【公報種別】特許法第１７条の２の規定による補正の掲載
【部門区分】第６部門第３区分
【発行日】平成23年5月26日(2011.5.26)

【公表番号】特表2007-529063(P2007-529063A)
【公表日】平成19年10月18日(2007.10.18)
【年通号数】公開・登録公報2007-040
【出願番号】特願2006-519993(P2006-519993)
【国際特許分類】
Ｇ０６Ｆ　9/45　(2006.01)
Ｇ０６Ｆ　9/455　(2006.01)
【ＦＩ】
Ｇ０６Ｆ　9/44　３２０Ｆ
Ｇ０６Ｆ　9/44　３１０Ａ

【誤訳訂正書】
【提出日】平成23年2月15日(2011.2.15)
【誤訳訂正１】
【訂正対象書類名】特許請求の範囲
【訂正対象項目名】全文
【訂正方法】変更
【訂正の内容】
【特許請求の範囲】
【請求項１】
対象プロセッサによって実行可能な対象プログラム・コードを、目的プロセッサによって実行可能であり且つトランスレータによって実行時にトランスレートされて生成される目的プログラム・コードに変換するための方法であって、コンピュータが、
前記対象プログラム・コードを複数の対象プログラム・コード単位に分割するステップと、
１以上の前記分割された対象プログラム・コード単位を１以上の目的プログラム・コード単位に変換するステップと、
前記目的プロセッサ上で前記変換された目的プログラム・コード単位を実行するステップと
を実行することを含み、
前記変換するステップは、
前記対象プログラム・コード中の対象関数を特定するステップであって、前記対象プログラム・コード中の対象関数はネイティブ・コードの対応する関数を有し、前記ネイティブ・コードが前記目的プロセッサによって実行可能なコードであり且つ前記トランスレータの外で直接コンパイルされて事前に生成されたものである、前記対象関数を特定するステップと、
前記ネイティブ・コードの前記ネイティブ関数を特定するステップであって、前記ネイティブ・コードの前記ネイティブ関数は前記特定された対象関数に対応する、前記ネイティブ関数を特定するステップと
を含み、
前記実行するステップは、
前記特定された対象関数の変換されたコードを実行する代わりに、前記目的プロセッサ上で前記ネイティブ関数を実行するステップ
を含む、前記方法。
【請求項２】
前記ネイティブ関数を実行するステップは、

関数パラメータを、目的コードからネイティブ・コードに変換するステップと、
前記ネイティブ関数のプロトタイプに従う前記変換された関数パラメータを用いて、前記ネイティブ関数を呼び出すステップと、
前記呼び出されたネイティブ関数の戻り値を、ネイティブ・コードから目的コードに変換するステップと
を含む、請求項１に記載の方法。
【請求項３】
前記変換するステップにおける変換は、該変換の中間コードを生成する、請求項１に記載の方法。
【請求項４】
前記変換するステップにおける変換は、目的コードを生成する、請求項１に記載の方法。
【請求項５】
前記ネイティブ関数を実行するステップは、
目的コードにおいて、すべての対象レジスタ値を目的コードからネイティブ・コードに変換するステップと、
統一された呼び出しスタブ・インターフェースに従う前記変換された対象レジスタ値を用いて、ネイティブ・コード呼び出しスタブ関数を目的コードから呼び出すステップと、
ネイティブ関数のプロトタイプに従う特定の対象レジスタ値及び／又はパラメータ・スタックを用いて、前記ネイティブ関数を前記呼び出されたネイティブ・コード呼び出しスタブ関数から呼び出すステップと
をさらに含む、請求項１～４のいずれか一項に記載の方法。
【請求項６】
前記ネイティブ関数を実行するステップは、
関数パラメータを、目的コードからネイティブ・コードに変換するステップと、
前記ネイティブ関数のプロトタイプに従う前記変換された関数パラメータを用いて、前記ネイティブ関数を呼び出すステップと、
前記呼び出されたネイティブ関数の結果を、ネイティブ・コードから目的コードに変換するステップと
を含む、請求項１～５のいずれか一項に記載の方法。
【請求項７】
前記関数パラメータを変換するステップ及び前記ネイティブ関数呼び出すステップは、対象コードにおいて、対象命令セットに加えられるトランスレータ固有の命令によって記述される、請求項２～６のいずれか一項に記載の方法。
【請求項８】
前記対象関数の特定及びその対応するネイティブ関数の特定が、結合点の記述を用いて行なわれる、請求項１～７のいずれか一項に記載の方法。
【請求項９】
前記対象コードのトランスレーションの間に、前記結合点の記述から特定される前記対象関数に遭遇すると、前記目的コード内に、前記ネイティブ関数に対する呼び出しスタブを挿入するステップをさらに含む、請求項８に記載の方法。
【請求項１０】
前記結合点の記述は、前記トランスレーションを行なうトランスレータ内に埋め込まれる、請求項８又は９に記載の方法。
【請求項１１】
トランスレーション実行の開始時に、前記結合点の記述を、記憶された結合点の記述ファイルから読み取るステップをさらに含む、請求項８～１０のいずれか一項に記載の方法。
【請求項１２】

前記結合点の記述は、対象コード内の位置及び対応するネイティブ関数を用いて行われる、請求項８～１１のいずれか一項に記載の方法。
【請求項１３】
前記結合点の記述は、対象コード内の位置及び呼び出されるべきコードに対する参照を用いて行われる、請求項８～１２のいずれか一項に記載の方法。
【請求項１４】
前記呼び出されるべきコードは目的コードである、請求項１３に記載の方法。
【請求項１５】
前記結合点の記述は、対象関数呼び出しの前、後、あるいはその代わりに目的コード内に挿入されるネイティブ関数呼び出しを用いて行われる、請求項８～１４のいずれか一項に記載の方法。
【請求項１６】
前記コンピュータが、
実行時シンボル・パッチングを実行するステップをさらに含み、
前記実行時シンボル・パッチングを実行するステップが、
前記対象プログラムのシンボル・テーブル内で対象対ネイティブ関数マッピングをエンコードするステップと、
前記対象プログラムの前記シンボル・テーブル内のエントリを特別なネイティブ結合マーカと取り替えるステップと、
前記特別なネイティブ結合マーカを、このマーカにトランスレーションの間に遭遇したときに、結合点の記述としてインタープリットして、呼び出すべき適切なネイティブ関数を特定するステップと
を含む、請求項８～１５のいずれか一項に記載の方法。
【請求項１７】
前記外部の呼び出しコマンドはトランスレータ固有のネイティブ結合命令であり、
前記方法は、前記コンピュータが、
前記対象コードのトランスレーションの間に、外部の呼び出しコマンドを特定する結合点の記述に遭遇したときに、トランスレーションのフローを変えて、前記外部の呼び出しコマンドを実行するステップを含む、請求項８～１６のいずれか一項に記載の方法。
【請求項１８】
前記外部の呼び出しコマンドを実行するステップは、
前記外部の呼び出しコマンドをインタープリットするステップと、
前記外部の呼び出しコマンドの中間コードを生成するステップとであって、前記コマンドは、関数パラメータを目的コードからネイティブ・コードに変換し、ネイティブ関数のプロトタイプに従う前記変換された関数パラメータを用いて、ネイティブ関数を呼び出す、前記生成するステップと
を含む、請求項１７に記載の方法。
【請求項１９】
前記外部の呼び出しコマンドを実行するステップは、
前記外部の呼び出しコマンドをインタープリットするステップと、
前記外部の呼び出しコマンドに対する目的コードを生成するステップであって、前記コマンドは、関数パラメータを目的コードからネイティブ・コードに変換し、ネイティブ関数のプロトタイプに従う前記変換された関数パラメータを用いて、ネイティブ関数を呼び出す、前記生成するステップと
を含む、請求項１７に記載の方法。
【請求項２０】
前記コンピュータが、
外部の呼び出しコマンドを対象コードに挿入するステップとであって、前記外部の呼び出しコマンドは、トランスレータ固有のネイティブ結合命令である、前記挿入するステップと、

対象コードのトランスレーションの間に前記外部の呼び出しコマンドを検出するステップと、
をさらに実行することを含む、請求項１～１９のいずれか一項に記載の方法。
【請求項２１】
前記コンピュータが、
対象コードのトランスレーションの間に前記外部の呼び出しコマンドに遭遇したときに、トランスレーションのフローを変えて前記外部の呼び出しコマンドを実行するステップをさらに含む、請求項２０に記載の方法。
【請求項２２】
前記外部の呼び出しコマンドを実行するステップは、
前記外部の呼び出しコマンドをインタープリットするステップと、
前記外部の呼び出しコマンドの中間コードを生成するステップとであって、前記コマンドは、関数パラメータを目的コードからネイティブ・コードに変換し、ネイティブ関数のプロトタイプに従う前記変換された関数パラメータを用いて、ネイティブ関数を呼び出す、前記生成するステップと
を含む、請求項２１に記載の方法。
【請求項２３】
前記外部の呼び出しコマンドを実行するステップは、
前記外部の呼び出しコマンドをインタープリットするステップと、
前記外部の呼び出しコマンドに対する目的コードを生成するステップであって、前記コマンドは、関数パラメータを目的コードからネイティブ・コードに変換し、ネイティブ関数のプロトタイプに従う前記変換された関数パラメータを用いて、ネイティブ関数を呼び出す、前記生成するステップと
を含む、請求項２１に記載の方法。
【請求項２４】
前記外部の呼び出しコマンドは、複数のサブ・コンポーネント命令を含む可変長命令である、請求項２０～２３のいずれか一項に記載の方法。
【請求項２５】
前記複数のサブ・コンポーネント命令はエスケープ・サブコンポーネント命令を含み、
前記スキゾ呼び出しコマンドを検出するステップは、前記エスケープ・サブコンポーネント命令を検出するステップをさらに含む、請求項２４に記載の方法。
【請求項２６】
前記エスケープ・サブコンポーネント命令はさらに、外部の呼び出しコマンドのその他のサブ・コンポーネント命令によって表わされる前記外部の呼び出しコマンドの形式を特定する、請求項２５に記載の方法。
【請求項２７】
前記コンピュータが、
ネイティブ結合実装スクリプト言語を構文解析してデコードするステップと、
トランスレーションの間に、前記ネイティブ結合実装スクリプト言語で作られたネイティブ結合スクリプトをインタープリットするステップと、
前記ネイティブ結合スクリプトの中間コードを生成して、関数パラメータを目的コードからネイティブ・コードに変換するステップと
をさらに実行することを含む、請求項１～２６のいずれか一項に記載の方法。
【請求項２８】
前記ネイティブ結合スクリプトの中間コードを、対象コードのブロックに対する中間コード・フォレストに統合するステップと、
前記中間コード・フォレストに対する目的コードを生成するステップと
をさらに含む、請求項２７に記載の方法。
【請求項２９】
前記コンピュータが、

