(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第4901075号
(P4901075)
(45)発行日　平成24年3月21日(2012.3.21)　(24)登録日　平成24年1月13日(2012.1.13)

(51)Int.Cl.　F I
G06F 11/28 (2006.01)　G06F 11/28 315A

請求項の数 29 (全 18 頁)

(21)出願番号　特願2004-125956 (P2004-125956)
(22)出願日　平成16年4月21日 (2004.4.21)
(65)公開番号　特開2004-326789 (P2004-326789A)
(43)公開日　平成16年11月18日 (2004.11.18)
審査請求日　平成19年3月20日 (2007.3.20)
(31)優先権主張番号　10/419,384
(32)優先日　平成15年4月21日 (2003.4.21)
(33)優先権主張国　米国 (US)

(73)特許権者　500046438
マイクロソフト　コーポレーション
アメリカ合衆国　ワシントン州　９８０５２－６３９９　レッドモンド　ワン　マイクロソフト　ウェイ
(74)代理人　100077481
弁理士　谷　義一
(74)代理人　100088915
弁理士　阿部　和夫
(72)発明者　ジョナサン　マイケル　ストール
アメリカ合衆国　９８０１１　ワシントン州　ボゼル　１２２　アベニュー　ノース　イースト　１６７２３

最終頁に続く

(54)【発明の名称】コンピュータ読取可能な媒体、方法及びコンピューティングデバイス

(57)【特許請求の範囲】
【請求項１】
デバッグに関するコンピュータプログラム命令を含むコンピュータ読取可能な媒体であって、前記コンピュータプログラム命令はプロセッサにより実行可能であり、
ユーザ入力によって生成されたコードを含むプログラムコードのリストを生成するステップであって、前記リストとアセンブリとを比較することにより、ユーザ入力によって生成されたコード部分とユーザ入力によって生成されていないコード部分とが区別される、ステップと、
ネイティブコードに、デバッグ処理関数への呼出し及びコンパイル時定数ポインタを含むプログラムコードであるデバッグプローブとグローバル変数であるフラグとを挿入するステップであって、前記デバッグプローブは、前記コンパイル時定数ポインタにより、コンパイル時における前記デバッグプローブと関連付けられたフラグを参照し、前記フラグが０でない値を示す場合にデバッグ処理関数を呼び出すことが記述され、前記デバッグプローブ及び前記フラグは、前記ネイティブコードを生成するためのコンピュータプログラムコンパイル動作中に挿入される、ステップと、
前記ネイティブコードをプロセスとして実行するステップであって、前記プロセスは、ステッピング動作を実行する１つまたは複数の実行のスレッドを含み、前記実行のスレッドが前記プロセスを介してステッピング動作を実行中に前記デバッグ処理関数を呼び出しているデバッグプローブに到達すると、当該到達したデバッグプローブに関連するコード部分の命令ポインタが前記リストに示されている場合に、前記実行のスレッドを停止する

、ステップと
を実行するためのコンピュータプログラム命令を含むことを特徴とするコンピュータ読取可能な媒体。
【請求項２】
前記ネイティブコードは、クラスメソッドおよび／または静的関数であることを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項３】
前記ステッピング動作は、デバッガによるステップインコマンド、ステップアウトコマンド、およびステップオーバコマンドの発行を含むことを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項４】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレッドを停止させるステップと、
を実行するためのコンピュータプログラム命令をさらに含むことを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項５】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレッドを停止するステップと、
前記デバッグプローブの各々に関連付けられたフラグの値を０にするステップと
を実行するためのコンピュータプログラム命令をさらに含むことを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項６】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記スレッドによりステップインが行なわれているコード部分への呼出しの直後のコードの行にブレークポイントを置くステップと
を実行するためのコンピュータプログラム命令をさらに含むことを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項７】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記到達したデバッグプローブに関連するコード部分への第１スタックフレームの戻りアドレスを識別するステップと、
前記戻りアドレスにブレークポイントを置くステップと
を実行するためのコンピュータプログラム命令をさらに含むことを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項８】
前記ネイティブコードの特定の１つとして前記ユーザ入力によって生成されたコードを識別するためのコンピュータプログラム命令をさらに含むことを特徴とする請求項１に記載のコンピュータ読取可能な媒体。
【請求項９】
前記ネイティブコードは、ユーザインターフェース、コマンドライン、および／または

構成ファイルデータ入力を介して識別されることを特徴とする請求項８に記載のコンピュータ読取可能な媒体。

【請求項１０】

デバッグの方法であって、

ユーザ入力によって生成されたコードを含むプログラムコードのリストを生成するステップであって、前記リストとアセンブリとを比較することにより、ユーザ入力によって生成されたコード部分とユーザ入力によって生成されていないコード部分とが区別される、ステップと、

ネイティブコードに、デバッグ処理関数への呼出し及びコンパイル時定数ポインタを含むプログラムコードであるデバッグプローブとグローバル変数であるフラグとを挿入するステップであって、前記デバッグプローブは、前記コンパイル時定数ポインタにより、コンパイル時における前記デバッグプローブと関連付けられたフラグを参照し、前記フラグが０でない値を示す場合にデバッグ処理関数を呼び出すことが記述され、前記デバッグプローブ及び前記フラグは、前記ネイティブコードを生成するためのコンピュータプログラムコンパイル動作中に挿入される、ステップと、

前記ネイティブコードをプロセスとして実行するステップであって、前記プロセスは、ステッピング動作を実行する１つまたは複数の実行のスレッドを含み、前記実行のスレッドが前記プロセスを介してステッピング動作を実行中に前記デバッグ処理関数を呼び出しているデバッグプローブに到達すると、当該到達したデバッグプローブに関連するコード部分の命令ポインタが前記リストに示されている場合に、前記実行のスレッドを停止する、ステップと

