(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2004-507832
(P2004-507832A)

(43)公表日　平成16年3月11日(2004.3.11)

| (51)Int.Cl.[7] | FI | | | テーマコード（参考） |
|---|---|---|---|---|
| G06F 9/45 | G06F | 9/44 | 320C | 5B033 |
| G06F 9/30 | G06F | 9/30 | 350A | 5B081 |
| G06F 9/44 | G06F | 9/44 | 530A | |
| G06F 9/455 | G06F | 9/44 | 310A | |

審査請求 未請求　予備審査請求 未請求　（全 29 頁）

(21)出願番号　特願2002-523152(P2002-523152)
(86)(22)出願日　平成13年8月22日(2001.8.22)
(85)翻訳文提出日　平成14年4月26日(2002.4.26)
(86)国際出願番号　PCT/EP2001/009694
(87)国際公開番号　WO2002/019100
(87)国際公開日　平成14年3月7日(2002.3.7)
(31)優先権主張番号　00203024.5
(32)優先日　平成12年8月31日(2000.8.31)
(33)優先権主張国　欧州特許庁(EP)
(81)指定国　EP(AT,BE,CH,CY,DE,DK,ES,FI,FR,GB,GR,IE,IT,LU,MC,NL,PT,SE,TR),JP

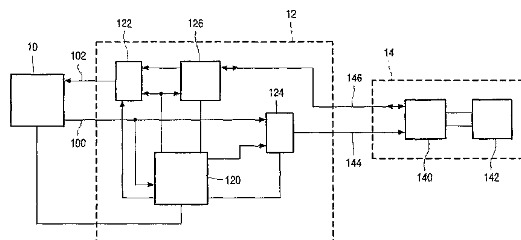
（特許庁注：以下のものは登録商標）
JAVA

(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　100075812
弁理士　吉武　賢次
(74)代理人　100088889
弁理士　橘谷　英俊

最終頁に続く

(54)【発明の名称】仮想マシン命令を実行するシステム

(57)【要約】
【課題】プロセサコア、メモリそして仮想マシンインタプリタを備えたデータ処理システムを提供する。
【解決手段】仮想マシンインタプリタはプロセサコアにより実行される仮想マシン命令を導入するネイティブマシン命令を発生する。仮想マシンインタプリタは繰り返し実行されることが予想される一連の仮想マシン命令の一群から先頭仮想マシン命令を検出する。仮想マシンインタプリタは一群中の先頭仮想マシン命令とメモリのメモリ位置との対応関係を記録し、このメモリ位置からメモリにネイティブマシン命令を書き込む。このメモリ位置からメモリに書き込まれたネイティブマシン命令を実行することにより、プロセサコア上記一群のためのネイティブマシン命令を繰り返し実行する。



【特許請求の範囲】
【請求項１】
ネイティブマシン命令を実行するプロセサコアを備えた仮想マシン命令プログラムを実行するデータ処理システムであって、
前記プロセサコアと、
メモリと、
前記プログラム実行中にプログラムフローに応じて選択された仮想マシン命令を受け取る仮想マシンインタプリタであって、前記プロセサコアに接続されて、前記プロセサコアにより実行される仮想マシン命令を導入するネイティブマシン命令を発生する仮想マシンインタプリタであって、繰り返し実行されることが予想される一連の前記選択された仮想マシン命令の一群から先頭仮想マシン命令を検出し、前記一群中の前記先頭仮想マシン命令と前記メモリのメモリ位置との対応関係を記録し、前記先頭仮想マシン命令から開始する仮想マシン命令に対して発生した前記一群のためのネイティブマシン命令を前記メモリ位置から前記メモリに書き込み、前記メモリ位置から前記メモリに書き込まれたネイティブマシン命令を実行することにより前記一群のためのネイティブマシン命令を繰り返し実行するよう前記プロセサコアを制御する仮想マシンインタプリタとを備えたことを特徴とするデータ処理システム。
【請求項２】
前記仮想マシンインタプリタは、前記一群の開始時にネイティブブランチバック命令を発生し、このネイティブブランチバック命令を前記メモリ中の前記一群の最後に置くこと特徴とする請求項１に記載のデータ処理システム。
【請求項３】
前記仮想マシンインタプリタは、前記一群が記憶されるアドレスレンジとは重ならないあるアドレスレンジ内のターゲットアドレスを有する無条件なさらなるネイティブブランチ命令を前記ネイティブブランチ命令の後に置き、そして前記プロセサコアのプログラムカウンタをモニタし、そして前記仮想マシン命令の選択と、前記ループ群の開始後に前記アドレスレンジに前記プログラムカウンタの値が入ったときに前記選択された仮想マシン命令からのネイティブマシン命令の発生を再開すること特徴とする請求項２に記載のデータ処理システム。
【請求項４】
前記仮想マシンインタプリタは、少なくとも前記先頭仮想マシン命令
を示し、プログラムフローに影響しないヒント情報を受け取り、前記対応関係を記録し、前記ヒント情報を受け取ったときに所定条件で、前記プログラムフローが前記先頭仮想マシン命令に達したときに前記一群用に前記ネイティブマシン命令を書き込むこと特徴とする請求項２に記載のデータ処理システム。
【請求項５】
ネイティブマシン命令を実行するプロセサコアを用いて仮想マシン命令プログラムを実行する方法であって、
プログラムフローの基に実行すべき仮想マシン命令を選択し、
前記選択された仮想マシン命令から、該選択された仮想マシン命令を実行すべくネイティブマシン命令を決定し、
繰り返し実行されることが予想される一連の前記選択された仮想マシン命令の一群から先頭仮想マシン命令を検出し、
前記先頭仮想マシン命令とメモリ位置との対応関係を記録し、
前記先頭仮想マシン命令から開始する仮想マシン命令から決定される前記一群のためのネイティブマシン命令を前記メモリ位置からメモリに書き込み、
前記メモリ位置から前記メモリに書き込まれた前記一群のためのネイティブマシン命令を実行することにより前記一群のためのネイティブマシン命令を繰り返し実行するようプロセサコアを制御することを特徴とする実行方法。
【請求項６】