目的コードにおいて、すべての対象レジスタ値を目的コードからネイティブ・コードに変換するステップと、
統一された呼び出しスタブ・インターフェースに従う前記変換された対象レジスタ値を用いて、ネイティブ・コード呼び出しスタブ関数を目的コードから呼び出すステップと、
前記ネイティブ・コード呼び出しスタブ関数をインタープリットするステップと、
ネイティブ・コード呼び出しスタブ関数結合スクリプトの中間コードを生成して、関数パラメータを目的コードからネイティブ・コードに変換するステップと
をさらに実行することを含む、請求項１～２８のいずれか一項に記載の方法。
【請求項３０】
前記コンピュータが、
ネイティブ・コード呼び出しスタブ関数の中間コードを、対象コードのブロックに対する中間コード・フォレストに統合するステップと、
前記中間コード・フォレストに対する目的コードを生成するステップと
をさらに実行することを含む、請求項１９に記載の方法。
【請求項３１】
前記ネイティブ関数を実行するステップは、
目的コードにおいて、すべての対象レジスタ値を目的コードからネイティブ・コードに変換するステップと、
前記変換された対象レジスタ値を用いて、目的コードから、ネイティブ・コード呼び出しスタブ関数を呼び出すステップと、
ネイティブ関数のプロトタイプに従う特定の対象レジスタ値及び／又はパラメータ・スタックを用いて、ネイティブ関数をネイティブ・コード呼び出しスタブ関数から呼び出すステップと
をさらに含む、請求項１に記載の方法。
【請求項３２】
前記コンピュータが、
ネイティブ・コード呼び出しスタブ関数実装スクリプト言語を構文解析するステップと、
前記構文解析されたネイティブ・コード呼び出しスタブ関数をコンパイルして、ネイティブ・コード実行可能モジュールにするステップと、
前記ネイティブ・コード実行可能モジュールを、トランスレーションを行なうための実行ファイルとリンクさせるステップと
をさらに実行することを含む、請求項１～３１のいずれか一項に記載の方法。
【請求項３３】
前記ネイティブ・コード実行可能モジュールは、
目的コードにおいて、すべての対象レジスタ値を目的コードからネイティブ・コードに変換するステップと、
前記変換された対象レジスタ値を用いて、ネイティブ・コード呼び出しスタブ関数を目的コードから呼び出すステップと、
ネイティブ関数のプロトタイプに従う特定の対象レジスタ値及び／又はパラメータ・スタックを用いて、ネイティブ関数を前記ネイティブ・コード呼び出しスタブ関数から呼び出すステップと
に対して実行可能である、請求項３２に記載の方法。
【請求項３４】
前記対象関数及びその対応するネイティブ関数を特定するステップは、結合点の記述を用いて行なわれ、
前記結合点の記述は、対象関数及びネイティブ・コード呼び出しスタブ関数を用いて行われる、請求項３２又は３３に記載の方法。
【請求項３５】
前記コンピュータが、

前記ネイティブ・コード実行可能モジュールのスクリプト言語中の前記ネイティブ・コード呼び出しスタブ関数のネイティブ関数の識別子をハード・コードするステップを実行するステップをさらに含む、請求項３４に記載の方法。
【請求項３６】
前記ネイティブ関数を実行するステップは、
目的コードにおいて、すべての対象レジスタ値を目的コードからネイティブ・コードに変換するステップと、
前記変換された対象レジスタ値を用いて、目的コード呼び出しスタブ関数を目的コードから呼び出すステップと、
ネイティブ関数のプロトタイプに従う特定の対象レジスタ値及び／又はパラメータ・スタックを用いて、ネイティブ関数を目的コード呼び出しスタブ関数から呼び出すステップと
をさらに含む、請求項１に記載の方法。
【請求項３７】
前記コンピュータが、
前記ネイティブ関数を実行するステップにおいて、中間コードを生成するステップと、
前記生成された中間コードを、対象コードのブロックに対する中間コード・フォレストに統合するステップと、
前記中間コード・フォレストに対する目的コードを生成するステップと
をさらに実行することを含む、請求項３６に記載の方法。
【請求項３８】
前記実行されるべき対象関数は、システム呼び出し関数、又は、ライブラリ関数である、請求項１～３７のいずれか一項に記載の方法。
【請求項３９】
コンピュータに請求項１～３８のいずれか一項に記載の方法の各ステップを実行させるコンピュータ・プログラム。
【請求項４０】
コンピュータであって、
目的プロセッサと、
前記目的プロセッサによって請求項１～３８のいずれか一項に記載の方法の各ステップが実行されるトランスレータ・コードを含むメモリと
を備えている、前記コンピュータ。
【誤訳訂正２】
【訂正対象書類名】明細書
【訂正対象項目名】全文
【訂正方法】変更
【訂正の内容】
【発明の詳細な説明】
【発明の名称】ネイティブ結合を行なうための方法および装置
【技術分野】
【０００１】
本発明は、コンピュータおよびコンピュータ・ソフトウェアの分野に関し、より詳細には、たとえば、関数呼び出しを含むプログラム・コードをトランスレートするコード・トランスレータ、エミュレータ、およびアクセラレータにおいて有用なプログラム・コード変換方法および装置に関する。
【背景技術】
【０００２】
組込型および非組込型ＣＰＵの両方において、主流的な命令セット・アーキテクチャ（ＩＳＡ：InstructionSet Architecture）に対して、大量のソフトウェアが存在する。これらのソフトウェアは、性能を上げるために「アクセラレート」されることもできるし、

より優れたコスト・パフォーマンスの利点を示す無数の高性能なプロセッサに「トランスレート」されることもできる。ただしそのようなプロセッサは、関連するソフトウェアにトランスペアレントにアクセスできなければならない。また、主流的なＣＰＵアーキテクチャは、そのＩＳＡに歩調を合わせなければならないため、性能または市場の範囲を発展させることができない。そのようなアーキテクチャには、「複合ＣＰＵ（synthetic CPU）」共通アーキテクチャが有用であろう。

【０００３】

前述のアクセラレーション、トランスレーション、および共通アーキテクチャ能力を促進するプログラム・コード変換方法および装置が、たとえば同時継続中の英国特許出願第０３０９０５６０号明細書（２００３年４月２２日出願）、発明の名称「プログラム・コード変換に対するブロック・トランスレーション最適化（Block Translation Optimization for Program Code Conversion）」において扱われている。

【発明の開示】

【発明が解決しようとする課題】

【０００４】

トランスレートされるべき対象（subject）プログラムは通常、対象コードの複数の単位からなる。これらには、対象実行可能コード自体および多くの対象ライブラリが含まれる。対象ライブラリのいくつかは独占的である場合があり、いくつかは対象ＯＳの一部（「システム・ライブラリ」）として提供される。対象プログラムが実行されると、関数呼び出しが外部のライブラリに対して行なわれるため、対象コードのこれらの異なる単位の間で制御フローが移る。状況によっては、ある特定の対象ライブラリのネイティブ（すなわち、目的（target）アーキテクチャ）バージョンが、目的アーキテクチャ上で利用可能な場合もある。

【課題を解決するための手段】

【０００５】

本発明によれば、添付請求項で述べるような装置および方法が提供される。本発明の好ましい特徴は、従属請求項および以下の説明から明らかとなる。

以下は、本発明による種々の実施形態による実現可能な種々の態様および優位性の概要である。この概要は、当業者が、後に続く詳細な構成の説明を迅速に理解するための助けとなる導入部として設けられている。概要は、添付の請求項の範囲を限定するものではなく、また決して限定することが意図されているわけではない。

【０００６】

詳細には、本発明者らは、プログラム・コード変換の促進に向けられた最適方法を開発した。この方法は、対象プログラム・コードの目的コードへのトランスレーションを用いるランタイム（実行時）トランスレータと共に用いると、特に有用である。対象コードを目的コードにトランスレーションする間にネイティブ関数に対する呼び出しを挿入するためのネイティブ結合技術が提供される。従って、対象コード関数に対する対象プログラム内での関数呼び出しが、目的コードにおいて、同じ関数のネイティブ同等物に対する呼び出しに置き換えられる。ネイティブ・コード表現、ネイティブ・コード呼び出し規約、およびネイティブ関数プロトタイプと整合するように、ネイティブ関数呼び出しのパラメータが目的コード表現から変換される。