を備え、ステッピング動作は、ステップイン動作、ステップアウト動作、またはステップオーバ動作を含むことを特徴とする方法。

【請求項１１】

前記ネイティブコードは、それぞれクラスメソッドおよび／または静的関数であることを特徴とする請求項１０に記載の方法。

【請求項１２】

前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、

前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレッドを停止させるステップと、

を含むことを特徴とする請求項１０に記載の方法。

【請求項１３】

前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、

前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレッドを停止させるステップと、

前記デバッグプローブの各々に関連付けられたフラグの値を０にするステップと

をさらに備えたことを特徴とする請求項１０に記載の方法。

【請求項１４】

前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、

前記スレッドによりステップインが行なわれているコード部分への呼出しの直後のコードの行にブレークポイントを置くステップと

をさらに備えたことを特徴とする請求項１０に記載の方法。

【請求項１５】

前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、

前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記到達したデバッグプローブに関連するコード部分への第１スタックフレームの戻りアドレスを識別するステップと、
前記戻りアドレスにブレークポイントを置くステップと
をさらに備えたことを特徴とする請求項１０に記載の方法。
【請求項１６】
前記ユーザ入力によって生成されたコードを、前記ネイティブコードの所定の１つと識別するステップをさらに備えたことを特徴とする請求項１１に記載の方法。
【請求項１７】
デバッグ用のコンピューティングデバイスであって
プロセッサと、
前記プロセッサに結合されたメモリであって、
ユーザ入力によって生成されたコードを含むプログラムコードのリストを生成するステップであって、前記リストとアセンブリとを比較することにより、ユーザ入力によって生成されたコード部分とユーザ入力によって生成されていないコード部分とが区別される、ステップと、
ネイティブコードに、デバッグ処理関数への呼出し及びコンパイル時定数ポインタを含むプログラムコードであるデバッグプローブとグローバル変数であるフラグとを挿入するステップであって、前記デバッグプローブは、前記コンパイル時定数ポインタにより、コンパイル時における前記デバッグプローブと関連付けられたフラグを参照し、前記フラグが０でない値を示す場合にデバッグ処理関数を呼び出すことが記述され、前記デバッグプローブ及び前記フラグは、前記ネイティブコードを生成するためのコンピュータプログラムコンパイル動作中に挿入される、ステップと、
前記ネイティブコードをプロセスとして実行するステップであって、前記プロセスは、ステッピング動作を実行する１つまたは複数の実行のスレッドを含み、前記実行のスレッドが前記プロセスを介してステッピング動作を実行中に前記デバッグ処理関数を呼び出しているデバッグプローブに到達すると、当該到達したデバッグプローブに関連するコード部分の命令ポインタが前記リストに示されている場合に、前記実行のスレッドを停止する、ステップと
を実行するための命令を含む前記プロセッサによって実行可能なコンピュータプログラム命令を含むメモリと
を備えたことを特徴とするコンピューティングデバイス。
【請求項１８】
ステッピング動作は、デバッガによるステップインコマンド、ステップアウトコマンド、およびステップオーバコマンドの発行を含むことを特徴とする請求項１７に記載のコンピューティングデバイス。
【請求項１９】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレッドを停止させるステップと
を実行するための命令をさらに含むことを特徴とする請求項１７に記載のコンピューティングデバイス。
【請求項２０】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレ

ッドを停止させるステップと、
前記デバッグプローブの各々に関連付けられたフラグの値を０にするステップと
を実行するための命令をさらに含むことを特徴とする請求項１７に記載のコンピューティングデバイス。
【請求項２１】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記スレッドによりステップインが行なわれているコード部分への呼出しの直後のコードの行にブレークポイントを置くステップと
を実行するための命令をさらに含むことを特徴とする請求項１７に記載のコンピューティングデバイス。
【請求項２２】
前記デバッグ処理関数を呼び出しているデバッグプローブに到達することに応答して、前記デバッグ処理関数を呼び出しているデバッグプローブに対応する前記コンパイル時定数ポインタの参照先の値を取得するステップと、
前記到達したデバッグプローブに関連するコード部分への第１スタックフレームの戻りアドレスを識別ステップと、
前記戻りアドレスにブレークポイントを置くステップと
を実行するための命令をさらに含むことを特徴とする請求項１７に記載のコンピューティングデバイス。
【請求項２３】
前記ユーザ入力によって生成されたコードを、前記プログラミング構造の所定の１つと識別するステップを実行するための命令をさらに含むことを特徴とする請求項１７に記載のコンピューティングデバイス。
【請求項２４】
デバッグ用のコンピューティングデバイスであって、
ユーザ入力によって生成されたコードを含むプログラムコードのリストを生成する手段であって、前記リストとアセンブリとを比較することにより、ユーザ入力によって生成されたコード部分とユーザ入力によって生成されていないコード部分とが区別される、手段と、
ネイティブコードに、デバッグ処理関数への呼出し及びコンパイル時定数ポインタを含むプログラムコードであるデバッグプローブとグローバル変数であるフラグとを挿入する手段であって、前記デバッグプローブは、前記コンパイル時定数ポインタにより、コンパイル時における前記デバッグプローブと関連付けられたフラグを参照し、前記フラグが０でない値を示す場合にデバッグ処理関数を呼び出すことが記述され、前記デバッグプローブ及び前記フラグは、前記ネイティブコードを生成するためのコンピュータプログラムコンパイル動作中に挿入され、前記ネイティブコードは、クラスメソッドおよび／または静的関数を含む手段と、
前記ネイティブコードをプロセスとして実行する手段であって、前記プロセスは、ステッピング動作を実行する１つまたは複数の実行のスレッドを含み、前記実行のスレッドが前記プロセスを介してステッピング動作を実行中に前記デバッグ処理関数を呼び出しているデバッグプローブに到達すると、当該到達したデバッグプローブに関連するコード部分の命令ポインタが前記リストに示されている場合に、前記実行のスレッドを停止する手段と
を備えたことを特徴とするコンピューティングデバイス。
【請求項２５】
ステッピング動作は、ステップイン動作、ステップアウト動作、またはステップオーバ動作を含むことを特徴とする請求項２４に記載のコンピューティングデバイス。
【請求項２６】

前記デバッグ処理関数を呼び出しているデバッグプローブの後のコードの行で前記スレッドを停止させる手段と
をさらに備えたことを特徴とする請求項２４に記載のコンピューティングデバイス。
【請求項２７】
前記スレッドによりステップインが行なわれているコード部分への呼出しの直後のコードの行で前記スレッドを停止させる手段と
をさらに備えたことを特徴とする請求項２４に記載のコンピューティングデバイス。
【請求項２８】
前記到達したデバッグプローブに関連するコード部分への第１スタックフレームの戻りアドレスを識別する手段と、
前記戻りアドレスで前記スレッドを停止させる手段と
をさらに備えたことを特徴とする請求項２４に記載のコンピューティングデバイス。
【請求項２９】
前記ユーザ入力によって生成されたコードを、前記ネイティブコードの所定の１つと識別する手段をさらに備えたことを特徴とする請求項２４に記載のコンピューティングデバイス。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、ソフトウェア開発に関する。
【背景技術】
【０００２】
デバッグには、通常、デバッガを使用して行われるが、デバッガとは、ソフトウェア開発者がコンピュータプログラムのランタイムでの挙動を観察し、セマンティックエラーを突き止めることを可能にするツールである。ストップコマンドなどのデバッグコマンドによって、プログラマは、いつでも動作中のプログラムの実行を止めることができる。一方、ブレークポイントを手動で挿入することによって、プログラマは、コードの予め定められた点に達した時にプロセスを停止させることができる。デバッグ対象プログラムは、命令ストリーム内のブレークオペコードに到達する（ｈｉｔ）まで自由に動作し、到達した点でオペレーティングシステムはデバッグ対象プログラムを停止し、その後デバッガはデバッグ対象プログラムを続行させる。したがって、コンピュータプログラムをデバッグする時には、そのプログラムは、動作中（すなわち、プロセスとして実行中）または停止のいずれかである。ステップイントゥ、ステップオーバ、およびステップアウトコマンドなどのある種のデバッグコマンドは、ブレークモード（すなわち、デバッグ対象プログラムが停止している時）に限って機能し、プログラマが、そのプログラムの状態のステップを抜けて（ｓｔｅｐ　ｔｈｒｏｕｇｈ）、変数、ポインタ、および／またはその他の内容を監視し、および／または修正できるようにする。
【０００３】
従来のステップは、命令ストリームの静的解析を通して戦略ポイントにブレークオペコードを置き、その後、それぞれのパッチに遭遇する（到達する）まで自由に動作させることにより行われる。たとえば、ステップインでは、ステップインされる関数の先頭にパッチが置かれ、ステップオーバでは、ステップオーバが行われる行の後にパッチが置かれ、ステップアウトでは、現在の関数がリターンした後に実行される命令にパッチが置かれる。しかしながら、そのような従来のステップコマンドでは、プログラマが「関心のないコード」を自動的にスキップすることができない。むしろ、上述のように、プログラマは関心のあるコードに手動でブレークポイントを挿入し、ブレークポイントに到達するまで実行し、および／または関心のないコードを手動でステップを抜けて関心のあるコードに達する必要がある。言い換えると、プログラマは、繰り返し（行ごとに）関心のないコードのステップイントゥを行い、ステップを抜け、関数およびおそらくは関心のないコードの内部のステップオーバを行い、および／または関数およびおそらくは関心のないコードの

