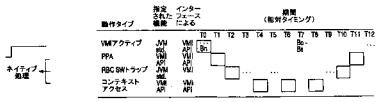
【図３】図３は、本発明の一実施例による方法のフローチャートである。

【図４】図４は、本発明の実施例に関わる演算の相対タイミングを示すタイムラインである。

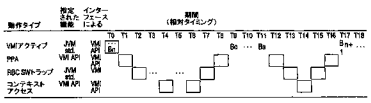
【図５】図５も、本発明の実施例に関わる演算の相対タイミングを示すタイムラインであ

る。

【図４】



【図５】



【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
3 April 2003 (03.04.2003)

PCT

(10) International Publication Number
WO 03/027842 A2

(51) International Patent Classification⁷: G06F 9/455

(21) International Application Number: PCT/IB02/03695

(22) International Filing Date:
6 September 2002 (06.09.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
01402455.8 25 September 2001 (25.09.2001) EP

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

(72) Inventors: **LINDWER, Menno, M.**; Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL). **BEN-YEDDER, Selim**; 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).

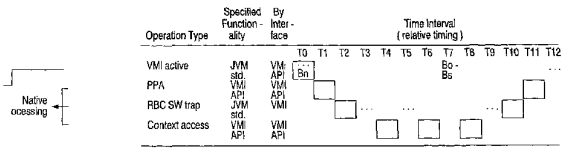
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, 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:
— *without international search report and to be republished upon receipt of that 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:** SOFTWARE SUPPORT FOR VIRTUAL MACHINE INTERPRETER (VMI) ACCELERATION HARDWARE



(57) **Abstract:** A system and method for processing virtual machine instructions which supports the software trap methodology. An application programming interface (API) prescribes additional functionality for software traps that remove the processing of recursive virtual machine instructions from virtual machine hardware and instead process recursive virtual machine instructions using software. The additional functionality includes the configuration of a context for processing recursive virtual machine instructions, which enables the virtual machine instructions to access CPU registers to retrieve and modify the values of variables as required, the release of the configured context when processing of recursive virtual machine instructions is completed, and the return of control to a virtual machine for processing non-recursive virtual machine instructions.

WO 03/027842 A2

Software support for virtual machine interpreter (VMI) acceleration hardware

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.

Further reductions in processing time can be achieved by implementing a hardware accelerator as disclosed in PCT Patent Application Nos. WO9918484 and WO9918486. First, a VMI reads (fetches) a bytecode from memory. Next, the VMI looks up a number of properties of (decodes) the fetched bytecode. One of the properties accessed by the VMI indicates whether the bytecode is simple or complex, which determines whether the VMI can translate the bytecode in hardware. The VMI translates simple Java™ bytecodes into a specialized and optimized sequence of native CPU instructions, which are then fetched and executed by the CPU. While the CPU is executing an instruction, the VMI fetches and translates the next bytecode into CPU instructions. The VMI can process simple bytecodes in 1-4 cycles. If its properties indicate that a bytecode is complex, the VMI generates a generic sequence of native CPU instructions, thereby implementing a "software trap" that directs the complex bytecode to software for interpretation and execution. Upon encountering a complex bytecode, a VMI issues machine code instructions to the CPU that cause the execution of the corresponding native function, i.e. a machine code subroutine residing in the CPU. In response, the CPU interrupts the execution of native instructions generated by the VMI as a result of previous bytecode translations and executes the native function called for by the complex bytecode. The VMI waits to detect renewed fetching from its output buffer and then resumes translation of the sequence of bytecodes. Although it accesses existing native functions rather than translating complex bytecodes instruction by instruction, the VMI reduces the effect of FDD time to almost 0 cycles, because VMI processing of each software trap (5-20 cycles) occurs concurrently with CPU execution of another bytecode.