【０００７】

ネイティブ結合は、本発明のメカニズムであって、トランスレート済み対象コードが、等価な対象コードをトランスレートしおよび実行するのではなくて、ネイティブ（すなわち、目的アーキテクチャ）バージョンのコード単位（たとえばネイティブ・バージョンのライブラリなど）を直接実行できるようにするものである。この結果、対象バージョンのこれらのライブラリをトランスレートするオーバーヘッドが回避される。加えて、ネイティブ・バージョンは、同じ機能をはるかに効率的な実装であり得る。なぜならば、ネイティブ・バージョンは、対象バージョンが認識していない目的アーキテクチャのアーキテクチャ特徴を利用できるからである。

【０００８】
添付図面は、本明細書に取り入れられて本明細書の一部を構成しているが、現時点で好ましい実施形態を例示するものであり、後述するように説明される。
【発明を実施するための最良の形態】
【０００９】
図１は、後述する種々の新しい特徴を実施するための例示的な装置を示す。図１には、目的レジスタ１５を含む目的プロセッサ１３とともに、メモリ１８が例示されている。メモリ１８には、多くのソフトウェア・コンポーネント１７，１９，２０，２１，２２，２８が記憶されている。ソフトウェア・コンポーネントには、トランスレートされるべき対象コード１７、オペレーティング・システム２０、トランスレータ・コード１９、トランスレート済みコード２１、グローバル・レジスタ記憶装置２７、ネイティブ結合メカニズム２２、およびネイティブ・コード２８が含まれる。トランスレート済みコード２１は、目的コード２１とも呼ばれる。グローバル・レジスタ記憶装置２７は、対象レジスタ・バンク２７とも呼ばれる。トランスレータ・コード１９は、たとえば、あるＩＳＡの対象コードを他のＩＳＡのトランスレート済みコードにトランスレートするエミュレータとしてか、あるいは対象コードをトランスレート済みコード（それぞれ、同じＩＳＡである）にトランスレートするためのアクセラレータとして機能しても良い。ネイティブ・コード２８は、目的プロセッサ１３に対して直接的にコンパイルされるコードである。
【００１０】
トランスレータ１９、すなわちトランスレータを実装するコンパイル済みバージョンのソース・コードと、トランスレート済みコード２１、すなわちトランスレータ１９によって生成された対象コード１７のトランスレーションとは、目的プロセッサ１３（通常は、マイクロプロセッサまたは他の好適なコンピュータ）上で実行されるオペレーティング・システム２０とともに実行される。当然のことながら、図１に例示した構造は単に典型であり、またたとえば本発明によるソフトウェア、方法、およびプロセスは、オペレーティング・システム内にまたはその下に存在するコードにおいて実装しても良い。対象コード１７、トランスレータ・コード１９、オペレーティング・システム２０、および記憶メカニズムは、多種多様のタイプのいかなるものであっても良い。これは当業者には知られている。
【００１１】
本明細書で用いられる場合、「目的コード」２１（対象コード１７のフラグメントの実行時トランスレーションから生じる）と、「ネイティブ・コード」２８（目的アーキテクチャのために直接的にコンパイルされる）との間に違いがある。目的オペレーティング・システムによって用意されるシステム・目的ライブラリは、ネイティブ・コード２８の例である。システム・対象ライブラリのトランスレーションは、目的コード２１の例である。ネイティブ・コード２８は、トランスレータ１９の外で生成される。すなわちトランスレータ１９はネイティブ・コード２８を生成しないし、トランスレータ１９はネイティブ・コード２８を最適化する機会もない。
【００１２】
図１による装置では、プログラム・コード変換は好ましくは、目的コード２１が実行されている実行時に動的に行なわれる。トランスレータ１９は、トランスレート済みプログラム２１にインラインで実行される。前述したトランスレータ１９は通常、目的アーキテクチャに対してコンパイルされるアプリケーションとして用いられる。対象プログラム１７は、実行時にトランスレータ１９によってトランスレートされて、目的アーキテクチャ上で直接的に実行される。またトランスレータ１９は、対象プログラム１７によって起こされる対象オペレーティング・システム（ＯＳ：Operating System）呼び出しを変換して、この呼び出しが目的ＯＳ２０に受け渡されたときに正しく動作するようにする。
【００１３】
目的コード２１を生成する過程で、中間表現（「ＩＲ：intermediaterepresentation」）ツリーが対象命令シーケンスに基づいて生成される。ＩＲツリーは、対象プログラムに

よって実行計算される式および対象プログラムによって行なわれる動作（operations）の抽象表現である。後に、目的コード２１がＩＲツリーに基づいて生成される。

【００１４】

本明細書で説明するＩＲノードを集めたもの（collections）が、俗称的に「ツリー」と呼ばれる。なお正式には、そのような構造は実際には、有向非循環グラフ（ＤＡＧ：Directed Acyclic Graph）であり、ツリーではないことに注意されたい。ツリーの正式な定義によれば、各ノードには、多くても１つの親が存在する必要がある。説明する実施形態では、ＩＲの生成中に共通部分式の除去が用いられるため、ノードには複数の親があることが多い。たとえば、フラッグに影響を与える命令結果のＩＲは、２つの抽象レジスタ（格納先対象レジスタおよびフラッグ結果パラメータに対応する）によって参照されうる。

【００１５】

たとえば、対象命令「ａｄｄ　％ｒ１，％ｒ２，％ｒ３」が、対象レジスタ％ｒ２および％ｒ３の内容の加算を実行し、結果を対象レジスタ％ｒ１内に記憶する。すなわち、この命令は、抽象式「％ｒ１＝％ｒ２＋％３」に対応する。この例は、抽象レジスタ％ｒ１の定義とともに、命令オペランド％ｒ２および％ｒ３を表わす２つの部分式を含む加算式を含む。対象プログラム１７に関連して言えば、これらの部分式は、それより前の他の対象命令に対応していても良いし、あるいはそれらの部分式は、現在の命令の詳細、たとえば直接定数値（immediate constant values）を表していても良い。

【００１６】

「ａｄｄ」命令が構文解析されると、加算に対する抽象的な数学演算子に対応する、新しい「＋」ＩＲノードが生成される。「＋」ＩＲノードは、オペランド（ＩＲにおいて部分式ツリーとして表わされ、対象レジスタに保持されることが多い）を表わす他のＩＲノードに対する参照を記憶する。「＋」ノード自体は、「＋」ノードによって値が規定される対象レジスタ（％ｒ１に対する抽象レジスタ、命令の格納先レジスタ）によって参照される。たとえば、図２０の中心より右側の部分には、Ｘ８６命令「ａｄｄ　％ｅｃｘ，％ｅｄｘ」に対応するＩＲツリーが示されている。

【００１７】

当業者であれば理解できるように、一実施形態において、トランスレータ１９は、オブジェクト指向のプログラミング言語、たとえばＣ＋＋を用いて実装される。たとえば、ＩＲノードは、Ｃ＋＋オブジェクトとして実装され、他のノードに対する参照は、この他のノードに対応するＣ＋＋オブジェクトに対するＣ＋＋参照として実装される。それ故に、ＩＲツリーは、互いに対する様々な参照を含むＩＲノード・オブジェクトを集めたもの（collections）として実装される。

【００１８】

また、説明している本実施形態において、ＩＲ生成は、抽象レジスタの組を用いる。これら抽象レジスタは、対象アーキテクチャの特定の特徴に対応する。たとえば、対象アーキテクチャ上の各物理レジスタ（「対象レジスタ」）に対して、一意の抽象レジスタが存在する。同様に、対象アーキテクチャ上に存在する各状態コード・フラッグに対して、一意の抽象レジスタが存在する。抽象レジスタは、ＩＲ生成中にＩＲツリーに対するプレースホルダとして機能する。たとえば、対象命令シーケンス内の所定の点における対象レジスタ％ｒ２の値は、対象レジスタ％ｒ２に対する抽象レジスタに関連付けられる特定のＩＲ式ツリーによって表わされる。一実施形態において、抽象レジスタはＣ＋＋オブジェクトとして実装される。Ｃ＋＋オブジェクトは、特定のＩＲツリーに、そのツリーのルート・ノード・オブジェクトに対するＣ＋＋参照を介して関連付けられる。

【００１９】

抽象レジスタの実装は、トランスレータ・コード１９および目的コード２１両方におけるコンポーネント間で分割される（divided）。トランスレータ１９内では、「抽象レジスタ」は、ＩＲ生成の過程で使用されるプレースホルダである。従って、抽象レジスタが、特定の抽象レジスタが対応する対象レジスタの値を計算するＩＲツリーに関連付けられる。したがって、トランスレータ内の抽象レジスタは、ＩＲノード・オブジェクト（すな

わちＩＲツリー）への参照を含むＣ＋＋オブジェクトとして実装されても良い。抽象レジスタ・セットによって参照されるすべてのＩＲツリーの集合体は、作業用ＩＲフォレストと呼ばれる（「フォレスト」と呼ばれる理由は、集合体が複数の抽象レジスタ・ルートを含み、それぞれがＩＲツリーを参照するからである）。作業用ＩＲフォレストは、対象コードの特定の点における対象プログラムの抽象的な操作のスナップショットを表わす。