内部からステップアウトを行う必要が生じる場合がある。

【０００４】

たとえば、ステップイントゥコマンドおよびステップオーバコマンドは、関数呼出しを扱う形だけが異なる。どちらのコマンドでも、デバッガはソースコードの次の行を実行するように指示される。その行に関数呼出しが含まれる場合に、ステップイントゥコマンドでは呼出し自体だけが実行され、その関数の内部のコードの最初の行が関心のないコードであるかどうかに関係なく、そのコードの最初の行で停止する。ステップオーバコマンドでは関数全体が実行され、その関数の外にある最初の行が関心のないコードであるかどうかとは無関係に、その最初の行で停止する。入れ子になった関数呼出しでは、ステップイントゥは、最も深く入れ子になった関数にステップインする。たとえば、ステップイントゥが、Ｆ１（Ｆ２（））のような呼出しに対して使用される場合に、デバッガは、関数Ｆ２にステップインする。ステップアウトコマンドは、関数がリターンするまでプログラム実行を再開することによって関数のステップアウトを行うために用いられ、および関数呼出しのリターンポイントが関心のないコードに対応するかどうかとは無関係に、そのリターンポイントでブレークを行う。

【０００５】

さらに、デバッガに、コードの各々の行のそれぞれについてステップインコマンドを反復して実行するように指示して、コードの一部を通ってシングルステップで実行することができるが、このプロセスはデバッグプロセスの性能がかなり低下し、プロセスデッドロック（すなわち、デバッガに起因する同一キャッシュラインに関する競合）の危険性がかなり高まる。たとえば、複数のステップイン動作の反復をエミュレートするために、デバッガは、コードの行ごとに次の命令にブレークポイントを挿入し、プロセスを動作させ、関連する例外をキャッチし、および命令ポインタを検査して、事前に構成されたステップポイントに達したかどうかを判定する（または、デバッガが手動で停止されるまでこれらを行う）。そのような反復動作では、デバッガは、別の例外（ｅｘｃｅｐｔｉｏｎ）が出力される前にコードの１行だけしか実行せず（すなわち、コードが非常に短い時間の間だけ「自由に動作」することだけを許可される）、デバッグ性能がかなり妨げられ、プロセスデッドロックの可能性が高まることが理解される。説明の目的で、自由に動作するコードとは、遭遇するブレークポイントによって妨げられずに実行されることを許可されるメソッド、または関心のないコードなどのコードの論理ブロックのことである。

【発明の開示】

【発明が解決しようとする課題】

【０００６】

そのようなデバッグの制限は、デバッグされるコンピュータプログラムが、大量のコードを統合した洗練された環境で動作するように設計され、このような統合されたコードにプログラマがデバッグの際関心を持たないとき特に問題となる。そのような関心のないコードに、たとえばプログラマが記述したのではないコード、既にデバッグされたコード、他の共有コード、ネットワーク化されたサービスのコード、相互運用性フレームワークコード、および／またはその他のものを含めることができる。そのようなシナリオでは、プログラマが関心のないコードを簡単にスキップできないようなプログラムのデバッグの既存の技法により、時間のかかる労働集中型の作業が必要となり、これによって、初心者と上級者の双方のプログラマの作業が困難となる可能性がある。

【０００７】

したがって、関心のあるコード内に手動でブレークポイントを設定し、および／または関心のあるコードに達するために手作業で関心のないコードのステップスルーを行う必要がない、ユーザが関心のあるコードだけをデバッグできるようにするシステムおよび方法が非常に望ましい。

【課題を解決するための手段】

【０００８】

本明細書において、ジャストマイコード（以下ＪＭＣという）デバッグのシステムおよ

び方法を説明する。本発明の一態様では、デバッグプローブが、関心のあるコードを構成するそれぞれのプログラミング構造に自動的に挿入される。デバッグプローブは、ネイティブコードを生成するコンピュータプログラムコンパイル動作中に挿入される。次に、ネイティブコードをプロセスとして実行する。このプロセスには、ＪＭＣステッピング動作中に関心のないコードを介して自由に動作する１つまたは複数の実行のスレッドが含まれる。１つまたは複数のスレッドのうちの１つのスレッドが、プロセスを介してＪＭＣのステップ実行中にデバッグプローブのアクティブな１つのデバッグプローブに到達すると、その１つのスレッドだけが関心のあるコード内で停止する。

【０００９】

以下に詳細な説明を、添付図面を参照して説明する。図面では、構成要素の符号の上１桁が、その構成要素が最初に現れる特定の図面を示すようになっている。

【発明を実施するための最良の形態】

【００１０】

（概要）

上述した通常のデバッグ技法の制限に対処するため、デバッグ動作中に関心のないコードを自動的にスキップするジャストマイコード（ＪＭＣ）デバッグのシステムおよび方法を説明する。説明において、関心のないコードとは、どのような理由であれ、その時にプログラマのデバッグの努力を生じさせないと思われるコード（たとえば、プログラマが記述したものでないコード、既にデバッグされたコードなど）のことである。「関心のあるコード」とは、デバッグ動作中にプログラマにとって関心のあるものなので、関心のないコードから類推（ｃｏｒｏｌｌａｒｙ）される。

【００１１】

具体的に言うと、本発明のさまざまな実施形態を用いて、デバッグアプリケーション（デバッガ）が、関心があるものとしてアセンブリ内の任意のメソッドにマークを付すようにすることができる。アセンブリからネイティブコードを生成するジャストインタイム（ＪＩＴ）コンパイラ動作中に、デバッグプローブが、関心のあるメソッドに自動的に挿入される。デバッガを使用して、ユーザはデバッグのためプロセス中にネイティブコードをロードする。ＪＭＣステッピングの動作に応答して、関心のあるメソッドにＪＭＣステッピングスレッドが到達する時に限って、プロセスの実行が自動的に止められる。すべての非ＪＭＣステッピングスレッドは、関心のあるコードおよび関心のないコードを通って自由に動作する。これは、ユーザが、関心のあるコード内にブレークポイントを手動でセットする必要がなく、プログラマが、関心のあるコードに達するために関心のないコードを徒渉（ｗａｄｅ－ｔｈｒｏｕｇｈ）する（すなわち、普通のステップ動作を実行する）必要がないことを意味する。

【００１２】

（例示的な動作環境）

図面を参照すると、等しい符号は等しい要素を指しており、本発明は適切なコンピューティング環境で実施されるものとして図示されている。必ずしも必要ではないが、パーソナルコンピュータによって実行されるプログラムモジュールなどのコンピュータ実行可能命令の全般的な文脈で本発明を説明する。プログラムモジュールには、全般的に特定のタスクを実行するか、または特定の抽象データ型を実施するルーチン、プログラム、オブジェクト、コンポーネント、データ構造などが含まれる。

【００１３】

図１は、ジャストマイコード（ＪＭＣ）型のデバッグに関する後述のシステム、装置、および方法を（完全または部分的のいずれかで）実施できる適切なコンピューティング環境１００の例を示す図である。例示的なコンピューティング環境１００は、適切なコンピューティング環境の１例にすぎず、本明細書に記載のシステムおよび方法の使用または機能性に関する制限を暗示することを意図したものではない。また、コンピューティング環境１００を、コンピューティング環境１００に示された構成要素の任意の１つまたはその組合せに関し依存または要件を必要とするものと解釈してはならない。