While interpreting a sequence of bytecodes, a virtual machine may encounter a complex bytecode that is "recursive." A challenge arises in executing recursive bytecodes (RBCs) because each RBC calls for a native function that eventually results in a method call

(i.e., the VMI is reactivated for another sequence of Java bytecodes). Thus, to decode the bytecodes from the other sequence, the native function called for by the RBC must access the values of variables stored in CPU registers. Native functions commonly (but do not always) include standard sets of instructions, hereinafter referred to as "preambles" and "postambles." The standard preamble is designed to save the content of some or all CPU registers before a subroutine is executed. Some of the content that is saved concerns the values of variables (such as a stack pointer and the return address for reactivating the virtual machine) that may need to be modified during the execution of the native function. However, these variables will not normally be accessible within the native function intended by the RBC because the RBC method call is defined in source code of the programming language, rather than in the machine code that is required to access CPU registers. Furthermore, because the RBC method call cannot access the CPU registers, the standard subroutine postamble cannot write the value of variables that have been modified by the RBC method call back into the CPU registers. In other words, the subroutine called for by a recursive bytecode may not be executed properly because the RBC subroutine cannot access variables stored in CPU registers, either due to a lack of context-setting (no preamble or postambles) or due to an association of incompatible context-setting with the native function called. It is possible to address this challenge by implementing hand-coded software that generates assembly language to write and modify the values of variables for RBC subroutines, however this approach requires a complicated development effort.

There is a need for a system of interpreting programming languages that accurately and efficiently executes instructions intended by recursive bytecodes, while having greater ease of implementation.

SUMMARY OF THE INVENTION

The present invention fulfills the needs described above by prescribing a special function that supplies a preamble and postamble for and executes the native functions called for by a recursive bytecode. The preamble and postamble override or modify any standard preambles and postambles associated with the native functions, and enable the native functions to access and modify the content of necessary CPU registers. In alternative embodiments, special functions can be fetched from processor memory or generated for any complex bytecode. The present invention thereby maintains the accuracy and speed while simplifying the implementation of the software trap or "VMI acceleration hardware" methodology.

More specifically, the present invention provides VMI support in the form of an application programming interface (API). When a VMI encounters a recursive bytecode while interpreting and executing a sequence of bytecodes, the VMI support software of the present invention encapsulates the top-level subroutine called for by the recursive bytecode with an additional preamble and an additional postamble. The additional preamble of the present invention manipulates the values of variables that are contained in CPU registers when the subroutine is called, thereby causing the CPU to write these values into variables that are accessible within the called subroutine. In this manner, the called subroutine can modify the values of the variables. Executed after the standard subroutine postamble, the additional postamble of the present invention causes the CPU to fetch the values of modified variables and to write the values into CPU registers. Accordingly, the modified variables will be available when the CPU resumes the execution of native instructions that result from subsequent bytecode translations.

Briefly, the present invention includes methods and systems for processing 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 embodiment, a VMI) is initialized. A table is compiled that includes entry addresses for all software traps, where the software traps remove the processing of certain complex bytecodes from VM hardware and instead process complex bytecodes using software. Parameters are initialized which categorize bytecodes according to the appropriate processing methodology for the characteristics of each bytecode. For example, bytecodes may be characterized as simple, complex, or recursive (RBC). A recursive bytecode is a virtual machine instruction in a first sequence of bytecodes that invokes a subroutine that may execute one or more bytecodes in a second sequence of bytecodes and then returns to the first sequence of bytecodes. Examples of recursive Java bytecodes are INVOKESTATIC and NEW. Those software traps that are configured to process RBCs are furnished with appropriate preambles and postambles.

The VMI proceeds to translate each of a series of bytecodes into one or more native instructions. When an RBC is encountered, the VMI suspends the translation of the sequence of bytecodes and processes the RBC via the appropriate software trap. Generally, the software trap retrieves the native function(s) that correspond to the virtual machine

instructions of the RBC. The application programming interface (API) of the present invention prescribes an appropriate subroutine that executes a preamble, calls the native function corresponding to the RBC, and executes a postamble. The preamble is executed before execution of the native function(s) called for by the RBC. The postamble is executed after execution of the native function(s) called for by the RBC, and after any RETURN statement in the RBC native function(s). If the subroutine called for by the RBC contains a standard preamble and/or a standard postamble, then the preamble of the present invention overrides and is typically executed before executing the standard preamble, and the postamble of the present invention overrides and is typically executed after execution of the standard postamble. Alternatively, the preamble and postamble of the present invention may physically modify or overwrite one or more instructions contained in a standard preamble or a standard postamble, respectively. According to an aspect of the exemplary embodiment of the present invention the preamble writes values to variables that are accessible to the subroutine called for by the RBC. The postamble of the present invention writes the values of variables that were modified by the called subroutine back to the CPU registers that contain the variables. The subroutine prescribed by the API then initiates the resumption of translation in the VMI of the sequence of bytecodes, which will continue until another RBC is encountered or the translation process otherwise terminates. Alternatively, the subroutine prescribed by the API may execute preambles and postambles as needed to give native operations access to the VMI context – for example, a first preamble before native processing begins, a first postamble before the method is processed, another preamble after the execution of the method, and a final postamble after native processing is complete.