前記一群の開始時にネイティブブランチバック命令を発生し、このネイティブブランチバック命令を前記メモリ中の前記ループ群の最後に置くこと特徴とする請求項５に記載の実行方法。
【請求項７】
前記一群が記憶されるアドレスレンジとは重ならないあるアドレスレンジ内のターゲットアドレスを有する無条件なさらなるネイティブブランチ命令を前記ネイティブブランチ命令の後に置き、そして前記プロセサコアのプログラムカウンタをモニタし、そして前記仮想マシン命令の選択と、前記ループ群の開始後に前記アドレスレンジに前記プログラムカウンタの値が入ったときに前記選択と決定を再開すること特徴とする請求項６に記載の実行方法。
【請求項８】
前記先頭仮想マシン命令の検出は、仮想ブランチバック命令により終了するループを検出する前記プログラムを事前処理し、前記先頭仮想マシン命令として前記仮想ブランチバック命令のターゲットアドレスを示す前記プログラムにヒントを加えること特徴とする請求項６に記載の実行方法。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】
この発明は、仮想マシン命令とは異なるネイティブ命令を実行するプロセッサコアを用いて仮想マシン命令プログラムを実行する方法及び装置に関する。
【０００２】
【従来の技術】
仮想マシンプログラムは、コンパイルされたＪＡＶＡプログラム実行時やこの仮想マシン命令プログラムとは異なる命令セットを備えたプロセサによりエミュレートされるプロセサによるプログラム実行時などの様々状況で実行される。
【０００３】
ＪＡＶＡの場合、ＪＡＶＡ言語によるプログラムがまず、ＪＡＶＡバイトコードと呼ばれるＪＡＶＡ仮想マシン命令のプログラムにコンパイルされる。コンパイルは、多くの実行時、即ち、”Ｊｕｓｔ－Ｉｎ－Ｔｉｍｅ”（ＪＩＴ）、ＪＡＶＡプログラム実行前、ＪＡＶＡプログラムのブロック実行前に一度だけ行われる。このプログラム又はブロックはＪＡＶＡ仮想マシン命令にコンパイルされてメモリにロードされる。そしてプロセサによりバイトコードが実行される。このプロセサによってＪＡＶＡ言語に規定されたそれらバイトコードのための効果が得られなければならない。
【０００４】
仮想マシンプログラムはインタプリタにより実行される。このインタプリタは、例えば、プロセサのエミュレーションプログラムにより実行され、又はインストラクションメモリとプロセッサコア間にプリプロセサを配して実行される。インタプリタプログラムは仮想マシン命令をロードし、仮想マシン命令により要求される効果を得るためにどんなアクションをとるかを判断し、そしてそれらのアクションをとるためのネイティブマシン命令を備える。同様にプリプロセサは仮想マシン命令をチェックし、上記要求される効果を得るためのプロセサに対し命令を発する。
【０００５】
両者いずれの場合においても、仮想マシン命令はロードされ直ぐに変換される。各仮想マシン命令は、実行時（又は直ぐ後で実行されるるとき）一つ以上のネイティブマシン命令に変換される。実行時、プログラムフローによりどの各仮想マシン命令を変換するかが指示される。このため、エミュレータが仮想プログラムカウンタを備え、命令実行中カウントアップし、又は仮想マシンブランチ命令を実行するように変更する。
【０００６】
【発明が解決しようとする課題】
通常、仮想マシン命令プログラムの実行エミュレレーションは、同様なネイティブマシン

命令プログラムの実行よりはるかに遅い。これを早めるためのあるタスクを備えたネイティブマシン命令プログラムライブラリが従来知られている。仮想マシン命令プログラムがこのタスクの実行を要求すると、ライブラリからあるプログラムが呼び出されて、エミュレータの管理なしに、タスクを速く実行するようプロセサを制御する。ＪＩＴコンパイラはＶＭの実行を早める他の手段である。しかし、これらはＶＭコードと変換されたコードの両方を記憶する多くのメモリを必要とする。これに加えて、実際の変換が行われる実行フェーズは非常に遅いためこれらの実行タイミングは予測しがたい。

【０００７】

ＰＣＴ特許出願番号９９／１８４８６によれば仮想マシン命令プログラムを実行するプリプロセサは公知である。このプリプロセサは、仮想マシン命令とネイティブマシン命令の実行を非常に早く切り替えることができる。このプリプロセサはプロセサコアのプログラムカウンタを監視する。仮想マシン命令とネイティブマシン命令のそれぞれに所定のプログラムカウンタ値範囲が規定されている。カウント値がネイティブマシン命令用に規定された範囲にある限り、プリプロセサは受動的で、メモリよりプロセサにフェッチされるネイティブマシン命令がプログラムカウンタにより指定される。一方、カウント値が仮想マシン命令用に規定された範囲にあるとプリプロセサは能動的になり、プロセサコア用ネイティブマシン命令はもはやメモリよりフェッチされなくなる。プリプロセサは仮想マシン命令をメモリよりロードし、これら仮想マシン命令よりネイティブマシン命令を発生する。ネイティブマシン命令のプログラムライブラリとして導入されているタスクの実行を仮想マシン命令が要求すると、プリプロセサはプロセサコアを制御してプロセサコアのプログラムカウンタをネイティブマシン命令プログラムの開始点に変更する。これによりネイティブマシン命令プログラム実行速度は速まるが、仮想マシン命令実行速度は依然として遅い。

【０００８】

米国特許番号５，８８９，９９６に記載されているインタプリタはネイティブマシン命令プログラムの、仮想マシン命令に応じた異なるブロックを示している。この特許によれば、これらのブロックはすべてキャッシュメモリにロードされ、インタプリタは仮想マシン命令を実行する度に、適切なブロックに制御を移す。これにより、キャッシュメモリより必要なネイティブマシン命令がフェッチされるのでプグラム実行速度が上がる。しかし、仮想マシン命令実行速度は依然としてネイティブマシン命令プログラム実行速度より遅い。

【０００９】

従って、この発明の目的は、実行速度が早い仮想マシン命令実行方法及び装置を提供するものである。

【００１０】

【課題を解決するための手段】

この発明の装置並びに方法は請求項１並びに請求項５に規定されている。繰り返し実行されることが予想される仮想マシン命令を導入するネイティブマシン命令群の少なくとも一部が規定されている。ネイティブマシン命令群の少なくとも一部、好ましくは、繰り返し実行されることが予想される命令の１セット又はそれ以上のループの開始点での仮想マシン命令に対応する第１のネイティブマシン命令からメモリに記憶される。エミュレータがネイティブマシン命令群の少なくとも一部の開始点の記憶位置とこの開始点に相当する仮想マシン命令の対応関係を記録する。続いて、エミュレータはプロセサコアを制御して、仮想マシン命令の各繰り返し実行のために対応するネイティブマシン命令を発生することなしに、メモリからネイティブマシン命令群の少なくとも一部を実行する。好ましくは、ネイティブマシン命令群はキャッシュメモリに保存される。

【００１１】

繰り返し実行されることが予想されるネイティブマシン命令群の好ましい例はループであり、割り込み命令が実行されることなく命令群は続いて繰り返し実行される。しかし、この発明は、例えば、サブルーチン命令群のように、他の理由で繰り返し実行されるループ

ではない命令群にも適用できる。ループの場合は命令群全体が保存されるとよい。さらに好ましくは、バックワードブランチネイティブマシン命令群が発生され、命令群の最後でメモリ保存されるとよい。これにより、エミュレータのさらなる介入なしにループが繰り返し実行される。代わりに、ループ実行毎にループの一部に対応したネイティブマシン命令を新たにエミュレータより発生してもよい。これにより実行効率は落ちるが、ループ内のある命令がエミュレータによる特別な処理を必要とする場合に効果的である。

【００１２】

好ましくは、エミュレータにより、プロセサコアの命令フェッチアドレスから命令群の命令をプロセサコアが実行中であるかどうか、又はプログラムカウンタが命令群のカウントを終了したかを判断する。後者の場合、エミュレータはネイティブマシン命令の発生を再開する。原理的に、エミュレータは命令群の最終アドレスを記憶し、このアドレスと命令フェッチアドレスを比較することにより、プロセサコアの命令フェッチアドレスにより命令群がフェッチされたことを検出する。しかし、好ましくは、命令群の最後でブランチ命令を追加してプロセサがいくつかの事前に規定されたアドレス範囲（例えば、アドレスの所定ビットが１に等しいが、所定範囲であればよい）に入り込む。この場合、エミュレータは命令フェッチアドレスがその所定範囲にあるか判断すればよい。

【００１３】

原理的に、命令群のネイティブマシン命令は発生されたとき初めてプロセサにより実行され、エミュレータにより記憶される。即ち、命令群全体が発生され記憶される前にネイティブマシン命令は実行される。この場合、プロセサコアはネイティブマシン命令の命令群を２度又はそれ以上実行する。しかし、好ましい実施形態では、エミュレータにより命令群をまず記憶し、そしてメモリから発生したループの命令をプロセサコアが実行開始するよう制御する。そしてプロセサコアは１度メモリからの命令群を実行する。従って、他の実行と異なるループの最初の実行を処理する必要が無く、エミュレータの動作が簡単になる。