【００１４】
本明細書に記載の方法およびシステムは、多数の他の汎用または特殊目的のコンピューティングシステム環境およびコンピューティングシステム構成と共に動作する。使用に適すると考えられる周知のコンピューティングシステム、コンピューティング環境、および／またはコンピューティング構成の例には、パーソナルコンピュータ、サーバコンピュータ、マルチプロセッサシステム、マイクロプロセッサベースのシステム、ネットワークＰＣ、ミニコンピュータ、メインフレームコンピュータ、上記のシステムまたはデバイスのいずれかを含む分散コンピューティング環境などが含まれるが、これに限定されない。このフレームワークのコンパクト版またはサブセット版を、ハンドヘルドコンピュータなどの限られたリソースを有するクライアントまたは他のコンピューティングデバイスで実施することもできる。本発明は、通信ネットワークを介してリンクされたリモート処理デバイスによってタスクが実行される分散コンピューティング環境で実施することもできる。分散コンピューティング環境では、プログラムモジュールを、ローカルとリモートの両方のメモリストレージデバイスに配置することができる。
【００１５】
図１からわかるように、コンピューティング環境１００には、コンピュータ１０２の形の汎用コンピューティングデバイスが含まれる。コンピュータ１０２のコンポーネントには、１つまたは複数のプロセッサまたは処理ユニット１０４、システムメモリ１０６、およびシステムメモリ１０６を含むさまざまなシステムコンポーネントをプロセッサ１０４に接続するバス１０８を含めることができるが、これに限定されない。システムバス１０８は、メモリバスまたはメモリコントローラ、周辺バス、ａｃｃｅｌｅｒａｔｅｄ ｇｒａｐｈｉｃｓ ｐｏｒｔ、およびさまざまなバスアーキテクチャのいずれかを使用するプロセッサバスまたはローカルバスを含む、複数のタイプのバスアーキテクチャのいずれかの１つまたは複数を表す。制限ではなく例として、そのようなアーキテクチャに、Ｉｎｄｕｓｔｒｙ Ｓｔａｎｄａｒｄ Ａｒｃｈｉｔｅｃｔｕｒｅ（ＩＳＡ）バス、マイクロチャネルアーキテクチャ（ＭＣＡ）バス、Ｅｎｈａｎｃｅｄ ＩＳＡ（ＥＩＳＡ）バス、Ｖｉｄｅｏ Ｅｌｅｃｔｒｏｎｉｃｓ Ｓｔａｎｄａｒｄｓ Ａｓｓｏｃｉａｔｉｏｎ（ＶＥＳＡ）ローカルバス、およびメザニンバスとも称するＰｅｒｉｐｈｅｒａｌ Ｃｏｍｐｏｎｅｎｔ Ｉｎｔｅｒｃｏｎｎｅｃｔｓ（ＰＣＩ）バスが含まれる。
【００１６】
コンピュータ１０２に、通常は、さまざまなコンピュータ読取可能な媒体が含まれる。そのような媒体は、コンピュータ１０２によってアクセス可能なすべての入手可能な媒体とすることができ、これには、揮発性および不揮発性、取外し可能および固定の両方の媒体が含まれる。図１では、システムメモリ１０６に、ランダムアクセスメモリ（ＲＡＭ）１１０などの揮発性メモリおよび／または読取専用メモリ（ＲＯＭ）１１２などの不揮発性メモリの形のコンピュータ読取可能な媒体が含まれる。起動中などにコンピュータ１０２内の要素の間での情報転送を支援する基本ルーチンを含む基本入出力システム（ＢＩＯＳ）１１４が、ＲＯＭ１１２に保管される。ＲＡＭ１１０には、通常は、プロセッサ１０４によって即座にアクセス可能および／または現在操作されているデータおよび／またはプログラムモジュールが含まれる。
【００１７】
コンピュータ１０２に、さらに、他の取外し可能／固定、揮発性／不揮発性のコンピュータ記憶媒体を含めることができる。たとえば、図１に、固定不揮発性磁気媒体（図示せず、通常は「ハードドライブ」と称する）から読み取り、これに書き込むハードディスクドライブ１１６、取外し可能不揮発性磁気ディスク１２０（たとえば、「フロッピディスク」）から読み取り、これに書き込む磁気ディスクドライブ１１８、ＣＤ－ＲＯＭ／Ｒ／ＲＷ、ＤＶＤ－ＲＯＭ／Ｒ／ＲＷ／＋Ｒ／ＲＡＭ、または他の光媒体などの取外し可能不揮発性光ディスク１２４から読み取り、これに書き込む光ディスクドライブ１２２が示されている。ハードディスクドライブ１１６、磁気ディスクドライブ１１８、および光ディスクドライブ１２２のそれぞれは、１つまたは複数のインターフェース１２６によってシ

ステムバス１０８に接続される。

【００１８】

ドライブおよび関連するコンピュータ読取可能な媒体によって、コンピュータ１０２のコンピュータ読取可能な命令、データ構造、プログラムモジュール、および他のデータの不揮発性ストレージが提供される。本明細書で説明する例示の実施形態では、ハードディスク、取外し可能磁気ディスク１２０、および取外し可能光ディスク１２４が使用されるが、磁気カセット、フラッシュメモリカード、ディジタルビデオディスク、ランダムアクセスメモリ（ＲＡＭ）、読取専用メモリ（ＲＯＭ）、およびその他のものなど、コンピュータによってアクセス可能なデータを保管できる他のタイプのコンピュータ読取可能な媒体も、例示の動作環境で使用できることを、当業者は理解するに違いない。

【００１９】

ユーザは、キーボード１４０およびポインティングデバイス１４２（「マウス」など）などの入力デバイスを介してコンピュータ１０２にコマンドおよび情報を供給することができる。他の入力デバイス（図示せず）に、マイクロホン、ジョイスティック、ゲームパッド、衛星パラボラアンテナ、シリアルポート、スキャナ、カメラなどを含めることができる。上記および他の入力デバイスは、バス１０８に結合されたユーザ入力インターフェース１４４を介してプロセッサ１０４に接続されるが、パラレルポート、ゲームポート、またはｕｎｉｖｅｒｓａｌ　ｓｅｒｉａｌ　ｂｕｓ（ＵＳＢ）など、他のインターフェースおよびバス構造によって接続することができる。

【００２０】

モニタ１４６または他のタイプのディスプレイデバイスも、ビデオアダプタ１４８などのインターフェースを介してバス１０８に接続される。モニタ１４６の他に、パーソナルコンピュータには、通常は、スピーカおよびプリンタなど、出力周辺インターフェース１５０を介して接続することができる、他の周辺出力デバイス（図示せず）が含まれる。

【００２１】

コンピュータ１０２は、リモートコンピュータ１５２などの１つまたは複数のリモートコンピュータへの論理接続を使用することによって、ネットワーク化された環境で動作することができる。リモートコンピュータ１５２に、コンピュータ１０２に関して本明細書で説明した要素および特徴の多数またはすべてを含めることができる。図１に示された論理接続は、ローカルエリアネットワーク（ＬＡＮ）１５４および全般的な広域ネットワーク（ＷＡＮ）１５６である。そのようなネットワーキング環境は、オフィス、会社全体のコンピュータネットワーク、イントラネット、およびインターネットでありふれたものである。

【００２２】

ＬＡＮネットワーキング環境で使用される時に、コンピュータ１０２は、ネットワークインターフェースまたはネットワークアダプタ１５８を介してＬＡＮ１５４に接続される。ＷＡＮネットワーキング環境で使用される時に、コンピュータに、通常はＷＡＮ１５６を介する通信を確立する、モデム１６０または他の手段が含まれる。モデム１６０は、内蔵または外付けとすることができるが、ユーザ入力インターフェース１４４または他の適当な機構を介してシステムバス１０８に接続することができる。図１には、インターネットを介するＷＡＮの特定の実施形態が図示されている。この図では、コンピュータ１０２によって、モデム１６０を使用して、インターネット１６２を介して少なくとも１つのリモートコンピュータ１５２との通信が確立される。

