

(19)日本国特許庁(JP)

(12)特許公報(B2)

(11)特許番号

特許第7202293号

(P7202293)

(45)発行日 令和5年1月11日(2023.1.11)

(24)登録日 令和4年12月27日(2022.12.27)

(51)国際特許分類

G 0 6 F 11/36 (2006.01)

F I

G 0 6 F 11/36 1 3 6

請求項の数 16 (全97頁)

(21)出願番号 特願2019-520612(P2019-520612)
(86)(22)出願日 平成29年10月10日(2017.10.10)
(65)公表番号 特表2019-537782(P2019-537782 A)
(43)公表日 令和1年12月26日(2019.12.26)
(86)国際出願番号 PCT/US2017/055986
(87)国際公開番号 WO2018/071450
(87)国際公開日 平成30年4月19日(2018.4.19)
審査請求日 令和2年10月9日(2020.10.9)
(31)優先権主張番号 62/406,829
(32)優先日 平成28年10月11日(2016.10.11)
(33)優先権主張国・地域又は機関 米国(US)
前置審査

(73)特許権者 519132791
グリーン ヒルズ ソフトウェア, エルエルシー
GREEN HILLS SOFTWARE, LLC
アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェストソーラー ストリート 30
30 West Sola Street
Santa Barbara, CA 93101 (US)
(74)代理人 100109634
弁理士 舩谷 威志
(74)代理人 100129263
弁理士 中尾 洋之

最終頁に続く

(54)【発明の名称】 垂直統合インストルメント化およびトレース再構成のためのシステム、方法およびデバイス

(57)【特許請求の範囲】

【請求項1】

複数の時点における1つ以上のプログラムの状態の少なくとも一部を決定する方法であって、

ベースラインイメージとして1つ以上のプログラムのベースライン状態を得て、1つ以上のプリイメージを最も近い時点でロギングされたものから最も遠い時点でロギングされたものへと順次、ベースラインイメージに適用し、それにより、ベースラインイメージがロギングされた時間よりも前の時間で1つ以上のプログラムのメモリの状態を再構成すること、

1つ以上のレジスタの状態を含む1つ以上のレジスタ状態スナップショットを取得すること、および、

最新から最古まで1つ以上のプリイメージを検索および解釈し、さらに1つ以上のレジスタ状態スナップショットを使用して所与の時間における1つ以上のレジスタの状態を再構成し、それにより、1つ以上のプログラムの最も近い時点での実行履歴の解析を可能とすることを含み、

1つ以上のプログラムの状態で発生した少なくとも1つの変更は、変更した状態の少なくとも一部をロギングすることによりロギングされた変更として記録されており、

変更した状態の前記一部は、ロギングされた変更の各々の発生前の状態の少なくとも前記一部の表現をプリイメージとしてロギングすることによりロギングされており、

前記1つ以上のレジスタに保存された値への変更が発生したときに、1つ以上のレジス

10

20

タ状態スナップショットの少なくとも一部がロギングされていない、方法。

【請求項 2】

ロギングされた変更の各々は、ロギングされた変更が発生する時間に記録されず、ロギングされた変更を前記 1 つ以上のレジスタ状態スナップショットを介して記録する、請求項 1 に記載の方法。

【請求項 3】

ベースラインイメージは、1 つ以上のプログラムのメモリ状態を含む、請求項 1 または 2 に記載の方法。

【請求項 4】

ベースラインイメージを取得することは、再構成に必要なときに 1 つずつベースラインイメージの複数のセクションを得ることを含む、請求項 3 に記載の方法。

10

【請求項 5】

少なくとも 1 つのセクションがダウンロードされた後にベースラインイメージを修正し、少なくとも修正は異なる時間に得られたセクションのロギングされたデータ間に 1 つ以上の不整合を引き起こし、方法はさらに、

1 つ以上の不整合を解消するためにベースラインイメージの修正と併せて記録された 1 つ以上のプリイメージを適用することを含む、請求項 4 に記載の方法。

【請求項 6】

ベースラインイメージは、システムメモリの少なくとも一部の以前に保存されたイメージを含む、請求項 1 ～ 5 のいずれかに記載の方法。

20

【請求項 7】

前記少なくとも 1 つの変更が発生する前に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行う、請求項 1 ～ 6 のいずれかに記載の方法。

【請求項 8】

前記少なくとも 1 つの変更が発生した後に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行う、請求項 1 ～ 6 のいずれかに記載の方法。

【請求項 9】

前記少なくとも 1 つの変更の発生中に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行う、請求項 1 ～ 6 のいずれかに記載の方法。

【請求項 10】

30

1 つ以上のプロセッサによって実行されるとき、前記 1 つ以上のプロセッサに、請求項 1 から 9 のいずれかに記載の方法を実行させる命令を含む非一時的コンピュータ記憶媒体。

【請求項 11】

複数の時点における 1 つ以上のプログラムの状態を決定するために 1 つ以上のコンピュータプログラムをインストール化する方法であって、

変更した状態の少なくとも一部をロギングすることによって、1 つ以上のプログラムの状態で発生する少なくとも 1 つの変更を、ロギングされた変更として記録するように構成された 1 つ以上の実行可能命令を挿入すること、および、

ロギングされた変更の各々の発生前の状態の少なくとも前記一部の表現をプリイメージとしてロギングすることにより変更した状態の前記一部をロギングするように構成された 1 つ以上の実行可能命令を挿入すること

40

を含み、

記録およびロギングにより、デバッガが、

1 つ以上のプログラムのベースライン状態に対応するベースラインイメージに、1 つ以上のプリイメージを最も近い時点でロギングされたものから最も遠い時点でロギングされたものへと順次、適用し、それによりベースラインイメージがロギングされた時間よりも前の時間で 1 つ以上のプログラムのメモリの状態を再構成することができ、

1 つ以上のレジスタの状態を含む 1 つ以上のレジスタ状態スナップショットを取得し、前記 1 つ以上のレジスタに保存された値への変更が発生したときに、1 つ以上のレジスタ状態スナップショットの少なくとも一部がロギングされておらず、および

50

最新から最古まで1つ以上のプリイメージを検索および解釈し、さらに1つ以上のレジスタ状態スナップショットを使用して所与の時間における1つ以上のレジスタの状態を再構成し、それにより、1つ以上のプログラムの最も近い時点での実行履歴の解析を可能とする、方法。

【請求項12】

ロギングされた変更の各々は、ロギングされた変更が発生する時間に記録されず、ロギングされた変更を前記1つ以上のレジスタ状態スナップショットを介して記録する、請求項11に記載の方法。

【請求項13】

前記少なくとも1つの変更が発生する前に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行う、請求項11または12に記載の方法。

【請求項14】

前記少なくとも1つの変更が発生した後に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行う、請求項11または12に記載の方法。

【請求項15】

前記少なくとも1つの変更の発生中に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行う、請求項11または12に記載の方法。

【請求項16】

1つ以上のプロセッサによって実行されるとき、前記1つ以上のプロセッサに、請求項11から15のいずれかに記載の方法を実行させる命令を含む非一時的コンピュータ記憶媒体。

【発明の詳細な説明】

【技術分野】

【0001】

【関連出願への相互参照】

本願は、その全体が参考として援用される2016年10月11日に出願された米国仮出願第62/406,829号の利益を主張する。

【0002】

開示の実施形態は概して、インストルメント化システムに関し、より詳細には、動的再構成およびデバッグのためのシステム、デバイスおよび方法に関する。

【背景技術】

【0003】

新たな高度技術の開発に伴って、これらの新たな技術革新を実行するために必要なソフトウェアはますます重要かつ複雑になる。ソフトウェアがますます複雑になるにつれて、このようなソフトウェアのデバッグもますます困難になる。例えば、乗用車およびトラックは、車両における様々なシステム構成要素を実行するために複雑なソフトウェアをますます必要とする。概して、典型的な乗用車は今日、何百人もの人々によって書かれた数千万行ものコンピュータコードが必要である。コードの行数が増加して、より多くのプログラムが関与するにつれて、ソフトウェアコード内のバグの数も増加する。また、ソフトウェアバグは、検出および／またはデバッグするのがますます複雑となっている。本明細書で使用される場合、用語「バグ」は概して、コンピュータプログラムにおけるエラーを指す。例えば、コンピュータプログラムにおける一般的なバグとは、コンピュータプログラムがその全機能を完了する前にクラッシュすることである。

【0004】

概してプログラムは、コンピュータプログラム内のエラー発見を支援するのに役立つソフトウェアデバッグを採用している。しかし、そのようなデバッグは、解析中のコンピュータプログラムの実行から生成されたトレースデータを解析するためにデバッグにとって長い時間期間を典型的には必要とする。本明細書で使用される場合、用語「データトレース」は概して、メモリに保存されたアドレスおよび／もしくは値、ならびに／または、各マシン命令によってアクセスおよび／もしくは修正されるメモリ位置の値とともに、時間

期間中にプログラムによって実行されるマシン命令のシーケンスの記録を指す。

【発明の概要】

【0005】

この概要の目的として、本開示の特定の態様、利点および新規な特徴が本明細書に記載されている。必ずしもすべてのこのような利点が任意の特定の実施形態によって達成され得るわけではないことを理解すべきである。したがって例えば、当業者は、必ずしも本明細書に教示または示唆され得るような他の利点を達成することなく、本明細書に教示されるような1つの利点または利点のグループを達成する方法で本開示を具現化または実行し得ることを認識するであろう。

【0006】

いくつかの実施形態では、複数の時点における1つ以上のプログラムの状態を決定するために1つ以上のコンピュータプログラムをインストルメント化する方法は、変更した状態の少なくとも一部をロギングすることにより1つ以上のプログラムの状態で発生する複数の変更を、ロギングされた変更として記録すること、ロギングされた変更の各々の発生前の時間から少なくとも前記一部の表現をプリイメージとしてロギングすることにより変更した状態の前記一部をロギングすること、1つ以上のプログラムのベースライン状態をベースラインイメージとして得て、1つ以上のプリイメージを最も近い時点でロギングされたものから最も遠い時点でロギングされたものへと順次、ベースラインイメージに適用し、それによりベースラインイメージよりも前の時間で状態を再構成すること、および、最新から最古まで1つ以上のプリイメージを検索および解釈し、それにより、検索、ダウンロードおよび／またはデコードされる対象の実行履歴を表すロギングされたデータの以前の部分を待たずに1つ以上のプログラムの最も近い時点での実行履歴の解析を可能とすることを含む。

【0007】

先行する段落の方法は、1つ以上の以下の特徴を含むことができる。ロギングされた変更の各々は、ロギングされた変更が発生する時間に記録されなくともよく、ロギングされた変更を1つ以上のレジスタの状態を含む少なくとも1つのレジスタ状態スナップショットを介して記録することができる。少なくとも1つのレジスタ状態スナップショットを記録することにより、1つ以上のプログラムのランタイムパフォーマンスおよび／またはメモリフットプリントに悪影響を与えることを回避することができる。ベースラインイメージは、1つ以上のプログラムのメモリ状態を含むことができる。ベースラインイメージを取得することは、再構成に必要なときに1つずつベースラインイメージの複数のセクションを得て、それによりデバッグを開始するのに必要な時間をバウンディングすることを含むことができる。少なくとも1つのセクションがダウンロードされた後にベースラインイメージを修正し、それにより異なる時間に得られたセクション間に1つ以上の不整合を引き起こすことができ、方法は、1つ以上の不整合を解消するためにベースラインイメージの修正と併せて記録された1つ以上のプリイメージを適用することを含むことができる。ベースラインイメージは、システムメモリの少なくとも一部の以前に保存されたイメージを含むことができる。ベースラインイメージとしてシステムメモリの少なくとも一部の以前に保存されたイメージを使用すると、プログラマはデバッグを実行することができる。デバッグは、1つ以上のプログラムが実行を停止した後のデバッグを含むことができる。ロギングされた変更が発生する前に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。ロギングされた変更が発生した後に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。ロギングされた変更中に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。

【0008】

いくつかの実施形態では、複数の時点における1つ以上のプログラムの状態の少なくとも一部を決定する方法は、ベースラインイメージとして1つ以上のプログラムのベースライン状態を得て、1つ以上のプリイメージを最も近い時点でロギングされたものから最も

10

20

30

40

50

遠い時点でロギングされたものへと順次、ベースラインイメージに適用し、それにより、ベースラインイメージよりも前の時間での状態を再構成すること、および、最新から最古まで1つ以上のプリイメージを検索および解釈し、それにより、検索、ダウンロードおよび／またはデコードされる対象の実行履歴を表すロギングされたデータの以前の部分を待たずに1つ以上のプログラムの最も近い時点での実行履歴の解析を可能とすることを含む。1つ以上のプログラムの状態で発生した少なくとも1つの変更は、変更した状態の少なくとも一部をロギングすることによりロギングされた変更として記録されていてもよく、変更した状態の前記一部は、ロギングされた変更の各々の発生前の時間から少なくとも前記一部の表現をプリイメージとしてロギングすることによりロギングされていてもよい。

【0009】

先行するいずれかの段落の方法は、1つ以上の以下の特徴を含むことができる。ロギングされた変更の各々は、ロギングされた変更が発生する時間に記録されなくともよく、ロギングされた変更を1つ以上のレジスタの状態を含む少なくとも1つのレジスタ状態スナップショットを介して記録することができる。少なくとも1つのレジスタ状態スナップショットを記録することにより、1つ以上のプログラムのランタイムパフォーマンスおよび／またはメモリフットプリントに悪影響を与えることを回避することができる。ベースラインイメージは、1つ以上のプログラムのメモリ状態を含むことができる。ベースラインイメージを取得することは、再構成に必要なときに1つずつベースラインイメージの複数のセクションを得て、それによりデバッグを開始するのに必要な時間をバウンディングすることを含むことができる。少なくとも1つのセクションがダウンロードされた後にベースラインイメージを修正し、それにより異なる時間に得られたセクション間に1つ以上の不整合を引き起こすことができ、方法は、1つ以上の不整合を解消するためにベースラインイメージの修正と併せて記録された1つ以上のプリイメージを適用することを含むことができる。ベースラインイメージは、システムメモリの少なくとも一部の以前に保存されたイメージを含むことができる。ベースラインイメージとしてシステムメモリの少なくとも一部の以前に保存されたイメージを使用すると、プログラマはデバッグを実行することができる。デバッグは、1つ以上のプログラムが実行を停止した後のデバッグを含むことができる。ロギングされた変更が発生する前に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。ロギングされた変更が発生した後に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。ロギングされた変更中に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。

【0010】

いくつかの実施形態では、複数の時点における1つ以上のプログラムの状態を決定するために1つ以上のコンピュータプログラムをインストルメント化する方法は、変更した状態の少なくとも一部をロギングすることによって、1つ以上のプログラムの状態で発生する少なくとも1つの変更を、ロギングされた変更として記録するように構成された1つ以上の実行可能命令を挿入すること、および、ロギングされた変更の各々の発生前の時間から少なくとも前記一部の表現をプリイメージとしてロギングすることにより変更した状態の前記一部をロギングするように構成された1つ以上の実行可能命令を挿入することを含む。記録およびロギングにより、デバッガが、1つ以上のプログラムのベースライン状態に対応するベースラインイメージに、1つ以上のプリイメージを最も近い時点でロギングされたものから最も遠い時点でロギングされたものへと順次、適用し、それによりベースラインイメージよりも前の時間での状態を再構成することができ、最新から最古まで1つ以上のプリイメージを検索および解釈し、それにより、検索、ダウンロードおよび／またはデコードされる対象の実行履歴を表すロギングされたデータの以前の部分を待たずに1つ以上のプログラムの最も近い時点での実行履歴の解析を可能とする。

【0011】

先行するいずれかの段落の方法は、1つ以上の以下の特徴を含むことができる。ロギングされた変更の各々は、ロギングされた変更が発生する時間に記録されなくともよく、ロ

10

20

30

40

50

ギングされた変更を1つ以上のレジスタの状態を含む少なくとも1つのレジスタ状態スナップショットを介して記録することができる。少なくとも1つのレジスタ状態スナップショットを記録することにより、1つ以上のプログラムのランタイムパフォーマンスおよび／またはメモリフットプリントに悪影響を与えることを回避することができる。ベースラインイメージは、1つ以上のプログラムのメモリ状態を含むことができる。記録およびロギングにより、デバッグがさらに、再構成に必要なときに1つずつベースラインイメージの複数のセクションを得ることによりベースラインイメージを取得することができ、それによりデバッグを開始するのに必要な時間をバウンディングすることができる。少なくとも1つのセクションがダウンロードされた後にベースラインイメージを修正し、それにより異なる時間に得られたセクション間に1つ以上の不整合を引き起こすことができ、記録およびロギングにより、デバッグがさらに、1つ以上の不整合を解消するためにベースラインイメージの修正と併せて記録された1つ以上のプリイメージを適用することができる。ベースラインイメージは、システムメモリの少なくとも一部の以前に保存されたイメージを含むことができる。ベースラインイメージとしてシステムメモリの少なくとも一部の以前に保存されたイメージを使用すると、プログラマはデバッグを実行することができる。デバッグは、1つ以上のプログラムが実行を停止した後のデバッグを含むことができる。ロギングされた変更が発生する前に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。ロギングされた変更が発生した後に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。ロギングされた変更中に、ロギングされた変更の各々に関連付けられたプリイメージのロギングを行うことができる。

10

20

【0012】

いくつかの実施形態では、ターゲットシステムの1つ以上の実行ユニット上で実行される1つ以上のコンピュータプログラムのメモリおよびレジスタの少なくとも一部の内容を再構成状態として宛先時点で再構成する方法は、複数の時点における再構成状態を再構成すること、1つ以上のコンピュータプログラムの実行に起因するロギングされたデータに再構成を基づかせることを含む。ロギングされたデータは、1つ以上のコンピュータプログラムによって、または、オペレーティングシステムもしくは1つ以上のコンピュータプログラムの外部の別のエージェントによって、修正前の時間における再構成状態の少なくとも一部を表すプリイメージを含む。ロギングされたデータはまた、宛先時点の前にロギングされた少なくとも1つのレジスタ状態スナップショットを含む。方法はまた、宛先時点における再構成状態の少なくとも一部を決定するために命令セットシミュレータを適用することを含む。

30

【0013】

先行するいずれかの段落の方法は、1つ以上の以下の特徴を含むことができる。方法はさらに、複数の時点のうちの少なくとも1つにおける再構成状態の表現を維持することを含むことができる。方法はさらに、再構成状態の再構成中にロギングされたデータからプリイメージをコピーすることによってメモリ修正の直前における再構成状態を再現することを含むことができる。方法はさらに、連続的に以前のメモリ修正記録の直前の時点における再構成状態を再現し、それにより1つ以上のコンピュータプログラムの実行中に連続的に以前のポイントを再現することを含むことができる。方法はさらに、ポストイメージを記憶するためのメモリ空間を予約することを含むことができる。方法はさらに、ポストイメージのために予約された同じメモリ空間にプリイメージを保存することを含むことができる。方法はさらに、ポストイメージを記憶するための予約されたメモリ空間にポストイメージとして表現の少なくとも一部をコピーすることを含むことができる。方法はさらに、ポストイメージをコピーすることによりメモリ修正直後の時点における再構成状態を再構成することを含むことができる。方法はさらに、連続的に以後のメモリ修正直後の時点における再構成状態を再構成し、それにより1つ以上のプログラムの実行中に連続的にますます以後のポイントを再現することを含むことができる。方法はさらに、実行ユニットが1つの時点で動作した再構成状態の少なくとも一部を再現すること、および、後の時

40

50

点における再構成状態の少なくとも一部を決定するために命令セットシミュレータを適用することを含むことができる。方法はさらに、実行ユニットが前方または後方にメモリを再構成することによって動作した再構成状態の少なくとも一部を再現することを含むことができる。後の時点を宛先時点とすることができる。

【0014】

先行するいずれかの段落の方法は、1つ以上の以下の特徴を含むことができる。方法はさらに、レジスタ状態が宛先時点において未知である実行ユニットのセットを決定することを含むことができる。方法はさらに、レジスタ状態が宛先時点において未知である各実行ユニットについて、再構成ポイントと宛先時点との間の少なくとも1つのレジスタ状態スナップショットがロギングされているように再構成ポイントを決定することを含むことができる。方法はさらに、再構成ポイントにおける再構成状態の少なくとも一部を再構成することを含むことができる。方法はさらに、宛先時点に戻るためにメモリ再構成技術および再構成シミュレーション技術の組み合わせを使用することを含むことができる。方法はさらに、実行ユニットのセットのうちの実行ユニットについての正確なレジスタ状態情報が利用可能であるとき再構成シミュレーション技術を使用することを含むことができる。方法はさらに、実行ユニットのセットのうちの実行ユニットについての正確なレジスタ状態情報が利用できないときメモリ再構成技術を使用することを含むことができる。

【0015】

先行するいずれかの段落の方法は、1つ以上の以下の特徴を含むことができる。方法はさらに、再構成状態のサブセットの再構成が可能ではないことを決定することを含むことができる。再構成状態のサブセットの再構成は、ロギングされないメモリ修正についての1つ以上のプリイメージ値のために可能ではないかもしれない。副作用揮発性メモリに保存するとき、メモリ修正についての1つ以上のプリイメージ値がロギングされないかもしれない。メモリの状態のサブセットの再構成は、確かではないロギングされたイベントの正しい順序のために可能ではないかもしれない。ロギングされたイベントの正しい順序は、1つ以上のコンピュータプログラム内の1つ以上の競合条件のために確かではないかもしれない。方法はさらに、1つ以上の競合条件に関する情報を表示することを含むことができる。表示される情報は、1つ以上のメモリ位置にアクセス競合する1つ以上のコンピュータプログラム内の位置を含むことができる。1つ以上のコンピュータプログラム内の位置は、ソースコード位置とすることができる。

【0016】

先行するいずれかの段落の方法は、1つ以上の以下の特徴を含むことができる。方法はさらに、再構成状態の未知のサブセットを追跡することを含むことができる。方法はさらに、シミュレーションを行う際に再構成状態の異なるサブセットに再構成状態のサブセットの未知性を伝搬することを含むことができる。1つ以上のコンピュータプログラムの命令の結果が既知である場合、未知性を伝播せずともよい。方法はさらに、値が既知になったときに既知であると再構成状態の未知のサブセットをマーキングすることを含むことができる。値は、既知の値を伴うレジスタまたはメモリ位置をロードする命令をシミュレートした結果として既知になることができる。レジスタ状態スナップショットから決定することができる場合に、値は既知になることができる。

【0017】

いくつかの実施形態において、非一時的コンピュータストレージは、1つ以上のプロセッサによって実行されるときに先行するいずれかの段落の方法を1つ以上のプロセッサに実行させる命令を保存する。いくつかの実施形態において、コンピュータシステムは、先行するいずれかの段落の方法を実装するように構成された少なくとも1つのメモリおよび1つ以上のプロセッサを含む。

【図面の簡単な説明】

【0018】

他の特徴に進む前に、本明細書に開示された実施形態の態様および利点は、様々な実施形態の図面を参照して以下に詳細に記載され、これらは、例示であって、様々な実施形態

10

20

30

40

50

を限定しないことが意図されている。図面は以下の図を含む。

【0019】

【図1A】図1Aは、デバッガおよびコンパイラを含む一般的なハードウェアおよびソフトウェアアーキテクチャのブロック図を示す。

【図1B】図1Bは、デバッガおよびコンパイラを含む一般的なハードウェアおよびソフトウェアアーキテクチャの別のブロック図を示す。

【図2】図2は、本明細書に開示されたデバッガおよびコンパイラのシステムおよび方法の実施形態を含むハードウェアおよびソフトウェアアーキテクチャのブロック図を示す。

【図3】図3は、本明細書に開示されたデバッガおよびコンパイラのシステムおよび方法の例示的な利点を示す棒グラフを示す。

【図4】図4は、タスクをコンパイルし、デバッグするための高レベルプロセスの実施形態を示すフローチャートである。

【図5】図5は、オペレーティングシステムおよびターゲットタスクと相互作用するコンパイラシステムおよびデバッグシステムの実施形態の高レベルの概要を示すブロック図である。

【図6】図6は、タスクの実行中に発生したイベントを再生および／またはシミュレートするように構成されたデバッグシステムの実施形態の高レベルの概要を示すブロック図である。

【図7】図7は、タスクをコンパイルし、デバッグする実施形態を示すフローチャートである。

【図8A】図8Aは、基本ブロックを含むソフトウェアプログラムの高レベルの概要を示すブロック図である。

【図8B】図8Bは、基本ブロックインストルメント化最適化を示すブロック図である。

【図8C】図8Cは、別の基本ブロックインストルメント化最適化を示すブロック図である。

【図8D】図8Dは、さらに別の基本ブロックインストルメント化最適化を示すブロック図である。

【図9】図9は、留保トレースデータ最適化の実施形態を示すフローチャートである。

【図10A】図10Aは、タスクの実行中にトレースデータを生成するためのプロセスの実施形態を示すフローチャートである。

【図10B】図10Bは、タスクの実行中にトレースデータを生成するためのプロセスの実施形態を示すフローチャートである。

【図11】図11は、関数を含むソフトウェアプログラムの実施形態の高レベルの概要を示すブロック図である。

【図12】図12は、プログラムの実行中にトレースデータを生成するためのプロセスの実施形態を示すフローチャートである。

【図13A】図13Aは、デバッガシステムのグラフィカルユーザインタフェースを示す。

【図13B】図13Bは、デバッガシステムのグラフィカルユーザインタフェースを示す。

【図14】図14は、本明細書に開示されたコンパイラおよび／またはデバッグのシステムおよび方法を動作させるように構成されたコンピュータシステムの実施形態を示すブロック図である。

【図15A】図15Aは、いくつかの実施形態に係るメモリの再構成を示す。

【図15B】図15Bは、いくつかの実施形態に係るメモリの再構成を示す。

【図15C】図15Cは、いくつかの実施形態に係るメモリの再構成を示す。

【図15D】図15Dは、いくつかの実施形態に係るメモリの再構成を示す。

【図16】図16は、別の実施形態に係るメモリの再構成を示す。

【図17】図17は、実施形態に係るレジスタの再構成を示す。

【図18A】図18Aは、いくつかの実施形態に係るプリイメージロギングを示す。

【図18B】図18Bは、いくつかの実施形態に係るプリイメージロギングを示す。

【図18C】図18Cは、いくつかの実施形態に係るプリイメージロギングを示す。

10

20

30

40

50

【図 1 8 D】図 1 8 D は、いくつかの実施形態に係るプリイメージロギングを示す。

【図 1 9】図 1 9 は、実施形態に係るデバッグ用データの利用可能性を示す。

【図 2 0】図 2 0 は、実施形態に係るログデータのマージを示す。

【図 2 1 A】図 2 1 A は、いくつかの実施形態に係るログデータの順序付けおよびマージを示す。

【図 2 1 B】図 2 1 B は、いくつかの実施形態に係るログデータの順序付けおよびマージを示す。

【図 2 1 C】図 2 1 C は、いくつかの実施形態に係るログデータの順序付けおよびマージを示す。

【図 2 1 D】図 2 1 D は、いくつかの実施形態に係るログデータの順序付けおよびマージを示す。

10

【図 2 2】図 2 2 は、実施形態に係るマージされたログからの特定のスレッドについてのログデータの利用を示す。

【図 2 3 A】図 2 3 A は、実施形態に係る特定のスレッドによって実行される命令のリストの決定を示す。

【図 2 3 B】図 2 3 B は、実施形態に係る特定のスレッドによって実行される命令のリストの決定を示す。

【図 2 4】図 2 4 は、実施形態に係る後のメモリ修正記録の適用例を示す。

【図 2 5 A】図 2 5 A は、実施形態に係る均一細分化によるタイムスタンプ補間実行を示す。

20

【図 2 5 B】図 2 5 B は、実施形態に係る不均一細分化によるタイムスタンプ補間実行を示す。

【図 2 6 A】図 2 6 A は、実施形態に係る初期チャンク割り当てを示す。

【図 2 6 B】図 2 6 B は、実施形態に係るチャンクアーカイブ化を示す。

【図 2 6 C】図 2 6 C は、実施形態に係るチャンクリサイクル化を示す。

【図 2 6 D】図 2 6 D は、実施形態に係る不良ストア回復を示す。

【図 2 6 E】図 2 6 E は、別の実施形態に係る不良ストア回復を示す。

【図 2 7 A】図 2 7 A は、いくつかの実施形態に係る S M P システムにおける競合条件を示す。

【図 2 7 B】図 2 7 B は、いくつかの実施形態に係る S M P システムにおける競合条件を示す。

30

【図 2 7 C】図 2 7 C は、いくつかの実施形態に係る S M P システムにおける競合条件を示す。

【図 2 7 D】図 2 7 D は、いくつかの実施形態に係る S M P システムにおける競合条件を示す。

【図 2 8】図 2 8 は、実施形態に係るキャッシュ管理のためのコードフロー解析を示す。

【発明を実施するための形態】

【0 0 2 0】

いくつかの実施形態、実施例および図が以下に開示されているが、本開示が具体的に開示された実施形態、実施例および図を越えて延長しており、本開示の他の使用および明らかな修正ならびにその均等物を含むことは当業者によって理解されるであろう。開示の実施形態は添付図面を参照して説明され、全体を通して同様の数字は同様の要素を指す。本明細書に提示される説明で使用される用語は、それが特定の実施形態の詳細な説明と併せて使用されているという理由だけで任意の限定または制限された方法で解釈されるものではないことが意図される。また、実施形態はいくつかの新規な特徴を含むことができ、単一の特徴が、その望ましい属性に単独で責任がある、または本明細書に記載の開示を実践するために不可欠であるというわけではない。

40

【0 0 2 1】

《概要》

コンピュータプログラマは、コンピュータプログラムのエラーすなわち「バグ」を発見

50

するためにソフトウェアデバッグを使用する。エラーには発見が困難なものがあり、それらが発生した原因となる状況の後、長く経ってからそれらが現れるからである。例えば、数百万または数十億の命令が最初の計算の後に実行されるまで、不正な計算の結果は使用されないかもしれない。

【0022】

他のバグとして、それらの再現に長い時間がかかるか、または、バグを引き起こす条件が不確定である、もしくは他の未知の要因に依存しているときに、発見が困難なものがある。例えば、乗用車エンジンまたはスマートフォンのジャイロセンサからの特定のセンサ読取値が値の特定のシーケンスを有する場合にのみ、これらのセンサ読取値などの変化する外部入力に依存するプログラムがエラーを示し得る。このような場合には、バグがデバ

10

【0023】

他のバグには、発見後であっても修正が困難なものがあるが、それは、それらが発生するソースコードの複雑さおよび巨大さのためである。プログラマは、プログラムの実行中にどこか他の箇所でエラーを発生させることなくバグを修正することができる前に、プログラムのソースコードの大部分を理解する必要があり得る。乗用車、スマートフォン、飛行機等などの埋込型コンピューティングシステムに依存している現代のデバイスは、数十人または数百人のエンジニアのチームによって書かれた数千万行のソースコードを実行す

20

【0024】

スマートフォン、タブレット、ラップトップ、制御システム、車両制御システム等などの複雑な現代のデバイスでしばしば見られるように、数百万または数十億の実行される命令および／またはコード行が存在する場合、再現するのに長い時間がかかるバグが存在する場合、またはソフトウェアが非常に複雑であるために修正が困難なバグがある場合、人間の心の中で、または紙と鉛筆とを使って、合理的時間内で人間がこのようなソフトウェアコードをデバッグすることはできない。さらに、ソフトウェアデバッグの問題はコンピ

30

【0025】

デバッグソリューションには、ハードウェア、ファームウェアおよびソフトウェア開発者がバグおよび／またはエラーを発見およびフィックスするだけでなくそれらのコードを最適化および／またはテストすることができるようにする様々な解析ツールを提供するものがある。これらの解析ツールの1つのクラスは、1つ以上のプロセッサ上で命令を実行しつつ生成されたログデータを調べる。プロセッサ自体（例えばプロセッサトレース）により、オペレーティングシステムにより、ソフトウェア開発者によって追加されたインストールメント化ログポイントにより、および／または、コンパイラまたはJIT（ジャストインタイム）コンパイラによって追加されたインストールメント化により、ログデータを生成することができる。ロジックアナライザなどの他のデータのソース、他のシステムのコレクション、および、検証スクリプトまたは他のソースからのログは、システムの外部にあり得る。これらの異なるソースによって生成されたデータを「トレースデータ」と称し得る。トレースデータの単一要素を「イベント」と称し得る。

40

【0026】

50

「時間走行デバッガ」と称され得るデバッガシステムには、プログラマが自身のプログラムの実行において前方および後方に移動可能とするものがある。時間走行デバッガは、バグの症状からバグの根本原因へとすべて遡ってプログラムをデバッグするために、または、プログラマがプログラムの一般的構造を理解して、プログラムがどのように編成されるのか、および、いつどこでプログラムの様々な関数が発生して実行されるのかを知ることができるようにするために、使用され得る。

【0027】

時間走行デバッガは、2つの構成要素：データの生成、収集、および（いくつかの実施形態では）解釈を担う時間走行デバッガバックエンド、ならびに、ユーザへのデータ提示を担う時間走行デバッガユーザインタフェース、を含むことができる。

【0028】

（「ハードウェアトレースバックエンド」と称され得る）時間走行デバッガバックエンドには、ハードウェアトレースの形態を含むものがある。ハードウェアトレースは、命令がCPUによって実行されたものを正確にプログラマが再構成することを可能にする、CPUによって放出された情報の高度圧縮ストリームである。ハードウェアトレースは、いくつかの制限を有する。例えば、ハードウェアトレースはすべてのCPUアーキテクチャで利用可能というわけではなく、したがって一部のCPU上で使用可能であるにすぎない。ハードウェアトレースを有するCPUであっても、メーカーは、CPUのチップパッケージサイズを縮小して生産コストを節約するために、ハードウェアトレースを放出するトレース出力ポートを備えるピンをインストールしないよう選択することがある。また、ハードウェアトレースは、トレースプローブなどの外部トレース収集ハードウェアを必要とし得、これは高価であり得、トレースデータを記録するためにメモリに制限があり得、いくつかのシステムにとってインストールするには物理的に困難であり得る。例えば、スマートフォンは小型でポータブルにする必要があり、そのトレース出力ポートに取り付けられた大規模で重いトレースプローブを使用するのは困難であろう。

【0029】

また、ハードウェアトレースは信頼できない可能性があり、トレース出力ポートがCPUの命令実行に追いつくことができないときに情報の欠落がある可能性がある。CPUを減速またはストールさせるよりもむしろ、ほとんどのメーカーは、CPUがトレース出力ポートをオーバーロードさせ得る場合にはトレースデータを放出しないことを選択している。メモリアクセスのハードウェアトレースは、トレース出力のオーバーロードに対して特に敏感であり、これはメモリアクセスが予測できず、したがって出力するには容易に圧縮され得ないからである。トレースデータにおけるメモリアクセスの省略は、時間走行デバッガの機能を厳しく阻害する。また、ハードウェアトレースは、オペレーティングシステムをよくトレースすることはできない。このようなデバッグシステムは、オペレーティングシステム内のタスクが作成または破棄されたときを伝えることができ得るが、リソースの割り当て、アドレス空間の作成、および様々な同期プリミティブの使用などのすべての他のOSの動作に盲目である。しかし、これらのOSの活動のすべてを知することは、バグの根本原因を発見するのに重要である可能性があり、それは特に、現代のデバイスが多くの場合、すべてが互いに相互作用する数十～数百もの同時実行中タスクを伴うオペレーティングシステムを使用するからである。

【0030】

また、ハードウェアトレースは十分に対称型マルチプロセッシング（SMP）システムを追跡することができない。ハードウェアトレースはしばしば、2つ超のコアについてのトレースデータ放出についての帯域幅制限のためにSMPシステムにおける2つ超のコアをトレースすることができない。このようなデバッグシステムは、限られた使用となる可能性があり、これはスマートフォンおよびタブレットなどの多くの現代のコンピューティングデバイスが一般に、SMP実行についての4つ超のコアを有しているからである。さらに、ハードウェアトレースは、複数の異なるタイプのCPUおよびシステムが互いに対して非同期的に実行される非対称マルチプロセッシングシステムを追跡する方法をほとんど有

10

20

30

40

50

していない。これは、このようなデバッグシステムを大きく制限する可能性があり、それは例えば、現代の乗用車が互いに非同期で動作する百超の独立したCPUを有するからである。さらに、ハードウェアトレースは、時間走行デバッガにおいて使用するのに適した形式にデコードすることが非常に遅いので、プログラマは、プログラムをデバッグするためにトレースデータを使用する前に長時間、待つ必要があり得る。市販の製品についての典型的なデコード速度は、毎秒1メガバイト未満から2MB/秒の範囲である。トレースプローブは典型的には、1GBのメモリを搭載しており、完全にトレースプローブのトレースデータをデコードするには20分超かかり得る。また、ハードウェアトレースデコードは、収集されたトレースデータの先頭から行われるので、トレースデータがより多く収集されるほど、トレースデータのデコードがより長くなる。今日では最大16GBのトレースメモリを伴って販売されているトレースプローブを用いれば、プログラマがプログラムをデバッグするためにトレースデータを使用することができるまで、完全な16GBのトレースバッファをデコードするのに、ほぼ6時間かかる可能性がある。ハードウェアトレースデコード時間は、トレースバッファのサイズとともに増大する。

【0031】

本明細書に開示された実施形態は、デバッグシステムに伴う、および／またはトレースのためのソフトウェアベースアプローチに伴う、上記問題を解決しようとする。実施形態では、本明細書に開示されたシステムは、コンパイラを使用して付加的命令を用いてプログラムをインストルメント化する。実施形態では、オペレーティングシステムを、デバッグシステムに伴う上記課題を解決するためにオペレーティングシステム上で実行されているインストルメント化ターゲットプログラムとともに作業および／または協働するように構成することができる。

【0032】

ソフトウェアインストルメント化の特定の形態を含む時間走行デバッガバックエンド（「コピーバックエンド」とも称され得る）は、ターゲットプログラムを大幅に遅くする可能性がある一方、本明細書に開示された実施形態は、これらの速度ペナルティを低減する特徴を含む。コピーバックエンドは、チェックポイントとして実行プログラムのメモリの大部分のコピー（または「フォーク」）を介してターゲット全体の動作状態を保存する負担の大きい方法を必要とする一方、本明細書に開示された実施形態は、このような状態のバルクコピーを必要としない。さらに、コピーバックエンドはまた、すべてのプロセスをトレースするために、1つのCPUにそれらを捕らえる必要がある。しかしバグには、同時実行で発生するタイミング依存性のために、異なるCPU上でプロセスが実行される場合にのみ発生するものがある。実施形態において、本明細書に開示されるシステムは、前述の制限に対処する。

【0033】

実施形態では、システムは、改善された時間走行デバッガバックエンド（「改善バックエンド」と称され得る）を実装するのに必要なトレース情報を生成するために、ターゲットプログラムおよびオペレーティングシステムにおいてインストルメント化を使用する。インストルメント化によって生成されたトレース情報を、デバッガによってそれが検索されて解釈されることができるまで、コンピュータプログラムを実行するCPUのメモリにおいてロギングすることができる。このロギングされたトレース情報は、「トレースログ」と呼ばれる。実施形態において、改善バックエンドは、最新から最古までトレース情報を検索して解釈する。トレース情報は逆時系列順に解釈されるように構成および／または設計されることができ、それにより、ユーザは、以前の部分が検索および／またはデコードされるのを待たずに、プログラムの最も近い時点での実行履歴のデバッグを開始し得る。

【0034】

実施形態では、改善バックエンドは、コンパイルツール（コンパイラ、アセンブラ、リンカ等）により自動的に挿入されたターゲットプログラムにおけるインストルメント化を含むことができる。ターゲットプログラムにおけるこのようなインストルメント化は、2つの主な目的を果たすことができる。

【0035】

第1に、このようなインストルメント化は、ターゲットプログラムの実行パス、例えば、どの命令が実行され、および／または、どのような順序で実行されるか、を記録するように構成されることができる。このようなインストルメント化は、実行される各基本ブロックのログエントリを作成することによってこれを行うように構成されることができる。基本ブロックは、外部割り込みが存在しない状態で単一エントリポイントおよび単一グジットポイントを有するプログラムにおける命令のシーケンスとして定義され得、ブロック内の各命令は、正確に一度だけ実行される。CPUアーキテクチャの機能に応じて、基本ブロックの定義は、例えば、条件付命令（特定の条件下では影響を及ぼさないで、実行されていないと考えられ得る）を考慮するよう、または例えばメモリコピーを実行するために複数回実行する命令を繰り返すよう修正され得、適宜、他の修正もなされ得る。