【００１４】

この発明においては、命令群の少なくとも一部の最初のネイティブマシン命令とこのネイティブマシン命令が記憶されるメモリ位置の対応関係を記録する必要がある。

【００１５】

これは、例えば、予め規定されたメモリ位置からの命令群の少なくとも一部に対して発生したネイティブマシン命令を記憶し始めることで達成され、これは実行されるある特定のプログラムとは無関係に行われる。

【００１６】

発生したネイティブマシン命令の記憶開始をエミュレータが知るためには、記憶されるべく発生したネイティブマシン命令に対するプログラムの一部の最初の仮想マシン命令を示すヒント情報が仮想マシン命令プログラムと組合わさるとよい。このヒント情報はプログラムの追加命令でもよく、又はプログラムのその部分の最初の仮想マシン命令位置を示すリストであってもよい。例えば、そのような処理に適するループ又はサブルーチン命令群を検出するプログラムを分析することによりこのヒント情報を先にコンパイルしてもよい。

【００１７】

代わりに、プログラム実行中に発生したネイティブマシン命令をほぼ区別無く記憶して、プログラム中の仮想マシン命令位置と対応する発生したネイティブマシン命令のメモリアドレスの組み合わせを記録してもよい。従って、仮想マシンブランチバック命令が発生したら、エミュレータによりブランチの仮想マシン目的位置により対応するネイティブマシン命令が開始するメモリ位置を検出することができる。これにより、プロセサコアはメモリからのこれらの命令を実行開始できる。

【００１８】

【発明の実施の形態】

以下、図面を参照して、この発明の実施形態を説明する。

【００１９】
図１はこの発明の一実施形態である装置を示している。この装置はプロセサコア１０，メモリシステム１４，そしてメモリシステム１４とプロセサコア１０に配された仮想マシンインタプリタ１２を備えている。プロセサコア１０は命令アドレス出力１００を出力し、仮想マシンインタプリタ１２から命令入力１０２を受ける。メモリシステム１４はアドレス入力１４４を受け、仮想マシンインタプリタ１２との間でデータ入出力１４６をやりとりする。メモリシステム１４にはキャッシュメモリ１４０と主メモリ１４２が備えられている。説明を簡単にするために、プロセサコア１０へのオペランドデータ入力は示していない。これはこの発明の理解には必ずしも必要ではないからである。オペランドデータは例えば別々の図示しないデータメモリからプロセサコア１０に対するそれ自身のデータアドレスとデータ値の関係から、プロセサコア１０に与えられる。又は、命令供給との時分割マルチプレックスによりメモリシステム１４から与えられても良い。このシステムはＶＭ命令と発生するネイティブマシン命令に対し物理的に異なるメモリを備えてもよく、ＶＭＩが書き込み接続を有するメモリに対しプロセサコアが読み出し接続を有することになる。

【００２０】
仮想マシンインタプリタ１２は、プリプロセサ１２０、命令マルチプレクサ１２２，アドレスマルチプレクサ１２４そして読み出し／書き込みスイッチ１２６を備える。仮想マシンインタプリタ１２の制御出力は命令マルチプレクサ１２２、アドレスマルチプレクサ１２４、読み出し／書き込みスイッチ１２６の制御入力となる。プロセサコア１０のアドレス出力１００がプリプロセサ１２０の入力となる。プリプロセサ１２０のアドレス出力はアドレスマルチプレクサ１２４を介してメモリシステム１４のアドレス入力１４４となる。プリプロセサ１２０の仮想マシン命令入力はメモリシステム１４のデータ入出力１４６となる。プリプロセサ１２０のネイティブマシン命令出力は命令マルチプレクサ１２２を介してプロセサコア１０の命令入力１０２となる。

【００２１】
これらの入出力関係により、仮想マシンインタプリタ１２が通常動作モードとなり、アドレスマルチプレクサ１２４を介して、またメモリシステム１４のデータ入出力１４２からの仮想マシン命令に応答して、プリプロセサ１２０が仮想マシン命令のアドレスをメモリシステム１４のアドレス入力１４０として発生する。プリプロセサは受け取った仮想マシン命令を解読して、この仮想マシン命令に対してどのネイティブマシン命令をプロセサコア１０が実行すべきかを判断する。そして、これらのネイティブマシン命令を命令マルチプレクサ１２２を介してプロセサコア１０命令入力１０２とする。

【００２２】
プリプロセサ１２０は（プロセサコア１０のプログラムカウンタとは異なる）仮想マシンプログラムカウンタを有し、これによりプログラムフロー中次に実行すべき仮想マシン命令のプログラムにおける位置を検出する。通常モードでは、プリプロセサ１２０自身が仮想マシン命令を取り扱い、これがプリプロセサのプログラムカウンタに働きかける。ブランチ命令の状態検出以外には、この目的のためにネイティブマシン命令を発生する必要はない。原理的には、このように通常モードで、繰り返し実行されるプログラムループとサブルーチンを含み、すべての仮想マシン命令が実行可能である。しかし、この発明では繰り返し実行されると予想されるプログラムのそのような部分には特別な処理が（必要ではないが）施されることもある。

【００２３】
ループ動作モードでは、仮想マシンインタプリタ１２は発生したネイティブマシン命令を直接プロセサコア１０に与えることはない。代わりに、仮想マシンインタプリタはこれらの命令をメモリシステム１４に記憶する。これは、メモリシステム１４のアドレス入力１４０に、アドレスマルチプレクサ１２４を介して、また、データ読み出し／書き込みスイッチ１２６を介してデータ入出力１４２にネイティブマシン命令を与えることにより行われる。この目的のため、データ読み出し／書き込みスイッチ１２６を介してネイティブマ

シン命令出力がメモリシステム１４のデータ入出力１４２に与えられる。
【００２４】
ネイティブ実行モードでは、プロセサコア１０がメモリシステム１４からのネイティブマシン命令を実行する。この目的のため、プロセサコア１０の命令アドレス出力１００が、アドレスマルチプレクサ１２４を介して、メモリシステム１４のアドレス入力１４０に与えられる。そしてプロセサコア１０の命令入力１０２が命令マルチプレクサ１２２を介してメモリシステム１４のデータ入出力１４２に与えられる。
【００２５】
図２は仮想マシンインタプリタ１２の動作を示すフローチャートである。通常動作モードでは、仮想マシンインタプリタ１２は、最終ステップ２９で仮想マシン命令の終了（又は仮想マシンプログラムを停止させるバイトコードの実行）を検知するまで第一ステップ２１，第二ステップ２２，第三ステップ２３を繰り返す。フローチャートのステップ２１において、仮想マシンインタプリタ１２はメモリシステム１４からある仮想マシン命令をロードし、この命令がループの開始命令か否かを判定する。開始命令でない場合、仮想マシンインタプリタ１２は第二ステップ２２を実行する。第二ステップ２２では、仮想マシンインタプリタ１２は第一ステップ２１でロードされた仮想マシン命令を解読し、この仮想マシン命令を実行する少なくとも一つのネイティブマシン命令を発生する。このネイティブマシン命令発生過程はすでに引用したＰＣＴ特許出願番号９９／１８４８６及びそれに対する引例に記載されている。第三ステップ２３で、仮想マシンインタプリタ１２は発生したネイティブマシン命令又をプロセサコア１０に与え、プロセサコアからの命令アドレス出力ををモニタして次の仮想マシン命令をいつ実行すべきか判定する。次の仮想マシン命令を実行するときは最終ステップ２９を実行してプログラムが終了したか判断する。終了してない場合は、第一ステップ２１，第二ステップ２２，第三ステップ２３を繰り返す。第一ステップ２１，第二ステップ２２，第三ステップ２３へ戻るのは無条件に行われてもよく、仮想マシンプログラムを停止させるバイトコードに応答してソフトウエアトラップ命令を実行してロープから抜けても良い。
【００２６】
通常動作モードでは、プロセサコア１０から発生した命令アドレスは命令を特定するのには用いられない。仮想マシンインタプリタ１２の仮想マシンプログラムカウンタがメモリシステム１４内のアドレスを制御しここから仮想マシン命令をフェッチする。仮想マシンインタプリタ１２はプロセサコア１０から発生した命令アドレスを用いてプロセサコア１０の状態テストを行ってもよい。これは、ネイティブ条件ブランチ命令をプロセサコア１０に与えプロセサコア１０がこのブランチを受け入れるかを見ることにより行われる。さらに、仮想マシンインタプリタ１２は、少なくとも二つの命令アドレスレンジを規定し、その一の最上位ビットを１とすることにより、プロセサコア１０から発生した命令アドレスをモード選択に用いてもよい。第１のレンジ内でプロセサコア１０がアドレスを発生する間は、上記したように、仮想マシンインタプリタ１２は通常モードで仮想マシン命令を解読する。しかし、プロセサコア１０が発生したアドレスが第１のレンジ以外であると、仮想マシンインタプリタ１２は別のモード、例えば、ネイティブモードで動作し、プロセサコア１０がメモリシステム１４からネイティブマシン命令をフェッチし実行する。
【００２７】
第一ステップ２１で、仮想マシンインタプリタ１２がループの開始命令がロードされたことを検出すると、仮想マシンインタプリタ１２は第四ステップ２４，第五ステップ２５，第六ステップ２６、第七ステップ２７，第八ステップ２８を実行する。第四ステップ２４は第二ステップ２２と似ており、現在の仮想マシン命令を実行するための一つ以上のネイティブマシン命令を発生する。しかし、第五ステップ２５では仮想マシンインタプリタ１２は、この（これらの）ネイティブマシン命令を、第三ステップ２３で実行する代わりに、メモリシステム１４に記憶する。第六ステップ２６で仮想マシンインタプリタ１２は次の仮想マシン命令をロードし、この仮想マシン命令がループの最終命令ではない場合は第四ステップ２４に戻る。このようにして、仮想マシンインタプリタ１２は発生した一連の