【００２０】

基本ブロック・モードでは、状態が、すべてのトランスレート済みコード・シーケンスにとってアクセス可能なメモリ領域、すなわちグローバル・レジスタ記憶装置２７を用いて、ある基本ブロックから次の基本ブロックへ受け渡される。グローバル・レジスタ記憶装置２７は、抽象レジスタに対するリポジトリである。各抽象レジスタは、特定の対象レジスタの値または他の対象アーキテクチャ特徴に対応し、且つこれをエミュレートする。目的コード２１を実行する間、抽象レジスタは、命令に関与できるように、目的レジスタに一時的に保持される。トランスレータ・コード２１を実行する間、抽象レジスタ値は、グローバル・レジスタ記憶装置２７または目的レジスタ１５内に記憶される。

【００２１】

目的コード２１内では、「抽象レジスタ」は、グローバル・レジスタ記憶装置内の特定の箇所であり、その箇所まで、およびその箇所から、対象レジスタ値が実際の目的レジスタと同期する。あるいは、値がグローバル・レジスタ記憶装置からロードされている場合には、目的コード２１内の抽象レジスタは、目的レジスタ１５であると理解することができ、それは、目的コード２１の実行中に、対象レジスタ値を、それがレジスタ記憶装置に再び保存される前に一時的に保持する。

【００２２】

こうして、トランスレータ１９の下で実行中の対象プログラムには、交互的に実行される２つの異なる形式のコードがある。すなわちトランスレータ・コード１９および目的コード２１である。トランスレータ・コード１９は、トランスレータ１９の高レベルのソース・コードの実装に基づいて、実行時の前にコンパイラによって生成される。目的コード２１は、トランスレートされるプログラムの対象コード１７に基づいて、実行時の間中、トランスレータ・コード１９によって生成される。

【００２３】

対象プロセッサ状態の表現は同様に、トランスレータ１９および目的コード２１のコンポーネント間で分割される。トランスレータ１９は、対象プロセッサ状態たとえば変数および／またはオブジェクトを、種々の明示的なプログラミング言語装置に記憶する。トランスレータをコンパイルするために用いられるコンパイラは、状態および動作が目的コードにおいてどのように実装されるかを決定する。比較して、目的コード２１は、対象プロセッサ状態を暗黙的に、目的レジスタおよびメモリ箇所に記憶する。目的レジスタおよびメモリ箇所は、目的コード２１の目的命令によって直接扱われる。

【００２４】

たとえば、グローバル・レジスタ記憶装置２７の低レベルの表現は単純に、割り当てメモリの領域である。これは、どのようにして目的コード２１が、定義されたメモリ領域と種々の目的レジスタとの間で保存および復元することによって、抽象レジスタを見て、そして抽象レジスタとやり取りするかである。しかし、トランスレータ１９のソース・コードにおいては、グローバル・レジスタ記憶装置２７は、より高いレベルでアクセスおよび扱われうるデータ・アレイまたはオブジェクトである。目的コード２１に関して言えば、単純に、高レベルの表現は存在しない。

【００２５】

図２～５は、図１のトランスレータ１９の種々の例示的な実施形態を示す。図２は、トランスレータ１０５によってトランスレートされるべき対象コード１７における対象プログラム１０６の異なるコンパイル単位の例示的な実施形態を示す。トランスレータ１０５は、ネイティブ・アーキテクチャ（さもなくば、目的アーキテクチャと呼ばれる）に対してコンパイルされた実行ファイル（executable）として実行する。ネイティブ・アーキテ

クチャは、ネイティブＯＳ１０３およびネイティブ・プロセッサ１０１（それらは図１の目的ＯＳ２０および目的プロセッサ１３に等価である）の両方を含む。この例の対象プログラム１０６は、対象実行ファイル１０７および多くの対象ライブラリを含む。対象ライブラリは、対象独占的ライブラリ１０９および対象システム・ライブラリ１１１を含みうる。ネイティブ結合が用いられない場合には、対象プログラム１０６のコンパイル単位は、目的コード２１にトランスレートされて、トランスレータ１０５内で実行される。

【００２６】

図３は、ネイティブ結合が、トランスレート済みプログラムのコード単位で利用される場合の例示的な実施形態の動作フローのブロック図である。

対象プログラム１０６は、対象実行ファイル１０７および多くの対象ライブラリを含む。対象ライブラリは、独占的ライブラリ１０９およびシステム・ライブラリ１１１を含む。トランスレータ１０５は、ネイティブ結合を用いて、対象システム・ライブラリ関数１１１に対する対象プログラム１０６呼び出しを、ネイティブ・システム・ライブラリ１１７における関数に対する呼び出しに置換する。

【００２７】

たとえば、ＭＩＰＳ－ｘ８６トランスレーションの場合、ｘ８６システム・目的ライブラリ「ｌｉｂｃ」は、ＳＳＥ２ベクトル操作を利用して極めて速いバイト・コピーを行なう高性能なｍｅｍｃｐｙ（）（メモリ・コピー）ルーチンを定義することができる。本出願で説明されるネイティブ結合メカニズム２２を用いて、ＭＩＰＳ対象コードにおけるｍｅｍｃｐｙに対するすべての呼び出しが、ネイティブｍｅｍｃｐｙ（）に結合される。これは、対象（ＭＩＰＳ）バージョンのｍｅｍｃｐｙ（）関数をトランスレートするコストを削減する。加えて、ネイティブ（ｘ８６）バージョンのｍｅｍｃｐｙ（）関数は、ネイティブ・ハードウェアの複雑さについてはるかに良く認識しているため、関数の所望の効果を達成するための最も効率的な方法を知っている。

【００２８】

対象プログラムの制御フローが、対象コードのネイティブ・バージョンが存在する対象コード１７のセクション（たとえば対象ライブラリ）に入ることが検出されたときに、ネイティブ結合が、トランスレータ１９によって実装されるである。トランスレータ１９は、対象コード１７をトランスレートするのではなくて、等価なネイティブ・コード２８を実行する。

【００２９】

目的システム・ライブラリの呼び出しに加えて、より多くの任意コードの代用に対して、たとえばネイティブでコンパイルされたバージョンの非ライブラリ関数を代用することに対して、ネイティブ結合が用いられうる。さらに、ネイティブ結合は、対象システム関数に対するすべての呼び出しを、代用のネイティブ関数と置換することによって、ネイティブ・アーキテクチャ上で対象システムの呼び出しを実装しうる。代用のネイティブ関数は、対象システム関数に対する呼び出しと同じ機能を実装し、あるいは目的システムの呼び出しの周りの呼び出しスタブとしての機能を果たす。またネイティブ結合は、関数呼び出しサイトを過ぎた任意の対象コード箇所で適用されて、（目的コードまたはネイティブ・コード内の）任意コードのシーケンスおよび／または対象プログラム内の任意の十分に定義された点で挿入または代用される関数呼び出しを可能にする。

【００３０】

結合点（Bind Point）の記述(Descriptions)

ネイティブ結合メカニズム２２では、トランスレータ１９が、特定の対象コード１７関数を、それらのネイティブ・コード２８対応ファイルと関連づける必要がある。その結果、トランスレータ１９は、どの対象関数を結合すべきか、およびどのネイティブ関数に対して対象関数を結合すべきかを知る。トランスレータ１９は、ネイティブ結合メカニズム２２の実装に依存して、種々の方法でこの関数マッピング情報を取得することができる。

【００３１】

一実施形態において、結合されるべき対象関数は、特別目的の「結合点」記述言語を用

いて識別される。結合点の記述は、以下を含む。すなわち、（ａ）結合されるべき対象関数、及び（ｂ）結合されるべき対応するネイティブ関数である。トランスレータ１９は、実行の開始時に結合点の記述を読み込んで、結合点（ネイティブ関数を呼び出す箇所）を識別する。対象プログラムのデコード中に、トランスレータ１９がこれらの結合点に遭遇すると、トランスレータ１９は、適切なネイティブ関数に対する呼び出しスタブを、目的コード２１内に挿入する。一実施形態において、特定の結合点の記述が、トランスレータ１９内に埋め込まれる。他の実施形態において、結合点の記述が、トランスレータ１９が実行時に読み込む個々のファイル中に記憶される。その結果、エンド・ユーザが、ネイティブ結合メカニズム２２の制御を特定の対象対ネイティブ関数マッピングを加えることによって行なうことを許す。

【００３２】

他の実施形態において、ネイティブ結合用結合点の記述言語は、任意の結合点が特定されることを許し、従って、ネイティブ関数呼び出しが、対象関数呼び出しを過ぎた対象コード内の他の点において挿入されうる。この実施形態において、結合点の記述は以下を含む。すなわち（ａ）対象プログラム内の定義された箇所（すなわち、関数呼び出しサイトだけではない）、及び（ｂ）結合されるべき対応するネイティブ関数である。たとえば、任意の結合点は、（１）関数の開始、（２）対象モジュールの初期化コードの開始、（３）特定の記号からの固定された相対位置（オフセット）（たとえば、関数の開始からの固定された相対位置）、（４）モジュール内の第１のテキスト・セグメントからの固定された相対位置、または（５）（特定のモジュール内か、またはある特定のモジュールを除いたすべてのモジュールにおける）特定の対象関数に対するすべての呼び出し、として識別されうる。結合点の形式（１）と（５）との間の差は、（１）は対象関数のエントリ・ポイントを結合し、一方、（５）は関数の呼び出しサイトを結合するということである。