10

【0036】

第2に、ターゲットプログラムにおけるこのようなインストルメント化は、ターゲットプログラムのメモリへの変更を追跡および／または監視するように構成されることができる。メモリが上書きされると、改善バックエンドは、修正アドレスおよび／または修正前のメモリの値（「プリイメージ」）をロギングするように構成されることができる。実施形態では、プリイメージは、以下のように再構成が逆時系列順で発生することができるようロギングされる。ターゲット上のメモリの現在の状態のイメージで始めると、デバッグ改善バックエンドは、過去の任意の時点でのメモリの状態を最も近い時点から始めてより以前へ以前へと進むよう再現するために逆時系列順にプリイメージを適用することができる。実施形態において、ターゲットメモリイメージに適用される各プリイメージは、プログラム実行中に実行されたメモリ修正のうちの1つの取り消しをシミュレートする。この逆再構成を行いながら、デバッグシステムがプリイメージでターゲットメモリイメージの一部を上書きするたびに、デバッグシステムが上書きしようとするメモリの内容を、「ポストイメージ」としてログに保存し得る。ポストイメージを保存すると、デバッグ改善バックエンドは、ターゲットメモリイメージにポストイメージを適用して時間的に前方に対応メモリ修正をシミュレートするように構成されることができる。したがって、プログラムの最終状態から開始して、改善バックエンドは、時間的に後方に、そして必要に応じて再び前方に、ターゲットプログラムのメモリ状態を自由に再構成するように構成されることができる。

20

30

【0037】

実施形態では、プリイメージのロギングが可能でない場合、例えば修正中のメモリがハードウェアペリフェラル上のメモリマップされたレジスタであって、読み出し時に望ましくない副作用を有する可能性がある場合、プリイメージロギング技術は調整される。

【0038】

実施形態では、ロギング値を他のロギングデータから推測することができる場合には、コンパイルツールは、インストルメント化を低減または排除するための様々な技術を使用する。これらの最適化は、ロギングオーバーヘッドを削減し、所与量のログに保存することができる実行履歴の量を増加させる。

【0039】

実施形態では、改善バックエンドは、アプリケーション外部から発信されるターゲットプログラムの実行パスおよびメモリへの変更をロギングするように構成されたオペレーティングシステムにおけるインストルメント化を含むことができる。ターゲットプログラムの実行パスおよびメモリへの変更の例として、割り込み、コンテキストスイッチ、オペレーティングシステムコールによって引き起こされるメモリ修正、ならびにペリフェラルからのプログラムおよび他のプログラムの外部から到来する通信を挙げることができるがこれらに限定されない。ターゲットプログラムの内部インストルメント化によってロギングされるメモリ修正を伴うので、改善バックエンドでは、オペレーティングシステムを、後方再構成の目的のためにプリイメージとしてメモリ修正をロギングするように構成することができる。

40

50

【0040】

本明細書に開示されたシステムのオペレーティングシステムは、ターゲットプログラムにおける実行のすべてのスレッド内のCPUレジスタの状態を定期的にロギングするように構成されることができる。実施形態では、前述のデータは、プログラム内のレジスタの状態を再構成するために使用される。メモリが本明細書に記載された技術を用いて所望の時点で再構成された後、デバッガは、シミュレータを使用して所望の時点の前に記録された最も近いレジスタ状態スナップショットから前方にシミュレートし、所望の時点でのCPUレジスタの状態を再構成することができる。

【0041】

実施形態では、本明細書に記載のインストルメント化および／または技術は、本明細書に記載のようにハードウェアトレースバックエンドおよびコピーバックエンドに通常は関連する様々な制限を負わない、改善された時間走行デバッガバックエンドを実装するのに十分である。

【0042】

任意選択的に、改善バックエンドを、様々な目的のための追加情報を収集するように構成することができる。例えば、デバッグ中のターゲットプログラムが、例えば対称または非対称マルチプロセッシング構成において同時に実行されている複数の実行スレッドを有する場合、改善バックエンドは、任意選択的にタイムスタンプとともに、CPU間の同期イベントおよび他の通信をロギングするように構成されることができる。十分な情報がロギングされて論理的に一貫した順序、例えば特定の重要イベントについてのタイムスタンプで異なる実行ユニットからのデータをマージできると仮定すると、改善バックエンドは、時間走行デバッガを複数の実行ユニットを含むシステム上で動作させることができる。このようなデータの収集も、ハザード、競合条件および／または他の潜在的プログラミングエラーの自動検出を、たとえこれらのエラーにより解析中のターゲットプログラム実行の一部において不正挙動とならなかったとしても、改善バックエンドに実行させることができる。

【0043】

実施形態では、改善バックエンドは、経時的にプログラムのコールスタックにおける変化を表示するために使用することができる関数エントリおよびイグジット（FEE）データを収集するように適合されるインストルメント化を含むように構成されることができる。FEEデータは、パフォーマンス解析および最適化について、ならびに、デバッグについて、データを価値あるものにするタイムスタンプ情報を含むことができる。

【0044】

本明細書に開示された実施形態は、本明細書に記載のハードウェアトレースバックエンドを用いて制限を克服し得る。本明細書で説明するように、ハードウェアトレースは、すべてのアーキテクチャで利用可能ではない。対照的に、本明細書に開示された実施形態は、ハードウェアトレース能力に依存しないため、実質上すべてのアーキテクチャ上で動作する。さらに、本明細書の実施形態は、外部トレース収集ハードウェアなしで動作することができる。したがって、実施形態は、高コスト、大サイズ、およびトレースプローブの限られたメモリ容量に服従することはないであろう。実施形態は、インストルメント化を使用することができ、これは、トレース出力ポートがプログラム実行に追いつくことができないときにハードウェアトレースで発生する可能性があるようなデータ喪失がないようにプログラム実行を遅くする。

【0045】

また、実施形態はオペレーティングシステムをインストルメント化することができ、それにより、実施形態は、ハードウェアトレースが認識していないオペレーティングシステムイベントを記録することが可能である。これらのオペレーティングシステムイベントは、バグの根本原因を発見するのに重要である可能性がある。実施形態では、システムは、CPU間の同期プリミティブおよび通信を認識するように構成され、SMP環境および複数の実行ユニットを伴う他の構成で実行されるプログラムのデバッグが可能となる。また

10

20

30

40

50

、異なるCPUからデータを論理的に一貫した順序でマージするのに十分な情報をロギングすることにより、実施形態は、非対称マルチプロセッシングシステムおよび他の分散システムで動作する能力を有することができる。実施形態において、本明細書で開示される実施形態でのプリイメージロギングの使用により、トレースデータが時間走行デバッガユーザインタフェースにおける使用に既に適した形態になるようにすることができ、よって、すべてのトレースデータがダウンロードおよびデコードされるのを待つことなく、ターゲットからダウンロードされるとすぐにトレースデータを用いてプログラマがデバッグを開始し得る。

【0046】

さらに、改善バックエンドは、すべてのトレースデータがダウンロードされる前に、プログラマがデバッグを開始できるように構成されることができる。ユーザが待たなければならない時間は、トレースメモリバッファのサイズの関数ではなく、はるかに大きなトレースバッファの実用的使用を可能にする。実施形態では、改善バックエンドは、アドレス空間をコピーすること、またはそれらをトレースするために1つのCPUにすべてのプロセスを捕らえることに依存しないという点で有利である可能性がある。そのため、実施形態は、このようなデータのバルクコピーに固有のパフォーマンスペナルティに悩まされず、プログラマが同時実行中にのみ発生するバグを発見するのに役立つことができる。

【0047】

開示されたシステム、方法およびデバイスは、コンピューティングの分野で生じる特定の技術的問題を解決し、コンピュータシステムの動作を改善する。デバッグの分野における特定の具体的な改善が開示されている。開示されたシステム、方法およびデバイスは、少なくとも、コンピュータコードデバッグ技術における大幅な改善を提供し、これはとりわけ、伝統的デバッグ技術と比較してそれらがより効率的なインストルメント化、ロギングおよび／または再構成を介してデバッグの速度および正確性を向上させるからである。開示されたシステム、方法およびデバイスは、単一の処理コアまたは複数の処理コアを有するコンピューティングシステム上で実行されるソフトウェアをデバッグするために使用されることができる。複数のコア上で実行されるソフトウェアをデバッグすることは、とりわけ複数コア上での実行スレッド追跡の複雑さのために、特に挑戦的な課題である可能性がある。開示された技術は、とりわけオペレーティングシステムイベントを追跡して一貫した順序で異なるコンピューティングコアからデータをマージすることにより、複数の処理コア上で実行されるコンピュータコードをデバッグすることに関連したこれらおよび他の問題をシームレスに解決する。

【0048】

プログラマによる手動操作および介入に依存するにもかかわらず遅くてエラーが発生しやすく、多くのタイプのバグを検出およびフィックスするのに不十分である従来のシステムとは異なり、開示された技術は、競合条件バグ、メモリ破損バグおよび他の再現困難なバグを含む多数のソフトウェアバグを自動的に検出してそのデバッグを容易にすることができる。それらの効率のため、開示されたシステム、方法およびデバイスは、コンピュータプログラムの実行へと追加されるオーバーヘッドをより少なくし得、言い換えれば、従来のシステムよりも使用するのにより少ない侵入性および／または煩雑性となり得る。開示されたシステム、方法およびデバイスは、シングルまたはマルチコアコンピューティングシステム上で実行される最も複雑なコンピュータコードであっても高速自動デバッグを可能にし、従来技術よりも効率的かつ正確なデバッグを提供する。これにより、コンピュータプログラムデバッグ技術を含むコンピュータ関連技術における大幅な改善だけでなく、本明細書に開示された特定の技術を使用してデバッグされたエラーフリーコンピュータプログラムを実行するこのようなシステムを介して達成されるコンピューティングシステムの動作改善が得られる。

【0049】

「コンピュータシステム」は、RAM、ROM、内蔵SRAM、オンチップRAM、オンチップフラッシュ、CD-ROM、ハードディスクおよび同様のものを含むがこれらに

10

20

30

40

50

限定されない、処理ユニットとデータおよび命令を交換するための1つ以上のデータ記憶デバイスと結合されたデータおよび命令の処理のための1つ以上の処理デバイス（中央処理装置CPUなどの）を含み得る。コンピュータシステムの例は、エンジンコントローラからラップトップまたはデスクトップコンピュータ、スーパーコンピュータまでに至るすべてのものを含むことができる。データ記憶デバイスは、専用で、例えば処理ユニットに直接に結合され、または遠隔で、例えばコンピュータネットワークを介して処理ユニットに結合されることができる。コンピュータネットワークを介して処理ユニットに結合されたりリモートデータ記憶デバイスが実行のために処理ユニットにプログラム命令を送信することができることを理解すべきである。また、処理デバイスを、1つ以上の追加の処理デバイスに、同じ物理的構造を介して（例えば並列プロセッサ）、またはコンピュータネットワークを介して（例えば分散プロセッサ）のいずれかで結合することができる。このような遠隔結合されたデータ記憶デバイスおよびプロセッサの使用は、コンピュータサイエンス分野の当業者によく知られているであろう。「コンピュータネットワーク」との用語は、互いに通信可能なコンピュータシステムのセットを相互接続する通信チャネルのセットを含み得る。通信チャネルは、ツイストペア線、同軸ケーブル、光ファイバ、衛星リンク、またはデジタルマイクロ波無線などの伝送媒体を含むことができるが、これらに限定されない。コンピュータシステムは、ラージまたは「ワイド」エリア（例えば、数十、数百、あるいは数千マイルにわたる、WAN）、またはローカルエリアネットワーク（例えば、数フィート〜数百フィートにわたる、LAN）を介して分散することができる。さらに、様々なローカルエリアおよびワイドエリアネットワークを、コンピュータシステムの集合体ネットワークを形成するために組み合わせることができる。コンピュータネットワークのこのような連合の一例は、「インターネット」である。

【0050】

「ターゲット」との用語は、「コンピュータシステム」と同義であり得る。ターゲットとの用語を、トレースイベントを生成するコンピュータシステムがトレースイベントを解析するために使用されるコンピュータシステムとは異なり得ることを示すために使用することができる。なお、同じコンピュータシステムがトレースイベントの生成および解析の両方を行うことができる。

【0051】

「スレッド」との用語は、命令の実行の任意のユニットを指すために使用され得る。スレッドは通常、主にそれ自身の使用のためである（レジスタなどの）状態を記憶する方法を有することができる。それは（そのアドレス空間内のRAMなどの）他のスレッドと追加の状態記憶空間を共有してもしなくてもよい。例えば、これはオペレーティングシステムで実行される際にプロセス内で実行中のスレッドを参照し得る。これはまた、オペレーティングシステムなしでプロセッサ上で命令を実行することを含むことができる。その場合には、「スレッド」は、命令を実行するプロセッサであり、いかなるコンテキストスイッチも存在しなくともよい。異なるオペレーティングシステムおよび環境は、スレッドとの用語でカバーされる概念を指すために異なる用語を使用し得る。同じ基本原理の他の一般的な用語は、限定されないが、ハードウェアスレッド、軽量プロセス、ユーザスレッド、グリーンスレッド、カーネルスレッド、タスク、プロセスおよびファイバを含む。

【0052】

《インストルメント化》

時間走行デバッガについての改善バックエンドは、コンピュータプログラムの実行に関するデータを収集するためのインストルメント化を含み得る。インストルメント化は、プログラムの動作に直接は寄与しないコンピュータプログラムに追加される命令を含むことができる。インストルメント化は、例えば、コンパイラ、JIT（ジャストインタイム）コンパイラ、アセンブラ、リンカ、もしくはソースコードから実行可能コードにプログラムを変換するプロセスにて使用される他のツールによって、または、完全コンパイル化実行可能ファイルのポストプロセッシングによって、自動的にプログラムに挿入されることができる。インストルメント化は、プログラマが明示的または黙示的にプログラムにインポ

ートした命令のライブラリ、ヘッダファイル、または他のボディに埋め込まれることができる。インストルメント化は、プログラマによって手動でコンピュータプログラムに書き込まれることができる。コンパイラは、プログラマがプログラムソースコードに挿入してコンパイラに適切なインストルメント化命令を生成させることができる利用可能な特別コンパイラキーワードまたはイントリンシックを作成することによりプログラムを手動でインストルメント化できるようプログラムを支援するように構成することができる。インストルメント化は、コンピュータプログラムを実行するオペレーティングシステムまたはハイパーバイザに埋め込まれることができる。インストルメント化は、コンピュータプログラムが実行されるハードウェアまたはシミュレータに、またはコンピュータシステムの動作を監視する補助ハードウェアに埋め込まれることができる。

10

【0053】

インストルメント化を、ほとんどすべての種類の情報をロギングするために使用することができる。改善バックエンドは、プログラムの実行パスに関する情報、およびプログラム内のメモリに記憶されたデータへの変更にに関する情報、ならびに、プログラマに直接的または間接的に有用であることを証明し得る他の情報をロギングするように構成されたインストルメント化を含むことができる。

【0054】

《インストルメント化の最適化および除去》

コンピュータプログラムに挿入されたインストルメント化は、ランタイムパフォーマンスおよび／またはそのコンピュータプログラムのメモリフットプリントに悪影響を与える可能性がある。インストルメント化はプログラムデバッグのためにおよび他の目的のために非常に有用である可能性があるが、いくつかまたはすべてのインストルメント化を除去したコンピュータプログラムの実行可能ファイルを構築することが有益であり得る状況もある。

20

【0055】

実施形態では、改善バックエンドによってコンピュータプログラムに挿入されたいくつかのインストルメント化は、他のロギングされたデータから、または他の情報のソースから推測することができるが、したがって、プログラム実行時に生成する必要がないデータを生成する。このような場合には、改善バックエンドは、そのようなインストルメント化を解消または簡素化して、データがターゲットシステムから収集された後のように後に推測データを挿入することができる。このような推測データは、「合成データ」、つまり、ランタイム時にインストルメント化によって記録されている可能性があるが、代わりに事実の後に生成されるデータの一例である。生成データの代わりに合成データを使用するプロセスは、インストルメント化、ランタイムパフォーマンスへの影響、およびバックエンドの機能実行のためのログ空間のうちの1つ以上を低減する方法を指すことができる「インストルメント化最適化」の一例である。インストルメント化最適化および合成データのいくつかの例は、本開示全体を通して現れる。

30

【0056】

実施形態では、改善バックエンドは、そもそもインストルメント化を生成せずに、手動で（例えば、プログラマによって）挿入される任意のインストルメント化を除去または無視するよう、コンパイラ、アセンブラ、および他の開発ツールを構成することにより、インストルメント化の生成を無効にしてインストルメント化することなく実行可能ファイルを構築できるようにすることができる。

40

【0057】

実施形態では、改善バックエンドは、インストルメント化を除去するように、リンカ、またはインストルメント化が挿入された後に呼び出されるソフトウェア開発ツールの別のコンポーネントを構成することによって、インストルメント化することなく、インストルメント化をストリップングして実行可能ファイルを構築することができる。インストルメント化のストリップングは、インストルメント化生成を無効にすることに比べていくつかの利点がある。インストルメント化を伴う実行可能ファイルが生成されると、実行可能フ

50

ファイルは最初から再構成される必要はなく、代わりに、ソフトウェア開発ツールを、すでに構築された実行可能ファイルおよび／またはオブジェクトファイルからインストール化をストリッピングするように構成することができる。これは、非インストール化実行可能ファイルを作成するのに必要な時間を非常に短縮することができる。非インストール化実行可能ファイルは単にインストール化が除去されたインストール化実行可能ファイルであるため、インストール化および非インストール化のバージョンが異なる挙動を有するリスクが大幅に軽減される一方、実行可能ファイルが最初から再構成される必要がある場合にそのような保証を行うことのほうがはるかに困難である。プログラムの非インストール化バージョンがユーザに配備されるバージョンである可能性が高く、プログラムのインストール化バージョンは開発中にほとんどの精査を受ける可能性があるため、バージョン間で整合性のある挙動を保つことは、インストール化を伴わない再構築のプロセスを通じた開発および配備の間にソフトウェアにバグが入る可能性を低減する。インストール化をストリッピングすることはまた、事前構築されたライブラリのプロバイダがインストール化を含むライブラリの単一のセットを分配することを可能とする。事前構築されたライブラリを使用するプログラマは、必要に応じてインストール化をストリッピングしてもしなくともよい。

10

【0058】

実施形態では、手動または自動でソフトウェアに挿入されたインストール化は、予約レジスタ、例えば、書き込まれるログ内の次の場所をポイントする予約レジスタ、またはロギング処理中に一時的な値を保持するために予約されたレジスタにアクセスする命令からなることができる。このような場合は、インストール化をストリッピングすることは、迅速かつ計算が簡単なプロセスとなることができ、予約レジスタを参照する命令の検出および除去のみに関与する。インストール化として挿入された命令のみが予約レジスタを使用し得、したがって、インストール化として挿入された、したがってコンピュータプログラムの動作に関与しない命令のみが除去される点において、インストール化を除去するプロセスの正しさを証明することも容易である。

20

【0059】

実施形態では、インストール化ストリッピングは、再配置可能オブジェクトファイルおよびライブラリを単一の実行可能イメージに結合するプロセスの前に、またはその間に発生することができ、このプロセスを「リンク」と称することができる。リンクプロセスは、最終実行可能ファイルにおいてその適切なターゲットにポイントするための再配置可能オブジェクトファイルおよびライブラリにおける、分岐、他の命令、およびポインタの修正を伴うことができ、このプロセスを「再配置」と呼ぶことができる。再配置プロセスは、命令の追加もしくは除去またはオブジェクトファイルおよびライブラリ内のマシン命令のサイズもしくは位置を変更する他のアクションに敏感であることができる。リンクプロセス中またはその前にインストール化ストリッピングを行うことにより、インストール化命令の除去に起因するコードサイズ変更によって再配置は崩壊しない。

30

【0060】

インストール化ストリッピングで行うことができるように、再配置可能オブジェクトファイルからの命令の除去は、ローカル分岐命令における、同じ再配置可能オブジェクトまたはライブラリ内の場所を参照する他の命令における、および、「再配置テーブル」と呼ぶことができる）再配置プロセス中の変更を必要とし得る命令およびポインタの位置をリスト化するテーブルにおける、変更を必要とする可能性がある。インストール化ストリッピングのプロセスはしたがって、インストール化命令の除去に加えておよびそれとともに、このような変更を行う必要がある可能性がある。

40

【0061】

実施形態では、プログラムに挿入される一部またはすべてのインストール化を、ランタイム時に無効にするように構成することができる。このような場合には、インストール化には、時にガード命令と称される、1つ以上の追加の命令が先行することができる。実施形態では、ガード命令は、デフォルトでは無効とすることができるが、デバッガ

50

により、オペレーティングシステムにより、または他の手段により、別の命令（複数可）、例えば何もロギングされないよう命令に残りのインストルメント化をスキップさせる分岐命令へとランタイム時に変更されることができる。実施形態において、上記プロセスは、ガード命令（複数可）への修正を逆転させることによって、インストルメント化の効果を復活させるよう逆転され得る。代替実施形態では、ガード命令は、ガード命令のデフォルト状態がインストルメント化をスキップするという点以外、本明細書に記載されるように動作することができる。

【0062】

実施形態では、実行可能バイナリイメージを生成するために使用されるコンパイラおよび／または他のツールは、ガード命令を含むコンパイルされたプログラム内の位置のテーブルを生成することができる。このようなテーブルは、デバッガによって、オペレーティングシステムによって、またはランタイム時にインストルメント化を有効および／または無効にするプロセスを案内する他の構成要素によって使用されることができる。

【0063】

本明細書に記載されるようにガード命令を用いてランタイム時にインストルメント化を無効にすることは、いくつかの実施形態において有利である可能性があり、それは、ユーザがアプリケーションプログラムを再コンパイル、再リンクまたは再始動すらずに、ロギングをオンおよび／またはオフすることができるからである。ユーザはまた、どのインストルメント化を有効／無効にするかについて、例えば1つの関数についてのみインストルメント化を有効にすること、または例えばインストルメント化の一種のみを有効にすることにおいて選択的であり得る。別の例として、ユーザは、特定の他の条件が埋め込まれた場合、例えば、デバッガ、オペレーティングシステム、またはアプリケーション自体が内部不整合を検出する場合にのみロギングを有効にすることができる。インストルメント化の無効は典型的には、実行をより迅速にし、有効インストルメント化に比べてより少ないログ空間を消費し、それにより、ランタイム時のインストルメント化の無効はアプリケーションパフォーマンスおよびログ使用量に対するより大きな制御をユーザに与えることができるものの、生成トレースデータの質および種類のトレードオフとなる。

【0064】

《インストルメント化のコンパイラ駆動最適化》

改善バックエンドは、プログラマの手作業による挿入、ライブラリもしくはまたはオペレーティングシステムにおける埋め込み、またはソースコードから実行可能ファイルにプログラムを変換するプロセス中の自動的挿入を含む、多くのソースからのインストルメント化を含むことができる。上記の場合のそれぞれにおいて改善バックエンドで、コンパイラは、実行可能コードにおけるインストルメント化のフットプリント低減、メモリ内のインストルメント化のフットプリント低減、および／またはインストルメント化のランタイムパフォーマンス影響低減、および／またはインストルメント化に必要なログ空間量低減のために、インストルメント化を最適化するように構成されることができる。

【0065】

実施形態において改善バックエンドで、コンパイラは、プログラムにおけるロギングポイントの位置ならびに定数ポインタおよび整数などのそのログポイントについて常に同じであるデータ（「静的データ」）と、タイムスタンプおよびプログラム変数の内容などの変更し得るデータ（「動的データ」）とを区別するために、各ロギングポイントでロギングされたデータを解析するように構成されることができる。その解析を考慮すると、コンパイラは、動的データのみをロギングするためのインストルメント化およびロギングポイントを表す一意識別子を構築するように構成されることができる。実施形態では、一意識別子を、プログラム内のロギングポイントの位置、または位置と相関する数とすることができる。改善バックエンドコンパイラは、ロギングポイントの一部またはすべてについての静的データを保存することができ、これは、実行可能イメージの非ロードセクションにて、またはデータファイルにて、または（プログラムまたは人であり得る）ロギングデータの消費者によってアクセス可能な別の場所にて、一意識別子によってインデックス付け

10

20

30

40

50

される。そのようなインデックスを「静的ルックアサイドテーブル」と称することができる。ロギングデータの消費者は、各ログエントリの長さを決定して欠落静的データを推測または合成するために、静的ルックアサイドテーブルを使用することができる。その結果、静的データはほとんど、あるいは全く、実行可能コードフットプリント、メモリフットプリント、ランタイムパフォーマンス、またはログ空間への影響を有しない。実施形態では、改善バックエンドは、ログからのタイムスタンプの上位ビットを省略して後でそれらを合成することなどの追加の最適化を実行し得る。

【0066】

いくつかの実施形態では、そのような最適化は、ロギング情報が静的データを含む場合には特に、非常に効率的なロギングをもたらすことができる。パフォーマンス制約により他の種類のロギングの使用が不可能である場合には、本明細書に記載するように最適化されたロギングを使用することが依然として可能であり得る。典型的な場合では、ランタイムパフォーマンスまたは他の制約に重大な影響を与えることなく、ユーザが自由にこのような最適化されたロギングを適用することができる。

【0067】

《実行パスについてのトレースデータ》

改善バックエンドは、インストルメント化を使用して、システムにおける実行のスレッドの実行パスをどれも記録しない、または、いくつかもしくはすべての実行のスレッドの実行パスを記録することができる。

【0068】

実施形態において、改善バックエンドにおけるコンパイラは、基本ブロックの第1の命令の前に各基本ブロックについてのインストルメント化を挿入するように構成されている。実施形態では、インストルメント化は、特定の基本ブロックが実行されたことを示すために、基本ブロックに関連付けられた（例えばプログラムカウンタの値など）の一意識別子をロギングするように構成されている。各命令の代わりに各基本ブロックをロギングすることは、実行のスレッドの実行パスをエンコードするためにロギングが必要なデータの量を低減することについて有利である可能性がある。

【0069】

実施形態において、改善バックエンドにおけるコンパイラは、1つ以上の分岐命令に関連付けられたインストルメント化を挿入するように構成されている。分岐命令とは、その実行が、プログラムメモリにおける分岐命令の直後に続かないコードの実行に直ちに先行し得る命令として定義され得る。実行される次のプログラム命令がプログラムメモリ内の分岐命令の直後の命令でない場合、分岐は、その実行の特定のインスタンスにて「行われる」と言われ得る。実施形態では、特定の分岐命令に関連付けられたインストルメント化は、分岐が行われているか否かに関係なく、ログデータを生成し得る。実施形態では、分岐が行われる場合、特定の分岐命令に関連付けられたインストルメント化は、ログデータを生成するのみであり得る。実施形態では、特定の分岐命令に関連付けられたインストルメント化は、分岐が行われていない場合に、ログデータを生成するのみであり得る。各命令の代わりに各ブランチをロギングすることは、実行のスレッドの実行パスをエンコードするためにロギングする必要があるデータの量を低減させるのに有利である可能性がある。

【0070】

実施形態では、分岐命令に関連付けられたインストルメント化は、分岐命令の後に実行される次の命令の命令をロギングするように構成されている。実施形態では、分岐命令に関連付けられたインストルメント化は、実行される次の命令のメモリアドレスなどの、実行される次の命令を一意に識別する識別子をロギングするように構成されている。別の実施形態では、分岐命令に関連付けられたインストルメント化は、実行される次の命令のメモリアドレスと、プログラムメモリ内の分岐命令に続く命令のメモリアドレスとの間の差などの、実行される次の命令の相対的指標をロギングするように構成されている。別の実施形態では、分岐命令に関連付けられたインストルメント化は、分岐命令の実行のその特定のインスタンスにおいて分岐が行われたか否かの命令をロギングするよう構成されてい

10

20

30

40

50

る。

【0071】

図8Aは、基本ブロックを含むソフトウェアプログラムの高レベルの概要を示すブロック図である。実施形態では、ソフトウェアプログラムは、複数の基本ブロックを含むことができる。本明細書で説明するように、基本ブロックは、1つのエントリポイントと1つのイグジットポイントとを有する命令のセットまたはシーケンスである。コンパイラは、1つ以上の基本ブロックを識別するためにソフトウェアプログラムを解析することができる。実施形態では、ソフトウェアプログラム800は、多数の基本ブロックを含むことができる。実施形態では、ソフトウェアプログラムは、最初的基本ブロック802、具体的には基本ブロック1を含むことができる。実施形態では、基本ブロック1は、804における基本ブロック2および810における基本ブロック3に分岐することができる。基本ブロック2はまた、他の基本ブロック808および806につながる多数の他の分岐を含むことができる。

10

【0072】

実施形態では、ソフトウェアプログラムは、以前または先行の基本ブロックを伴って常に行実行されて先行ブロックによって呼び出される基本ブロックのみである、以後または後続の基本ブロックを含むことができる。例えば、図8Bに示されるように、後続基本ブロックB(822)の実行が直ちに、先行基本ブロックA(820)の実行に続くことができる。基本ブロックBは基本ブロックAの唯一の後続であるため、基本ブロックBが実行されたとき、基本ブロックAも実行されていることが既知である。実施形態では、トレースデータログが基本ブロックBからのインストルメント化データを含む場合、基本ブロックAもまた実行されたことが既知であるので、基本ブロックBのみがインストルメント化される。ソフトウェアコードに挿入されるインストルメント化の量を低減するためにシステムがこの情報を利用することができるので、後続基本ブロックを伴って常に行実行される基本ブロックの識別は、有利である。

20

【0073】

図8Cは、実施形態に係る別の基本ブロックインストルメント化最適化を示す。図示されたように、基本ブロックA(830)は、基本ブロックC(834)を直接、呼び出すことができ、または、基本ブロックB(832)を呼び出して、その次に基本ブロックCを呼び出すことができる。トレースデータが基本ブロックCにのみ関連するデータを含む場合、実行中に基本ブロックAは、基本ブロックCを直接、呼び出して基本ブロックBは実行されなかった。トレースデータが基本ブロックBに関連するデータを含む場合、実行中に基本ブロックAは、基本ブロックBを呼び出して、その次に基本ブロックCを呼び出した。いずれの場合においても、基本ブロックAは別個にインストルメント化されない。基本ブロックBおよびCのみのインストルメント化で十分である。

30

【0074】

図8Dは、実施形態に係るさらに別の基本ブロックインストルメント化最適化を示す。図示されたように、基本ブロックC(844)は、基本ブロックB(842)または基本ブロック $N_1 \sim N_n$ (846)のいずれかを介して基本ブロックA(840)から呼び出される。トレースデータが基本ブロックCにのみ関連するデータを含む場合、実行中に基本ブロックAは、基本ブロックCを基本ブロック $N_1 \sim N_n$ を介して呼び出して基本ブロックBは実行されなかった。トレースデータが基本ブロックBに関連するデータを含む場合、実行中に基本ブロックAは、基本ブロックBを呼び出して、その次に基本ブロックCを呼び出した。いずれの場合においても、基本ブロックAおよび $N_1 \sim N_n$ は別個にインストルメント化されない。基本ブロックBおよびCのみのインストルメント化で十分である。

40

【0075】

実施形態では、複数の直後の後続ブロックを伴う基本ブロックは、後続ブロックのうちの1つの第1の命令による継続を実行させる命令で終了する。このような命令を、条件分岐命令と呼ぶことができる。最適化コンパイラには、条件分岐命令の実行後に実行される

50

可能性が最も高いのはどの後続ブロックであるかを決定するための経験則および他の技術を使用することができるものがある。いくつかのCPUアーキテクチャでは、そのようなコンパイラは、CPUキャッシュを最大限に活用するようコードを編成することによって、および／または、CPUに分岐が行われそうかそうでないかを教える条件分岐命令のフォームを生成することによって、および／または他の技術を使用することによって、生成された実行可能ファイルの効率を高めるためにこのような決定を使用することができる。実施形態において、改善バックエンドは、このような分岐予測能力を伴うコンパイラを含むことができ、その能力を、インストルメント化しないためにはどの基本ブロックが最も有益であるかの決定を知らせるために最適化インストルメント化の生成時に使用することができる。

【0076】

図8Dの例では、ブロックA(840)は、2つの直後の後続ブロック：実施形態に係るブロックB(842)およびブロックN₁(846における第1のブロック)を有する。本明細書に記載のインストルメント化最適化を適用することにより、コンパイラは、ブロックA(840)およびB(842)から、またはあるいはブロックA(840)およびN₁~N_n(846)から、インストルメント化を省略することができる。ブロックA(840)の両方の後続ブロックが実行される可能性が等しいと仮定すると、最も効率的なコードは、より多くのブロックを含むパスからのインストルメント化を省略することによって生成される。しかし、実施形態では、コンパイラは、それらの後続ブロックの各々がブロックA(840)の実行後に実行される相対的可能性を予測することができる。例えばコンパイラが、ブロックB(842)がN₁~N_n(846)よりも10倍、ブロックA(840)に続く可能性が高いと予測し、一連のブロックN₁~N_n(846)で実行されるブロックの数「n」が10未満である場合、コンパイラは、インストルメント化を最適化する場合には、ブロックN₁~N_n(846)およびブロックC(844)をインストルメント化してブロックA(840)およびB(842)からのインストルメント化を省略することがより効率的であると決定することができる。これは、実行されるインストルメント化命令の合計期待数、およびターゲットプログラムを実行する際に消費される合計ログ空間が低減されて、ランタイムパフォーマンスへの影響およびインストルメント化のログ空間消費が減少するという点で、有利である。他に対する1つのコードパス実行の可能性が大きくなればなるほど、この技術のおかげで潜在的パフォーマンスが大きくなり、ログが節約される。特定の分岐が1,000回に一度または1,000,000回以上に一度のみ行われることが珍しくないことを考えると、この技術の潜在的な節約はかなりのものである。

【0077】

実施形態ではシステムは、基本ブロックへのエントリを識別するインストルメント化コードを生成して挿入する。例えば、1つ以上のインストルメント化命令を、基本ブロックの先頭に挿入することができる。実施形態では、基本ブロックのエントリポイントの位置は、メモリアドレスによって表され、これは、仮想メモリ位置に対応することができる。実施形態では、インストルメント化は、その後続ブロックが問題の基本ブロックによって常に先行されている一部またはすべての基本ブロックへのエントリを別途、記録しないことによって最適化されることができる。後続基本ブロックへのエントリを記録することは、先行基本ブロックの実行を識別するのに十分であることができる。有利なことに、これは、インストルメント化の結果として生成および記録されたトレースデータのサイズを減らすだけでなく、ターゲットプログラムに挿入する必要があるインストルメント化コードの量またはボリュームを減らして、それによりターゲットプログラムの実行パフォーマンスへの影響を低減することができる。ソフトウェアコードに挿入されるインストルメント化の量を低減することにより、システムはより効率的にターゲットソフトウェアをデバッグすることができる。ターゲットソフトウェアに挿入されたインストルメント化コード要素の数を減少させることによって、より少ないトレースデータが生成される。生成されるトレースデータのサイズ減少、および／または、インストルメント化コード量の減少によ

10

20

30

40

50

り、ターゲットコンピュータプログラムの実行についてのインストルメント化の影響を低減し、停止条件につながるイベントの再生／再構成に必要な時間も改善することができる。

【0078】

実施形態は、実行パスをトレースするためのインストルメント化の一部または全部を省略し得る。このような省略は、トレースされたプログラムのランタイムパフォーマンスを改善し、トレースされたプログラムのコードサイズを低減し、および／または、より少ないログデータを使用することができる。このような場合、実施形態は、「再構成レジスタ状態」のセクションに記載されているものなどの技術を使用して実行パスを再構成し得、これは、プログラムの実行の一部またはすべての実行パスの実行、または、実行シミュレートを行うために、本明細書に記載されるような「再構成シミュレーション」技術を使用する。レジスタ状態などの状態情報のうち、このような技術により再構成されるのが、プログラムカウンタ（PC）である。再構成中に実行される各命令の後のPCの記録により、プログラムの一部の実行パスが有効に再構成される。実施形態はまた、所望の停止条件のためにバイナリサーチを実行するために再構成シミュレーションを使用するなど、実行パスを再構成するための他の技術を採用し得る。様々な条件に応じて、そのような技術は、より少ないメモリを消費し、および／または、より迅速に所望の停止条件を発見するという点で有利であり得る。

10

【0079】

《メモリ修正のためのトレースデータ》

実施形態において、改善バックエンドは、最新から最古までなど、データが時間的に逆方向にデコードされることを可能にするような方法でトレースデータをロギングすることができる。この目的のために、改善バックエンドは、変更されようとしているメモリの内容（「プリイメージ」）をロギングすることができる。プリイメージロギングの1つの方法が図18Aである。この例では、プリイメージがロードされ、そしてロギングされる。修正されたデータのアドレス、修正されたデータの長さ等などのメタデータが、同時にロギングされ得る。最後に、メモリの修正が行われる。プリイメージロギングの他の方法が、ロギングが行われる環境に応じて使用され得る。

20

【0080】

プリイメージロギングは、コンピュータプログラムがストア命令でメモリを修正する前に、プリイメージをロギングするように構成されたコンピュータプログラム内でインストルメント化することにより行うことができる。このようなプリイメージロギングは、CPUレジスタにプリイメージをロードし、（任意選択的に）追加メタデータとともにログにCPUレジスタの内容を書き込んでストア命令でメモリ修正を行う形態をとることができる。CPUアーキテクチャに応じて、例えば、CPUアーキテクチャがCPUレジスタの値を用いてメモリ内の値をスワップするメモリ交換命令を有する場合、他の技術が可能であり得る。

30

【0081】

コンピュータプログラム外部のオペレーティングシステムまたは他のエージェントがプログラムのメモリを修正する場合、プリイメージロギングは、コンピュータプログラム、ライブラリ、オペレーティングシステムまたは他の場所におけるインストルメント化によって行われ得る。例えば、プログラムは、POSIX「読み出し」コールなどのプログラムのメモリを修正するシステムコールを行うことができる。オペレーティングシステムが「読み出し」コールにおいてメモリ修正を行う前に、オペレーティングシステムは、プログラムに代わって、上書きされようとしているメモリのプリイメージをロギングすることができる。別の例として、プログラムは、ハードウェアデバイスからプログラムのメモリ空間へのダイレクトメモリアクセス（DMA）を要求することができる。DMAを要求する前に、プログラムは、DMAによって上書きされる可能性があるメモリのプリイメージを保存することができる。

40

【0082】

いくつかの実装では、プリイメージロギング手順が、コンテキストスイッチまたはPO

50

SIX信号などの外部イベントによって中断される可能性があり得る。このような場合には、実行が中断コードに戻ってメモリ修正が行われる場合、メモリから読み出された、および／またはロギングされたプリイメージはもはや正確でないかもしれない。この場合、特別なアクションがとられなければ、ロギングされたプリイメージ値が正しくない可能性があり、そのような不正データを使用しようとする時間走行デバッガが不正に動作する可能性がある。この問題の例を図18Bに示す。中断(X4b01と標識されたボックス)が、もはや正確でない、以前にロードされたプリイメージ(X4b02)をもたらし、不正データがロギングされる(X4b03)ことに留意されたい。具体的には、ロギングされたプリイメージデータ(X4b03)は、実際に上書きされるメモリと一致しない(X4b04)。図には示さないが、プリイメージがロギングされた(X4b03)後であってメモリ修正が発生する(X4b04)前に割り込みが発生した場合、同じ問題が発生することに留意されたい。この問題に対処するための技術のいくつかの例は、本明細書に記載されている。

10

【0083】

実施形態において、改善バックエンドは、プリイメージロギングおよびメモリ修正のアトミック性を保証するために「再始動可能ウィンドウ」を使用することができる。再始動可能ウィンドウとは、コードのセクションが中断された場合に、中断が完了した後、プログラム、ライブラリ、オペレーティングシステムまたは他の場所における機構が中断コードを以前のポイントから再開させる技術を指すことができる。これは、図18Cに示されている。中断(破線矢印)がプリイメージのロード(X4c02)とメモリ修正の実行(X4c04)との間の任意のポイントで発生する可能性があり、他のコードの実行(X4c01)およびロードされたプリイメージ値の無効化(X4c02)となることに留意されたい。他のコードが完了すると、実行がインストルメント化の開始(X4c02)に戻ることができる(X4c05)。このような場合には、不正データが中断前にロギングされたかまたは部分的にロギングされた場合、ランタイム時にログからデータを除去することができ、または、ロギングされたデータの消費者が条件を検出して不正なまたは部分的にロギングされたデータを無視することができる。

20

【0084】

再始動可能ウィンドウを含む実施形態では、インストルメント化が完全に実行されるまでログポインタが進んでいないようにインストルメント化を構成することによってランタイム時に部分的にロギングされたデータの除去が達成される。このような場合、中断されたコードの再始動時にそれが新たなデータでログの同じセクションを上書きする、または、ログのその部分が続いて別の目的のために使用されている場合、それはログの新たなセクションに新たなデータを書き込むであろう。

30

【0085】

再始動可能ウィンドウを含む実施形態では、中断の近傍でアプリケーションコードをディスアセンブルすることにより、インストルメント化のパターンまたはインストルメント化の進行を示す他の命令を探すことにより、または、プログラムのコンパイルおよび／または解析に関与するコンパイラまたは他のツールにより生成されるメタデータを調べることにより、部分的にロギングされたデータをどのくらい除去しなければならないかを決定することができる。