ネイティブマシン命令をメモリシステム１４に記憶する。仮想マシンインタプリタ１２が仮想マシン命令のループの最後まで達したことを検知したら、第七ステップ２７，第八ステップ２８を実行する。第七ステップ２７では、ループに対して発生した一連の命令の最後で、仮想マシンインタプリタ１２がネイティブ条件ブランチバック命令をメモリシステム１４に記憶する。ネイティブ条件ブランチバック命令の後、仮想マシンインタプリタ１２は無条件ブランチ命令を所定のレンジ内の命令アドレスに記憶する。この所定レンジは、例えば、プロセサコア１０から発生する仮想マシンインタプリタ１２が通常モードで動作すべきアドレスから始まるレンジである。

【００２８】

第八ステップ２８で、仮想マシンインタプリタ１２はブランチ命令をプロセサコア１０に与えて、ループに対して発生した一連のネイティブ命令をフェッチする。この点に関して、第八ステップ２８は第三ステップ２３に似ている。ただし、第八ステップ２８ではプロセサコア１０がメモリシステム１４からのネイティブ命令を実行するのに対し、第三ステップ２３ではネイティブ命令は仮想マシンインタプリタ１２から供給される点が異なる。第八ステップ２８において、プロセサコア１０が発生した命令アドレス値を仮想マシンインタプリタ１２がモニタしてプロセサコア１０がループから抜けたことを検知するまで、プロセサコア１０はメモリシステム１４から命令をフェッッチし実行する。そして、仮想マシンインタプリタ１２は最終ステップ２９を実行して通常モード動作を再開する。

【００２９】

図２のフローチャートは、ネイティブ命令が発生しメモリに記憶された後にループが実行される例を示している。これがこの発明の最も簡潔な応用例であるが、ループ以外の繰り返し実行されると予想される他の仮想マシン命令セットにもこの発明は応用できる。例としては、サブルーチン、例外的ハンドラ又はより大きなループの最も処理しやすい命令群等である。これらの場合、記憶されたネイティブマシン命令は発生した命令の最後にブランチバックを含まない。一連の仮想マシン命令の実行を仮想マシンプログラムが呼び出す度に、記憶された対応するネイティブ命令に制御が戻る。

【００３０】

図２の実施形態では、繰り返し実行される仮想マシン命令セットに対して発生した第１のネイティブ命令からのみ、仮想マシンインタプリタ１２が発生したネイティブマシン命令のメモリシステム１４への書き込みを開始する。仮想マシンインタプリタ１２がネイティブマシン命令の書き込みを開始するメモリアドレスは、実行される仮想マシン命令プログラムとは無関係な、所定のメモリアドレスでもよい。これにより、ループ外の仮想マシン命令に対して発生するネイティブマシン命令の書き込みにオーバヘッドが生ずることがなくなる。さらには、メモリシステム１４にキャッシュメモリを用いるとキャッシュ交換量がこのように減少する。

【００３１】

しかし、これは仮想マシンインタプリタ１２が、発生したネイティブマシン命令をいつ書き込み始めるかを示す信号が必要であることを意味する。所定数の仮想マシン命令を所定回数実行することを示すループ命令を仮想マシン命令が含む場合は、発生したネイティブマシン命令を記憶開始するためのトリガーとして用いることができる。代わりに、解読結果がメモリに記憶されるべき命令群の開始を示すメタ命令を仮想マシン命令プログラムに含んでもよい。この場合、メタ命令により、第１ステップにおいて仮想マシンインタプリタ１２が次の仮想マシン命令をロードし、第４ステップ２４から実行する。なお、必ずしも必要ではないがメタ命令は仮想マシン命令群のサイズを示しても良く、命令群の位置に等しいブランチターゲットを用いて仮想マシンブランチバック命令から命令群の最後が検出される。代わりに、そのような命令群の先頭にある仮想マシン命令のアドレスリストと仮想マシンプログラムを組み合わせても良い。この場合、第１ステップ２１で、仮想マシンインタプリタ１２がこのリストの少なくとも一部をロードして現在の仮想マシン命令アドレスと比較する。

【００３２】

仮想マシン命令の位置は、例えば、高水準言語から仮想マシン命令へのコンパイル最中に、コンパイルされた高水準言語ループ（例えばループ用）位置として検出してもよい。代わりに、仮想マシンプログラムをを事前処理してこれらの位置を検出し、ブランチバック命令を検出し、そして検出されたブランチバック命令のターゲットをループの開始点としてマークしてもよい。従って、高水準言語ソースコードが不要となる。同様に、サブルーチンの開始点（仮想マシン命令がサブルーチン命令にジャンプするためのターゲット）及び／又はキャッシングの例外としての命令をこのようにマークしてもよい。

【００３３】

最も簡潔な実施形態としては、メモリから発生したネイティブマシン命令シーケンスの実行を、ブランチ命令を含まない仮想マシン命令セット又はこのセットの最後のブランチバック以外の他の制御転送命令用のネイティブマシン命令シーケンスのみに適用してもよい。これにより、毎回実行すべきネイティブマシン命令シーケンスが同じになる。もし、セットの最後には位置しないブランチ命令を仮想マシン命令セットが含む場合は多くの手法がとれる。第１の手法は通常モードでのこのセットの実行であり、即ち、毎回仮想マシン命令の解読を実行し、メモリからの解読された命令は実行しない。