【００２３】

ネットワーク化された環境では、コンピュータ１０２に関して図示されたプログラムモジュールまたはその一部を、リモートメモリストレージデバイスに保管することができる。したがって、たとえば図１に示されているように、リモートアプリケーションプログラム１６４が、リモートコンピュータ１５２のメモリデバイスに常駐することができる。図示され説明されたネットワーク接続が、例示的なものであり、コンピュータの間の通信リンクを確立する他の手段を使用できることを理解されたい。

【００２４】
ランタイム環境を提供するオペレーティングシステム（ＯＳ）１２８、ジャストマイコード（ＪＭＣ）デバッグ用のアプリケーションプログラム１３０、他のプログラムモジュール１３２（たとえばデバイスドライバなど）、およびソースコード、中間アセンブリ、および／またはその他のものなどのプログラムデータ１３４を含む複数のプログラムモジュールを、ハードディスク、磁気ディスク１２０、光ディスク１２４、ＲＯＭ１１２、またはＲＡＭ１１０に保管することができる。
【００２５】
図２は、ジャストマイコード（ＪＭＣ）型のデバッグに関するアプリケーションプログラム１３０およびプログラムデータ１３４を含む、図１のシステムメモリ１０６のさらなる例示の態様を示すブロック図である。アプリケーションプログラム１３０およびプログラムデータ１３４には、たとえばプロセス２０２、プライマリコンパイラ２０４、プロセス２０２をデバッグするデバッグプログラム（「デバッガ」）２０６、および共有ランタイムコンポーネント（「ランタイム」）２０８が含まれる。プロセス２０２は、ソースコード（図示せず）をアセンブリ（図示せず）にコンパイルした結果であり、アセンブリは、ネイティブコード２１０にコンパイルされており、プロセス２０２は、デバッガ２０６によりネイティブコード２１０が実行されるのを表している。
【００２６】
ソースコードは、あらゆるタイプのコンピュータプログラミング言語で記述された、すべてのタイプのコンピュータプログラムコードを意味する。デバッグスイッチをオンにされた（たとえば「／ｄｅｂｕｇ」）、ソースコードをアセンブリ（図示せず）にコンパイルするためのプライマリコンパイラ２０４。そのようなプライマリコンパイラは、Ｃ、Ｃ＋＋、Ｃ＃、Ｖｉｓｕａｌ　Ｂａｓｉｃなどの任意のコンピュータプログラム言語コンパイラおよび／またはＪＭＣデバッグを実施するように修正された他のタイプのコンパイラとすることができる。アセンブリは、既知であり、実行のためのソースコードからプラットフォーム固有のネイティブコードへの変換の中間ステージを表す。このために、そのようなアセンブリには、たとえばプラットフォーム／プロセッサ依存中間言語（ＩＬ）および対応するメタデータが含まれる。
【００２７】
ユーザは、デバッガ２０６にアセンブリをロードする。アセンブリをデバッグアプリケーションにロードする技法は、既知のものである。たとえば、アセンブリ名を指定する（たとえば、コマンドラインまたはＵＩを介して）または他の形でアセンブリを選択することによって、アセンブリをデバッガ２０６にロードすることができる。アセンブリローディング動作中に、デバッガ２０６によって、関心のあるコードのリスト２１２が生成されて、アセンブリのうちで、関心のないコードと比較した関心のあるコードである部分が区別される。ユーザコード（繰り返すが、ソースコードをアセンブリに変換したものであり、最終的には、後述するようにプロセスとして実行されるネイティブコードに変換される）には、通常、関心のあるコードと関心のないコードとの所定の組合せが含まれる。しかし、ユーザコードには、ランタイム２０８に属するホスティングされるコードは含まれない。一実施形態では、関心のあるコード２１２は、いずれかの形態のユーザ入力インターフェース１４４（図１）へのユーザ入力によって生成される。たとえば、ユーザは、デバッグサービスによってダイアログボックスに表示される複数のメソッドの特定の１つを選択することができ（たとえば、ポインティングデバイス、キーボード、音声アクティベーテッドテクニックなどを介して）、あるいは、ユーザが、コマンドラインインターフェースおよび／またはその他のものにプログラミング構造の名前を入力することができる。デバッガ２０６によって、公開されたデバッグＡＰＩ２１６（これによって２つの部分を実施することもでき、その部分の一方は、ランタイム２０８のサービスとして実施され、もう１つは２１６を実施するパブリックコンポーネントとして実施され、デバッガ２０６によって消費される）によって、識別された関心のあるコード２１２に関してデバッグサービス２１４に通知される。

【００２８】
この点で、ユーザが、たとえばＪＭＣメニュー項目、コマンドライン、または他のＪＭＣイネーブリングインストラクションを使用可能にし、実行コマンドまたは開始コマンドを発行することによって、ＪＭＣデバッグ動作の開始をデバッガ２０６に指示する。これによって、デバッガ２０６が、ＪＩＴコンパイラ２２０に、プロセス２０２としての後の実行のためにアセンブリをネイティブコード２１０に変換するように指示する。
【００２９】
アセンブリ変換動作中に、ＪＩＴコンパイラ２２０は、ＪＭＣイネーブリングコンポーネント、フラグ２２２、およびデバッグプローブ２２４を、ネイティブコード２１０のプログラミング構造（たとえば、実行可能ファイル／バイナリ、ダイナミックリンクライブラリ、クラスオブジェクトおよびメソッド、静的関数、および／またはその他のもの）に挿入する。この実施形態では、ＪＩＴイネーブリングコンポーネントが、クラスメソッドおよび／または静的関数のプロローグの直後に挿入される。ＪＩＴコンパイラ２２０は、上で説明したようにデバッガ２０６によって生成された関心のあるコードリスト／ＩＤ２１２を解析することによって、そのようなクラスメソッドおよび／または静的関数を識別する。
【００３０】
デバッグプローブ２２４は、コンパイル時定数ポインタ値によって、関連するフラグ２２２を参照して、アクティブ化されたプローブ２２４に到達した特定の実行のスレッドが、ＪＭＣステッピング動作を実行しているかどうかを評価し、そうである場合にプロセス２０２を停止するようにデバッグサービス２１４に要求するかどうかを判定する。今説明したように、そのような判定は、アクティブ化されていないフラグについては行われず、アクティブ化されたフラグだけについて行われる。例示的なデバッグプローブ２２４は、次のように表される。

ｉｆ（＊ｐＦｌａｇ）｛ｃａｌｌ　ＪＭＣ＿Ｐｒｏｂｅ｝

ここで、ｐＦｌａｇは、対応するフラグ２２２へのコンパイル時定数ポインタである（「＊」は、アドレスが、関連するフラグ２２２の値を得るために実行時に参照先の値を取得することを示す）。プローブ２２４の「ＪＭＣ＿Ｐｒｏｂｅ」部分は、ＪＭＣ＿Ｐｒｏｂｅ関数２２６への呼出しである。ポインタ間接指定の使用によって、ｐＦｌａｇｓがそれに関して挿入された複数のプログラミング構造からのｐＦｌａｇｓはすべて、同一のフラグ２２２を指示することができる。
【００３１】
図からわかるように、ＪＭＣ＿Ｐｒｏｂｅ２２６への呼出しは条件により可能であり、参照されるフラグ２２２がアクティブ化されているか非アクティブ化されているかに依存する。本説明において、アクティブ化されたフラグ２２２は非０値を有し、非アクティブ化されたフラグ２２２は０またはヌルの値を有する。フラグ２２２がアクティブ化されている時、デバッグプローブ２２４はアクティブである（たとえば、アクティブなプローブ）といい、逆も同様である。たとえば、実行のスレッドが、アクティブなプローブ２２４に到達する時に、ＪＭＣ＿Ｐｒｏｂｅ２２６への呼出しが行われる。同様に、非アクティブなプローブ２２４に到達しても、そのような呼出しは行われない。これは、単一のフラグ２２２のトグルによって、各関連するプローブ２２４がそれぞれアクティブ化または非アクティブ化されることを意味する。次に、この新規のＪＭＣデバッグプローブ２２４が、デバッグ動作中にどのように使用されるかを説明する。
【００３２】
プロセス２０２を（たとえば普通のブレークポイントを使用するなど、伝統的な停止機構によって）停止させている時に、デバッガ２０６を用いると、ユーザはＪＭＣステッピングコマンド２２８（すなわち、ＪＭＣステップイントゥコマンド、ＪＭＣステップアウトコマンド、およびＪＭＣステップオーバコマンド）によってプロセス２０２にＪＭＣステップスルーを行うことができるようになる。そのようなコマンドは、たとえばコマンドラインおよび／またはユーザ入力インターフェース１４４（図１）のＵＩ部分を介して、