40

【0086】

いくつかの実装では、改善バックエンドは、中断されたプリイメージロギングに対処するために、以下の技術を使用することができる。オペレーティングシステム、ライブラリ、または他のコンポーネントは、割り込みからの復帰時に、正しいデータを含む特別なレコード(以下「フィックスアップレコード」)をロギングすることができる。そのようなレコードを検出すると、ロギングされたデータの消費者は、メモリストアに関連付けられている任意の誤ったまたは部分的にロギングされたデータを無視するように構成されることができる。この例は、図18Dに示されている。割り込みが完了すると、新たなプリイメージを、中断されたインストルメント化に戻ってメモリ修正を行う前にロギングするこ

50

とができる（X 4 d 0 5）。図 1 8 B におけるように、この例ではプリイメージのロード（X 4 d 0 2）直後の中断のみが示されているものの、この技術は、プリイメージのロード（X 4 d 0 2）とメモリ修正の実行（X 4 d 0 4）との間の任意のポイントで発生する割り込みについて作用する。

【0 0 8 7】

実施形態では、フィックスアップレコードは、中断がプリイメージのロードおよびメモリ変更レコードのロギングの間に発生したことをシステムが検出したときにロギングされるのみである。そのような検出は、中断の近傍でのアプリケーションコードのディスアセンブリ、インストルメント化のパターンまたはメモリ修正を示す他の命令の探索、または、プログラムのコンパイルおよび／または解析に参与するコンパイラまたは他のツールで生成されるメタデータの調査によって達成することができる。

10

【0 0 8 8】

実施形態では、フィックスアップレコードは、中断が発生した時点に関係なくロギングされることができ、フィックスアップレコードがプリイメージのロードとメモリ変更レコードのロギングとの間に発生したかどうかを後で決定することができる。（人またはプログラムであり得る）フィックスアップレコードの消費者がフィックスアップレコードは異なる時間にロギングされたと判断する場合、フィックスアップレコードを無視することができる。中断が発生した時点にかかわらずフィックスアップレコードをロギングする技術は、フィックスアップレコードを条件付きでロギングするよりも優れたランタイムパフォーマンスを持つことができ、それはフィックスアップレコードが必要であるかどうかを判断するためにディスアセンブリまたは他の技術を適用する必要があるからである。どの技術がより有益であるかは、CPUアーキテクチャ、インストルメント化の構造、オペレーティングシステムの特長（適用可能な場合）および他の要因に依存する可能性がある。

20

【0 0 8 9】

《留保トレースデータ最適化》

プリイメージログインストルメント化は、本明細書に記載されるように、いくつかの実施形態では、基本ブロック内の各メモリストアオペレーションについて、影響を受けたメモリのアドレス、プリイメージ、ストアの幅等などの情報をロギングすることができる。しかし、ロギングされた情報の消費者は、他のソースからのこの情報のサブセットが、情報のサブセットが冗長であってロギングの必要がない場合に当たるかを決定し得る。不要なデータをロギングしないようにインストルメント化を最適化することは、不要なロギング命令を排除することでランタイムパフォーマンスを向上させることができ、ログ空間のより効率的な使用につながるという点で有利である。本明細書中に記載の留保トレースデータ最適化は、このような最適化である。

30

【0 0 9 0】

改善バックエンドにおけるコンパイラは、同じメモリアドレスへのメモリアクセスのブロックを識別するためにターゲットプログラムを解析し、そのようなブロックについて、アクセス中のメモリアドレスを記録する1つ以上のインストルメント化命令を挿入することによりインストルメント化を最適化するように構成されることができる。他のインストルメント化命令は、関連データのみを記録する必要があり、メモリアドレスは不要である。例えば、ターゲットプログラム命令のセットが、アドレスAを有するメモリ位置へのアクセスをN回（Nは1、2、3、4…などの整数である）行ったとする。アドレスAへの最後のメモリアクセスについて、コンパイラは、関連するデータとともにアドレスAをトレースデータに記録するように構成されたインストルメント化命令を生成することができる。残りのN-1回のアドレスAへのアクセスについて、コンパイラは、関連データのみを記録するがアドレスAを記録しないインストルメント化命令を生成するように構成することができる。このような最適化は有利には、トレースデータのサイズを小さくすることができる。実施形態では、複数のメモリアドレス（AおよびB、A、B、ならびにCなど）が例えば、レジスタに保存され得る同じベースアドレスからの異なるオフセットによりアクセスされる場合に、このタイプの最適化が行われる。ベースアドレスは一度だけ記録

40

50

され、後続のメモリ位置アドレスはそれらのオフセットのみにより記録される。実施形態では、このタイプの最適化は留保トレースデータ最適化と称される。

【0091】

実施形態では、コンパイラは、基本ブロックの先頭の代わりに基本ブロックの最後に、基本ブロックの実行を識別するインストルメント化コードを挿入するように構成されることができる。このように、時間的後方へと実行される再生および／または再構成を最適化することができ、それは、基本ブロックを表すレコードがメモリストアを表すレコードおよび／またはその基本ブロック内で発生した他のイベントの前に遭遇することになるからである。再生および／または再構成中のトレースデータのデコードをさらに高速化するために、コンパイラは、基本ブロックの最後に関連付けられたプログラムカウンタの値を保存する命令に加えて、特定の値でレジスタをロードするなどの1つ以上の追加の命令を挿入することができる。1つ以上の追加命令を、（ノーオペレーションすなわちNOP命令実行と同様に）副作用がない命令とすることができる。システムが再生および／または再構成中にこのような1つ以上の追加命令を検出すると、これはトレースデータに保存されるプログラムカウンタ（または別のタイプの一意識別子）が基本ブロックの最後についてのものであることをシステムに示すであろう。システムはそして、基本ブロックの先頭プログラムカウンタ（PC）を発見してトレースデータにそれを挿入し、それにより、先頭PCが元のトレースデータに記録されたトレースデータデコードにそれが現れる。実施形態では、このような動作は留保トレースデータ最適化と称され、これは、その通常位置におけるPC値の挿入が、トレースデータがデコードされるまで留保されるからである。

【0092】

実施形態では、このようなシステムは有利には、記録トレースデータの量を減らすことができる。PCインストルメント化が基本ブロックの最後に挿入される場合、その前のプリライトメモリ値は多くの場合、サイズ低減されることができ、それは、このようなエントリが、記録および保存されたPC値から自身を区別するために追加注釈を必要とせず、システムが、それらがプリイメージ（またはプリライト）メモリ値であると想定することができるからである。対照的に、PCインストルメント化が基本ブロックの先頭に記録されている場合は、トレースデータを後方にデコードすると、基本ブロックについて記録されたプリイメージメモリ値に最初に遭遇する。したがって、これらの値は、PC値から区別するために追加注釈を必要とする。

【0093】

実施形態では、図9に示されているように、留保トレースデータ最適化が以下のように行われる。トレースデータフラグメント930は、基本ブロックの最後にPCを保存するエントリ932を含む。エントリ934は、（アドレスa+8への書き込み前の）メモリアドレスへの書き込みに対応し、プリイメージ値とアドレス（すなわち「a+8」）とを含む。エントリ936および938は、メモリアドレスへの書き込みに対応するが、プリイメージ値のみを含み、アドレスを含まない。再生および／または再構成中、システムは、トレースデータフラグメントをデコードするためのテンプレートまたはマップを含む特別セクション940を利用することによってトレースデータフラグメント930をデコードするであろう。このようなセクションは、本明細書に記載されるように、静的ルックアップサイドテーブルの一例とすることができる。特別セクション内のエントリ942は、トレースデータ内のエントリ932に関連付けられ、PCの値がトレースデータに保存されていることを示している。特別セクション内のエントリ944は、トレースデータ内のエントリ934に関連付けられ、書き込まれたアドレスとともに「フルプリイメージ」またはプリライト値がトレースデータに保存されていることを示している。特別セクション内のエントリ946は、トレースデータ内のエントリ936に関連付けられ、プリライト値のみがトレースデータに保存されていることを示している。書き込まれたアドレス（すなわち「a+4」）は、トレースデータに保存されないが、特別セクション内のエントリ946から再構成される。実施形態ではアドレスは、トレースデータにてエントリ934に保存されたアドレス「a+8」から暗示される（例えば、「-4」の相対オフセットが特別

セクション内のエントリ 946 に保存され、これは、アドレス「 $a + 8 - 4$ 」すなわちより単純に「 $a + 4$ 」に対応する）。特別セクション内のエントリ 948 は、トレースデータ内のエントリ 938 に関連付けられ、プリライト値がトレースデータに保存されていることを示している（しかし、「 -8 」の相対オフセットなどの暗黙アドレスが特別セクション内のエントリ 948 に保存され、これは、アドレス「 $a + 8 - 8$ 」すなわちより単純に「 $a + 0$ 」または「 a 」に対応する）。例えば、特別セクション内のエントリ 948、946 および 944 は、プログラムがアレイまたはこのような一連の近くのメモリアドレスへの書き込み中であつたことを示し、それは、連続したメモリ位置（「 a 」「 $a + 4$ 」および「 $a + 8$ 」）が、（ベースアドレスであるアドレス「 a 」とともに）書き込み中であつたからである。再生および／または再構成の間、トレースデータをデコード（または解凍）し、欠落情報を挿入する。例えば、アドレス $a + 0$ および $a + 4$ は、特別セクション内のエントリ 946 および 948 に保存された暗黙アドレスを使用して、トレースデータに挿入されるであろう。このように、すべてのアドレスが元のトレースデータに記録されたことになるであろう。

【0094】

実施形態では、留保トレースデータ最適化は、連続メモリ位置への書き込みの場合に限定されるものではない。他の実施形態では、特定のデータは、デバッグ中のタスクの実行中にトレースデータログから省略され得る。例えば、最後のプリイメージメモリ値のアドレスのみを記録することができ、定数であるかまたは最後のアドレスからのオフセットとして表すことができる基本ブロックによって書き込まれる他のすべてのアドレスを、セクション 940 などの特別セクションに記録する。欠落データは、タスクの実行が停止された後、後でトレースデータログに挿入されることができる。

【0095】

実施形態では、セクション 940 は、静的ルックアサイドテーブルであり、ELF（実行可能ファイルおよびリンク可能フォーマット）ファイルのセクションに含まれる。ELF ファイルのセクション 940 は、インストルメント化実行可能コードとともにターゲットコンピュータシステムにダウンロードされない。むしろ、ELF ファイルのセクション 940 は、スキップされたトレースデータをデコードまたは補充するために、再生および／または再構成中に使用される。有利には、留保トレースデータ最適化は、トレースデータログのサイズを低減し、コンピュータプログラムの実行前にターゲットコンピュータシステムに転送されるデータのサイズを低減する。

【0096】

《副作用メモリ》

いくつかの場合、特定のメモリ位置（「副作用メモリ」と称することができる）からプリイメージ値をロードしようとすることは、望ましくない効果を有し得る。例えば、特定のメモリマップ化レジスタは、読み出し時にハードウェアデバイスの状態に影響を与える可能性がある。副作用メモリへのストアが行われた場合、改善バックエンドにおけるコンパイラは、プリイメージを読み出さない代替インストルメント化を生成するように構成することができ、したがって、任意の望ましくない副作用を回避する。このようなインストルメント化は、例えばストアのアドレスをロギングすることにより、通常の情報サブセットをロギングすることができる。プリイメージを除いて、通常情報をロギングすることにより、プリイメージに依存しない改善バックエンドおよび時間走行デバッグは、副作用メモリ上で動作しているときに、依然として正常に機能することができる。例えば、時間的に前方または後方への移動を停止する条件内でも、このようなストアを依然として使用することができる。

【0097】

いくつかの場合において、所与のメモリストアが副作用メモリに影響を与えるか否かをコンパイル時に決定することが可能ではないかもしれない。例えば、ポインタを介したメモリストアは、ストア命令が実行されるたびに変わる可能性があるポインタによるポイント箇所に応じて、いずれかの種類のメモリに影響を与える可能性がある。このような場合

10

20

30

40

50

には、改善バックエンドにおけるコンパイラは、影響を受けたメモリが副作用であるか否かをランタイム時に決定するインストルメント化を出力するように構成されることができる。このような決定は、既知の安全および／または既知の非安全メモリ範囲に対してストアアドレスを比較することにより、または他の試験を行うことにより行うことができる。実施形態では、改善バックエンドが所与のメモリストアが副作用メモリに影響を与えるか否かを判断することができなければ、プリイメージをロードせず、これは、デバッグ機能の低下が一般的に、ターゲットハードウェアの破壊、CPUのクラッシュまたは他のこのような負の影響の発生の危険を招きやすいからである。

【0098】

実施形態では、改善バックエンドにおけるコンパイラは、ストアが副作用メモリに影響するか否かについてのコンパイラの決定に影響を与えるために、ソースコードにおける特別なキーワード、またはコマンドラインオプション、または命令の他の形態をプログラマから受け入れるように構成されることができる。このような機構により、プログラマは、コンパイラのデフォルト挙動がサブ最適または不正であるかもしれない場合にコンパイラのデフォルト挙動を上書きできる。

レジスタ状態スナップショット

【0099】

CPUのレジスタ状態を改善バックエンドによって記録することができる多くの方法がある。実施形態では、レジスタの値が変更されるたびにレコードをログに書き込むことができる。そのようなレコードは、レジスタが修正されたときに上書きされる値などのプリイメージ値を含むことができる。しかし、レジスタ状態はコンピュータプログラムの実行中に頻繁に変化する傾向があるので、この技術は、プログラムへの非常に多数のインストルメント化命令の挿入を含み得、これは、プログラムのランタイムパフォーマンスおよび／またはメモリフットプリントに悪影響を与える可能性がある。

【0100】

実施形態では、改善バックエンドは、定期的に1つ以上のCPUレジスタ状態の「レジスタ状態スナップショット」を記録することができる。これらのレジスタ状態スナップショットは、アプリケーションによって、ライブラリコードによって、オペレーティングシステムによって、またはいくつかの他のコンポーネントによって、記録することができる。このようなスナップショットから様々な時点でのCPUレジスタ状態を再構成するための技術は、「レジスタ状態再構成」のセクションに記載されている。レジスタ状態スナップショットは、レジスタのみを含むことに限定されない。レジスタ状態スナップショットは、プログラムメモリ、タイムスタンプ、スレッド情報、アドレス空間情報、または任意の他のデータの部分の表現などの他の状態またはメタ情報を含むことができる。このようなデータは典型的には、スナップショットの他の内容の使用を、増強し、識別し、または容易にするが、他の目的にも供することができる。

【0101】

本明細書に記載のレジスタ状態スナップショットのロギングは、いくつかの実施形態では、個別にすべてのレジスタ変更をロギングするよりもランタイムパフォーマンスおよびメモリフットプリントにはるかに小さい影響を有するように構成されることができる。いくつかの実装では、本明細書に記載のレジスタ状態スナップショットをロギングする改善バックエンドは、バックエンドのコピーよりもトレース中のシステムのランタイムパフォーマンスにはるかに小さい影響を有することができる。コピーバックエンドは、各チェックポイントでのアプリケーションのメモリ空間およびレジスタ状態の一部またはすべてをコピーし、その影響は、アプリケーションによって使用されているメモリの量に比例する可能性がある。対照的に、改善バックエンドは、アプリケーションのレジスタ状態の一部またはすべてをコピーするのみであり、これは、CPUレジスタの数によってバウンディングされ、ランタイムパフォーマンスへの影響をアプリケーションによって使用されているメモリの量とは無関係にする。さらに、CPUのレジスタに含まれるデータの量は一般的に、アプリケーションによって使用されているメモリに含まれるデータの量よりもはる

10

20

30

40

50

かに小さく、よって一般的には、改善バックエンドにより各チェックポイントにてロギングされるデータの量はコピーバックエンドによるよりもはるかに小さく、ランタイムパフォーマンスへの影響およびログ使用量をはるかに低減する。

【0102】

改善バックエンドは、実行のスレッドが実行開始または停止するとき、外部変更がレジスタ状態になされるとき（例えば、オペレーティングシステムがシステムコールの一部としていくつかのレジスタを変更するとき）、各種の割り込みが発生したとき、および、スナップショットが最近ロギングされていないときなどに、レジスタ状態スナップショットを記録することができる。状況に応じてスナップショットは、完全レジスタ状態（「フルスナップショット」と称することができる）、またはサブセットのみを含むことができる。

10

【0103】

スナップショットを記録するための改善バックエンドのタイミングを構成する際の1つの考慮事項は、CPU状態を再構成するために必要な時間に影響を与えることである。実施形態では、「レジスタ状態再構成」のセクションに記載されているものなどのアルゴリズムを使用して、CPU状態を再構成するために必要な時間をバウンディングするために、改善バックエンドは、スナップショットロギングをスケジュールすることができる。スナップショットベースレジスタ再構成アルゴリズムは典型的には、線形時間で実行され、アルゴリズムを実行するために横断しなければならないログの量に比例する。レジスタ再構成を実行するために横断しなければならないログの量をバウンディングすること、したがって、レジスタ再構成にかかる時間の量をバウンディングすることにより、改善バック

20

【0104】

実施形態では、改善バックエンドは、レジスタ状態スナップショットのロギングのための次のスケジュールを使用してレジスタ再構成時間をバウンディングすることができる。外部エージェント（例えばオペレーティングシステム）が実行の特定のスレッドに属しているレジスタを修正する場合、少なくとも変更したレジスタを含むスナップショットのロギングは、変更がログに反映されることを保証する。さらに、実行のスレッドが実行を停止したときのフルスナップショットのロギングは、実行のスレッドが実行されていないログ内のすべてのポイントで、遭遇した最新のスナップショットが正確なレジスタ状態を含んでおり、何の再構成も必要とされないということを保証する。実行のスレッドが実行されている時点でレジスタ再構成時間をバウンディングするために、実行のスレッドのレジスタ状態は、定期的にロギングされる必要があり、それにより、レジスタ状態スナップショット間でロギングされたトレースデータの量がバウンディングされる。実施形態では、これは、ログ空間の特定のバウンディングされた量が消費されるたびにレジスタ状態スナップショットをロギングすることにより達成することができる。

30

【0105】

《オペレーティングシステムイベント》

実施形態では、改善バックエンドは、プログラマが関心のあり得るオペレーティングシステムイベントをロギングすることができる。例としては、実行のスレッドおよび／またはアドレス空間の作成および破壊、実行のスレッドおよび／またはアドレス空間の間の通信、システム上で実行中のアプリケーションのパフォーマンスに影響を与える可能性がある、および／または、データの到着もしくは関心のあるなんらかの他の外部イベントを示す可能性があるものを含むハードウェア割り込み、実行のスレッドのスケジューリングに影響を及ぼす、および／または、アプリケーションレジスタまたはメモリ状態を修正するオペレーティングシステムイベント、仮想メモリのマッピングおよび／またはマッピング解除、ミューテックス、セマフォなどの同期プリミティブの使用が挙げられる、これらに限定されない。

40

【0106】

時間走行デバッガおよび／または他のユーザインタフェースコンポーネントは、プログ

50

ラマによるシステム挙動の全体的な理解を助けるためにプログラマにそのようなイベントを表示することができる。そのようなイベントが、例えばそのメモリまたはレジスタ状態を変更することによってアプリケーションに影響を与える場合、改善バックエンドは、例えば、時間内移動についての終結条件内でこのようなイベントをユーザが特定できるようにすることによって、このような変化に起因するバグを検出および理解するプログラマの能力を向上させるためにそのような情報を使用することができる。このような機能は、例えば、メモリ破損の原因を発見することにおいて便利である。改善バックエンドを採用しない時間走行デバッガは、アプリケーション自体によって開始された変化を検出するのみなど、メモリ変更の原因を突き止める能力を、あるとしても、限定的にしか有し得ない。対照的に、改善バックエンドを採用する時間走行デバッガは、これらに限定されないが、アプリケーション自体、それ自身に作用するオペレーティングシステム、別のアプリケーションまたは実行のスレッドのために作用するオペレーティングシステム、DMAデータの到着もしくは他のCPU上で実行中のスレッドおよびオペレーティングシステムのアクションなどのオペレーティングシステムが見ることのできる外部イベントを含む、より広い範囲のソースによって引き起こされるメモリ変更を検出することができる。

10

【0107】

《非決定論的命令ロギング》

アプリケーション内外で発生するメモリおよびレジスタ変更のロギングのための本明細書に記載された技術は、その実行時のアプリケーションの状態へのほとんどの変更を捕捉するのに十分である。しかし、改善バックエンドが様々な時点でシステムの状態を正確に再現するように別個に取り扱うことができる捕捉されない付加的な場合がある可能性がある。

20

【0108】

（「非決定論的命令」と称され得る）いくつかのCPU命令は、以前の状態から予測または推測することができない結果を生成する可能性がある。例えば、CPU命令は、DMAによって修正される可能性があるメモリマップされたレジスタもしくはメモリのセクション、または、その行動がロギングされていないもしくは別様に未知である別のアプリケーションと共有されるメモリなどの、以前の状態から知るまたは推測することができないメモリ位置の内容をレジスタにロードすることができる。別の例では、CPU命令が、クロックもしくは乱数発生器から読み出され、または、以前の状態から予測もしくは推測することができない結果を別様に生成することができる。別の例では、CPU命令の結果は、非同期的に動作するシステムの他の部分などの非決定論的ソースの挙動によって影響を受ける可能性がある。このような場合には、CPU命令の結果が非決定論的であるときに、改善バックエンドにおけるインストルメント化は、命令の効果をロギングすることができる。改善バックエンドは、メモリおよびレジスタ状態の再構成を助けるために、後の時点でこのような情報を使用することができる。

30

【0109】

《メモリへのロギング》

実施形態では、トレースデータの一部または全部は、ターゲットコンピュータシステムの（RAMなどの）メモリに保存することができる。実施形態では、トレースデータの一部または全部は、コンパイラを実行するコンピュータシステムなどの別のコンピュータシステムのメモリに記憶することができる。実施形態では、トレースデータのストレージを、ターゲットコンピュータシステムおよび別のコンピュータシステムのメモリにわたって分割することができる。

40

【0110】

実施形態では、改善バックエンドは、RAMなどのメモリに情報をロギングするインストルメント化を含むことができ、それは、インストルメント化が実行されるCPUによって書き込み可能である。メモリが限られた資源であり、ロギングできるデータ量に最大値があることを考えると、改善バックエンドはそれを最大限に活用するために利用可能なRAMを管理するための戦略を採用することができる。そのような戦略の一例は、本明細書

50

に記載の単一化ログである。

【0111】

単一化ログを、1つ以上のログにメモリを割り当てるために（「単一化ログメモリプール」と称され得る）利用可能メモリの1つ以上のプールから引き出すことができる。例えば、単一化ログは、RAMの2つの500メガバイトプールを取り、カーネルログおよび6つのアプリケーションログの間でそれらを割り当てることができる。ログの数は、例えば、実行中のインストルメント化アプリケーションの数の変更を提供するために、時間の経過とともに変化する可能性がある。単一化ログは、利用可能なメモリを1つ以上のプールから（「チャンク」と称され得る）部分へと分割し、これは、1つ以上のログの使用のために配分されている。このようなチャンクはすべて同じサイズであり得、または異なるサイズであり得る。

10

【0112】

各ログは、書き込まれるべき次のエントリの位置を特定するための機構（「ログポインタ」と称され得る）、および、単一化ログによって割り当てられた現在のチャンクの最後を識別するための機構（「ページ制限機構」と称され得る）を含むことができる。このような機構の一例は、現在、割り当てられているチャンクのバウンドを含むデータ構造、および、ログ内の次の未書き込みエントリへのポインタである。

【0113】

単一化ログは、各ログにチャンクを割り当てることによって開始することができる。いくつかの実施形態では、この初期割り当ての一部ではない単一化ログのプール内のメモリは、本明細書に記載されるように、後の使用のために利用可能なままであることができる。チャンクのこの初期割り当ては典型的には、それに応じて各ログについてのログポインタおよびページ制限機構を初期化することを含むことができる。

20

【0114】

初期チャンク割り当ての例が、図26Aに現れる。この例では、単一化ログは、3つのログ、ログ1（X13a-Log1）、ログ2（X13a-Log2）、およびログ3（X13a-Log3）を管理する。単一化ログは、単一化ログ（X13a-MP）に割り当てられたメモリプールからこれらのログのそれぞれに初期チャンク（X13a-c1、X13a-c2およびX13a-c3）を割り当てる。ログがひとたび現在のチャンクを有すると、現在のチャンクを、例えばそれらへのインストルメント化書き込みおよびログポインタ更新によって埋め込むことができる。ログのページ制限機構が現在のチャンクが埋め込まれていることを検出した場合、単一化ログは埋め込まれたチャンクをアーカイブ化して新たな現在のチャンクをログに割り当て、それに応じてログのログポインタおよびページ制限機構を更新する。埋め込まれたチャンクをアーカイブ化することは典型的には、埋め込まれたチャンクの位置、そのデータがそれに含まれているログのアイデンティティ、同じログで埋め込まれた他のチャンクに対するそれが埋め込まれた順序、単一化ログが管理するすべてのチャンクに対するそれが埋め込まれた順序、を追跡し続けることを含む。ログの現在のチャンクがひとたびアーカイブ化されると、単一化ログは、ログに新たな現在のチャンクを割り当てることができる。メモリが単一化ログのメモリプールで依然として利用可能である場合、新たな現在のチャンクをメモリプールから割り当てることができる。単一化ログのメモリプールが使い尽くされている場合は、いくつかの実施形態における単一化ログは、本明細書に説明するように、以前に埋め込まれたチャンクをリサイクルすることができる。

30

40

【0115】

埋め込まれたチャンクのアーカイブ化の例が図26Bに表示される。例では、ログ2（X13b-Log2）が現在のチャンクを埋め込み、そしてアーカイブされている（X13b-ac）。例では、メモリは依然として単一化ログのメモリプール（X13b-MP）で入手可能であり、よって、新たなチャンク（X13b-nc）がログ2（X13b-Log2）に追加される。ログ1（X13b-Log1）および3（X13b-Log3）およびそれらのそれぞれの現在のチャンク（X13b-c1、X13b-c3）は影響

50

されない。

【0116】

いかなるメモリもプール内に残っていない場合、単一化ログは、既に埋め込まれたチャンクをリサイクルすることができ、これらのチャンク内のデータが上書きできる。改善バックエンドによって採用される場合、単一化ログは、改善バックエンドによって使用可能なログの量を最大化する目的でリサイクル戦略を採用することができる。実施形態では、改善バックエンドは、最新のデータで始まる、すべてのログからの連続したデータを必要とする。そのような改善バックエンドの実施形態を最高にサポートするために、単一化ログは、少なくとも最近アーカイブ化されたチャンクをリサイクルすることができる。ログ間のメモリ使用率の差を考慮しながら、このリサイクル戦略は、改善バックエンドに利用できる使用可能なログを最大化する。実施形態では、少なくとも最近、ログに追加されたチャンクをリサイクルするなどの他のリサイクル戦略を使用することができる。このような他のリサイクル戦略は、究極的にはシステム停止時に利用可能であるデータに対する異なる特性を有し得る。

10

【0117】

チャンクリサイクルの例が図26Cである。例では、単一化ログは、3つの別個のログ(X13c-Log1, X13c-Log2およびX13c-Log3)をサービス中である。各ログは、それぞれの現在のチャンクを有し、X13c-Log1はX13c-c1を有し、X13c-Log2はX13c-c2を有し、X13c-Log3はX13c-c3を有する。各ログはまた、後にリサイクルされたチャンクに以前に保存された、失われたデータ(X13c-ld1, X13c-ld2, X13c-ld3)のボディを有する。例では、ログ1(X13c-Log1)は単にその現在のチャンク(X13c-c1)を埋め込み、新たなチャンク(X13c-nc)を必要としている。単一化ログのメモリプール(X13c-MP)が使い尽くされ、よって、単一化ログは、以前に使用されたチャンクをリサイクルする必要がある。ログ1(X13c-Log1)に新たな現在のチャンク(X13c-nc)を与えるために、単一化ログは、リサイクル化チャンク(X13c-rc)となるよう少なくとも最近、アーカイブ化されたページを発見し、これは、新たなチャンクを受容するログを含む任意のログの一部となることができる。この例では、リサイクル化チャンク(X13c-rc)がログ2(X13c-Log2)の一部である。単一化ログは、(任意選択的に)リサイクル化チャンク(X13c-rc)をクリアし、それをログ1(X13c-Log1)についての新たな現在のチャンク(X13c-nc)とする。以前にリサイクル化チャンク(X13c-rc)に保存されたデータは失われ、ログ2の失われたデータ(X13c-ld2)の一部になる。リサイクルが完了するとすぐに、ログ1(X13c-Log1)は、その新たな現在のチャンク(X13c-nc)の埋め込みを開始することができる。リサイクル戦略は、残りのデータが改善バックエンドで使用可能なログの量を最大にすることを確実にする。

20

30

【0118】

なお、本明細書に記載のリサイクル戦略を採用する実施形態では、1つの実行のスレッドがデバッグのために停止された場合、または、エラーが発生したため、または他の何らかの理由で、同じ単一化ログを共有する実行のスレッドが実行を継続できるようにすることは有害であり得、これは、停止スレッドによって書き込まれたデータを含むチャンクのリサイクルをもたらし得、停止スレッドをデバッグするために有用であり得るデータを上書きするからである。このような実施形態では、1つの実行のスレッドの停止が同様に停止中のシステムにおける他の実行のスレッドの一部または全部となる実行制御戦略を採用することが有利である可能性がある。改善バックエンドは、このような実行制御戦略(「同期実行制御」)を含むことができる。

40

【0119】

必要な機能を伴うCPU上で、単一化ログを、CPUの仮想メモリ機能を使用して実装することができる。このような実施形態では、各ログの現在のチャンクを、仮想アドレス空間内で固定されたウィンドウにマッピングすることができる。このようなウィンドウを

50

「ロギングアパーチャ」と呼ぶことができる。このような実施形態では、ページ制限機構を、ログが書き込まれる方向にロギングアパーチャに隣接する仮想メモリアドレス（複数可）をマッピングされないままとすることによって実装することができ、これにより、ロギングアパーチャのエッジを越えて書き込もうとすると、ページフォールトとなる。このような場合、単一化ログは、ページフォールトハンドラがトリガされたときに適切なログについてのチャンクをアーカイブ化するように構成されることができる。有利なことにこのような実施形態において、ターゲットコンピュータプログラムの視点からは、トレースデータを保存するための空間を割り当てるすべての詳細はオペレーティングシステムによって取り扱われるため、ログはトレースデータを保存するための無限の能力を有しているように見える。これは、ターゲットコンピュータプログラムの実行へのインストルメント化の影響を低減することができ、ターゲットコンピュータプログラムの再生および／または再構成に関連する時間を減らすことができる。このような実施形態では、インストルメント化は、現在のログチャンクの境界を越える書き込みをチェックする命令を含む必要がないかもしれない。これらの境界チェックを削除すると、インストルメント化のランタイムコストを削減することができ、インストルメント化のフットプリントを減らすこともでき、コードサイズおよび速度の1つまたは両方を改善する。

【0120】

いくつかの実施形態では、ロギングアパーチャの使用は、本明細書に記載されるように、他の方法でも有利である。合計ログサイズは、仮想メモリのサイズよりも大きくすることができる。例えば、仮想アドレスが32ビット幅であるコンピュータシステムは、仮想アドレスを使用して、RAMの4ギガバイトまでアドレスアップすることができる。ロギングアパーチャを伴わなければ、RAMベースロギングを使用するこのようなコンピュータシステムは、4ギガバイト未満のログに制限される可能性がある。いくつかの実装では、本明細書に記載されるようにロギングアパーチャを伴えば、仮想アドレスを使用してアドレス可能なRAMの量は、全く仮想アドレス幅によって限定される必要はない。さらに、特定の实施形態では、ロギングアパーチャは、本明細書に記載されるように、ロギングのために必要な仮想メモリの量を最小限に抑え、したがって、その自身のプログラム命令、データ等についてのアプリケーションによる使用のために残る仮想メモリの量を最大化する。

【0121】

実施形態では、単一化ログは、各仮想アドレス空間において実行単位あたり（例えばマルチコアCPUにおけるコアあたり）1つのロギングアパーチャを作成する。このような実施形態において、実行のスレッドがコア上で実行するように割り当てられている場合、そのログポインタは、オペレーティングシステムによって、または単一化ログに関連付けられたソフトウェアの別のボディによって、そのコアに関連付けられたロギングアパーチャをポイントするように設定される。これにより、実行のスレッドが実行を開始および停止する、および／または、1つの実行ユニットから別の実行ユニットへと移行する場合でさえ、同じ仮想アドレス空間内での複数の実行のスレッドが、それ自身のロギングアパーチャにそれぞれ、同時にデータをロギングすることが可能となる。このような実施形態では、仮想アドレス空間ごとに必要なロギングアパーチャの総数は、CPU上のコアの数によってバウンディングされる。

【0122】

不正に設定されたポインタを間接参照することにより、または他の手段により故障したソフトウェアプログラムは、例えばロギングアパーチャ内に書き込むことによって、ログにデータを不正に書き込む可能性がある。実施形態では、ロギングされたデータの消費者は、最新から最古までのロギングされたデータを読み出すことによって、本明細書に記載された技術を用いて、このような「不良ストア」を検出することができ、部分的または完全な回復を可能にする。不良ストアは、現在のログポインタの前または後にログメモリに書き込むであろう。ログポインタの後にメモリへ書き込む不良ストアは、それらが上書きするメモリがまだ有効なログデータを含まず、よって有効なログデータが上書きされない

10

20

30

40

50

ので、ほとんど重大な結果を有しない。さらに、ログポインタの後にメモリに書き込む不良ストアによって書かれたガベージデータは、インストルメント化によって生成される「良好」データによって最終的に上書きされるであろう。他の場合、ログポインタ前にメモリへ書き込む不良ストアは、良好ログデータを上書きして破壊する可能性があり、しばしば、ログの破損の時点またはそれ以前でロギングされたデータを使用することは困難または不可能となる。不良ストアによって破損したログの位置にかかわらず、不良ストア自体の証拠を、いくつかの実施形態によって本明細書に記載されるようなプリイメージインストルメント化などの通常のインストルメント化によってログに記録することができる。最新から最古までのロギングされたデータを読み出すことによりロギングされたデータの消費者は、その不良ストアによって破損した任意のデータに遭遇する前に、不良ストアに対応するインストルメント化生成ログデータに遭遇するであろう。したがって、すべての場合において、データの消費者は、例えば、ストアがロギングアパーチャに関連付けられたメモリアドレスを修正したことを認識することによって、不良ストアを検出し、適切な是正措置をとることができる。ログポインタに対する不良ストアのアドレスが、ログが破損データを含むことを示す場合には、データの消費者は、破損データに到達する前にデータの処理を停止し、したがって、破損ログデータを処理しようとすることによって引き起こされるデバッグシステムの誤動作を防止することができる。不良ストアが検出されるすべての場合において、データの消費者は、例えば、警告メッセージを表示することによって、および、不良ストアが発生したプログラム内の位置を強調表示することによって、プログラマに知らせることができる。

【0123】

不良ストア回復の例が図26Dに表示される。この例では、まだデータが含まれていないログ(X13d-log-1)内のポイント(X13d-badstore-1)で破損が起こる。ログを破損させたメモリストアは、それ自体が、メモリ変更レコード(X13d-mchg-1)を用いてロギングされる。完了ログチャンク(X13d-log-2)において、不良ストア(X13d-badstore-2)によって破損した位置が、良好データで上書きされている。ログチャンク全体は、使用可能な情報を含み、ログデータの消費者が不良ストアを示すメモリ変更レコード(X13d-mchg-2)に遭遇すると、プログラマにエラーを知らせることができる。

【0124】

不良ストア回復の第2の例が図26Eに現れる。この例では、破損は、貴重なログデータを含んだログ(X13e-log-1)におけるポイント(X13e-badstore-1)で起こる。そのログデータは今や上書きされ、回復することはできない。ログを破損させたメモリストアは、それ自体が、メモリ変更レコード(X13e-mchg-1)を用いてロギングされている。完了ログチャンク(X13e-log-2)において、データは最新から最古まで消費され、それはつまり、それ自体の破損(X13e-badstore-2)に遭遇する前に、消費者が不良ストア(X13e-mchg-2)の指示に遭遇することを意味する。データの消費者は、したがって、破損ポイント(X13e-badstore-2)におけるログが使用不能であると判断することができ、破損データに到達する前に、ログデータの処理を停止することができる。消費者はまた、プログラマにエラーを知らせることができる。

【0125】

1つのログによって書かれたログデータがシステムの他の部分による読み取りについて利用可能とされないことを保証することは、セキュリティ目的または他の理由のために有利であり得る。そのために、実施形態ではリサイクルされる前に、単一化ログは各チャンクをクリアすることができる。CPUの能力が十分である実施形態では、仮想アドレス空間へマッピングされるチャンク上のアクセス権を「書き込み専用」に設定することができる。

【0126】

実施形態では、(「ログポインタレジスタ」と称され得る)CPUレジスタを、ログポ

10

20

30

40

50

インタとして指定することができる。このような実施形態では、ログへの書き込みに使用されるインストルメント化命令は、CPUによってサポートされている場合、プリインクリメント、ポストインクリメント、プリデクリメント、および／またはポストデクリメントストア命令を含むことができる。このような命令は、単一の命令でのログへの書き込みおよびログポインタ更新により、パフォーマンスを向上させてインストルメント化フットプリントを削減することができる。

【0127】

実施形態では、アプリケーションバイナリインタフェース（ABI）または別のそのようなコーディング規則を、ログポインタレジスタがロギング使用のために予約されるように定義され得る。このような実施形態では、コンパイラ、アセンブラおよび他のコード生成ツールは、非ロギング目的のためにログポインタレジスタを使用することを許されないであろうことから、したがってログポインタレジスタがロギング目的のためにすぐに利用可能であることが保証される。このようにログポインタレジスタを予約すると、ログに書き込むために必要な命令の数を減らすことにより、パフォーマンスが向上し、インストルメント化フットプリントを削減することができる。

【0128】

実施形態では、JIT（ジャストインタイム）コンパイラまたはインストルメント化を付加するプリプロセッシングパスなどの、コンパイル後にインストルメント化を追加するコンポーネントは、非ロギング目的のためにログポインタレジスタを使用しないように、および／または、ロギング目的のために必要とされる場合はログポインタレジスタを使用できるように、および／または、ログポインタレジスタの代わりに1つ以上の他のレジスタを使用するように、コードを書き換えることができる。このようにログポインタレジスタを予約すると、ログに書き込むために必要な命令の数を減らすことにより、パフォーマンスが向上し、インストルメント化フットプリントを削減することができる。

【0129】

実施形態では、CPUレジスタを使用する代わりに、ログポインタレジスタとしてメモリ位置を使用し得る。ログポインタレジスタとしてメモリ位置を使用すると、ストア命令についてのターゲットとしてメモリ位置からメモリ位置またはオフセットを用いることをサポートするCPU上で有利であり得る。このような場合、実施形態は、CPUレジスタを予約するオーバーヘッドなしで（上記のような）予約されたログポインタレジスタの利点のいくつかまたはすべてを得られ得る。このようなオーバーヘッドには、コードサイズ増加、および／またはCPUレジスタの使用不能に起因するコードパフォーマンス低下が含まれる可能性がある。様々な実施形態では、ログポインタレジスタとして使用されるこのようなメモリ位置をシステム全体で共有することができ、または、アドレス空間ごとに、または実行のスレッドごとに、またはログごとに、別個のログポインタレジスタメモリ位置が存在する可能性があり、または、ログポインタレジスタメモリ位置が別の方法で割り当てられる可能性がある。

【0130】

いくつかの実装では、単一化ログは、本明細書に記載されるように、多くの利点を有することができる。例えば、単一化ログを構成することは、それをメモリの1つ以上のプールを割り当てる必要があるのみであり、セットアップおよび使用が非常にシンプルとなる。単一化ログは、ログに書き込む実行のスレッドごとに別個の現在のチャンクを維持することができ、それにより、共有リソースについてのスレッド間の競合が低く維持され、このような単一化ログは効率的となり、多数のログおよび大量のログプールメモリに対してスケーラブルとなる。単一化ログは、チャンクを各ログに動的に配分および再配分することができ、書き込まれることがない、または使用されないデータが書き込まれるメモリの量を最小化する。

【0131】

《キャッシュ管理》

いくつかの実施形態では、改善バックエンドは、コンピュータプログラムに挿入される

10

20

30

40

50

インストルメント化がRAM内のログにデータを挿入することができる、本明細書に記載のロギング機構などの1つ以上の機構を含むことができる。システムのCPUアーキテクチャおよびキャッシュの構造に依存するが、このようなログにログデータを書き込むと、プログラムデータがキャッシュから削除される可能性があり、アプリケーションプログラムのより遅いランタイムパフォーマンスになってしまう。キャッシュ使用量を管理するための様々な技術により、改善バックエンドがこのパフォーマンスへの影響を軽減または解消することを可能にし得る。