【００３４】

仮想マシン命令セット内のブランチのための第２の手法は、ネイティブマシン命令セットを、同じネイティブマシン命令と共にいつも実行される複数部分に分岐し、これら部分用に記憶されたネイティブマシン命令を記憶することである。この場合、仮想マシン命令セットの最後にはない制御命令の転送において終了する各部分のネイティブマシン命令の終了を仮想マシンインタプリタ１２が検出する。これは、通常モードでの実行を示すレンジにジャンプする各部分の最後へのネイティブマシン命令の挿入と組み合わせたい場合は、プロセサコア１０により発生した命令アドレスをモニタすることにより行ってもよい。ネイティブマシンブランチ命令のブランチターゲットは終了した部分を示すのに用いてもい。この部分の実行終了時に、制御転送を命令する仮想マシン命令を仮想マシン命令が処理し、そしてプロセサコアにメモリシステム１４からの、仮想マシン命令セットの選択された部分のために発生したネイティブマシン命令を実行する。又は命令セット以外に仮想マシン命令が制御を転送する場合は、通常モードが再開する。

【００３５】

第３の手法は、ネイティブマシン命令間のネイティブマシンブランチ命令を用いて、命令セット内で制御転送命令を実行するそれら制御を転送を発生することある。これによりネストループが実現できる。

【図面の簡単な説明】

【図１】

この発明の装置の一実施形態を示す図である。

【図２】

仮想マシンプログラムをエミュレートする方法を示すフローチャートを示す図である。

【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
7 March 2002 (07.03.2002)

PCT

(10) International Publication Number
**WO 02/19100 A1**

(51) International Patent Classification[7]: G06F 9/318, 9/32, 9/455, 9/38

(21) International Application Number: PCT/EP01/09694

(22) International Filing Date: 22 August 2001 (22.08.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
00203024.5 31 August 2000 (31.08.2000) EP

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

(72) Inventors: STEINBUSCH, Otto, L.; Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL). LINDWER, Menno, M.; Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL).

(74) Agent: DE JONG, Durk, J.; Internationaal Octrooibureau B.V., Prof. Holstlaan 6, NL-5656 AA Eindhoven (NL).

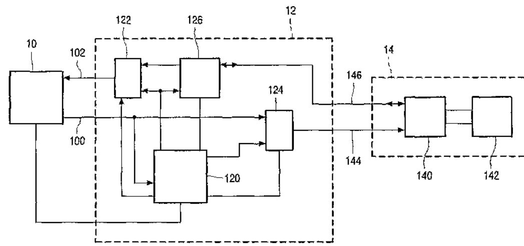
(81) Designated State *(national)*: JP.

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR).

**Published:**
— *with international search report*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

*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:** SYSTEM FOR EXECUTING VIRTUAL MACHINE INSTRUCTIONS




WO 02/19100 A1

**(57) Abstract:** A data processing system has a processor core, memory and a virtual machine interpreter. The virtual machine interpreter receives virtual machine instructions selected dependent on program flow during execution of a virtual machine program. The virtual machine interpreter generates native machine instructions that implement the virtual machine instructions for execution by the processor core. The virtual machine interpreter identifies an initial virtual machine instruction from a body of virtual machine instructions, where the body is expected to be executed repeatedly. The virtual machine interpreter records a correspondence between the initial virtual machine instructionin the body and a memory location in the memory and writes native instructions for the body into the memory from said memory location. The processor core executes the native instructions for the body and repeats execution of the native instructions for the body by executing the written native machine instructions for the body from memory starting from said memory location.

System for executing virtual machine instructions

The field of the invention is a method and device for executing a program of virtual machine instructions with a processor core that is arranged to execute native instructions different from the virtual machine instructions.

Execution of a virtual machine program can be used under various circumstances, such as during execution of compiled JAVA programs or when execution of the program by a processor has to be emulated by a processor with a different instruction set.

In case of JAVA, a program in the JAVA language is first compiled into a program of JAVA virtual machine instructions, instructions, which are commonly called JAVA byte codes. Compilation may be performed once for a number of executions, or "Just-In-Time" (JIT), just before execution of the JAVA program or blocks of the JAVA program. The program or block is compiled into JAVA virtual machine instructions, loaded into memory and execution of the byte codes by a processor is started. The processor has to ensure that the effect is produced that has been defined for those byte codes in the JAVA language definition.

Execution of a virtual machine program can be realized with an interpreter, which is implemented for example by executing an emulator program with the processor or by inserting a preprocessor between instruction memory and a processor core. An interpreter program contains native machine instructions to load the virtual machine instructions, to determine what actions to take in order to produce the effect required by the virtual machine instructions and to take those actions. Similarly a preprocessor inspects the virtual machine instructions and generates instructions for the processor core that produce the required effect.

In both cases virtual machine instructions are loaded and translated "on the fly": each virtual machine instruction is translated into one or more native machine instructions when this virtual machine instruction has to be executed (or when it is expected that it has to be executed soon). Program flow during execution dictates which virtual machine instructions are translated. To realize this, the emulator has to maintain a virtual program counter, which the emulator must increment during sequential instruction execution or which the emulator must change upon executing a virtual machine branch instruction.

Usually, emulation of execution of a program of virtual machine instructions is much slower than execution of a similar program of native instructions. It has been known to speed up processing by providing a library of programs of native instructions to implement certain tasks. When a program of virtual machine instructions calls for the execution of such a task, a program from the library is given control over the processor, so that the task is executed much faster, without the supervision of the emulator. JIT compilers are another way of speeding up VM execution. However, they consume much more memory, because they need to store both VM code and translated code. Besides that, their timing behaviour is inpredictable, because the execution phase in which the actual translation takes place is very slow.

From PCT patent application No. 99/18486 a preprocessor is known for implementing the execution of virtual machine programs. This preprocessor is capable of switching very rapidly between execution of virtual machine instructions and native machine instructions. The preprocessor monitors the program counter of the processor core. Certain ranges of program counter values have been defined for virtual machine instructions and for native machine instructions. As long as the program counter is in a range defined for native machine instructions, the preprocessor is passive and the bus control unit allows native instructions addressed by the program counter to be fetched from memory to the processor. When the program counter is in a range defined for virtual machine instructions, the preprocessor steps in. In this case, the native instructions for the processor core are no longer fetched from memory. The preprocessor loads virtual machine instructions from memory and from these virtual machine instructions it generates the native machine instructions. When a virtual machine instructions calls for execution of a task that is implemented as a library program of native machine instructions, the preprocessor causes the processor core to change theprogram counter of the processor core to the starting point of the program of native machine instructions. This speeds up execution of programs of such native machine instructions, but execution of virtual machine instructions is still much slower.

US patent No. 5,889,996 describes an interpreter which contains different blocks of native machine instructions, each block for a respective type of the virtual machine instruction. According to this patent these blocks are all loaded into cache memory together and each time the interpreter executes a virtual machine instruction the interpreter transfers control to the appropriate block. Thus, program execution is accelerated because the required native machine instructions can be fetched from cache memory. However, virtual machine program execution is still slower than native machine program execution.

Among others, it is an object of the invention to provide for a method and device for executing virtual machine programs that speeds up execution of virtual machine programs.

5

The device according to the invention is set forth in Claim 1 and the method according to the invention is set forth in claim 5. At least part of a body of native machine instructions that are generated to implement virtual machine instructions that are expected to be executed repeatedly is identified. Thus at least part of the body is written to memory, 10 preferably starting from the first native machine instruction that corresponds to the virtual machine instruction at the start of a loop, a loop being a set of one or more instructions which is to be executed repeatedly. The emulator records a correspondence between a memory location of the start of the at least part of the body and an identity of the virtual machine instruction corresponding to the start of the at least part of the body. Subsequently the 15 emulator enables the processor core to execute native machine instructions of the at least part of the body repeatedly from memory, without generating these native instructions anew for each repeated execution of the corresponding virtual machine instructions. Preferably, the body is kept in a cache memory.