VMI translation is resumed using code such as a C RETURN statement. In the exemplary embodiment of the present invention however, the C RETURN statement that is issued by an RBC is augmented by a resumption subroutine that releases the areas of the CPU configured as an environment for processing the RBC and returns VMI execution to a position in the sequence of bytecodes, for example to the position immediately following the most recently translated RBC.

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, a translator (a JVM), and an application programming interface (the API). 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. For certain types of bytecode, the API is configured to write values from CPU registers into variables such that the variables are accessible to the native CPU instructions represented by the bytecode, and to modify the values of the variables according to the native CPU instructions represented by the bytecode. In the exemplary embodiment of the present invention, the API prescribes a subroutine that implements a preamble before and a postamble after execution of the subroutine called for by the bytecode. The preamble writes values from CPU registers to variables accessible to the called subroutine. After executing the instructions represented by the bytecode, the API modifies the values of the variables accordingly by executing a postamble subroutine.

Although it is possible to implement software trap methodology combined with the API of the present invention to process various types of bytecode, the exemplary embodiment of the present invention is directed to the processing of recursive 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 MSIL (Microsoft Intermediate Language).

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

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

FIG. 2 illustrates an exemplary bytecode processing sequence.

FIG. 3 is a flow chart of a method according to an exemplary embodiment of the present invention.

FIG. 4 is a timeline illustrating the relative timing of the operations involved in the exemplary embodiment of the present invention.

DESCRIPTION OF PREFERRED EMBODIMENTS

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 at least one table 130 and a translator 140. A hardware platform 100 typically includes a central processing unit (CPU), basic peripherals, and 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 preprocessor 120 in the exemplary embodiment is preferably the Virtual Machine Interpreter (VMI) disclosed in WO9918486, and is configured to fetch bytecode from the instruction memory and to translate the bytecode into 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 indicating the current (or next) virtual machine instruction in the instruction memory 150. The instruction memory 150 contains virtual machine instructions, for example, Java™ bytecode 170.

The VMI 120 accelerates the interpretation of Java™ bytecode 170, by translating most bytecodes 170 into optimal sequences of native CPU instructions. However, the VMI 120 implements a software solution for processing complex bytecodes 170, by executing a software trap. The present invention is generally directed to systems and methods for augmenting the execution of software traps that target a specific type of complex bytecode, known as a recursive bytecode (RBC) 230. FIG. 2 illustrates the definitive characteristic of an RBC 230, which is that processing of an RBC 230 in a first series 210 of bytecodes 170 causes the processing of bytecodes 170 in a second series 220 of bytecodes 170 and a subsequent return to processing of bytecodes 170 in series 210. Thus, the execution of an RBC 230 results in a bytecode subroutine known as a "method". Note that if series 210 is equal to series 220, then series 210 constitutes a "recursive method," a term which is not equivalent to "recursive bytecode." The present invention supports the general

operation of software traps that process RBCs 230, by defining an API that contains a prescription of the general structure of the software traps.

According to the present invention and referring now to FIG. 3, at runtime a VMI 120 is initialized, a process which includes setting the translation control registers of the VMI 120. A table 130 is compiled that includes entry addresses for all software traps. Parameters are initialized which enable the VMI to process each bytecode according to the properties of the bytecode. When the bytecode is complex, the VMI processes the bytecode via the appropriate software trap. According to the present invention, those software traps that are configured to process RBCs 230 are directed to the appropriate preambles and postambles, as part of the generic sequence of native instructions that together form the software trap. The preambles and postambles may be generated by the VMI 120, but in the exemplary embodiment are stored in, instantiated in, and executed from processor memory. Alternatively, the preambles and postambles could be stored in tables 130 within the VMI. Thus in the API implementation, upon system initialization each RBC software trap entry point is programmed to point to a native subroutine (a "PPA subroutine"). The PPA subroutine includes the execution of a preamble and a postamble, and has been instantiated to contain jumps to the native function(s) called for by the RBC.