ユーザによって選択される。デバッガ２０６によってデバッグサービス２１４に送られるＪＭＣステッピングコマンド２２８（すなわち、ＪＭＣステップインコマンド、ＪＭＣステップアウトコマンド、およびＪＭＣステップオーバコマンド２２８）に応答して、デバッグサービス群（ｍａｓｓ）は、フラグ２２２を使用可能にし、これによって関連するプローブ２２４をアクティブ化する。ユーザは、どのプローブがアクティブであるかを知る必要がない。

【００３３】

アクティブ化されたデバッグプローブ２２４によって、プロセス２０２の実行スレッドは、関心のないコードを通り抜けて自由に動作できるようになる（すなわち、実質的にフルスピードで不要な一時停止がない）。アクティブ化されたプローブ２２４は関心のあるコード内に配置されるが、それが到達するすべてのスレッドは、ＪＭＣ＿Ｐｒｏｂｅ２２６により遭遇したプローブ２２４が、そのスレッドがコードにＪＭＣステップスルーを行っていると判定する時に限って停止する（ＴｒｉｇｇｅｒＭｅｔｈｏｄＥｎｔｒｙ２３０を介して）。このため、ＪＭＣ＿Ｐｒｏｂｅ２２６によってコールスタック（他のデータ２３２参照）をデコードして、到達した（「トリガされた」）デバッグプローブ２２４に関連するメソッドの命令ポインタ（ｉｐ）およびフレームポインタ（ｆｐ）を識別することによって、ＪＭＣステッピングを行っていないスレッドがフィルタにかけられる。現在のスレッドが、ＪＭＣステッピングを行っている場合に、ＪＭＣ＿Ｐｒｏｂｅ２２６によって、ｉｐパラメータおよびｆｐパラメータが、ＴｒｉｇｇｅｒＭｅｔｈｏｄＥｎｔｒｙ２３０に渡され、ＴｒｉｇｇｅｒＭｅｔｈｏｄＥｎｔｒｙ２３０は、ｉｐをメソッドにマッピングして、関心のあるコードリスト／ＩＤ２１２に示されるように、メソッドが「関心のある」ものであるかどうかを判定する（デバッグプローブ２２４が関心のあるコード内に配置されるかどうかをそのプローブがルックアップ動作を介して判定できることを、ＪＩＴコンパイラ２２０が実質的に保証するのに十分な記録（ｂｏｏｋｋｅｅｐｉｎｇ）が、ランタイム２０８によって行われる）。このメソッドが、関心のあるメソッドである場合、ＪＭＣ＿Ｐｒｏｂｅ２２６によって、デバッグプローブ２２４の後にブレークオペコード（すなわち、ブレークポイント２３４。すべてのブレークポイント２３４がＪＭＣ＿Ｐｒｏｂｅによって注入される）が挿入され、プロセスの実行が継続され、実行中のスレッドが、ブレークポイント２３４に到達して関心のあるコード内で停止する。

【００３４】

さらに、ＪＭＣステッピングコマンド２２８が、

ＪＭＣステップインである場合、デバッグサービス２１４によって、すべてのＪＭＣ＿ｐｒｏｂｅ２２４がアクティブ化され、ステップインを行っているメソッドへの呼出しの後にブレークポイントが置かれる（ＪＭＣ＿Ｐｒｏｂｅによって挿入されるブレークポイントではない）。ＪＭＣプローブ２２４は、すべての関心のあるコードの先頭にあり、プローブがアクティブ化されているので、呼出しによって最終的に「関心のある」コードが（おそらくは、最終的に「関心のある」コードを呼び出す関心のない／フレームワークコードの呼出しによって間接的に）呼び出される場合、スレッドは、アクティブ化されたプローブに到達し、このプローブによって、ＴｒｉｇｇｅｒＭｅｔｈｏｄＥｎｔｒｙ２３０が呼び出され、スレッドが停止されてステップは関心のあるコード内で完了する。呼出しによって、関心のあるコードが呼び出されない場合には、スレッドは、ステップイントゥが行われるメソッドの呼出しの後に挿入されたブレークポイント（すなわち、ＪＭＣ＿Ｐｒｏｂｅによって挿入されたブレークポイント２３４でないブレークポイント）に到達し、ＪＭＣステップイン動作は完了する。このようにして、プロセス２０２は、ＪＭＣステップインが開始される時とそれが終了する時の間にフルスピードで（すなわち、中間ブレークポイントオペコードによって割り込まれることなく）動作する。

【００３５】

ＪＭＣステップアウトである場合、ＪＭＣ＿Ｐｒｏｂｅ２２６は、最初のスタックフレームによって識別される関心のあるメソッドへの戻りアドレスに、ブレークポイント２３４を自動的に挿入する。したがって、このブレークポイントは、デバッグプローブ２２

４に到達したメソッドと同一のメソッドには挿入されない。関心のあるメソッドへの戻りアドレスを有する最初のスタックフレームを突き止めるため、デバッグサービス２１４によって、プロセススタックのフルウォークアップ（すなわち、スタックの開放（ｕｎｗｉｎｄ））が実行される。デバッグサービス２１４は、「他のデータ」２３２の「関心のあるメソッドデータ」部分に鑑みて１つまたは複数のランタイム２０８サービスを使用し、どのスタックフレームアドレスが関心のあるメソッドの戻りアドレスであるかを判定する。そのような関心のあるメソッドデータには、たとえば、関心のあるメソッドの各々について実質的に一意のＩＤ（たとえば、ＧＵＩＤ）がそれぞれ含まれる。一実施形態で、関心のあるメソッドのデータが、デバッグプローブ２２４の挿入動作中にＪＩＴコンパイラ２２０によって生成される。

【００３６】

ＪＭＣステップオーバであり、スレッドがメソッドの終りの命令を実行していない場合、ＪＭＣ＿Ｐｒｏｂｅ２２６は、従来のステップオーバのように振る舞うことになる。スレッドが、メソッドの終りに来ている場合、ＪＭＣ＿Ｐｒｏｂｅ２２６は、ステップインおよびステップアウトの双方のアクションを実行し、スレッドは、実行される関心のあるコードの最初のインスタンス（行）で停止することとなる。

【００３７】

ＪＭＣステッピングスレッドの停止によって示されるＪＭＣステッピングコマンド２２８の完了に応答して、デバッグサービス２１４によって、すべてのプローブが使用不可にされる（デバッガ２０６は、ＪＭＣステッピング動作だけを理解し、プローブについては何も知らない）。デバッグプローブ２２４を非アクティブ化することにより性能が最適化され、そうでなければ実行中のスレッドがＪＭＣステッピングを行っていない時に発生するＪＭＣ＿Ｐｒｏｂｅ２２６の無用な呼出しを回避する。