A preferred example of a body of native machine instructions that is expected 20 to be executed repeatedly is a loop, where the repeated executions of the body are consecutive, without execution of intervening instructions. But the invention may also be applied to bodies of instructions that are not a loop body, but are expected to be executed repeatedly for another reason, for example a subroutine body. In case of a loop, the entire body of the loop is preferably stored. Furthermore, a backward branch native machine 25 instruction is preferably generated and stored in memory at the end of the body of native machine instructions, so that the loop can be executed repeatedly without further intervention of the emulator. As an alternative, the emulator might generate native instructions for part of the loop anew for each time that the loop is executed. This reduces the efficiency of execution, but it may be advantageous, for example if some instructions in the loop require 30 special treatment by the emulator.

Preferably, the emulator detects from the instruction fetch addresses of the processor core whether the processor core is still executing instructions from the body, or whether the program counter passes out of the body. In the latter case, the emulator resumes the generation of native machine instructions. In principle, the emulator can detect that the

instruction fetch address of the processor core passes out of the body if the emulator stores an address of the end of the body and compares this address with the instruction fetch address. Preferably, however the emulator adds a branch instruction at the end of the body, for causing the processor core to branch into some predefined range of addresses (for example a
5 range where a predetermined bit of the address is equal to one, but any range within certain bounds will do). In this case, it suffices that the emulator detects whether the instruction fetch address is in that predefined range.

In principle, the native machine instructions in the body can be executed for the first time by the processor core as they are generated and stored by the emulator, that is,
10 the native instructions can be executed before the instructions of the entire body have been generated and stored. In this case the processor core executes the native machine instructions of the body from memory only the second and further times the instructions are executed. However, in a preferred embodiment, the emulator first stores the entire body and then causes the processor core to start executing the generated instructions of the loop from memory.
15 Thus, the processor core executes the body from memory also the first time. Thus, there is no need to treat the first execution of the loop different from other executions, which simplifies the operation of the emulator.

For the implementation of the invention it is necessary to record a correspondence between a native machine instruction at the start of the at least part of the
20 body and a memory location where that native machine instruction is stored.

This may be realized for example by starting storage of generated native machine instructions for the at least part of the body from a predefined memory location, which is independent of the particular program being executed. In order that the emulator knows when to start storing the generated native machine instructions, the program of virtual
25 machine instructions is preferably accompanied by hint information that indicates a virtual machine instruction at the start of a part of the program for which generated native machine instructions must be stored. This hint information may be in the form of an additional instruction in the program, or in the form of an entry in list that indicates the location virtual machine instructions at the start of such parts of the program. Such hint information may be
30 compiled in advance, for example by analyzing the program to detect loops or subroutine bodies suitable for such treatment.

Alternatively, one may store native machine instructions more or less indiscriminately as they are generated during program execution, and record the combination of a location of virtual machine instructions in the program and a memory addresses of

corresponding generated native machine instructions. Thus, when a virtual machine branch back instruction occurs, the emulator can use the virtual machine target location of the branch to determine the memory location where the corresponding native machine instructions start, so that the processor core can start executing these instructions from memory.

5

These and other advantageous aspects of the method and device according to the invention will be described using the following Figures, of which

Figre 1 shows an embodiment of a device according to the invention.

Figure 2 shows a flow chart of emulation of a virtual machine program.

10

Figure 1 shows an embodiment of a device according to the invention. The device contains a processor core 10, a memory system 14 and a virtual machine interpreter 12 between the memory system 14 and the processor core 10. The processor core 10 has an instruction address output 100 and an instruction input 102 coupled to the virtual machine
15 interpreter 12. The memory system 14 has an address input 144 and a data input/output 146 coupled to the virtual machine interpreter 12. The memory system 14 is shown to contain a cache memory 140 and a main memory 142. For the sake of simplicity, connections for supplying operand data to processor core 10 are not shown in Figure 1, because such connections are not essential for understanding the invention. Operand data may be supplied
20 to the processor core 10 for example using a separate data memory (not shown), with its own data address and data value connection (not shown) to the processor core 10, or from memory system 14 in time share multiplexing with the supply of instructions. The system may have physically different memories for VM instructions and generated native instructions, provided that the processor core has read connections to the same memory to which the VMI
25 has write connections.

The virtual machine interpreter 12 contains a preprocessor 120, an instruction multiplexer 122, an address multiplexer 124 and a read/write switch 126. Virtual machine interpreter 12 has control outputs coupled to control inputs of instruction multiplexer 122, address multiplexer 124 and read/write switch 126. The preprocessor 120 has an input
30 coupled to the address output 100 of processor core 10, an address output coupled to the address input 144 of memory system 14 via address multiplexer 124, a virtual machine instruction input coupled to the data input/output 146 of memory system 14 and a native machine instruction output coupled to the instruction input 102 of the processor core 10 via the instruction multiplexer 122.

These connections serve a normal mode of operation of the virtual machine interpreter 12, in which the preprocessor 120 issues addresses of virtual machine instructions to the address input 140 of the memory system 14 via address multiplexer 124 and in response receives virtual machine instructions from the data input/output 142 of the memory
5 system. The preprocessor analyzes the received virtual machine instructions, determines which native machine instructions should be executed by the processor core 10 to implement the received virtual machine instructions and supplies these native machine instructions to the instruction input 102 of the processor core 10 via instruction multiplexer 122.

The preprocessor 120 maintains its own virtual machine program counter
10 (distinct from the program counter of the processor core 10), which determines the location in the program of the next virtual machine instruction that should be executed during program flow. In the normal mode, the preprocessor 120 itself handles virtual machine branch instructions, which affect the program counter in the preprocessor. No native machine instructions need be generated for this purpose, except to determine any conditions for the
15 branch instruction. In principle, all virtual machine instructions can be processed in this way in the normal mode, including instructions in program loops which are executed repeatedly and subroutines. However, according to the invention, a special treatment may (not "need") be given to such parts of the program that are expected to be executed repeatedly.

In a loop mode of operation, the virtual machine interpreter 12 does not supply
20 generated native machine instructions directly to the processor core 10. Instead, the virtual machine interpreter stores these instructions in memory system 14, by supplying storage addresses to the address input 140 of memory system 14 via address multiplexer 124 and by supplying the native instructions to the data input/output 142 via read/write switch 126. For this purpose, the native machine instruction output is coupled to the data input/output 142 of
25 memory system 14 via the read/write switch 126.

In a native execution mode, the processor core 10 is allowed to execute native machine instructions from memory system 14. For this purpose, the instruction address output 100 of processor core 10 is coupled to the address input 140 of memory system 14 via address multiplexer 124. And instruction input 102 of processor core 1 is coupled to the data
30 input/output 142 of memory system 14 via the instruction multiplexer 122.

Figure 2 shows a flow-chart of operation of the virtual machine interpreter 12. In the normal mode, virtual machine interpreter 12 repeatedly executes a first, second and third step 21, 22, 23 until a final step 29 detects an end of the virtual machine program (or executes a byte code that has the effect of terminating execution of the virtual machine

program). In the first step 21 of the flow-chart, the virtual machine interpreter loads a virtual machine instruction from memory system 14 and determines whether this instruction is a starting instruction of a loop. If not, virtual machine interpreter executes the second step 22. In the second step 22, the virtual machine interpreter 12 analyzes the virtual machine
5 instruction that has been loaded in the first step 21 and generates one or more native machine instructions that implement the virtual machine instruction. The process of generation of native machine instructions has been described in PCT patent application No. 99/18486 cited hereinbefore and its references. In the third step 23, virtual machine interpreter supplies the generated native machine instruction or instructions to processor core 10 and monitors the
10 instruction address output of processor core to determine when a next virtual machine instruction must be processed. When a next virtual machine instruction must be processed, the final step 29 is executed, to determine whether the program has finished. If not, the first, second and third step 21, 22, 23 are repeated. Alternatively, the return to the first, second and third step 21, 22, 23 is unconditional, an exit from the loop being realized by executing a
15 software trap instruction in response to byte codes that cause the virtual machine program to terminate.