【0132】

CPUアーキテクチャおよびキャッシュ構造には、CPUメモリ管理ユニット(MMU)の構成を介して、および／または、コンピュータプログラムにおける特別な命令を使用することによって、または、他の方法によって、特定のストアについてのキャッシュの無効化を可能にするものがある。実施形態では、改善バックエンドは、ログメモリへのストアのキャッシュを無効し、キャッシュへのログストアの影響を低減または解消する。

10

【0133】

実施形態では、コンパイラは、ターゲットシステムのプロセッサのキャッシュ(複数可)を操作する命令を生成するように構成されている。命令は、トレースバッファへのトレースデータの将来の書き込みについてキャッシュ(複数可)を準備することができ、それにより、これらの書き込みは、効率的に、かつターゲットプログラムの実行への影響を最小限に抑えて行われる。例えば、命令は、トレースログへの予想される将来の書き込みについてキャッシュでの空間を、割り当ておよび／または予約することができる。本実施形態では、トレースログへの将来の書き込みを予測してそれに応じてキャッシュまたはメモリ内の空間を割り当て／予約することによって、本明細書に開示されたシステムは、ターゲットプログラムによってキャッシュに保存されたデータがトレースログ書き込みにより追い出されないことを確実にするために(キャッシュラインのサブセットなどの)キャッシュのサブセットにトレースログへの書き込み動作の影響を制限するように構成されることができる。ターゲットプログラムの実行に干渉および／またはそれを遅くしないように、命令は、インストルメント化の結果としてのキャッシュへの影響を追加的または代替的に低減することができる。

20

【0134】

実施形態では、必要な機能を伴うCPUアーキテクチャ上で、改善バックエンドによって挿入されたインストルメント化は、間隔をおいて、キャッシュを操作するための命令を含むことができる。インストルメント化によるログメモリの使用が非常に予測しやすい傾向にあり、例えば最低から最高へと順に各ログアドレスに保存されると考えると、そのようなキャッシュ操作命令は、プログラムメモリについてのキャッシュの利用可能性を最大化するように挿入されることができる。

30

【0135】

例えば、キャッシュからメモリの特定の範囲をフラッシュするようプログラムがCPUに命令することが可能なCPUアーキテクチャ上で、改善バックエンドによって挿入されたインストルメント化は、ログメモリを含むキャッシュライン数をできるだけ1に近い状態に維持するために、間隔をおいて、最近書き込まれたログメモリをキャッシュからフラッシュする命令を含むことができる。キャッシュが所与のメモリ位置について複数の可能な位置を有する場合には、キャッシュを「マルチウェイ」キャッシュと称することができる。ログメモリが使用するキャッシュラインの定期的フラッシングのおかげで、マルチウェイキャッシュにおける1以下のウェイがログメモリのために使用されることを保証することにより、ランタイムパフォーマンスを劇的に向上させることができ、残りのウェイをキャッシュプログラムメモリに利用可能なままとすることができる。

40

【0136】

別の例では、CPUアーキテクチャにより、メモリの内容を最初にキャッシュにフェッチすることなく、メモリの特定の範囲についてキャッシュラインを予め割り当てるようプログラムがCPUに命令することができる。このような命令を使用することにより、その

50

メモリの現在の内容がすぐに上書きされるであろうことをプログラムがCPUに通知し、これによりCPUは不要な作業を避けて、メモリの現在の内容をキャッシュにコピーする通常のステップをスキップすることによりメモリバス競合を減らすことができる。そのような機能を有するCPUアーキテクチャ上では、改善バックエンドによって挿入されたインストルメント化は、メモリはすぐに上書きされるのでメモリの内容を最初にフェッチすることなく、間隔において、書き込まれるログメモリの次のセクションについてキャッシュラインを予め割り当てる命令を含むことができる。ログメモリによる使用のためにキャッシュラインを定期的に予め割り当てることにより、ログメモリをキャッシュにコピーするCPUの必要性を最小化することによってランタイムパフォーマンスを劇的に向上させることができる。

10

【0137】

いくつかの実施形態において、キャッシュフラッシュ命令およびキャッシュ事前割当命令は集合的に、とりわけ、キャッシュ操作命令と呼ぶことができる。本明細書に記載のキャッシュ操作命令があまりにもまれにしか実行されない場合は、キャッシュ操作の潜在的利点は失われる可能性があり、プログラムデータに利用可能なキャッシュラインは少ないままとなり、ランタイムパフォーマンスを低減させる。本明細書中に記載のキャッシュ操作命令があまりにも頻繁に実行される場合は、CPU時間が不必要にキャッシュ操作命令を実行して無駄になる。プログラムへと改善バックエンドによって挿入されたインストルメント化がキャッシュ操作命令を含む場合、改善バックエンドにおけるコンパイラは、このようなキャッシュ操作命令の適切な配置を決定するためのコードフロー解析を使用することができる。例えば、このようなコンパイラは、ログに書き込まれたログデータに対して各キャッシュラインにつき1回、プログラムがキャッシュ操作（複数可）を実行するために、このようなキャッシュ操作命令を挿入することができ、それによってパフォーマンスゲインを最大にする。

20

【0138】

実施形態では、コンパイラは、次のようにキャッシュ操作命令の適切な配置を決定するために、コードフロー解析を行うことができる。コンパイラを、ターゲットCPUのキャッシュラインサイズを決定するように構成することができる。コンパイラは、各基本ブロックにおけるインストルメント化によってログに書き込まれたバイトの数を追跡し続けるように構成されることができる。基本ブロック内において、書き込まれた合計ログデータがCPUキャッシュラインサイズに等しいインストルメント化が挿入されるたびに、コンパイラは、適切なキャッシュ操作命令（複数可）を挿入するように構成されることができる。実施形態では、コンパイラは、各基本ブロックの先行ブロックを決定するようにさらに構成されることができ、「先行ブロック」は、その実行が当該ブロックの実行の直後に行なわれることができる基本ブロックとして定義することができる。基本ブロックをコンパイルする場合、コンパイラは、問題のブロックの実行につながる可能性のある各実行パスをトレースするために先行ブロック情報を使用することができる。そのような各実行パスについて、コンパイラは、最後のキャッシュ操作以降にログに書き込まれたバイトの最大可能数を決定することができ、これは「ブロックエントリにおけるワーストケースキャッシュライン消費量」と呼ぶことができる。ブロックエントリにおける最大ワーストケースキャッシュライン消費量が現在の基本ブロックにつながる各コードパスについて計算されると、コンパイラは、ワーストケースキャッシュライン消費量（ブロックエントリにおけるワーストケースキャッシュライン消費量とブロックの開始以降にログに書き込まれたデータとの和）がキャッシュラインサイズに等しいブロック内のポイントにおける現在のブロックのキャッシュ操作命令を挿入することができる。このようなアルゴリズムにより、データに対する1以下のキャッシュラインが最後のキャッシュ操作以降にログに書き込まれることが保証される。

30

40

【0139】

図28は、このようなコードフロー解析の例を示している。図は、コンパイル中のプログラムにおける基本ブロック（ブロックA）、その直接の先行ブロック（ブロックB、ブ

50

ロック E)、およびブロック B の先行ブロック (ブロック C、ブロック D) を示している。改善バックエンドにおけるコンパイラは、どこであれキャッシュ操作命令を挿入するためのブロック A における位置を決定しようとする。例では、キャッシュラインサイズが 32 バイトである。ブロック C からブロック B へのそしてブロック A へのパスにおいて、ブロック A へのエン트리におけるワーストケースキャッシュライン消費量は、(ブロック B にてロギングされた) 8 バイトと (ブロック C の最後のキャッシュ操作の後にロギングされた) 16 バイトとの和であり、すなわち合計 24 バイトである。ブロック D からブロック B へのそしてブロック A へのパスにおいて、ブロック A へのエン트리におけるワーストケースキャッシュライン消費量は、(ブロック B にてロギングされた) 8 バイトと (ブロック D の最後のキャッシュ操作の後にロギングされた) 20 バイトとの和であり、すなわち合計 28 バイトである。ブロック E からブロック A へのパスにおいて、ブロック A へのエン트리におけるワーストケースキャッシュライン消費量は、(ブロック E の最後のキャッシュ操作の後にロギングされた) 24 バイトである。したがって、すべてのパスからブロック A へのエン트리におけるワーストケースキャッシュライン消費量は、各パスについてのキャッシュライン消費量の最大値であり、28 バイトである。キャッシュラインサイズからこの最大値を減算することにより、コンパイラは、ブロック A におけるキャッシュ操作命令の適切な配置が、ブロック A 後ログの 32 バイト - 28 バイト、すなわちブロック A 後ログの 4 バイトであると計算することができる。

【0140】

いくつかの実施形態において、本明細書に記載されるようにキャッシュ操作およびコードフロー解析を適用することによって可能となるランタイムパフォーマンスゲインは、プログラム挙動、CPU キャッシュアーキテクチャ、および他の多くの要因に依存して、かなり変化する。しかし、2 倍、3 倍、5 倍、またはそれ以上のランタイムパフォーマンスの向上は珍しいことではない。実施形態では、コンパイラは、利点がある可能性がある場合にのみ、キャッシュ操作命令を挿入および／またはコードフロー解析を実行するように構成されることができる。例えば、コンパイラは、キャッシュアーキテクチャが多くの利益を与えないであろう CPU についてコンパイルする時には、キャッシュ操作命令を挿入しないように構成されることができる。

【0141】

以下は、PowerPC の CPU においてインストルメント化コードに挿入されたキャッシュ操作命令の一例である。

; 次の例では、r15 が、書き込まれる次のログ位置をポイントする。

【0142】

li r17, 0x40; 「dcbal」命令の準備

【0143】

dcbal r15, r17; 「データキャッシュブロック割り当て」が

; 前の命令とともに、その内容をプリフェッチせずに、

; 次のキャッシュラインがログに割り当てられることを確保する

【0144】

li r17, 0xffffffffc4; 「dcbf」の準備命令

【0145】

dcbf r15, r17; 「データキャッシュブロックフラッシュ」

; 前の命令とともに、最も近い時点で書かれたログの

; セクションをキャッシュからフラッシュし、よってそれは

; 再利用され得る。

【0146】

lis r17, 0x100; 「stw」の準備

【0147】

addi r17, r17, 0x164; 「stw」の準備

【0148】

10

20

30

40

50

s t w u r 1 7 , 4 (r 1 5) ; いくつかの実際のデータロギングを実行
【 0 1 4 9 】

《ロギングされたデータの収集》

実施形態では、改善バックエンドは、最新から最古までのロギングされたデータを収集する。所与のレコードの解釈が以前にロギングされたレコードに依存しないようにロギングされたデータが構成されている場合には、改善バックエンドは時間走行デバッガユーザインタフェースへの検索時にすぐに、収集されたデータを利用可能にすることができ、プログラマは、収集されるとすぐにデータの解析およびデバッグを開始することができる。より多くのデータが収集されると、すべてのデータが収集されるまで、それはプログラマに利用できるようになる。

10

【 0 1 5 0 】

デバッグ用データの利用可能性が図 1 9 に示される。図示の例では、ロギングされたデータの収集は、ログの終わり (X 5 0 1) で始まり、後に記録されるデータから以前に記録されたデータに向かって進む。収集されたデータの領域 (X 5 0 2) は、ロギングされたデータの継続的な検索 (X 5 0 5) のために時間をかけて成長する。デバッグに利用可能なデータの領域 (X 5 0 4) は、収集されたデータの領域 (X 5 0 2) と同じであることに留意されたい。すべてのデータが収集されるまで、データ検索は継続し、まだ収集されていないデータの領域 (X 5 0 3) がなくなると、その時点でログ全体からのデータがデバッグに利用可能となる。このようにデータをデバッグ用に利用可能にすることにより、典型的にはデバッグについて最も関心のある、後に記録されたデータをまず、プログラマに利用可能にすることができる。収集されるとすぐにデータはプログラマに利用可能となることができるため、最新データを解析およびデバッグする前にプログラマが待機しなければならない時間は、収集されるデータの合計量には依存せず、大規模なトレースログを任意に使用することが実用的となる。

20

【 0 1 5 1 】

《複数ログからのデータマージ》

改善バックエンドは、いくつかの実施形態による複数ログへとデータをロギングすることができる。複数ログにロギングされたデータはマージされて、全体として解析されることができる。複数ログへのロギングにより、複数の実行のスレッドを有するシステムにおいて、より効率的なロギングが可能となり、ロックなどの競合処理機構を必要とせずともよい。複数ログからロギングされたデータをマージすることにより、SMPシステム、および、実行のスレッドが同時に実行されてメモリを共有し得る他の構成のデバッグが可能になる。複数ログからロギングされたデータをマージすることにより、分散システム、および実行のスレッドが同時に実行されるがメモリを共有しない他の構成のデバッグが可能になる。複数ログからロギングされたデータをマージすることにより、異なるタイプおよび／またはアーキテクチャの複数のCPUを有するもの、メモリを共有するおおよびしない同時に実行される実行のスレッドの任意の組み合わせを有するもの、または、CPUが共有メモリによってまたはメッセージパッシングによってまたは他の手段によって互いに通信するもののなどの複雑なシステムのデバッグが可能になる。

30

【 0 1 5 2 】

いくつかの実施形態では、改善バックエンドは、1つ以上のマージされたログに複数ログからのデータをマージすることができる。改善バックエンドは、各ログから処理すべき次のレコードを読み出して、マージされたログに次のものを挿入するのに論理的に最も適切であるのはそれらのレコードのどれであるかを決定することにより、複数ログからのデータをマージする。レコードがマージされたログに挿入されると、それはもはやマージ用とは考えられず、そのログから処理されるべき次のレコードが代わりに考慮される。レコードは、同じ時系列順に各ログから考慮されるので、よって例えば、改善バックエンドが逆時系列順にログデータを処理している場合、マージ処理中に、各個々のログの記録は逆時系列順に考慮される。

40

【 0 1 5 3 】

50

異なるソースからのログデータをマージすることの例が図 20 に現れている。図の左側は、3つの異なるソースによって記録されたログである。例では、ログデータは、最新（後で記録されるデータ）から最古（以前に記録されたデータ）まで収集される。検討中の次のレコード（X 6 0 1, X 6 0 2, X 6 0 3）が、まだ各データソースからマージされていない最新のレコードである。例では、ソース # 2 からの最も近い時点でのレコード（X 6 0 2）がマージされる次のレコードであり、したがってマージされたログ（X 6 0 4）に付加されると決定されている。これが行われると、ソース # 2 からの最も近い時点でのレコード（X 6 0 2）が、すでにマージされたデータ（X 6 0 5）の一部とみなされ、もはやマージの候補になることはないであろう。プロセスが次に繰り返されるときを考慮すれば、X 6 0 2 の直前のレコードが、ソース # 2 からの最も近い時点でのレコードとなるであろう。

10

【0154】

マージ処理中に、改善バックエンドは、マージされたログに次に挿入するのに論理的に最も適切であるレコードがどれであるかを決定する際に（「順序付け技術」と称され得る）多くの異なる技術のうちの1つ以上を採用し得る。論理的に最も適切な順序とは典型的には、マージされたログにおいてレコードが論理的に狂った順序で発生しない順序である。例えば、マージされたログにおいて時系列的に先に現れるレコードが、その対応するレコードがマージされたログにおいて時系列的に後で現れるイベントによって引き起こされるイベントを表す場合に、2つのレコードを論理的に狂った順序であると考えることができる。なお、複数ソースからのログデータの任意の所与のボディについて、論理的に狂った順序であるレコードを含まない多くの順序付けが存在し得る。「同期イベント」とは、複数ソースからのデータを順序付けする際にシーケンスが重要となるイベントを指し得る。例えば、対応するレコードがマージされたログ内のそれらの相対的配置によって狂った順序となる可能性があるのであれば、2つのイベントは同期イベントである。同期イベントには、CPU間の通信イベント、オペレーティングシステムイベント、ミューテックスまたは他の同期プリミティブ上のアクションが含まれるが、これらに限定されない。

20

【0155】

1つの順序付け技術は、データの様々な部分がロギングされた順序を明示的に説明する1つ以上のログに記録されたメタ情報を使用することである。例えば、カーネルおよびアプリケーションが単一CPUを共有する場合、カーネルはそのログに、制御がアプリケーションとの間で転送される時点と、転送時点でアプリケーションによって書き込まれたログの量を記録することができる。この例では、この情報を、カーネルおよびアプリケーションによってロギングされたデータの論理的に最も適切な順序付けを決定することができる順序付け技術への入力として使用することができる。この順序付け技術の例が図 21 A である。ソース # 1 からデータは、データ領域（X 7 a - S r c 2 - 1）のソース # 2 によって、それぞれロギングの開始（X 7 a 0 1）およびロギングの終わり（X 7 a 0 2）を表すメタデータレコードを含む。このメタデータを使用して、順序付け技術は、マージされたログにおけるソース # 1 からのデータ領域（X 7 a - S r c 1 - 1 - m e r g e d および X 7 a - S r c 1 - 2 - m e r g e d）間に、ソース # 2 からのデータ（X 7 a - S r c 2 - 1 - m e r g e d）を正しく挿入することができる。この例では、メタデータレコードは、2つの異なるレコード（X 7 a 0 1 および X 7 a 0 2）として表されているが、実施形態では、メタデータは、1つ以上のレコードで表すことができる。この例では、メタデータレコード（X 7 a 0 1 および X 7 a 0 2）がマージされたログから省略されているが、実施形態では、メタデータはマージされたログに含まれても含まれていなくともよい。

30

40

【0156】

別の順序付け技術は、同期イベントを表す特定のレコードにタイムスタンプすることである。このようなタイムスタンプが単一のクロックからまたは十分に同期される複数のクロックから生成される場合、これらのタイムスタンプは、最高または最低のタイムスタンプを伴うレコードを選択することにより、ログが読み出される方向に応じて、順序付けの

50

ために使用することができる。タイムスタンプ順序付け技術の例が図 2 1 B に表示される。ソース # 1 からの次のレコード (X 7 b - S r c 1 - 1) のタイムスタンプ (1 0 0 2) が、ソース # 2 からの次のレコード (X 7 b - S r c 2 - 1) のタイムスタンプ (1 0 0 0) よりも大きいと考えると、順序付け技術は、マージされたログにおいてソース # 2 からのレコード (X 7 b - S r c 2 - 1 - m e r g e d) がソース # 1 からのレコード (X 7 b - S r c 1 - 1 - m e r g e d) の前に現れなければならないと決定することができる。

【0157】

さらに別の順序付け技術は、他の同期イベントに対してその順序を決定するために用いることができるシリアルナンバまたは他のそのような識別子を伴う特定の同期イベントをタグ付けすることである。例えば、1つのCPUから別のCPUに送信されたメッセージが1つのログにおけるメッセージ送信レコードおよび別のログにおけるメッセージ受信レコードとなり、両方のレコードがそのメッセージに一意である識別子を含む場合、順序付け技術は、マージされたログ内のレコードを順序付けするためには送信イベントが受信イベントに先行している必要があるとの知識を使用することができる。シリアルナンバ順序付けの例が図 2 1 C に表示される。レコード X 7 c - S r c 1 - 1 がシリアルナンバ # 2 0 0 を伴うメッセージの受信を表し、レコード X 7 c - S r c 2 - 1 がシリアルナンバ # 2 0 0 を伴うメッセージの送信を表すと考えると、順序付け技術は、送信が受信に先行している必要があり、したがって、マージされたログにおいてレコード X 7 c - S r c 2 - 1 - m e r g e d がレコード X 7 c - S r c 1 - 1 - m e r g e d に先行している必要があると決定することができる。

【0158】

さらに別の順序付け技術は、マージされたログにおいてレコードが論理的な順序で現れることを保証するために、プリイメージおよびポストイメージデータなどのログからのデータを使用することである。例えば、連続して実行される2つの異なるスレッドによって共有メモリ位置が書き込まれ、プリイメージおよびポストイメージが両方のスレッドのストアについて既知であり、1つのストアのプリイメージおよび他のストアのポストイメージが一致する唯一の順序付けが存在する場合、順序付け技術は、マージされたログにおいて対応するメモリストアレコードを正しく順序付けするために、この情報を使用することができる。データベース順序付けの例が図 2 1 D である。例では、2つの異なるソースからのレコード (X 7 d - S r c 1 - 1, X 7 d - S r c 2 - 1) が、共有変数「x」に対する変更を表す。ソース # 1 からのレコード (X 7 d - S r c 1 - 1) は、3とのプリイメージ値 (レコードにおいて表される修正前の「x」の値)、および4とのポストイメージ値 (レコードにおいて表される修正後の「x」の値) を含む。ソース # 2 からのレコード (X 7 d - S r c 2 - 1) は、2とのプリイメージ値および3とのポストイメージ値を含む。したがって例では、データベース順序付け技術は、レコード X 7 d - S r c 2 - 1 - m e r g e d がマージされたログにおいて X 7 d - S r c 1 - 1 - m e r g e d に先行する必要があると決定することができ、これは、X 7 d - S r c 2 - 1 - m e r g e d のポストイメージ (3) が X 7 d - S r c 1 - 1 - m e r g e d のプリイメージ (3) と一致する一方、(2とのプリイメージを伴う X 7 d - S r c 2 - 1 - m e r g e d の前に4とのポストイメージを伴う X 7 d - S r c 1 - 1 - m e r g e d を配置する) 逆順序付けでは、前のレコードのポストイメージと後のレコードのプリイメージとの論理的対応が得られないからである。

【0159】

《メモリ再構成》

改善バックエンドは、いくつかの実施形態によれば、ログに表される任意の時点におけるターゲットプログラムのメモリの状態を再構成することができる。プリイメージデータを含む以前に記録されたメモリ変更レコードを使用して、次のようにターゲットプログラムのメモリ状態を再構成することができる。

【0160】

実施形態では、改善バックエンドは、所与の時間におけるターゲットプログラムのメモリの状態の表現を維持する。最初は、この状態はライブターゲットプログラムからコピーされ得る。

【0161】

例えば、トレースログは、ゼロ以上のメモリ変更レコードを含み得る。このような各メモリ変更レコードは、改善バックエンドがどのメモリ範囲が修正されたかを決定することができる十分な情報、および、（メモリ修正が行われる前にそのメモリ範囲に存在した値などの）「プリイメージ」値を含み得る。実施形態では、メモリ変更レコードはまた、ポストイメージ値を記憶するために予約された空き空間を含む。

【0162】

実施形態では、改善バックエンドがメモリ変更レコードの前に初めてメモリを再構成するときに、その位置のデバッガの表現における現在の値がポストイメージを保存するための予約空間にコピーされ、プリイメージ値がその位置のデバッガの表現へとログからコピーされる。これは、メモリ変更レコードに表されるメモリ変更の直前にターゲットプログラムのメモリ状態を再現する効果がある。ログからの連続したメモリ変更レコードで上記を繰り返すことによって、改善バックエンドは、プログラムの元の実行中に連続するますます以前のポイントにおけるプログラムのメモリ状態を再構成し得る。実施形態では、ポストイメージがメモリ変更レコードに既に存在する場合、改善バックエンドが既にポストイメージを保存しているため、または、それが他の手段によって決定されているため、ログにポストイメージをコピーすることを省略し得る。

【0163】

実施形態では、このプロセスは、連続する後の時点にプログラムのメモリ状態を移動するために逆転される。具体的にはデバッガは、保存されたポストイメージをメモリ変更レコードから関連位置のデバッガの表現にコピーして、メモリ変更レコードによって表される修正の直後にプログラムのメモリ状態を再構成し得る。ログからの連続したメモリ変更レコードで上記を繰り返すことによって、改善バックエンドは、プログラムの元の実行中に連続するますます後のポイントにおけるプログラムのメモリ状態を再構成し得る。

【0164】

図15A、図15A、図15Cおよび図15Dはメモリ再構成の例を示す。示されているように、ログにおけるメモリ変更レコード（X100）は、アドレスおよびプリイメージを含み、空間がポストイメージについてのログにおいて予約されているが、そこには何もまだ書かれていないので、値は未定義である。その位置についてのシミュレートされたメモリ（X101）は、初期値「4」を含む。

【0165】

ログエントリに記録されたメモリ変更（X100）前のターゲットのメモリ状態を再構成するために、シミュレートされたメモリ内の現在の値はまず、予約されたポストイメージ空間にコピーされる（X102）。次いで、プリイメージはシミュレートされたメモリにメモリ変更レコードからコピーされる（X103）。今や、シミュレートされたメモリ（図15B、X104）は、メモリ変更レコード前のターゲットの状態を表し、メモリ変更レコード（X105）は、その変更に関連付けられたプリイメージおよびポストイメージの両方を含む。

【0166】

時間的に前方に移動すると、ログエントリに表されるメモリ変更（X105）の後にターゲットのメモリ状態を再構成することにより、改善バックエンドはシミュレートされたメモリ（X104）へのメモリ変更レコードからポストイメージをコピーする（X106）。今や、シミュレートされたメモリ（図15C、X107）は今一度、メモリ変更後のターゲットプログラムの状態を表す。このポイントでメモリ内に表される状態（X107）は、初期状態（図15A、X101）に一致することに留意されたい。

【0167】

時間的に後方に移動すると、ログエントリに表されるメモリ変更（図15C、X108

10

20

30

40

50

）の前にターゲットのメモリ状態を再構成することにより、改善バックエンドはシミュレートされたメモリ（X107）へのメモリ変更レコードからプリイメージをコピーする（X109）。以前に行っているので、メモリ変更レコードのポストイメージを埋め込む必要はない。今や、シミュレートされたメモリ（図15D、X110）は今一度、メモリ変更前のターゲットプログラムの状態を表す。このポイントでメモリ内に表される状態（X110）は、改善バックエンドが初めてメモリ変更レコードを交差させた後の状態（図15B、X104）と一致することに留意されたい。

【0168】

図16は、ログが多数のメモリ変更レコードを含む場合に異なる時点でのターゲットのメモリ状態を再構成するためにこのような技術をいかに使用することができるかの例を示している。この例では、メモリ変更レコードは、図中で古いレコードの右側に新しいレコードが現れる時系列順にログに記録されている。メモリ変更レコード（X201、X202、X203）間の各ギャップは、以前のメモリ変更（図のギャップの左側に現れる）がすでに起こり、後のメモリ変更（図のギャップの右側に現れる）はまだ発生していない、ターゲットプログラムの実行中のポイントを表す。図16のギャップによって表される任意の所与の時点から出発して、改善バックエンドは、図15A～Dで実証されているものなどの技術を適用することによって、時間的に前または後のいずれかで隣接ギャップに移動することができる。繰り返しそのような移動を行うことで、改善バックエンドは、ログに表される任意の時点におけるターゲットプログラムのメモリの状態を再構成することができる。

【0169】

実施形態では、プリイメージおよびポストイメージを、メモリ変更レコード内の同じ位置に保存することができる。このような実施形態では、プリイメージおよびポストイメージは、単一の「メモリエイメージ」位置に保存される。メモリ変更レコード内のメモリエイメージ位置は、現在の再構成時点がメモリ変更レコードの前または後のいずれに発生したかに応じて、プリイメージまたはポストイメージのいずれかを含むことができる。現在の再構成時点が所与のメモリ変更レコードの後であれば、メモリ変更レコード内のメモリエイメージ位置は、プリイメージを含む。メモリ変更レコードにわたって時間的に後方に再構成するとき、メモリエイメージ位置の内容が再構成メモリ内の対応する位置の内容とスワップされ、したがってメモリエイメージ位置の値をポストイメージに変換する。メモリ変更レコードにわたって前方に再構成するとき、同一のスワップ動作がメモリエイメージ位置の値をプリイメージに変換し戻す。ポストイメージについてログ内で空間を予約する必要がないので、より効率的にログ空間を使用するという点で、このような技術は有利である。

【0170】

《ベースラインメモリエイメージ》

メモリ変更レコードは、本明細書に記載されるように、いくつかの実施形態では、メモリ状態の漸進的变化を表すことができる。それら自身によりそのような変化を解析することによって、改善バックエンドは経時的に変化するメモリ位置の内容を再構成することができる。デバッグ中のシステムのメモリの完全なイメージを作成するために、改善バックエンドは、既知の時点でのシステムのメモリの一部またはすべての内容を含むベースラインイメージを使用することができる。改善バックエンドは、ベースラインイメージから始まって、メモリ変更レコードを使用してメモリを再構成する以前に記載された技術、または同様の技術を用いて、時間的に前または後にメモリを再構成することができる。そのようなベースラインイメージは、システムメモリの単一スナップショットであることができ、または、それは、同時にまたは異なる時間に撮られた複数のスナップショットを含むことができる。ベースラインイメージは、ログによってカバーされる任意の時間に由来することができる。

【0171】

実施形態では、システムは、デバッグのために停止され、改善バックエンドは、ベースラインイメージとしてシステムメモリ自体を使用する。改善バックエンドは、システムの

メモリからメモリ状態をすべて一度に、または必要に応じてセクションでダウンロードすることができる。ターゲットプログラムメモリが任意に大きく、プログラムが任意の時間量でデバッグを開始するのを遅らせる可能性があるので、ターゲットプログラムメモリをすべて一度にダウンロードすると非効率的である可能性がある。実施形態では、改善バックエンドは、必要に応じて一度にプログラムメモリのセクションをダウンロードすることができ、デバッグを開始するのに必要な時間をバウンディングする。デバッグのために停止している間にシステムがどのような挙動を有するかによって、システムが停止状態にある間にメモリが変化し続けることが可能であり得、その場合に、改善バックエンドは、メモリが変更中の間であっても、例えば後のセクションで記述されるようなメモリ整合性技術を使用することによりメモリーイメージの整合性を確実にする様々な技術を採用することができる。

10

【0172】

実施形態では、改善バックエンドは、ベースラインとして（「コアダンプ」と称され得る）システムメモリの以前に保存されたイメージを使用することができ、このようなコアダンプは、エラーの結果として、ユーザの要求によって、または他のいくつかの理由によってのいずれであっても作成される。そのようなコアダンプは、メモリーイメージ単独で構成することができ、または、レジスタ状態情報、実行のスレッドに関する情報、および／または仮想メモリマッピングに関する情報を含むがこれらに限定されない他の有用なデータを含むことができる。実施形態では、改善バックエンドによって収集されたログデータをメモリに保存することができ、またはコアダンプイメージと併せて別様に利用可能であり、改善バックエンドを採用する時間走行デバッグが、コアダンプが生成された時点に至るまでのアプリケーションの実行履歴をステップスルーすることができる。このように、改善バックエンドは、デバッグの非常に強力なモデルを可能にし、エラーの検出、またはユーザの要求、またはいくつかの他の条件によりコアダンプイメージの生成が得られる。そのようなコアダンプイメージは、所定の箇所にて解析され、後の収集のために保存され、ならびに／または、記憶および／もしくは解析のために別の位置に送信されることができる。改善バックエンドを採用する時間走行デバッグはそして、そのようなコアダンプイメージを解釈することができ、プログラムがコアダンプイメージが撮られたシステムの「ポストモテム」デバッグを実行することができる。ポストモテムデバッグは、コアダンプに含まれる情報、またはプログラムが停止した後にアクセス可能であり、あるいは、もはや別様に実行されない情報のいくつかの他のソースに基づいて、1つ以上のプログラムのデバッグを参照することができる。このようなポストモテムデバッグは、改善バックエンドで行われた場合、コアダンプに至るモーメントにおけるシステムの状態の再構成、システムの実行履歴のステップスルー、および実行についての様々な種類の解析の実行を含むがこれらに限定されない改善バックエンドの機能を含むことができる。

20

30

【0173】

実施形態では、改善バックエンドは、コピーバックエンドによって採用されるものなどのコピー技術を使用して保存されたメモリのイメージを使用することができる。

【0174】

《メモリが変更されているときのメモリーイメージ整合性維持》

40

改善バックエンドが、ベースラインメモリーイメージの構築のために、またはなんらかの他の目的のために必要な時に一度に1つのプログラムメモリのセクションをダウンロードする実施形態では、プログラムメモリの内容が2つのセクションのコピー間隔を変更することが起こり得、プログラムのメモリの不整合イメージとなってしまう。いくつかの実施形態は、この問題を緩和するために（本明細書に記載の）同期実行制御などの実行制御戦略を採用することができる。しかし、採用される実行制御戦略または他の理由により、改善バックエンドがこのようなプログラムメモリのダウンロードを開始した後にプログラムメモリが修正されないことを保証しない実施形態では、改善バックエンドは、プログラムのメモリの整合性のあるイメージを保証するために、本明細書に記載されるものなどの技術を使用することができる。

50

【0175】

実施形態では、改善バックエンドは、コピーされる各セクションに影響を与えるメモリ修正レコードについてのログをチェックすることができる。そのようなレコードが最初のセクションがコピーされた後に発生したメモリ修正を表す場合、（「後のメモリ変更レコード」と称され得る）そのようなレコードは、コピーされた最初のセクションとコピー中の現在のセクションとの間の不整合を表す。改善バックエンドはそして、コピー中のメモリに対する逆時系列順に、または、以前にコピーされたメモリイメージに対する前方時系列順に、後のメモリ変更レコードを適用するために、先に説明したもののような技術を使用して、メモリ再構成を行うことができる。これは、コピーされたメモリセクションを以前にコピーされたメモリセクションと整合させ、異なる時間にコピーされたプログラムのメモリのセクション間の不整合を解消する。

10

【0176】

図24は、後のメモリ変更レコードの適用例を含む。図示の例では、改善バックエンドは、ロギングされたデータ（X1000）を解釈する。ターゲットシステムが実行を停止した場合、改善バックエンドは、ターゲットプログラムが停止した時点（X1001）までターゲットが実行している間にロギングされたデータ（X1006）の読み出しを始める。ターゲットプログラムが停止した時点（X1001）で、改善バックエンドは、ログデータを解釈するのを助けるために、ターゲットからメモリブロック#1をコピーする。後の時点（X1002）で、改善バックエンドは、メモリブロック#2を必要とし、これをターゲットからコピーする。しかし、メモリブロック#1がコピーされた時点（X1001）とメモリブロック#2がコピーされた時点（X1002）との間に、追加のメモリ変更レコード（X1004）がロギングされる。これらは「後のメモリ変更レコード」と呼ばれ、これはターゲットプログラムが停止した後にそれらがロギングされたからである。それらの間の後のメモリ変更レコード（X1004）が、メモリブロック#2の内容がメモリブロック#1がコピーされた時間以降に変更されているかもしれないことを示すので、改善バックエンドによりなされたメモリブロック#1および#2のコピーは、不整合となり得る。メモリブロックの整合性を保つために、改善バックエンドは、メモリブロック#2の内容に対して逆時系列順にメモリ変更レコード（X1004）を適用し、メモリブロック#1がコピーされた時間（X1001）におけるメモリブロック#2の内容の表現をもたらす。得られたイメージはしたがって、メモリブロック#1のコピー内容と整合している。さらに後の時点（X1003）で、改善バックエンドはターゲットからメモリブロック#3をコピーする。改善バックエンドは、X1005およびX1004における後のメモリ変更レコードをメモリブロック#3のコピー内容に適用し、メモリブロック#3のコピー内容をX1001でターゲットからコピーされたメモリブロック#1の内容と整合させる。このように、改善バックエンドは、ターゲットの実行中にロギングされたデータ（X1006）を解釈するために各ブロックが必要な場合にのみターゲットからメモリをコピーし、メモリブロックがコピーされた時にターゲットメモリがポイント（X1001, X1002, X1003）間で変更されているかもしれない場合であっても、改善バックエンドはターゲットメモリの整合したイメージを構築する。

20

30

【0177】

《仮想マッピング》

いくつかの実施形態では、ターゲットシステムのメモリを再構成するとき、改善バックエンドは、仮想メモリアドレス、物理メモリアドレス、またはその両方で動作することができる。改善バックエンドは、仮想アドレスと物理アドレスとの間の関係に関する情報を含む仮想メモリマッピング情報をターゲットシステムから収集することができる。このようなセキュリティモデルをサポートするCPU上では、改善バックエンドによって収集された仮想メモリマッピング情報は、保護リングに関する情報、およびより少ない特権コードからデータを保護する他の手段を含むことができる。改善バックエンドは、ターゲットシステムのその内部表現を編成するのを助けるために、仮想メモリマッピング情報と他のメモリ保護情報とを使用することができる。例えば、改善バックエンドは、異なる仮想ア

40

50

ドレス空間で同じアドレスを区別するために仮想メモリマッピング情報を使用することができ、これは、異なる物理的メモリを表し、したがって異なる値を含むことができる。改善バックエンドは、共有メモリを表すために仮想メモリマッピング情報を使用することができ、そこでは、同じまたは異なる仮想アドレス空間内の異なる仮想アドレスが同じ物理メモリを表している。共有メモリを表すために仮想メモリマッピング情報を用いることにより、改善バックエンドは、任意の特別なインストルメント化または共有メモリへのストアのロギングを提供する改善バックエンドの必要性を排除することができる。

【0178】

《再構成シミュレーション》

時には、改善バックエンドが、デバッグ中のシステム内の1つ以上のCPUの実際のまたは理論的な挙動を再構成するなど、1つ以上のCPUの命令レベルシミュレーションを実行できるようにすることが有用であり得る。これは再構成シミュレーションによって達成することができる。

10

【0179】

再構成シミュレーションについての次の技術は、命令セットシミュレータに依存することができ、これは、初期レジスタ状態とメモリの初期状態とが与えられると、1つ以上のマシン命令が実行された後にCPU内で起こるであろうレジスタ状態を生成することができる任意の機構である。ソフトウェアシミュレータ、ジャストインタイム(JIT)コンパイラ、ソフトウェアおよびハードウェア仮想化機構、およびCPUを含むがこれらに限定されない多くのそのような機構がある。そのような命令セットシミュレータは、CPUのレジスタセット全体、またはターゲットプログラムが実行されるCPUの挙動を正確に表すのに十分である任意のサブセット上で動作し得る。異なるタイプの複数のCPUを伴うシステムをデバッグする際に、複数の命令セットシミュレータが必要であり得ることに留意されたい。

20

【0180】

再構成シミュレーションは、CPUが行った、または動作し得る歴史的または理論的環境を再現し、なんらかの後の時点でシステムの状態を決定するために命令セットシミュレータを適用することによって達成され得る。再作成された環境には、初期レジスタ状態、初期メモリ状態、およびCPUの挙動に影響を与える可能性のある任意の外部イベントの作成または再作成が含まれるが、これらに限定されない。そのような環境は、ロギングされた情報を使用して作成され、または、レジスタ状態スナップショット、「メモリ再構成」セクションに記載された技術を使用して再構成されたメモリなどの前述の技術、および、ロギングされた、デバッグ中、製造中、もしくは合成中のプログラムの最終状態から収集された、または、これらの様々なソースに由来する他の情報によって再構成され得る。

30

【0181】

《レジスタ状態再構成》

改善バックエンドの実装は、いくつかの実施形態によればCPUレジスタ状態の再構成を必要とし得る。(本明細書に記載されるように)レジスタ状態スナップショットを定期的にロギングする場合は、次の技術が、ログ内の任意のポイント(「宛先ポイント」と称され得る)でのレジスタ状態を再構成するために使用し得、そこでは、そのレジスタ状態が宛先ポイントにおいて未知である各CPUの宛先ポイントの前にロギングされた少なくとも1つのレジスタ状態スナップショットが存在している。

40

【0182】

宛先ポイントでのレジスタ状態を再構成するために、時間走行デバッガはそのレジスタ状態が宛先ポイントにて未知であるCPUのセットを最初に決定することができる。これは典型的には、宛先ポイントでコードを実行していたCPUであるが、そのリストからいくつかのCPUを含むかまたは除外する理由があり得る。レジスタ状態がすでにすべてのCPUについて既知である場合、このまたは任意の他のレジスタ再構成アルゴリズムの使用が必要とされないことに留意されたい。

【0183】

50

そのレジスタ状態が再構成されるべきCPUのセットが決定されると、改善バックエンドは、（「再構成ポイント」と称され得る）ログにおける以前の時点でのメモリの状態を再構成する「メモリ再構成」のセクションで説明したものなどの技術を使用することができ、これにより、そのレジスタ状態が未知である各CPUについて、再構成ポイントと宛先ポイントとの間にロギングされた少なくとも1つのレジスタ状態スナップショットが存在する。そして、デバッガは、（「メモリ再構成」セクションに記載されているものなどの）メモリ再構成技術、および（「再構成シミュレーション」セクションに記載されているものなどの）再構成シミュレーション技術の組み合わせを使用して、宛先ポイントに戻り、その前方の各段階で、メモリ状態およびレジスタ状態の正しさが維持されることを確実にし得る。これは例えば、次のような処理を行うことにより行なわれ得る。

【0184】

再構成ポイントから開始して：

【0185】

・デバッガ内のメモリおよびCPU状態で現在、表される時点で、どのCPUが次の実行ステップを取るかを決定する。（本明細書に記載された技術を含め、どのCPUが次の実行ステップを取るかを決定するための多くの技術が存在し得る）。