【００３８】

この実施形態では、フラグ２２２が、モジュールごとの基礎で使用可能／使用不可にされる（すなわち、１つのフラグが、各モジュールに挿入される）。このようにして、モジュール内の各メソッドが、同一の値のｐＦｌａｇを有することとなる。この特定の実施形態は、コードはモジュールごとに関心のあるものまたは関心のないものとなるという論理に基づく。特定の実施形態の設計の関数として、異なる実施形態で異なる推理を実施できることを理解されたい。たとえば、フラグ２２２を、メソッドごとに挿入することができるが、これによりフラグ２２２が多いほど、単一のフラグ２２２の値のフリップ／トグルによって使用可能にできるプローブが減ることとなる。

【００３９】

したがって、スレッドがアクティブなプローブ２２４に到達する場合、現在のスレッドにＪＭＣステッピングを行っていないときであっても、常にＪＭＣ＿Ｐｒｏｂｅ２２６の呼出しが行われる。ＪＭＣ＿Ｐｒｏｂｅ２２６により、スレッドにＪＭＣステッピングが行われていると判定される時に限って、そのスレッドは停止する。同様に、到達したデバッグプローブ２２４がアクティブでない場合、ＪＭＣ＿Ｐｒｏｂｅ２２６の呼出しは完全にスキップされ、スレッドはプロセスを介して通常の実行を続行する（すなわち、自由に動作する）。

【００４０】

この実施形態では、ＪＭＣ＿Ｐｒｏｂｅ関数２２６への呼出しは引数をとらず、戻り値がないので、実質的にサイズが最適化される。これによって、すべての「呼出しサイト（ｃａｌｌｓｉｔｅ）」（ＪＭＣ＿Ｐｒｏｂｅ関数２２６へのそれぞれの呼出しが行われる位置）が最適化される。したがって、デバッグプローブ２２４からのＪＭＣ＿Ｐｒｏｂｅへの呼出しがプローブから行われる場合（たとえば、＊ｐＦｌａｇ！＝０の時）、その呼出しは重い呼出しではない。これは、呼出しが関数パラメータのスタックへの挿入および取り出しを必要としないことを意味し、これは、ＪＭＣ＿Ｐｒｏｂｅ呼出しサイトが、パラメータの挿入および取り出しを行う関数への呼出しサイトと比較して、かなり負荷が小さいことを意味する。非アクティブのプローブ２２４に到達するスレッドがこうむるは、

唯一フラグ２２２の評価（＊ｐＦｌａｇを介する）および挿入されたプローブ２２４の次の命令へのジャンプに要するオーバヘッド処理だけである。このようにして、本明細書に記載のＪＭＣデバッグ動作を用いると、スレッドは、関心のあるコードに到達するまで、関心のないコードのすべてを抜けて実質的にフルスピードで自由に動作できるようになる。

【００４１】

本実施形態では、ランタイム２０８は実行可能コードおよびそれが動作する仮想実行環境の仕様を提供するＣＬＩ（Ｃｏｍｍｏｎ Ｌａｎｇｕａｇｅ Ｉｎｆｒａｓｔｒｕｃｔｕｒｅ）に基づく。したがって、図２では別々に図示されているが、プロセス２０８およびランタイム２０２は、同一の実行環境の一部である（すなわち、プロセス２０２によってランタイム２０８がホスティングされる）。

【００４２】

（例示的な手順）

図３に、ＪＭＣデバッグの例示の手順３００を示す。説明を目的として、手続き動作は、図１および２のプログラムモジュールおよびデータの特徴に関し説明される。ブロック３０２において、デバッガ２０６（図２）により、関心のあるコンピュータプログラムプログラミング構造（たとえば、オブジェクトメソッドおよび／または静的関数）が識別される。一実施形態では、これは、一態様としてのユーザ入力インターフェース１４４（図１）へユーザ入力を行うことによって達成される。たとえば、ユーザはデバッグサービスによりダイアログボックスに表示される複数のメソッドの特定の１つを（たとえば、ポインティングデバイス、キーボード、音声アクティベーテッドテクニックなどにより）選択することができ、またはユーザは、コマンドラインインターフェースおよび／またはその他のものにプログラミング構造の名前を入力することができる。ユーザ入力は、関心のあるコードリスト／ＩＤ２１２（図２）に保存され、リスト／ＩＤ２１２がランタイムコンパイラ２２０（図２）によって読み取られ、該当するプログラミング構造（すなわち関心のあるコード）が識別される。

【００４３】

ブロック３０４において、デバッガ２０６（図２）は、図２のジャストインタイム（ＪＩＴ）コンパイラ２２０などのコンパイラに対し、コンパイル動作中にコンピュータプログラムの各々のプログラミング構造（たとえばメソッド）にそれぞれＪＭＣイネーブリングコンポーネント２２２および２２４を自動的に挿入するように指示する。本実施形態では、ＪＩＴコンパイラ２２０がアセンブリをコンパイルしてネイティブコード２１０（図２）を生成し、このネイティブコード２１０に、それぞれのフラグ２２２およびデバッグプローブ２２４が自動的に挿入される。ブロック３０６において、ネイティブコード２１０（図２）が、プロセス２０２（図２）としてデバッガ２０６の制御の下で実行される。プロセス２０２には、プロセスの特定の実施形態の関数として任意の個数の実行のスレッドが含まれる。

【００４４】

ブロック３０８において、ユーザは、デバッガ２０６（図２）を使用し、ＪＭＣステッピングコマンド２２８（図２）の入力することにより、プロセス２０２を介してスレッドにＪＭＣステッピングを行うようにすることができる。スレッドにＪＭＣステッピングを行うことから、このスレッドを「関心のあるスレッド」と呼ぶ。プロセス２０２は、マルチスレッドとし、「関心のないスレッド」と呼ばれるＪＭＣステッピングを行われないスレッドを含めることができる。関心のあるスレッドは、関心のないコード（すなわち、ブロック３０２において選択されたプログラミング構造の選択された／個々の１つによって識別される関心のあるコードに対応しないすべてのコード）を介して自由に（停止することなく）実行される。関心のあるスレッドは、最初にアクティブのデバッグプローブ２２４に到達すると、デバッグサービス２１４により必ず自動的に停止される。プロセス２０２（図２）のスレッドが関心のあるスレッドであるかどうか、および関心のあるスレッドが選択されたプログラミング構造の１つの実行中／アクセス中のコードであるかどうかの

判定は、到達したデバッグプローブ２２４によって実行時に判定される。したがって、ＪＭＣステッピングを行われないすべてのスレッドは、関心のあるコードおよび関心のないコードの双方とも、すべての到達したデバッグプローブ２２４を介して自由に実行される。

【００４５】

（結論）

本明細書に記載のシステムおよび方法によって、ＪＭＣデバッグがもたらされる。システムおよび方法は、構造的特徴および方法論的動作に固有の言語で説明されたが、請求項で定義される主題は、必ずしも説明した特定の特徴または動作に制限されない。むしろ、特定の特徴および動作は、請求される主題を実施する例示的な形態として開示される。