【００３３】

いくつかの実施形態において、ネイティブ結合用結合点の記述言語は、エンド・ユーザが相対的な結合点を特定することを許す。この場合、ネイティブ関数呼び出しが、結合点（たとえばシステム・対象関数呼び出し）の前、後、またはその代わりに挿入されうる。たとえば、ネイティブ結合用結合点の記述は、ネイティブ関数「ｆｏｏ（）」が対象関数「ｂａｒ（）」に対するすべての呼び出しの直後に呼び出されることを特定しうる。

【００３４】

また結合点の記述言語は、ネイティブ関数呼び出し以外のコードを結合点において挿入するために使用されうる。そのような実施形態において、結合点の記述は、（ａ）対象プログラム内の定義された箇所、（ｂ）呼び出されるべき目的コード・ブロックまたはネイティブ・コード関数に対する参照、を含む。挿入されるコードが目的コード２１である場合には、トランスレータ１９は、結合点において、パラメータ変換およびネイティブ呼び出し規約（後述する）に関連付けられたほとんどの動作を行なう必要がない。この場合、トランスレータ固有の目的コード呼び出し規約に準拠することで十分である。任意の目的およびネイティブ・コードの挿入は、トランスレータ１９が、トランスレート済みプログラム上の他のタスク、たとえばデバッギングおよびパフォーマンス・プロファイリングを実行することを許す。

【００３５】

代替的な実施形態において、対象対ネイティブ関数マッピングは、対象プログラムの記号テーブルにおいて、実行時前に、実行時記号パッチングと呼ばれるプロセスにおいてエンコードされる。実行時記号パッチングは、対象プログラムの記号テーブル内のエントリを、特別なネイティブ結合マーカに置換する。これは、対象プログラムの扱いを、対象プログラムがコンパイルされた（コンパイル時）後で、しかし対象プログラムがトランスレートされる（実行時）前に、行なうことが必要とする。トランスレータ１９が実行時に記号テーブル・マーカーに遭遇すると、トランスレータ１９は、記号テーブル・マーカーを結合点の記述としてインタープリットし、かつ、記号テーブル・マーカーをインタープリットして、どのネイティブ関数を呼び出すべきかを識別する。本実施形態において、結合

されるべき対象関数の識別子（identity；以下、身元ともいう）は、記号テーブル内の記号テーブル・マーカーの箇所において暗黙的である。なぜならば、記号テーブル・マーカーは、特定の対象関数に対応する記号テーブル・エントリ内に置かれるからである。

【００３６】

結合点の記述によって結合点を明白に識別する代わりに、結合点が、代替的に、トランスレータ固有の対象命令セット拡張部分によって暗黙的に識別されうる。トランスレータ固有の対象命令セット拡張部分は、コンパイルされると、対象コードにおいて移植（plant）される（後述の「Ｓ－呼び出し」を参照のこと）。

【００３７】

パラメータ変換

ネイティブ関数を呼び出すときに、トランスレータ１９は、目的アーキテクチャの呼び出し規約に適合しなければならない。比較して、目的コード２１が目的アーキテクチャ呼び出し規約に準拠することは、トランスレータ１９が目的コード２１の全体を通して何らかの一貫した呼び出し規約に準拠する限り、必ずしも必要ではない。加えて、トランスレータ１９は、対象装置状態（目的コード２１において表わされる）とネイティブ装置状態（ネイティブ・コード２８において表わされる）との間のデータ変換を実行することを必要としうる。両状態とも、ネイティブ関数の入力パラメータおよびその戻り値（もしあれば）に対するものである。そのようなデータ変換の例は、（ｉ）エンディアン変換（すなわち、バイト交換（swapping））、（ｉｉ）データ構造アライメント（alignment）、（ｉｉｉ）対象アドレスと目的アドレスとの間の変換、及び（ｉｖ）値変換（たとえば、定数変換または値スケーリング）、を含む。

【００３８】

たとえば、ＭＩＰＳアーキテクチャ上では、関数パラメータがレジスタ内で受け渡され、一方、ｘ８６アーキテクチャ上では、パラメータがスタック上で受け渡される。ＭＩＰＳｘ８６トランスレータがネイティブ関数を呼び出すためには、ｘ８６呼び出し規約は、関数パラメータを対象レジスタからスタックへ移動させることを必要とする。

【００３９】

図６は、ネイティブ関数を呼び出すためにトランスレータ１９によって実行されるステップを示す。ネイティブ関数を呼び出すためには、トランスレータ１９は、複数のステップを実行しなければならない。すなわち、パラメータ・セットアップ５０１、入力パラメータ変換５０３、ネイティブ関数呼び出し５０５、及び結果変換５０７である。パラメータ・セットアップ５０１は、関数呼び出しパラメータの値を計算する目的コード２１を参照する。入力パラメータ変換５０３は、関数呼び出しパラメータ値を、それらの目的コード２１から、ネイティブ関数コードが予想するフォーマットおよび箇所に構成する。ネイティブ関数呼び出し５０５は、ネイティブ関数に対する実際の関数呼び出しであり、且つ、ネイティブ呼び出し規約に準拠する仕方で、関数プロトタイプによって必要とされる順番に、（再フォーマット済み）パラメータを構成することを含む。関数プロトタイプは、関数のパラメータの順番および形式と、関数の戻り値の形式とを示す。たとえば、ネイティブ呼び出し規約が、引数がスタック上で受け渡されることを必要とする場合には、ネイティブ関数を呼び出す目的コード２１は、引数をスタック上に正しい順番で置いて、スタック・ポインタを相応に進めなければならない。結果変換５０７は、関数の戻り値（もしあれば）を変換し、当該関数は、ネイティブ・アーキテクチャに整合するフォーマットで値を戻し、トランスレータはこの値を、目的コード２１によって用いられる表現に変換する。

【００４０】

パラメータ・セットアップ５０１は、ネイティブ結合メカニズム２２に専用のものではない。なぜならば、目的コード２１は、関数がトランスレート済み対象コード２１として又はネイティブ・コード２８として呼び出されるのかに関係なく、パラメータ値を計算しなければならないからである。ネイティブ関数呼び出しがどの特定の対象レジスタを（パラメータ値として）使用するのかを、トランスレータ１９が知らない場合には、トランス

レータ１９は、使用される対象レジスタの値を修正して、対象レジスタ・バンク２７が一貫した状態にあることを保証しなければならない。対象レジスタ値の計算は、遅延評価などのトランスレータ最適化によって、これらの値が必要とされるまで延期されても良い。この場合、評価が延期されたレジスタの強制的な計算は、修正によって参照される。修正されると、対象レジスタ値は次に、対象レジスタ・バンク２７に記憶される。

【００４１】

関数の明示的なパラメータの値を計算すること（場合によっては、すべての対象レジスタの修正を必要とする）に加えて、パラメータ・セットアップ・ステップ５０１は、対象メモリ空間が一貫した状態にあることも保証しなければならない。なぜならば、ネイティブ呼び出しが、メモリ・アクセスの形態において副作用を有する場合があるからである。一実施形態において、（ネイティブ呼び出しスタブに対するか、あるいは基礎となるネイティブ関数に対する）ネイティブ関数呼び出しをエンコードするＩＲによって、対象メモリ状態を修正する。この修正は、対象プログラムにおいて関数呼び出しの前に行なわれるであろうすべてのロードおよび記憶が、ネイティブ関数呼び出しの前に目的コードにおいてプラントされるように、かつ、同様に、関数呼び出しの後に行なわれるべきメモリ・アクセスが、ネイティブ呼び出しの前にまったくプラントされないように行なわれる。

【００４２】

「パラメータ変換」５０９は、ステップ５０３，５０５，及び５０７を一括して参照するために、すなわち、異なるデータ・フォーマット間のすべての個々の変換と、目的コード２１およびネイティブ・コード２８の呼び出し規約とを参照するために用いられる。パラメータ変換５０９を実行するコードは、「呼び出しスタブ」と呼ばれる。「呼び出しスタブ」は、基礎となるネイティブ関数呼び出しの周りの最小限のラッパーを含み、その唯一の目的は、目的コード２１呼び出し元が、ネイティブ・コード２８呼び出し先とやり取りできることを許すことである。したがって単一の「呼び出しスタブ」は、目的コード２１およびネイティブ・コード２８のコンポーネントに分割されうる。パラメータ変換５０９が、全体に目的コード２１において起こるのか、あるいは部分的にネイティブ・コード２８において起こるのかは、ネイティブ結合の実装に依存する。

【００４３】

ネイティブ・コード・スタブ

いくつかの実施形態において、ネイティブ結合に対するパラメータ変換５０９は、ネイティブ・コード２８によって部分的に行なわれる。ネイティブ・コード・スタブは、トランスレータ１９によって最適化されないという不利な点を有する。これらの実施形態において、目的コード２１が、なんらかのパラメータ変換を実行し、ネイティブ呼び出し規約を用いてネイティブ呼び出しスタブを呼び出す。次に、呼び出しスタブのネイティブ・コード２８は、さらなるパラメータ変換を実行し、基礎となるネイティブ関数を呼び出す。