In the normal mode, the exact instruction address issued by the processor core is not used to address instructions. Virtual machine interpreter 12 keeps its own virtual machine program counter to control the addresses in memory system 14 from which it
20 fetches virtual machine instructions. Virtual machine interpreter 12 may use the instructions addresses issued by processor core 10 to test the state of processor core 10, by supplying native conditional branch instructions to processor core 10 and observing whether or not processor core 10 takes the branch. Furthermore, virtual machine interpreter 12 may use the instruction address issued by the processor core 10 for the purpose of mode selection: at least
25 two ranges of instruction addresses are defined, one for example having an MSB (most significant bit) equal to one. As long as the processor core 10 issues addresses in a first range, the virtual machine interpreter 12 operates in the normal mode translating virtual machine instructions as described. However, when the instruction addresses issued by the processor core 10 are not in the first range, the virtual machine interpreter operates in a different mode,
30 for example a native mode, in which processor core 10 is allowed to fetch native instructions from memory system 14 for execution.

When virtual machine interpreter 12 determines in first step 21 that a starting instruction of a loop has been loaded, virtual machine interpreter 12 executes a fourth, fifth, sixth, seventh and eight step 24, 25, 26, 27, 28. The fourth step 24 is similar to the second

step 22, in that one or more native machine instructions are generated that implement a current virtual machine instruction. However, in the fifth step 25, virtual machine interpreter 12 stores this native machine instruction or these native machine instructions in memory system 14, instead of executing them as in the third step 23. In the sixth step 26, virtual
5 machine interpreter 12 loads a subsequent virtual machine instruction en repeats from the fourth step 24 if the subsequent virtual machine instruction is not the final instruction of the loop. Thus, virtual machine interpreter 12 stores a sequence of generated native machine instructions in memory system. When virtual machine interpreter 12 determines that it has reached the end of the loop of virtual machine instructions, the seventh and eight steps 27, 28
10 are executed. In the seventh step 27 virtual machine interpreter 12 stores a native conditional branch back instruction in memory system 14 at the end of the sequence of instructions that have been generated for the loop. After the branch back instruction, the native machine interpreter 12 stores an unconditional branch instruction to an instruction address in a predetermined range, for example the range from which the addresses, when issued by
15 processor core 10, indicate that virtual machine interpreter 12 must operate in the normal mode.

In the eight step 28, virtual machine interpreter 12 supplies a branch instruction to processor core 10, to cause the processor core 10 to start fetching instructions from the sequence of native instructions that has been generated for the loop. In this respect,
20 eight step 28 is similar to third step 23, except that in eight step 28 the processor core 10 addresses and executes native instructions from memory system 14, whereas in third step 23 the native instructions are generated by and supplied from virtual machine interpreter 12. In the eight step 28, virtual machine interpreter 12 allows processor core 10 to fetch and execute instructions from memory system 14 until virtual machine interpreter 12 detects from the
25 value of the instruction address issued by the processor core 10 that the processor core 10 has exited from the loop. Thereupon, virtual machine interpreter 12 executes final step 29, to resume operation in the normal mode.

By way of example, the flow chart of Figure 2 has been described for the case of a loop, which is executed immediately after generation of the native instructions and their
30 storage in memory. This is the most compact example of implementation of the invention, but the invention may be applied to other sets of virtual machine instructions than loops, when such a set of virtual machine instructions are expected to be executed repeatedly. An example is the body of a subroutine, or an exception handler, or the most computation sensitive part of a larger loop. In these cases, the stored native machine instructions will not

contain a branch back at the end of the generated instructions. Control is transferred back to the stored native machine instructions each time the virtual machine program calls for the execution of the corresponding series of virtual machine instructions.

In the embodiment shown in Figure 2, the virtual machine interpreter 12 starts
5 writing generated native machine instructions to memory system 14 only from the first native instruction generated for a set of repeatedly executed virtual machine instructions. The starting memory address from which virtual machine interpreter 12 starts writing these native machine instructions may be a predetermined memory address, which is independent of the virtual machine program being executed. This is advantageous, in that no unnecessary
10 overhead is involved in writing native machine instructions generated for virtual machine instructions outside the loop. Furthermore, if the memory system 14 uses a cache, the required amount of cache replacement is reduced in this way.

However, this approach means that the virtual machine interpreter 12 needs a signal to indicate when it should start writing generated native machine instructions to the
15 memory system 14. In case the virtual machine instructions include a "loop" instruction, which indicates that a specified number of virtual machine instructions must be executed a specified number of times, the loop instruction can be used as a trigger to start storing generated native machine instructions. Alternatively, a "meta-instruction" may be included in the virtual machine program to indicate the start of a body of instructions whose translation
20 must be stored in memory. In this case, the meta-instruction causes the virtual machine interpreter 12 in the first step 21 to load the next virtual machine instruction and to proceed from the fourth step 24. The meta-instruction may indicate the size of this body of virtual machine instructions, but this is not necessary: the end of the body may be detected from a virtual machine branch back instruction with a branch target equal to the location of the start
25 of the body. Alternatively, the virtual machine program may be combined with a list of addresses of virtual machine instructions at the start of such bodies. In this case, virtual machine interpreter loads at least part of this list and compares the current virtual machine instruction address in the first step 21.

The location of the virtual machine instruction may be determined for example
30 during compilation of a high level language into virtual machine instructions, as the location where high level language loops (e.g. for loops) have been compiled. Alternatively, these locations may be determined by preprocessing the virtual machine instruction program, to detect branch back instructions and to mark the target of detected branch back instructions as starting points of loops. Thus, the high level language source code is not required. Similarly,

the starting points of subroutines (targets of virtual machine jump to subroutine instructions) and/or instructions for catching exceptions may be marked in this way.

In the simplest embodiment, the execution of generated native instruction sequences from memory is applied only to native instruction sequences for sets of virtual machine instructions that do not contain branch instructions or other control transfer instructions other than a branch back at the end of the set. This guarantees that the same sequence of native instructions must be executed each time. If the set of virtual machine instructions contains a branch instruction that is not at the end of the set a number of measures can be taken. A first solution is to execute the set in the normal mode, i.e. by translation each time the virtual machine instructions are executing and not to execute the translated instructions from memory.

A second solution for branches inside the set of virtual machine instructions is to split the set of native instructions into parts that, if executed, are always executed implemented with the same native machine instructions and to store the native machine instructions stored for the various parts. In this case, the virtual machine interpreter 12 detects completion of the native machine instructions of each part that ends in a transfer of control instruction that is not at the end of the set of virtual machine instructions. This may be realized by monitoring the instruction addresses issued by the processor core 10, if desired in combination with the insertion of a native machine branch instruction at the end of each part to jump into the range identifying execution in the normal mode. The branch target of the native machine branch instruction may be used to identify the part that has completed. Upon completion of execution of a part, the virtual machine instruction takes care of virtual machine instructions that command transfer of control and subsequently causes the processor core to start executing from memory system 14 those native instructions that where generated for the selected part of the set of virtual machine instructions. Or, if the virtual machine instruction transfers control out of the set of instructions, normal mode execution may be resumed.

A third solution is to generate native machine instructions that implement the "transfer of control" instructions inside the set of instructions by means of appropriate native machine branch instructions between the generated native machine instructions. Thus nested loops can be realized.