As an example of the operation of the present invention, the VMI 120 proceeds to translate each of a series 210 of bytecodes 170 into one or more native instructions. Referring now to FIG. 2, bytecodes B0 through B2 are non-recursive, 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 an RBC 230 the execution of which may result in the execution of one or more bytecodes (here, Bo through Bs) from a second sequence 220. Note that in Java, method calls always enter a sequence at the first bytecode, however there is no technical reason that other languages would not allow entry at other positions along an instruction sequence. After execution of bytecodes in the second sequence 220, Bs is a RETURN bytecode that causes bytecode execution to continue somewhere in the first sequence 210. A Java method can have several exit points (RETURN bytecodes), as indicated by Bs not being the last bytecode of sequence 220. After Bs, execution typically continues in the first sequence 210 at B_{n+1}, but this does not necessarily have to be the case.

As shown in block 310 of FIG. 3, the VMI 120 increments a virtual machine counter before proceeding in block 320 to fetch each bytecode 170. In block 330, the VMI 120 decodes the bytecode 170, by determining from the properties of the bytecode 170

whether the bytecode 170 is "simple" – that is, whether there is a hardware translation consisting of at least one native instruction for the bytecode 170. If there is an existing hardware translation for the bytecode 170, the method proceeds to block 370 and processes the simple bytecode accordingly. If there is no hardware translation existing for the bytecode 170, then the bytecode 170 is complex. In block 340, the VMI tests the bytecode 170 against the parameters in the table 140 that identify the appropriate generic sequence of native instructions for the complex bytecode. The VMI 120 suspends the translation of the sequence 210 of bytecodes 170 and processes the complex bytecode according to this generic sequence which makes up the appropriate software trap, the address of which is located in the table 140. A software trap typically processes a complex bytecode by retrieving the native function that corresponds to the bytecode 170 and dispatching the native function to the CPU 110, rather than by interpreting the subroutine represented by the bytecode 170 instruction by instruction. The application programming interface (API) of the present invention is a specification that prescribes the functionality needed to augment the software trap methodology as implemented to process RBCs, and that prescribes how the functionality is accessed. If the bytecode 170 is not recursive as is assumed in block 380, according to the appropriate software trap as identified in block 340, the native operations are processed in block 384 until control is returned to the VMI as in block 368. If the bytecode 170 is recursive as is assumed in block 350, the API of the present invention prescribes an appropriate PPA subroutine 360 that configures the environment for an RBC 230 so that the native function represented by RBC 230 can access registers in the CPU 110 that contain values of variables that are necessary to the execution of the subroutine. The necessary variables may include pointers to the constant pool, stack, locals, and program code. The PPA subroutine 360 prescribed by the API executes a preamble in block 362 before the CPU begins processing native operations corresponding to the RBC 230 in block 364. Native processing continues until a native context access function is encountered in block 364, causing the reactivation of the VMI. The VMI's parameters are changed for processing the method 220 in block 365 until a RETURN bytecode causes the resumption of any remaining native processing (block 366). Encountering another RETURN bytecode during native processing shifts control back to the PPA subroutine to which executes a postamble in block 367, and in block 368 control is returned to the VMI for processing according to new parameters after execution of the PPA subroutine 360 is complete. The address of the appropriate PPA subroutine 360 is stored in the table 140, and corresponds to the software trap that is configured to process bytecodes 170 that are recursive. The preamble is executed

before execution of the native operations (native processing) called for by the RBC 230. If a standard preamble is associated with the native function called for by the RBC 230, the preamble of the present invention is executed so as to either override or modify the standard preamble associated with the native function. The postamble is executed after execution of the native function called for by the RBC 230. Similarly, the postamble of the present invention is executed so as to either override or modify a standard postamble associated with the native function called for by the RBC (if there is a standard postamble), and after the C RETURN statement in the RBC native function.