【００４４】

図４は、ネイティブ・コード呼び出しスタブ１１３に基づいてネイティブ結合を用いるトランスレータ１２０の例示的な実施形態における関数呼び出しの動作フローを示すブロック図を示す。この例での対象プログラム１０６は、対象実行ファイル１０７および多くの対象ライブラリを含む。対象ライブラリは、独占的ライブラリ１０９およびシステム・ライブラリ１１１を含む。トランスレータ１２０は、対象システム・ライブラリ関数１１１に対する呼び出しを、ネイティブ・コード呼び出しスタブ１１３に対する呼び出しに置換する。ネイティブ・コード呼び出しスタブ１１３を呼び出す目的コード２１は、パラメータ変換を実行する。ネイティブ・コード呼び出しスタブ１１３は、さらなるパラメータ変換およびパラメータ・マッピングを実行する。次に、ネイティブ・コード呼び出しスタブ１１３は、ネイティブ・システム・ライブラリ関数１１７を呼び出す。

【００４５】

ネイティブ・コード・スタブ：統一されたインターフェース（Uniform Interface）

一実施形態において、ネイティブ・コード・パラメータ変換５０９は、統一された呼び出しスタブ関数インターフェースを規定することによって容易にされる。統一された呼び

出しスタブ関数インターフェースは、すべてのネイティブ呼び出しスタブに対する固定された関数署名と、対応するデータ形式とを規定し、それは、トランスレータ１２０が、基礎となるネイティブ関数の関数署名（プロトタイプ）を何ら知ることなしに、呼び出しスタブを呼び出すことを許す。これは、呼び出しスタブが、高レベルのプログラミング言語（たとえばＣまたはＣ＋＋）において実装されることを許し、それは、ネイティブ結合メカニズムが、トランスレータ１９のエンド・ユーザにとって、より利用しやすいものとする。

【００４６】

この実施形態において、呼び出しスタブ関数は、実行可能なトランスレータにリンクされる実行可能なネイティブ・コードとしてコンパイルされる。実行中、トランスレータ１２０は、ネイティブ呼び出し規約を用いて、統一されたインターフェースを通して呼び出しスタブを呼び出す。呼び出しスタブ・インターフェースは統一されているため、呼び出しスタブを呼び出す目的コードのシーケンスは、すべてのネイティブ呼び出しに対して同じである。

【００４７】

たとえば、一実施形態において、統一された呼び出しスタブ・インターフェースは、Ｃ関数であって、正確に２つのパラメータ（呼び出しサイトの対象アドレスと、すべての対象レジスタ値を含む統一されたデータ構造に対する参照）を取り、および１つの値（トランスレータが実行すべき次の対象命令の対象アドレス）を戻すものである。呼び出しスタブに受け渡される統一されたデータ構造は常に、すべての対象レジスタの現在の値（さもなくば、対象コンテキストと呼ばれる）を含む。

【００４８】

統一された呼び出しスタブ・インターフェースに基づくネイティブ結合メカニズムでは、ネイティブ結合は、複数のコンポーネント、すなわち、（ｉ）すべての対象レジスタ値を修正する特別なＩＲノード形式、（ｉｉ）ネイティブ呼び出し規約に基づいて、すべての対象レジスタを統一されたコンテキスト構造に整列し、且つ、呼び出しスタブを呼び出す目的コード２１、（ｉｉｉ）特定の対象レジスタ値を関数パラメータに整列し、且つ、ネイティブ関数を呼び出すネイティブ呼び出しスタブ、に分割される。

【００４９】

トランスレーション中に、ネイティブ呼び出しサイトが、ネイティブ呼び出しＩＲノードにトランスレートされる。ネイティブ呼び出しＩＲノードは、すべての対象レジスタのＩＲに対する依存関係参照を含む。目的コード生成フェーズにおいて、対象レジスタ値に対応する目的コードが、ネイティブ呼び出しの前に生成されることを、ネイティブ呼び出しＩＲノードのこれらのＩＲ依存関係が保証する。遅延評価などのトランスレータ最適化は、これらの値が必要とされるまで対象レジスタ値の計算を延期しても良い。ネイティブ呼び出しＩＲ依存関係は、コード生成フェーズに、ネイティブ呼び出しスタブがすべての対象レジスタ値を「必要とする」ことを通知する。したがって、トランスレータ１９は、ネイティブ呼び出しスタブ１１３を呼び出す目的コード２１を生成する前に、すべての対象レジスタ値を修正する目的コード２１を生成する。同様に、ネイティブ呼び出しＩＲノードは、コードを生成するためにメモリ参照として処理される。従って、対象コード１７における関数呼び出しの前にあるすべてのロードおよび記憶が、目的コード２１における関数呼び出しの前に（生成されて）実行される。同様に、対象コード１７において関数呼び出しの後に行なわれるすべてのロードおよび記憶は、ネイティブ呼び出しの後まで延期される。

【００５０】

必要に応じて、トランスレータ１９は、ネイティブ呼び出しＩＲを保持するための特別な抽象レジスタを含む。トランスレーションのＩＲ生成フェーズでは、抽象レジスタが、ＩＲツリーのためのプレースホルダ（すなわちＩＲツリーのルート・ノード）として機能する。ＩＲツリーは、抽象レジスタにリンクされなければならない。そうでないと、ＩＲツリーは目的コードとして放出されない。他のトランスレータでは、ネイティブ呼び出し

ＩＲノードは、既存の抽象レジスタ、たとえば（現在のブロックの）後継アドレスに対する抽象レジスタに付されうる。

【００５１】

ネイティブ呼び出しスタブ１１３を呼び出す目的コード２１は、対象レジスタ値を修正し、次に、その値を対象コンテキスト構造に記録する。本実施形態において、ネイティブ・コード呼び出しスタブ１１３はネイティブ・コード２８によって実装されるため、対象コンテキストは、ネイティブ・アーキテクチャに整合する表現で構築されなければならない。したがって、目的コード２１は、必要に応じてパラメータ変換５０３を行なって、対象レジスタ値を目的コードからネイティブ・コードに変換する。複数の値を他のアーキテクチャと整合するデータ構造に変換するプロセスは、しばしばマーシャリング（marshalling）と呼ばれる。

【００５２】

目的コード２１は、すべての対象レジスタ値のネイティブ表現を含む対象コンテキストを構築する。次に目的コード２１は、ネイティブ・コード呼び出しスタブ１１３を呼び出し、呼び出しスタブ１１３に対象コンテキストをパラメータとして受け渡す。呼び出しスタブ１１３は、基礎となるネイティブ関数を呼び出し、対象コンテキストから必要とされる特定の対象レジスタを引き出して、その対象レジスタをネイティブ関数に適切なパラメータとして受け渡す。こうして呼び出しスタブ１１３は、ネイティブ関数のプロトタイプをエンコードし、且つ、特定の対象レジスタの、対応するネイティブ関数パラメータへのマッピングを定義する。

【００５３】

場合によっては、ネイティブ関数インターフェースがその対象コード同等物と実質的に異なっており、従って対象データがネイティブ関数パラメータとしての使用に適するように、（データ表現に対する変換および呼び出し規約を超えて）さらなる計算が対象データに対して実行されなければならない。そのような場合には、呼び出しスタブ１１３は、対象レジスタ値に対してさらなるパラメータ変換を実行しうる。たとえば、ネイティブ関数は、その対象コード同等物とは異なる単位での特定のパラメータを予想しても良い。この場合、呼び出しスタブ１１３は、ネイティブ関数を呼び出す前に、適切な対象レジスタ値に対して定数変換を実行して、そのパラメータに対する単位形式の差に対処する。

【００５４】

統一されたネイティブ呼び出しスタブ・インターフェースを用いる実施形態において、目的コード２１は、すべての対象レジスタを、目的コード表現からネイティブ・コード表現に無差別に変換する。次に、ネイティブ呼び出しスタブ１１３は、ネイティブ関数のプロトタイプがパラメータとして必要とする特定の（変換済み）対象レジスタを取り出す。ネイティブ呼び出しスタブ１１３は、呼び出し中の関数の対象バージョンとネイティブ・バージョンとの間の差を反映するために、さらなるパラメータ変換を行なっても良い。目的コード２１は、目的コード２１とネイティブ・コード２８との間の表現差に対する調整を行ない、一方、呼び出しスタブ１１３は特定の、基礎となるネイティブ関数の署名に対処する。

【００５５】

本実施形態において、ネイティブ結合用結合点の記述は、結合するための対象関数および、対応するネイティブ呼び出しスタブ関数を識別し、一方、結合先の基礎となるネイティブ関数の身元は、呼び出しスタブの実装において暗黙的である（すなわち、ハード・コードされている）。

【００５６】

ネイティブ・コード・スタブ：コンパイル済みスクリプト

他の実施形態において、ネイティブ・コード・パラメータ変換が、特別目的のネイティブ結合プログラミング言語（以後、「スクリプト言語」と呼ぶ）を用いて実装される。実行前または実行中に、トランスレータ１９は、呼び出しスタブ関数実装スクリプト言語を構文解析し、それを、ネイティブ実行可能モジュールにコンパイルする。次に、前述した

通り、呼び出しスタブ・モジュールがトランスレータ実行ファイルとリンクされ、（ネイティブ）呼び出しスタブ機能がネイティブ呼び出し規約を用いて呼び出される。

【００５７】

ネイティブ結合スクリプトは、呼び出しスタブの実行可能な表現に、コンパイルされるか又はインタープリットされる。一実施形態において、結合点の記述は、トランスレータの実行前に、個々のツールによって実行可能なネイティブ・コードにインタープリットされる。他の実施形態において、トランスレータ自体が、実行時に結合点の記述を、実行可能なネイティブ・コードに又はトランスレータＩＲ（後で目的コードとして生成される）に、インタープリットまたはコンパイルする。