CLAIMS:

1. A data processing system for executing a program of virtual machine instructions with a processor core that is arranged to execute native instructions comprising
- the processor core;
- a memory;
5 - a virtual machine interpreter for receiving virtual machine instructions selected dependent on program flow during execution of the program, the virtual machine interpreter being coupled to the processor core to generate native machine instructions that implement the virtual machine instructions for execution by the processor core, the virtual machine interpreter being arranged
10 - to identify an initial virtual machine instruction from a body of successive ones of the selected virtual machine instructions, where the body is expected to be executed repeatedly;
- to record a correspondence between the initial virtual machine instruction in the body and a memory location in the memory;
15 - to write native instructions for the body into the memory from said memory location, the native instructions for the body being generated for virtual machine instructions starting from the initial virtual machine instruction;
- to cause the processor core to execute the native instructions for the body and to repeat execution of the native instructions for the body by executing the written native
20 machine instructions for the body from memory starting from said memory location.

2. A data processing system according to Claim 1, the virtual machine interpreter being arranged to generate a native branch back instruction to the a start of the body and placing the native branch back instruction at the end of the body in the memory.

25

3. A data processing system according to Claim 2, the virtual machine interpreter being arranged to place an unconditional further native branch instruction behind the native branch instruction, the unconditional further native branch instruction having a target address in a range of addresses that does not overlap a further range of addresses in which the body is

stored, the virtual machine interpreter being arranged to monitor a program counter address of the processor core and to resume selection of the virtual machine instructions and generation of native machine instructions from the selected virtual machine instructions when the program counter address enters said range of addresses after execution of the loop body.

5

4. A data processing system according to Claim 1, the virtual machine interpreter being arranged to receive hint information, which does not affect program flow, the hint information indicating at least said initial virtual machine instruction, the virtual machine interpreter recording said correspondence and writing the native instructions for the body

10 when program flow reaches the initial virtual machine instruction, conditional upon receiving said hint information.

5. A method of executing a program of virtual machine instructions with a processor core that is arranged to execute native instructions, the method comprising

15 - selecting, under control of program flow, virtual machine instructions to be executed;

- determining native instructions from the selected virtual machine instructions, to implement the selected virtual machine instructions;

- identifying an initial virtual machine instruction from a body of successive

20 ones of the selected virtual machine instructions that is expected to be executed repeatedly;

- recording a correspondence between the initial virtual machine instruction and a memory location;

- writing native instructions for the body into a memory from said memory location, the native instructions for the body being determined from virtual machine

25 instructions starting from the initial virtual machine instruction ;

- causing the processor core to execute the native instructions for the body and to repeat execution of the native instructions for the body by executing the written native machine instructions for the body from memory starting from said memory location.

30 6. A method according to Claim 5, comprising generating a native branch back instruction to a start of the body and placing the native branch back instruction at the end of the loop body in the memory.

7. A method according to Claim 6, comprising placing an unconditional further native branch instruction behind the native branch instruction, the unconditional further native branch instruction having a target address in a range of addresses that does not overlap a further range of addresses in which the body is stored, the method comprising the step of
5 monitoring a program counter address of the processor core and to resume said selecting and determining when the program counter address enters said range of addresses after execution of the loop body.

8. A method according to Claim 4, said identifying comprising preprocessing the
10 program to detect loop terminating with a virtual branch back instruction and adding a hint to the program which identifies a target address of the virtual branch back instruction as the initial virtual machine instruction.

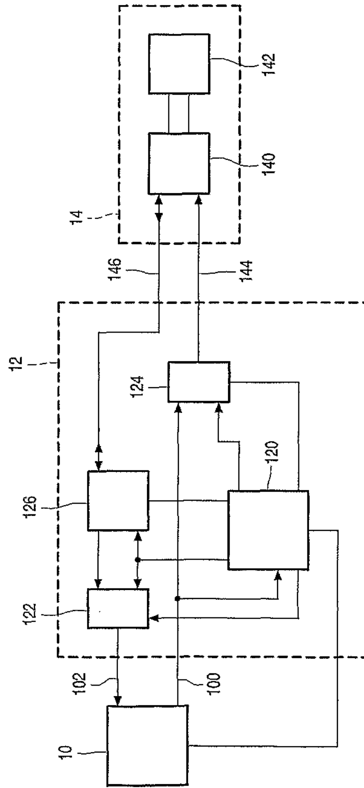



FIG. 1

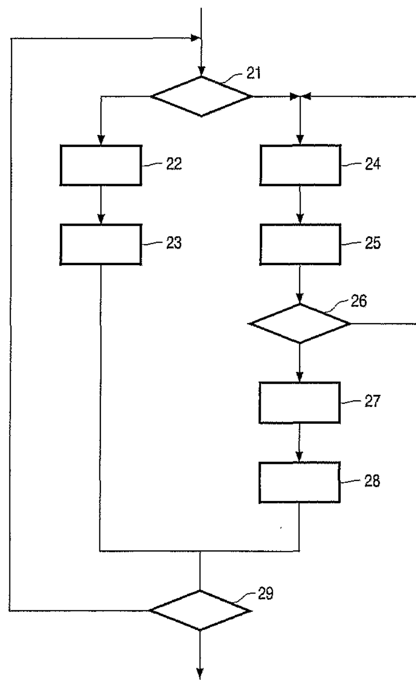



FIG. 2

【国際調査報告】

## INTERNATIONAL SEARCH REPORT

International Application No
PCT/EP 01/09694

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/318 G06F9/32 G06F9/455 G06F9/38

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. |
|---|---|---|
| Y | WO 99 18486 A (KONINKL PHILIPS ELECTRONICS NV ;PHILIPS SVENSKA AB (SE)) 15 April 1999 (1999-04-15) cited in the application | 1,5 |
| A | page 4, line 30 -page 6, line 24 page 10, line 7 -page 11, line 3 --- | 3,7 |
| Y | US 5 768 593 A (BROWN JORG ANTHONY ET AL) 16 June 1998 (1998-06-16) column 2, line 5 - line 23 column 3, line 35 -column 4, line 3 column 5, line 66 -column 7, line 62 --- | 1,5 |
| A | US 5 872 978 A (HOSKINS ASHER J) 16 February 1999 (1999-02-16) column 2, line 4 - line 65; claim 18 --- -/-- | 4,8 |

[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 |
|---|---|
| 9 January 2002 | 17/01/2002 |

| 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 | Daskalakis, T |

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

INTERNATIONAL SEARCH REPORT

International Application No
PCT/EP 01/09694

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 4 638 423 A (BALLARD DANNY B)<br>20 January 1987 (1987-01-20)<br>column 2, line 44 -column 3, line 46<br>column 4, line 24 - line 62<br>----- | 1,5 |

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

## INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/EP 01/09694

| Patent document cited in search report | | Publication date | Patent family member(s) | | Publication date |
|---|---|---|---|---|---|
| WO 9918486 | A | 15-04-1999 | 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 | 6292883 B1 | 18-09-2001 |
| | | | US | 6298434 B1 | 02-10-2001 |
| US 5768593 | A | 16-06-1998 | NONE | | |
| US 5872978 | A | 16-02-1999 | EP | 0811190 A2 | 10-12-1997 |
| | | | WO | 9723823 A2 | 03-07-1997 |
| | | | JP | 11502964 T | 09-03-1999 |
| US 4638423 | A | 20-01-1987 | NONE | | |

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

フロントページの続き

(74)代理人　100082991
弁理士　佐藤　泰和
(74)代理人　100096921
弁理士　吉元　弘
(74)代理人　100103263
弁理士　川崎　康
(72)発明者　オット、エル．スタインブッシュ
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　メノ、エム．リンドワー
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) 5B033 AA02 BA03 BE00 EA01 EA09
5B081 AA09 CC01 CC51 DD00