According to an aspect of the exemplary embodiment of the present invention the preamble writes values to variables that represent certain vital pointers such as the stack pointer, thereby making the pointers accessible to the native function called for by the RBC 230. For example, to provide access to the context of the native function called for by the RBC 230, several functions are defined:

```
void* vmi_bcc(void* jfp);
void* vmi_object(void* jfp);
void* vmi_jsp(void* jfp);
void* vmi_cpd(void* jfp);
void* vmi_cpt(void* jfp);
void* vmi_bcc(void* jfp);
```

Similarly, the following functions give access to the constant pool:

```
<t> vmi_cpdEntry(void* jfp, unsigned n, <t>)
unsigned char vmi_cptEntry(void* jfp, unsigned n)
```

The following functions give access to the current and next bytecode 170:

```
unsigned char vmi_bc(void* jfp, n)
unsigned char vmi_shortPar(void* jfp, n)
unsigned char vmi_3bytePar(void* jfp, n)
unsigned char vmi_wordPar(void* jfp, n)
```

The following function is used to access the Java™ stack:

```
<t> vmi_stkEntry(void* jfp, n, <t>)
```

Referring now to FIG. 4, the VMI's active translation (Java bytecode processing) functionality is prescribed by the Java Virtual Machine (JVM) standard. FIG. 4 represents a hypothetical bytecode processing timeline, where the intervals T0-T12 represent arbitrary periods of time that are not necessarily equal. During interval T0 the VMI translates bytecodes 170 using the API of the present invention as an interface until a recursive

bytecode (RBC) 230 is encountered. Cross-referencing the hypothetical diagram in FIG. 2, the VMI processes bytecodes B0 through B2 and then encounters Bn. The RBC 230 (Bn) causes control to be passed to native processing in interval T1, when the appropriate RBC software trap (as indicated by the API) is identified and the corresponding PPA subroutine is executed. In intervals T2 through T10, the software trap executes the native functions called for by the RBC 230, accessing (in interval T4) the context initialized by the preamble. The processing cycle (native operations specific to the RBC 230 and calls to context-accessing functions) of intervals T3 and T4 is repeated as needed, for example, in intervals T5 and T6. The native function eventually (possible after thousands of natively executed cycles) results in a native context access function which reactivates the VMI in interval T7. The VMI actively processes the bytecodes for the newly called method (hypothetical bytecodes Bo through Br) until a RETURN bytecode (Bs) is encountered in interval T7, which returns control back to the native function processing. In interval T8, control is returned from the VMI 120 to native processing implicitly either by the VMI 120 hardware or by software (via the API and accomplished via code such as a C programming language RETURN statement). Native processing continues through interval T10, when another C RETURN statement transfers control to the PPA subroutine so that the postamble is executed in interval T11. The postamble of the present invention writes the values of variables that were modified by the RBC subroutine back into the CPU registers that contain the variables, before control is again returned to the VMI 120 (at hypothetical bytecode B_{n+1}) in time interval T12. According the alternative embodiment shown in FIG. 5, preambles and postambles may be executed so as to effectuate context changes needed to process the newly called method (exemplary bytecodes Bo through Bs) – for example, the first preamble before native processing begins in interval T1, the first postamble before the method is processed in T9 through T11, another preamble in interval T12 after the execution of the method, and a final postamble in interval T16 after native processing is complete.

Referring again to FIG. 3, in block 368 the PPA subroutine 360 prescribed by the API is also configured to initiate the resumption of translation in the VMI 120 of the sequence 210 of bytecodes 170. Control is returned from the VMI 120 to native processing implicitly, either by the VMI 120 or via the API implementation and thus are implemented either in hardware or in software. Software implementations of this resumption routine are accomplished via code such as a C programming language RETURN statement, which is ordinarily included in a native function. After return from native processing, the context is destroyed, for example by "vmi_freeFrame" so that control can be passed back to the VMI

120 . Destruction of the context releases the areas of the CPU 110 configured as an environment for processing the RBC 230 and returns VMI 120 translation to a position in the sequence 210 of bytecodes 170, for example (but not necessarily) to the position immediately following the most recently translated RBC 230.

When control is returned to the VMI 120 in block 310, the virtual machine counter is incremented and the VMI 120 fetches the next bytecode 170. Thus, translation by the VMI 120 will continue until another RBC 230 is encountered or the translation process otherwise terminates (e.g., the VMI encounters another complex bytecode, or runs out of bytecodes to process).