【００５８】

いくつかの実施形態において、スクリプト言語は、トランスレータに固有である特別の目的言語である。スクリプト言語は、広範囲の可能なパラメータ変換動作を説明するためにプリミティブ（プログラミング言語構築ブロック）を含む。ありうるパラメータ変換動作は、（ｉ）データ形式の記述、（ｉｉ）これらのデータ形式の目的コードとネイティブ表現との間の変換、（ｉｉｉ）ネイティブ関数の識別、（ｉｖ）特定の対象レジスタの特定の関数パラメータへのマッピング、（ｖ）対象プログラムのメモリ空間に対するメモリ・アクセス、及び（ｖｉ）基本的な数学演算、を含む。代替的な実施形態において、スクリプト言語はさらに、（ｖｉ）基本的な論理動作、及び（ｖｉｉ）複数のネイティブ結合スクリプトに渡る一時的な値のグローバルな記憶を含む。これらのプリミティブ（すなわちトランスレータによって又は特別なツールによって、スクリプトから生成されるネイティブ・コード）を実装することは、目的アーキテクチャの表現および呼び出し規約と整合されなければならない。

【００５９】

本実施形態において、ネイティブ結合用結合点の記述は、結合すべき対象関数および対応するネイティブ呼び出しスタブ関数を識別し、一方、結合先の基礎となるネイティブ関数の身元は、呼び出しスタブのスクリプト言語の実装においてハード・コード化される。

【００６０】

目的コード・スタブ

他の実施形態において、ネイティブ結合に対するパラメータ変換は、目的コードにおいて全体的に実施される。本実施形態において、トランスレータ１９は、デコード時にネイティブ結合呼び出しを検出し、且つ、ＩＲツリー（それは最終的に目的コードとして生成される）としてパラメータ変換をエンコードする。パラメータ変換とネイティブ関数プロトタイプの詳細とをＩＲにおいて表わすことによって、呼び出しスタブ・コードは、対象コードに統合されることになる（呼び出しスタブのＩＲは、対象コードのＩＲと区別できない）。これは、トランスレータが、パラメータ変換コードに対して最適化（たとえば、グループ・ブロック）を施すことを許す。対照的に、ネイティブ・コードにおいて実行されるパラメータ変換、たとえばネイティブ・コード呼び出しスタブにおいて行なわれる対象レジスタの関数パラメータへのマッピング（前述）は、トランスレータの外であり、最適化されることができない。

【００６１】

図５は、目的コード呼び出しスタブ１１５に基づいてネイティブ結合を用いるトランスレータ１２２の例示的な実施形態における関数呼び出しの動作フローを示すブロック図を示す。対象プログラム１０６は、対象実行ファイル１０７および多くの対象ライブラリからなる。対象ライブラリは、独占的ライブラリ１０９およびシステム・ライブラリ１１１を含みうる。トランスレータ１２２は、対象システム・ライブラリ関数１１１に対する呼び出しを、目的コード呼び出しスタブ１１５に置換する。目的コード呼び出しスタブ１１５は、パラメータ変換およびパラメータ・マッピングを実行し、次にネイティブ・システム・ライブラリ１１７内の関数を呼び出す。

【００６２】

トランスレータ１２２が目的コード呼び出しスタブ１１５を用いる実施形態において、

目的コード２１が結合済みネイティブ関数を直接的に呼び出し、すべてのパラメータ変換およびパラメータ・マッピングが目的コード２１によって実行される。これらの実施形態において、目的コード２１が、特定のネイティブ関数の表現変換および署名（すなわち、特定の対象レジスタの対応するネイティブ関数パラメータへのマッピング）の両方に対処する。目的コード２１がパラメータ・マッピングを実行するためには、トランスレータ１２２は、トランスレーション中に、ネイティブ関数プロトタイプおよび対象レジスタ対パラメータ・マッピングを知っていなければならない。場合によっては、目的コード呼び出しスタブ１１５を、このスタブ１１５を呼び出す目的コード２１（「呼び出し元目的コード」）から、別個のブロックにおいてトランスレートする。他の場合には、目的コード呼び出しスタブ１１５は、このスタブ１１５を呼び出す目的コード２１と同じブロックにおいてトランスレートされ、それは、呼び出しスタブ・コードが、呼び出し元目的コードと統合し且つこの呼び出し元目的コードで最適化すること（「初期結合」メカニズムと呼ばれる）を許す。対象プログラムが多くの呼び出しサイトから特定の対象関数を呼び出す場合に、目的コード呼び出しスタブ１１５をあらゆる呼び出しサイトにおいてインラインすることは不都合である。なぜならば、過剰なメモリが、呼び出しスタブ・コードの結果として生じる重複によって消費されるからである。そのような場合、トランスレータ１２２は、呼び出しスタブをあらゆる呼び出しサイトにおいてインラインするのではなくて、各トランスレート済み呼び出しサイトが呼び出す別個のブロックとして、目的コード呼び出しスタブ１１５を保持する。ネイティブ結合の最適化の１つは、非常に頻繁に実行されるブロックでのみ、初期結合を用いる（すなわち、呼び出しスタブをインラインする）ことである。

【００６３】

目的コード・スタブ：スキゾ（Schizo）呼び出し

他の実施形態において、トランスレータ１９は、スキゾ呼び出し、すなわち「Ｓ－呼び出し」と呼ばれるメカニズムを用いる。このメカニズムでは、対象コード単位がコンパイルされるときに対象コードに挿入されるトランスレータ固有のネイティブ結合命令（Ｓ呼び出しコマンドと呼ばれる）を含むように対象命令セットを拡張することによって、目的コード・パラメータ変換が容易にされる。トランスレータ１９が対象コードをデコードするとき、トランスレータ１９は、これらのＳ－呼び出しコマンドを検出し且つインタープリットし、そして適切なＩＲ（または目的コード）をプラントしてパラメータ変換を実行する。

【００６４】

Ｓ－呼び出しメカニズムは、対象プログラムの開発者からのサポートを必要とする。対象プログラムがコンパイルされるとき、Ｓ－呼び出しコマンドは、コンパイル済み対象プログラムにおいて対象命令としてエンコードされる。ネイティブ関数に対する呼び出しを記述する呼び出しスタブは、パラメータ変換を支援する正規の合法的な対象命令を含みうるＳ－呼び出しコマンドを含むことだけに制約されない。組み合わせて、Ｓ－呼び出しコマンドおよび正規の対象命令は、パラメータ変換に必要とされるすべての情報および動作をエンコードする。

【００６５】

一実施形態において、Ｓ－呼び出しコマンドは、複数のサブ・コンポーネントから構築される可変長命令にエンコードされる。従って、１つのＳ－呼び出しコマンド命令が、正規の対象コード命令よりも大きく、且つ、複数の正規の対象命令のサイズでさえあり得る。Ｓ－呼び出しコマンドはサブ・コンポーネントで始まり、当該サブ・コンポーネントは、命令をＳ－呼び出しコマンドとして識別し、且つ対象アーキテクチャ上での違法命令（スキゾ・エスケープと呼ばれる）としてインタープリットされることが知られている。スキゾ・エスケープ・サブ・コンポーネントは任意的に、Ｓ－呼び出しコマンド内に含まれるコマンドの形式をさらに識別しうる。Ｓ－呼び出しコマンドは好ましくは、５つの形式のコマンド・クラスに分割される。すなわち（１）マーカ、（２）パラメータ、（３）呼び出し、（４）コピー、および（５）ヌリファイである。Ｓ－呼び出しコマンドのクラス

は、先頭のスキゾ・エスケープ・サブ・コンポーネントの未使用のビットにエンコードされる。スキゾ・エスケープの後に、クラス固有のオプションおよび／または引数の組み合わせが続く。これらはそれぞれ、予め定義されたオプコード（ｏｐｃｏｄｅ）およびフォーマットを有し、またそれぞれ、Ｓ－呼び出しコマンド命令において１つまたは複数のワード（すなわち４バイト単位）としてエンコードされる。Ｓ－呼び出しコマンド命令は、初期スキゾ・エスケープの再現であるサブ・コンポーネントで終了する。以下、前述した５つの形式のＳ－呼び出しコマンドについて、より詳しく説明する。しかし当然のことながら、他の形式のトランスレータ固有のネイティブ結合命令を、すでに列挙した５つの形式以外のＳ－呼び出しコマンドとして用いても良いことは、本発明の発明者が完全に意図するところである。

【００６６】

Ｓ－呼び出しの「マーカ」コマンドは、アプリケーション・バイナリ・インターフェース（ＡＢＩ）固有の最適化を可能にするために用いられる任意的なマーカである。Ｓ－呼び出しマーカー・コマンドは、トランスレータ固有のコード（たとえばネイティブ関数に対する呼び出しスタブ）を含む対象コード１７の範囲（開始および終了アドレス）を示す。この範囲は、対象関数の全範囲と一致していても良いし、一致していなくても良い。Ｓ－呼び出しマーカー・コマンドは、２つの形式からなる。すなわち、開始および終了である。Ｓ－呼び出しマーカー・コマンドは、１つの文字列引数を有していて、マーク済みの点に名前（たとえば、結合されている対象関数の名前）を割り当てる。

【００６７】

Ｓ－呼び出しの「パラメータ」コマンドは、ネイティブ関数パラメータとして用いられる値を識別し、且つ、その値に対する適切なパラメータ変換をエンコードする。一実施形態において、各Ｓ－呼び出しパラメータ・コマンドは、対応する値をスタック上にプッシュすることによって、関数呼び出しに対する「次の」パラメータを定義し、それ故にＳ－呼び出しパラメータ・コマンドが、関数プロトタイプに対応する順序に設けられていることを必要とする。