【0186】

・現時点に関連したレジスタ状態スナップショットがある場合は、デバッガの状態を保存し、将来の再構成シミュレーションのためにそれを利用可能にする。

【0187】

・デバッガが次のステップを取るCPUについての正確なレジスタ状態情報を有する場合、増分的に後の時点でターゲット状態を決定するために（「再構成シミュレーション」セクションに記載されているものなどの）技術を使用する。本明細書で使用する場合、「増分的に後」との用語は、使用されるシミュレータの機能に応じて、「1つの命令の後」または「1つのサイクルの後」として、またはなんらの他の方法において、定義することができる。

【0188】

・デバッガが次のステップを取るCPUの正確なレジスタ状態情報を有していない場合、増分的に後の時点にメモリ状態を移動させるために「メモリ再構成」セクションで説明したものなどのメモリ再構成技術を適用する。

【0189】

・宛先ポイントに到達するまで、上記の手順を繰り返す。

【0190】

3つのCPUで構成されるシステムのレジスタ状態再構成の例が図17である。図において、各行（X301，X302，X303）は、システム内の特定のCPUについてロギングされたデータを表す。（X304における垂直線によって表される）宛先ポイントでのレジスタ状態を再構成するために、改善バックエンドは再構成ポイント（X305における垂直線）でメモリを再構成することによって開始できる。CPU#1（X301）および#2（X302）のレジスタ状態は宛先ポイント（X304）で未知であり、それは、コードがその時点でそれらのCPU上で実行中であり、よって、再構成ポイントを、これらのCPUについてのスナップショット（X306およびX307）が再構成ポイント（X305）と宛先ポイント（X304）との間で発生するように選択しなければならないからである。いかなるコードも宛先ポイントにおいてCPU#3上で実行されていないので、CPU#3上でレジスタ状態を再構成する理由はありません。したがって、再構成ポイント（X305）と宛先ポイント（X304）との間でCPU#3上でのレジスタ状態スナップショット（例えばX308）が発生する必要はない。

【0191】

改善バックエンドが再構成ポイント（X305）におけるメモリ状態を再構成すると、それは次のように時間的に前方にメモリおよびレジスタを再構成し得る。CPU#1の状態を再構成するために、それはレジスタ状態スナップショット（X306）の前に領域（

10

20

30

40

50

X309)を介してメモリ再構成を実行し得、その後、命令セットシミュレータに利用可能なスナップショット(X306)から状態を作成し、そして、そのシミュレータを使用してスナップショットと宛先ポイントとの間の領域(X310)において再構成シミュレーションを実行する。CPU#2についての状態を再構成するために、デバッガは直ちに命令セットシミュレータに利用可能なスナップショット(X307)から状態を作成し得、そして、そのシミュレータを使用してスナップショットと宛先ポイントとの間の領域(X311)上で再構成シミュレーションを実行する。CPU#3の状態を再構成するために、再構成ポイントと宛先ポイントとの間の領域(X312)全体上でメモリ再構成が使用され得る。

【0192】

システムが1つのCPUのみを含む場合、記載されたレジスタ再構成プロセスを、以下のように低減できることに留意されたい。

【0193】

宛先ポイントから開始して：

【0194】

- ・1つのCPUの状態が既知である場合は、再構成が必要とされない。さもなければ次のように続行する：

【0195】

- ・1つのCPUについてのレジスタ状態スナップショットが存在している最も近い時点に戻るなど、再構成ポイントにメモリの状態を移動し戻すために「メモリ再構成」セクションで説明したものなどのメモリ再構成技術を使用する。

【0196】

- ・デバッガのレジスタ状態スナップショットからの状態を保存して、再構成シミュレーションに利用可能にする。

【0197】

- ・増分的に後の時点でターゲット状態を決定するために（「再構成シミュレーション」セクションに記載されているものなどの）技術を使用する。

【0198】

- ・宛先ポイントに到達するまで、前の手順を繰り返す。

【0199】

《特定の宛先への後方／前方実行》

いくつかの実施形態では、改善バックエンドは、特定の宛先への後方／前方実行などの様々な時間走行デバッグ動作を実行するための基礎として本明細書に記載の技術を使用することができる。いくつかの典型的な宛先は、特定のメモリ位置に保存された命令が実行される次／前のポイント、特定のメモリ位置またはレジスタが修正される次／前のポイント、または、特定のタイムスタンプなどの特性の区別によって識別される特定のイベントである。

【0200】

ほとんどの場合、特定の宛先への実行は、3つのステップで達成され得る：（例えば「メモリ再構成」セクションに記載された技術を用いて）宛先が検出されるまで前方または後方にメモリを再構成すること、（例えば本明細書に記載された技術のいずれかを使用して）停止すべき箇所の正確な命令を決定すること、そして（例えば「レジスタ状態再構成」セクションに記載された技術を用いて）その時点でレジスタ状態を再構成すること。

【0201】

《単一の実行のスレッドのデバッグおよび全体システム》

いくつかの実施形態では、本明細書に記載されるような複数のソースからマージされたログ上で動作する改善バックエンドが、単一の実行のスレッドで、記録された実行のスレッドのサブセットで、または必要に応じてログ全体で、動作を行うことができる。改善バックエンドは、そのスレッドに関連しないデータを無視することによって、単一の実行のスレッドからのデータ上で動作することができる。例えば、図22は、スレッド1に関す

10

20

30

40

50

るデータ（X802）および他のスレッドに関するデータ（X803）を含むマージされたログ（X801）を示す。データは、本明細書に記載されたものなどの1つ以上のマージ技術を使用して単一ログ（X801）にマージされ得、それにより、それらはマージされたログにおいて論理的に狂った順序とならない。この例ではプログラマは、時間走行デバッガが、スレッド1に関するログの後のセクション（X804）で始まり、スレッド1に関するログの前のセクション（X805）で終わるバックステップ動作などのスレッド1に特有の動作を実行するよう要求することができる。バックステップ動作の開始ポイントおよび終了ポイントを決定するために、改善バックエンドはスレッド1に関するデータ（X802）のみを考慮する必要がある。しかし、バックステップ動作を行う際に、改善バックエンドは、スレッド1に関しないデータを含む可能性があるマージされたログからのデータの範囲で動作する、本明細書に記載のものなどの技術を使用してメモリ再構成およびレジスタ再構成を適用することができる。結果として、バックステップなどの単一の実行のスレッドにのみ関する動作を行うと、他の実行のスレッドのシミュレートされた状態を変更して、論理的に整合性のあるシステムの全体の状態を維持することができる。

【0202】

また、全体としてシステム上で動作する改善バックエンドの能力とは、その終了条件が全体としてシステム内の任意の条件に基づく実行制御動作を行うことができることを意味する。例えば、特定の関数をデバッグしたいユーザは、任意のCPU上での任意の実行のスレッドがその関数を実行するまで、実行し戻すことができる。または、ユーザは、1つの実行のスレッドでバックステップを実行することができるが、任意のCPU上での任意の実行のスレッドが特定のメモリ位置を修正した場合にバックステップが中断されるべきであると指定することができる。そのため、改善バックエンドを採用する時間走行デバッガにより、プログラマは、競合条件バグ、メモリ破損バグ、および、改善バックエンドなしでは追跡がはるかに困難または不可能である多くの他の種類の問題を検出およびデバッグすることができる。

【0203】

《実行された命令のリストの生成》

本明細書に記載の機能を使用して、改善バックエンドは、いくつかの実施形態では、典型的には時間走行デバッガによって必要とされるステッピングおよび実行動作の一部またはすべてを実行することができる。このような動作はしばしばマシン命令粒度で動作し、これは、デバッグ中のシステム内のCPUによって実行される個々のマシン命令がそのような動作を実行するために異なる時点が考慮されることを意味する。改善バックエンドは一意かつ明確にマージされたログ内の個々のマシン命令の実行を表すことができる。個々のマシン命令は、マージされたログ内の位置およびメモリ内のマシン命令のアドレスによって一意かつ明確に表すことができる。

【0204】

いくつかの実施形態では、（「開始命令」と称され得る）個々のマシン命令の実行の一意の表現を与えられると、改善バックエンドは、その実行のスレッドによって実行される以前の命令（「以前の命令」と称され得る）の一意の表現を決定することができる。ログデータを解析することにより、改善バックエンドは、開始命令が現在の基本ブロックで実行される最初の命令であるか否かを決定することができる。開始命令が現在の基本ブロックにおいて実行される最初の命令ではない場合は、以前の命令が、現在の基本ブロックにおいて実行される以前の命令である。開始命令が現在の基本ブロックにおいて実行される最初の命令である場合は、改善バックエンドは、現在の実行のスレッドによって実行される前の基本ブロックについてログ内でスキャンバックし、その基本ブロック内で実行される最後の命令を決定することによって以前の命令を発見することができる。

【0205】

例が図23Aおよび図23Bに示されている。図示の例では、マージされたログ（X901）の抜粋は、スレッド1についての実行データを含む2つの基本ブロック（X902およびX903）、ならびに、他のスレッドに関するデータ（X904）を含む。

【0206】

例として開始命令は、アドレス0x104にある可能性があり、これはアドレス(0x104)およびログ位置(X903)によって一意に識別される。この場合、開始命令(0x104)は、現在の基本ブロック(X903)によって実行される最初の命令ではないので、以前の命令は、現在の基本ブロック(X903)内で実行される以前の命令(0x100)である。この以前の命令は、アドレス(0x100)およびログ位置(X903)によって一意に識別される。

【0207】

例として開始命令は、アドレス0x100にある可能性があり、これはアドレス(0x100)およびログ位置(X903)によって一意に識別される。この場合、開始命令(0x100)は、現在の基本ブロック(X903)によって実行される最初の命令である。改善バックエンドは、現在の実行のスレッドで実行される以前の基本ブロック(X902)を発見するためにログにてスキャンバックし、その基本ブロック(X902)によって実行される最後の命令(0x208)を決定することによって、以前の命令を決定することができる。この以前の命令は、アドレス(0x208)およびログ位置(X902)によって一意に識別される。

【0208】

いくつかの実施形態では、(「開始命令」と称され得る)個々のマシン命令の実行の一意の表現を与えられると、改善バックエンドは、その実行のスレッドによって実行される次の命令(「次の命令」と称され得る)の一意の表現を決定することができる。ログデータを解析することにより、改善バックエンドは、開始命令が現在の基本ブロックで実行される最後の命令であるか否かを決定することができる。開始命令が現在の基本ブロックにおいて実行される最後の命令ではない場合は、次の命令が、現在の基本ブロックにおいて実行される次の命令である。開始命令が現在の基本ブロックにおいて実行される最後の命令である場合は、改善バックエンドは、現在の実行のスレッドによって実行される次の基本ブロックについてログ内で前方にスキャンし、その基本ブロック内で実行される最初の命令を決定することによって次の命令を発見することができる。

【0209】

例として、図23Aおよび図23Bを参照すると、開始命令は、アドレス0x104にある可能性があり、これはアドレス(0x104)およびログ位置(X903)によって一意に識別される。この場合、開始命令(0x104)は、現在の基本ブロック(X903)によって実行される最後の命令ではないので、次の命令は、現在の基本ブロック(X903)内で実行される次の命令(0x108)である。この次の命令は、アドレス(0x108)およびログ位置(X903)によって一意に識別される。

【0210】

例として、開始命令は、アドレス0x208にある可能性があり、これはアドレス(0x208)およびログ位置(X902)によって一意に識別される。この場合、開始命令(0x208)は、現在の基本ブロック(X902)によって実行される最後の命令である。改善バックエンドは、現在の実行のスレッドで実行される次の基本ブロック(X903)を発見するためにログにて前方にスキャンし、その基本ブロック(X903)によって実行される最初の命令(0x100)を決定することによって、次の命令を決定することができる。この次の命令は、アドレス(0x100)およびログ位置(X903)によって一意に識別される。

【0211】

マージされたログに表される実行のスレッドによって実行される次または以前の命令を決定するための上記の手順を繰り返すことにより、改善バックエンドはマージされたログでの実行のスレッドによって実行されたすべての命令アドレスのリスト(X905)を生成することができる。これは、ハードウェアトレースプローブによって生成された実行データに匹敵し、それにより、前方および後方へのステップ、および、ブレークポイントへの前方および後方への実行を含む時間走行デバッガの動作をサポートするのに十分である

10

20

30

40

50

。実施形態では、改善バックエンドは、実行のスレッドごとに実行される命令のこのようなリストを伴うデバッガを提供することにより、時間走行デバッガとインタフェースすることができる。

【0212】

《高度ステッピング、実行およびブレークポイントサポート》

本明細書に記載の機能は、改善バックエンドが時間走行デバッガをサポートするのに十分ではあるが、改善バックエンドのいくつかの実施形態が、時間的に高いレベルの移動に関する責任の一部を実行するのに、より効率的である可能性がある。時間走行デバッガは、改善バックエンドでこれらの機能を利用するように構成されることができ、それにより、それ自身の作業負荷を低減する。

【0213】

いくつかの実施形態では、改善バックエンドは、バック単一命令ステップ動作を行うことができる。このような動作は、指定された実行のスレッドがその実行における以前の1つのマシン命令である時点までシミュレートされた環境を戻って移動する。例えば、改善バックエンドは、本明細書に記載の技術（または同様の効果を伴う異なる技術）を使用してバック単一命令ステップ動作を行うことができる。例えば、バック単一命令ステップは、以下のプロセスで行うことができる：

【0214】

- ・実行された以前の命令を決定するための本明細書に記載された技術または同様の技術を使用して、指定された実行のスレッドによって実行される以前の命令のアドレスおよびログ位置を決定する。

【0215】

- ・以前の命令のログ位置が現在の位置と異なる場合、本明細書に記載された技術または同様の技術を使用して、以前の命令のログ位置に到達するようメモリ再構成を実行する。

【0216】

- ・本明細書に記載された技術または同様の技術を使用して、レジスタ再構成を実行する。

【0217】

改善バックエンドは、いくつかの実施形態によって実行された以前の命令の代わりに実行される次の命令をターゲットとすることにより前方命令ステップを実行するよう同様の手順を使用することができる。代替的にいくつかの実施形態では、改善バックエンドは、本明細書に記載されるような再構成シミュレーションまたは類似のシミュレーション技術を用いて前方単一命令ステップを実行することができる。複数の技術が所与の移動動作を実行するために利用可能であるこのような場合に、改善バックエンドは、現在の状況下での技術の予想される効率に基づいて、または他の基準に基づいて、技術を選択するように構成されることができる。

【0218】

いくつかの実施形態では、改善バックエンドは、特定の条件が満たされるまで時間的に前方または後方に移動することができる。このような条件としては、おそらくはシステム内での実行のスレッドの指定されたサブセット内で指定されたアドレスまたはアドレスの範囲で命令を実行すること、指定されたメモリアドレスまたはアドレスのセットをターゲットするメモリ修正に遭遇すること、新たな実行のスレッドの作成などの特定のロギングされたイベントまたはロギングされたイベントのタイプに遭遇すること、ログ内の最も以前のまたは最新のイベントに遭遇すること、ロギングまたは合成されたタイムスタンプによって示されるような特定の時間に発生したロギングされたイベントに遭遇すること、プログラマが関心のあり得るなんらかの他の条件に遭遇すること、または、これらまたは他の条件で構成される任意のより複雑な条件を含むことができるが、これらに限定されない。そのような移動は典型的には、ログ位置または実行される命令が、停止の条件（複数可）を満たすことに遭遇するまで、必要に応じて本明細書に記載の技術または類似の技術を用いて、ログにおいて前方または後方にメモリを再構成することによって改善バックエンドにより達成することができる。改善バックエンドは、移動を完了して所望の時点でのシ

10

20

30

40

50

ステムの状態を再構成するために、本明細書に記載のレジスタ再構成技術または同様の効果を有する技術を使用することができる。

【0219】

いくつかの実施形態では、改善バックエンドは、追加のログデータをロギングまたは合成することによって増強されることができ、移動動作が停止することができる条件の種類を増加させる。例えば、改善バックエンドは、どのCPUレジスタが各基本ブロック内で修正されているかについての情報をロギングまたは合成することができる。そのような情報を、指定されたレジスタが修正されたときに移動を停止するために使用することができる。

【0220】

《高度なデバッグ機能》

いくつかの実施形態では、改善バックエンドは、改善バックエンドを採用しないデバッガでは可能ではない多くの強力なデバッグ機能を提供することができる。例えば、改善バックエンドは、改善バックエンドを採用しないデバッガでは容易にまたは全く検出することができないメモリ破損および／または競合を伴うバグを検出するために使用されることができる。改善バックエンドを採用しないデバッガは典型的には、単一CPU上で発生したときにこのようなバグを検出することに限定されているのに対し、改善バックエンドは、単一のCPU上でもしくはSMP構成などのメモリを共有する複数のCPU上で発生する場合、または、CPUがメッセージパッシングによってまたは他の手段によって通信する分散環境において、または、上記もしくは他の構成のうちの1つ以上を伴う複雑なシステムにおいて、このようなバグを検出し、プログラマがデバッグするのを助けることができる。例えば競合は、メモリ位置が複数の実行のスレッドによって読み出しおよび／または書き込みされる場合に発生する可能性がある。このような競合は、予測不能なシステム挙動および他のこのような診断困難バグにつながる可能性がある。

【0221】

このような高度なデバッグ機能の例として、メモリが破損している複雑なシステムであって、システム内での実行のスレッドが失敗してしまう場合を考える。改善バックエンドを採用する時間走行デバッガを使用するプログラマは、実行のスレッドが失敗したときに停止し、そして改善バックエンドに後方に実行するよう命令し、破損メモリが修正されたときに実行を停止するようにシステムを構成することができる。改善バックエンドなしの時間走行デバッガは、失敗した実行のスレッドによって、または、おそらくは失敗したスレッドと同じCPU上で実行中の別の実行のスレッドによって破損が引き起こされた場合に、破損の原因を発見するために、この技術を使用することができるのみであろう。改善バックエンドを採用する時間走行デバッガは、失敗した実行のスレッドによって、または失敗したスレッドと同じCPU上で実行中の別の実行のスレッドによって、または異なるCPU上で実行中の実行のスレッドによって、または独自に動作するもしくは別の実行のスレッドに代わって動作するカーネルによって、または正しくもしくは誤って破損メモリと同じ物理メモリにマップする仮想メモリアドレスに書き込むことによりメモリを破損する別の実行のスレッドによって、破損が引き起こされた場合に、破損の原因を発見するために、この技術を使用することができるであろう。

【0222】

《コールスタック深度計算および例外処理》

いくつかの実施形態では、改善バックエンドは、時間の経過とともにコールスタック深度の変化に関する情報をロギングまたは合成することができ、これは、本明細書に記載の移動技術または類似の技術と組み合わせて、改善バックエンドがスタック深度感受性移動を実装することを可能にできる。スタック深度感受性移動の例としては、これらに限定されないが、固定コールスタック深度でのソースコードの行の間で（それぞれ）前または後にステップすることができるソースレベル前方および後方ステップアップ、および、現在の関数を呼び出した関数へステップするフォワードおよびバックステップアップが挙げられる。例えば、ソースレベルバックステップは、現在の命令が以前のソース行に関連付けら

10

20

30

40

50

れた命令範囲内に収まり、コールスタック深度が変更されないままである限り継続する後方移動として実装することができる。

【0223】

いくつかの実施形態では、エントリブロックは、その関数の呼び出しの結果として関数において実行される最初の基本ブロックとして定義することができる。イグジットブロックは、関数の呼び出し元に実行を戻すコードを含む任意の基本ブロックとして定義することができる。改善バックエンドは、スタック深度がそれぞれ増加および減少したことを示すレコードをロギングする関数のエントリおよびイグジットブロックをインストルメント化することにより、コールスタック深度情報をロギングすることができる。基本ブロックレコードについてのログをスキャンして関連する基本ブロックがエントリブロックおよび／またはイグジットブロックであるかどうかを決定することによって、データがデバッグ中のシステムから収集された後に、改善バックエンドはまた、そのようなレコードを合成することができる。このような決定は、プログラムのマシン命令のディスアセンブリまたは解析によって、またはソースコードからマシンコードへとプログラムを変換するプロセスにて使用されるコンパイラおよび／または他のツールによって生成されたメタ情報を調べることによってなど、多くの方法で行うことができる。改善バックエンドは、ロギングまたは合成されたエントリおよびイグジットレコードまたはエントリおよびイグジットブロックのオンザフライ検出を使用して、ログを解釈しつつコールスタック深度の変化を認識することができる。

10

【0224】

いくつかのCPUアーキテクチャでは、改善バックエンドはまた、(「SP」と称され得る)スタックポインタレジスタを解析することによって、現在のスタック深度に関する情報を収集することができる。いくつかの実施形態では、改善バックエンドは、レジスタ状態スナップショットおよび本明細書に記載のものなどのレジスタ再構成技術またはCPUレジスタ値を再構成する他の技術を使用して、SPの値を決定することができる。いくつかの実施形態では、改善バックエンドはまた、SPへの変更をロギングするインストルメント化を含むことができる。いくつかの実施形態では、改善バックエンドは、SPを修正する命令について基本ブロックを解析することなどの技術によって、またはコンパイラまたはソースコードをマシンコードに変換するために使用される他のツールによって生成されたメタ情報を読み出すことによって、ログデータがシステムから収集された後に、SP変更レコードを合成することができる。

20

30

【0225】

コンピュータプログラムは、例えばC++のトライ／キャッチ機構およびCのlong jmp機構などの、一度にコールスタックのいくつかのレベルを終了するための機構(「例外処理機構」と称され得る)を使用することができる。コールスタック深度のSPベースの決定は一般的に、このような例外処理機構の存在下で作動するが、エントリおよびイグジットブロックのインストルメント化は、複数のコールスタックレベルを同時に終了させることができるときに正確に相対コールスタック深度を計算するにはそれ自体では十分ではない。このような場合には、改善バックエンドは、元に戻されるコールスタックレベルの数を決定し、それに応じて変更をロギングする例外処理コードにおける追加のインストルメント化を使用することができる。

40

【0226】

改善バックエンドは、いくつかの実施形態では、状況に最良に合わせて、または各アプローチの長所と短所を最大限に活用するために、SPベースのコールスタック深度決定、エントリ／イグジットブロックベースのコールスタック決定、または他のコールスタック決定技術の組み合わせを使用することができる。

【0227】

《一意にロギングされたイベントの識別》

いくつかの実装では、一意にロギングされたイベントが識別できるようにすることが有用であり得る。例えば、時間走行デバッガ、または改善バックエンドによって収集された

50

データを利用する他のデバッグツールは、プログラマにデータのグラフィカル表現を提供し得る。例えば、このようなデバッグツールは、システムの実行中に発生した関心対象のイベントのタイムラインを表示するために改善バックエンドからのデータを使用することができる。この表現内でクリックするなど、ユーザインタフェース手段を介して、プログラマは、対応する時点へと時間走行デバッガを走行させたいという要望を示すことができる。これを達成するために、時間走行デバッガは、改善バックエンドに所望の宛先を指定することができ、改善バックエンドはそして、所望の時間におけるシステムの状態を再構成することができる。これを達成するために、改善バックエンドは、ログ内のすべての異なる時点（「モーメント」と称され得る）について一意識別子を提供することができる。モーメントは、限定されるものではないが、単一マシン命令の実行、単一オペレーティングシステムイベント、または2つの実行のスレッド間の通信の単一インスタンスを含むことができる。いくつかの実施形態では、改善バックエンドは、本明細書に記載されているように、移動動作の停止ポイントを制御する条件のコンポーネントとして、このような識別子を受け入れることができる。別の例では、改善バックエンドは、改善バックエンドによって収集されたデータをグラフィカルに表示するユーザインタフェースで使用するために、各モーメントに一意タイムスタンプを割り当てることができる。このようなユーザインタフェースは、プログラムパフォーマンスのビジュアル化および解析、システムのタイミング特性の解析、ならびに、システムの競合条件および他の同時実行関連態様の検出および解析を含むがこれらに限定されない多くのもののために使用されることができる。いくつかの実施形態では、ユーザインタフェースは、システムの状態を再構成する時点の選択などの、ユーザ活動に応答して動的に更新または変更される。例えば、ユーザインタフェースは、選択された時点でのメモリおよび／またはレジスタの状態（例えばメモリおよび／またはレジスタに保存されている値）を表示するように更新または変更されることができる。システムの状態は、システムの1つ以上のCPUの状態を含むことができる。

【0228】

実施形態では、改善バックエンドは、時間の線形測定値を含む識別子（「タイムスタンプ」と称され得る）および（任意選択的に）CPU識別子で一意に各モーメントを識別することができる。CPU識別子は、含まれている場合、イベントが発生したまたはログインされたCPUまたは実行のスレッドを表すことができ、それにより、2つの異なるCPUまたは実行のスレッドが同時にイベントを実行した場合をタイムスタンプの精度内で明確にするよう機能する。CPU識別子は、CPUまたはコアなどの実行ユニットを一意に識別する整数または他のなんらかのものとするすることができる。例えば、デバッグ中のシステムが4つの実行ユニットを伴う単一マルチコアプロセッサを含む場合、ユニットは、CPU識別子0、1、2および3により一意に識別することができる。別の例として、デバッグ中のシステムが、ネットワークを介して通信する20個の別々のコンピュータシステムを含み、それらの各システムが8コアCPUを含む場合、 8×20 、すなわち160の実行ユニットがあり、それらにCPU識別子0～159を割り当てることができる。所与の実行ユニットでの実行から生じるログエントリは、対応するCPU識別子でタグ付けされることができるので、改善バックエンドはシステムの実行の履歴を正確に再構成することができる。

【0229】

いくつかの実施形態では、CPUの実行において各モーメントに一意タイムスタンプを割り当てるために、改善バックエンドはリニアカウンタを使用することができ、これは、各モーメントについて少なくとも一単位、またはクロックによって測定される実際の時間に基づく測定値、または推定時間、またはログにおけるモーメントの位置に基づく測定値、または他の線形測定値、またはこれらまたは他の線形測定値の組み合わせだけ進む。

【0230】

特定の実装では、タイムスタンプを生成するために改善バックエンドで使用されるクロックは、各モーメントに一意時間を割り当てるために十分な精度または正確さを有していません。各モーメントについてタイムスタンプをロギングすることが実現可能でな

くともよい。このような場合には、改善バックエンドは、各モーメントが一意識別子を有していることを確実にするために、以下の方法などの補間方法を採用することができる。実施形態では、改善バックエンドは、（「実際のタイムスタンプ」と称され得る）クロックから読み出された値に基づいて（「合成タイムスタンプ」と称され得る）一意タイムスタンプを合成することができる。実際のタイムスタンプの精度が各モーメントを一意に表すのに不十分である場合、改善バックエンドは、任意選択的に追加の精度の1つ以上のビット（「サブタイムスタンプビット」と称され得る）を各実際のタイムスタンプに追加できる。改善バックエンドは、タイムスタンプ補間のプロセスを使用して、実際のタイムスタンプに対応しない各モーメントに一意合成タイムスタンプ値を割り当てることができる。補間の目的で、改善バックエンドは、実際のタイムスタンプ間の時間を均一にまたは不均一に細分化することができる。具体的には、改善バックエンドは、ロギングされたイベントの相対的な期間の推定値を使用して実際のタイムスタンプを不均一に細分することができ、それにより、合成タイムスタンプがそれぞれのモーメントの実際の時間の合理的な推定値となる。改善バックエンドによって生成されたタイムスタンプデータがソフトウェアパフォーマンスのビジュアル化および解析のために使用される場合、このような不均一に割り当てられた合成タイムスタンプは特に有用である可能性がある。

【0231】

タイムスタンプ補間を行う改善バックエンドの例が図25Aである。例では、2つのモーメント（X11a.1およびX11a.7）は、実際のタイムスタンプを有し、これらは、おそらくリアルタイムクロックから記録され、対応するイベントでロギングされている。残りのモーメント（X11a.2～X11a.6）は、実際のタイムスタンプを有していない。補間を可能にするために、例における改善バックエンドは、それぞれの実際のタイムスタンプに、3つの16進数の0で表される12個のサブタイムスタンプビットを付加する。最後に、例における改善バックエンドは、モーメントX11a.1およびX11a.7間の経過時間を均一に細分化することによってモーメントX11a.2～X11a.6の各々に合成タイムスタンプを割り当てる。

【0232】

不均一な細分化を伴うタイムスタンプ補間を行う改善バックエンドの例が図25Bである。この例は以下の点以外は図25Aのものと同様であり、この例では、改善バックエンドは、モーメントX11b.3およびX11b.4を他のモーメント（X11b.1～X11b.2およびX11b.5～X11b.7）よりも長い期間であると認め、それに応じてモーメントX11b.1およびX11b.7間の経過時間を不均一に細分化する。

【0233】

《ブックマークおよびアンドゥ》

ユーザまたはプログラマが、改善バックエンドを伴って実装された時間走行デバッガを使用する過程を経て、関心対象モーメントに遭遇した場合、改善バックエンドは、例えばユーザの要求に応じて、そのモーメント（「ブックマーク」と称され得る）に対応する識別子を保存するために命令され得る。実施形態では、改善バックエンドは、識別子を保存して後でそれを再呼び出しすることができ、ユーザがブックマークモーメントに容易に戻ることができる。いくつかの実施形態によれば、改善バックエンドは、本明細書に記載されるように移動動作の終了条件の一部として、このようなブックマークを使用することができる。例えば、改善バックエンドは、時間の移動を実行するように依頼されることができ、指定されたブックマークで表されるモーメントにおいてデバッグされるシステムの状態を再構成する。プログラマは、システムの実行中の関心対象モーメントの記録および再訪、ならびに、他のプログラマへの関心対象モーメントの通信を含むがこれらに限定されない多くの目的のために、ブックマークを使用することができる。

【0234】

いくつかの実施形態では、改善バックエンドは、自動的に1つ以上のデバッグセッションの過程にわたって訪問されたモーメントの履歴を維持するように構成されることができる。そのような履歴により、改善バックエンドが時間走行デバッガに「アンドゥ」機能を

提供することができ、これにより、単一キーストロークまたは他のユーザインタフェースコマンドによって、プログラマが時間走行デバッグに命令して、改善バックエンドのアンドゥ機能を活性化して訪問した最新モーメントにそれを戻すことができる。例えば、プログラマは、改善バックエンドを採用する時間走行デバッグを伴うシステムをデバッグすることができる。プログラマが誤って、デバッグに実行コマンドを発行し、関心対象時点から遠いポイントへの移動が生じる。「アンドゥ」機能がなければ、プログラマは、システム内の関心対象時点に戻ることが困難な可能性がある。「アンドゥ」機能があれば、プログラマは、例えば単一キーストロークで戻ることができる。

【0235】

実施形態では、アンドゥ機能の連続的適用により、プログラマのデバッグセッションの履歴を遡って改善バックエンドで示されるシステム状態を歩行することができる。第2のキーストロークまたはユーザインタフェースコマンドは「リドゥ」コマンドを実行することができる。同様にプログラマのデバッグセッションの履歴の前方へと改善バックエンドで示されるシステム状態を歩行することができる。このように、改善バックエンドにより、プログラマがデバッグ中のシステムの実行履歴だけでなくデバッグセッションの履歴そのものをステップスルーできる。ワードプロセッサにおけるこのようなアンドゥ機能により文書編集に生じたミスを元に戻すのがライターにとって迅速かつ容易になるのと同様に、改善バックエンドにおけるアンドゥ機能により、デバッグ中に生じたミスを元に戻すのがプログラマにとって迅速かつ容易なものとなる。ワードプロセッサにおけるこのようなアンドゥ機能によりライターは文書の改訂履歴を前後に移動することができるのと同様に、改善バックエンドにおけるアンドゥ機能により、プログラマがデバッグセッション中に訪問したモーメントを前後に移動できる。いくつかの実施形態では、「アンドゥ」または「リドゥ」機能の1つ以上を活性化することで、モーメントの履歴などのデバッグセッションの履歴を更新するための「アンドゥ」または「リドゥ」機能のユーザ選択に応じて、ユーザインタフェースが動的に更新または変更される。

【0236】

《ロギングされたデータの自動解析》

時間走行デバッグをサポートすることに加えて、いくつかの実装では、改善バックエンドは、アプリケーション実行中にこのようなバグにより不正アプリケーション挙動が生じなかった場合でも、（他のタスクの間でも）ロギングされたデータの自動解析、多くの異なる種類のバグの検出を行うことができる。このようなバグは、特定の再現困難な状況下でのみ、すなわち非常にまれにしかアプリケーションの不正挙動として現れ得ず、改善バックエンドの支援なしには発見およびフィックスするには非常に困難であり、時間がかかる。

【0237】

実施形態では、改善バックエンドは、マルチスレッドアプリケーションにおける競合条件を検出するためにロギングされたデータの自動解析を使用することができる。改善バックエンドは、アプリケーションまたはシステムが実行された後にこのような検出を行うことができ、したがって、システムに追加のランタイムパフォーマンスペナルティを招くことがない。そのような改善バックエンドは、メモリアクセス（読み出しおよび書き込み）ならびにセマフォおよびミューテックスなどの同期プリミティブの使用についてのデータを収集し、そのデータを解析し、共有メモリおよび／または他の共有リソースの安全でない使用を探す。例えば、適切な同期を伴わない同じメモリ位置への書き込み実行の2つのスレッドは、競合を示す可能性があり、これは、発見することが困難なバグにつながる可能性があり、まれにしかアプリケーションの不正挙動とならない。改善バックエンドは、自動的にそのような競合を検出して、プログラマに報告することができる。このような報告は、競合メモリ位置または共有リソース、競合に関与したスレッドのアイデンティティ、および非安全に同一の共有リソースについて競合する命令のアドレスを含むがこれらに限定されない、バグフィックスのために有用な情報を含むことができる。これは典型的には、プログラマがすぐにバグをフィックスするのに十分な情報である。

【0238】

実施形態では、改善バックエンドは、対応するスタックフレームまたはヒープ割り当てから外れるスタックおよび／またはヒープ位置へのメモリアクセスを検出するためにロギングされたデータの自動解析を使用することができる。そのようなメモリアccessオーバーフローバグは、まれな状況でしか不正なアプリケーション挙動として現れ得ず、したがって改善バックエンドなしでは検出、発見およびフィックスが困難であり得る。

【0239】

時間走行デバッガおよび非時間走行デバッガの両方を含む多くのデバッガおよび開発ツールは、インストルメント化（「ランタイムエラーチェックインストルメント化」と称され得る）が手動または自動でコンピュータプログラムに挿入されて、様々なバグの検出を支援し得る特徴を有する。そのようなランタイムエラーチェックインストルメント化は、不良ポインタを介するメモリアccess、アレイバウンドを超えるメモリアccess、ゼロ除算エラー、メモリ割り当ておよび割り当て解除ライブラリの不正使用等などの多くの問題を検出することができる。そのようなランタイムエラーチェックインストルメント化は、をするプログラムフットプリントおよびランタイムパフォーマンスにおいてオーバーヘッドを追加する。このようなオーバーヘッドは、特定の問題がもはや現れないポイントにアプリケーションの挙動の変化を引き起こす可能性があり、または、アプリケーションを効率的におよび／または正しく動作させない。いくつかの実施形態では、改善バックエンドは、同じチェックを実行するためにロギングされたデータの自動解析を使用することができるが、改善バックエンドインストルメント化のオーバーヘッドを超えるプログラムフットプリントおよびランタイムパフォーマンスにおける追加のインストルメント化またはオーバーヘッドを必要としない。改善バックエンドはそれゆえ、はるかに汎用性の高い方法でランタイムエラーチェックインストルメント化の機能を包括することができる。改善バックエンドは、アプリケーションプログラムの再コンパイルまたは再実行を必要とすることなく、収集されたデータの単一ボディについてのチェックを幅広く行うことができる。改善バックエンドはまた、必要な計算量またはその他の要因により、ランタイムに実行するにはあまりにも破壊的または面倒になるであろうことのチェックを行うことができる。

【0240】

特定の実装では、改善バックエンドは、デバッガおよび／または他のツールとユーザインタフェースとに自動解析の結果を利用できるようにすることができる。実施形態では、自動解析中に改善バックエンドによって発見されたバグ、潜在的バグ、競合および他の関心対象事象は、システムの過去の状態の改善バックエンドによる再構成を停止するための条件として取り扱うことができ、プログラマが、システムのロギングされた履歴をステップスルーまたは実行して、潜在的バグの事象、あるいは改善バックエンドなしでは発見するのが非常に困難かもしれないような他のそのような関心対象事象で停止するのを容易にする。

【0241】

実施形態では、自動解析中に改善バックエンドによって発見されたバグ、潜在的バグ、競合および他の関心対象事象は、グラフィカルな形式で表示されることができ、それにより、ユーザは、個々の事象をクリック（または別のユーザインタフェースによってそれらを選択）することができ、事象に関する情報の表示、および／または、改善バックエンドによる事象発生時におけるターゲットの状態の再構成を含むがこれらに限定されない、いくつかのアクションの1つ以上が得られ、よって、ユーザがデバッガでこのような状態を調べることができる。このようなユーザインタフェースは、ユーザアクティビティまたは選択に応じて動的に更新または変更されることができ、それにより、ユーザが自動解析中に改善バックエンドによって発見された任意のバグ、潜在的バグ、または他の関心対象事象をすばやく移動できるようにすることができ、そしてすぐにその原因および／またはその発生の頻度およびパターンについての情報を収集することができる。実施形態では、このようなユーザインタフェースは、時間軸を伴うグラフ、チャート、表、またはビジュアル化の他の形態上で自動解析の結果を表示することができる。そのような表示は、そうで

10

20

30

40

50

なければ検出することが困難であろうことが非常に明白な一定時間ベースのパターンを作ることができる。実施形態では、これらのイベントを表示するためのユーザインタフェースは、表示されたイベントを非表示、並べ替え、却下、および／または別様に編成および制御するための手段を提供することができ、ユーザが最も直接的な関心を有するイベントに集中するのを容易にする。

【0242】

《コードカバレッジおよびプロファイリング解析》

いくつかの実施形態では、改善バックエンドによってロギングされたデータの解析を、本明細書に記載されるように、ログに表される実行についてのコードカバレッジデータを生成するために使用することができる。コードカバレッジデータは、プログラムの実行中に実行されたソフトウェアプログラム内の実行可能コードのサブセットに関する情報を含むことができる。コードカバレッジデータは、改善バックエンドによって生成されたログをスキャンし、基本ブロック実行情報を集約し、ログに表される時間の間に実行される各基本ブロック（またはその一部）のリストを生成することによって生成することができる。このリストは、プログラム内のすべての実行可能コード、または、プログラムの関心対象のプログラム内のコードのサブセット（例えば、システムおよび／またはサードパーティライブラリを省略する単なるユーザ書き込みコード）のリストと比較されることができる。実施形態では、得られたコードカバレッジデータをテーブル内に表示することができる。実施形態では、結果として得られるコードカバレッジデータは、ソースコードと併せて、例えば、ソースコード編集ウィンドウもしくはソースレベルデバッガにおけるコードの未実行の行を強調表示することによって、または他の方法で提示されることができる。実施形態では、本明細書に記載の技術は、テストスイートがソフトウェアプログラム内の実行パスをいかに良好にカバーするかを決定するために、テストスイートの実行中にコードカバレッジ情報を生成するために使用されることができる。実施形態では、結果として得られたコードカバレッジデータを、修正条件／決定カバレッジ解析（「MC／DC解析」）または他のそのようなソフトウェアテスト解析のために使用することができる。実施形態では、結果として得られたコードカバレッジデータを、DO-178B（「エアボーンシステムおよび機器のソフトウェア考慮事項」）認証、IEC 61508（「機能安全」）認証、ISO／IEC 15408（「コモンクライテリア」）認証、およびその他を含むがこれらに限定されないコードカバレッジ解析を必要とする多くの種類のソフトウェア認証のために使用することができる。改善バックエンドによってロギングされたデータからコードカバレッジデータを生成することは有利であり、それは、既に改善バックエンドによって必要とされるものを超えてコードカバレッジデータを生成するために追加のインストルメント化が必要とされず、コードカバレッジデータを生成するための定期的なサンプリングなどの不正確な技術に頼る必要はないからである。

【0243】

いくつかの実装形態では、改善バックエンドによってロギングされたデータの解析は、本明細書に記載されるように、ログに表される実行についてのプロファイリングデータを生成するために使用されることができる。そのようなプロファイリングデータは、各基本ブロックが実行される頻度に関する情報、および／または、所与のメモリアドレスがアクセスされる頻度についての情報を含むことができるが、これらに限定されない。そのような情報は、改善バックエンドによって生成されたもののスキャン、ならびに、実行される各基本ブロック（またはその一部）のカウントおよび／または書き込まれた各メモリアドレスのカウントを生成することによるデータの集約によって生成されることができる。実施形態では、このようなプロファイリングデータは、テーブル内でユーザに提示され、および／または、（例えば、最も頻繁に書き込まれたメモリアドレスを色分けすることにより）メモリマップ上にオーバーレイされ、および／または、（例えば、ソースコードエディタ内で、ソースレベルデバッガ内で、または他の方法で、各ソース行が実行された回数、または、各ソース行によって表される全体的な実行時間の割合を表示することにより）ソースコードの表現にオーバーレイされることができる。このようなプロファイリングデ