In view of the foregoing, it will be appreciated that the present invention provides a system and a method for supporting the software trap approach for processing virtual machine instructions, so as to ease the implementation of a method for accurate and efficient processing recursive 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 comprising:

initializing parameters to identify a subset of the virtual machine instructions having a particular set of characteristics;

translating a range of the virtual machine instructions to native instructions executable by a processor, until a member of said identified subset of the virtual machine instructions is encountered;

upon encountering a member of said identified subset of the virtual machine instructions, suspending the translation of the range of virtual machine instructions;

implementing an interface, where said interface prescribes a set of native processor instructions that:

executes a preamble before executing the native processor instructions represented by said member of said identified subset of the virtual machine instructions;

retrieves native functions from the processor that correspond to the instructions represented by said member of said identified subset of the virtual machine instructions;

executes the retrieved native instructions;

executes a postamble after executing the native instructions represented by said member of said identified subset of the virtual machine instructions; and

resumes the translation in the virtual machine of the range of the virtual machine instructions, where execution continues until another member of said identified subset of the virtual machine instructions is encountered; and

executing said set of native processor instructions.

2. The method of Claim 1, further comprising:

executing a postamble before executing retrieved native instructions that constitute a method, where said postamble comprises native instructions that modify the values of variables in registers in the processor so as to effectuate context changes needed to execute the method; and

executing a preamble after executing retrieved native instructions that constitute a method, where said preamble comprises native instructions that write values from registers in the processor to variables that are accessible to the native instructions executed after the method has been executed.

3. The method of Claim 1, wherein said preamble and said postamble are generated by the virtual machine.

4. The method of Claim 1, wherein said preamble and said postamble are retrieved from processor memory.

5. The method of Claim 1, wherein said preamble comprises native instructions that write values from registers in the processor to variables that are accessible to the instructions represented by said identified subset of the virtual machine instructions.

6. The method of Claim 1, wherein said postamble comprises native instructions that modify the values of variables in registers in the processor according to the instructions represented by said identified subset of the virtual machine instructions into registers in the processor.

7. The method of Claim 1, wherein said preamble and said postamble override conflicting instructions associated with said member of said identified subset of the virtual machine instructions.

8. The method of Claim 1, wherein said preamble and said postamble modify conflicting instructions associated with said member of said identified subset of the virtual machine instructions.

9. The method of Claim 1, further comprising initiating a resumption subroutine, wherein said resumption subroutine comprises:

releasing the registers in the processor that were accessed by the preamble and postamble; and

directing the virtual machine to resume translation at a position in the range of virtual machine instructions.

10. A method of processing virtual machine instructions comprising:

configuring an environment in a central processing unit;

implementing an interface that prescribes a set of native processor instructions, said set of native processor instructions comprising:

executing a preamble before executing the native processor instructions represented by the virtual machine instructions;

retrieving native functions from the processor that correspond to the instructions represented by the virtual machine instructions;

executing the retrieved native instructions;

executing a postamble after executing the native instructions represented the virtual machine instructions; and

releasing the configured environment; and

executing said set of native processor instructions.

11. 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), where the preprocessor (120) is a virtual machine configured to fetch virtual machine instructions from the instruction memory (150) and configured to translate fetched virtual machine instructions into native instructions executable by the processor; and

an interface between the instruction memory (150) and the processor (110), configured to initiate a subroutine that writes register values of variables in the processor unit (100) that are accessible to the instructions represented by the virtual machine instructions, issues a native subroutine call that corresponds to the instructions represented by the virtual machine instructions, modifies the values of said variables according to the virtual machine instructions, and returns control to the preprocessor.

12. A method for executing a native processor subroutine called for by virtual machine instructions, comprising:

executing a preamble set of native instructions before executing the called subroutine, where said preamble is configured to write values from registers in the processor to variables that are accessible to the called subroutine; and

executing a postamble set of native instructions after executing the called subroutine, where said postamble is configured to modify the values of variables in registers in the processor according to the called subroutine.