【図面の簡単な説明】

【００４６】

【図１】ジャストマイコードのデバッグのシステムおよび方法をその中で実施できる例示的コンピューティング環境を示す図である。

【図２】ジャストマイコードのデバッグに関するアプリケーションプログラムおよびプログラムデータを含む、図１のシステムメモリの例示的態様をさらに示す図である。

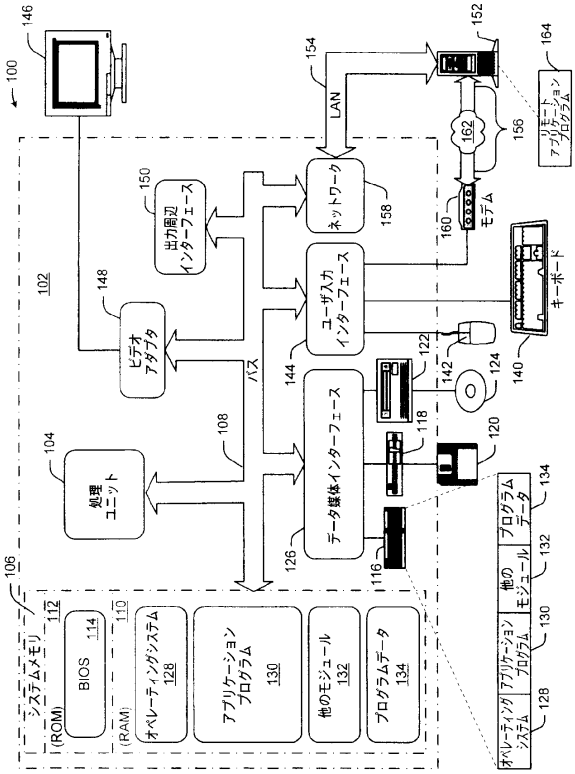
【図３】ジャストマイコードのデバッグの例示的手順を示す図である。

【符号の説明】

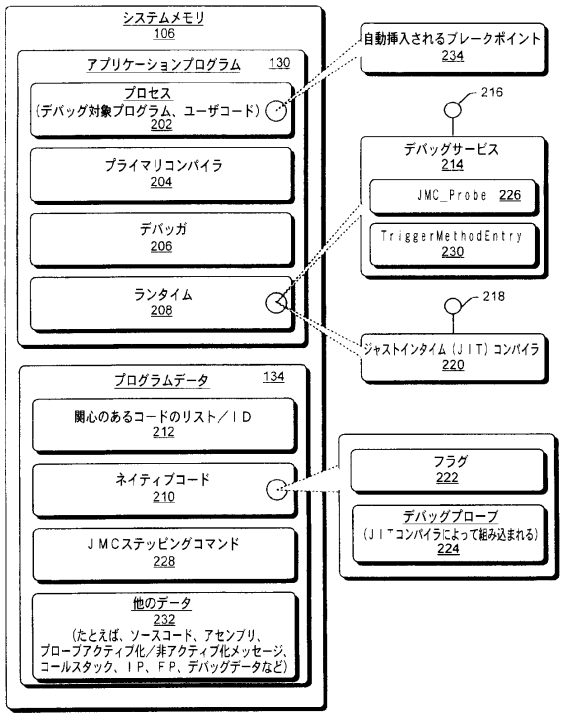
【００４７】

２０２　プロセス
２０４　プライマリコンパイラ
２０６　デバッグプログラム（「デバッガ」）
２０８　共有ランタイムコンポーネント（「ランタイム」）
２１０　ネイティブコード
２１２　関心のあるコードのリスト
２１４　デバッグサービス
２１６　デバッグＡＰＩ
２２０　ＪＩＴコンパイラ
２２２　フラグ
２２４　デバッグプローブ
２２６　ＪＭＣ＿Ｐｒｏｂｅ関数
２２８　ＪＭＣステッピングコマンド
２３０　ＴｒｉｇｇｅｒＭｅｔｈｏｄＥｎｔｒｙ
２３２　他のデータ
２３４　ブレークポイント

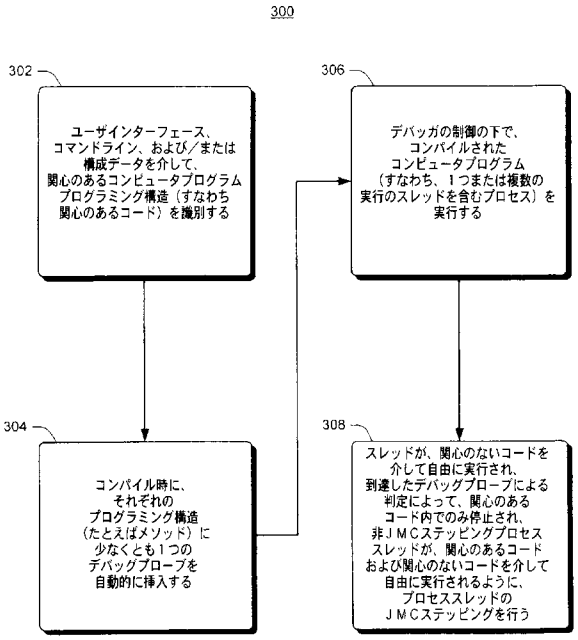
【図１】


100
102
104
106
108
110
112
114
116
118
120
122
124
126
128
130
132
134
140
142
144
146
148
150
152
154
156
158
160
162
164
LAN
モデム
ネットワーク
出力周辺インターフェース
ユーザ入力インターフェース
ビデオアダプタ
バス
処理ユニット
データ媒体インターフェース
キーボード
リモートアプリケーションプログラム
システムメモリ
(ROM)
(RAM)
BIOS
オペレーティングシステム
アプリケーションプログラム
他のモジュール
プログラムデータ

【図２】


システムメモリ
106
アプリケーションプログラム 130
プロセス
(デバッグ対象プログラム、ユーザコード)
202
プライマリコンパイラ
204
デバッガ
206
ランタイム
208
プログラムデータ 134
関心のあるコードのリスト／ＩＤ
212
ネイティブコード
210
ＪＭＣステッピングコマンド
228
他のデータ
232
(たとえば、ソースコード、アセンブリ、
プローブアクティブ化／非アクティブ化メッセージ、
コールスタック、ＩＰ、ＦＰ、デバッグデータなど)
自動挿入されるブレークポイント
234
216
デバッグサービス
214
JMC_Probe 226
TriggerMethodEntry
230
218
ジャストインタイム(ＪＩＴ)コンパイラ
220
フラグ
222
デバッグプローブ
(ＪＩＴコンパイラによって組み込まれる)
224

【図３】


300
302
ユーザインターフェース、
コマンドライン、および／または
構成データを介して、
関心のあるコンピュータプログラム
プログラミング構造（すなわち
関心のあるコード）を識別する
304
コンパイル時に、
それぞれの
プログラミング構造
（たとえばメソッド）に
少なくとも１つの
デバッグプローブを
自動的に挿入する
306
デバッガの制御の下で、
コンパイルされた
コンピュータプログラム
（すなわち、１つまたは複数の
実行のスレッドを含むプロセス）を
実行する
308
スレッドが、関心のないコードを
介して自由に実行され、
到達したデバッグプローブによる
判定によって、関心のある
コード内でのみ停止され、
非ＪＭＣステッピングプロセス
スレッドが、関心のあるコード
および関心のないコードを介して
自由に実行されるように、
プロセススレッドの
ＪＭＣステッピングを行う

フロントページの続き

(72)発明者　マイケル　エム．マグルーダ
アメリカ合衆国　９８０７５　ワシントン州　サマミッシュ　サウスイースト　１１　プレイス
２４０１８

審査官　多胡　滋

(56)参考文献　特開平０８－０２２４０１（ＪＰ，Ａ）
特開平０９－１４６７７６（ＪＰ，Ａ）
特開平０５－２３３３６４（ＪＰ，Ａ）
特開２０００－１８１７４７（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.，ＤＢ名)
Ｇ０６Ｆ　１１／２８
Ｇ０６Ｆ　１１／３６