ータは、プログラマによって使用され得、コードの最も頻繁に実行されるセクションを合理化することによって、または、最も頻繁にアクセスされるコードおよびデータのセクションをメモリおよび／またはキャッシュ効果の速度がプログラムの実行速度に最も有益であるメモリのセクションに移動することによってなど、プログラムのパフォーマンスが向上する。改善バックエンドによってロギングされたデータからプロファイルデータを生成することは有利であり、それは、プロファイリングデータを生成するのに追加のインストールメント化が必要ではなく、プロファイリングデータを生成するための定期的なサンプリングなどの不正確な技術に頼る必要はないからである。

【0244】

実施形態では、本明細書に記載されるように生成されたコードカバレッジおよび／またはプロファイリングデータを、手動および／または自動回帰テストに使用して、ソフトウェアパフォーマンスおよびテストカバレッジを含むがこれらに限定されない様々な指標が経時劣化していないことを確認することができる。

【0245】

《未知の値》

ほとんどの場合、改善バックエンドの特定の実装が正確にメモリおよびレジスタの内容を再構成することができるが、データ値の再構成が可能ではない場合がある可能性がある。例えば、改善バックエンドは、例えば副作用揮発性メモリに保存する際に、メモリ変更についてのプリイメージ値をロギングしないかもしれない場合がある。また、ロギングされたデータの自動解析または他のそのような解析技術は、改善バックエンドがロギングされたイベントの正しい順序を特定することができないので、メモリまたはレジスタ値が未知である状況をもたらし得る。このような状況は一般的に、そのようなソフトウェアがSMP環境で実行されると、解析されているソフトウェアでの競合条件が原因で発生するが、他の原因も同様に可能である。このような場合についての特別な処理がなければ、デバッガがメモリまたはレジスタの内容を不正にプログラマに報告する可能性があり、混乱を招き、バグの発見およびフィックスが困難になる。

【0246】

実施形態では、改善バックエンドは、既知の値とは異なる未知のメモリおよび／またはレジスタ値を扱う。このような未知の値が検出されると、改善バックエンドは、どのメモリ位置および／またはレジスタ値が未知であるかを追跡することができる。改善バックエンドは、未知のメモリ位置および／またはレジスタ値をデバッガに報告することができ、それにより、それらはプログラマに異なって表示され、プログラマが既知の値と区別することができる。

【0247】

実施形態では、未知の値で動作する改善バックエンドは、例えば（本明細書に記載の）再構成シミュレーションを行う場合、計算の結果に未知性を伝播することができる。例えば、2つのレジスタの値を加算し、第3のレジスタに結果を配置する命令をシミュレートする改善バックエンドは、加数のいずれかまたは両方が未知である場合、第3のレジスタの値を未知であるとマークすることができる。同様に、レジスタにメモリから値をロードすることをシミュレートする改善バックエンドは、ロードされたメモリ値が未知の場合にレジスタの値を未知であるとマークすることができる。そのような伝播技術を使用して、改善バックエンドは、それが実際に既知である場合は、それが値を「既知である」と報告するのみであろうことを保証することができる。

【0248】

実施形態では、改善バックエンドは、値の未知状態が伝播されるべきでない場合を知ることができる。例えば、2つのレジスタの値を減算する命令をシミュレートする改善バックエンドは、減算される値が未知であっても、このような減算が常にゼロ結果となるような場合など、減算された値が同じであることが既知である場合、「既知である」と結果をマークし得る。

【0249】

実施形態では、改善バックエンドは、それらの値が既知になったときに既知であると未知の値をマークすることができる。例えば、レジスタの値がレジスタ状態スナップショットから決定することができる場合に、改善バックエンドは、そのレジスタの値を既知であるとマークすることができる。同様に、命令が既知の値でレジスタの内容を上書きすることをシミュレートされたときに、その値が以前に既知でなかったとしても、レジスタの内容を既知であるとマークすることができる。このような技術を使用して、改善バックエンドは、それが実際に未知である場合に「未知」として値を報告のみすることを保証することができる。典型的な場合では、プリイメージを含むレジスタ状態スナップショットおよびメモリストアレコードが頻繁に発生する可能性があり、よって、所与の値は、アプリケーションまたはシステムのロギングされた履歴内で未知のままである期間は、典型的には限られている。

10

【0250】

実施形態では、未知のメモリおよび／またはレジスタ値は多くの場合、デバッグされるプログラムまたはシステム内の競合によって引き起こされ、それにより、潜在バグが存在していることのプログラマにとっての有用な指標とすることができる。別の方法で表現すると、改善バックエンドが、実施形態において、メモリ位置またはレジスタの正しい値を決定することができない場合は、プログラマが最も修正したい可能性の高いシステム内欠陥を示すようには、正しい値をプログラマが確認することができないようにコンピュータシステムおよびソフトウェアが構成されていることを示すことができる。自動的にこのような場合を検出し、プログラマにそれらを表現する改善バックエンドの能力はしたがって、極めて貴重なデバッグツールである。実施形態では、改善バックエンドは、このような競合条件の詳細を、直接にまたは時間走行デバッガまたは他のこのようなユーザインタフェースでインタフェースすることによってプログラマに表示するように構成されることができ、これは、ユーザアクティビティまたは選択に応答して動的に更新または変更することができる。検出されてプログラマに示される競合条件の詳細は、同じメモリ位置へのアクセス衝突のソースコード内の位置、そのような衝突が発生したロギングされた実行中の時間（複数可）、および、そのような各衝突中および後のメモリ位置の可能な値を含むことができるが、これらに限定されない。

20

【0251】

《例示的ユースケース：SMP競合条件》

いくつかの実施形態では、改善バックエンドは、本明細書に記載されるように、時間走行デバッガおよび／または他のデバッグおよび解析ツールと関連して使用される場合、そうでなければ可能ではない方法でプログラマが複雑なコンピュータシステムの問題を診断およびデバッグすることを可能にする。本明細書に記載のシステムおよび技術を使用することにより、そうでなければ発見およびフィックスに数週間ないしは数か月かかるかもしれない、または全くフィックスされないかもしれない困難なバグを、典型的には数時間ないしは数日で解決することができる。

30

【0252】

例えば、バグの一般的な種類は、同時の実行のスレッドによって共有されるメモリ位置へのアクセスが適切に守られておらず、各種の不安定な挙動となってしまう競合条件である。マルチコアCPU（例えば、デュアルコア、クワッドコアなど）の使用が増加して、そのようなバグがますます一般的になってきており、それは特に、シングルコアCPUのために書かれレガシーコードがより現代的なCPUにポートされるからである。従来のデバッグ技術では、そのようなバグの追跡に対して少しの助けにしかならず、プログラマはあまりそれに頼ることはできないままであり、コードを慎重に解析しなければならず、プロセスには数週間以上かかり、そして結果が得られないことがある。

40

【0253】

競合条件の例が図27A～Dに表示される。例では、カウンタ（「カウンタ」）が2回インクリメントされる。「カウンタ」の初期値は4であり、よって「カウンタ」の最終的な値は、2つの成功したインクリメント動作を反映して6となるべきである。各インクリ

50

メント動作は、読み出し動作（例えばX14a-read1）および書き込み動作（例えばX14a-write1）からなる。

【0254】

単一コアの例（図27A）では、両方のインクリメント動作は、単一コア（X14a-core0）によって実行されるので、何の競合も発生していない。インクリメント動作（X14a-inc1、X14a-inc2）が順次、行なわれ、続く読み出しおよび書き込み動作（X14a-read1、X14a-write1、X14a-read2、X14a-write2）も厳密な順序で発生する。

【0255】

図27Bは、デュアルコアSMP環境で実行されるインクリメントの同一の対を示している。第1のインクリメント動作（X14b-inc1）は、コア0（X14b-core0）上で行われ、第2の動作（X14b-inc2）は、コア1（X14b-core1）で行われる。この場合は幸運であり、2つのインクリメント動作（X14b-inc1およびX14b-inc2）は時間的にばらばらであり、順次、実行される。予想されるように「カウンタ」の最終的な値は6である。

【0256】

図27Cは、不運であり、時間内の2つのインクリメント動作（X14c-inc1、X14c-inc2）がオーバーラップしてしまう点以外は図27Bと同様である。第2のインクリメント動作（X14c-inc2）は、第1のインクリメント動作（X14c-inc1）の完了前、具体的には第1の書き込み動作（X14c-write1）の発生前にその読み出し動作（X14c-read2）を実行する。その結果、第2の読み出し動作（X14c-read2）にて読み出された値が5の代わりに4である。第2の書き込み動作（X14c-write2）が発生すると、「カウンタ」に書き込まれた最終的な値は、6の代わりに5である。競合条件のため、「カウンタ」の値は、2つのインクリメント動作の実行にもかかわらず、4から6への代わりに4から5へと進む。

【0257】

図27Cにおいて実証されたものなどの問題は、それらが2つのコア（X14c-core0、X14c-core1）におけるインクリメント動作（X14c-inc1、X14c-inc2）の相対的タイミングに依存するので、不規則に発生し得る。さらに、ソースコードが正しく見えて2つのインクリメント動作を示すが、2つのインクリメント動作があるのに「カウンタ」が1しか進まない理由が明白ではないので、プログラマがソースコードを読み取ることによってそのようなバグを識別するのは非常に困難である可能性がある。改善バックエンドがなければ、時間走行デバッグはこの種のバグの発見およびフィックスにはあまり有用ではなく、それは、典型的には（図27Aに示すような）単一コア上でプログラムを実行して問題発生を防止する必要があるからである。

【0258】

ある実装では、（本明細書に記載の）改善バックエンドを伴う時間走行デバッグおよび／または他のビジュアル化ツールを使用するプログラマは、迅速かつ簡単に図27Cに示したものなどのバグを発見するためにいくつかの技術のうちの1つ以上を使用することができる。例えば、競合条件を疑うプログラマは、（本明細書に記載の）ロギングされたデータの自動解析を使用でき、改善バックエンドは、第1および第2の書き込み動作（X14c-write1、X14c-write2）が適切に守られていないことを検出し、プログラマに通知する。自動解析技術は、実際の実行が正しい最終値が生成された図27Bに似ている場合であっても、潜在的問題を識別するであろうことに留意されたい。

【0259】

競合条件を発見するためのこの自動解析技術は、非常に強力である可能性がある。一般的に使用されているデバッグの大半は、ソフトウェアアプリケーションを用いて問題に気づくよう、またはユーザより問題について通知され、問題の原因を発見するようプログラマを支援する。これとは対照的に、この自動解析技術は、プログラマおよびプログラムのユーザがまだ気づいていない、および／または、まだ不正挙動として現れていないソフト

10

20

30

40

50

ウェアアプリケーションに伴う問題を発見することができる。入力タイミング、キャッシュ挙動、CPU挙動、割り込みタイミング、および／または他の再現困難要因の変動のため、このような潜在的バグは、100回に1回、1,000回に1回、1,000,000回に1回、またはさらに少ない頻度で、一見ランダムな時間に不正挙動として現れるにすぎない。リリースされたソフトウェア製品、または、携帯電話、自動車エンジン、アビオニクスデバイス等などの正しく動作するべくソフトウェアに依存する製品では、そのようなバグは、製品が普及するまで気づかれない不安定な挙動になる可能性があり、これは、製品開発者の評判、および、バグを発見してパッチを適用するのにかかる労力の両方で非常にコストがかかる可能性がある。このような費用は、数百万ドル以上に及ぶ可能性があり、また、市場シェアの回復不能な損失をもたらす可能性もあり、これは、競合条件を発見するための自動解析技術を用いて完全に回避することができるため、このようなデバッグ機能はソフトウェア開発者にとって非常に貴重なものとなる。

10

【0260】

いくつかの実施形態によれば、改善バックエンドが、この種の競合条件を追跡するのにプログラマにとって貴重である可能性がある他の方法がある。第2の例では、「カウンタ」の疑わしい値に注目するプログラマは、改善バックエンドを備えた時間走行デバッガを使用することができ、これは、時間を遡ってプログラムを実行し、「カウンタ」の値が修正された時点で停止し、それらの時点で「カウンタ」の再構成された値を表示する。プログラマは、「カウンタ」をインクリメントすることになっている時間的に隣接する2つの動作が、両方とも代わりに5にその値を設定しており、インクリメント動作が互いに干渉しているかもしれないことに気づくであろう。

20

【0261】

第3の例では、プログラマは、時間経過とともに「カウンタ」の値をグラフ化して「カウンタ」が修正されたポイントを強調表示するグラフィカルツールヘデータを提供するように改善バックエンドの実施形態を使用することができる。そのようなツールでは、プログラマは、2つのインクリメント動作がほぼ同時に発生することを容易に確認することができ、「カウンタ」の値は1だけ変化するのみである。プログラマがより密接にプログラムの動作を検査したい場合、インクリメント動作の1つをクリックすると、改善バックエンドを備えた時間走行デバッガに、選択された時点での両方のコアの活動を表示させることができる。

30

【0262】

これらの例の3つすべては、いくつかの実施形態による改善バックエンドの独自の機能に依存している。改善バックエンドなしの時間走行デバッガは一般的にまったく同時実行から生じる問題をデバッグすることができないのに対し、改善バックエンドは、それができ、本明細書に記載されるものなどの競合条件の検出およびデバッグを可能にする。改善バックエンドの特定の実装により可能となる本明細書に記載されているものなどの技術を使用して、プログラマは、ほんの数分で問題の原因を特定することができる一方、改善バックエンドなしでは、プログラマは問題を特定するために、あまり効率的ではない技術または完全当て推量に頼ることを余儀なくされ、これは、数日、数週またはそれ以上かかる可能性があり、多くの場合、全ての問題をフィックスすることは非現実的または経済的に実行不可能である。

40

【0263】

プログラマが競合の原因を識別すれば、図27Dに示すように問題をフィックスすることは比較的容易である。ミューテックスロック(X14d-lock)でインクリメント動作(X14d-inc1、X14d-inc2)を包むなど、同期プリミティブを適用すると、続く読み出しおよび書き込み動作(X14d-read1、X14d-write1、X14d-read2、X14d-write2)は強制的にすべての場合において厳密な順序で発生することになり、コア(X14d-core0、X14d-core1)の相対的タイミングにかかわらず、「カウンタ」について6との正しい最終的な値が得られる。

50

【0264】

《他の変形例》

本明細書の開示は、垂直統合インストルメント化についての方法、システム、およびデバイス、ならびに、従来のデバッガと比較してデバッガのコンピュータ技術の機能、速度および動作を劇的に改善するトレース再構成システムを提供する。実施形態では、本明細書に開示されるシステムおよび方法は、ターゲットプログラムの実行中に発生したイベントの再生および／または再構成を提示するのに、ある場合には10倍以上の係数で時間を減少させることによって、ソフトウェアデバッガ技術分野における改善をもたらすように構成されることができる。

【0265】

実施形態では、本明細書に開示されるシステムおよび方法は、トレースデータログにギャップまたは欠落データがないか、または実質的にないトレースデータを生成するように構成されることができる。比較すると、従来のデバッガシステムは一般的に、システムがトレースデータログファイルにトレースデータを記録するのを困難にするかなりの数のメモリアクセスを生成するターゲットプログラムの命令シーケンスの間に特に、トレースデータのギャップまたは欠落要素もしくはデータを有するトレースデータログファイルを生成する。ソフトウェアデバッガの技術分野における上記の改善を達成するために、本明細書に開示されるシステムおよび方法は、様々な機能、技術および方法を採用する。

【0266】

新たな高度技術は、動作するのにこれまで以上に複雑なソフトウェアプログラムを必要とし、本明細書に開示されるようなソフトウェアデバッグシステムおよび方法の実施形態の必要性がますます重要になる。例えば、スマートフォンはますます電話の様々なシステムコンポーネントを実行するために複雑なソフトウェアを必要とする。一般的には、典型的なスマートフォンは、いくつかの場合にはプログラマの大規模なチームによって開発される数百万行のコンピュータコードを必要とする。コード行数の増加および関与プログラムの増加に伴い、ソフトウェアコードのバグ数も増加する。多くの場合、これらのソフトウェアバグを解決することがますます複雑になっている。例えば、いくつかのソフトウェアバグは、断続的および／または低頻度で発生し、つまり、日に一度、週に一度、月に一度、年に一度、またはそれ未満の低頻度で発生する。

【0267】

従来のソフトウェアデバッグプログラムは多くの場合、バグが断続的および／または低頻度でしか発生しないソフトウェアをデバッグしようとするとき、面倒で非効率的である可能性がある。対照的に、「再現性」であるソフトウェアバグは一般的に、コンピュータプログラマが特定および／またはフィックスするのが簡単である。本明細書で使用する場合、「再現性バグ」との用語は一般的に、プログラムが同一のユーザ制御可能入力を用いて実行されるたびに同じようにそれ自体が現れるバグを指す。一般的には、再現性バグは多くの場合、プログラムの実行中に予測可能なように起こるのに対し、低頻度で発生するソフトウェアバグは一般的に、プログラムの実行中にこのように予測可能なようには発生しない。

【0268】

実施形態では、本明細書に開示されたシステムは、ターゲットコンピュータプログラムがハードウェアトレースポート、検出器またはプローブを使用せずに実行されているときに、トレースデータを生成し、効率的に生成中のすべてのそのようなトレースデータを取得するように構成されている。実施形態では、システムは、実行中にターゲットプログラムに関係する特定のデータ要素のみをストレージに保存することにより、トレースデータをより効率的に生成するように構成されている。不要データ要素を収集するおよび／またはコンピュータプログラムに関する貴重なデータ要素を省略する従来のデバッガプログラムとは対照的に、システムは、ターゲットコンピュータプログラムの状態を再構成するために必要なトレースデータをより効率的に生成するように構成されることができる。

【0269】

実施形態では、例えばマルチコアシステム内の複数のコアからログデータをマージする間、トレースデータの順序があいまいであると改善バックエンドが判断した場合、ターゲットプログラムにおける可能なバグを示すことができる。例えば、順序における曖昧さは、1つ以上のコンピュータプログラム命令などのタスクを実行するターゲットプログラムに起因する可能性があり、その完了の順序は予測不能であり、その出力は完了の順序に依存する。これは時にハザードと称され、不正確な実行に潜在的につながる可能性がある。一般的に3つの種類のハザードが存在する可能性があり、異なる命令によるデータの修正に起因するデータハザード、分岐ターゲットでの曖昧さに起因する制御ハザード、および、異なる命令によって同時にアクセスされるメモリに起因する構造ハザードである。ハザードは、シングルコアおよびマルチコアシステムで起こる可能性がある。マルチコアシステムでは、同時に命令を実行する2つ以上の異なるコアに起因してハザードが発生する可能性がある。

【0270】

オペレーティングシステムは典型的には、実体の内外から来ている複数の実行のスレッドを管理している。従来のソフトウェアデバッグシステムでは、実体の内外から来ている実行のスレッドに関するデータは、一般的に収集されない。対照的に、本明細書に開示されるシステムおよび方法の実施形態は、スレッドの作成および破壊に関連するすべてのトレースデータを収集するように構成されている特殊なオペレーティングシステムと連携して動作するように構成されることができる。複数の実行のスレッドを実行可能なオペレーティングシステムは、マルチタスクオペレーティングシステムと称されることができる。本明細書で開示されるシステムおよび方法の実施形態は、マルチタスクオペレーティングシステムによって実行される複数のタスクからタスクのいずれかをデバッグするために使用することができる。デバッグは、実行中の現在の実行のスレッドに限定されるものではなく、オペレーティングシステムによって実行された実行のスレッドの1つ以上をデバッグすることができる。実施形態では、ターゲットシステムが並列に（すなわち同時にまたはほぼ同時に）複数の実行のスレッドを実行するマルチプロセッサまたはマルチコアシステムである場合は、複数の現在実行中または以前に実行されたタスクをデバッグすることができる。理解を容易にするために本開示は、実行のスレッド（または複数の実行のスレッド）としてオペレーティングシステムによって現在、実行されているコンピュータプログラムのデバッグを参照し得るが、開示されたシステムおよび方法の実施形態を、マルチタスクオペレーティングシステムによって実行されている任意の1つまたは複数の実行のスレッドをデバッグするために使用できることを当業者は理解するであろう。

【0271】

実施形態では、システムは、トレースログデータを取得してそれをデコードするように構成されることができ、それにより、システムは、ターゲットプログラムの実行を停止する前の任意の時点でのターゲットコンピュータシステムの状態をユーザに伝えるように構成されることができる。ターゲットコンピュータシステムの状態を再構成すると、プログラムがその時点でターゲットプログラムが何を行っていたのかを決定することを容易にすることができ、実行中に遭遇した1つ以上のエラーの特定および修復が可能になる。

【0272】

実施形態では、プロセッサおよび／またはコアがインストルメント化コードを実行すると、別様に実行されるよりも多くのコードが実行され、その結果、ターゲットプログラムの動作／関数における不可避の遅延および／または減速がある。デバッグプロセスが完了した後、システムは、ターゲットプログラムにインストルメント化コードを挿入する必要なく、ターゲットプログラムをコンパイルするように構成することができ（またはインストルメント化コードは、リンク時に無効化または除去することができ）、それにより、ターゲットプログラムがピークパフォーマンスで動作することが可能となる。本明細書で説明されるように、1つ以上のインストルメント化命令をリンク時にストリップिंगすることができ、これにより、ターゲットコンピュータプログラムが、フルインストルメント化が保持される場合よりも、より効率的に（例えばより高速に）実行することができる。対

照的に、従来のデバッグシステムは一般的に、実行モードにあるときにターゲットプログラムの動作速度を制御することはできない（そして、トレースデータを待つためにコンピュータ処理ユニット（「CPU」）を停止することができるこのようなシステムでは、このようなシステムは、常に適切に動作するわけではなく、すなわち予測不可能である可能性がある）。したがって、従来のデバッグシステムは、貴重なトレースデータを失ってトレースデータログファイルにギャップを作成する可能性があり、このようなギャップはターゲットプログラムの停止条件にまで至るイベントの再生および／または再構成を防止または困難にし、これは、クラッシュ、ブレークポイント、特定のメモリアドレスへのアクセス、実行停止へのユーザ要求などの1つ以上を含む可能性がある。オペレーティングシステムがターゲットプログラムの速度を制御することができない場合、コンピュータのプロセッサは時に、トレースデータをログファイルに保存するには忙しくなりすぎ、トレースログファイルにギャップが生じることがある。

【0273】

ターゲットプログラムのイベントを再生および／または再構成するために20分以上、プログラマが待つ必要がある場合は、ターゲットプログラムをデバッグしようとすることは、プログラマにとって非常に破壊的である可能性がある。プログラマがより多くのデータを収集した場合、従来のソフトウェアデバッグプログラムでは、収集したトレースデータの量に比例したデコード時間となるので、待機時間はさらに長くなり得る。例えば、ターゲットプログラムにおけるクラッシュにまで至るイベントを単に再生および／または再構成するためにプログラマが定期的に20分超待たなければならないとすると、プログラマは、ターゲットプログラムのデバッグ中に自身の思考列を失う可能性がある。したがって、数秒または数分以内にターゲットプログラムのイベントを再生および／または再構成するように構成することができるシステムを利用することは、プログラマにとって有利である可能性がある。実施形態においてシステムは、最後から始まる、つまりターゲットプログラムのクラッシュの直前に発生したイベントに関連付けられたトレースデータから開始するトレースデータを解析することによってターゲットプログラムにおけるクラッシュにまで至るイベントを再生および／または効率的に再構成するように構成される。

【0274】

実施形態では、コンパイラは、ターゲットプログラムのソースコードにて識別された、異なるコンポーネント、関数、イベント等に基づいてインストルメント化を生成するように構成されている。実施形態では、コンパイラは、ターゲットプログラムにおける基本ブロックの識別に基づいてインストルメント化を挿入するように構成されている。いくつかの場合において、基本ブロックは、1つのエン트리ポイントと1つのイグジットポイントとを有する命令のセットまたはシーケンスである。すなわち、基本ブロックは典型的には、ブロックへのエントリを除いて入来ブランチを有し得ず、ブロックからのイグジットを除いて退出ブランチを有し得ない。実施形態では、命令のシーケンスは、次の2つの条件が満たされることを条件とする基本ブロックを形成する：（i）各位置における命令はすべての後続の命令の前に常に実行される、（ii）他の命令は命令のシーケンスにおける2つの命令の間には実行されない。実施形態では、例えば同時に多くの命令を実行するように構成することができるスーパースカラCPUを使用するシステムにおいて、各位置での命令が1つ以上の後続の命令と同時に実行される場合および／または1つ以上の命令が命令のシーケンスで2つの命令間で実行されている場合、命令のシーケンスは基本ブロックを形成することができる。実施形態では、命令の出力および／または結果は、このような命令が、スーパースカラCPUで構成された特定のシステム環境においてなど、同時に処理され得るとしても、基本ブロック内の命令の順序またはシーケンスに保存または適用または結合される。

【0275】

《タイムスタンプ補間》

実施形態では、タイムスタンプは、基本ブロックへのエントリ時または基本ブロックからのイグジット時に挿入されない。代わりに、タイムスタンプは、とりわけ、FEEイン

10

20

30

40

50

ストルメント化（本明細書で説明された）およびオペレーティングシステムとの通信などの他の動作の間に挿入される。補間は、1つ以上の命令または複数の命令の実行時間を決定するために用いられることができる。例えば、システムは、10個の命令などの命令のセットの平均実行時間を推定または測定することができ、この平均実行時間は、特定のタイムスタンプが挿入されなかった特定の複数の命令の実行時間を決定するために使用されることができる。具体的には、命令の平均実行時間は、2つのロギングされたタイムスタンプを発見し、これら2つのタイムスタンプがロギングされたポイント間に実行された命令の数をカウントし、タイムスタンプ間の経過時間を実行された命令の総数で除算することによって計算されることができる。

【0276】

《関数エントリおよびイグジットのロギング》

実施形態では、コンパイラは関数エントリおよび関数のイグジットに基づいてインストルメント化を挿入するように構成されている。以上はFEEインストルメント化と称され得、「関数エントリおよびイグジット」インストルメント化を意味する。本明細書で使用する場合、関数との用語は一般的に、特定のタスクを実行するソフトウェアプログラムの名前付きセクションまたは命令のグループを指す。FEEインストルメント化は、関数を実行するために使用される時間量の正確な推定値を提供することができる。この情報は、デバッグのためだけでなく、特定の関数の実行に費やされる時間量および／またはリソースを減らすなどの最適化のためにも有用である可能性がある。例えば、FEEインストルメント化に基づく情報は、視覚的にコールスタックを提示するために使用されることができる。実施形態では、インストルメント化は、関数へのエントリおよび関数からのイグジットの時にタイムスタンプを挿入することができる。実施形態では、補間は、その実行が例えばタイムスタンプの対の間に発生するエントリおよびイグジットポイント間にある関数の1つ以上の命令などの1つ以上の命令の実行時間を決定するために使用することができる。補間は、本明細書に記載されているように、線形、加重、非線形などである可能性がある。実施形態では、1つ以上のタイムスタンプを、関数へのエントリ後であるが関数からのイグジット前のトレースデータに追加的または代替的に挿入することができる。FEEインストルメント化を、基本ブロックのインストルメント化の代わりに、またはそれと組み合わせて使用することができる。実施形態において、コンパイラは、その実行が別の関数の実行から決定されることができる関数のインストルメント化をスキップするように構成されることができる。

【0277】

特定の場合において、FEEインストルメント化を単独で使用する場合、それは不正確な結果を生成する可能性がある。関数エントリにてロギングされたタイムスタンプは、タイムスタンプを記録するのに使用されるインストルメント化が実際の関数エントリの後にいくつかの命令が発生する可能性があるという点で、少し遅くなる可能性がある。同様に、関数イグジットにてロギングされたタイムスタンプは、タイムスタンプの記録と実際の関数イグジットとの間にいくつかの命令が存在する可能性があるという点で、少し早くなる可能性がある。このような不正確さは、修正されない場合、ソフトウェアプログラムのパフォーマンスの解析の誤解を含む多くの問題につながる可能性がある。このような誤解を招くデータは、プログラマがソフトウェアプログラムのパフォーマンスを最大化するのをはるかに困難にする可能性がある。

【0278】

実施形態では、（本明細書に記載の）タイムスタンプ補間が、これらの不正確さを補正するためにFEEインストルメント化と組み合わせて使用される。タイムスタンプ補間は、より正確に関数の最初と最後の命令を表すタイムスタンプを決定するために用いられることができる。例えば、関数エントリにおけるタイムスタンプが実際の関数エントリから11個の命令の後に算出される場合、タイムスタンプは、11個の命令を実行するのにかかる平均時間を減算することにより調整することができ、より正確な関数エントリのタイムスタンプが得られる。同様の調整を、関数イグジットタイムスタンプが決定された時点

10

20

30

40

50

と関数の実際の終了との間で実行される命令の数を1つの命令を実行するための平均時間に乘じた数をそれらに加算することにより、関数イグジットタイムスタンプに対して行うことができる。これらの補間されたタイムスタンプは、プログラマに関数エントリおよびイグジットに関する情報を提示する場合、または他の方法でF E Eデータを使用する場合、F E Eインストルメント化によってロギングされたタイムスタンプの代わりに使用されることができる。F E Eデータの不正確さを補正するためにタイムスタンプ補間を使用すると、プログラムパフォーマンスのはるかにより正確な解析を可能にすることができる。

【0279】

《システムアーキテクチャ》

図1Aは、デバッガおよびコンパイラを含む一般的なハードウェアおよびソフトウェアアーキテクチャのブロック図を示す。典型的なデバッガシステムは、実行可能コンピュータプログラムを生成するためのコンパイラおよびリンカプログラム108を有するホストシステム118を含む。デバッガプログラムは、トレースデータを収集してコンピュータプログラムの実行中またはその後にログファイルにそのようなデータを保存するように構成されているログダウンローダ112を含む。一般的に、コンパイラプログラムおよびリンカ108は、プログラマにより書かれたソースコード（図示せず）からターゲットプログラム114を生成する。コンパイラプログラムおよびリンカ108はまた、デバッガプログラム110の動作に役立つ情報を含むデバッガプログラム110によって使用されるデータファイル（図示せず）を生成することができる。一般に、デバッガプログラム110は、ユーザがターゲットプログラム114をデバッグできるように、ターゲットシステム120の要素と相互作用することができる。多くの場合、従来のデバッガシステムは、インタフェースマネージャ116を含み、これは、ユーザがデバッガプログラム110と相互作用するためのグラフィカルユーザインタフェースを提供するように構成されることができる。コンパイラおよびリンカプログラム108、デバッグプログラム110、ログダウンローダ112およびインタフェースマネージャ116は、ホストシステム118によって実行され、これは、例えばデバッグ接続117を介するターゲットプログラム114のように、コンピュータプログラムが実行されるターゲットシステム120とインタフェースする。ターゲットシステムは、ホストシステム118のハードウェアおよびオペレーティングシステム（図示せず）とは異なることができるオペレーティングシステム104を実行するハードウェア102を含む。オペレーティングシステム104は、コンピュータハードウェアおよびソフトウェアリソースを一般的に管理するシステムソフトウェアである。オペレーティングシステム104は、ターゲットプログラム114などのコンピュータプログラムのための共通のサービスを提供するためにシステムライブラリ106と併せて作動する。オペレーティングシステム104は、デバッガプログラム110にデバッグデータを提供するように構成されたエージェント122を含むことができる。デバッガプログラム110は、エージェント122に命令およびデータを提供することができる。

【0280】

図1Bは、デバッガおよびコンパイラを含む一般的なハードウェアおよびソフトウェアアーキテクチャの別のブロック図を示す。図1Aとは異なり、ターゲットコンピュータプログラムは、ホストシステム118上で実行される。例えば、図1Bは、Windowsプログラムがコンパイルおよびリンクされてコンピュータシステム上で実行され、Windowsを実行している場合を示している。

【0281】

図2は、本明細書に開示されたデバッガおよびコンパイラシステムの実施形態を含むハードウェアおよびソフトウェアアーキテクチャのブロック図を示す。実施形態では、本明細書に開示されたシステムは、オペレーティングシステム204によって動作および管理されるハードウェア層202を含む。多くの従来のデバッガシステムとは異なり、本明細書に開示されたシステムは、（本明細書に記載の）単一化ログなどのトレースデータロガー207を含むことができる。実施形態では、トレースデータロガー207は、ターゲットプログラム214の実行から生成されたトレースデータを収集するように構成される。

実施形態では、オペレーティングシステム 204 は、デバッガ 212 にトレースデータを提供するためにトレースデータロガー 207 にアクセスするように構成されているエージェント 222 を含むことができる。実施形態では、エージェント 222 は、ターゲットプログラム 214 と通信するように構成されることができる。例えば、システムは、さもなければターゲットプログラム 214 によって直接にアクセスできないトレースデータを検索するため、および他のデバッグタスクを実行するためにエージェント 222 を利用することができる。実施形態では、（例えばカーネルなどの）オペレーティングシステム 204 は、さもなければターゲットプログラム 214 によって直接にアクセスできないトレースデータを記録するように修正され、それは、ターゲットプログラム、レジスタデータ等の動作環境の外部のメモリへのアクセスなどのオペレーティングシステムイベントを含み、さらに、バッファ管理を行い、デバッグに関連する追加のタスクを実行するように修正される。実施形態では、デバッガ 210 は、トレースデータロガー 207 により記憶されたトレースデータにアクセスするためにログダウンローダ 212 を介してエージェント 222 と通信するように構成されることができる。

【0282】

実施形態では、デバッガ 210 は、ターゲットプログラムのクラッシュまたは停止にまで至るイベントの再生および／または再構成を提示するためにトレースデータをデコードして処理するように構成されている。実施形態では、インタフェースマネージャ 216 は、（本明細書に記載の）時間走行デバッガならびに他の構成要素を含むことができ、トレースデータに基づいてイベントの再生および／または再構成を表示するためのグラフィカルユーザインタフェースを提供するように構成されることができる。実施形態では、コンパイラおよびリンカプログラム 208 は、オブジェクトコードにコンパイルされて実行可能コードにリンクされるべきソースコードを受信するように構成されることができる。リンクは、実行可能コードへと必要なライブラリ 206 を伴うコンパイルされたオブジェクトコードをリンクするように構成されることができる。

【0283】

実施形態では、コンパイラおよびリンカプログラム 208 は、インストルメント化ジェネレータ 211 を含む。実施形態では、コンパイラおよびリンカプログラム 208 は、ターゲットプログラムを解析し、および／またはターゲットプログラムのコード内の位置を識別するように構成され、そこには、インストルメント化コードがターゲットプログラムの実行中に有用なトレースデータを生成するために挿入される。実施形態では、インストルメント化ジェネレータ 211 は、コンパイラおよびリンカプログラム 218 の解析に基づいて、ターゲットプログラム内の識別された位置（複数可）にコードを生成および／または挿入するように構成される。実施形態において、インストルメント化コードは、ターゲットプログラムのコードに挿入され、それにより、コンパイラおよびリンカプログラム 218 が、ターゲットプログラムのコードならびにターゲットプログラムのコードに埋め込まれているインストルメント化コードを、コンピュータが実行するためのバイナリターゲットプログラム 214 にコンパイルできる。実施形態では、インストルメント化コードはまた、ターゲットライブラリ 206 のコードに挿入される。

【0284】

実施形態では、トレースロガー 207 およびインストルメント化ジェネレータ 211 によって生成されたインストルメント化コードは、1 つ以上のトレースデータログ 209 に記憶するためのトレースデータを生成するように構成される。例えば、トレースデータログ 209 は、（オペレーティングシステム 204 によって実行中の 1 つ以上の実行のスレッドの）インストルメント化コードの実行に関連するトレースデータを保存するための第 1 のログ、および、オペレーティングシステムによってそれぞれ実行されるタスクに関連付けられているトレースデータを保存するための第 2 のログを含むことができる。別の例として、単一の組み合わせられたトレースデータログ 209 は、インストルメント化コードによって、およびオペレーティングシステムによって生成されたトレースデータのために使用されることができる。

【0285】

実施形態では、システムは、システムの一部であるデバッガ210を有することなく、ターゲットプログラムをデバッグするためのトレースデータを収集するように構成されることができる。例えば、システムは、ターゲットプログラムにインストール化を挿入するように構成されたインストール化ジェネレータ211を有するコンパイラおよびリンカ208などのコンパイラを含むことができる。前記インストール化は、ターゲットプログラムが動作している遠隔位置またはフィールドから検索されることができるトレースデータを生成および／または収集するように構成されている。検索されたトレースデータは、ターゲットプログラムのバグを解決するためのデバッガを有している別のシステムで解析されることができる。別の例として、システムは、インストール化ジェネレータ211を有するコンパイラおよびリンカ208を含まなくともよく、トレースデータを生成および／または収集するように構成された、以前にインストール化（およびコンパイル）されたターゲットプログラムを実行し得、これは、デバッガを有する別のシステム上で検索および解析されることができる。図1Bに示されているように、図2のシステムは、コンパイラおよびリンカ208とインタフェースマネージャ216とがターゲットプログラム214と同じコンピュータシステム上で実行されるように修正されることができる。

10

【0286】

図3は、本明細書に開示されたデバッガおよびコンパイラシステムの実施形態（または改善バックエンドの実施形態）の例示的な利点を示す棒グラフを示す。グラフ302を参照して、従来のデバッグシステムに対して本明細書に開示されたシステムの実施形態によって達成される技術的改良の例が示されている。実施形態では、システムは、システムのユーザが1秒以内にデバッグを開始することができるように、ターゲットプログラムのクラッシュまたは他の停止にまで至るイベントを再生および／または再構成するように構成されることができる。従来のデバッグシステムを使用すると、同じターゲットプログラムのクラッシュまたは他の停止にまで至るイベントの再生および／または再構成は、ログのサイズに比例した時間がかかるだろう。この例では、従来のデバッグシステムは、トレースデータファイルの先頭から開始するおよそ1ギガバイトのトレースデータを解析するのに約20分かかるであろう。本明細書に開示されるシステムおよび方法の実施形態を利用することによって、1ギガバイトのトレースデータファイルを処理する、または、ユーザにそれへのアクセスを提供するのに約1秒しかかからず、これは、本明細書のシステムが、ファイルの最後から出発してトレースデータファイルを解析し、数秒以内に使用可能情報をユーザに提供する（それにより、いくつかの従来のデバッグシステムが、必要とした1ギガバイトのデータファイル全体を処理するのに必要な時間を大幅に回避する）よう構成されることができるからである。

20

30

【0287】

グラフ304を参照すると、ターゲットプログラムの実行中にイメージ化されるターゲットコンピュータシステムのメモリの一部の例が示されている。実施形態では、本明細書に開示されるシステムは、最大でメモリの100%を含むメモリの任意のサブセットをイメージ化するよう構成されることができる。比較して、いくつかの従来のデバッグシステムは、同一のターゲットプログラムの実行中に利用されるメモリの25%以下をイメージ化し得るのみである。

40

【0288】

グラフ306を参照すると、従来のデバッグシステムとは対照的に、本明細書に開示されたシステムの実施形態を用いてプログラマが達成することができる時間節約の例が示されている。例えばプログラマは、本明細書に開示されたシステムの実施形態を利用することにより、わずか5分以下でプログラムのバグを解決することができる。対照的に、従来のデバッガを使用する同じプログラマは、同じターゲットプログラム内の同じバグを解決しようとする、少なくとも20分、おそらくはそれ以上を費やす必要がある。一般的に、プログラマは、トレースデータログファイルを処理するためにデバッグプログラムを待

50

つのに費やす時間が少ないので、本明細書に開示されたシステムの実施形態を利用してバグを解決する上でより効率的になることができる。トレースデータログファイルを数秒以内に処理することができる場合、プログラマはバグの解決により容易に集中することができる。対照的に、プログラマがトレースデータログファイルを処理するために20分以上待たなければならない場合、プログラマはプログラムのバグの解決への集中力を失い始める可能性がある。

【0289】

《コンパイルおよびデバッグ》

図4は、コンピュータプログラムをコンパイルしてデバッグするための高レベルプロセスの実施形態を示すフローチャートである。本明細書に開示されたシステムの実施形態は、ターゲットプログラムの実行を再構成および／またはシミュレートするためのコンパイラおよびデバッグシステムを含むことができる。ターゲットプログラムの実行を再構成および／またはシミュレートすることで、プログラマは、迅速に、効率的に、かつ正確にコンピュータプログラムをデバッグするために、このような情報を利用することができる。実施形態では、高レベルプロセスは、ブロック402で開始することができ、ブロック404において、システムは、コンピュータプログラムのソースコードファイルにアクセスするように構成されることができる。ブロック406において、システムは、システムのコンパイラを使用してターゲットプログラムに関連付けられたソースコードをコンパイルするように構成されることができる。コンパイラはさらに、トレースデータを生成するように構成されたインストルメント化命令を挿入することによって、ソースコードをインストルメント化するように構成されることができる。ブロック408において、システムは、実行可能な機械可読プログラムを出力するためにリンカを利用するように構成されることができる。ブロック410において、システムは、トレースデータを収集し、ログファイルにそのようなデータを保存するように構成されたオペレーティングシステム上で機械可読プログラムを実行するように構成されることができる。ブロック412において、システムは、保存されたトレースデータに基づいてプログラムの実行を再構成および／またはシミュレートするために、デバッガシステムを利用するように構成されることができる。ブロック414において、プログラマは、再構成および／またはシミュレーションを解析するために、デバッガシステムのグラフィカルユーザインタフェースを利用することができる。グラフィカルユーザインタフェースを動的に変更または更新することができる。この解析を通して、プログラマは、プログラムのバグを特定し、任意のそのようなエラーを解決することができる。ブロック416では、プログラマはデバッガシステムによって生成された再構成および／またはシミュレーションの自身の解析に基づいて、ターゲットプログラムのソースコードをリバイズすることができる。任意選択的に、このプロセスは、ブロック404に戻って再度、処理を開始することができる。代替的に、プロセスは、ブロック418で終了することができる。実施形態では、プロセスを、オペレーティングシステムによって実行中の（すべての実行のスレッドについてを含む）複数の実行のスレッドについて実行することができる。