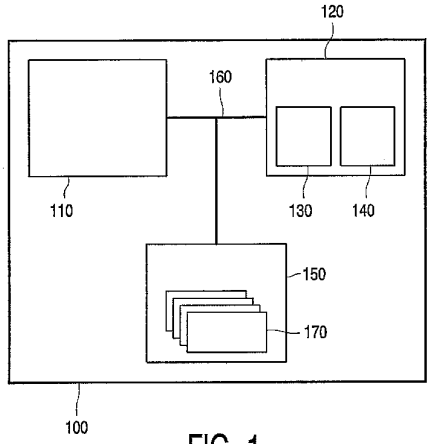

120
160
110
130
140
150
170
100

FIG. 1

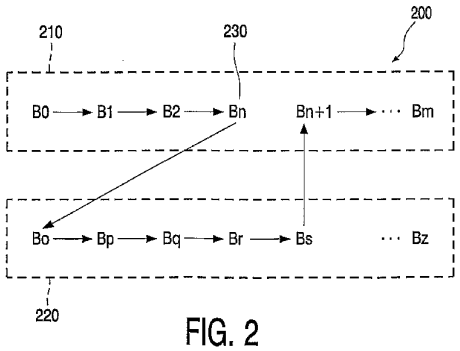

200
210
230
B0 B1 B2 Bn Bn+1 Bm
Bo Bp Bq Br Bs Bz
220

FIG. 2

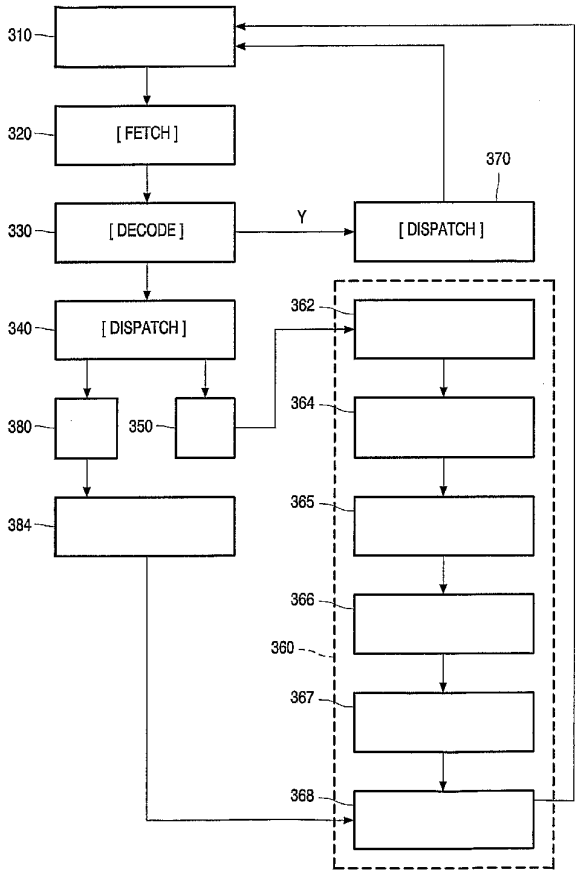

310
320
[FETCH]
370
330
[DECODE]
Y
[DISPATCH]
340
[DISPATCH]
362
364
380
350
365
384
366
360
367
368

FIG. 3

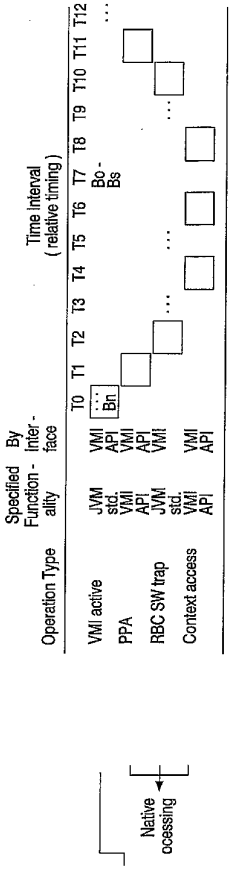

Time Interval
(relative timing)
Operation Type
Specified Function - ality
By Inter - face
T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12
VMI active
JVM std.
VMI API
PPA
VMI API
API JVM
RBC SW trap
JVM std.
VMI API
Context access
Bn
Bo - Bs
Native ocessing

FIG. 4

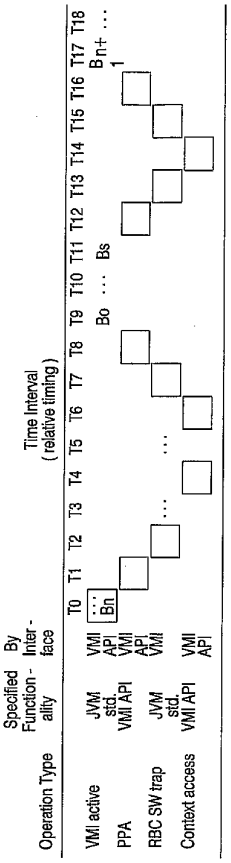

Operation Type
Specified Function-ality
By Inter-face
Time Interval (relative timing)
T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15 T16 T17 T18
VMI active
JVM std. VMI API
VMI API
Bn
Bo ... Bs
Bn+1 ...
PPA
VMI API
VMI
RBC SW trap
JVM std. VMI API
VMI
Context access
VMI API
VMI API