【００６８】

Ｓ－呼び出しの「呼び出し」コマンドは、ネイティブ関数に対する実際の関数呼び出しをエンコードする。Ｓ－呼び出しの呼び出しコマンド引数は、関数の戻り値を記憶する箇所と、呼び出されている関数の対象装置状態における名前、絶対アドレス、またはアドレス箇所のいずれかと、を含む。

【００６９】

Ｓ－呼び出しの「コピー」コマンドは、対象レジスタまたは対象メモリ箇所に対する又はそれらからの値をコピーする動作と、目的コードとネイティブ表現との間の変換を実行する動作とをエンコードする。

【００７０】

Ｓ－呼び出しの「ヌリファイ」コマンドは、このコマンドが従う命令を無効にする。このコマンドを、無条件のブランチとともに用いられて（これについては後述する）、ネイティブで実行されるときに対象プログラムの適切な実行を許し、さらに、トランスレータがコードのトランスレータ固有部分を識別し且つインタープリットすることを許す。

【００７１】

トランスレーションの間、トランスレータ１９は、スキゾ・エスケープを検出し、且つ、Ｓ－呼び出しコマンドを、対応するパラメータ変換およびネイティブ関数呼び出し動作のＩＲ表現にデコードする。トランスレータ１９は、パラメータ変換ＩＲを現在のブロックのＩＲフォレストに統合し、それは、その後目的コードとして生成される。パラメータ変換動作をＩＲにエンコードすることは、パラメータ変換コードが、対象コードをエンコードするＩＲに統合され且つ当該IRで最適化されることを許す。

【００７２】

Ｓ－呼び出しコマンドは、対象プログラムにおいて、先頭の命令のオプコード（ｏｐｃｏｄｅ）、トランスレータ１９だけが分かっているスキゾ・エスケープを用いてエンコー

ドされる。なぜならば、Ｓ－呼び出しコマンドは、トランスレータ固有のネイティブ結合命令であるからである。対象プロセッサは、Ｓ－呼び出しコマンドを、違法の対象命令としてインタープリットする。したがって、Ｓ－呼び出しコマンドの実行は、対象プログラムがネイティブで（すなわち、対象アーキテクチャ上で）実行されているときには、回避されなければならない。対象プログラム・ソフトウェアの開発者は、Ｓ－呼び出し拡張対象プログラムがネイティブで実行することを許すために複数の方法を用いる。この方法は、（ａ）条件付きの実行、および（ｂ）ブランチ・ルック・アヘッド構文解析を含む。条件付きの実行は、条件付きの対象コードからなる。この条件付きの対象コードは、対象プログラムがネイティブで実行されているのか又はトランスレート済みプログラムとして実行されているのかを、実行時の環境に基づいてチェックし、且つ、ネイティブで実行されている場合にはＳ－呼び出しコマンドを省略する（スキップする）。ブランチ・ルック・アヘッド構文解析は、無条件のブランチ命令からなる。無条件のブランチ命令は、対象コード内でプラントされて、すべてのトランスレータ固有の命令（たとえば（これに限定されないが）Ｓ－呼び出しコマンド）を省略する。ネイティブで実行するときには、無条件のブランチが実行され、その結果、トランスレータ固有のコードがスキップされる。トランスレート済みプログラムとして実行されるときは、トランスレータは、Ｓ－呼び出しヌリファイ・コマンドが後に続く任意の無条件のブランチ命令を無視する（すなわち、無条件のブランチは、デコーディングにおいてトランスレータ固有のコードを識別するためにトランスレータによって用いられる対象命令パターンの一部である）。

【００７３】

目的コード・スタブ：外部のスキゾ・スタブ

他の実施形態において、スキゾ呼び出しコマンドと普通の対象命令とを含む呼び出しスタブは、対象コードの別個にコンパイルされる単位内に配置される。特別目的のネイティブ結合用結合点の記述言語（前述したように）は、対象コード箇所を結合点として特定するためにトランスレータ１９によって使用される。トランスレータ１９がそのような結合点に到達すると、制御フローは変えられて、外部のスキゾ・スタブを代わりに実行する。この点から以降では、トランスレータの作用は、スキゾ呼び出しに対して説明した作用と同じである。外部のスキゾ・スタブは、Ｓ－呼び出しコマンドを対象コードに直接挿入できないとき（たとえば対象ライブラリ／アプリケーションに対するソース・コードが利用できないとき）に、Ｓ呼び出しコマンドを用いることを許す。

【００７４】

目的コード・スタブ：インタープリット済みスクリプト

代替的な実施形態において、目的コード・パラメータ変換は、前述された通り、特別目的のネイティブ結合実装スクリプト言語（「スクリプト言語」）によって容易にされる。この実施形態において、実行時に、トランスレータが、ネイティブ結合スクリプトを、パラメータ変換のＩＲ表現にインタープリットする。トランスレータは、パラメータ変換ＩＲを現在のブロックのＩＲフォレストに統合し、それは、その後に、最適化されて、目的コードとして生成される。そのようなトランスレータは、スクリプト言語を構文解析し且つデコードすることができるフロント・エンド・コンポーネントを含まなければならない。

【００７５】

本実施形態において、ネイティブ結合用結合点の記述は、結合すべき対象関数および対応するネイティブ呼び出しスタブ関数を特定し、一方、結合先の基礎となるネイティブ関数の身元は、呼び出しスタブの実装においてハード・コードされる。

【００７６】

目的コード・スタブ：統一されたインターフェース

代替的な実施形態において、目的コード・パラメータ変換は、前述された通り、統一された呼び出しスタブ関数インターフェースを定義することによって容易にされる。統一された呼び出しスタブ関数インターフェースは、すべてのネイティブ呼び出しスタブに対する固定された関数署名と、対応するデータ形式とを定義し、それは、トランスレータが、

基礎となるネイティブ関数の関数署名（プロトタイプ）を何ら知ることなしに、呼び出しスタブをネイティブ関数として呼び出すことを許す。これは、呼び出しスタブが高レベル・プログラミング言語（たとえばＣまたはＣ＋＋）において実装されることを許し、それは、ネイティブ結合メカニズムがトランスレータ１９のエンド・ユーザにとって、より利用しやすいものとする。

【００７７】

前述のネイティブ・コードの統一された呼び出しスタブ・インターフェースとは対照的に、いくつかの実施形態において、トランスレータは、実行時に呼び出しスタブ実装を構文解析し、且つ、呼び出しスタブ実装を、パラメータ変換のＩＲ表現にインタープリットする。言い換えれば、トランスレータは、呼び出しスタブの実装をトランスレータＩＲにコンパイルする。トランスレータは、パラメータ変換ＩＲを現在のブロックのＩＲフォレストに統合し、それは、その後最適化されて、目的コードとして生成される。そのようなトランスレータは、コンパイラと同様に、高レベルのプログラミング言語を構文解析し且つデコードすることができるフロント・エンド・コンポーネントを含まなければならない。

【００７８】

本実施形態において、ネイティブ結合用結合点の記述は、結合すべき対象関数および対応するネイティブ呼び出しスタブ関数を識別し、一方、結合すべき基礎となるネイティブ関数の身元は、呼び出しスタブの実装においてハード・コードされる。

【００７９】

いくつかの好ましい実施形態について図示して説明してきたが、当業者であれば理解するように、添付請求項において規定されるように、本発明の範囲から逸脱することなく、種々の変形および変更を行なっても良い。

【００８０】

本出願に関連して本明細書と同時にまたは以前に出願され、本明細書とともに公衆の便覧に公開されているすべての論文および文献に注意を払うものであり、そのような論文および文献はすべて、その内容が本明細書において参照により取り入れられている。

【００８１】

本明細書（任意の添付請求項、要約、および図面を含む）で開示されるすべての特徴、および／または同様に開示される任意の方法またはプロセスのすべてのステップは、そのような特徴および／またはステップの少なくとも一部が互いに相容れない組み合わせを除いて、任意の組み合わせで組み合わせても良い。

【００８２】

本明細書（任意の添付請求項、要約、および図面を含む）で開示される各特徴を、特にことわらない限り、同じ、同等、または類似の目的を満たす代替的な特徴と取り替えても良い。したがって、特にことわらない限り、開示される各特徴は、同等または類似の特徴の包括的な組の１つの例に過ぎない。

【００８３】

本発明は、前述の実施形態の詳細に限定されない。本発明は、本明細書（任意の添付請求項、要約、および図面を含む）で開示される特徴の任意の新しい１つまたは任意の新しい組み合わせに及ぶか、または同様に開示される任意の方法またはプロセスのステップの任意の新しい１つまたは任意の新しい組み合わせに及ぶ。

【図面の簡単な説明】

【００８４】

【図１】本発明の実施形態が応用例を見出す装置のブロック図である。

【図２】本発明の例示的な実施形態によるネイティブ結合プロセスを例示する概略図である。

【図３】本発明の例示的な実施形態によるネイティブ結合プロセスを例示する概略図である。

【図４】本発明の例示的な実施形態によるネイティブ結合プロセスを例示する概略図であ

る。
【図５】本発明の例示的な実施形態によるネイティブ結合プロセスを例示する概略図である。
【図６】本発明の例示的な実施形態によるネイティブ関数呼び出しを例示するフロー図である。