【0290】

図5は、オペレーティングシステムおよびターゲットプログラムと相互作用するコンパイラシステムおよびデバッグシステムの実施形態の高レベルの概要を示すブロック図である。実施形態では、本明細書に開示されたシステムは、コンパイラシステム（これはまたリンカシステムを含むことができる）、オペレーティングシステム、および／または、デバッグシミュレーションシステムを含むことができる。実施形態では、ブロック502で、コンピュータプログラムは、コンパイラシステムを使用してコンパイルされ、リンクされる。コンピュータプログラムはまた、トレースデータを生成するためにインストルメント化される。実施形態では、ブロック504において、コンパイルされたターゲットプログラムは、アプリケーション層で、オペレーティングシステム、または両方で実行される。ターゲットプログラムは、実行時にログファイルにトレースデータを出力するように構成されることができる。トレースデータの収集を、オペレーティングシステムによって部

分的に完了することができる。実施形態では、オペレーティングシステムは、ターゲットプログラムがアクセス権を持たないターゲットプログラムの代わりに、オペレーティングシステムによって実行されるそれらのアクションをトレースデータとして記録するよう修正される（例えばそのカーネルが修正される）。そのようなアクションとして（i）オペレーティングシステムサービスの呼び出し（例えばAPI）、（ii）1つ以上のプリライト値の記録を含む、（例えば、オペレーティングシステムサービスまたはAPIに応答した）ターゲットプログラムのメモリ内のオペレーティングシステムによって実行される変更、（iii）ターゲットコンピュータプログラムに代わって実行されるオペレーティングシステムサービス、（iv）タスク切り替え、などが挙げられる。トレースデータは、コンパイルされたターゲットプログラムおよびオペレーティングシステムの両方の実行によって収集されることができる。1つの実施形態では、オペレーティングシステムおよびターゲットプログラムによって収集されたトレースデータは、単一トレースデータログファイルに保存される。別の実施形態では、オペレーティングシステムおよびターゲットプログラムによって収集されたトレースデータは、デコードプロセス中に合成されてもされなくともよい2つ以上の別個のトレースデータログファイルに保存される。デバッグ／シミュレーションシステムは、ターゲットプログラムの実行から生成されたデバッグデータをブロック506で処理およびデコードするように構成されることができる。デコードされたデバッグデータは、1つ以上のバグの識別および解決を容易にするためにグラフィカルユーザインタフェースでユーザに提示されることができる。グラフィカルユーザインタフェースを動的に変更または更新することができる。

【0291】

図6は、ターゲットコンピュータプログラムの実行中に発生するイベントを再生および／またはシミュレートするように構成されたデバッグシステムの実施形態の高レベルの概要を示すブロック図である。タイムライン602は、ターゲットプログラムの実行を示す。図示のように、タイムライン602は、プログラムが実行されると生じるメモリアクセス、関数呼び出しなどの多数のイベントを含む。最終的に、ターゲットプログラムは、クラッシュ、ブレークポイント、特定のメモリアドレスへのアクセス、実行を停止するユーザ要求等のうちの1つ以上を含むことができる、停止条件604に達する。停止条件604にまで至る一連のイベント608、610、612を含むことができる関心対象の領域606がある。関心対象の領域606は、ターゲットプログラムをデバッグするために非常に重要であり、それは、多くの場合、即座に停止条件にまで至るイベントがプログラマに、ソフトウェアのバグの性質に最も重要な洞察力を提供するからである。バグの性質を理解することで、プログラマはより容易にプログラムのエラーをフィックスすることができる。実施形態では、システムは、時間走行デバッグおよびグラフィカル表示ツールなどの1つ以上のユーザインタフェースを使用してプログラマに一連のイベント614を提示するように構成されることができる。そのようなユーザインタフェースを動的に変更または更新することができる。イベントは、まずプログラマに最も関連性の高い情報を利用可能にするように、停止条件618から始まる逆時間順616でデコードされる。例えば、システムが停止条件604からイベント612にステップバックするのにかかる時間を約1秒とすることができ、一定のアルゴリズム的複雑さでシステムによって必要な計算が実行される。トレースデータの収集ならびに再生および／またはシミュレーションのさらなる詳細は、特許第7,653,899号；第8,015,552号；第8,132,159号；第8,136,096号；第8,271,955号；第8,584,097号；第8,789,023号；および第8,914,777号および特許公開第2014/0298301号において記載され、これらの各々の開示はその全体が参考として援用される。

【0292】

実施形態では、ブロック622において、システムは、システムによって実行されるターゲットコンピュータシステムの解析および／または状態のビジュアル化を表示するように構成されることができ、よってプログラマは、ソフトウェアプログラムでのエラーを識別することができる。

【0293】

図7は、コンピュータプログラムをコンパイルしてデバッグする実施形態を示すフローチャートである。実施形態では、システムは、コンパイラシステム（これはまたリンカシステムを含む）、オペレーティングシステム、およびデバッグ／シミュレーションシステムを含むことができる。実施形態では、プロセスは、ブロック702で開始することができ、コンパイラシステムは、ブロック704でコンパイルするためのソフトウェアコードを解析する。ブロック706で、コンパイラは、ソフトウェアコードにインストルメント化コードを挿入するように構成されることができる。ブロック708で、コンパイラシステムは、オブジェクトファイル（複数可）を生成するように構成されることができる。ブロック710において、リンカシステムは、オブジェクトファイル（複数可）をリンクするように構成されることができる。任意選択的に、ブロック710において、リンカシステムは、1つ以上のライブラリオブジェクトファイル（複数可）をリンクすることができる。ブロック712において、コンパイラシステムは、実行可能な機械可読プログラムファイルを出力するよう構成されることができる。ブロック714で、オペレーティングシステムは、（タスクとしてオペレーティングシステムによって実行されることができる）コンパイラシステムによって出力されたプログラムファイルの実行を協調させるように構成されることができる。ブロック716で、オペレーティングシステムは、基本ブロックなどのブロックの実行、および／またはプログラム内の関数を協調させるように構成されることができる。ブロック718で、オペレーティングシステムは、様々なメモリ値、プロセッサレジスタなどへのオペレーティングシステムのアクセスを介してなど、デバッグデータ（例えばオペレーティングシステムイベント）を収集するように構成されることができる。ブロック718で収集されたデバッグデータは、レジスタスナップショットおよびメモリアクセス（プリライト値などの）を含むことができる。ブロック720で、オペレーティングシステムは、メモリ、例えばデバッグデータを保存するために指定されたログまたはデータベース722にデバッグデータを保存するように構成されることができる。

【0294】

ブロック724において、システムは、メモリ内にデバッグデータを保存するためのプログラムの命令のインストルメント化命令を実行するように構成されることができる。ブロック724において収集されたデバッグデータは、不揮発性メモリ位置および副作用なしの揮発性メモリ位置についてのプリライトメモリ値、副作用を伴う揮発性メモリ位置から読み出した値、基本ブロックへのエントリを示す値、および／またはFEEデータを含むことができるが、これらに限定されない。ブロック726では、システムは、クラッシュ、ブレークポイント、メモリアクセス、ユーザの介入などを含むことができる停止条件を検出するように構成されることができる。ブロック728において、システムは任意選択的に、メモリ内のデバッグデータ、例えば停止条件時におけるオペレーティングシステム内のメモリ値を記憶するように構成されることができる。このデバッグデータは、停止条件をトリガした1つ以上の条件のクラッシュまたは実行の時のターゲットコンピュータシステムのメモリアイメージ（またはターゲットコンピュータシステムのメモリに保存された値）および／または1つ以上のレジスタスナップショットを含むことができる。実施形態では、データベース730は、ブロック724および728にてデバッグデータを記憶するために利用される。実施形態では、データベース722および730は、単一データベースに統合されることができる。ブロック732で、デバッグ／シミュレーションシステムは、データベース730、722に保存されたデバッグデータにアクセスするように構成される。ブロック734で、デバッグ／シミュレーションシステムは、保存されたデバッグデータに基づいて停止条件における、およびその前のシステムの状態を再構成／再生するよう構成される。

【0295】

図10Aは、コンピュータプログラムの実行中にデバッグ用トレースデータを生成するためのプロセスの実施形態を示すフローチャートである。実施形態では、コンパイルされたソフトウェアプログラムの実行は、コンパイルプロセス中にソフトウェアプログラムに

10

20

30

40

50

挿入されたインストルメント化コードの実行を引き起こすであろう。例えば、インストルメント化コードの実行により、プリライトメモリ値が、ターゲットプログラムの実行中にトレースログファイルに保存されることが可能である。トレースログファイルは、停止条件にまで至るイベントを再生および／または再構成するために利用されることができる。

【0296】

ターゲットコンピュータプログラムを実行するためのプロセスは、ブロック1002で開始することができ、ブロック1004で、プログラムは、コンピュータプログラムの第1の基本ブロックを入力することによって実行されることができる。ブロック1006で、基本ブロック内のインストルメント化が実行され、基本ブロックに関連付けられた一意識別子（または基本ブロックが実行されていることを識別する他の値）が識別される。ブロック1008で一意識別子は、トレースデータログに保存される。実施形態では、一意識別子は、ブロック1008において（予約レジスタなどの）レジスタに保存されたポインタによってポイントされたメモリに保存されることができ、これは、インストルメント化コードの実行をスピードアップし、それによりターゲットコンピュータプログラムの実行への影響を低減し得る。ブロック1010では、基本ブロック内の命令が実行される。ブロック1012で、追加のインストルメント化は、任意選択的に実行される。実施形態では、基本ブロックがメモリへ書き込みしようとしているときのインストルメント化は、任意選択的にブロック1012で実行される。インストルメント化は、基本ブロックがメモリ位置に書き込む前に、メモリ位置からトレースログファイルへのプリライト値の読み出しおよび／または記憶を行うように構成されることができる。任意選択的に、ブロック1014で、ブロック1012のインストルメント化は、メモリ内のプリライト値をトレースログファイルに保存させ、基本ブロックはメモリへの書き込みを行える。ブロック1012および1014がトレースデータログにプリライトメモリ値を保存するよう実行されない場合であっても、基本ブロックは、依然としてメモリへの書き込みを行い得る。ブロック1016において、実行のための次の基本ブロックがあるかどうかをプロセスは判断する。そのような基本ブロックが識別された場合、プロセスはブロック1006に戻ることができる。そのような基本ブロックが検出されない場合、プロセスはブロック1018で終了することができる。

【0297】

留保トレースデータ最適化を利用する場合、図10Aに示すように、ブロック1006および1008が実行されない。むしろ、ブロック1006および1008が、ブロック1014の後などの、基本ブロックの最後で実行される。

【0298】

図10Bは、コンピュータプログラムの実行中にトレースデータを生成するためのプロセスの実施形態を示すフローチャートである。ブロック1002～1014、1016および1018の動作は、図10Aに関連して記載されている。図10A中のプロセスと同様に、実施形態では、本明細書に開示されたシステムは、不揮発性メモリ位置および副作用なし揮発性メモリ位置に記憶されているプリライト値をトレースデータログファイルへと保存するだけではないように構成されることができる。システムはまた、基本ブロックの命令でアクセスされる副作用つき揮発性メモリのメモリ位置のアドレスを記録するように構成されることができる。ブロック1015では、プロセスは任意選択的に、副作用つき揮発性メモリ位置にプログラムによるストアのアドレスを記録する。ここで説明したように、副作用を有する揮発性メモリの位置に保存された値は、他のプログラムにより変更することができ、および／または、このような揮発性メモリ位置を読み取ることにより、保存された値を変更することができる。したがって、プリライト値を記録するためのインストルメント化は、このような揮発性メモリ位置について使用されなくともよい。ブロック1015が実行されない場合であっても、基本ブロックは、依然としてそのような揮発性メモリ位置へのアクセスが可能であり得る。

【0299】

《関数エントリおよびイグジット（FEE）インストルメント化》

図 1 1 は、関数からなるソフトウェアプログラムの実施形態の高レベルの概要を示すブロック図である。実施形態では、ソフトウェアプログラム 1 1 0 0 は、複数の関数から構成されることができる。本明細書で説明されるように、関数との用語は、一般的にソフトウェアプログラムの名前付きセクションまたは特定のタスクを実行する命令のグループを指す。実施形態では、ソフトウェアプログラム 1 1 0 0 は、第 1 の関数（関数 1）1 1 0 2 を含むことができ、これはプログラム（例えば主関数）へのエントリポイントとすることができる。実施形態では関数 1 は、1 1 0 4 で関数 2 に、1 1 0 6 で関数 3 に分岐することができる。関数 2 は、他の関数 1 1 0 8 および 1 1 1 0 につながる多数の他の分岐を含むことができる。

【0300】

実施形態では、1 つ以上のリーフ関数は、F E E インストルメント化でインストルメント化されない。リーフ関数は、別の関数によって呼び出され得るが、それ自体は任意の他の関数を呼び出さない関数として定義することができる。後続する基本ブロックでのみ実行される基本ブロックに関連して説明した最適化と同様に、F E E インストルメント化は、リーフ関数では省略され得る。代わりに、実施形態では、（本明細書に記載されるような）静的ルックアサイドテーブルが、別個にインストルメント化されなかったリーフ関数の実行を示すことができる。本明細書に記載されるように、いくつかの実施形態では、静的ルックアサイドテーブルは、再生および／または再構成の間にトレースデータファイルにリーフ関数の実行に関連する適切なデータを挿入するために使用されることができる。実施形態では、同様のアプローチを、小さな関数、例えば X 個（X は調整可能パラメータ）未満の命令に適用することができる。

【0301】

《最適化》

本明細書で説明されるシステムおよび方法の実施形態は、とりわけ、本明細書に記載される以下の最適化の 1 つ以上を利用することができる：後続する基本ブロックでのみ常に行われるいくつかのまたはすべての基本ブロックへのエントリを別個に記録しないこと、図 8 C および図 8 D に関連して説明された最適化を行うこと、基本ブロックの最後に関連付けられたプログラムカウンタ値を記録すること、留保トレースデータ最適化、リーフ関数のインストルメント化をスキップすること、スタックエントリおよびイグジットマーカーを挿入しないこと。これらの最適化の 1 つ以上を使用することは、とりわけ、トレースデータの大きさの減少、ならびに、再生、再構成および／またはシミュレーションについての時間の短縮のうちの 1 つ以上に有利につながるることができる。

【0302】

《ユーザインタフェース》

図 1 3 A、図 1 3 B は、時間走行デバッガシステムのグラフィカルユーザインタフェースを示す。図示のグラフィカルユーザインタフェースは、再生および／または再構成中に提示されることができる。実施形態では、ユーザは、グラフィカルユーザインタフェース 1 3 0 0 と相互作用し、ターゲットコンピュータプログラムの実行をステップスルーしてターゲットコンピュータプログラムのソースコード内の対応する行と実行とを比較することができる、これは、プログラマがターゲットプログラムをデバッグするのを助けることができるプロセスである。本明細書に記載されるように、ユーザアクティビティまたは選択に応答して、グラフィカルユーザインタフェースを動的に変更または更新することができる。実施形態では、グラフィカルユーザインタフェース 1 3 0 0 は、ユーザにソースコードを表示するための表示領域 1 3 2 2 を含む。実施形態において、表示領域 1 3 2 2 はまた、ターゲットプログラムの実行の再生または再構成を提示するための各種のデータを含み、例えば、実行コマンド、入力／出力、変数値／変更、レジスタ値／変更、メモリ値／変更、スタックトレース、ブレークポイント、ソースコード、ブレークポイントが配置され得る箇所を示すブレイクドット等である。実施形態では、プログラムの現在の行および／またはデバッグされる時点は、矢印 1 3 3 2 によって示される。実施形態では、コマンドの現在のコンテキストは、矢印 1 3 3 0 によって示される。

【0303】

実施形態では、グラフィカルユーザインタフェース1300は、ボタン1302～1316を含み、これらによりユーザはターゲットコンピュータプログラムを時間的に前方および後方に実行または実施することができ、共通のデバッグコントロールおよびそれらの時間的後方変種を使用してトレースログの時間的に前方および後方への移動が可能となる。それらは、ソース行およびアセンブリ命令の両方のデバッグで同様に動作する。実施形態では、ボタン1302により、ユーザが後方に（現在の関数などの）ターゲットコンピュータプログラムを実行し、または、その実行をシミュレートすることができる。実施形態では、ボタン1304により、ユーザが、現在の関数を呼び出した（以前の関数などの）ソースコード内の命令の以前のシーケンスまで戻ることができる。実施形態において、ボタン1306により、ユーザが同じスタックレベルで実行される以前の命令またはソース行に単一ステップ戻ることができる。実施形態では、ボタン1308により、ユーザが、実行された以前のソース行が関数呼び出しであった場合に、実行された以前の命令またはソース行に、および、関数呼び出しの命令またはソースコードに単一ステップ戻ることができる。ボタン1302～1308により、ユーザは、時間的に後方にターゲットコンピュータプログラムの実行を再生および／または再構成することができる。

10

【0304】

実施形態では、ボタン1310により、ユーザが次の命令またはソース行が関数呼び出しである場合、実行またはシミュレートされる次の命令またはソース行におよび関数へと単一ステップ進むことができる。実施形態では、ボタン1312により、ユーザはソースコードにおける次の命令または行（または単一のステップ）を実行またはシミュレートすることができる。実施形態では、ボタン1314により、ユーザが（次の関数へとコールスタックを上げてゆくことをもたすことができる関数などの）命令の現在のシーケンスからステップアウトすることができる。実施形態では、ボタン1316により、ユーザが（現在の関数などの）ターゲットコンピュータプログラムを前方に実行またはシミュレートすることができる。実施形態では、ボタン1318により、ユーザがターゲットコンピュータプログラムの実行またはシミュレーションを停止することができる。ボタン1310～1316により、ユーザはターゲットコンピュータプログラムを時間的に前方に実行またはその実行をシミュレートすることができる。

20

【0305】

実施形態では、本明細書に開示されるシステムは、（1秒以内または数秒以内などの）一定のアルゴリズムの複雑さにてトレースデータデバッグを開始するのに必要な計算を実行するように構成され、これにより、ユーザがプログラム停止のほぼ直後に本明細書のユーザボタンを利用することができ、よってユーザは、従来のデバッグシステムと比較してより迅速にターゲットプログラムをデバッグすることができる。例えば、ボタン1302～1318の1つ以上に関連したプロセスの各々は、ユーザにリアルタイムまたはほぼリアルタイムの経験を提供するために、1秒以内または数秒以内に実行またはシミュレートされることができる。

30

【0306】

実施形態では、グラフィカルユーザインタフェース1300は、1つ以上の追加ボタンおよび／または構成要素を含む。実施形態では、グラフィカルユーザインタフェース1300は、ターゲットコンピュータシステムのメモリの一部または全体のビューを可能にするボタン1332を含む。実施形態では、グラフィカルユーザインタフェース1300は、ターゲットコンピュータシステムのレジスタの一部または全体のビューを可能にするボタン1334を含む。実施形態では、グラフィカルユーザインタフェース1300は、ターゲットコンピュータプログラムのローカル変数の一部または全体のビューを可能にするボタン1336を含む。

40

【0307】

実施形態では、グラフィカルユーザインタフェース1300により、ユーザがその時点でその変数の値を示すソースコードディスプレイ1322内の任意の変数を（例えばダブ

50

ルクリックまたは他の作用を介して) 選択することができる。ユーザはまた、先に本明細書に記載した任意のアクションを行うために領域 1 3 2 0 にテキストコマンドを入力し得る。領域 1 3 2 4 は、実行中のオペレーティングシステムタスクのビューを含む。矢印 1 3 2 6 によって示されるように、デバッグ中のタスクは、強調表示されたタスク(「初期」)である。

【0308】

《コンピューティングシステム》

実施形態では、本明細書で説明されるシステム、プロセスおよび方法が、図 1 4 に示したものなどのコンピューティングシステムを使用して実装される。例示的なコンピュータシステム 1 4 0 2 は、1 つ以上のネットワーク 1 4 1 8 を介して 1 つ以上のコンピューティングシステム 1 4 2 0 および／または 1 つ以上のデータソース 1 4 2 2 と通信している。図 1 4 はコンピューティングシステム 1 4 0 2 の実施形態を示すが、コンピュータシステム 1 4 0 2 の構成要素およびモジュール内で提供される機能は、より少数の構成要素およびモジュールに結合、または、さらに追加的な構成要素およびモジュールに分離され得ることが認識される。

【0309】

《ソフトウェア開発環境》

コンピュータシステム 1 4 0 2 は、本明細書で説明する機能、方法、作用および／またはプロセスを実行するソフトウェア開発環境およびターゲットシステム 1 4 1 4 を含む。ソフトウェア開発環境およびターゲットシステム 1 4 1 4 は、本明細書にさらに論じられる中央処理ユニット 1 4 0 6 によりコンピュータシステム 1 4 0 2 上で実行される。実施形態ではターゲットシステムは、本明細書に示されるように、ソフトウェア開発環境 1 4 1 4 と同じコンピュータシステム 1 4 0 2 に常駐するのとは反対に、別のコンピュータシステム内に常駐し得る。

【0310】

《コンピューティングシステムコンポーネント》

コンピュータシステム 1 4 0 2 は、マイクロプロセッサなどのプロセッサを含み得る 1 つ以上の処理ユニット(CPU) 1 4 0 6 を含む。実施形態において、コンピュータシステムは、対称型マルチプロセッサまたは対称マルチコア(SMP)システムなどの SMP システムである。コンピュータシステム 1 4 0 2 はさらに、情報の一時的保存のためのランダムアクセスメモリ(RAM)、情報の永久記憶のための読み出し専用メモリ(ROM)、および、ハードディスクドライブ、ソリッドステートドライブ(SSD)、フロッピーディスクまたは光学媒体記憶デバイスなどのマスストレージデバイス 1 4 0 4 などのメモリ 1 4 1 0 を含む。あるいは、マスストレージデバイスは、1 つ以上のサーバで実装され得る。典型的には、コンピュータシステム 1 4 0 2 の構成要素は、規格ベースのバスシステムを使用してコンピュータに接続される。バスシステムは、ペリフェラルコンポーネントインターコネクト(PCI)、マイクロチャネル、SCSI、インダストリアルスタンダードアーキテクチャ(ISA)およびエクステンデッド ISA(EISA)アーキテクチャなどの様々なプロトコルを使用して実装することができる。

【0311】

コンピュータシステム 1 4 0 2 は、キーボード、マウス、タッチパッドおよびプリンタなどの 1 つ以上の入力／出力(I/O) デバイスおよびインタフェース 1 4 1 2 を含む。I/O デバイスおよびインタフェース 1 4 1 2 は、ユーザにデータを視覚的に提示することができる、モニタなどの 1 つ以上の表示デバイスを含むことができる。より具体的には、表示デバイスは、例えばアプリケーションソフトウェアデータおよびマルチメディアプレゼンテーションなどのグラフィカルユーザインタフェース(GUI)の提示を提供する。I/O デバイスおよびインタフェース 1 4 1 2 はまた、様々な外部デバイスへの通信インタフェースを提供することができる。コンピュータシステム 1 4 0 2 は、例えばスピーカ、ビデオカード、グラフィックスアクセラレータおよびマイクロフォンなどの 1 つ以上のマルチメディアデバイス 1 4 0 8 を含み得る。

【0312】

《コンピューティングシステムデバイス／オペレーティングシステム》

コンピュータシステム1402は、サーバ、Windowsサーバ、UNIXサーバ、パーソナルコンピュータ、ラップトップコンピュータおよび同様のものなどの様々なコンピューティングデバイス上で実行され得る。コンピューティングシステム1402は一般的に、z/OS、Windows95、Windows98、WindowsNT、Windows2000、WindowsXP、WindowsVista、Windows7、Windows8、Windows10、Linux、OSX、BSD、SunOS、Solaris、INTEGRITY、または独自のオペレーティングシステムを含む他の互換性のあるオペレーティングシステムなどの、オペレーティングシステムソフトウェアによって制御されて協調されている。オペレーティングシステムは、実行についてのコンピュータプロセスの制御およびスケジューリング、メモリ管理の実行、ファイルシステム、ネットワークングおよびI/Oサービスの提供、ならびに／または、とりわけGUIなどのユーザインタフェースの提供を行うことができる。

10

【0313】

《ネットワーク》

図14に示すコンピュータシステム1402は、LAN、WANまたはインターネットなどのネットワーク1418に、通信リンク1416（有線、無線、またはそれらの組み合わせ）を介して結合されることができる。ネットワーク1418は、様々なコンピューティングデバイスおよび／または他の電子デバイスと通信する。ネットワーク1418は、1つ以上のコンピューティングシステム1420および1つ以上のデータソース1422と通信することができる。

20

【0314】

出力モジュールは、陰極線管（CRT）、液晶ディスプレイ（LCD）、プラズマディスプレイ、または、ディスプレイの他のタイプおよび／もしくは組み合わせなどとして全点アドレス可能ディスプレイの組み合わせとして実装され得る。出力モジュールは、入力デバイス1412と通信するように実装され得、それらはまた、ユーザが、メニュー、ウィンドウ、ダイアログボックス、ツールバーおよびコントロール（例えばラジオボタン、チェックボックス、スライドスケール等）などの定型化された画面要素の使用を介してデータにアクセスできる適切なインタフェースを伴うソフトウェアを含む。さらに、出力モジュールは、ユーザから信号を受信するために入力および出力デバイスのセットと通信し得る。

30

【0315】

《追加変形例》

実施形態では、非一時的コンピュータ記憶媒体は、タスクとしてターゲットコンピュータシステムのオペレーティングシステムによって実行されるコンピュータプログラムをデバッグするための命令を含み、命令は、ターゲットコンピュータシステムによって実行されると、実行を停止するための少なくとも1つの条件がトリガされるまで、ターゲットコンピュータシステム上でコンピュータプログラムの複数のプログラム命令を実行することを含む方法を実行し、ターゲットコンピュータシステムはメモリおよび少なくとも1つのプロセッサを含む。方法はまた、ターゲットコンピュータシステム上でコンピュータプログラムの複数の命令を実行しながら、コンピュータプログラムの実行に関連するトレースデータログにトレースデータを記録することを含むことができ、トレースデータは、実行中のプログラム命令のサブセットを識別するデータの第1のセット、コンピュータプログラムによって実行中の複数のメモリ書き込みを識別する第2のデータセット、および、実行を停止するための少なくとも1つの条件がトリガされたときにターゲットコンピュータシステムのメモリに保存されたメモリ値を含む第3のデータセットを含む。この方法はまた、複数のプログラム命令からのプログラム命令実行時のターゲットコンピュータシステムの状態の、少なくとも1つの条件がトリガされたときに実行中のプログラム命令からのコンピュータプログラムの実行とは逆の順序で、再構成用のトレースデータを提供すること

40

50

とを含むことができ、再構成は一定のアルゴリズム的複雑さに応じて行われる。

【0316】

実施形態では、前段落のコンピュータ記憶媒体は、以下の特徴の1つ以上を含むことができる。複数のメモリ書き込みを識別する第2のデータセットは、上書き前にメモリに保存されたデータ値を含むことができる。方法はまた、定期的にコンピュータプログラムの実行中にターゲットコンピュータシステムの少なくとも1つのプロセッサの複数のレジスタに保存されたデータ値の複数のスナップショットを記録すること、および、ターゲットコンピュータシステムの状態の再構成のために複数のスナップショットを提供することを含むことができる。コンピュータプログラムは、トレースデータの少なくとも一部を記録するように構成された複数のインストルメント化命令を含むことができる。複数のインストルメント化命令は、基本ブロックに入ると複数の基本ブロックの各基本ブロックについて実行されるように構成された1つ以上のインストルメント化命令を含むことができ、複数の基本ブロックはコンピュータプログラムに関連付けられ、複数の基本ブロック内の各基本ブロックは、1つのエントリポイントおよび1つのイグジットポイントを有する命令のシーケンスを含む。1つ以上のインストルメント化命令はさらに、基本ブロックの第1の命令を実行する前に実行されるように構成されることができる。1つ以上のインストルメント化命令はさらに、複数の基本ブロックの各基本ブロックの実行に関連付けられた複数の一意識別子をトレースデータログに記録するように構成されることができる。基本ブロックの実行に関連付けられた一意識別子をトレースデータログに記録するように構成された1つ以上のインストルメント化命令は、基本ブロックからのイグジット時に実行されるように構成されることができる。一意識別子は、基本ブロックからのイグジットに関連付けられたプログラムカウンタ値を含むことができる。複数のインストルメント化命令はさらに、コンピュータプログラムの実行中にターゲットコンピュータシステムの少なくとも一部の揮発性メモリ位置の複数のメモリ読み出しを識別するデータを記録しないように構成されることができる。複数のインストルメント化命令はさらに、少なくとも一部の揮発性メモリ位置のアドレスを記録するように構成されることができる。実行を停止するための少なくとも1つの条件は、プログラム命令の実行、メモリアクセス、または別の停止条件の1つ以上を含むことができる。

【0317】

実施形態では、前の段落のいずれかのコンピュータ記憶媒体は、以下の特徴の1つ以上を含むことができる。方法は、トレースデータログにトレースデータを保存するためのメモリバッファへのポインタとしてターゲットコンピュータシステムの少なくとも1つのプロセッサの予約レジスタを設定することを含むことができる。方法はまた、ターゲットコンピュータシステム上でのコンピュータプログラムの実行中に、メモリバッファへのアクセスに関連したページフォルトをターゲットコンピュータシステムのオペレーティングシステムによって検出すること、トレースデータログにトレースデータを保存するための新たなメモリバッファを割り当てること、および、新たなメモリバッファをポイントするように予約レジスタを設定することを含むことができる。方法はまた、コンピュータプログラムの実行中に、コンピュータプログラムの実行に関連付けられた複数のオペレーティングシステムイベントを別個のトレースデータログに記録することを含むことができる。方法はまた、コンピュータプログラムの実行中に、オペレーティングシステムによって上書きされる1つ以上のメモリ値の前にターゲットコンピュータシステムの1つ以上のメモリ位置に対応する1つ以上のメモリ値を記録することを含むことができる。少なくとも1つのプロセッサは、複数の対称マルチコアプロセッサ（SMP）を含むことができ、複数のインストルメント化命令はさらに、複数のSMPの各々のトレースデータに関連付けられている複数のトレースデータログを生成するように構成されることができる。方法はまた、複数のSMPによるコンピュータプログラムの実行に関連するトレースデータを複数のトレースデータログに記録することを含むことができる。方法はまた、複数のタイムスタンプを複数のトレースデータログに定期的書き込むことを含むことができ、複数のタイムスタンプは、複数のSMPから収集されたトレースデータの同期化を容易にするように

10

20

30

40

50

構成される。トレースデータログは、コンピュータシステムのメモリに保存することができる。

【0318】

実施形態では、非一時的コンピュータ記憶媒体は、コンピュータプログラムをデバッグするための命令を含み、命令は、コンピュータシステムによって実行されると、少なくとも1つのトレースデータログに保存されるように構成されたトレースデータを生成するように構成された複数のインストルメント化命令を有するコンピュータプログラムをインストルメント化することを含む方法を実行する。インストルメント化は、コンピュータプログラムに関連付けられた複数の基本ブロックを決定することを含むことができ、複数の基本ブロック中の各基本ブロックは、1つのエントリポイントおよび1つのイグジットポイントを有する命令のシーケンスを含み、インストルメント化はさらに、複数の基本ブロックのうちの少なくともいくつかの基本ブロックについて、コンピュータプログラムに、少なくともいくつかの基本ブロックの複数の基本ブロックエントリポイントを記録するように構成された1つ以上のインストルメント化命令を生成して挿入することを含む。方法は、コンピュータプログラムの実行中に、ターゲットコンピュータシステム上でコンピュータプログラムの実行に関連するトレースデータを少なくとも1つのトレースデータログに記録させることを含むことができ、ターゲットコンピュータシステムはメモリおよび少なくとも1つのプロセッサを含む。

10

【0319】

実施形態では、前の段落のいずれかのコンピュータ記憶媒体は、以下の特徴の1つ以上を含むことができる。インストルメント化はさらに、上書きされる前にメモリに保存された1つ以上のメモリ値を記録するように構成されたインストルメント化命令を生成することを含むことができる。インストルメント化はまた、上書きされる少なくともいくつかのメモリ位置のメモリアドレスの記録をスキップすることを含むことができる。インストルメント化はまた、別の基本ブロックと一緒に実行されるように構成された基本ブロックをスキップすることを含むことができる。基本ブロックエントリポイントの少なくとも一部またはターゲットコンピュータシステムの1つ以上のメモリ位置の少なくともいくつかを、仮想メモリ位置とすることができる。複数のインストルメント化命令はまた、ターゲットコンピュータシステムの少なくとも1つのプロセッサの複数のレジスタに保存された値を定期的に記録するように構成されることができる。複数のインストルメント化命令はまた、キャッシュ使用量を改善するように構成されることができる。複数のインストルメント化命令はまた、キャッシュへの影響を軽減することができる。複数のインストルメント化命令の少なくとも一部のインストルメント化命令を、ターゲットコンピュータシステムの少なくとも1つのプロセッサ上で実行するための実行可能ファイルに1つ以上のオブジェクトファイルをリンクする時に無効にすることができる。コンピュータプログラムのインストルメント化はさらに、コンピュータプログラムの1つ以上のコンパイラ指令に遭遇することに応答して、コンピュータプログラムのコンパイル時に少なくとも1つの追加のインストルメント化命令を挿入することを含むことができる。1つ以上のコンパイラ指令は、コンピュータプログラムで同期動作をマークするように構成されることができる。複数のインストルメント化命令はさらに、第2の基本ブロックではなく第1の基本ブロックからのイグジット時に実行されるように構成されることができ、第2の基本ブロックは後続として第1の基本ブロックのみを有する。複数のインストルメント化命令はさらに、第3の基本ブロックではなく第1および第2の基本ブロックからのイグジット時に実行されるように構成されることができ、第3の基本ブロックは後続として第1および第2の基本ブロックのみを有する。

20

30

40

【0320】

実施形態では、前の段落のいずれかのコンピュータ記憶媒体は、以下の特徴の1つ以上を含むことができる。複数のインストルメント化命令はさらに、コンピュータプログラムの複数の関数について複数の関数エントリおよびイグジットに関連付けられたトレースデータを記録するように構成されることができる。コンピュータプログラムのインストルメ

50

ント化は、コンピュータプログラムの複数の関数の少なくとも1つのリーフ関数をスキップすることを含むことができる。複数のインストルメント化命令はさらに、トレースデータに複数のタイムスタンプを記録するように構成されることができ、複数のタイムスタンプは、コンピュータプログラムの複数の関数のうちの関数の関数エントリの時間および関数イグジットの時間を記録するように構成される。少なくとも1つのトレースデータログは、複数のインストルメント化命令に関連するトレースデータを記憶するための第1のトレースデータログと、オペレーティングシステムに関連するトレースデータを記憶するための第2のトレースデータログとを含むことができる。

【0321】

実施形態では、非一時的コンピュータ記憶媒体は、オペレーティングシステムによって実行される複数のタスクからの少なくとも1つのタスクをデバッグするための命令を含み、命令は、ターゲットコンピュータシステムによって実行されたときに、ターゲットコンピュータシステム上のタスクの実行中に収集されたトレースデータを用いて特定の時間におけるターゲットコンピュータシステムの状態を再構成することを含む方法を実行し、ターゲットコンピュータシステムはメモリおよび少なくとも1つのプロセッサを含み、方法はさらに、ターゲットコンピュータシステムの状態を表示することにより、少なくとも1つのタスクでエラーの識別を可能にすることを含む。トレースデータは、少なくとも1つのタスクの複数の基本ブロックの複数の基本ブロックエントリポイントに関連するターゲットコンピュータシステムの複数のメモリ位置、上書きされる1つ以上のメモリ値の前にターゲットコンピュータシステムの1つ以上のメモリ位置に対応する1つ以上のメモリ値、および、ターゲットコンピュータシステムの少なくとも1つのプロセッサの複数のレジスタに保存されたデータ値の複数のスナップショットであって、少なくとも1つのタスクの実行中に定期的に記録される複数のスナップショットを含むことができる。特定の時間にターゲットコンピュータシステムの状態を再構成することはさらに、少なくとも1つのタスクの実行を停止する前に実行される最後の基本ブロックで開始すること、少なくとも1つのタスクの実行の逆の順序でトレースデータをデコードすることであって、ターゲットコンピュータシステムのメモリおよび複数のレジスタに保存されたメモリ値を決定することを含むデコードすること、メモリに格納されたデコードデータ値および複数のレジスタに保存された決定データ値を用いて、特定の時間より前の時間から特定の時間までのターゲットコンピュータシステム上での少なくとも1つのタスクの実行をシミュレートすることを含むことができる。

【0322】

実施形態では、前の段落のいずれかのコンピュータ記憶媒体は、以下の特徴の1つ以上を含むことができる。トレースデータは、少なくとも1つのタスクの実行が停止されたときにターゲットコンピュータシステムのメモリに記憶されたメモリ値を含むことができ、ターゲットコンピュータシステムの状態を再構成することはさらに、少なくとも1つのタスクの実行が停止されたときにトレースデータに記憶されたメモリ値を使用することを含む。ターゲットコンピュータシステムの状態を再構成することはさらに、複数のメモリ位置の第1のメモリ位置について、第1のメモリ位置アドレスおよび上書きされる前の値をトレースデータから検索すること、および、上書きされる前のメモリ値を第1のメモリアドレスに書き込むことを含むことができる。ターゲットコンピュータシステムの状態を再構成することはさらに、トレースデータに第1のメモリ位置の現在のメモリ値を記憶することを含むことができ、少なくとも1つのタスクの実行をシミュレートすることはさらに、第1のメモリ位置が書き込まれていると判定することに応答して、トレースデータに保存されている現在のメモリ値を検索すること、および、第1のメモリ位置に現在のメモリ値を保存することを含むことができる。ターゲットコンピュータシステムの状態を再構成することはさらに、複数のメモリ位置の第2のメモリ位置について、マッピングファイルに保存された第1のメモリ位置のアドレスおよび情報を用いて第2のメモリ位置のアドレスを決定することを含むことができる。複数の基本ブロックにおける各基本ブロックは、1つのエントリポイントおよび1つのイグジットポイントを有する命令のシーケンスを含

むことができる。特定の時間に先行する時間におけるターゲットシステムの少なくとも1つのプロセッサの複数のレジスタに保存されたデータ値を決定することは、時間的に最も近く特定の時間に先行するスナップショットを使用することを含むことができる。ターゲットシステムの少なくとも1つのプロセッサの複数のレジスタに保存されたデータ値を決定することは、別のタスクへのコンテキストスイッチ時に保存されたレジスタ値を使用することを含むことができる。少なくとも1つのタスクは、オペレーティングシステムによって実行される複数のタスクを含むことができる。

【0323】

《用語》

一般的に本明細書中で使用される用語「モジュール」は、ハードウェアまたはファームウェアで具現されるロジックまたはソフトウェア命令の集合を指す。モジュールは、J A V A、CまたはC++などのプログラム言語で書かれる。ソフトウェアモジュールは実行可能プログラムにコンパイルまたはリンクされ、ダイナミックリンクライブラリにインストールされ得、または、B A S I C、P e r l、L U AもしくはP y t h o nなどのインタープリタ言語で書かれ得る。ソフトウェアモジュールは、他のモジュールから、または、自身から呼び出され得、および／または、検出されたイベントまたは中断に応じて開始され得る。ハードウェアで実現されたモジュールは、ゲートおよびフリップフロップなどの接続された論理ユニットを含み、および／または、プログラム可能ゲートアレイまたはプロセッサなどのプログラム可能ユニットを含み得る。