FIG. 5

【国際公開パンフレット（コレクトバージョン）】

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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
3 April 2003 (03.04.2003)

PCT

(10) International Publication Number
WO 2003/027842 A3

(51) International Patent Classification⁷: G06F 9/455, 9/40

(21) International Application Number:
PCT/IB2002/003695

(22) International Filing Date:
6 September 2002 (06.09.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
01402455.8 25 September 2001 (25.09.2001) EP

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

(72) Inventors: LINDWER, Menno, M.; Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL). BEN-YEDDER, Selim; 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.

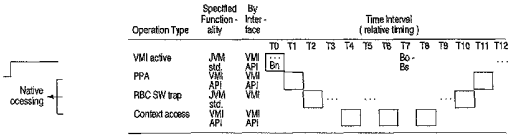
(84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM,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

Published:
— with international search report

(88) Date of publication of the international search report:
17 June 2004

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: SOFTWARE SUPPORT FOR VIRTUAL MACHINE INTERPRETER (VMI) ACCELERATION HARDWARE



(57) Abstract: A system and method for processing virtual machine instructions which supports the software trap methodology. An application programming interface (API) prescribes additional functionality for software traps that remove the processing of recursive virtual machine instructions from virtual machine hardware and instead process recursive virtual machine instructions using software. The additional functionality includes the configuration of a context for processing recursive virtual machine instructions, which enables the virtual machine instructions to access CPU registers to retrieve and modify the values of variables as required, the release of the configured context when processing of recursive virtual machine instructions is completed, and the return of control to a virtual machine for processing non-recursive virtual machine instructions.

WO 2003/027842 A3

【国際調査報告】

INTERNATIONAL SEARCH REPORT

Intern: Application No
PCT/IB 02/03695

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/455 G06F9/40

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, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 99/18486 A (KONINKL PHILIPS ELECTRONICS NV ; PHILIPS SVENSKA AB (SE)) 15 April 1999 (1999-04-15) cited in the application page 10, line 25 - page 11, line 24 page 12, line 9 - page 13, line 12 page 14, line 19 - line 23 page 15, line 34 - page 16, line 14	1-4,9-11
A	US 6 199 095 B1 (ROBINSON SCOTT G) 6 March 2001 (2001-03-06) column 9, line 42 - column 10, line 6 column 15, line 65 - column 16, line 28 column 26, line 14 - column 27, line 47 column 27, line 66 - column 28, line 34 column 29, line 54 - column 32, line 63	1-12

-/--

[X] Further documents are listed in the continuation of box C. [X] 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
6 February 2004	18/02/2004

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	Carciofi, A

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

INTERNATIONAL SEARCH REPORT

Interna pplication No
PCT/IB 02/03695

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 875 336 A (DICKOL JOHN EDWARD ET AL) 23 February 1999 (1999-02-23) abstract; table 3 column 5, line 33 - line 54	1-12

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

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB 02/03695

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9918486 A	15-04-1999	DE 69820027 D1	08-01-2004
		EP 1359501 A2	05-11-2003
		EP 0950216 A2	20-10-1999
		EP 0941508 A1	15-09-1999
		EP 1019794 A2	19-07-2000
		WO 9918484 A2	15-04-1999
		WO 9918485 A2	15-04-1999
		WO 9918486 A2	15-04-1999
		JP 2001508907 T	03-07-2001
		JP 2001508908 T	03-07-2001
		JP 2001508909 T	03-07-2001
		US 2002129225 A1	12-09-2002
		US 6292883 B1	18-09-2001
		US 6349377 B1	19-02-2002
		US 6298434 B1	02-10-2001
US 6199095 B1	06-03-2001	NONE	
US 5875336 A	23-02-1999	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 リンドウェル 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
(72)発明者 ベン－イェダー セリム
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) 5B081 AA09 DD01