【0324】

一般に、本明細書に記載されるモジュールとは、他のモジュールと組み合わせられ得、または、それらの物理的編成またはストレージにかかわらずサブモジュールに分割され得る論理モジュールを指す。モジュールは、1つ以上のコンピューティングシステムによって実行され、任意の適切なコンピュータ可読媒体上またはその内部に保存され得、全体または一部において特別設計ハードウェアまたはファームウェア内に実装され得る。上記の方法、計算、プロセスまたは解析のいずれかがコンピュータの使用を介して容易にされ得るが、すべての計算、解析および／または最適化が、コンピュータシステムを使用する必要があるわけではない。さらに、いくつかの実施形態において、本明細書で説明されるプロセスブロックは、改変され、再配置され、組み合わせられ、および／または省略され得る。

(任意の添付の特許請求の範囲、要約書および図面を含む) 本明細書に開示された特徴のすべて、および／または、そのように開示された任意の方法またはプロセスのステップのすべては、任意の組み合わせで組み合わせられ得る。保護は、(任意の添付の特許請求の範囲、要約書および図面を含む) 本明細書に開示された特徴の任意の新規な1つ、または任意の新規な組み合わせに、または、そのよう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規な1つ、または任意の新規な組み合わせに及ぶ。

【0325】

本開示において、用語「リアルタイム」は、実質的に瞬間的またはほぼ瞬間的なイベントを指すことができる。換言すれば、「リアルタイム」とはユーザ相互作用時間の概念を指し得、コンテキスト仮想ワークスペースによるまたはそれを伴う処理時間量が、ユーザの反応時間未満(例えば1秒または0.5秒未満)であることなどである。本開示の他の位置では、用語「リアルタイム」は「リアルタイムクロック」というフレーズの一部として使用される。フレーズ「リアルタイムクロック」の一部として使用される場合、「リアルタイム」とは時間の客観的通過を指し、「リアルタイムクロック」とはミリ秒などの共通の単位で時間の客観的経過を測定する機構である。

【0326】

とりわけ「できる」、「可能性がある」、「かもしれない」または「し得る」などの条件付き言語は、具体的に別様に明記されない限り、または使用される文脈内で別様に理解されない限り、その特定の実施形態は、他の実施形態が含まなくとも、特定の特徴、要素および／またはステップを含むことを伝えることを一般的に意図している。したがって、そのような条件付き言語は、特徴、要素および／またはステップが1つ以上の実施形態の

10

20

30

40

50

ために任意の方法で必要であること、または、1つ以上の実施形態が必ず、ユーザの入力またはプロンプトの有無にかかわらず、これらの特徴、要素および／またはステップが任意の特定の実施形態で含まれる、または実行されるかどうかを決定するためのロジックを含むことを意味することを一般的には意図していない。「備える」、「含む」、「有する」、「含有する」などの用語は同義語であり、包括的に、オープンエンド様式で使用され、追加の要素、特徴、作用、動作などを除外しない。また、用語「または」は、包括的な意味で使用されており（その排他的な意味ではない）、よって、使用される場合、例えば要素のリストを接続するために、用語「または」は、リスト内の1つ、いくつか、またはすべての要素を意味している。さらに、本明細書で使用する用語「それぞれ」は、その通常の意味を有することに加えて、用語「それぞれ」が適用される要素の集合の任意のサブセットを意味することができる。

10

【0327】

「X、Y、およびZのうちの少なくとも1つ」というフレーズなどの論理積言語は、具体的に別様の記載がない限り、別様に文脈で理解されない限り、項目、ターム等がX、YまたはZのいずれであってもよいことを伝えるために一般的に使用される。したがって、そのような論理積言語は、特定の実施形態が、Xの少なくとも1つ、Yの少なくとも1つおよびZの少なくとも1つが存在することを必要とすることを意味することを一般に意図していない。本明細書で使用される用語「ほぼ」、「約」、「概して」および「実質的に」などの本明細書で使用される言語の程度は、値、量または特性が、依然として所望の機能を実行する、または所望の結果を実現する、記載された値、量または特性に近いことを表す。例えば、用語「ほぼ」、「約」、「概して」および「実質的に」は、記載された量の50%未満以内、10%未満以内、5%未満以内、1%未満以内、0.1%未満以内、0.01%未満以内を指し得る。本明細書で使用される見出しは読者の便宜のためだけのものであり、本開示または特許請求の範囲を限定することを意味するものではない。本開示は特定の好ましい実施形態および実施例に関連して説明されてきたが、本開示が具体的に開示された実施形態を超えて他の代替実施形態および／または本開示の使用ならびに明白な修正およびその均等物に及ぶことは当業者によって理解されるであろう。さらに当業者は、上記の方法のいずれもが任意の適切な装置を使用して実施できることを認識するであろう。さらに、実施形態に関連した任意の特定の特徴、態様、方法、性質、特性、品質、属性、要素などの本明細書における開示は、本明細書に記載の他のすべての実施形態において使用することができる。本明細書に記載のすべての実施形態について、方法のステップは順次、実行される必要はない。したがって、本開示の範囲は、上記の特定の開示された実施形態によって限定されるべきではないことが意図されている。

20

30

【0328】

本明細書に記載される方法およびタスクのすべては、コンピュータシステムによって実行され、完全に自動化され得る。コンピュータシステムは、いくつかの場合には、説明した機能を実行するために、ネットワークを介して通信および相互運用する（例えば、物理サーバ、ワークステーション、ストレージレイ等の）複数の異なるコンピュータまたはコンピューティングデバイスを含み得る。そのような各コンピューティングデバイスは典型的には、メモリまたは他の非一時的コンピュータ可読記憶媒体に保存されたプログラム命令またはモジュールを実行するプロセッサ（または複数のプロセッサ）を含む。本明細書に開示される様々な機能は、そのようなプログラム命令で実施され得るが、開示された機能の一部またはすべては、コンピュータシステムの特定期間向け回路（例えばASICまたはFPGA）で代替的に実装され得る。コンピュータシステムが複数のコンピューティングデバイスを含む場合、これらのデバイスは、同じ位置に配置され得るが、その必要ではない。開示された方法およびタスクの結果は、固体メモリチップおよび／または磁気ディスクなどの物理記憶デバイスを異なる状態に変換することによって永続的に保存され得る。

40

【0329】

本明細書に開示した範囲はまた、任意のおよびすべてのオーバーラップ、部分範囲、お

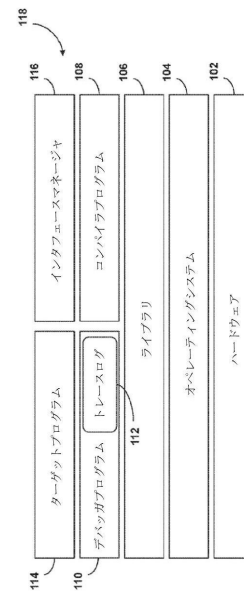
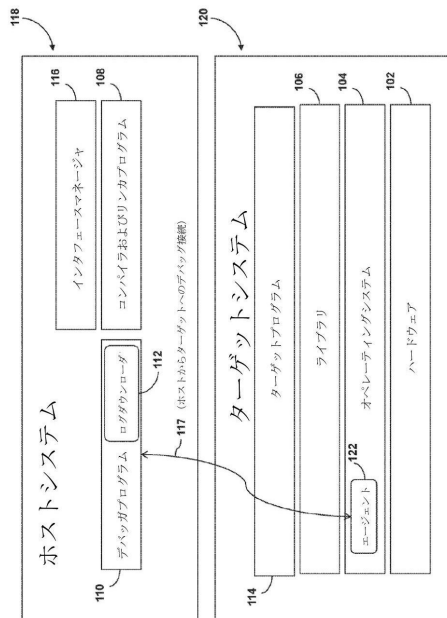
50

よびそれらの組み合わせを包含する。「まで」、「少なくとも」、「よりも大きい」、「未満」、「間」などの言語は、記載された数を含む。本明細書で使用される「ほぼ」、「約」および「実質的に」などの用語が先行する数値は、記載された数を含み、また、依然として所望の機能を実行する、または所望の結果を達成する記載された量に近い量を表す。例えば、用語「ほぼ」、「約」および「実質的に」は、記載された量の10%未満以内、9%、8%、7%、6%、5%、4%、3%、2%または1%未満以内、0.1%未満以内、および0.01%未満以内である量を指し得る。

【図面】

【図1A】

【図1B】



10

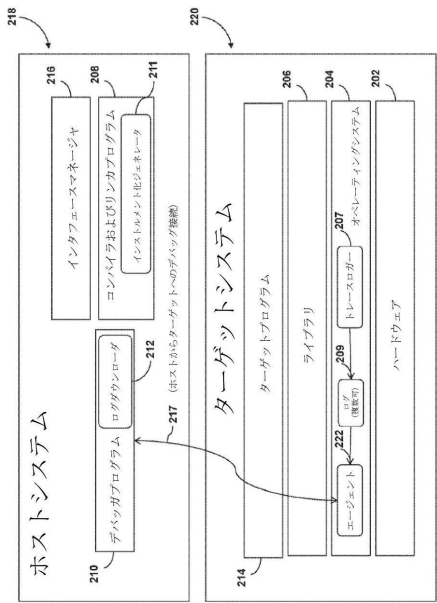
20

30

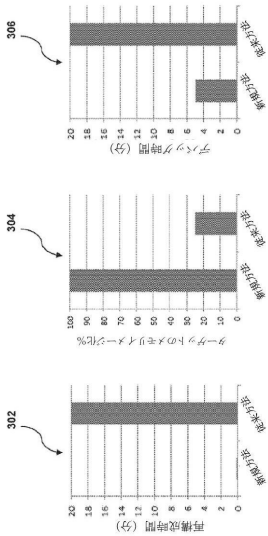
40

50

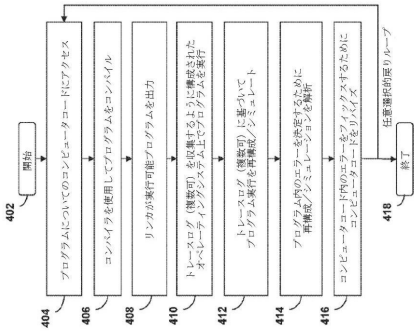
【図 2】



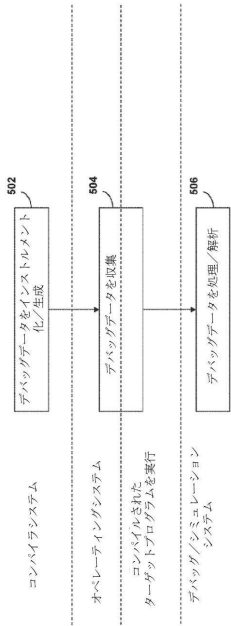
【図 3】



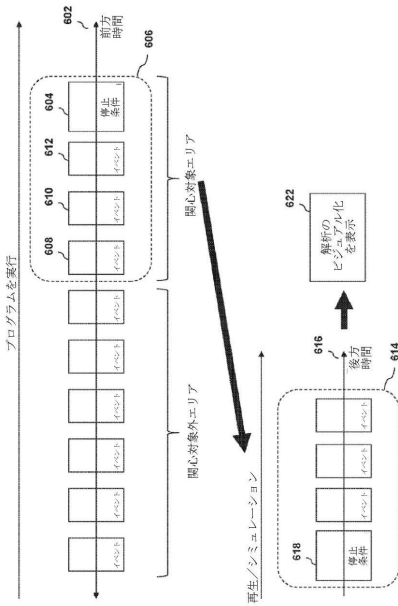
【図 4】



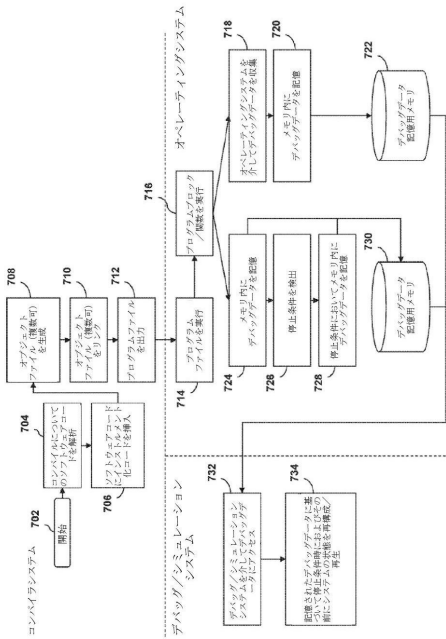
【図 5】



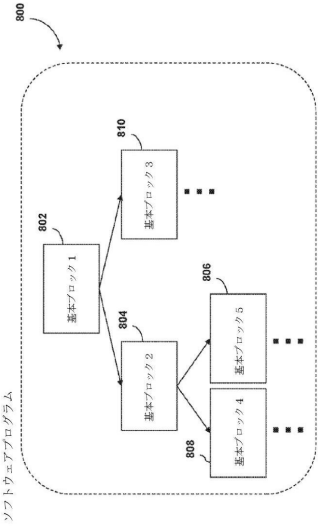
【図 6】



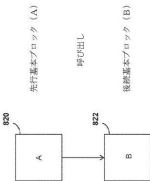
【図 7】



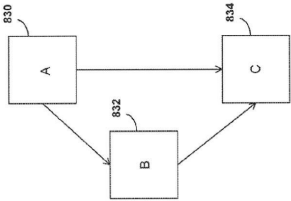
【図 8 A】



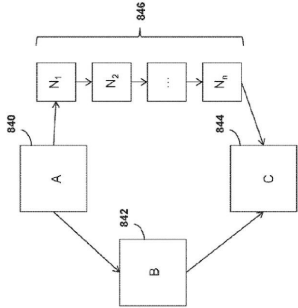
【図 8 B】



【図 8 C】



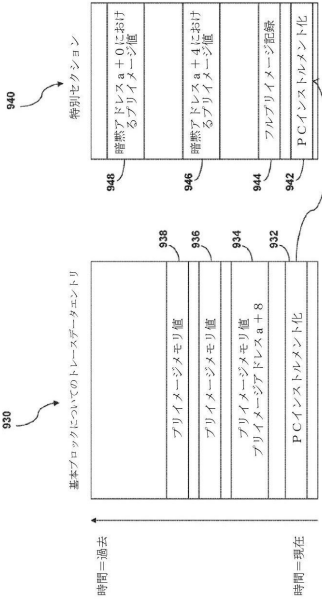
【図 8 D】



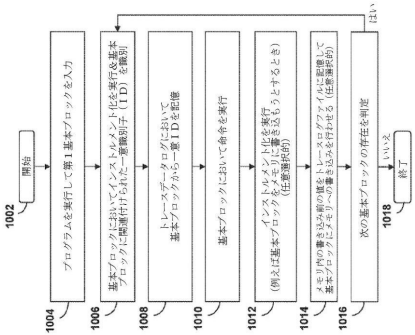
10

20

【図 9】



【図 10 A】

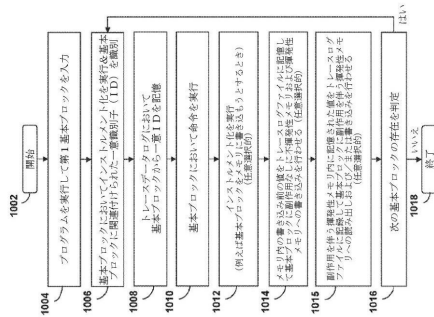


30

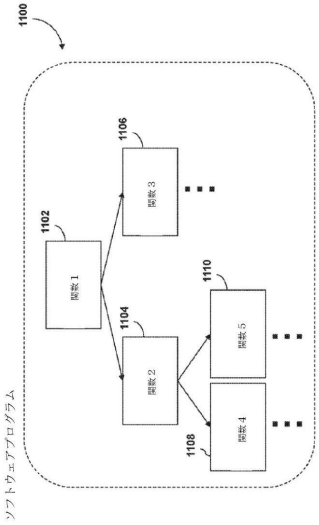
40

50

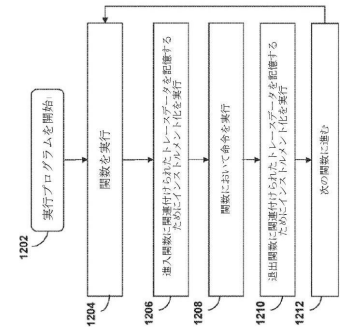
【図 10 B】



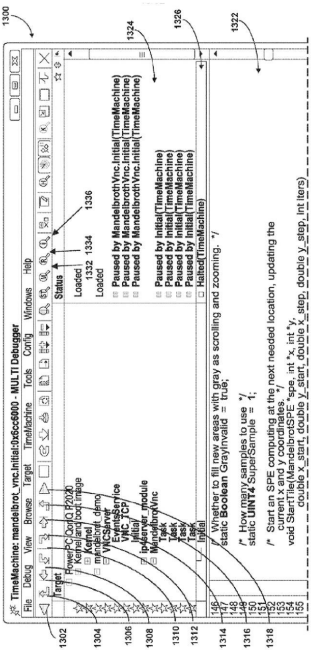
【図 11】



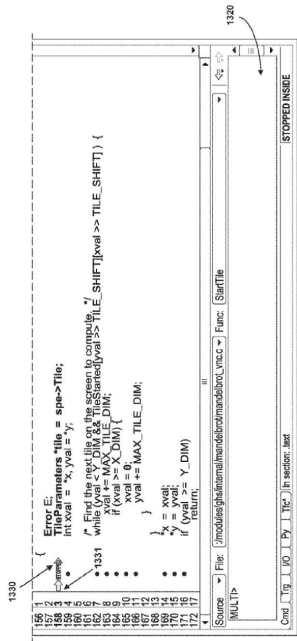
【図 12】



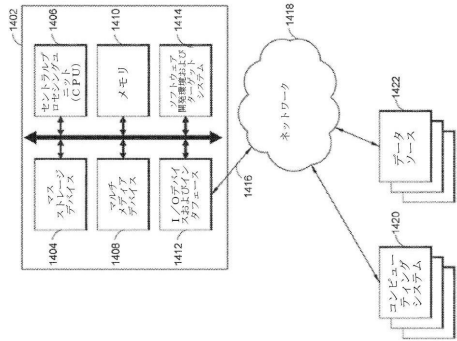
【図 13 A】



【図 1 3 B】



【図 1 4】

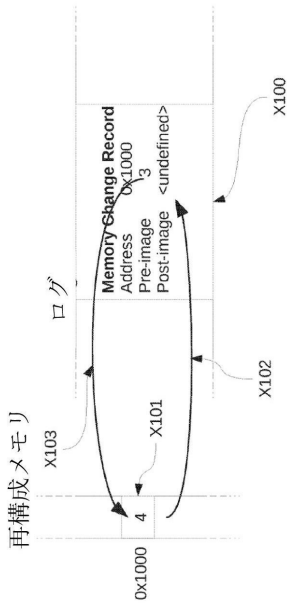


10

20

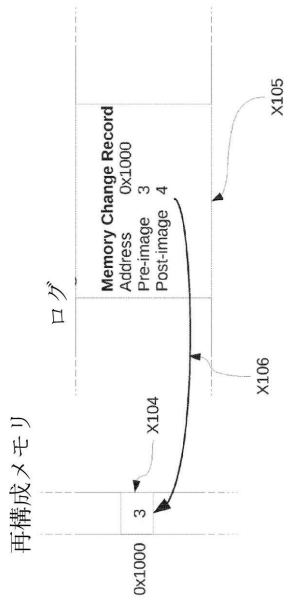
【図 1 5 A】

一回目の時間的後方進行



【図 1 5 B】

時間的前方進行



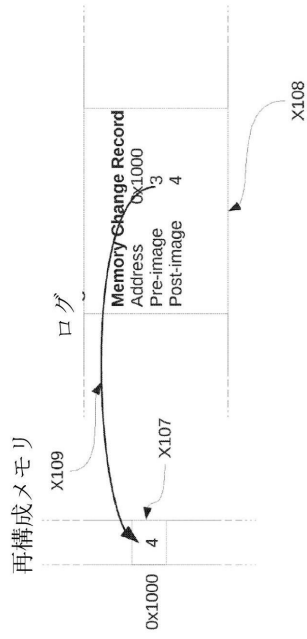
30

40

50

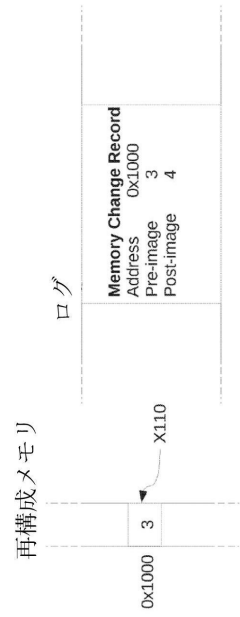
【図 1 5 C】

一回目の後の時間的後方進行



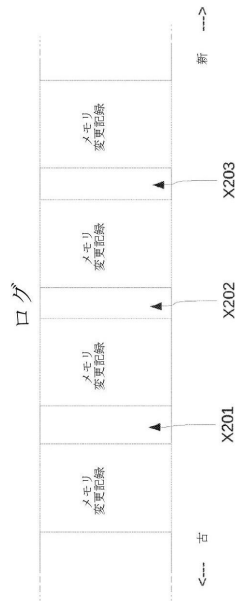
【図 1 5 D】

時間的後方進行後の最終状態



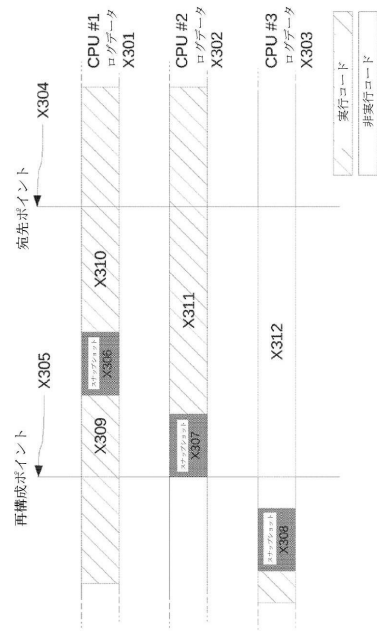
【図 1 6】

メモリ再構成



【図 1 7】

レジスタ再構成例



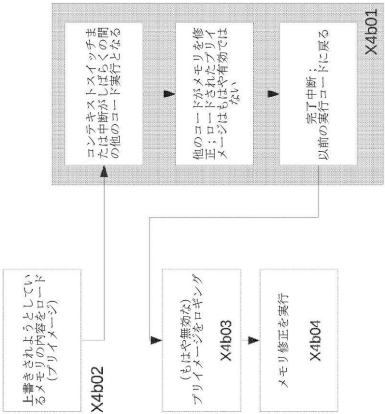
【図 1 8 A】

プライメージロギング



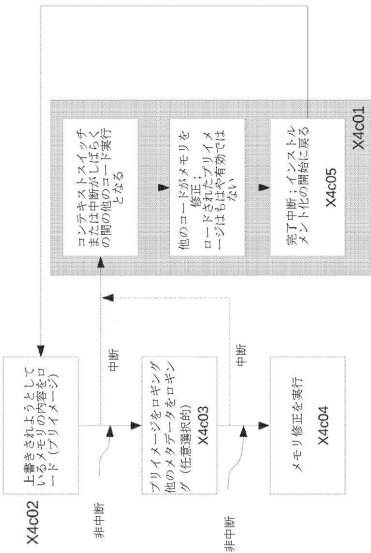
【図 1 8 B】

中断プライメージロギング



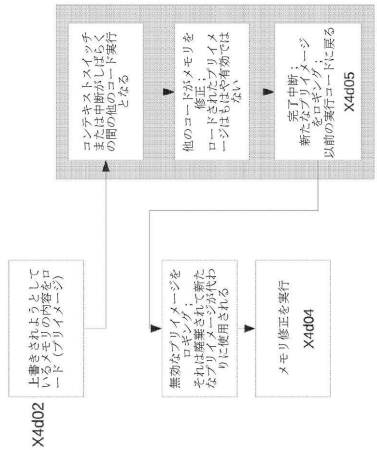
【図 1 8 C】

再始動可能ウィンドウを伴う
中断プライメージロギング

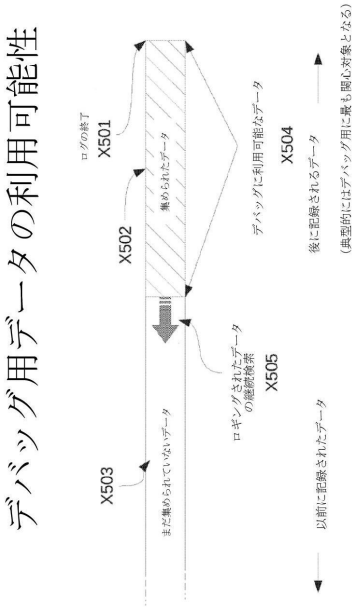


【図 1 8 D】

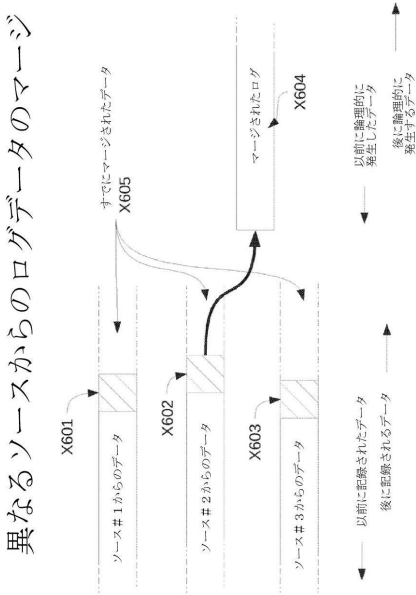
訂正データを伴う中断プライメージロギング



【図 1 9】

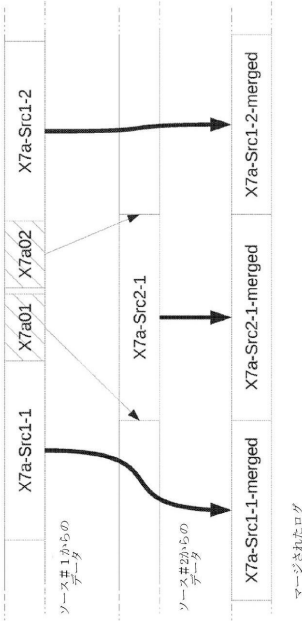


【図 2 0】



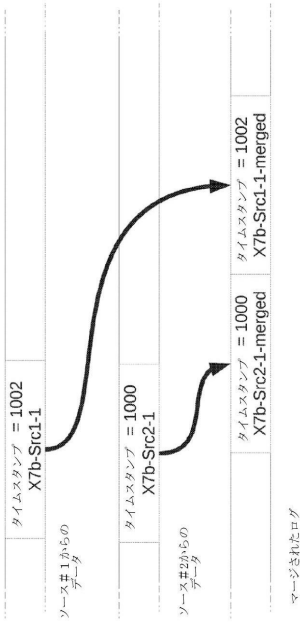
【図 2 1 A】

メタ情報順序付け技術

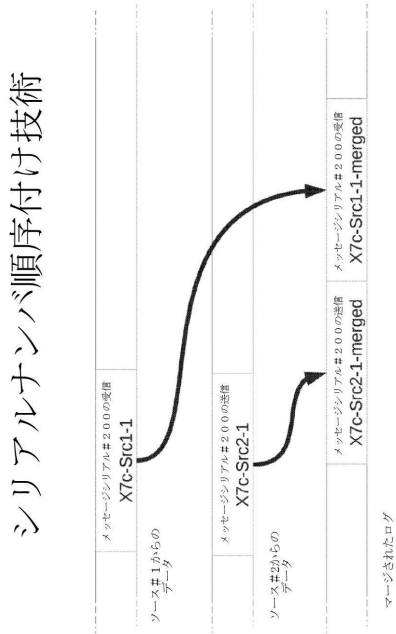


【図 2 1 B】

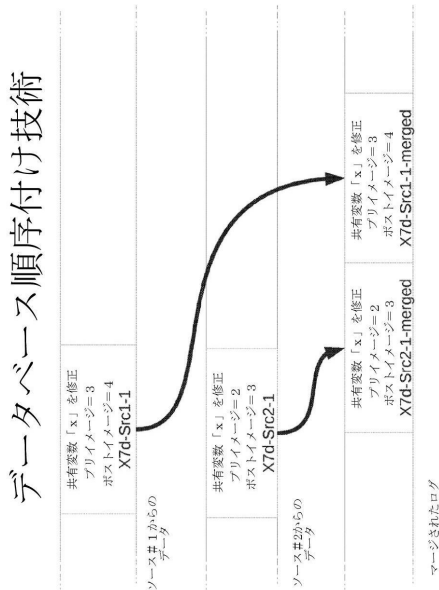
タイムスタンプ順序付け技術



【図 2 1 C】

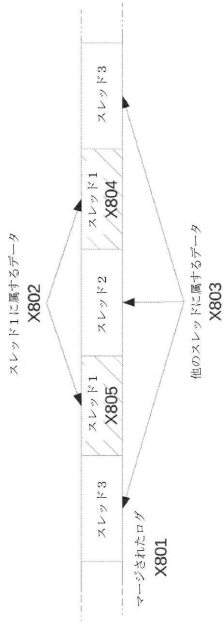


【図 2 1 D】



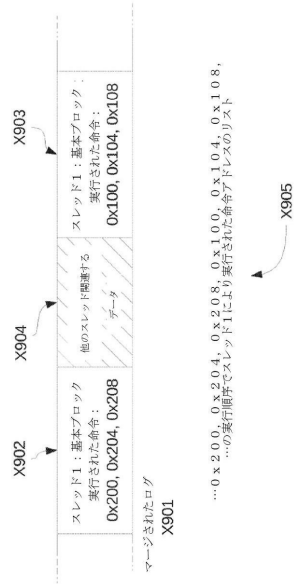
【図 2 2】

マージされたログ内のシングルスレッド



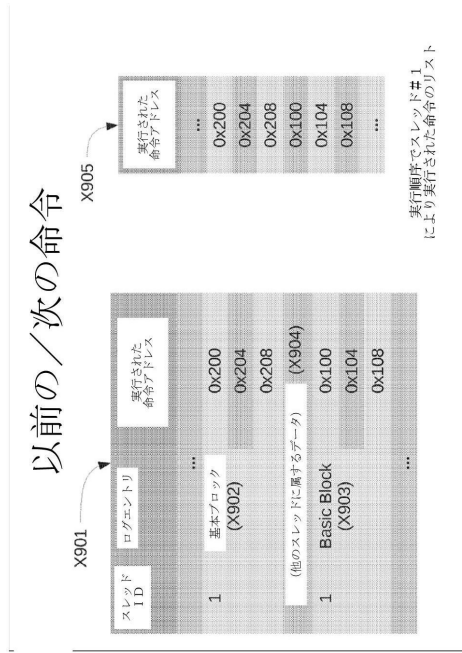
【図 2 3 A】

以前の／次の命令

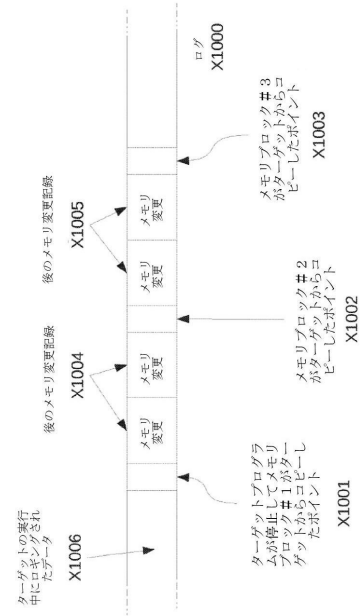


【圖 23 B】

【図 24】



後のメモリ変更記録



【図 25 A】

【図 25 B】

タイムスタンブ補間
(均一細分化)

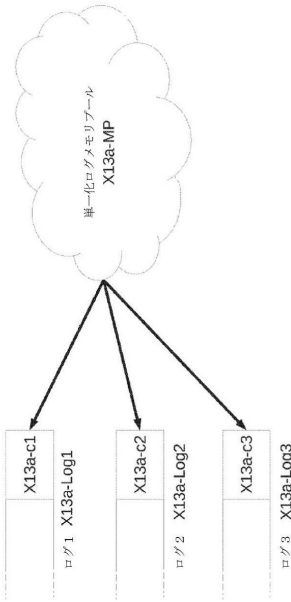
実験の タイムスタンプ	サブタイムスタンプ ビット付き	合意タイムスタンプ	イベント増設期間 (台成)
X11a.1	0x100	0x100000	0x800
X11a.2	-	-	0x800
X11a.3	-	-	0x800
X11a.4	-	-	0x800
X11a.5	-	-	0x800
X11a.6	-	-	0x800
X11a.7	0x103	0x103000	-

タイムスランプ補間
(不均一細分化)

実験の タイムスタンプ	サブタイムスタンプ ビット付き	合成タイムスタンプ	イベント持続期間 (台数)
X11b.1	0x100	0x100000	0x200
X11b.2	-	0x100200	0x200
X11b.3	-	0x100400	0x1400
X11b.4	-	0x101800	0x1400
X11b.5	-	0x102600	0x200
X11b.6	-	0x102e00	0x200
X11b.7	0x103	0x103000	

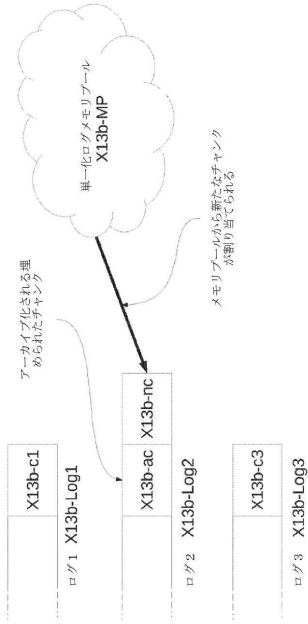
【図 2 6 A】

初期チャンク割り当て



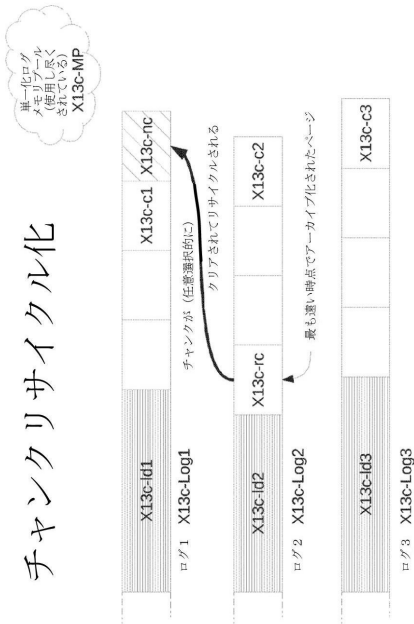
【図 2 6 B】

チャンクアーカイブ化



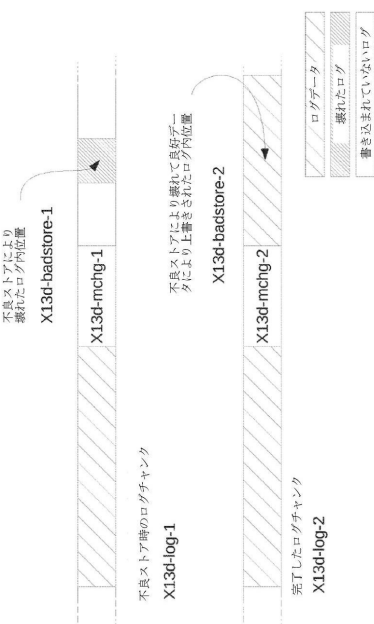
【図 2 6 C】

チャンクリサイクル化

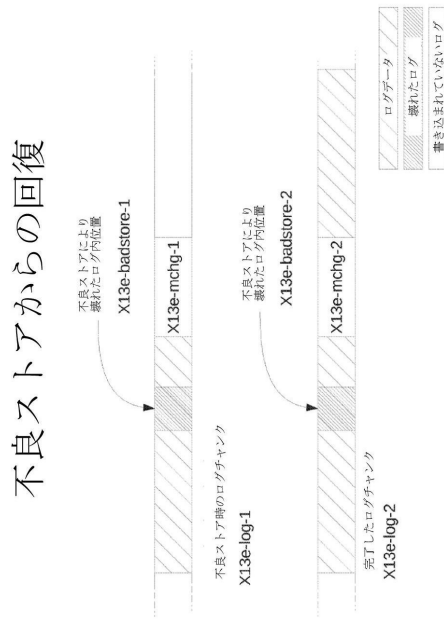


【図 2 6 D】

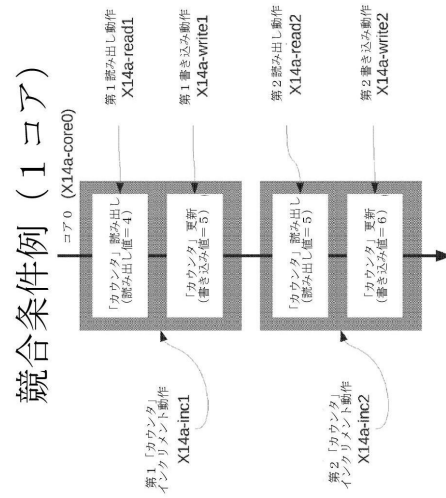
不良ストアからの回復



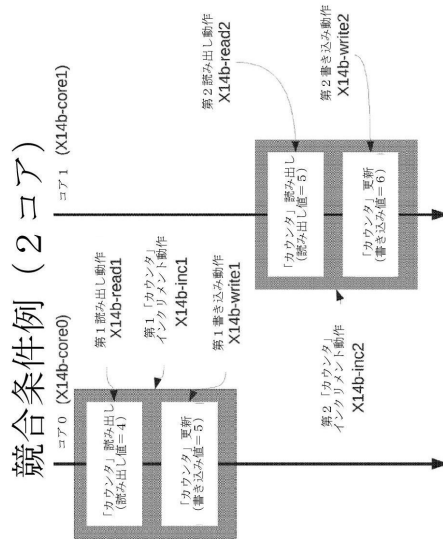
【図 26 E】



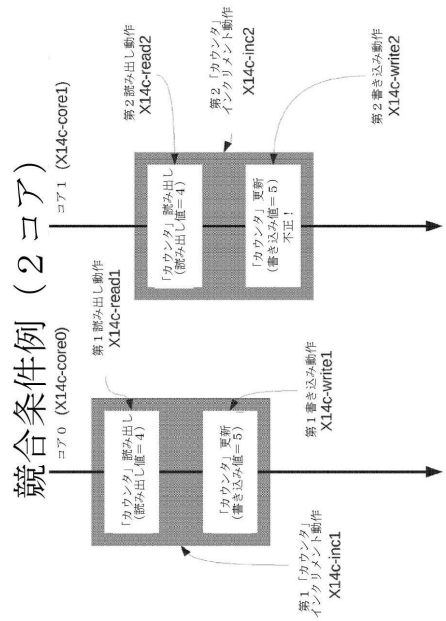
【図 27 A】



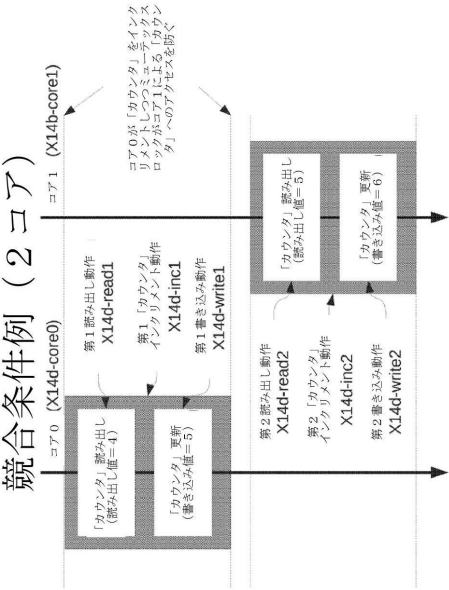
【図 27 B】



【圖 27 C】

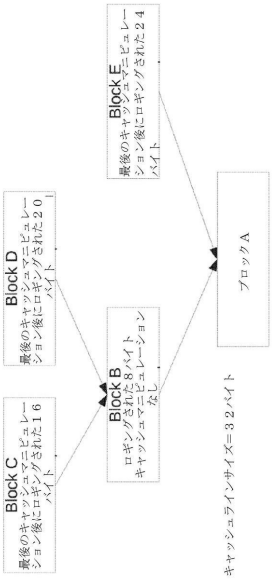


【図 27D】



【図 28】

キャッシュマニピュレーション命令の
挿入についてのコードフロー解析



10

20

30

40

50

フロントページの続き

- (72)発明者 オッドウッド, ダニエル, ディー.
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 ギンズバーグ, スティーブン, エイチ.
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 ベレルジェブ, ニコーラ
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 ディビス, グレゴリー
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 エディントン, グレグ
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 フィールド, ナザン
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 グリーン, マロリー, エム.
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 ケリー, フィリップ
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 ウォルフ, マイケル, ビー.
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30
- (72)発明者 ザビスカ, トム
 アメリカ合衆国、93101 カリフォルニア州 サンタ バーバラ、ウェスト ソーラー ストリート 30

審査官 関口 明紀

- (56)参考文献 特開平09-006647 (JP, A)
 特開2012-256279 (JP, A)
 米国特許第05784552 (US, A)
 特開平04-068446 (JP, A)
 米国特許出願公開第2005/0086246 (US, A1)
 米国特許出願公開第2015/0052403 (US, A1)
- (58)調査した分野 (Int.Cl., DB名)
 G06F 11/36